코틀린 언어 문서

경고 - 이 문서는 비공식 번역 문서로
오역이나 잘못된 정보를 포함할 수 있으며
자바스크립트, 멀티플랫폼, 문법은 포함하지 않습니다.

잘못된 정보 보낼 곳: 최범균(madvirus@madvirus.net)

Table of Contents

개요

서버 사이드를 위한 코틀린

코틀린을 사용하면 간결하고 표현력있는 코드를 작성할 수 있다. 또한 기존의 자바 기반 기술 스택과 호환성을 완벽하게 유지하며 학습이 어렵지 않기에 서버 어플리케이션 개발에 매우 적합하다:

- **표현력** : 코틀린은 타입에 안전한 빌더 나 위임 프로퍼티 와 같은 혁신적인 언어 특징을 제공하며, 이를 통해 강력하고 사용하기 쉬운 추상화를 만들 수 있다.
- **확장성** : 코틀린이 지원하는 코루틴 은 적정 수준의 하드웨어 수준에서 대량의 클라이언트를 처리할 수 있는 확장성있는 서버 어플리케이션을 구축할 수 있게 해준다.
- **상호운용성** : 코틀린은 모든 자바 기반 프레임워크와 완전히 호환되므로 익숙한 기술 스택을 유지하면서 최신 언어의 장점을 누릴 수 있다.
- **마이그레이션** : 코틀린은 코드베이스를 자바에서 코틀린으로 단계적이면서 점진적으로 이전할 수 있도록 지원한다. 자바로 작성한 시스템을 유지하면서 신규 코드를 코틀린으로 작성할 수 있다.
- **도구** : IDE를 위한 코틀린 지원이 대체로 훌륭하며, 인텔리J IDEA Ultimate는 플러그인으로 프레임워크에 특화된 도구(예, 스프링을 위한 도구)를 제공한다.
- **학습 곡선** : 자바 개발자는 쉽게 코틀린을 시작할 수 있다. 코틀린 플러그인에 포함된 자바-코틀린 자동 변환기로 시작할 때 도움을 받을 수 있다. 코틀린 콘 은 대화식으로 구성된 연습을 통해 언어의 핵심 특징을 익힐 수 있는 가이드를 제공한다.

코틀린을 지원하는 서버 개발 프레임워크

- 스프링 은 5.0 버전부터 더 간결한 API 를 제공하기 위해 코틀린 언어 특징을 사용한다. 온라인 프로젝트 생성기 를 사용하면 신규 코틀린 프로젝트를 빠르게 생성할 수 있다.
- Vert.x 는 JVM에서 동작하는 리액티브 웹 어플리케이션을 만들기 위한 프레임워크로 완전한 문서 를 포함한 코틀린 전용 모듈을 제공한다.
- Ktor 은 JetBrains가 만든 코틀린-네이티브 웹 프레임워크이다. 이 프레임워크는 높은 확장성과 사용하기 쉽고 관용적인 API를 제공하기 위해 코루틴을 사용한다.
- kotlinx.html 은 웹 어플리케이션에서 HTML을 만들 때 사용할 수 있는 DSL이다. JSP나 FreeMarker와 같은 전통적인 템플릿 시스템에 대한 대안으로 사용한다.
- JDBC를 통한 직접 접근 외에 JPA, 자바 드라이버를 통한 NoSQL 데이터베이스 사용 등의 영속성 기술을 사용할 수 있다. JPA의 경우 kotlin-jpa 컴파일러 플러그인 을 사용하면 코틀린으로 컴파일한 클래스를 프레임워크의 요구에 맞출 수 있다.

코틀린 서버 사이드 어플리케이션 배포

아마존 웹 서비스, 구글 클라우드 플랫폼 등 자바 웹 어플리케이션을 지원하는 모든 호스트에 코틀린 어플리케이션을 배포할 수 있다.

Heroku 에 코틀린 애플리케이션을 배포하려면 공식 Heroku 튜토리얼 을 따라하면 된다.

AWS Labs는 코틀린을 사용해서 AWS Lambda 함수를 작성하는 예제 프로젝트 를 제공한다.

서버 사이드 코틀린 사용자

Corda 는 주요 은행이 지원하는 오픈소스 분산 원장(ledger) 플랫폼으로, 전체를 코틀린으로 만들었다.

JetBrains Account 는 JetBrains의 전체 라이선스 판매와 검증을 담당하는 시스템으로 100% 코틀린으로 작성했고 2015년부터 큰 문제 없이 운영에서 돌고 있다.

다음 단계

- HTTP 서블릿으로 웹 어플리케이션 만들기 와 스프링 부트로 RESTful 웹 서비스 만들기 튜토리얼을 보면 코틀린으로 매우 작은 웹 어플리케이션을 만드는 방법을 배울 수 있다
- 언어에 대한 더 자세한 소개는 이 사이트의 레퍼런스 문서 와 코틀린 콘 에서 확인할 수 있다.

안드로이드를 위한 코틀린

코틀린은 안드로이드 어플리케이션 개발에 안성맞춤이다. 현대 언어의 장점을 새로운 제약없이 안드로이드 플랫폼에 도입할 수 있다.

- **호환성** : 코틀린은 JDK 6과 완전히 호환되므로, 오래된 버전의 안드로이드 장비에서 문제없이 코틀린 어플리케이션을 실행할 수 있다. 안드로이드 스튜디오는 코틀린 도구를 완벽히 지원하며 안드로이드 빌드 시스템과 호환된다.
- **성능** : 코틀린 어플리케이션은 동일한 자바 어플리케이션만큼 빠른데, 이는 매우 유사한 바이트코드 구조 덕분이다. 코틀린의 인라인 함수 지원으로 종종 람다를 사용한 코드가 자바로 작성한 동일 코드보다 더 빠를 때도 있다.
- **상호운용성** : 코틀린은 자바와 100% 상호운용할 수 있으며 존재하는 모든 안드로이드 라이브러리를 코틀린 어플리케이션에서 사용할 수 있다. 여기에는 애노테이션 처리도 포함되므로 Databinding과 Dagger 역시 동작한다.
- **풋프린트** : 코틀린 라이브러리는 매우 작고, ProGuard를 사용해서 더 줄일 수 있다. 실제 어플리케이션 에서 코틀린 런타임은 수백 개의 메서드만 추가하는데 이는 .apk 파일 크기를 100K 미만으로 증가시킨다.
- **컴파일 시간** : 코틀린은 효율적인 증분 컴파일을 지원하며, 클린 빌드에 약간의 추가 오버헤드만 발생한다. 증분 빌드는 보통 자바만큼 빠르거나 더 빠르다 .
- **학습 곡선** : 자바 개발자는 쉽게 코틀린을 시작할 수 있다. 코틀린 플러그인에 포함된 자바-코틀린 자동 변환기로 시작할 때 도움을 받을 수 있다. 코틀린 콘 은 대화식으로 구성된 연습을 통해 언어의 핵심 특징을 익힐 수 있는 가이드를 제공한다.

안드로이드 케이스 스터디를 위한 코틀린

주요 회사에서 성공적으로 코틀린을 도입했다. 다음은 그 회사 중 일부의 경험을 공유한 것이다:

- Pinterest는 매달 1억 5천만명 사용자가 사용하는 어플리케이션에 성공적으로 코틀린을 적용했다 .
- Basecamp는 안드로이드 앱을 100% 코틀린 코드 로 만들었다. 프로그래머의 만족도에서 큰 차이를 보였고 결과물의 품질과 속도에서 높은 향상이 있다고 보고했다.
- Keepsafe의 App Lock 앱 또한 100% 코틀린으로 전환했으며 소스 코드 줄 수를 30% 줄였고 메서드 개수를 10% 줄였다.

안드로이드 개발을 위한 도구

코틀린 팀은 표준 언어 특징 이상을 지원하는 안드로이드 개발 도구를 제공한다.

- 코틀린 안드로이드 확장 은 컴파일러 확장 도구로서 코드에서 `findViewById()` 호출을 제거하고 이를 컴파일러가 생성한 합성 프로퍼티로 대체할 수 있게 한다.
- Anko 는 안드로이드 API의 래퍼(wrapper)로 코틀린 친화적인 API를 제공한다. 또한 레이아웃 .xml 파일을 코틀린 코드로 대체할 수 있는 DSL도 제공한다.

다음 단계

- 안드로이드 스튜디오 3.0 프리뷰 를 다운받아 설치한다. 이 버전은 코틀린을 기본으로 지원한다.
- 안드로이드와 코틀린 시작하기 튜토리얼을 보고 첫 번째 코틀린 애플리케이션을 만들어본다.
- 언어에 대한 더 자세한 소개는 이 사이트의 레퍼런스 문서 와 코틀린 콘 에서 확인할 수 있다.
- Kotlin for Android Developers 은 또 다른 좋은 자료이다. 이 책은 코틀린을 이용한 실제 안드로이드 어플리케이션 제작 과정을 단계적으로 안내한다.
- 구글의 코틀린으로 작성한 예제 프로젝트 를 참고한다.

코틀린 자바스크립트 개요

코틀린은 자바스크립트를 대상(target)으로 할 수 있는 기능을 제공한다. 코틀린을 자바스크립트로 트랜스파일링해서 이를 제공한다. 현재 구현은 ECMAScript 5.1에 맞추
져 있는데 향후 ECMAScript 2015도 지원할 계획이다.

자바스크립트를 대상으로 선택하면 프로젝트 코드와 코틀린 표준 라이브러리의 모든 코드를 자바스크립트로 트랜스파일한다. 하지만 여기에는 사용한 JDK와 JVM 또는 지
바 프레임워크나 라이브러리는 포함하지 않는다. 코틀린이 아닌 모든 파일은 컴파일 과정에서 생략한다.

코틀린 컴파일러는 다음 목표를 지키기 위해 노력한다.

— 크기를 최적화한 결과를 제공한다.
— 가독성있는 자바스크립트 결과를 제공한다.
— 기존의 모듈 시스템과의 상호운용성을 제공한다.
— 자바스크립트나 JVM에 상관없이 표준 라이브러리에 (최대한 가능한 수준에서) 동일 기능을 제공한다.

어떻게 사용할 수 있나

다음 시나리오에서 코틀린을 자바스크립트로 컴파일할 수 있다:

— 클라이언트 사이드 자바스크립트를 대상으로 코틀린 코드 생성
  — **DOM 요소 다루기** . 코틀린은 DOM을 다룰 수 있는 정적 타입 인터페이스를 제공하며, 이를 이용해 DOM 요소를 생성하고 수정할 수 있다.
  — **WebGL과 같은 그래픽 다루기** . 코틀린으로 WebGL을 이용한 웹 페이지 그래픽 요소를 생성할 수 있다.
— 서버 사이드 자바스크립트를 대상으로 코틀린 코드 생성
  — **서버 사이드 기술로 실행하기** . node.js와 같은 서버 사이드 자바스크립트를 다루는데 코틀린을 사용할 수 있다.

JQuery나 ReactJS와 같은 기존의 서드파티 라이브러리나 프레임워크와 함께 코틀린을 사용할 수 있다. 강한 타입형 API에서 서드파티 프레임워크에 접근하려면,
Definitely Typed 타입 정의 리포지토리에서 구한 타입스크립트 정의를 ts2kt 도구를 사용해서 코틀린으로 전환할 수 있다. 또는 강한 타입없이 동적 타입 을 사용해서 프레
임워크에 접근할 수 있다.

코틀린은 또한 CommonJS, AMD and UMD 모듈 시스템과 호환되며, 다른 모듈 시스템과 상호운용 할 수 있다

자바스크립트에 대해 코틀린으로 시작하기

자바스크립트를 위한 코틀린을 시작하는 방법은 튜토리얼 에서 확인할 수 있다.

코틀린 1.1의 새로운 점

내용

자바스크립트

코틀린 1.1부터 자바스크립트 대상(target)은 더 이상 실험단계가 아니다. 모든 언어 특징을 지원하고 프론트엔드 개발 환경과 통합을 위한 많은 새 도구를 제공한다. 자세한 변경 목록은 아래 에서 확인할 수 있다.

코루틴 (실험)

코틀린 1.1의 핵심 특징은 *코루틴*이다. 코루틴은 `async` / `await` , `yield` 내지 이와 유사한 프로그램 패턴을 지원한다 코틀린 설계의 핵심 특징은 코루틴 실행 구현이 언어가 아닌 라이브러리의 일부라는 점이다. 이는 코루틴이 특정 프로그래밍 패러다임이나 병행 라이브러리에 묶이지 않도록 한다.

코루틴은 효과적인 경량 쓰레드로 나중에 중지나 재시작할 수 있다. *서스펜딩 함수* 를 통해 코루틴을 지원한다. 이 함수를 호출하면 코루틴을 잠재적으로 지연할 수 있고, 익명 지연 함수(예, 지연 람다)를 사용하면 새로운 코루틴을 시작할 수 있다.

외부 라이브러리인 kotlinx.coroutines 로 구현한 `async` / `await` 를 보자:

```
// 백그라운드 쓰레드 풀에서 코드 실행
fun asyncOverlay() = async(CommonPool) {
    // 두 개의 비동기 오퍼레이션 시작
    val original = asyncLoadImage("original")
    val overlay = asyncLoadImage("overlay")
    // 그리고 두 결과에 Overlay를 적용
    applyOverlay(original.await(), overlay.await())
}

// UI 컨텍스트에서 새로운 코루틴을 시작
launch(UI) {
    // 비동기 Overlay가 끝날때까지 대기
    val image = asyncOverlay().await()
    // 그리고 UI에 표시
    showImage(image)
}
```

여기서 `async { ... }` 는 코루틴을 시작하고, `await()` 를 사용하면 대기하는 오퍼레이션을 실행하는 동안 코루틴 실행을 지연하고, (아마도 다른 쓰레드에서) 대기중인 오퍼레이션이 끝날 때 실행을 재개한다.

표준 라이브러리는 `yield` 와 `yieldAll` 함수를 사용해서 *시퀀스 지연 생성* 을 지원하기 위해 코루틴을 사용한다. 그 시퀀스에서, 시퀀스 요소를 리턴하는 코드 블럭은 각 요소를 읽어온 이후에 멈추고 다음 요소를 요청할 때 재개한다. 다음은 예이다:

```
  val seq = buildSequence {
      for (i in 1..5) {
          // i의 제곱을 생성
          yield(i * i)
      }
      // 범위를 생성
      yieldAll(26..28)
  }

  // 시퀀스를 출력
  println(seq.toList())
```

위 코드를 실행하고 결과를 보자. 마음껏 수정해서 실행해보자!

코루틴 문서 와 튜토리얼 에서 더 많은 정보를 참고할 수 있다.

코루틴은 현재 *실험적인 특징* 으로, 코틀린 팀은 최종 1.1 버전 이후에 코루틴의 하위호환성을 지원하지 않을 수도 있다.

다른 언어 특징

타입 별칭

타입 별칭을 사용하면 기존 타입을 다른 이름으로 정의할 수 있다. 이는 함수 타입이나 콜렉션과 같은 지네릭 타입에 매우 유용하다. 다음은 예이다.

```
typealias OscarWinners = Map<String, String>

fun countLaLaLand(oscarWinners: OscarWinners) =
        oscarWinners.count { it.value.contains("La La Land") }

// 타입 이름(초기와 타입 별칭)을 상호사용할 수 있다:
fun checkLaLaLandIsTheBestMovie(oscarWinners: Map<String, String>) =
        oscarWinners["Best picture"] == "La La Land"
```

다 자세한 내용은 문서 와 KEEP 을 참고한다.

객체에 묶인 호출가능 레퍼런스

이제 `::` 연산자를 이용해서 특정 객체 인스턴스의 메서드나 프로퍼티를 가리키는 멤버 레퍼런스 를 구할 수 있다. 이전 버전에서는 람다에서만 표현할 수 있었다. 다음은 예이다:

```
val numberRegex = "\\d+".toRegex()
val numbers = listOf("abc", "123", "456").filter(numberRegex::matches)
```

다 자세한 내용은 문서 와 KEEP 을 참고한다.

실드(sealed)와 데이터 클래스

코틀린 1.1은 1.0 버전에 있는 실드(sealed)와 데이터(data) 클래스의 일부 제약을 없앴다. 이제 최상위 실드 클래스의 하위클래스를 실드 클래스의 중첩 클래스뿐만 아니라 같은 파일에 정의할 수 있다. 데이터 클래스가 다른 클래스를 상속할 수 있다. 이를 사용하면 식(expression) 클래스의 계층을 멋지고 깔끔하게 정의할 수 있다:

```
sealed class Expr

data class Const(val number: Double) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()
object NotANumber : Expr()

fun eval(expr: Expr): Double = when (expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
}
val e = eval(Sum(Const(1.0), Const(2.0)))
```

자세한 내용은 문서 , 실드 클래스 KEEP 또는 데이터 클래스 KEEP을 참고한다.

람다에서의 분리 선언

이제 람다에 인자를 전달할 때 분리 선언 구문을 사용할 수 있다. 다음은 예이다:

```
    val map = mapOf(1 to "one", 2 to "two")
    // 이전
    println(map.mapValues { entry ->
        val (key, value) = entry
        "$key -> $value!"
    })
    // 이제
    println(map.mapValues { (key, value) -> "$key -> $value!" })
```

더 자세한 내용은 문서 와 KEEP 을 참고한다.

사용하지 않는 파라미터를 위한 밑줄

여러 파라미터를 갖는 람다에 대해, 사용하지 않는 파라미터 자리에 이름 대신에 `_` 글자를 사용할 수 있다:

```
    map.forEach { _, value -> println("$value!") }
```

이는 또한 분리 선언 에서도 동작한다:

```
    val (_, status) = getResult()
```

더 자세한 정보는 KEEP 을 참고한다.

숫자 리터럴에서의 밑줄

자바 7처럼 코틀린도 숫자를 그룹으로 나누기 위해 숫자 리터럴에서 밑줄을 사용할 수 있다.

```
val oneMillion = 1_000_000
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

자세한 내용은 KEEP 을 참고한다.

프로퍼티를 위한 짧은 구문

프로퍼티의 getter를 식 몸체로 정의한 경우, 프로퍼티 타입을 생략할 수 있다:

```
data class Person(val name: String, val age: Int) {
    val isAdult get() = age >= 20 // 프로퍼티 타입을 'Boolean'으로 추론
}
```

인라인 프로퍼티 접근자

프러퍼티가 지원 필드를 갖지 않으면 프로퍼티 접근자에 `inline` 수식어를 붙일 수 있다. 이 접근자는 인라인 함수 와 동일한 방법으로 컴파일된다.

```
public val <T> List<T>.lastIndex: Int
    inline get() = this.size - 1
```

또한 전체 프로퍼티에 `inline` 을 붙일 수 있으며, 이 경우 두 접근자에 수식어를 적용한다.

자세한 내용은 문서 와 KEEP 을 참고한다.

로컬 위임 프로퍼티

이제 로컬 변수에 위임 프로퍼티 구문을 사용할 수 있다. 한 가지 가능한 용도는 지연 평가되는 로컬 변수를 정의하는 것이다:

```
    val answer by lazy {
        println("Calculating the answer...")
        42
    }
    if (needAnswer()) {                     // 임의 값을 리턴
        println("The answer is $answer.")    // 이 시점에 answer를 계산
    }
    else {
        println("Sometimes no answer is the answer...")
    }
```

자세한 내용은 KEEP 을 참고한다.

위임 프로퍼티 바인딩 인터셉션

위임 프로퍼티 에 대해 `provideDelegate` 연산자를 사용해서 프로퍼티 바인딩에 대한 위임을 가로챌 수 있다. 예를 들어 바인딩 전에 프로퍼티 이름을 검사하고 싶다면 다음과 같이 작성할 수 있다:

```
class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(thisRef: MyUI, prop: KProperty<*>): ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, prop.name)
        ... // 프로퍼티 생성
    }

    private fun checkProperty(thisRef: MyUI, name: String) { ... }
}

fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { ... }

class MyUI {
    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}
```

`MyUI` 인스턴스를 생성하는 동안 각 프로퍼티에 대해 `provideDelegate` 메서드를 호출하고, 바로 필요한 검증을 수행할 수 있다.

자세한 내용은 문서 를 참고한다.

지네릭 enum 값 접근

이제 지네릭 방식으로 enum 클래스의 값을 차례로 열거할 수 있다.

```
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    print(enumValues<T>().joinToString { it.name })
}
```

DSL에서 암묵적 리시버를 위한 범위 제어

`@DslMarker` 애노테이션은 DSL 컨텍스트의 바깥 범위에서 리시버 사용을 제한한다. 기준이 되는 HTML 빌더 예제 를 보자:

```
table {
    tr {
        td { +"Text" }
    }
}
```

코틀린 1.0에서 `td` 에 전달한 람다에 있는 코드는 `table` 에 전달된 것, `tr` 에 전달된 것, `td` 에 전달된 것 이렇게 세 개의 암묵적인 리시버에 접근한다. 이는 컨텍스트에서 의미없는 메서드를 호출할 수 있게 허용한다. 예를 들어 `td` 안에서 `tr` 을 호출해서 `<td>` 에 `<tr>` 태그를 넣는 것이 가능하다.

코틀린 1.1에서는 `td` 의 암묵적 리시버에 정의된 메서드는 오직 `td` 에 전달된 람다에서만 접근할 수 있게 제한할 수 있다. `@DslMarker` 메타 애노테이션을 지정한 애노테이션을 정의하고 이를 태그 클래스의 베이스 클래스에 적용하면 된다.

더 많은 정보는 문서 와 KEEP 을 참고한다.

`rem` 연산자

`mod` 연산자를 디프리케이트했고 대신 `rem` 을 사용한다. 이슈 에서 이렇게 바꾼 동기를 확인할 수 있다.

표준 라이브러리

String에서 숫자로의 변환

`String.toIntOrNull(): Int?` , `String.toDoubleOrNull(): Double?` 등 잘못된 숫자에 대해 익셉션 발생없이 문자열을 숫자로 변환할 수 있는 확장을 String에 추가했다:

```
val port = System.getenv("PORT")?.toIntOrNull() ?: 80
```

또한 `Int.toString()` , `String.toInt()` , `String.toIntOrNull()` 과 같은 정수 변환 함수도 `radix` 파라미터를 가진 오버로딩한 함수를 갖는다. 이 함수는 변환의 진수(2에서 36)를 지정할 수 있다.

onEach()

`onEach` 는 콜렉션과 시퀀스의 확장 함수로 작지만 유용하다., 이 함수는 오퍼레이션 체인에서 콜렉션/시퀀스의 각 요소에 대해 어떤 액션-아마도 부수효과를 가진-을 수행할 수 있게 한다. Iterable에 대해 이 확장함수는 `forEach` 처럼 동작하지만, 이어지는 Iterable 인스턴스를 리턴한다. 시퀀스의 경우 이 확장함수는 래핑한 시퀀스를 리턴하며, 요소를 반복할 때 주어진 액션을 지연시켜 적용한다.

```
inputDir.walk()
        .filter { it.isFile && it.name.endsWith(".txt") }
        .onEach { println("Moving $it to $outputDir") }
        .forEach { moveFile(it, File(outputDir, it.toRelativeString(inputDir))) }
```

also(), takeIf() 그리고 takeUnless()

모든 리시버에 적용할 수 있는 범용 목적의 확장 함수 세 개가 있다.

`also` 는 `apply` 와 유사하다. 리서버를 받고, 리시버에 어떤 행위를 하고, 리시버를 리턴한다. 차이는 `apply` 안의 블록에서는 리시버를 `this` 로 접근 가능한데 반해 `also` 의 블록에서는 `it` 으로 접근 가능하다는 것이다(원한다면 다른 이름을 줄 수도 있다). 이는 바깥 범위의 `this` 를 감추고 싶지 않을 때 쓸모 있다:

```
fun Block.copy() = Block().also {
    it.content = this.content
}
```

`takeIf` 는 단일 값에 대해서 `filter` 와 유사하다. 리시버가 조건을 충족하는지 검사하고 충족하면 리시버를 리턴하고 그렇지 않으면 `null` 을 리턴한다 엘비스-연산자, 이른 리턴과 조합하면 다음과 같은 구성이 가능하다:

```
val outDirFile = File(outputDir.path).takeIf { it.exists() } ?: return false
// 존재하는 outDirFile로 뭔가 하기
```

```
    val index = input.indexOf(keyword).takeIf { it >= 0 } ?: error("keyword not found")
    // 입력 문자열에서 키워드를 발견하면 해당 키워드의 인덱스를 갖고 무언가 함
```

`takeUnless` 는 `takeIf` 와 동일하다. 단 조건의 반대를 취한다. 조건을 충족하지 않을 때 리시버를 리턴하고 그렇지 않으면 `null` 을 리턴한다. 위 예를 `takeUnless` 로 재작성하면 다음과 같다:

```
val index = input.indexOf(keyword).takeUnless { it < 0 } ?: error("keyword not found")
```

또한 람다 대신에 호출가능 레퍼런스를 함께 사용하면 편리하다:

```
    val result = string.takeUnless(String::isEmpty)
```

groupingBy()

키를 이용해서 콜렉션을 그룹으로 나누고 동시에 그룹을 접을(fold) 때 이 API를 사용할 수 있다. 예를 들어, 다음과 같이 이 API를 이용해서 각 글자로 시작하는 단어의 수를 셀 수 있다:

```
    val frequencies = words.groupingBy { it.first() }.eachCount()
```

Map.toMap()과 Map.toMutableMap()

이 함수를 이용하면 맵을 쉽게 복사할 수 있다:

```
class ImmutablePropertyBag(map: Map<String, Any>) {
    private val mapCopy = map.toMap()
}
```

Map.minus(key)

`plus` 연산자는 읽기 전용 맵에 키값 쌍을 추가해서 새로운 맵을 생성하는 방법을 제공하는데, 반대 기능은 간단한 방법이 없었다. 맵에서 키를 삭제하려면 `Map.filter()` 나 `Map.filterKeys()` 와 같이 직접적이지 않은 방법을 사용해야 했다. 이제는 `minus` 연산자를 사용하면 된다. 한 개 키 삭제, 키 콜렉션 삭제, 키의 시퀀스로 삭제, 키 배열로 삭제하는 4개의 연산자를 사용할 수 있다.

```
val map = mapOf("key" to 42)
val emptyMap = map - "key"
```

minOf() and maxOf()

이 함수는 두 개 이상의 값(기본 숫자 타입이나 `Comparable` 객체)에서 최솟값과 최댓값을 찾는다. `Comparable` 이 아닌 객체를 비교할 수 있도록 `Comparator` 인스턴스를 받는 함수도 추가로 제공한다:

```
val list1 = listOf("a", "b")
val list2 = listOf("x", "y", "z")
val minSize = minOf(list1.size, list2.size)
val longestList = maxOf(list1, list2, compareBy { it.size })
```

배열과 같은 리스트 생성 함수

`Array` 생성자와 유사하게, `List` 와 `MutableList` 인스턴스를 생성하고 람다를 통해 각 요소를 초기화하는 함수를 제공한다:

```
val squares = List(10) { index -> index * index }
val mutable = MutableList(10) { 0 }
```

Map.getValue()

`Map` 에 대한 이 확장은 주어진 키가 존재하면 해당하는 값을 리턴하고 키가 없으면 익셉션을 발생한다. `withDefault` 를 사용해서 맵을 생성한 경우, 이 함수는 익셉션을 발생하는 대신에 기본 값을 리턴한다.

```
val map = mapOf("key" to 42)
// 널일수 없는 Int 값 42를 리턴
val value: Int = map.getValue("key")

val mapWithDefault = map.withDefault { k -> k.length }
// returns 4
val value2 = mapWithDefault.getValue("key2")

// map.getValue("anotherKey") // <- NoSuchElementException을 발생
```

추상 콜렉션

코틀린 콜렉션 클래스를 구현할 때 추상 클래스를 기반 클래스로 사용할 수 있다. 읽기 전용 콜렉션을 구현할 때에는 `AbstractCollection` , `AbstractList` , `AbstractSet` , `AbstractMap` 을 사용하며, 수정 가능 콜렉션은 `AbstractMutableCollection` , `AbstractMutableList` , `AbstractMutableSet` , `AbstractMutableMap` 을 사용한다. JVM에서 이들 수정 가능한 추상 콜렉션은 JDK의 추상 콜렉션의 대부분 기능을 상속한다.

배열 조작 함수

이제 표준 라이브러리로 배열의 요소간 연산을 위한 함수를 제공한다. 이 연산에는 비교(`contentEquals` 와 `contentDeepEquals`), 해시 코드 계산(`contentHashCode` 와 `contentDeepHashCode`), 문자열 변환(`contentToString` 와 `contentDeepToString`)이 있다. JVM(`java.util.Arrays` 에 대응하는 함수에 대한 별칭으로 동작)과 JS(코틀린 표준 라이브러리가 구현을 제공)에서 모두 지원한다.

```
val array = arrayOf("a", "b", "c")
println(array.toString())  // JVM 구현: 타입과 알 수 없는 해시 값 출력
println(array.contentToString())  // 리스트처럼 보기 좋게 출력
```

JVM 백엔드

자바 8 바이트코드 지원

이제 자바 8 바이트코드 생성 옵션을 제공한다(`-jvm-target 1.8` 명령행 옵션이나 대응하는 앤트/메이븐/그레이들 옵션). 현재는 이 옵션이 바이트코드의 세만틱을 변경하지 않지만(특히 인터페이스의 디폴트 메서드와 람다를 코틀린 1.0처럼 생성한다), 향후에 이를 더 사용할 계획이다.

자바 8 표준 라이브러리 지원

이제 자바 7과 8에 추가된 새로운 JDK API를 지원하는 표준 라이브러리 버전을 따로 제공한다. 새 API에 접근하려면 표준 `kotlin-stdlib` 대신에 `kotlin-stdlib-jre7` 과 `kotlin-stdlib-jre8` 메이븐 아티팩트를 사용하면 된다. 이 아티팩트는 `kotlin-stdlib` 를 일부 확장한 것으로 의존성 전이로 `kotlin-stdlib` 를 포함한다.

바이트코드의 파라미터 이름

이제 바이트코드의 파라미터 이름 저장을 지원한다. `-java-parameters` 명령행 옵션을 사용해서 활성화할 수 있다.

상수 인라인

컴파일러는 이제 `const val` 프로퍼티의 값을 상수 사용 위치에 인라인한다.

수정가능 클로저 변수

람다에서 수정가능한 클로저 변수를 캡처하기 위해 사용한 박싱 클래스는 더 이상 volatile 필드를 갖지 않는다. 이 변화는 성능을 향상시키지만, 매우 드물게 새로운 경쟁 조건(race condition)을 유발할 수 있다. 경쟁 조건에 영향을 받는다면 변수에 접근할 때 자신만의 동기화를 제공해야 한다.

javax.script 지원

코틀린은 이제 javax.script API (JSR-223)와의 통합을 지원한다. 이 API를 사용하면 런타임에 코드를 평가할 수 있다:

```
val engine = ScriptEngineManager().getEngineByExtension("kts")!!
engine.eval("val x = 3")
println(engine.eval("x + 2"))  // 5를 출력
```

이 API를 사용하는 더 큰 예제 프로젝트는 여기 를 참고한다.

kotlin.reflect.full

자바 9 지원을 준비하기 위해 `kotlin-reflect.jar` 라이브러리에 있는 확장 함수와 프로퍼티를 `kotlin.reflect.full` 로 옮겼다. 이전 패키지(`kotlin.reflect`)의 이름을 디프리케이트했고 코틀린 1.2에서는 삭제할 것이다. (`KClass` 와 같은) 핵심 리플렉션 인터페이스는 `kotlin-reflect` 가 아닌 코틀린 표준 라이브러리에 속하므로 이동에 영향을 받지 않는다.

자바스크립트 백엔드

표준 라이브러리 통합

자바스크립트로 컴파일한 코드에서 코틀린 표준 라이브러리의 더 많은 부분을 사용할 수 있다. 콜렉션(`ArrayList` , `HashMap` 등), 익셉션(`IllegalArgumentException`), 몇 가지 다른 것(`StringBuilder` , `Comparator`) 등의 핵심 클래스가 이제 `kotlin` 패키지 아래에 정의되어 있다. JVM에서 이들 이름은 대응하는 JDK 클래스에 대한 타입 별칭이며 JS의 경우 코틀린 표준 라이브러리에 클래스를 구현했다.

더 나아진 코드 생성

자바스크립트 백엔드는 이제 더욱 정적인 검사 가능한 코드를 생성하며, minifier나 최적화 linter 등의 JS 코드 처리 도구에 친화적이다.

`external` 수식어

타입에 안전한 방법으로 자바스크립트로 정의한 클래스를 코틀린에서 접근하고 싶다면, `external` 수식어를 사용해서 코틀린 선언을 작성할 수 있다. (코틀린 1.0에서는 `@native` 애노테이션을 사용했다.) JVM 대상과 달리 JS 대상은 클래스와 프로퍼티에 `external` 수식어를 사용하는 것을 허용한다. 예를 들어 다음은 DOM `Node` 클래스에 선언한 예를 보여준다:

```
external class Node {
    val firstChild: Node

    fun appendChild(child: Node): Node

    fun removeChild(child: Node): Node

    // etc
}
```

import 처리 개선

자바스크립트 모듈에 정확하게 임포트하기 위한 선언을 제공할 수 있다. external 선언에 `@JsModule("<module-name>")` 애노테이션을 추가하면, 컴파일 과정에서 모듈 시스템(CommonJS나 AMD)에 알맞게 임포트할 수 있다. 예를 들어 CommonJS를 사용하면 `require(...)` 함수로 선언을 임포트한다. 추가로 모듈이나 글로벌 자바스크립트 객체로 선언을 임포트하고 싶다면 `@JsNonModule` 애노테이션을 사용한다.

다음은 JQuery를 코틀린 모듈로 임포트한 예를 보여준다:

```
external interface JQuery {
    fun toggle(duration: Int = definedExternally): JQuery
    fun click(handler: (Event) -> Unit): JQuery
}

@JsModule("jquery")
@JsNonModule
@JsName("$")
external fun jquery(selector: String): JQuery
```

이 예에서 JQuery를 이름이 `jquery` 인 모듈로 임포트한다. 코틀린 컴파일러가 사용하기로 설정한 모듈 시스템에 따라 대안으로 $-객체를 사용해서 접근할 수 있다

어플리케이션에서 선언을 다음과 같이 사용할 수 있다:

```
fun main(args: Array<String>) {
    jquery(".toggle-button").click {
        jquery(".toggle-panel").toggle(300)
    }
}
```

시작하기

기초 구문

패키지 정의

패키지 정의는 소스 파일의 처음에 위치해야 한다:

```
package my.demo

import java.util.*

// ...
```

디렉토리와 패키지가 일치할 필요는 없으며, 파일 시스템 어디든 소스 파일을 위치시킬 수 있다.

패키지 를 보자.

함수 정의

두 개의 `Int` 파라미터와 리턴 타입이 `Int` 인 함수:

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

식 몸체(expression body)를 사용하고 리턴 타입을 추론:

```
fun sum(a: Int, b: Int) = a + b
```

의미없는 값을 리턴하는 함수:

```
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
```

`Unit` 리턴 타입은 생략 가능:

```
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}
```

함수 를 보자.

로컬 변수 정의

한 번만 할당 가능한(읽기 전용) 로컬 변수:

```
    val a: Int = 1  // 특시 할당
    val b = 2   // `Int` 타입으로 추론
    val c: Int  // 초기화를 하지 않으면 타입 필요
    c = 3       // 나중에 할당
```

변경가능 변수:

```
    var x = 5 // `Int` 타입 추론
    x += 1
```

또한 프로퍼티 필드 를 참고하자.

주석

자바나 자바스크립트와 마찬가지로 코틀린은 주석 블록과 줄 주석을 지원한다.

```
// 이는 줄 주석

/* 이는 여러 줄에 걸친
   주석 블록이다. */
```

자바와 달리 코틀린은 블록 주석을 중첩할 수 있다.

문서화 주석 구문은 코틀린 코드 문서화 문서를 참고한다.

문자열 템플릿 사용

```
    var a = 1
    // 템플릿에서 단순 이름 사용:
    val s1 = "a is $a"

    a = 2
    // 템플릿에서 임의의 식 사용:
    val s2 = "${s1.replace("is", "was")}, but now is $a"
```

문자열 템플릿 을 참고한다.

조건 식 사용

```
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}
```

`if` 를 식으로 사용하기:

```
fun maxOf(a: Int, b: Int) = if (a > b) a else b
```

`if` -식 을 참고한다.

null 가능 값 사용과 `null` 검사

`null` 값이 가능할 때 반드시 레퍼런스를 명시적으로 null 가능(nullable)으로 표시해야 한다.

아래 코드가 `str` 이 정수를 갖지 않으면 `null` 을 리턴한다고 할 때:

```
fun parseInt(str: String): Int? {
    // ...
}
```

null 가능 값을 리턴하는 함수는 다음과 같이 사용한다:

```
fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // `x * y` 코드는 에러를 발생하는데, 이유는 null을 가질 수 있기 때문이다.
    if (x != null && y != null) {
        // null 검사 이후에 x와 y를 자동으로 null 불가로 변환
        println(x * y)
    }
    else {
        println("either '$arg1' or '$arg2' is not a number")
    }
}
```

또는

```
    // ...
    if (x == null) {
        println("Wrong number format in arg1: '$arg1'")
        return
    }
    if (y == null) {
        println("Wrong number format in arg2: '$arg2'")
        return
    }

    // null 검사 이후에 x와 y를 자동으로 null 불가로 변환
    println(x * y)
```

Null-안전성 을 참고한다.

타입 검사와 자동 변환 사용

`is` 연산자는 식이 타입의 인스턴스인지 검사한다. 불변 로컬 변수나 프로퍼티가 특정 타입인지 검사할 경우 명시적으로 타입을 변환할 필요가 없다:

```
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // 이 블록에서는 `obj`를 자동으로 `String`으로 변환
        return obj.length
    }

    // 타입 검사 블록 밖에서 `obj`는 여전히 `Any` 타입
    return null
}
```

또는

```
fun getStringLength(obj: Any): Int? {
    if (obj !is String) return null

    // `obj`를 자동으로 `String`으로 변환
    return obj.length
}
```

또는 심지어 다음도 가능

```
fun getStringLength(obj: Any): Int? {
    // 우변의 `&&`에서 `obj`를 자동으로 `String`으로 변환
    if (obj is String && obj.length > 0) {
        return obj.length
    }

    return null
}
```

클래스 와 타입 변환 을 참고한다.

for 루프 사용

```
    val items = listOf("apple", "banana", "kiwi")
    for (item in items) {
        println(item)
    }
```

또는

```
    val items = listOf("apple", "banana", "kiwi")
    for (index in items.indices) {
        println("item at $index is ${items[index]}")
    }
```

for 루프 를 참고한다.

while 루프 사용

```
    val items = listOf("apple", "banana", "kiwi")
    var index = 0
    while (index < items.size) {
        println("item at $index is ${items[index]}")
        index++
    }
```

while 루프 를 참고한다.

when 식 사용

```
fun describe(obj: Any): String =
    when (obj) {
        1          -> "One"
        "Hello"    -> "Greeting"
        is Long    -> "Long"
        !is String -> "Not a string"
        else       -> "Unknown"
    }
```

when 식 을 참고한다.

범위 사용

`in` 연산자를 사용해서 숫자가 범위에 들어오는지 검사한다:

```
    val x = 10
    val y = 9
    if (x in 1..y+1) {
        println("fits in range")
    }
```

`!in` 연산자를 사용해서 숫자가 범위를 벗어나는지 검사한다:

```
    val list = listOf("a", "b", "c")

    if (-1 !in 0..list.lastIndex) {
        println("-1 is out of range")
    }
    if (list.size !in list.indices) {
        println("list size is out of valid list indices range too")
    }
```

범위를 반복:

```
for (x in 1..5) {
    print(x)
}
```

또는 단순한 범위 이상:

```
for (x in 1..10 step 2) {
    print(x)
}
for (x in 9 downTo 0 step 3) {
    print(x)
}
```

범위 를 참고한다.

콜렉션 사용

콜렉션에 대한 반복:

```
for (item in items) {
    println(item)
}
```

`in` 연산자로 콜렉션이 객체를 포함하는지 검사:

```
when {
    "orange" in items -> println("juicy")
    "apple" in items -> println("apple is fine too")
}
```

콜렉션을 걸러내고 변환하기 위해 람다 식 사용:

```
fruits
    .filter { it.startsWith("a") }
    .sortedBy { it }
    .map { it.toUpperCase() }
    .forEach { println(it) }
```

고차 함수와 람다 를 참고한다.

기본 클래스와 인스턴스 만들기:

```
val rectangle = Rectangle(5.0, 2.0) // 'new' 키워드 필요하지 않음
val triangle = Triangle(3.0, 4.0, 5.0)
```

클래스 , 오브젝트와 인스턴스 를 참고한다.

이디엄

코틀린에서 종종 사용되는 이디엄을 정리했다. 선호하는 이디엄이 있다면 풀리퀘스트를 날려 기여해보자.

DTO 생성 (POJO/POCO)

```
data class Customer(val name: String, val email: String)
```

다음 기능을 가진 `Customer` 클래스를 제공한다:

- 모든 프로퍼티에 대한 getter (그리고 `var` 의 경우 setter)
- `equals()`
- `hashCode()`
- `toString()`
- `copy()`
- 모든 프로퍼티에 대한 `component1()` , `component2()` , ..., (data 클래스 참고)

함수 파라미터의 기본 값

```
fun foo(a: Int = 0, b: String = "") { ... }
```

리스트 필터링

```
val positives = list.filter { x -> x > 0 }
```

더 짧게 표현:

```
val positives = list.filter { it > 0 }
```

스트링 삽입

```
println("Name $name")
```

인스턴스 검사

```
when (x) {
    is Foo -> ...
    is Bar -> ...
    else   -> ...
}
```

쌍으로 맵이나 목록 탐색

```
for ((k, v) in map) {
    println("$k -> $v")
}
```

`k` , `v` 대신 임의 이름을 사용할 수 있다.

범위 사용

```
for (i in 1..100) { ... }  // 닫힌 범위: 100 포함
for (i in 1 until 100) { ... } // 반만 열린 범위: 100 미포함
for (x in 2..10 step 2) { ... }
for (x in 10 downTo 1) { ... }
if (x in 1..10) { ... }
```

읽기 전용 리스트

```
val list = listOf("a", "b", "c")
```

읽기 전용 맵

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

맵 접근

```
println(map["key"])
map["key"] = value
```

지연(lazy) 프로퍼티

```
val p: String by lazy {
    // 문자열 계산
}
```

확장 함수

```
fun String.spaceToCamelCase() { ... }

"Convert this to camelcase".spaceToCamelCase()
```

싱글톤 생성

```
object Resource {
    val name = "Name"
}
```

If not null 축약

```
val files = File("Test").listFiles()

println(files?.size)
```

If not null and else 축약

```
val files = File("Test").listFiles()

println(files?.size ?: "empty")
```

null이면 문장 실행

```
val values = ...
val email = values["email"] ?: throw IllegalStateException("Email is missing!")
```

null이 아니면 실행

```
val value = ...

value?.let {
    ... // null이 아닌 블록 실행
}
```

null이 아니면 null 가능 값 매핑

```
val value = ...

val mapped = value?.let { transformValue(it) } ?: defaultValueIfValueIsNull
```

when 문장에서 리턴

```
fun transform(color: String): Int {
    return when (color) {
        "Red" -> 0
        "Green" -> 1
        "Blue" -> 2
        else -> throw IllegalArgumentException("Invalid color param value")
    }
}
```

'try/catch' 식

```
fun test() {
    val result = try {
        count()
    } catch (e: ArithmeticException) {
        throw IllegalStateException(e)
    }

    // 결과로 작업
}
```

'if' 식

```
fun foo(param: Int) {
    val result = if (param == 1) {
        "one"
    } else if (param == 2) {
        "two"
    } else {
        "three"
    }
}
```

`Unit` 을 리턴하는 메서드의 빌더-방식 용법

```
fun arrayOfMinusOnes(size: Int): IntArray {
    return IntArray(size).apply { fill(-1) }
}
```

단일 식 함수

```
fun theAnswer() = 42
```

이는 다음과 동일하다.

```
fun theAnswer(): Int {
    return 42
}
```

단일 식 함수는 효과적으로 다른 이디엄과 쓸 수 있으며 코드를 더 짧게 만들어준다. 예를 들어, 다음은 `when` 식과 함께 사용하는 코드이다:

```
fun transform(color: String): Int = when (color) {
    "Red" -> 0
    "Green" -> 1
    "Blue" -> 2
    else -> throw IllegalArgumentException("Invalid color param value")
}
```

객체 인스턴스의 메서드를 여러 번 호출('with')

```
class Turtle {
    fun penDown()
    fun penUp()
    fun turn(degrees: Double)
    fun forward(pixels: Double)
}

val myTurtle = Turtle()
with(myTurtle) { //100 pix 사각형 그리기
    penDown()
    for(i in 1..4) {
        forward(100.0)
        turn(90.0)
    }
    penUp()
}
```

자바 7의 자원을 사용한 try

```
val stream = Files.newInputStream(Paths.get("/some/file.txt"))
stream.buffered().reader().use { reader ->
    println(reader.readText())
}
```

지네릭 타입 정보를 요구하는 지네릭 함수를 위한 편리한 양식

```
//  public final class Gson {
//      ...
//      public <T> T fromJson(JsonElement json, Class<T> classOfT) throws JsonSyntaxException {
//      ...

inline fun <reified T: Any> Gson.fromJson(json: JsonElement): T = this.fromJson(json, T::class.java)
```

null 가능 Boolean 사용

```
val b: Boolean? = ...
if (b == true) {
    ...
} else {
    // `b`는 false나 null
}
```

코딩 관례

이 문서는 코틀린 언어를 위한 현재의 코딩 관례를 포함한다.

명명 규칙

다음의 자바 코딩 관례를 기본으로 한다:

- 이름은 낙타표기법을 사용(이름에 밑줄 사용하지 않음)
- 타입은 대문자로 시작
- 메서드와 프로퍼티는 소문자로 시작
- 4개의 공백문자 들여쓰기 사용
- public 함수는 코틀린 Doc에 보이도록 문서화해야 함

콜론

타입과 하위 타입을 구분하는 콜론에는 이전에 공백을 넣고 콜론이 인스턴스와 타입을 구분하면 공백을 넣지 않는다:

```
interface Foo<out T : Any> : Bar {
    fun foo(a: Int): T
}
```

람다

람다식 안에서 중괄호 주변에 공백을 사용하고 파라미터와 몸체를 구분하기 위해 화살표 앞뒤로 공백을 넣는다. 가능하면 람다를 괄호 밖에 전달한다.

```
list.filter { it > 10 }.map { element -> element * 2 }
```

짧고 중첩되지 않은 람다에서는 파라미터를 명시적으로 선언하는 대신 `it` 을 사용할 것을 권장한다. 파라미터를 가진 중첩된 람다에서는 파라미터를 항상 명시적으로 선언한다.

클래스 헤더 포맷팅

인자 개수가 적은 클래스는 한 줄로 작성할 수 있다:

```
class Person(id: Int, name: String)
```

긴 헤더를 가진 클래스는 주요 생성자의 각 인자를 별도 줄로 들여쓰기해서 포맷팅한다. 또한 닫는 괄호는 새 줄에 작성한다. 상속을 사용하면 상위 클래스의 생성자 호출이나 구현한 인터페이스 목록을 괄호와 같은 줄에 위치시킨다:

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name) {
    // ...
}
```

다중 인터페이스를 상속한 경우 상위 생성자 호출을 먼저 위치시키고 각 인터페이스를 다른 줄에 위치시킨다:

```
class Person(
    id: Int,
    name: String,
    surname: String
) : Human(id, name),
    KotlinMaker {
    // ...
}
```

생성자 파라미터는 보통의 들여쓰기를 사용하거나 연속 들여쓰기(보통 들여쓰기의 두 배)를 사용한다.

Unit

함수가 Unit을 리턴하면 리턴 타입을 생략한다:

```
fun foo() { // ": Unit"을 생략

}
```

함수 vs 프로퍼티

인자 없는 함수의 경우 읽기 전용 프로퍼티로 교체할 수 있다. 비록 의미는 비슷하지만 프로퍼티를 더 선호하는 경우가 있다.

다음 경우 함수보다 프로퍼티를 선호한다:

- 익셉션을 발생하지 않음
- `O(1)` 복잡도임
- 계산 비용이 작음(또는 첫 실행에 캐시함)
- 호출마다 같은 결과를 리턴함

기초

기본 타입

코틀린에서 모든 것은 객체로서 변수에 대한 멤버 함수나 프로퍼티를 호출할 수 있다. 어떤 타입은 특별한 내부 표현을 갖지만-예를 들어 숫자, 문자, 불리언 같은 타입은 런타임에 기본 값으로 표현된다- 사용자에게는 일반 클래스처럼 보인다. 이 절에서는 코틀린의 기본 타입인 숫자, 문자, 불리언, 배열, 문자열에 대해 설명한다.

숫자

코틀린은 자바와 유사한 방법으로 숫자를 다루는데 완전히 같지는 않다. 예를 들어 숫자에 대해 넓은 타입으로의 자동 변환이 없고, 어떤 경우에는 리터럴도 약간 다르다.

코틀린이 제공하는 숫자 내장 타입은 다음과 같다(이는 자바와 유사하다):

타입	비트 크기
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

코틀린에서 문자는 숫자가 아니다.

리터럴 상수

정수 값을 위한 리터럴 상수 종류는 다음과 같다:

- 십진수: `123`
  - Long은 대문자 `L` 로 표시: `123L`
- 16진수: `0x0F`
- 2진수: `0b00001011`

주의: 8진수 리터럴은 지원하지 않음

또한 부동소수점을 위한 표기법을 지원한다:

- 기본은 Double: `123.5` , `123.5e10`
- Float은 `f` 나 `F` 로 표시: `123.5f`

숫자 리터럴에 밑줄(1.1부터)

숫자 상수의 가독성을 높이기 위해 밑줄을 사용할 수 있다:

```
val oneMillion = 1_000_000
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

표현

자바 플랫폼에서는 JVM 기본 타입으로 숫자를 저장한다. 만약 null 가능 숫자 레퍼런스(예 `Int?`)가 필요하거나 지네릭이 관여하면 박싱(boxing) 타입으로 저장한다.

숫자를 박싱하면 동일성(Identity)을 유지하지 않는다:

```
val a: Int = 10000
print(a === a) // 'true' 출력
val boxedA: Int? = a
val anotherBoxedA: Int? = a
print(boxedA === anotherBoxedA) // !!!'false' 출력!!!
```

하지만 값의 동등함은 유지한다:

```
val a: Int = 10000
print(a == a) // 'true' 출력
val boxedA: Int? = a
val anotherBoxedA: Int? = a
print(boxedA == anotherBoxedA) // 'true' 출력
```

명시적 변환

표현이 다르므로 작은 타입이 큰 타입의 하위 타입은 아니다. 하위 타입이 된다면 다음과 같은 문제가 발생한다:

```
// 가상의 코드로, 실제로는 컴파일되지 않음:
val a: Int? = 1 // 박싱된 Int (java.lang.Integer)
val b: Long? = a // 자동 변환은 박싱된 Long을 생성 (java.lang.Long)
print(a == b) // 놀랍게도 이는 "false"를 리턴한다! 왜냐면 Long의 equals()은 비교 대상도 Long인지 검사하기 때문이다
```

동일성뿐만 아니라 동등함조차 모든 곳에서 나도 모르게 사라지게 된다.

이런 이유로 작은 타입을 큰 타입으로 자동으로 변환하지 않는다. 이는 명시적 변환없이 `Byte` 타입 값을 `Int` 변수에 할당할 수 없음을 뜻한다.

```
val b: Byte = 1 // OK, 리터럴은 정적으로 검사함
val i: Int = b // ERROR
```

명시적으로 숫자를 넓히는 변환을 할 수 있다:

```
val i: Int = b.toInt() // OK: 명시적으로 넓힘
```

모든 숫자 타입은 다음 변환을 지원한다:

- `toByte(): Byte`
- `toShort(): Short`
- `toInt(): Int`
- `toLong(): Long`
- `toFloat(): Float`
- `toDouble(): Double`
- `toChar(): Char`

자동 변환의 부재를 거의 느낄 수 없는데 이유는 타입을 컨텍스트에서 추론하고 수치 연산을 변환에 알맞게 오버로딩했기 때문이다. 다음은 예이다.

```
val l = 1L + 3 // Long + Int => Long
```

연산

코틀린은 숫자에 대한 표준 수치 연산을 지원한다. 이들 연산은 알맞은 클래스의 멤버로 선언되어 있다. (하지만 컴파일러는 메서드 호출을 대응하는 명령어로 최적화한다.) 연산자 오버로딩 을 참고하자.

비트 연산자를 위한 특스 문자는 없고 중의 형식으로 호출할 수 있는 함수를 제공한다. 다음은 예이다:

```
val x = (1 shl 2) and 0x000FF000
```

다음은 전체 비트 연산자 목록이다(`Int` 와 `Long` 에서만 사용 가능):

- `shl(bits)` – 부호있는 왼쪽 시프트 (자바의 `<<`)
- `shr(bits)` – 부호있는 오른쪽 시프트 (자바의 `>>`)

— `ushr(bits)` – 부호없는 오른쪽 시프트 (자바의 `>>>`)

— `and(bits)` – 비트 AND

— `or(bits)` – 비트 OR

— `xor(bits)` – 비트 XOR

— `inv()` – 비트 역

부동소수 비교

이 절에서는 실수에 대한 연산을 설명한다:

— 동등함 비교: `a == b` 와 `a != b`

— 비교 연산자: `a < b` , `a > b` , `a <= b` , `a >= b`

— 범위 생성과 검사: `a..b` , `x in a..b` , `x !in a..b`

피연산자 `a` 와 `b` 가 정적으로 `Float` 나 `Double` 이거나 또는 이에 해당하는 null 가능 타입일 때(타입을 선언하거나 추정하거나 또는 스마트 변환 의 결과), 숫자나 범위에 대한 연산은 부동소수 연산에 대한 IEEE 754 표준을 따른다.

하지만 범용적인 용례를 지원하고 완전한 순서를 제공하기 위해, 피연산자가 정적으로 부동소수점 타입이 아니면(예 `Any` , `Comparable<...>` , 타입 파라미터) 연산은 `Float` 과 `Double` 을 위한 `equals` 와 `compareTo` 구현을 사용한다. 이는 표준과 비교해 다음이 다르다.

— `NaN` 은 자신과 동일하다고 간주한다

— `NaN` 은 `POSITIVE_INFINITY` 를 포함한 모든 다른 요소보다 크다고 간주한다

— `-0.0` 는 `0.0` 보다 작다고 간주한다

문자

`Char` 타입으로 문자를 표현한다. 이 타입을 바로 숫자로 다룰 수 없다.

```
fun check(c: Char) {
    if (c == 1) { // ERROR: 비호환 타입
        // ...
    }
}
```

문자 리터럴은 작은 따옴표 안에 표시한다: `'1'` . 특수 문자를 역슬래시로 표시한다. 다음 특수문자를 지원한다: `\t` , `\b` , `\n` , `\r` , `\'` , `\"` , `\\` , `\$` . 임의의 문자를 인코딩하려면 유니코드 표기법을 사용한다: `'\uFF00'` .

문자를 `Int` 숫자로 명시적으로 변환할 수 있다:

```
fun decimalDigitValue(c: Char): Int {
    if (c !in '0'..'9')
        throw IllegalArgumentException("Out of range")
    return c.toInt() - '0'.toInt() // 숫자로 명시적 변환
}
```

숫자와 마찬가지로, 문자도 null 가능 레퍼런스가 필요하면 박싱된다. 박싱 연산을 하면 동일성은 유지되지 않는다.

불리언

`Boolean` 타입은 불리언을 표현하고 두 값이 존재한다: `true` 와 `false` .

null 가능 레퍼런스가 필요하면 불리언도 박싱된다.

불리언에 대한 내장 연산에는 다음이 있다.

— `||` – 지연 논리합

— `&&` – 지연 논리곱

— `!` - 역

배열

코틀리은 `Array` 클래스로 배열을 표현하며, 이 클래스는 `get` 과 `set` 함수(연산자 오버로딩 관례에 따라 `[]` 로 바뀜), `size` 프로퍼티와 그외 유용한 멤버 함수를 제공한다:

```
class Array<T> private constructor() {
    val size: Int
    operator fun get(index: Int): T
    operator fun set(index: Int, value: T): Unit

    operator fun iterator(): Iterator<T>
    // ...
}
```

라이브러리 함수 `arrayOf()` 를 사용해서 배열을 생성할 수 있다. 이 함수에는 항목 값을 전달하는데 `arrayOf(1, 2, 3)` 은 [1, 2, 3] 배열을 생성한다. `arrayOfNulls()` 라이브러리 함수를 사용하면 주어진 크기의 null로 채운 배열을 생성할 수 있다.

배열을 생성하는 다른 방법은 팩토리 함수를 사용하는 것이다. 팩토리 함수는 배열 크기와 해당 인덱스에 위치한 요소의 초기값을 리턴하는 함수를 인자로 받는다:

```
// Creates an Array<String> with values ["0", "1", "4", "9", "16"]
val asc = Array(5, { i -> (i * i).toString() })
```

앞서 말했듯이, `[]` 연산자는 `get()` 과 `set()` 멤버 함수 호출을 의미한다.

주의: 자바와 달리 코틀린 배열은 무공변(invariant)이다. 이는 코틀린은 `Array<String>` 을 `Array<Any>` 에 할당할 수 없음을 의미하며, 런타임 실패를 방지한다. (하지만 `Array<out Any>` 을 사용할 수 있다. 타입 프로젝션 을 참고하자.)

코틀린은 또한 `ByteArray` , `ShortArray` , `IntArray` 등 기본 타입의 배열을 표현하기 위한 특수 클래스를 제공한다. 이들 클래스는 박싱 오버헤드가 없다. 이 클래스는 Array 클래스와 상속 관계에 있지 않지만 같은 메서드와 프로퍼티를 제공한다. 각 배열 타입을 위한 팩토리 함수를 제공한다:

```
val x: IntArray = intArrayOf(1, 2, 3)
x[0] = x[1] + x[2]
```

문자열

문자열은 `String` 타입으로 표현한다. 문자열은 불변이다. 문자열의 요소는 문자로서 `s[i]` 와 같은 인덱스 연산으로 접근할 수 있다. `for` -루프로 문자열을 순회할 수 있다:

```
for (c in str) {
    println(c)
}
```

문자열 리터럴

코틀린은 두 가지 타입의 문자열 리터럴을 지원한다. 하나는 escaped 문자열로 탈출 문자를 가질 수 있다. 다른 하나는 raw 문자열로 뉴라인과 임의 텍스트를 가질 수 있다. escaped 문자열은 자바 문자열과 거의 같다:

```
val s = "Hello, world!\n"
```

특수 문자는 전형적인 방식인 역슬래시를 사용한다. 지원하는 특수 문자 목록은 앞서 문자 를 참고한다.

raw 문자열은 세 개의 따옴표로 구분하며(`"""`), 특수 문자를 포함하지 않고 뉴라인과 모든 다른 문자를 포함할 수 있다:

```
val text = """
    for (c in "foo")
        print(c)
"""
```

`trimMargin()` 함수를 사용해서 앞쪽 공백을 제거할 수 있다:

```
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

기본으로 `\` 를 경계 접두문자로 사용하지만 `trimMargin(">")` 과 같이 파라미터를 이용해서 다른 문자를 경계 문자로 사용할 수 있다.

문자열 템플릿

문자열은 템플릿 식을 포함할 수 있다. 예를 들어, 코드 조각을 계산하고 그 결과를 문자열에 넣을 수 있다. 템플릿 식은 달러 부호($)로 시작하고 간단한 이름으로 구성된다:

```
val i = 10
val s = "i = $i" // "i = 10"로 평가
```

또는 중괄호 안에 임의의 식을 넣을 수 있다:

```
val s = "abc"
val str = "$s.length is ${s.length}" // "abc.length is 3"로 평가
```

raw 문자열과 escaped 문자열 안에 템플릿을 넣을 수 있다. 역슬래시 특수 문자를 지원하지 않는 raw 문자열에서 `$` 문자를 표현하고 싶다면 다음 구문을 사용한다:

```
val price = """
${'$'}9.99
"""
```

패키지

소스 파일은 패키지 선언으로 시작한다:

```
package foo.bar

fun baz() {}

class Goo {}

// ...
```

소스 파일의 모든 내용(클래스와 함수 등)은 선언된 패키지에 포함된다. 위 예의 경우 `baz()` 의 전체 이름은 `foo.bar.baz` 이고 `Goo` 의 전체 이름은 `foo.bar.Goo` 이다.

패키지를 지정하지 않으면, 파일의 내용은 이름이 없는 "디폴트" 패키지에 속한다.

디폴트 임포트

모든 코틀린 파일은 다음 패키지를 기본적으로 임포트한다:

- kotlin.*
- kotlin.annotation.*
- kotlin.collections.*
- kotlin.comparisons.* (1.1부터)
- kotlin.io.*
- kotlin.ranges.*
- kotlin.sequences.*
- kotlin.text.*

대상 플랫폼에 따라 추가 패키지를 임포트한다:

- JVM:
  - java.lang.*
  - kotlin.jvm.*
- JS:
  - kotlin.js.*

임포트

디폴트 임포트에 추가로 각 파일은 자신만의 import 디렉티브를 포함할 수 있다. import 문법은 문법 에서 설명한다.

한 개 이름을 임포트하는 예:

```
import foo.Bar // 완전한 이름이 아닌 Bar로 접근 가능
```

범위(패키지, 클래스, 오브젝트 등)의 모든 내용에 접근 가능:

```
import foo.* // 'foo'의 모든 것에 접근 가능
```

이름 충돌이 발생하면, `as` 키워드로 충돌하는 대상에 다른 이름을 부여해서 모호함을 없앨 수 있다:

```
import foo.Bar // Bar로 접근 가능
import bar.Bar as bBar // bBar로 'bar.Bar'에 접근 가능
```

`import` 키워드는 클래스뿐만 아니라 다음 선언을 임포트할 수 있다:

- 최상위 함수와 프로퍼티;
- 오브젝트 선언 에 있는 함수와 프로퍼티;
- enum 상수 .

자바와 달리 코틀린은 별도의 "import static" 구문이 없다. `import` 키워드로 모든 선언을 임포트할 수 있다.

최상위 선언의 가시성

최상위 선언을 `private` 로 선언한 경우, 선언된 파일에 대해 private 이다(가시성 수식어 를 보자).

흐름 제어: if, when, for, while

If 식

코틀린에서 `if` 는 식(expression)으로, 값을 리턴한다. 그래서 삼항 연산자(condition ? then : else)가 없다. 일반 `if` 로 동일하게 할 수 있기 때문이다.

```
// 전통적인 용법
var max = a
if (a < b) max = b

// else 사용
var max: Int
if (a > b) {
    max = a
} else {
    max = b
}

// 식으로
val max = if (a > b) a else b
```

`if` 브랜치는 블록일 수 있으며 마지막 식이 블록이 값이 된다.

```
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```

문장이 아닌 식으로 `if` 를 사용하면(예를 들어 식의 값을 리턴하거나 변수에 값을 할당), `else` 브랜치를 가져야 한다.

`if 문법` 을 보자.

When 식

`when` 은 C와 같은 언어의 switch 연산에 해당한다. 가장 간단한 형식은 다음과 같다:

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> { // Note the block
        print("x is neither 1 nor 2")
    }
}
```

`when` 은 특정 브랜치의 조건을 충족할 때까지 순차적으로 모든 브랜치의 인자가 일치하는지 확인한다. `when` 은 식이나 문장으로 사용할 수 있다. 식으로 사용하면 충족한 브랜치의 값이 전체 식의 값이 된다. 문장으로 사용하면 개별 브랜치의 값은 무시한다. (`if` 처럼 각 브랜치는 블록이 될 수 있으며, 블록의 마지막 식의 값이 브랜치의 값이 된다.)

모든 `when` 브랜치가 충족하지 않으면 `else` 브랜치를 평가한다. `when` 을 식으로 사용할 때, 모든 가능한 경우를 브랜치 조건으로 확인했는지 컴파일러가 알 수 없는 경우 `else` 브랜치를 필수로 추가해야 한다.

여러 경우를 동일한 방법으로 처리할 경우 브랜치 조건을 콤마로 묶을 수 있다:

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

브랜치 조건으로 (상수뿐만 아니라) 임의의 식을 사용할 수 있다.

```
when (x) {
    parseInt(s) -> print("s encodes x")
    else -> print("s does not encode x")
}
```

`in` , `!in` , 범위 , 콜렉션을 사용해서 값을 검사할 수 있다:

```
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
}
```

`is` 나 `!is` 로 특정 타입 여부를 확인할 수 있다. 스마트 변환 덕분에 추가 검사없이 타입의 메서드와 프로퍼티에 접근할 수 있다.

```
fun hasPrefix(x: Any) = when(x) {
    is String -> x.startsWith("prefix")
    else -> false
}
```

`if - else if` 체인 대신에 `when` 을 사용할 수도 있다. 인자를 제공하지 않으면 브랜치 조건은 단순한 블리언 식이 되고, 브랜치의 조건이 true일 때 브랜치를 실행한다:

```
when {
    x.isOdd() -> print("x is odd")
    x.isEven() -> print("x is even")
    else -> print("x is funny")
}
```

`when 문법` 을 참고하자.

for 루프

`for` 는 이터레이터를 제공하는 모든 것에 대해 반복을 수행한다. C#과 같은 언어의 `foreach` 루프와 동일하다. 구문은 다음과 같다:

```
for (item in collection) print(item)
```

몸체는 블록일 수 있다.

```
for (item: Int in ints) {
    // ...
}
```

이전에 언급했듯이, `for` 는 이터레이터를 제공하는 모든 것에 동작한다.

- `iterator()` 멤버 함수나 확장 함수를 갖고 있고, 함수의 리턴 타입이
  - `next()` 멤버 함수나 확장 함수를 갖고,
  - 리턴 타입이 `Boolean` 인 `hasNext()` 멤버 함수나 확장 함수를 가짐

이 세 함수는 모두 `operator` 로 지정해야 한다.

배열에 대한 `for` 루프는 이터레이터 객체를 생성하지 않는 인덱스 기반 루프로 컴파일된다.

인덱스를 이용해서 배열이나 리스트를 반복하려면 다음과 같이 하면 된다:

```
for (i in array.indices) {
    print(array[i])
}
```

"범위에 대한 반복"은 최적화한 구현으로 컴파일해서 객체를 추가로 생성하지 않는다.

indices 대신 `withIndex` 라이브러리 함수를 사용해도 된다:

```
for ((index, value) in array.withIndex()) {
    println("the element at $index is $value")
}
```

for 문법 을 참고하자.

While 루프

while 과 do .. while 예:

```
while (x > 0) {
    x--
}

do {
    val y = retrieveData()
} while (y != null) // 여기서 y를 사용할 수 있다!
```

while 문법 을 참고하자.

루프에서의 break와 continue

코틀린은 루프에서 전통적인 break 와 continue 연산자를 지원한다. 리턴과 점프 를 참고한다.

리턴과 점프

코틀린에는 세 가지 구조의 점프 식이 있다.

- `return` : 기본으로 가장 가깝게 둘러싼 함수나 익명 함수 에서 리턴한다.
- `break` : 가장 가깝게 둘러싼 루프를 끝낸다.
- `continue` : 가장 가깝게 둘러싼 루프의 다음 단계를 진행한다.

세 식 모두 더 큰 식의 일부로 사용할 수 있다:

```
val s = person.name ?: return
```

이 세 식의 타입은 Nothing 이다.

break와 continue 라벨

코틀린의 모든 식에 `label` 을 붙일 수 있다. 라벨은 `@` 부호 뒤에 식별자가 붙는 형식으로, 예를 들어 `abc@` , `fooBar@` 는 유효한 라벨이다(문법 참고). 식 앞에 라벨을 위치시켜 라벨을 붙인다.

```
loop@ for (i in 1..100) {
    // ...
}
```

이제 라벨을 사용해서 `break` 나 a `continue` 를 한정할 수 있다:

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (...) break@loop
    }
}
```

라벨로 한정한 `break` 는 해당 라벨이 붙은 루프 이후로 실행 지점을 점프한다. `continue` 는 루프의 다음 반복을 진행한다.

라벨에 리턴하기

코틀린은 함수 리터럴, 로컬 함수, 오브젝트 식에서 함수를 중첩할 수 있는데, 한정한 `return` 을 사용하면 바깥 함수로부터 리턴할 수 있다. 가장 중요한 용도는 람다 식에서 리턴하는 것이다. 아래 코드를 보자:

```
fun foo() {
    ints.forEach {
        if (it == 0) return  // 내부 람다에서 foo()의 콜러로 바로 리턴하는 비로컬 리턴
        print(it)
    }
}
```

`return` -식은 가장 가깝게 둘러싼 함수(예, `foo`)에서 리턴한다. (이런 비로컬 리턴은 인라인 함수)로 전달한 람다 식에서만 지원한다.) 람다 식에서 리턴하고 싶다면 담다 식에 라벨을 붙여 `return` 을 한정해야 한다:

```
fun foo() {
    ints.forEach lit@ {
        if (it == 0) return@lit
        print(it)
    }
}
```

이제 위 코드는 람다 식에서 리턴한다. 종종 암묵적으로 제공하는 라벨을 사용하는 게 편리할 때가 있다. 이런 라벨은 람다를 전달한 함수와 같은 이름을 갖는다.

```
fun foo() {
    ints.forEach {
        if (it == 0) return@forEach
        print(it)
    }
}
```

람다 식 대신 익명 함수 를 사용해도 된다. 익명 함수에서 `return` 문장은 익명 함수 자체에서 리턴한다.

```
fun foo() {
    ints.forEach(fun(value: Int) {
        if (value == 0) return  // 익명 함수 호출에 대한 로컬 리턴. 예, forEach 루프로 리턴
        print(value)
    })
}
```

값을 리턴할 때 파서는 한정한 리턴에 우선순위를 준다.

```
return@a 1
```

이 코드는 "라벨 `@a` 에 `1` 을 리턴"하는 것을 의미한다. "라벨을 붙인 식 `(@a 1)` 을 리턴"하는 것이 아니다.

클래스와 오브젝트

클래스와 상속

클래스

코틀린에서 클래스는 `class` 키워드를 사용해서 선언한다:

```
class Invoice {
}
```

클래스 선언은 클래스 이름과 클래스 헤더(타입 파라미터를 지정, 주요 생성자 등), 중괄호로 둘러 싼 클래스 몸체로 구성된다. 헤더와 몸체는 선택적이다. 클래스에 몸체가 없으면 중괄호를 생략할 수 있다.

```
class Empty
```

생성자

코틀린 클래스는 **주요(primary) 생성자** 와 한 개 이상의 **보조(secondary) 생성자** 를 가질 수 있다. 주요 생성자는 클래스 헤더의 한 부분으로 클래스 이름(그리고 필요에 따라 타입 파라미터) 뒤에 위치한다.

```
class Person constructor(firstName: String) {
}
```

주요 생성자에 애노테이션이나 가시성 수식어가 없으면, `constructor` 키워드를 생략할 수 있다:

```
class Person(firstName: String) {
}
```

주요 생성자는 어떤 코드도 포함할 수 없다. **초기화 블록** 에 초기화 코드가 위치할 수 있다. 초기화 블록에는 `init` 키워드를 접두사로 붙인다:

```
class Customer(name: String) {
    init {
        logger.info("Customer initialized with value ${name}")
    }
}
```

초기화 블록에서 주요 생성자의 파라미터를 사용할 수 있다. 주요 생성자 파라미터는 클래스 몸체에 선언한 프로퍼티 초기화에서도 사용할 수 있다:

```
class Customer(name: String) {
    val customerKey = name.toUpperCase()
}
```

사실 코틀린은 주요 생성자에서 프로퍼티를 선언하고 초기화할 수 있는 간결한 구문을 제공한다:

```
class Person(val firstName: String, val lastName: String, var age: Int) {
    // ...
}
```

일반 프로퍼티와 마찬가지로 주요 생성자에 선언한 프로퍼티는 수정가능(`var`)이거나 읽기 전용(`val`)일 수 있다.

생성자가 애노테이션이나 가시성 수식어를 가지면 `constructor` 키워드가 필요하며, 키워드 앞에 수식어가 온다:

```
class Customer public @Inject constructor(name: String) { ... }
```

자세한 내용은 가시성 수식어 를 참고한다.

보조 생성자

클래스는 **보조 생성자** 를 선언할 수 있다. 보조 생성자는 `constructor` 를 접두사로 붙인다:

```
class Person {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

클래스에 주요 생성자가 있다면, 각 보조 생성자는 직접적으로 또는 다른 보조 생성자를 통해 간접적으로 주요 생성자를 호출해야 한다. 같은 클래스의 다른 생성자를 호출할 때에는 `this` 키워드를 사용한다:

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

비추상 클래스에 어떤 생성자(주요 또는 보조)도 없으면, 인자가 없는 주요 생성자를 생성한다. 이 생성자의 가시성은 public이다. public 생성자를 원하지 않으면 기본 가시성이 아닌 빈 주요 생성자를 선언해야 한다.

```
class DontCreateMe private constructor () {
}
```

주의 : JVM에서 주요 생성자의 모든 파라미터가 기본 값을 가지면, 컴파일러는 추가로 파라미터가 없는 생성자를 생성한다. 이 생성자는 기본 값을 사용하는 주요 생성자를 사용한다. 이는 Jackson이나 JPA처럼 파라미터가 없는 생성자를 사용해서 인스턴스를 생성하는 라이브러리에서 코틀린을 쉽게 사용할 수 있게 해준다.

```
class Customer(val customerName: String = "")
```

클래스의 인스턴스 생성

클래스의 인스턴스를 생성하려면 일반 함수와 같이 생성자를 호출한다:

```
val invoice = Invoice()

val customer = Customer("Joe Smith")
```

코틀린에는 `new` 키워드가 없음에 유의하자.

중첩된, 내부 또는 임의 내부 클래스의 생성은 중첩 클래스 을 참고한다.

클래스 멤버

클래스는 다음을 가질 수 있다:

- 생성자와 초기화 블록
- 함수
- 프로퍼티
- 중첩 클래스와 내부 클래스
- 오브젝트 선언

상속

코틀린의 모든 클래스는 공통의 최상위 클래스인 `Any` 를 상속한다. 상위 타입을 선언하지 않으면 `Any` 가 기본 상위 타입이다:

```
class Example // 기본으로 Any를 상속한다
```

`Any` 는 `java.lang.Object` 가 아니다. 특히 `Any` 는 `equals()` , `hashCode()` , `toString()` 외에 다른 멤버를 갖지 않는다. 더 자세한 내용은 자바 상호운용성 을 참고한다.

상위타입을 직접 선언하려면 클래스 헤더에서 콜론 뒤에 타입을 위치시킨다:

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

클래스에 주요 생성자가 있으면, 주요 생성자의 파라미터를 사용해서 기반 타입을 그 자리에서 초기화할 수 있다.

클래스에 주요 생성자가 없으면, 각 보조 생성자에서 `super` 키워드로 기반 타입을 초기화하거나 그걸 하는 다른 생성자를 호출해야 한다.

```
class MyView : View {
    constructor(ctx: Context) : super(ctx)

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
}
```

`open` 애노테이션은 자바의 `final` 과 반대이다. `open` 은 다른 클래스가 이 클래스를 상속할 수 있게 허용한다. 기본으로 코틀린의 모든 클래스는 final인데 이는 Effective Java 의 Item 17: *Design and document for inheritance or else prohibit it* 를 따른 것이다.

메서드 오버라이딩

코틀린은 메서드 오버라이딩을 명시적으로 해야 한다. 자바와 달리 코틀린에서는 오버라이딩 가능한 멤버에 (*open*) 애노테이션을 명시적으로 설정해야 한다:

```
open class Base {
    open fun v() {}
    fun nv() {}
}
class Derived() : Base() {
    override fun v() {}
}
```

`Derived.v()` 에는 `override` 애노테이션이 필요하다. 이를 누락하면 컴파일에 실패한다. `Base.nv()` 처럼 `open` 애노테이션이 없는 경우, `override` 를 사용하든 안 하든 하위클래스에서 동일 시그너처를 갖는 메서드를 선언할 수 없다. final 클래스(예, `open` 애노테이션이 없는 클래스)는 open 멤버를 가질 수 없다.

`override` 가 붙은 멤버는 그 자체가 open 이며, 하위 클래스에서 다시 오버라이딩하는 것을 막고 싶다면 `final` 을 사용해야 한다:

```
open class AnotherDerived() : Base() {
    final override fun v() {}
}
```

프로퍼티 오버라이딩

프로퍼티 오버라이딩은 메서드 오버라이딩과 유사하게 동작한다. 상위클래스에 선언된 프로퍼티를 하위 클래스에 재선언하려면 `override` 를 사용해야 하며 호환되는 타입을 사용해야 한다. 선언한 프로퍼티는 initializer를 가진 프로퍼티나 getter 메서드를 가진 프로퍼티로 오버라이딩할 수 있다.

```
open class Foo {
    open val x: Int get() { ... }
}

class Bar1 : Foo() {
    override val x: Int = ...
}
```

또한 `val` 프로퍼티를 `var` 프로퍼티로 재정의할 수 있지만 반대는 안 된다. 이것을 허용하는 이유는, `var` 프로퍼티는 근본적으로 getter 메서드를 선언하는데 그것을 `var` 로 오버라이딩하면 추가로 하위 클래스에서 setter 메서드를 선언하기 때문이다.

주요 생성자에 선언한 프로퍼티에도 `override` 키워드를 사용할 수 있다.

```
interface Foo {
    val count: Int
}

class Bar1(override val count: Int) : Foo

class Bar2 : Foo {
    override var count: Int = 0
}
```

상위클래스 구현 호출

하위클래스는 `super` 키워드를 사용해서 상위클래스의 함수와 프로퍼티 접근 구현를 호출할 수 있다:

```
open class Foo {
    open fun f() { println("Foo.f()") }
    open val x: Int get() = 1
}

class Bar : Foo() {
    override fun f() {
        super.f()
        println("Bar.f()")
    }

    override val x: Int get() = super.x + 1
}
```

내부 클래스는 `super@Outer` 와 같이 외부 클래스의 이름을 사용해서 외부 클래스의 상위 클래스에 접근할 수 있다:

```
class Bar : Foo() {
    override fun f() { /* ... */ }
    override val x: Int get() = 0

    inner class Baz {
        fun g() {
            super@Bar.f() // Foo의 f()의 구현
            println(super@Bar.x) // Foo의 x의 getter 사용
        }
    }
}
```

오버라이딩 규칙

코틀린의 구현 상속은 다음 규칙에 따라 제한된다: 클래스가 바로 위의 상위 클래스에서 같은 멤버의 구현을 여러 개 상속받으면, 반드시 멤버를 오버라이딩하고 자신의 구현(아마도, 상속한 것 중 하나를 사용하는 코드)를 제공해야 한다. 어떤 상위타입의 구현을 사용할지 선택하려면 홑화살괄호 안에 상위 타입의 이름을 붙인 `super` 를 사용한다.

```
open class A {
    open fun f() { print("A") }
    fun a() { print("a") }
}

interface B {
    fun f() { print("B") } // 인터페이스 멤버는 기본이 'open'이다
    fun b() { print("b") }
}

class C() : A(), B {
    // 컴파일러는 f()를 오버라이딩할 것을 요구한다:
    override fun f() {
        super<A>.f() // A.f() 호출
        super<B>.f() // B.f() 호출
    }
}
```

`A` 와 `B` 를 둘 다 상속해도 `a()` 와 `b()` 는 문제가 되지 않는다. `C` 가 이 함수의 구현을 한 개만 상속하기 때문이다. 하지만 `f()` 의 경우 `C` 는 두 개의 구현을 상속받으므로 `C` 에 `f()` 를 오버라이딩하고 자신의 구현을 제공해서 애매함을 없애야 한다.

추상 클래스

클래스나 클래스의 멤버를 `abstract` 로 선언할 수 있다. 추상 멤버는 그 클래스에 구현을 갖지 않는다. 추상 클래스나 함수에 open을 붙일 필요는 없다.

추상이 아닌 open 멤버를 추상 멤버로 오버라이딩할 수 있다.

```
open class Base {
    open fun f() {}
}

abstract class Derived : Base() {
    override abstract fun f()
}
```

컴페니언 오브젝트

코틀린은 자바나 C#과 달리 클래스에 정적 메서드가 없다. 많은 경우 간단하게 패키지 수준의 함수를 대신 사용할 것을 권한다.

클래스 인스턴스없이 클래스의 내부에 접근해야 하는 함수를 작성해야 한다면(예를 들어, 팩토리 메서드), 그 클래스에 속한 오브젝트 선언의 멤버로 함수를 작성할 수 있다.

더 구체적으로 컴페니언 오브젝트를 클래스 안에 선언하면, 한정자로 클래스 이름을 사용해서 자바나 C#의 정적 메서드를 호출하는 것과 동일 구문으로 컴페니언 오브젝트의 멤버를 호출할 수 있다.

프로퍼티와 필드

프로퍼티 선언

코틀린에서 클래스는 프로퍼티를 가질 수 있다. `var` 키워드로 변경 가능하게 선언하거나 `val` 키워드로 읽기 전용으로 선언할 수 있다.

```
class Address {
    var name: String = ...
    var street: String = ...
    var city: String = ...
    var state: String? = ...
    var zip: String = ...
}
```

자바 필드처럼 이름으로 프로퍼티를 참조할 수 있다:

```
fun copyAddress(address: Address): Address {
    val result = Address() // 코틀린에 'new' 키워드 없음
    result.name = address.name // 접근자(accessor)를 실행
    result.street = address.street
    // ...
    return result
}
```

Getters와 Setters

프로퍼티 선언의 전체 구문은 다음과 같다.

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]
    [<getter>]
    [<setter>]
```

initializer, getter, setter는 선택사항이다. 프로퍼티 타입은 initializer(또는 다음에 보여줄 getter)의 리턴 타입에서 유추할 수 있으면 생략가능하다.

예제:

```
var allByDefault: Int? // 에러: initializer 필요, 기본 getter와 setter 적용
var initialized = 1 // Int 타입을 갖고, 기본 getter와 setter
```

읽기 전용 프로퍼티 선언은 변경가능 프로퍼티 선언과 두 가지가 다르다. `var` 대신에 `val` 로 시작하고 setter를 허용하지 않는다:

```
val simple: Int? // Int 타입이고, 기본 getter, 생성자에서 초기화해야 함
val inferredType = 1 // Int 타입이고, 기본 getter
```

일반 함수처럼 프로퍼티 선언에 커스텀 접근자를 작성할 수 있다. 다음은 커스텀 getter의 예이다:

```
val isEmpty: Boolean
    get() = this.size == 0
```

커스텀 setter는 다음과 같다:

```
var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value) // 문자열을 파싱해서 다른 프로퍼티에 값을 할당한다
    }
```

setter 파라미터의 이름을 `value` 로 하는 것은 관례인데, 선호하는 다른 이름을 사용할 수 있다.

코틀린 1.1부터, getter에서 타입을 추론할 수 있으면 프로퍼티 타입을 생략할 수 있다:

```
val isEmpty get() = this.size == 0  // Boolean 타입임
```

프로퍼티의 기본 구현을 바꾸지 않고 애노테이션을 붙이거나 접근자의 가시성을 바꾸고 싶다면 몸체 없는 접근자를 정의하면 된다:

```
var setterVisibility: String = "abc"
    private set // setter를 private으로 하고 기본 구현을 가짐

var setterWithAnnotation: Any? = null
    @Inject set // setter에 @Inject 애노테이션 적용
```

지원(Backing) 필드

코틀린 클래스는 필드를 가질 수 없다. 하지만 때때로 커스텀 접근자를 사용할 때 지원(backing) 필드가 필요할 때가 있다. 이런 목적으로 코틀린은 `field` 식별자로 접근할 수 있는 지원 필드를 자동으로 제공한다:

```
var counter = 0 // 초기값을 지원 필드에 직접 쓴다
    set(value) {
        if (value >= 0) field = value
    }
```

`field` 식별자는 오직 프로퍼티의 접근자에서만 사용할 수 있다.

접근자의 기본 구현을 적어도 한 개 이상 사용하거나 또는 커스텀 접근자에서 `field` 식별자로 지원 필드에 접근할 경우, 프로퍼티를 위한 지원 필드를 생성한다.

예를 들어 다음 경우 지원 필드가 없다:

```
val isEmpty: Boolean
    get() = this.size == 0
```

지원(Backing) 프로퍼티

"자동 지원 필드" 방식이 맞지 않을 때 *지원(backing) 프로퍼티*로 대신할 수 있다:

```
private var _table: Map<String, Int>? = null
public val table: Map<String, Int>
    get() {
        if (_table == null) {
            _table = HashMap() // 타입 파라미터 추론
        }
        return _table ?: throw AssertionError("Set to null by another thread")
    }
```

모든 면에서, 이는 자바와 거의 같다. 왜냐면 기본 getter와 setter를 가진 private 프로퍼티에 접근하면 함수 호출에 따른 오버헤드가 발생하지 않도록 최적화하기 때문이다.

컴파일 타임 상수

컴파일 시점에 알아야 할 프로퍼티 값을 `const` 수식어를 이용해서 _컴파일 타임 상수_로 표시할 수 있다. 그런 프로퍼티는 다음 조건을 충족해야 한다:

- 최상위 또는 `오브젝트`의 멤버
- `String`이나 기본 타입 값으로 초기화
- 커스텀 getter가 아님

이런 프로퍼티는 애노테이션에서 사용할 수 있다:

```
const val SUBSYSTEM_DEPRECATED: String = "This subsystem is deprecated"

@Deprecated(SUBSYSTEM_DEPRECATED) fun foo() { ... }
```

초기화 지연(Late-Initialized) 프로퍼티

보통 null이 아닌 타입으로 선언한 프로퍼티는 생성자에서 초기화해야 한다. 하지만 이게 편리하지 않을 때가 있다. 예를 들어 의존 주입이나 단위 테스트의 셋업 메서드에서 프로퍼티를 초기화한다고 하자. 이 경우 생성자에 null이 아닌 초기값을 제공할 수는 없는데, 클래스 몸체에서는 프로퍼티를 참조할 때 null 검사는 피하고 싶을 것이다.

이런 경우 프로퍼티에 `lateinit` 수식어를 붙일 수 있다:

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method()  // 직접 참조하는 객체에 접근
    }
}
```

`lateinit` 는 (주요 생성자가 아닌) 클래스 몸체에 정의된 `var` 프로퍼티가 커스텀 getter나 setter가 없는 경우에만 적용할 수 있다. 프로퍼티 타입은 null 불가여야 하고 기본 타입이면 안 된다.

`lateinit` 프로퍼티를 초기화하기 전에 접근하면 특수한 익셉션을 발생한다. 이 익셉션은 접근한 프로퍼티와 아직 초기화하지 않았다는 것을 명확하게 식별한다.

프로퍼티 오버라이딩

프로퍼티 오버라이딩 을 보자.

위임 프로퍼티(Delegated Properties)

보통 대부분 프로퍼티는 단순히 지원 필드에서 읽어오거나 쓴다. 반면에 커스텀 getter와 setter를 가진 프로퍼티는 동작을 구현할 수 있다. 그 중간에 프로퍼티가 동작하는 공통 패턴이 존재한다. 패턴의 예로 지연 값, 주어진 키로 맵에서 읽기, 데이터베이스 접근하기, 접근할 때 리스너에 통지하기 등이 있다.

*위임 프로퍼티*를 사용해서 그런 공통 기능을 구현할 수 있다.

인터페이스

코틀린의 인터페이스는 자바 8 인터페이스와 매우 유사하다. 추상 메서드를 선언할 수 있고 또한 메서드 구현을 포함할 수 있다. 추상 클래스와의 차이점은 인터페이스는 상태를 가질 수 없다는 것이다. 인터페이스가 프로퍼티를 가질 수 있지만 프로퍼티는 추상이거나 또는 접근자 구현을 제공해야 한다.

인터페이스는 `interface` 키워드로 정의한다.

```
interface MyInterface {
    fun bar()
    fun foo() {
      // optional body
    }
}
```

인터페이스 구현

클래스나 오브젝트는 한 개 이상의 인터페이스를 구현할 수 있다.

```
class Child : MyInterface {
    override fun bar() {
        // body
    }
}
```

인터페이스의 프로퍼티

인터페이스에 프로퍼티를 선언할 수 있다. 인터페이스에 선언한 프로퍼티는 추상이거나 또는 접근자를 위한 구현을 제공할 수 있다. 인터페이스의 프로퍼티는 지원(backing) 필드를 가질 수 없으므로, 인터페이스에 선언한 프로퍼티에서 지원 필드를 참조할 수 없다.

```
interface MyInterface {
    val prop: Int // 추상

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(prop)
    }
}

class Child : MyInterface {
    override val prop: Int = 29
}
```

오버라이딩 충돌 해결

상위 타입 목록에 여러 타입을 지정하면, 같은 메서드에 대한 구현을 두 개 이상 상속받을 수 있다. 다음은 예이다.

```
interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }

    override fun bar() {
        super<B>.bar()
    }
}
```

인터페이스 *A* 와 *B* 는 둘다 *foo()* 와 *bar()* 함수를 선언하고 있다. 둘 다 *foo()* 함수를 구현하고 있고 *bar()* 는 *B* 만 구현하고 있다. (*A* 에서 *bar()* 에 abstract를 붙이지 않았는데 그 이유는 인터페이스의 메서드는 몸체가 없으면 기본으로 추상이기 때문이다.) *A* 를 상속받은 콘크리트 클래스인 *C* 는 당연히 *bar()* 를 오버라이딩해서 구현을 제공해야 한다.

하지만 *A* 와 *B* 를 상속한 *D* 는 두 인터페이스에서 상속한 모든 메서드를 구현해야 하며 *D* 가 어떻게 메서드를 구현할지 지정해야 한다. 이 규칙은 한 개 구현을 상속한 메서드(*bar()*)나 여러 구현을 상속한 메서드(*foo()*)에 모두 적용된다.

가시성 수식어

클래스, 오브젝트, 인터페이스, 생성자, 함수, 프로퍼티 및 프로퍼티의 setter는 _가시성 수식어_를 가질 수 있다. (getter는 항상 프로퍼티와 동일한 가시성을 갖는다.) 코틀린에는 `private`, `protected`, `internal`, `public`의 가시성 수식어가 있다. 수식어를 지정하지 않을 경우 기본 가시성은 `public`이다.

타입과 선언 범위에 따른 가시성에 대해서는 뒤에서 설명한다.

패키지

함수, 프로퍼티와 클래스, 오브젝트와 인터페이스는 "최상위(top-level)"에 선언할 수 있다. 예를 들어 패키지에 직접 선언할 수 있다.

```
// file name: example.kt
package foo

fun baz() {}
class Bar {}
```

- 가시성 수식어를 명시하지 않으면 기본으로 `public`을 사용한다. 이는 모든 곳에서 접근 가능함을 뜻한다.
- `private`으로 선언하면, 그 선언을 포함한 파일 안에서만 접근할 수 있다.
- `internal`로 선언하면, 같은 모듈에서 접근 가능하다.
- `protected`는 최상위 선언에 사용할 수 없다.

예제:

```
// 파일 이름: example.kt
package foo

private fun foo() {} // example.kt 안에서 접근 가능

public var bar: Int = 5 // 모든 곳에서 접근 가능
    private set         // setter는 example.kt에서만 접근 가능

internal val baz = 6    // 같은 모듈에서 접근 가능
```

클래스와 인터페이스

클래스에 선언한 멤버에 대해서는 다음과 같다:

- `private`은 오직 클래스 안에서만(그리고 클래스의 모든 멤버에서) 접근 가능함을 의미한다.
- `protected` — `private` + 하위클래스에서 접근 가능함과 동일하다.
- `internal` — 선언한 클래스를 볼 수 있는 *모듈 안의* 모든 클라이언트가 `internal` 멤버를 볼 수 있다.
- `public` — 선언한 클래스를 볼 수 있는 클라이언트가 `public` 멤버를 볼 수 있다.

주의 자바와 달리 코틀린에서 외부 클래스는 내부 클래스의 private 멤버를 볼 수 없다.

`protected` 멤버를 오버라이딩할 때 가시성을 명시적으로 지정하지 않으면, 오버라이딩한 멤버 또한 `protected` 가시성을 갖는다.

예제:

```
open class Outer {
    private val a = 1
    protected open val b = 2
    internal val c = 3
    val d = 4  // 기본으로 public

    protected class Nested {
        public val e: Int = 5
    }
}

class Subclass : Outer() {
    // a는 접근 불가
    // b, c, d는 접근 가능
    // Nested와 e는 접근 가능

    override val b = 5   // 'b'는 protected
}

class Unrelated(o: Outer) {
    // o.a, o.b는 접근 불가
    // o.c 와 o.d는 접근 가능(같은 모듈)
    // Outer.Nested는 접근 불가며, Nested::e 역시 접근 불가
}
```

생성자

```
class C private constructor(a: Int) { ... }
```

위 코드의 생성자는 private이다. 기본적으로 모든 생성자는 `public` 이며 실질적으로 클래스를 접근할 수 있는 모든 곳에서 생성자에 접근할 수 있다. (예를 들어 `internal` 클래스의 생성자는 오직 같은 모듈에서만 보인다.)

로컬 선언

로컬 변수, 로컬 함수, 로컬 클래스에는 가시성 수식어를 지정할 수 없다.

모듈

`internal` 가시성 수식어는 같은 모듈에서 멤버에 접근할 수 있음을 의미한다. 더 구체적으로 모듈은 함께 컴파일되는 코틀린 파일 집합이다.

- IntelliJ IDEA 모듈
- 메이븐 프로젝트
- 그레이들 소스 집합
- kotlinc 앤트 태스크를 한 번 호출할 때 컴파일되는 파일 집합

확장(Extensions)

C#이나 Gosu처럼, 코틀린은 클래스 상속이나 데코레이터 같은 설계 패턴없이 클래스에 새로운 기능을 확장할 수 있는 기능을 제공한다. _확장_이라고 부르는 특별한 선언을 통해 기능을 확장한다. 코틀린은 _확장 함수_와 _확장 프로퍼티_를 지원한다.

확장 함수

확장 함수를 선언하려면 _리시버(receiver) 타입_의 이름을 접두어로 가져야 한다. 리시버 타입의 이름은 확장할 타입의 이름이다. 다음 코드는 `MutableList<Int>`에 `swap` 함수를 추가한다:

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this'는 List에 대응한다.
    this[index1] = this[index2]
    this[index2] = tmp
}
```

확장 함수에서 `this` 키워드는 리시버 객체에 대응한다(리시버 객체는 함수 이름에서 점 앞에 위치한 타입이다):

```
val l = mutableListOf(1, 2, 3)
l.swap(0, 2) // 'swap()'에서 'this'는 'l' 값을 갖는다
```

이 함수는 `MutableList<T>`에도 적용되므로 지네릭으로 만들 수 있다:

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this'는 List에 대응한다
    this[index2] = this[index1]
    this[index2] = tmp
}
```

리시버 타입 식에서 사용할 수 지네릭 타입 파라미터를 함수 이름 앞에 선언했다. 지네릭 함수 를 참고하자.

정적인 확장 결정

확장이 실제로 확장할 클래스를 변경하지 않는다. 확장을 정의하는 것은 클래스에 새 멤버를 추가하기보다는, 그 타입의 변수에 점 부호로 호출할 수 있는 새 함수를 만드는 것이다.

확장 함수는 **정적으로** 전달된다는 점을 강조하고 싶다. 예를 들어, 리시버 타입에 따라 동적으로(virtual) 확장 함수를 결정하지 않는다. 이는 함수 호출 식의 타입에 따라 호출할 확장 함수를 결정한다는 것을 뜻한다. 런타임에 식을 평가한 결과 타입으로 결정하지 않는다. 다음 예를 보자:

```
open class C

class D: C()

fun C.foo() = "c"

fun D.foo() = "d"

fun printFoo(c: C) {
    println(c.foo())
}

printFoo(D())
```

이 예는 "c"를 출력한다. printFoo() 함수의 c 파라미터 타입이 `C` 클래스이므로 `C` 타입에 대한 확장 함수를 호출하기 때문이다.

클래스가 해당 클래스를 리시버 타입으로 갖는 확장 함수와 동일한 멤버 함수를 가진 경우, **항상 멤버 함수가 이긴다** . 다음 예를 보자:

```
class C {
    fun foo() { println("member") }
}

fun C.foo() { println("extension") }
```

`C` 의 모든 `c` 에 대해 `c.foo()` 호출은 "extension"이 아닌 "member"를 출력한다.

하지만 멤버 함수와 이름이 같지만 다른 시그너처를 갖도록 오버로딩한 확장 함수는 완전히 괜찮다:

```
class C {
    fun foo() { println("member") }
}

fun C.foo(i: Int) { println("extension") }
```

`C().foo(1)` 는 "extension"을 출력한다.

null 가능 리서버

확장이 null 가능 리시버 타입을 가질 수 있도록 정의할 수 있다. 이 확장은 객체 변수가 null인 경우에도 호출할 수 있고, 몸체 안에서 `this == null` 로 이를 검사할 수 있다. 이는 코틀린에서 null 검사 없이 toString()을 호출할 수 있도록 한다. 확장 함수 안에서 이 검사를 한다.

```
fun Any?.toString(): String {
    if (this == null) return "null"
    // null 검사 후에 'this'는 자동으로 non-null 타입으로 변환된다. 따라서 아래 toString()을 모든 클래스의
    // 멤버 함수로 처리한다.
    return toString()
}
```

확장 프로퍼티

함수와 유사하게, 코틀린은 확장 프로퍼티를 지원한다:

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

확장은 실제로 클래스에 멤버를 추가하지 않으므로, 지원 필드 를 가진 확장 프로퍼티를 위한 효율적인 방법이 없다. 이것이 **확장 프로퍼티에 대한 초기화** 를 허용하지 않는 이유이다. 이 기능은 명시적으로 getter/setter를 제공해서 정의할 수 있다.

예제:

```
val Foo.bar = 1 // 에러: 확장 프로퍼티에 대한 초기화는 허용하지 않는다.
```

컴페니언 오브젝트 확장

클래스에 컴페니언 오브젝트 가 있으면, 컴페니언 오브젝트를 위한 확장 함수와 프로퍼티를 정의할 수 있다:

```
class MyClass {
    companion object { }  // "Companion"으로 접근
}

fun MyClass.Companion.foo() {
    // ...
}
```

컴페니언 오브젝트의 일반 멤버처럼 클래스 이름만 한정자로 사용해서 호출할 수 있다:

```
MyClass.foo()
```

확장의 범위

대부분 패키지와 같은 최상위 수준에 확장을 정의한다.

```
package foo.bar

fun Baz.goo() { ... }
```

확장 함수를 선언한 패키지 밖에서 확장을 사용하려면 사용 위치에서 확장 함수를 임포트해야 한다:

```
package com.example.usage

import foo.bar.goo // "goo"의 모든 확장을 임포트
                   // 또는
import foo.bar.*   // "foo.bar"로부터 모든 것을 임포트

fun usage(baz: Baz) {
    baz.goo()
}
```

더 많은 정보는 임포트 를 참고한다.

멤버로 확장 선언하기

클래스 안에 다른 클래스를 위한 확장을 선언할 수 있다. 그런 확장안에서는 한정자 없이 접근할 수 있는 *암묵적인(implicit) 리시버* 객체 멤버가 존재한다. 확장 함수를 선언하고 있는 클래스의 인스턴스를 _디스패치 리시버(전달 리시버)_라고 부르며, 확장 메서드의 리시버 타입 인스턴스를 _확장 리시버_라고 부른다.

```
class D {
    fun bar() { ... }
}

class C {
    fun baz() { ... }

    fun D.foo() {
        bar()   // D.bar 호출
        baz()   // C.baz 호출
    }

    fun caller(d: D) {
        d.foo()   // 확장 함수를 호출
    }
}
```

디스패치 리시버와 확장 리시버의 멤버 간에 이름 충돌이 있는 경우 확장 리시버가 우선한다. 디스패치 리시버의 멤버를 참조하려면 한정한 `this` 구문 을 사용하면 된다.

```
class C {
    fun D.foo() {
        toString()         // D.toString() 호출
        this@C.toString()  // C.toString() 호출
    }
```

멤버로 선언한 확장을 `open` 으로 선언할 수 있고, 이를 하위 클래스에서 오버라이딩할 수 있다. 이는 디스패치 리시버 타입에 따라 확장 함수를 선택함을 의미한다. 하지만 확장 리시버 타입에 대해서는 정적이다.

```
open class D {
}

class D1 : D() {
}

open class C {
    open fun D.foo() {
        println("D.foo in C")
    }

    open fun D1.foo() {
        println("D1.foo in C")
    }

    fun caller(d: D) {
        d.foo()   // 확장 함수를 호출
    }
}

class C1 : C() {
    override fun D.foo() {
        println("D.foo in C1")
    }

    override fun D1.foo() {
        println("D1.foo in C1")
    }
}

C().caller(D())   // "D.foo in C" 출력
C1().caller(D())  // "D.foo in C1" 출력 - 디스패치 리시버를 동적으로 선택
C().caller(D1())  // "D.foo in C" 출력 - 확장 리시버를 정적으로 선택
```

동기

자바는 `FileUtils` , `StringUtils` 등 "*Utils"라는 이름을 가진 클래스를 사용한다. 유명한 `java.util.Collections` 도 같은 부류다. 이런 Utils 클래스에서 마음에 안 드는 점은 코드가 다음과 같은 모양을 갖는다는 것이다:

```
// 자바
Collections.swap(list, Collections.binarySearch(list, Collections.max(otherList)), Collections.max(list));
```

클래스 이름이 항상 나온다. 정적 임포트를 사용하면 다음과 같다:

```
// 자바
swap(list, binarySearch(list, max(otherList)), max(list));
```

조금 나아졌지만 IDE의 강력한 코드 완성 기능을 거의 사용할 수 없다. 다음과 같이 코딩할 수 있다면 훨씬 더 좋을 것이다:

```
// Java
list.swap(list.binarySearch(otherList.max()), list.max());
```

하지만 `List` 클래스에 모든 가능한 메서드를 구현할 수는 없다. 확장이 돕는 것이 바로 이 지점이다.

데이터 클래스

데이터를 보관하기 위한 목적으로 클래스를 자주 만든다. 그런 클래스는 종종 데이터에서 표준 기능이나 유틸리티 함수를 기계적으로 생성한다. 코틀린에서 이를 _데이터 클래스_라고 부르며 `data` 로 표시한다:

```
data class User(val name: String, val age: Int)
```

컴파일러는 주요 생성자에 선언한 모든 프로퍼티로부터 다음 멤버를 생성한다:

- `equals()` / `hashCode()` 쌍
- `"User(name=John, age=42)"` 형식의 `toString()`
- 선언 순서대로 프로퍼티에 대응하는 componentN() 함수
- `copy()` 함수 (아래 참고).

생성한 코드가 일관성있고 의미있는 기능을 제공하기 위해 데이터 클래스는 다음 조건을 충족해야 한다:

- 주요 생성자는 최소 한 개의 파라미터가 필요하다.
- 모든 주요 생성자 파라미터는 `val` 이나 `var` 여야 한다.
- 데이터 클래스는 추상, open, sealed, 내부 클래스일 수 없다.
- (1.1 이전에) 데이터 클래스는 오직 인터페이스만 구현해야 한다.

추가로 멤버 생성은 멤버 상속에 대해 다음 규칙을 따른다:

- 데이터 클래스 몸체에 `equals()` , `hashCode()` , `toString()` 구현이 있거나 또는
- 상위클래스에 `final` 구현이 있으면, 이 함수를 생성하지 않고 존재하는 구현을 사용한다.
- 상위타입에 `open` 이고 호환하는 타입을 리턴하는 `componentN()` 함수가 있다면 데이터 클래스는 오버라이딩하는 대응하는 함수를 생성한다. 상위타입 함수의 시그너처가 호환되지 않거나 final이어서 오버라이딩할 수 없는 경우 에러를 발생한다.
- `componentN()` 과 `copy()` 함수를 직접 구현하는 것은 허용하지 않는다.

1.1부터, 데이터 클래스가 다른 클래스를 상속할 수 있다(실드 클래스 에서 예를 볼 수 있다).

JVM에서 생성한 클래스에 파라미터 없는 생성자가 필요하면 모든 프로퍼티에 기본 값을 지정해야 한다(생성자 참고).

```
data class User(val name: String = "", val age: Int = 0)
```

복사

객체를 복사할 때 프로퍼티의 _일부_만 변경하고 나머지는 그대로 유지해야 할 때가 있다. 이것이 `copy()` 함수를 생성한 이유이다. 위 `User` 클래스의 경우 구현이 다음과 같다:

```
fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
```

이를 통해 다음과 같이 코드를 작성할 수 있다:

```
val jack = User(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

데이터 클래스와 분리 선언

데이터 클래스에 생성되는 _컴포넌트 함수_는 분리 선언 에 데이터 클래스를 사용할 수 있게 한다.

```
val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age") // "Jane, 35 years of age" 출력
```

표준 데이터 클래스

표준 라이브러리는 `Pair` 와 `Triple` 을 제공한다. 많은 경우 이름을 가진 데이터 클래스를 사용하는 것이 더 나은 설계 선택이다. 왜냐면 프러피티에 의미있는 이름을 사용해서 코드 가독성을 높여주기 때문이다.

실드 클래스

값이 제한된 타입 집합 중 하나를 가질 수 있지만 다른 타입은 가질 수 없을 때, 클래스 계층의 제한을 표현하기 위해 실드(Sealed) 클래스를 사용한다. 어떤 의미에서 실드 클래스는 enum 클래스를 확장한 것이다 enum 타입도 값 집합이 제한되어 있지만, 각 enum 상수는 단지 한 개 인스턴스로 존재하는 반면에 실드 클래스의 하위 클래스는 상태를 가질 수 있는 여러 인스턴스가 존재할 수 있다.

실드 클래스를 선언하려면 클래스 이름 앞에 `sealed` 수식어를 붙인다. 실드 클래스는 하위 클래스를 가질 수 있지만 모든 하위클래스는 반드시 실드 클래스와 같은 파일에 선언해야 한다. (코틀린 1.1 이전에, 이 규칙은 더 엄격했었다. 실드 클래스 선언 내부에 클래스를 중첩해야 했다.)

```
sealed class Expr
data class Const(val number: Double) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()
object NotANumber : Expr()
```

(위 예는 코틀린 1.1에 추가된 기능을 사용한다. 데이터 클래스는 실드 클래스를 포함한 클래스를 확장할 수 있다.)

실드 클래스 자체는 추상 이므로, 바로 인스턴스화할 수 없으며 `abstract` 멤버를 가질 수 있다.

실드 클래스는 `private` 이 아닌 생성자를 허용하지 않는다(실드 클래스의 생성자는 기본적으로 `private` 이다).

실드 클래스의 하위 클래스를 확장하는 클래스는 (간접 상속) 어디든 위치할 수 있다. 반드시 같은 파일일 필요는 없다.

실드 클래스를 사용해서 얻게 되는 핵심 이점은 `when 식` 과 함께 사용할 때 드러난다. when 식이 모든 경우를 다루는 것을 확신할 수 있다면 `else` 절을 문장에 추가할 필요가 없다. 하지만 이는 `when` 을 문장이 아닌 식으로 사용하는 경우에만 동작한다.

```
fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // 모든 경우를 다루므로 `else` 절이 필요 없다
}
```

지네릭

자바와 마찬가지로, 코틀린 클래스는 타입 파라미터를 가질 수 있다:

```
class Box<T>(t: T) {
    var value = t
}
```

일반적으로 지네릭 클래스의 인스턴스를 생성할 때 타입 인자를 전달해야 한다:

```
val box: Box<Int> = Box<Int>(1)
```

하지만 파라미터를 추론할 수 있다면(예를 들어 생성자 인자나 다른 방법으로), 타입 인자를 생략할 수 있다.

```
val box = Box(1) // 1은 Int 타입이므로 컴파일러는 Box<Int>를 말한다는 것을 알아낼 수 있다
```

변성(Variance)

자바 타입 시스템에서 가장 다루기 힘든 것 중 하나가 와일드카드 타입이다(자바 지네릭 FAQ 참고), 코틀린에는 와일드카드 타입이 없다. 대신 선언 위치 변성(declaration-site variance)과 타입 프로젝션(type projections)을 제공한다.

첫째로 자바에 왜 그런 미스테리한 와일드카드가 필요한자 생각해보자. Effective Java , Item 28: *Use bounded wildcards to increase API flexibility* 에서 문제를 설명하고 있다. 먼저 자바의 지네릭 타입은 **무공변(invariant)** 이다. 이는 `List<String>` 이 `List<Object>` 의 하위타입이 **아님을** 의미한다. 왜 그럴까? 만약 리스트가 **무공변** 이면 자바 배열보다 나을 게 없다. 왜냐면 다음 코드가 컴파일되고 런타임에 익셉션이 발생하기 때문이다:

```
// 자바
List<String> strs = new ArrayList<String>();
List<Object> objs = strs; // !!! 뒤이어 오는 문제 원인이 여기에 있다. 자바는 이를 허용하지 않는다!
objs.add(1); // 여기서 Integer를 String 배열에 넣는다
String s = strs.get(0); // !!! ClassCastException: Integer를 String으로 변환할 수 없다
```

따라서 자바는 런타임 안정성을 보장하기 위해 이런 것을 막는다. 하지만 이는 일부 영향을 준다. `Collection` 인터페이스의 `addAll()` 메서드를 보자. 이 메서드의 시그너처는 어떨까? 직관적으로 다음과 같이 생각할 수 있다:

```
// 자바
interface Collection<E> ... {
  void addAll(Collection<E> items);
}
```

하지만 이는 다음과 같은 (완전히 안전한) 간단한 것도 할 수 없다:

```
// 자바
void copyAll(Collection<Object> to, Collection<String> from) {
  to.addAll(from); // !!! addAll의 실제 선언으로는 컴파일할 수 없다:
                   //       Collection<String>은 Collection<Object>의 하위 타입이 아니다
}
```

(자바에서 이를 힘겹게 배웠다. Effective Java , Item 25: *Prefer lists to arrays* 를 보자.)

이것이 `addlAll()` 의 실제 시그너처가 다음과 같은 이유이다:

```
// 자바
interface Collection<E> ... {
  void addAll(Collection<? extends E> items);
}
```

와일드카드 타입 인자 `? extends E` 는 이 메서드가 `E` 객체의 콜렉션이나 또는 `E` 자체만이 아닌 `E` 의 *하위 타입* 콜렉션을 허용한다는 것을 나타낸다. 이는 items에서 안전하게 `E` 를 **읽을* 수 있지만(이 콜렉션의 요소는 E의 하위 타입 인스턴스이다), 콜렉션에 **쓸 수는 없다** 는 것을 의미한다. 왜냐면 items가 `E` 의 어떤 하위 타입인지 알 수 없기 때문이다. 이런 제약의 대가로 `Collection<String>` 가 `Collection<? extends Object>` 의 하위 타입이 *되는* 것을 얻는다.

전문 용어로 **extends** -bound(**upper** bound)를 가진 와일드카드는 타입을 **공변(covariant)** 으로 만든다.

이 트릭이 어떻게 동작하는지 이해하기 위한 핵심은 다소 간단하다. 콜렉션에서 항목을 **가져** 올 수만 있다면 `String` 콜렉션을 사용하고 `Object` 로 읽어도 괜찮다. 역으로 콜렉션에 항목을 **넣을** 수만 있다면 `Object` 콜렉션에 `String` 을 넣어도 괜찮다. 자바에서 `List<Object>` 의 **상위타입** 이 `List<? super String>` 이다.

후자를 **반공변(contravariance)** 이라고 부른다. `List<? super String>` 타입 인자에 대해 String을 취하는 메서드를 호출할 수 있다(예, `add(String)` 이나 `set(int, String)` 을 호출할 수 있다). 반면에 List 에서 `T`를 리턴하는 무언가를 호출하면 `String`이 아닌 `Object`를 얻는다.

Joshua Bloch는 **읽기만** 할 수 있는 오브젝트를 **Producer** 라 부르고 **쓰기만* 할 수 있는 오브젝트를 **Consumer** 라고 부른다. 그는 "최대의 유연함을 위해, Producer 나 Consumer를 표현하는 입력 파라미터에 와일드카드를 사용하라"고 권하고 있으며 쉽게 기억할 수 있는 약자를 제시했다.

PECS는 Producer-Extends, Consumer-Super를 의미한다.

주의 : `List<? extends Foo>` 타입의 Producer 객체를 사용하면 `add()` 나 `set()` 을 호출할 수 없는데, 그렇다고 이 객체가 **불변** 인 것은 아니다. 예를 들어, 리스트의 모든 항목을 삭제하기 위해 `clear()` 를 호출하는 것은 막을 수 없다. 왜냐면 `clear()` 는 어떤 파라미터도 받지 않기 때문이다. 와일드카드가(또는 다른 타입의 공변성) 보장하는 것은 **타입 안정성** 뿐이다. 불변성은 완전히 다른 얘기다.

선언 위치 변성(Declaration-site variance)

지네릭 인터페이스 `Source<T>` 가 파라미터로 `T` 를 갖는 메서드가 없고 오직 `T` 를 리턴하는 메서드만 있다고 하자:

```
// Java
interface Source<T> {
  T nextT();
}
```

`Source<String>` 인스턴스에 대한 레퍼런스를 `Source<Object>` 타입 변수에 저장하는 것은 완전히 안전하다. Consumer 메서드 호출이 없기 때문이다. 하지만 자바는 이를 알지 못하기에 여전히 이를 금지한다:

```
// 자바
void demo(Source<String> strs) {
  Source<Object> objects = strs; // !!! 자바는 허용하지 않음
  // ...
}
```

이를 고치려면 `Source<? extends Object>` 타입의 객체를 선언해야 하는데 이는 의미가 없다. 왜냐면 전처럼 변수에 대해 동일 메서드를 호출할 수 있고 이득없이 타입만 더 복잡해지기 때문이다. 하지만 컴파일러는 이를 모른다.

코틀린에서는 이를 컴파일러에 알려주는 방법을 제공한다. 이를 **선언 위치 변성(declaration-site variance)** 이라고 부른다. Source의 **타입 파라미터** T에 대해 `Source<T>` 의 멤버에서 T를 **리턴** (제공)만 하고 소비하지 않는다는 것을 명시할 수 있다. 이를 위해 **out** 수식어를 제공한다:

```
abstract class Source<out T> {
    abstract fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs // 괜찮다. T가 out 파라미터이기 때문이다.
    // ...
}
```

일반적인 규칙은 다음과 같다. 클래스 `C` 의 타입 파라미터 `T` 를 **out** 으로 선언하면, `C` 의 멤버에서 **out** -위치에만 T가 위치할 수 있지만, 리턴에서 `C<Base>` 가 안전하게 `C<Derivied>` 의 상위타입이 될 수 있다.

전문 용어로 클래스 `C` 는 파라미터 `T` 에 **공변** 한다 또는 `T` 가 **공변** 타입 파라미터라고 말한다. `C` 를 `T` 의 **소비자** 가 아닌 `T` 의 **생산자** 로 볼 수 있다.

out 수식어를 **변성 애노테이션** 이라고 부른다. 이 애노테이션은 타입 파라미터 선언 위치에 제공하므로 **선언 위치 공변** 에 대해 말한다. 이는 타입을 사용할 때 와일드카드로 타입을 공변으로 만드는 자바의 **사용 위치 변성** 과 대조된다.

out 외에 코틀린은 추가 공변 애노테이션인 **in** 을 제공한다. 이는 타입 파라미터를 **반공변** 으로 만든다. 반공변 파라미터는 오직 소비만 할 수 있고 생산할 수는 없다. 반공변의 좋은 예가 `Comparable` 이다:

```
abstract class Comparable<in T> {
    abstract fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0은 Double 타입인데 이는 Number의 하위타입이다.
    // 따라서 Comparable<Double> 타입의 변수에 x를 할당할 수 있다.
    val y: Comparable<Double> = x // OK!
}
```

in 과 **out** 단어가 자명하다고 믿기에(이미 C#에서 꽤나 오랫동안 성공적으로 이 용어를 사용하고 있다), 앞서 언급한 약자 PECS는 더 이상 필요 없으며, 더 상위 목적으로 약어를 바꿔볼 수 있다:

실존주의 **변환: Consumer in, Producer out!** :-)

타입 프로젝트

사용 위치 변성(Use-site variance): 타입 프로젝션

타입 파라미터 T를 *out* 으로 선언하면 매우 편리하며, 사용 위치에서 하위타입에 대한 문제를 피할 수 있다. 하지만 어떤 클래스는 실제로 오직 `T` 만 리턴하도록 제약할 수 없다. 배열이 좋은 예이다:

```
class Array<T>(val size: Int) {
    fun get(index: Int): T { /* ... */ }
    fun set(index: Int, value: T) { /* ... */ }
}
```

이 클래스는 `T` 에 대해 공변이나 반공변일 수 없다. 이는 배열을 유연하지 못하게 만든다. 다음 함수를 보자:

```
fun copy(from: Array<Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices)
        to[i] = from[i]
}
```

이 함수는 한 배열에서 다른 배열로 항목을 복사한다. 실제로 적용해보자:

```
val ints: Array<Int> = arrayOf(1, 2, 3)
val any = Array<Any>(3) { "" }
copy(ints, any) // Error: (Array<Any>, Array<Any>)를 기대함
```

여기서 유사한 문제가 발생한다. `Array<T>` 는 `T` 에 대해 **무공변(invariant)** 이므로 `Array<Int>` 와 `Array<Any>` 는 각각 상대의 하위타입이 아니다. 왜? copy가 잘못된 것을 할 **수도** 있기 때문이다. 예를 들어, `from` 이 Int 배열인데 `from` 에 String을 **쓰는** 시도를 할 수 있으며 이는 나중에 `ClassCastException` 을 발생할 수 있다.

우리가 원하는 것은 `copy()` 가 잘못된 것을 하지 않도록 확신할 수 있는 것이다. 다음과 같이 `from` 에 **쓰는** 것을 막을 수 있다:

```
fun copy(from: Array<out Any>, to: Array<Any>) {
 // ...
}
```

여기서 일어난 일을 **타입 프로젝션(type projection)** 이라고 부른다. `from` 은 단순 배열이 아니고 **제한된(projected)** 배열이다. 이제 `from` 에서는 타입 파라미터 `T` 를 리턴하는 메서드만 호출할 수 있는데, 이 경우에는 `get()` 만 호출할 수 있음을 의미한다. 이것이 코틀린의 **사용 위치 변성** 에 대한 접근법이며, 이는 자바의 `Array<? extends Object>` 에 대응하지만 더 간단한 방법이다.

타입 프로젝션에 **in** 을 사용할 수도 있다:

```
fun fill(dest: Array<in String>, value: String) {
    // ...
}
```

`Array<in String>` 은 자바의 `Array<? super String>` 에 대응한다. 예를 들어, `fill()` 함수에 `Object` 배열이나 `CharSequence` 배열을 전달할 수 있다.

스타 프로젝션(Star-projections)

때때로 타입 인자에 대해 알지 못하지만 안전한 방법으로 타입 인자를 사용하고 싶을 때가 있다. 여기서 안전한 방법은 지네릭 타입의 프로젝션을 정의하는 것이다. 그 지네릭 타입의 모든 실제 인스턴스는 그 프로젝션의 하위타입이 된다.

코틀린은 이를 위해 **스타 프로젝션** 이라 불리는 구문을 제공한다:

- `Foo<out T>` 에서 `T` 는 상위 한계로 `TUpper` 를 가진 공변 타입 파라미터면, `Foo<*>` 는 `Foo<out TUpper>` 와 동일하다. 이는 `T` 를 알 수 없을 때 `Foo<*>` 에서 안전하게 `TUpper` 의 값을 *읽을* 수 있다는 것을 의미한다.
- `Foo<in T>` 에서 `T` 가 반공변 타입 파라미터면, `Foo<*>` 는 `Foo<in Nothing>` 과 동일하다. 이는 `T` 를 알 수 없을 때 안전하게 `Foo<*>` 에 *쓸* 수 없다는 것을 의미한다.
- `Foo<T>` 에서 `T` 가 상위 한계로 `TUpper` 를 가진 무공변 타입 파라미터면, `Foo<*>` 는 값을 읽는 것은 `Foo<out TUpper>` 와 동일하고 값을 쓰는 것은 `Foo<in Nothing>` 과 동일하다.

지네릭 타입이 타입 파라미터가 여러 개이면, 타입 파라미터별로 따로 프로젝션할 수 있다. 예를 들어, `interface Function<in T, out U>` 타입이 있을 때 다음의 스타-프로젝션을 만들 수 있다:

- `Function<*, String>` 은 `Function<in Nothing, String>` 을 의미한다.
- `Function<Int, *>` 은 `Function<Int, out Any?>` 를 의미한다.
- `Function<*, *>` 은 `Function<in Nothing, out Any?>` 를 의미한다.

주의 : 스타-프로젝션은 자바의 raw 타입과 거의 같지만 안전하다.

지네릭 함수

클래스뿐만 아니라 함수도 타입 파라미터를 가질 수 있다. 함수 이름 앞에 타입 파라미터를 위치시킨다:

```
fun <T> singletonList(item: T): List<T> {
    // ...
}

fun <T> T.basicToString() : String {  // 확장 함수
    // ...
}
```

지네릭 함수를 호출하려면 호출 위치에서 함수 이름 **뒤에** 타입 인자를 지정한다:

```
val l = singletonList<Int>(1)
```

지네릭 제한

주어진 타입 파라미터에 올 수 있는 모든 가능한 타입 집합은 **지네릭 제한** 에 제약을 받는다.

상위 한계

대부분 공통 타입의 제한은 자바의 *extends* 키워드에 해당하는 **상위 한계** 이다:

```
fun <T : Comparable<T>> sort(list: List<T>) {
    // ...
}
```

콜론 뒤에 지정한 타입이 **상위 한계** 이다. 오직 `Comparable<T>` 의 하위타입만 `T` 에 사용할 수 있다. 다음 예를 보자:

```
sort(listOf(1, 2, 3)) // OK. Int는 Comparable<Int>의 하위타입이다.
sort(listOf(HashMap<Int, String>())) // 에러: HashMap<Int, String>는
                                     // Comparable<HashMap<Int, String>>의 하위타입이 아니다.
```

기본 상위 한계는 (지정하지 않은 경우) `Any?` 이다. 홑화살괄호 안에는 한 개의 상위 한계만 지정할 수 있다. 한 개의 타입 파라미터에 한 개 이상의 상위 한계를 지정해야 하면 별도의 **where** -절을 사용해야 한다:

```
fun <T> cloneWhenGreater(list: List<T>, threshold: T): List<T>
    where T : Comparable,
          T : Cloneable {
  return list.filter { it > threshold }.map { it.clone() }
}
```

중첩 클래스와 내부 클래스

다른 클래스에 클래스를 중첩할 수 있다:

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

내부 클래스

`inner` 로 지정한 클래스는 외부 클래스의 멤버에 접근할 수 있다. 내부 클래스는 외부 클래스의 객체에 대한 레퍼런스를 갖는다:

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

내부 클래스에서 `this` 에 대한 모호함에 대한 것은 한정된 `this` 식 을 참고한다.

익명 내부 클래스

오브젝트 식 을 사용해서 익명 내부 클래스를 생성할 수 있다:

```
window.addMouseListener(object: MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // ...
    }

    override fun mouseEntered(e: MouseEvent) {
        // ...
    }
})
```

객체가 함수형 자바 인터페이스의 인스턴스라면(예를 들어 한 개의 추상 메서드를 가진 자바 인터페이스), 인터페이스 타입을 접두어로 갖는 람다 식을 사용해서 익명 내부 객체를 생성할 수 있다:

```
val listener = ActionListener { println("clicked") }
```

Enum 클래스

enum 클래스의 가장 기본적인 용도는 타입에 안전한 열거값을 구현하는 것이다:

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
```

각 enum 상수는 객체이며, 콤마로 구분한다.

초기화

각 enum 값은 enum 클래스의 인스턴스로 초기화할 수 있다:

```
enum class Color(val rgb: Int) {
        RED(0xFF0000),
        GREEN(0x00FF00),
        BLUE(0x0000FF)
}
```

익명 클래스

enum 상수는 자신만의 익명 클래스를 선언할 수 있다:

```
enum class ProtocolState {
    WAITING {
        override fun signal() = TALKING
    },

    TALKING {
        override fun signal() = WAITING
    };

    abstract fun signal(): ProtocolState
}
```

익명 클래스는 자신의 메서드를 가질 수 있고 또한 기반 메서드를 오버라이딩할 수 있다. enum 클래스가 멤버를 정의할 경우, 자바와 마찬가지로 enum 상수 정의와 멤버를 세미콜론으로 구분해야 한다.

enum 상수 사용

자바와 동일하게 코틀린의 enum 클래스는 정의한 enum 상수 목록을 구하거나 이름으로 enum 상수를 구하는 메서드를 제공한다. 이 메서드의 시그너처는 다음과 같다 (enum 클래스의 이름이 `EnumClass` 라고 가정):

```
EnumClass.valueOf(value: String): EnumClass
EnumClass.values(): Array<EnumClass>
```

`valueOf()` 메서드는 지정한 이름에 해당하는 enum 상수가 존재하지 않으면 `IllegalArgumentException` 을 발생한다.

코틀린 1.1부터 `enumValues<T>()` 와 `enumValueOf<T>()` 함수를 사용해서 지네릭 방식으로 enum 클래스에 있는 상수에 접근할 수 있다:

```
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    print(enumValues<T>().joinToString { it.name })
}

printAllValues<RGB>() // RED, GREEN, BLUE 출력
```

모든 enum 상수에는 enum 클래스에 선언한 상수의 이름과 위치를 제공하는 프로퍼티가 존재한다:

```
val name: String
val ordinal: Int
```

enum 상수는 또한 Comparable 인터페이스를 구현하고 있어서 enum 클래스에 정의된 순서를 기준으로 비교할 수 있다.

오브젝트 식과 선언

때때로 하위클래스를 새로 만들지 않고 특정 클래스를 약간 수정한 객체를 만들고 싶을 때가 있다. 자바에서는 *익명 내부 클래스* 를 사용해서 이런 경우를 처리한다. 코틀린은 *오브젝트 식* 과 *오브젝트 선언* 으로 이 개념을 일부 일반화했다.

오브젝트 식

특정 타입을 상속한 익명 클래스의 객체를 생성하려면 다음과 같이 작성한다:

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // ...
    }

    override fun mouseEntered(e: MouseEvent) {
        // ...
    }
})
```

상위타입이 생성자를 가지면, 알맞은 생성자 파라미터를 전달해야 한다. 상위타입이 여러 개면, 콜론 뒤에 콤마로 구분해서 지정한다:

```
open class A(x: Int) {
    public open val y: Int = x
}

interface B {...}

val ab: A = object : A(1), B {
    override val y = 15
}
```

만약 별도 상위타입 없이 "단지 객체"가 필요한거라면, 다음과 같이 단순히 생성할 수 있다:

```
fun foo() {
    val adHoc = object {
        var x: Int = 0
        var y: Int = 0
    }
    print(adHoc.x + adHoc.y)
}
```

익명 객체는 로컬과 private 선언에서만 타입으로 사용할 수 있다. public 함수의 리턴 타입이나 public 프로퍼티의 타입으로 익명 객체를 사용하면, 그 함수나 프로퍼티의 실제 타입은 익명 객체에 선언한 상위 타입이 된다. 상위 타입을 선언하지 않았다면 `Any` 가 된다. 익명 객체에 추가한 멤버는 접근하지 못한다.

```
class C {
    // private 함수이므로 리턴 타입은 익명 객체 타입이다
    private fun foo() = object {
        val x: String = "x"
    }

    // public 함수이므로 리턴 타입은 Any이다
    fun publicFoo() = object {
        val x: String = "x"
    }

    fun bar() {
        val x1 = foo().x        // 동작
        val x2 = publicFoo().x  // 에러: 레퍼런스'x'를 찾을 수 없음
    }
}
```

자바의 익명 내부 클래스와 마찬가지로, 오브젝트 식의 코드는 둘러싼 범위의 변수에 접근할 수 있다. (자바와 달리, final 변수로 제한되지 않는다.)

```
fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0

    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }

        override fun mouseEntered(e: MouseEvent) {
            enterCount++
        }
    })
    // ...
}
```

오브젝트 선언

싱글톤 은 매우 유용한 패턴이며, 코틀린은 싱글톤을 쉽게 선언할 수 있다(스칼라를 따라했다):

```
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ...
    }

    val allDataProviders: Collection<DataProvider>
        get() = // ...
}
```

이를 *오브젝트 선언*이라고 부른다. 오브젝트 선언에는 `object` 키워드 뒤에 이름이 위치한다. 변수 선언처럼, 오브젝트 선언은 식이 아니며 할당 문장의 오른쪽에 사용할 수 없다.

오브젝트를 참조하려면 이름을 직접 사용한다:

```
DataProviderManager.registerDataProvider(...)
```

오브젝트는 상위타입을 가질 수 있다:

```
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // ...
    }

    override fun mouseEntered(e: MouseEvent) {
        // ...
    }
}
```

주의 : 오브젝트 선언은 로컬일 수 없다(예, 함수 안에 바로 중첩할 수 없다). 하지만, 다른 오브젝트 선언이나 내부가 아닌 클래스에는 중첩할 수 있다.

컴패니언 오브젝트

클래스 내부의 오브젝트 선언은 `companion` 키워드를 붙일 수 있다:

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

컴패니언 오브젝트의 멤버는 클래스 이름을 한정자로 사용해서 호출할 수 있다:

```
val instance = MyClass.create()
```

컴페니언 오브젝트의 이름은 생략이 가능하며, 이 경우 이름으로 `Companion` 을 사용한다:

```
class MyClass {
    companion object {
    }
}

val x = MyClass.Companion
```

컴페니언 오브젝트의 멤버가 다른 언어의 정적 멤버처럼 보이겠지만, 런타임에 컴페니언 오브젝트는 실제 객체의 인스턴스 멤버이므로 인터페이스를 구현할 수 있다:

```
interface Factory<T> {
    fun create(): T
}


class MyClass {
    companion object : Factory<MyClass> {
        override fun create(): MyClass = MyClass()
    }
}
```

하지만 JVM에서 `@JvmStatic` 애노테이션을 사용하면 컴페니언 오브젝트의 멤버를 실제 정적 메서드와 필드로 생성한다. 더 자세한 내용은 자바 상호운용성 을 참고한다.

오브젝트 식과 선언의 의미 차이

오브젝트 식과 오브젝트 선언 사이에는 중요한 의미 차이가 있다:

— 오브젝트 식은 사용한 위치에서 **즉시** 초기화되고 실행된다.

— 오브젝트 선언은 처음 접근할 때까지 초기화를 **지연** 한다.

— 컴페니언 오브젝트는 대응하는 클래스를 로딩할 때 초기화된다. 이는 자바의 정적 초기화와 동일하다.

위임

클래스 위임

위임 패턴 은 구현 상속보다 좋은 대안임이 증명됐다. 코틀린은 중복 코드없는 위임 패턴을 지원한다. 아래 코드에서 `Derived` 클래스는 `Base` 인터페이스를 상속할 수 있으며, 모든 public 메서드를 지정한 객체로 위임한다:

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main(args: Array<String>) {
    val b = BaseImpl(10)
    Derived(b).print() // 10 출력
}
```

`Derived` 의 상위타입 목록에 있는 `by` 절은 `Derived` 의 객체 내부에 `b` 를 저장한다는 것을 의미한다. 컴파일러는 `Base` 의 모든 메서드를 `b` 로 전달하도록 `Derived` 를 생성한다.

override는 그대로 동작한다. `override` 가 존재하면 컴파일러는 위임 객체의 메서드 대신 `override` 구현을 사용한다. `Derived` 에 `override fun print() { print("abc") }` 를 추가하면 프로그램은 "10" 대신 "abc"를 출력한다.

위임 프로퍼티

필요할 때마다 수동으로 구현할 수 있지만 한 번만 구현하고 라이브러리에 추가하면 좋을 만한 공통 프로퍼티가 존재한다. 다음은 예이다.

— lazy 프로퍼티: 처음 접근할 때 값을 계산한다.

— observable 프로퍼티: 프로퍼티가 바뀔 때 리스너에 통지한다.

— 맵에 저장할 프로퍼티: 각 프로퍼티를 별도 필드에 저장하는 대신 맵에 저장한다.

이런 (그리고 다른) 경우를 다룰 수 있도록 코틀린은 _위임 프로퍼티_를 지원한다:

```
class Example {
    var p: String by Delegate()
}
```

구문은 `val/var <property name>: <Type> by <expression>` 이다. `by` 뒤의 식이 _대리객체(delegate)_로 프로퍼티에 대응하는 `get()` (그리고 `set()`)은 대리객체의 `getValue()` 와 `setValue()` 메서드에 위임한다.

프로퍼티 대리객체는 인터페이스를 구현하면 안 되며, `getValue()` 함수를 제공해야 한다(그리고 `var` 프로퍼티의 경우 `setValue()` 도 제공). 다음은 예이다:

```
class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return "$thisRef, thank you for delegating '${property.name}' to me!"
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
        println("$value has been assigned to '${property.name} in $thisRef.'")
    }
}
```

위 예제의 `p` 프로퍼티를 읽을 때, 위임한 `Delegate` 인스턴스의 `getValue()` 함수를 호출한다. `getValue()` 의 첫 번째 파라미터는 `p` 를 포함한 객체이고, 두 번째 파라미터는 `p` 자체에 대한 설명을 포함한다(예를 들어, 이름을 포함한다). 다음 예를 보자.

```
val e = Example()
println(e.p)
```

이 코드는 다음을 출력한다:

```
Example@33a17727, thank you for delegating ‘p’ to me!
```

비슷하게 `p` 에 할당할 때 `setValue()` 함수를 호출한다. 첫 번째와 두 번째 파라미터는 동일하고, 세 번째 파라미터는 할당할 값이다:

```
e.p = "NEW"
```

이 코드는 다음을 출력한다.

```
NEW has been assigned to ‘p’ in Example@33a17727.
```

대리 객체에 대한 요구사항 명세는 아래 에서 볼 수 있다.

코틀린 1.1부터 함수나 코드 블록에 위임 프로퍼티를 선언할 수 있으므로 반드시 클래스의 멤버로 선언할 필요는 없다. 아래에서 예제 를 볼 수 있다.

표준 위임

코틀린 표준 라이브러리는 여러 유용한 위임을 위한 팩토리 메서드를 제공한다.

Lazy

`lazy()` 는 람다를 파라미터로 받고 `Lazy<T>` 인스턴스를 리턴하는 함수이다. 이는 lazy 프로퍼티를 구현하기 위한 대리객체로 동작한다. 이 객체는 `get()` 을 처음 호출할 때 `lazy()` 에 전달한 람다를 실행하고 그 결과를 기억한다. 이어진 `get()` 호출은 단순히 기억한 결과를 리턴한다.

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}

fun main(args: Array<String>) {
    println(lazyValue)
    println(lazyValue)
}
```

이 예는 다음을 출력한다:

```
computed!
Hello
Hello
```

기본적으로 lazy 프로퍼티의 평가를 **동기화** 한다. 한 쓰레드에서만 값을 계산하고, 모든 쓰레드는 같은 값을 사용한다. 위임 초기화에 동기화가 필요없으면 `lazy()` 함수에 파라미터로 `LazyThreadSafetyMode.PUBLICATION` 을 전달해서 동시에 여러 쓰레드가 초기화를 실행할 수 있게 허용할 수 있다. 단일 쓰레드가 초기화를 할 거라 확신할 수 없다면 `LazyThreadSafetyMode.NONE` 모드를 사용한다. 이는 쓰레드 안정성을 보장하지 않으며 관련 부하를 발생하지 않는다.

Observable

`Delegates.observable()` 은 두 개의 인자를 갖는다. 첫 번째는 초기 값이고 두 번째는 수정에 대한 핸들러이다. 프로퍼티에 값을 할당할 때마다 (할당이 끝난 *이후에*) 핸들러를 호출한다. 핸들러는 할당된 프로퍼티, 이전 값, 새 값의 세 파라미터를 갖는다:

```
import kotlin.properties.Delegates

class User {
    var name: String by Delegates.observable("<no name>") {
        prop, old, new ->
        println("$old -> $new")
    }
}

fun main(args: Array<String>) {
    val user = User()
    user.name = "first"
    user.name = "second"
}
```

이 예는 다음을 출력한다:

```
<no name> -> first
first -> second
```

만약 할당을 가로채서 그것을 "거부"하고 싶다면, `observable()` 대신 `vetoable()` 을 사용한다. 프로퍼티에 새 값을 할당하기 *전에* `vetoable` 에 전달한 핸들러를 호출한다.

맵에 프로퍼티 저장하기

위임 프로퍼티의 공통된 용도는 맵에 프로퍼티 값을 저장하는 것이다. JSON을 파싱하거나 다른 "동적인" 것을 하는 어플리케이션에 주로 사용한다. 이 경우 위임 프로퍼티를 위한 대리 객체로 맵 인스턴스 자체를 사용할 수 있다.

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}
```

이 예제에서 생성자는 맵을 받는다:

```
val user = User(mapOf(
    "name" to "John Doe",
    "age"  to 25
))
```

위임 프로퍼티는 이 맵에서 값을 구한다(프로퍼티의 이름을 키로 사용):

```
println(user.name) // "John Doe"를 출력
println(user.age)  // 25를 출력
```

읽기 전용 `Map` 대신에 `MutableMap` 을 사용하면 `var` 프로퍼티에 동작한다:

```
class MutableUser(val map: MutableMap<String, Any?>) {
    var name: String by map
    var age: Int     by map
}
```

로컬 위임 프로퍼티 (1.1부터 지원)

로컬 변수를 위임 프로퍼티로 선언할 수 있다. 예를 들어, 로컬 변수를 lazy로 만들 수 있다:

```
fun example(computeFoo: () -> Foo) {
    val memoizedFoo by lazy(computeFoo)

    if (someCondition && memoizedFoo.isValid()) {
        memoizedFoo.doSomething()
    }
}
```

`memoizedFoo` 변수에 처음 접근할 때만 계산을 한다. 만약 `someCondition` 이 false면 `memoizedFoo` 변수를 계산하지 않는다.

프로퍼티 위임 요구사항

아래 대리객체에 대한 요구사항을 요약했다.

읽기 전용 프로퍼티의 경우(`val`), 위임 객체는 이름이 `getValue` 이고 다음 파라미터를 갖는 함수를 제공해야 한다:

— `thisRef` — _프로퍼티 소유자_와 같거나 또는 상위 타입이어야 한다(확장 프로퍼티의 경우 — 확장한 타입).

— `property` — `KProperty<*>` 타입 또는 그 상위타입이어야 한다.

이 함수는 프로퍼티와 같은 타입(또는 하위 타입)을 리턴해야 한다.

변경 가능 프로퍼티의 경우 (`var`), 위임 객체는 *추가로* 이름이 `setValue` 이고 다음 파라미터를 갖는 함수를 제공해야 한다:

— `thisRef` — `getValue()` 와 동일

— `property` — `getValue()` 와 동일

— 새 값 — 프로퍼티와 같은 타입 또는 상위타입이어야 한다.

`getValue()` 와 `setValue()` 함수는 위임 클래스의 멤버 함수나 확장 함수로도 제공할 수 있다. 원래 이 함수를 제공하지 않는 객체에 위임 프로퍼티가 필요한 경우 확장 함수가 편하다. 두 함수 모두 `operator` 키워드를 붙여야 한다.

위임 클래스는 요구한 `operator` 메서드를 포함하고 있는 `ReadOnlyProperty` 와 `ReadWriteProperty` 인터페이스 중 하나를 구현할 수 있다. 이 두 인터페이스는 코틀린 표준 라이브러리에 선언되어 있다:

```
interface ReadOnlyProperty<in R, out T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
}

interface ReadWriteProperty<in R, T> {
    operator fun getValue(thisRef: R, property: KProperty<*>): T
    operator fun setValue(thisRef: R, property: KProperty<*>, value: T)
}
```

변환 규칙

모든 위임 프로퍼티에 대해, 코틀린 컴파일러는 보조 프로퍼티를 생성하고 그 프로퍼티에 위임한다. 예를 들어, `prop` 프로퍼티에 대해 `prop$delegate` 라는 숨겨진 프로퍼티를 생성하고, 접근자 코드는 단순히 이 프로젝트에 위임한다:

```
class C {
    var prop: Type by MyDelegate()
}

// 이 코드는 컴파일러가 대신 생성한다
class C {
    private val prop$delegate = MyDelegate()
    var prop: Type
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::prop, value)
}
```

코틀린 컴파일러가 `prop` 에 대한 모든 필요한 정보를 인자로 제공한다. 첫 번째 인자 `this` 는 외부 클래스인 `C` 의 인스턴스를 참조하고, `this::prop` 는 `prop` 자체를 설명하는 `KProperty` 타입의 리플레션 객체이다.

코드에서 직접 객체에 묶인 호출가능 레퍼런스 를 참조하는 `this::prop` 구문은 코틀린 1.1부터 사용 가능하다.

위임객체 제공 (1.1부터)

`provideDelegate` 연산자를 정의하면, 위임 프로퍼티가 위임할 객체를 생성하는 로직을 확장할 수 있다. `by` 의 오른쪽에서 사용할 객체가 `provideDelegate` 를 멤버 함수나 확장 함수로 정의하면, 프로퍼티의 위임 대상 인스턴스를 생성할 때 그 함수를 호출한다.

`provideDelegate` 의 가능한 한 가지 용도는 프로퍼티의 getter나 setter뿐만 아니라 생성할 때 프로퍼티의 일관성을 검사하는 것이다.

예를 들어, 값을 연결하기 전에 프로퍼티 이름을 검사하고 싶다면, 다음 코드로 이를 처리할 수 있다:

```
class ResourceLoader<T>(id: ResourceID<T>) {
    operator fun provideDelegate(
            thisRef: MyUI,
            prop: KProperty<*>
    ): ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, prop.name)
        // 대리 객체 생성
    }

    private fun checkProperty(thisRef: MyUI, name: String) { ... }
}

fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { ... }

class MyUI {
    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}
```

`provideDelegate` 파라미터는 `getValue` 와 동일하다:

- `thisRef` — _프로퍼티 소유자_와 같거나 또는 상위 타입이어야 한다(확장 프로퍼티의 경우 — 확장한 타입).
- `property` — `KProperty<*>` 타입 또는 그 상위타입이어야 한다.

`MyUI` 인스턴스를 생성하는 동안 각 프로퍼티에 대해 `provideDelegate` 메서드를 호출하고 즉시 필요한 검증을 수행한다.

프로퍼티와 대리객체 사이의 연결을 가로채는 능력이 없는데, 같은 기능을 구현하려면 편한 방법은 아니지만 명시적으로 프로퍼티 이름을 전달해야 한다.

```
// "provideDelegate" 기능 없이 프로퍼티 이름을 검사
class MyUI {
    val image by bindResource(ResourceID.image_id, "image")
    val text by bindResource(ResourceID.text_id, "text")
}

fun <T> MyUI.bindResource(
        id: ResourceID<T>,
        propertyName: String
): ReadOnlyProperty<MyUI, T> {
   checkProperty(this, propertyName)
   // 대리객체를 생성한다
}
```

생성한 코드는 보조 프로퍼티인 `prop$delegate` 를 초기화하기 위해 `provideDelegate` 를 호출한다. `val prop: Type by MyDelegate()` 프로퍼티 선언을 위해 생성한 코드와 위에서 생성한 코드(`provideDelegate` 메서드가 없는 경우)를 비교하자.

```
class C {
    var prop: Type by MyDelegate()
}

// 'provideDelegate'가 사용가능할 때
// 컴파일러가 이 코드를 생성한다:
class C {
    // 추가 "delegate" 프로퍼티를 생성하기 위해 "provideDelegate"를 호출
    private val prop$delegate = MyDelegate().provideDelegate(this, this::prop)
    val prop: Type
        get() = prop$delegate.getValue(this, this::prop)
}
```

`provideDelegate` 메서드는 보조 프로퍼티의 생성에만 영향을 주고, getter나 setter를 위한 코드 생성에는 영향을 주지 않는다.

함수와 람다

함수

함수 선언

코틀린에서 함수는 `fun` 키워드를 사용해서 선언한다:

```
fun double(x: Int): Int {
    return 2*x
}
```

함수 사용

전통적은 방식으로 함수를 호출한다:

```
val result = double(2)
```

멤버 함수 호출은 점 부호를 사용한다:

```
Sample().foo() // Sample 클래스의 인스턴스를 생성하고 foo를 호출
```

파라미터

파스칼 표기법(*name* : *type*)을 사용해서 파라미터를 정의한다. 각 파라미터는 콤마로 구분하며 타입을 가져야 한다:

```
fun powerOf(number: Int, exponent: Int) {
...
}
```

기본 인자

함수 파라미터는 기본 값을 가질 수 있다. 이 경우 대응하는 인자를 생략하면 기본 값을 사용한다. 이는 다른 언어와 비교해 오버로딩 함수의 개수를 줄여준다.

```
fun read(b: Array<Byte>, off: Int = 0, len: Int = b.size) {
...
}
```

타입 뒤에 등호(**=**)와 값을 사용해서 기본 값을 정의한다.

오버라이딩 메서드는 항상 베이스 메서드와 같은 기본 파라미터 값을 사용한다. 기본 파라미터 값을 갖는 메서드를 오버라이딩 할때, 시그너처에서 기본 파라미터를 값을 생략해야 한다:

```
open class A {
    open fun foo(i: Int = 10) { ... }
}

class B : A() {
    override fun foo(i: Int) { ... }  // 기본 값을 허용하지 않음
}
```

기본 값을 갖는 파라미터가 기본 값이 없는 파라미터보다 앞에 위치하면, 이름 가진 인자 로 함수를 호출할 때에만 기본 값을 사용한다:

```
fun foo(bar: Int = 0, baz: Int) { /* ... */ }

foo(baz = 1) // 기본 값 bar = 0을 사용한다
```

하지만 함수 호출시 마지막 인자를 괄호 밖에 람다 로 전달하면, 기본 파라미터에 값을 전달하지 않는 것을 허용한다:

```
fun foo(bar: Int = 0, baz: Int = 1, qux: () -> Unit) { /* ... */ }

foo(1) { println("hello") } // 기본 값 baz = 1을 사용
foo { println("hello") }    // 기본 값 bar = 0와 baz = 1를 사용
```

이름 가진 인자(Named Argument)

함수를 호출할 때 함수 파라미터에 이름을 줄 수 있다. 이름을 가진 인자를 사용하면 함수에 파라미터 개수가 많거나 기본 값이 많을 때 편리하다.

다음과 같은 함수가 있다고 하자:

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             divideByCamelHumps: Boolean = false,
             wordSeparator: Char = ' ') {
...
}
```

기본 인자를 사용해서 다음과 같이 호출할 수 있다:

```
reformat(str)
```

하지만 기본 인자가 없다면 다음과 같이 호출해야 한다:

```
reformat(str, true, true, false, '_')
```

이름 가진 인자를 사용하면 다음과 같이 코드의 가독성을 높일 수 있다:

```
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
```

그리고 모든 인자가 필요한 것이 아니면 다음과 같이 작성할 수 있다:

```
reformat(str, wordSeparator = '_')
```

위치 기반 인자와 이름 가진 인자를 함께 사용해서 함수를 호출할 때, 모든 위치 기반 인자는 첫 번째 이름 가진 인자보다 앞에 위치해야 한다. 예를 들어, `f(1, y = 2)` 는 허용하지만 `f(x = 1, 2)` 는 허용하지 않는다.

펼침(spread) 연산자를 사용하면 이름 붙인 형식에 인자의 변수 이름(`vararg`)을 전달할 수 있다:

```
fun foo(vararg strings: String) { /* ... */ }

foo(strings = *arrayOf("a", "b", "c"))
foo(strings = "a") // 단일 값에는 필요 없다
```

자바 함수를 호출할 때에는 이름 가진 인자 구문을 사용할 수 없다. 왜냐면 자바 바이트코드가 함수 파라미터의 이름을 항상 유지하는 것은 아니기 때문이다.

Unit 리턴 함수

함수가 어떤 유용한 값도 리턴하지 않으면, 리턴 타입으로 `Unit` 을 사용한다. `Unit` 은 `Unit` 을 유일한 값으로 갖는 타입이다. 이 값을 명시적으로 리턴하면 안 된다:

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
    else
        println("Hi there!")
    // `return Unit` 또는  `return` 생략
}
```

리턴 타입으로 `Unit` 을 선언하는 것 또한 생략 가능하다. 위 코드는 다음과 동일하다:

```
fun printHello(name: String?) {
    ...
}
```

단일 식 함수

함수가 단일 식을 리턴할 때 중괄호를 생략하고 등호(**=**) 뒤에 몸체로 단일 식을 지정할 수 있다:

```
fun double(x: Int): Int = x * 2
```

컴파일러가 리턴 타입을 유추할 수 있으면 리턴 타입을 명시적으로 선언하는 것을 생략 할 수 있다.

```
fun double(x: Int) = x * 2
```

명시적 리턴 타입

블록 몸체를 갖는 함수는 `Unit` 을 리턴한다는 것을 의도하지 않는 이상 항상 리턴 타입을 명시적으로 지정해야 한다. Unit을 리턴하는 경우 생략 가능하다 . 코틀린은 블록 몸체를 가진 함수의 리턴 타입을 유추할 수 없다. 왜냐면 그런 함수가 복잡한 제어 흐름을 가지면 리턴 타입을 독자가 (그리고 때때로 컴파일러 조차도) 알 수 없기 때문이다.

가변 인자 (Varargs)

함수 파라미터에 (보통 마지막 파라미터) `vararg` 수식어를 붙일 수 있다:

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts is an Array
        result.add(t)
    return result
}
```

이는 함수에 가변 개수의 인자를 전달할 수 있도록 한다.

```
val list = asList(1, 2, 3)
```

`T` 타입의 `vararg` 파라미터는 함수에서 `T` 배열로 접근한다. 예를 들어, 위 코드에서 `ts` 변수는 `Array<out T>` 타입이다. 가변 파라미터 뒤의 파라미터 값은 이름 가진 인자 구문으로 전달할 수 있다. 또는 가변 파라미터 뒤의 파라미터가 함수 타입이면 람다를 괄호 밖으로 전달할 수 있다.

`vararg` 함수를 호출할 때 인자를 `asList(1, 2, 3)` 와 같이 한 개 씩 전달할 수 있다. 또는 이미 배열이 있다면 **펼침(spread)** 연산자(배열 앞에 `*` 사용)를 사용해서 배열을 함수의 `vararg` 인자로 전달할 수 있다:

```
val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)
```

중위 부호

다음 경우에 중위 부호를 사용해서 함수를 호출할 수 있다.

- 멤버 함수이거나 확장 함수 인 경우
- 파라미터가 한 개인 경우
- `infix` 키워드를 붙인 경우

```
// Int에 대한 확장 함수 정의
infix fun Int.shl(x: Int): Int {
...
}

// infix 부호를 사용한 확장 함수를 호출

1 shl 2

// 이는 다음과 같음

1.shl(2)
```

함수 범위

코틀린은 함수를 파일에서 최상위 수준으로 선언할 수 있다. 이는 자바, *C#*, 스칼라와 달리 함수를 포함할 클래스를 만들 필요가 없다는 것을 의미한다. 최상위 수준 함수뿐만 아니라 함수를 로컬로, 멤버 함수로, 확장 함수로 선언할 수 있다.

로컬 함수

코틀린은 다른 함수 안에 선언한 로컬 함수를 지원한다.

```
fun dfs(graph: Graph) {
    fun dfs(current: Vertex, visited: Set<Vertex>) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v, visited)
    }

    dfs(graph.vertices[0], HashSet())
}
```

로컬 함수는 외부 함수의 로컬 변수에 접근할 수 있으며(클로저), 위 경우에 *visited* 를 로컬 변수로 할 수 있다:

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0])
}
```

멤버 함수

멤버 함수는 클래스나 오브젝트에 선언된 함수이다:

```
class Sample() {
    fun foo() { print("Foo") }
}
```

점 부호를 사용해서 멤버 함수를 호출한다:

```
Sample().foo() // Sample 클래스의 인스턴스를 생성하고 foo를 호출
```

클래스와 멤버 오버라이딩에 대한 정보는 클래스 와 상속 을 참고한다.

지네릭 함수

함수는 지네릭 파라미터를 가질 수 있다. 지네릭 파라미터는 함수 이름 앞에 홑화살괄호를 사용해서 지정한다:

```
fun <T> singletonList(item: T): List<T> {
    // ...
}
```

지네릭 함수에 대한 정보는 지네릭 을 참고한다.

인라인 함수

인라인 함수는 여기 에서 설명한다.

확장 함수

확장 함수는 여기 에서 설명한다.

고차 함수와 람다

고차 함수와 람다는 여기 에서 설명한다.

꼬리 재귀 함수

코틀린은 꼬리 재귀 로 알려진 함수형 프로그래밍 방식을 지원한다. 이는 스택 오버플로우 위험 없이 루프 대신 재귀 함수로 알고리즘을 작성할 수 있도록 한다. 함수에 `tailrec` 수식어가 있고 요구 형식을 충족하면, 컴파일러는 재귀를 최적화해서 빠르고 효율적인 루프 기반 버전으로 바꾼다:

```
tailrec fun findFixPoint(x: Double = 1.0): Double
        = if (x == Math.cos(x)) x else findFixPoint(Math.cos(x))
```

이 코드는 코사인의 fixpoint를 계산한다. 이 코드는 Math.cos 1.0에서 시작해서 더 이상 값이 바뀌지 않을 때까지 반복해서 호출하며 결과로 0.7390851332151607을 생성한다. 결과 코드는 다음의 전통적인 방식과 동일하다:

```
private fun findFixPoint(): Double {
    var x = 1.0
    while (true) {
        val y = Math.cos(x)
        if (x == y) return y
        x = y
    }
}
```

`tailrec` 수식어가 가능하려면, 함수는 수행하는 마지막 연산으로 자기 자신을 호출해야 한다. 재귀 호출 이후에 다른 코드가 있으면 꼬리 재귀를 사용할 수 없다. 그리고 try/catch/finally 블록 안에서 꼬리 재귀를 사용할 수 없다. 현재 꼬리 재귀는 JVM 백엔드에서만 지원한다.

고차 함수와 람다

고차 함수

고차 함수는 함수를 파라미터로 받거나 함수를 리턴하는 함수이다. 고차 함수의 좋은 예로 `lock()` 이 있다. 이 함수는 Lock 객체와 함수를 받아서, Lock을 얻고, 그 함수를 실행하고, Lock을 해제한다:

```
fun <T> lock(lock: Lock, body: () -> T): T {
    lock.lock()
    try {
        return body()
    }
    finally {
        lock.unlock()
    }
}
```

위 코드를 보자. `body` 는 함수 타입 인 `() -> T` 이다. 이 함수 타입은 파라미터가 없고 `T` 타입을 리턴하는 함수이다. `lock` 으로 보호하는 동안 `try` 블록에서 이 함수를 호출하고 그 결과를 `lock()` 함수의 결과로 리턴한다.

`lock()` 을 호출할 때 인자로 다른 함수를 전달할 수 있다(함수 레퍼런스 참고):

```
fun toBeSynchronized() = sharedResource.operation()

val result = lock(lock, ::toBeSynchronized)
```

보통 더 간편한 방법은 람다 식 을 전달하는 것이다:

```
val result = lock(lock, { sharedResource.operation() })
```

람다 식에 대한 자세한 내용은 아래에서 설명한다 . 하지만 이 절을 계속하기 위해 간단하게 람다 식의 개요를 정리했다.

- 람다 식은 항상 중괄호로 둘러 싼다.
- `->` 앞에 파라미터를 선언한다. (파라미터 타입은 생략할 수 있다.)
- `->` 뒤에 몸체가 온다(몸체가 존재할 때).

코틀린에서 함수의 마지막 파라미터가 함수면, 괄호 밖에서 람다 식을 인자로 전달할 수 있다.

```
lock (lock) {
    sharedResource.operation()
}
```

고차 함수의 다른 예는 `map()` 이다:

```
fun <T, R> List<T>.map(transform: (T) -> R): List<R> {
    val result = arrayListOf<R>()
    for (item in this)
        result.add(transform(item))
    return result
}
```

이 함수는 다음과 같이 부를 수 있다:

```
val doubled = ints.map { value -> value * 2 }
```

함수를 호출할 때 인자가 람다뿐이면 괄호를 완전히 생략할 수 있다.

`it` : 단일 파라미터의 암묵적 이름

다른 유용한 한 가지 규칙은 함수 리터럴의 파라미터가 한 개면, (`->` 를 포함한) 파라미터 선언을 생략할 수 있고, 파라미터 이름이 `it` 이 된다는 것이다:

```
ints.map { it * 2 }
```

이 규칙은 LINQ-방식 코드를 작성할 수 있게 한다:

```
strings.filter { it.length == 5 }.sortedBy { it }.map { it.toUpperCase() }
```

사용하지 않는 변수의 밑줄 표기 (1.1부터)

람다의 파라미터를 사용하지 않으면 이름 대신에 밑줄을 사용할 수 있다:

```
map.forEach { _, value -> println("$value!") }
```

람다에서의 분리 선언 (1.1부터)

람다에서의 분리 선언은 분리 선언 에서 설명한다.

인라인 함수

때때로 고차 함수의 성능을 향상시키기 위해 인라인 함수 를 사용하는 것이 이익이 된다.

람다 식과 익명 함수

람다 식 또는 익명 함수는 함수 선언 없이 바로 식으로 전달한 함수인 "함수 리터럴"이다. 다음 예를 보자:

```
max(strings, { a, b -> a.length < b.length })
```

`max` 함수는 고차 함수로서 두 번째 인자로 함수 값을 취한다. 두 번째 인자는 자체가 함수인 식으로 함수 리터럴이다. 다음 함수와 동등하다:

```
fun compare(a: String, b: String): Boolean = a.length < b.length
```

함수 타입

함수가 다른 함수를 파라미터로 받을 때, 파라미터를 위한 함수 타입을 지정해야 한다. 예를 들어, 앞서 언급한 `max` 함수는 다음과 같이 정의했다:

```
fun <T> max(collection: Collection<T>, less: (T, T) -> Boolean): T? {
    var max: T? = null
    for (it in collection)
        if (max == null || less(max, it))
            max = it
    return max
}
```

`less` 파라미터는 `(T, T) -> Boolean` 타입인데, 이 함수는 `T` 타입의 파라미터가 두 개이고 `Boolean` 을 리턴한다. `less` 는 첫 번째가 두 번째보다 작으면 true를 리턴한다.

위 코드의 4번째 줄에서 `less` 를 함수로 사용한다. `T` 타입의 두 인자를 전달해서 `less` 를 호출한다.

함수 타입을 위와 같이 작성하거나 또는 각 파라미터의 의미를 문서화하고 싶다면 파라미터에 이름을 붙일 수 있다.

```
val compare: (x: T, y: T) -> Int = ...
```

함수 타입을 null 가능 변수로 선언하려면 전체 함수 타입을 괄호로 감싸고 그 뒤에 물음표를 붙인다:

```
var sum: ((Int, Int) -> Int)? = null
```

람다 식 구문

람다 식(즉 함수 타입의 리터럴)의 완전한 구문 형식은 다음과 같다:

```
val sum = { x: Int, y: Int -> x + y }
```

람다 식은 항상 중괄호로 감싼다. 완전한 구문 형식에서 파라미터 선언은 중괄호 안에 위치하고 선택적으로 타입을 표시한다. 몸체는 `->` 부호 뒤에 온다. 람다의 추정한 리턴 타입이 `Unit` 이 아니면 람다 몸체의 마지막(또는 단일) 식을 리턴 값으로 처리한다.

모든 선택 사항을 생략하면 람다 식은 다음과 같이 보인다:

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

람다 식이 파라미터가 한 개인 경우는 흔하다. 코틀린이 시그너처 자체를 알아낼 수 있다면, 한 개인 파라미터를 선언하지 않는 것이 가능하며 `it` 이라는 이름으로 파라미터를 암묵적으로 선언한다:

```
ints.filter { it > 0 } // 이 리터럴은 '(it: Int) -> Boolean' 타입이다.
```

한정한 리턴 구문을 사용해서 람다에서 값을 명시적으로 리턴할 수 있다. 그렇지 않으면 마지막 식의 값을 암묵적으로 리턴한다. 따라서 다음의 두 코드는 동일하다:

```
ints.filter {
    val shouldFilter = it > 0
    shouldFilter
}

ints.filter {
    val shouldFilter = it > 0
    return@filter shouldFilter
}
```

함수가 다른 함수를 마지막 파라미터로 받으면 괄호로 둘러싼 인자 목록 밖에 람다 식 인자를 전달할 수 있음에 주목하자. callSuffix 에 대한 문법을 참고한다.

익명 함수

위에서 보여준 람다 식 구문에서 빼 먹은 한 가지는 함수의 리턴 타입을 지정할 수 있다는 점이다. 많은 경우, 리턴 타입을 자동으로 유추할 수 있기 때문에 이것이 불필요하다. 하지만, 명확하게 리턴 타입을 지정하고 싶다면 다른 구문인 _익명 함수_를 사용할 수 있다.

```
fun(x: Int, y: Int): Int = x + y
```

익명 함수는 일반 함수 선언과 비슷해 보인다. 차이점은 이름을 생략했다는 점이다. 몸체는 (위에서 보이는) 식이거나 블록일 수 있다:

```
fun(x: Int, y: Int): Int {
    return x + y
}
```

파라미터와 리턴 타입은 일반 함수와 같은 방법으로 지정한다. 문맥에서 파라미터 타입을 유추할 수 있다면 생략할 수 있다는 점은 다르다.

```
ints.filter(fun(item) = item > 0)
```

익명 함수에 대한 리턴 타입 추론은 보통 함수와 동일하게 동작한다. 몸체가 식인 익명 함수에 대한 리턴 타입은 자동으로 유추한다. 블록 몸체를 가진 익명 함수의 경우 명시적으로 리턴 타입을 지정해야 한다(아니면 `Unit` 으로 가정).

익명 함수 파라미터는 항상 괄호 안에 전달해야 한다. 괄호 밖에 함수 전달을 허용하는 간단 구문은 람다 식에 대해서만 동작한다.

람다 식과 익명 함수 간의 한 가지 다른 차이점은 비로컬 리턴 의 동작 방식이다. 라벨이 없는 `return` 문장은 항상 `fun` 키워드로 선언한 함수에서 리턴한다. 이는 람다 식의 `return` 은 둘러싼 함수를 리턴하는 반면 익명 함수의 `return` 은 익명 함수 자체로부터 리턴한다는 것을 의미한다.

클로저

람다 식 또는 익명 함수(그리고 로컬 함수 와 오브젝트 식)은 그것의 _클로저_에, 즉 외부 범위에 선언된 변수에 접근할 수 있다. 자바와 달리 클로저에 캡처한 변수를 수정할 수 있다:

```
var sum = 0
ints.filter { it > 0 }.forEach {
    sum += it
}
print(sum)
```

리시버를 가진 함수 리터럴

코틀린은 _리시버 객체_를 가진 함수 리터럴을 호출하는 기능을 제공한다. 함수 리터럴 몸체 안에서 별도 한정자 없이 리시버 객체의 메서드를 호출할 수 있다. 이는 함수 몸체 안에서 리서버 객체의 멤버에 접근할 수 있는 확장 함수와 유사하다. 이의 중요한 예제 중 하나는 타입에 안전한 그루비 방식 빌더 이다.

이런 함수 리터럴의 타입은 리시버를 가진 함수 타입이다:

```
sum : Int.(other: Int) -> Int
```

마치 리시버 객체에 메서드가 존재하는 것처럼 함수 리터럴을 호출할 수 있다:

```
1.sum(2)
```

익명 함수 구문은 함수 리터럴의 리시버 타입을 직접 지정할 수 있게 한다. 이는 리시버를 사용한 함수 타입의 변수를 선언하고 나중에 사용해야 할 때 유용하다.

```
val sum = fun Int.(other: Int): Int = this + other
```

리시버 타입을 갖는 함수의 비-리터럴 값을 할당하거나 추가로 리시버 타입의 *첫 번째* 파라미터를 가진 걸로 기대하는 일반 함수의 인자로 전달할 수 있다. 예를 들어, `String.(Int) -> Boolean` 과 `(String, Int) -> Boolean` 은 호환된다:

```
val represents: String.(Int) -> Boolean = { other -> toIntOrNull() == other }
println("123".represents(123)) // true

fun testOperation(op: (String, Int) -> Boolean, a: String, b: Int, c: Boolean) =
    assert(op(a, b) == c)

testOperation(represents, "100", 100, true) // OK
```

문맥에서 리시버 타입을 추론할 수 있는 경우 람다 식을 리시버를 가진 함수 리터럴로 사용할 수 있다.

```
class HTML {
    fun body() { ... }
}

fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()  // 리시버 객체 생성
    html.init()        // 리시버 객체를 람다에 전달
    return html
}


html {       // 리시버로 시작하는 람다
    body()   // 리시버 객체에 대한 메서드 호출
}
```

인라인 함수

고차 함수를 사용하면 런타임에 일부 불이익이 발생한다. 각 함수는 객체이고 함수의 몸체에서 접근하는 변수인 클로저를 캡처한다. 메모리 할당(함수 객체와 클래스 둘에 대해)과 가상 호출로 런타임 오버헤드가 증가한다.

하지만 많은 경우에 람다 식을 인라인해서 이런 종류의 오버헤드를 없앨 수 있다. 아래 함수가 이런 상황의 좋은 예다. `lock()` 함수를 쉽게 호출 위치에 인라인할 수 있다. 다음 경우를 보자:

```
lock(l) { foo() }
```

파라미터를 위한 함수 객체를 만들고 호출을 생성하는 대신에 컴파일러는 다음 코드를 만든다:

```
l.lock()
try {
    foo()
}
finally {
    l.unlock()
}
```

이것이 우리가 앞에서 원한 것 아닌가?

컴파일러가 이렇게 하도록 만들려면, `lock()` 함수에 `inline` 수식어를 붙이면 된다:

```
inline fun lock<T>(lock: Lock, body: () -> T): T {
    // ...
}
```

`inline` 수식어는 함수 자체와 함수에 전달되는 람다에 영향을 준다. 모두 호출 위치에 인라인 된다.

인라인은 생성한 코드의 크기를 증가시킨다. 하지만 합리적인 방법으로 처리하면(예, 큰 함수의 인라인을 피함), 성능에서, 특히 루프의 "변형" 호출 위치에서, 큰 성과를 낼 수 있다.

인라인 안하기

인라인 함수에 전달되는 람다 중 일부만 인라인되길 원할 경우, 함수 파라미터에 `noinline` 수식어를 붙일 수 있다:

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) {
    // ...
}
```

인라인 가능한 람다는 인라인 함수 안에서만 호출할 수 있고 또는 인라인 가능한 인자로만 전달할 수 있지만, `noinline` 은 필드에 저장하거나 다른 곳에 전달하는 등 원하는 방식으로 다룰 수 있다.

만약 인라인 함수가 인라인 가능한 함수 파라미터를 갖고 있지 않고 구체화한 타입 파라미터를 갖지 않으면, 컴파일러는 경고를 발생한다. 왜냐면 그런 함수를 인라인 하는 것은 이득이 없기 때문이다(인라인이 확실히 필요할 경우 `@Suppress("NOTHING_TO_INLINE")` 애노테이션을 사용해서 경고를 감출 수 있다).

비-로컬 리턴

코틀린에서 이름을 가진 함수나 익명 함수에서 나가려면 일반적인 한정하지 않은 `return` 을 사용해야 한다. 이는 람다를 나가려면 라벨을 사용해야 한다는 것을 의미한다. 람다에서의 순수 `return` 은 금지하는데 람다가 둘러싼 함수를 리턴할 수 없기 때문이다:

```
fun foo() {
    ordinaryFunction {
        return // 에러: `foo`를 리턴할 수 없다
    }
}
```

만약 람다를 전달한 함수가 인라인되면, 리턴도 인라인 되기 때문에 다음이 가능하다:

```
fun foo() {
    inlineFunction {
        return // OK: 람다도 인라인됨
    }
}
```

이런 리턴(람다에 위치하지만 둘러싼 함수에서 나가는)을 *비-로컬(non-local)* 리턴이라고 부른다. 종종 인라인 함수를 둘러싼 루프에서 이런 종류의 구성을 사용한다:

```
fun hasZeros(ints: List<Int>): Boolean {
    ints.forEach {
        if (it == 0) return true // hasZeros에서 리턴한다
    }
    return false
}
```

일부 인라임 함수는 파라미터로 그 함수에 전달한 람다를 함수 몸체가 아닌 (로컬 객체나 중첩 함수 같은) 다른 실행 컨텍스트에서 호출할 수 있다. 그런 경우 람다에서 비-로컬 흐름 제어를 허용하지 않는다. 이를 명시하려면 람다 파라미터에 `crossinline` 수식어를 붙여야 한다:

```
inline fun f(crossinline body: () -> Unit) {
    val f = object: Runnable {
        override fun run() = body()
    }
    // ...
}
```

인라인된 람다에서의 `break` 와 `continue` 는 아직 사용할 수 없지만, 향후 지원할 계획이다.

구체화한(Reified) 타입 파라미터

종종 파라미터로 전달한 타입에 접근해야 할 때가 있다:

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
    while (p != null && !clazz.isInstance(p)) {
        p = p.parent
    }
    @Suppress("UNCHECKED_CAST")
    return p as T?
}
```

이 코드는 트리를 탐색하고 노드가 특정 타입인지 검사하기 위해 리플렉션을 사용한다. 잘 동작하지만, 호출 위치가 그다지 이쁘지 않다:

```
treeNode.findParentOfType(MyTreeNode::class.java)
```

실제로 원하는 것은 아래 코드처럼 호출해서 함수에 타입을 전달하는 것이다:

```
treeNode.findParentOfType<MyTreeNode>()
```

이를 가능하게 하기 위해 인라인 함수는 *구체화한(reified) 타입 파라미터* 를 지원하며, 다음과 같은 코드를 작성할 수 있다:

```
inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p.parent
    }
    return p as T?
}
```

`reified` 수식어로 타입 파라미터를 한정하면, 함수 안에서 마치 일반 클래스처럼 타입 파라미터에 접근할 수 있다. 함수를 인라인하기 때문에 리플렉션이 필요없고 `!is` 나 `as` 와 같은 보통의 연산자가 동작한다. 또한 위에서 보여준 `myTree.findParentOfType<MyTreeNodeType>()` 처럼 그것을 호출할 수 있다:

많은 경우에 리플렉션이 필요없지만, 구체화한(reified) 타입 파라미터에 대해 리플렉션을 사용할 수 있다:

```
inline fun <reified T> membersOf() = T::class.members

fun main(s: Array<String>) {
    println(membersOf<StringBuilder>().joinToString("\n"))
}
```

일반 함수(inline을 붙이지 않은)는 구체화한(reified) 파라미터를 가질 수 없다. 런타임 표현을 갖지 않는 타입(예, 구체화한 타입 파라미터가 아니거나 `Nothing` 과 같은 가상 타입)은 구체화한(reified) 타입 파라미터를 위한 인자로 사용할 수 없다.

저수준 설명은 스펙 문서 에서 볼 수 있다.

인라인 프로퍼티 (1.1부터)

지원 필드가 없는 프로퍼티 접근자에 `inline` 수식어를 사용할 수 있다. 개별 프로퍼티 접근자에 사용한다:

```
val foo: Foo
    inline get() = Foo()

var bar: Bar
    get() = ...
    inline set(v) { ... }
```

또한 전체 프로퍼티에 붙여서 두 접근자를 모두 인라인으로 설정할 수 있다:

```
inline var bar: Bar
    get() = ...
    set(v) { ... }
```

호출 위치에서 인라인 접근자는 일반 인라인 함수처럼 인라인된다.

공개 API 인라인 함수에 대한 제약사항

`public` 이나 `protected` 인 인라인 함수가 `private` 또는 `internal` 선언에 포함되어 있지 않으면 모듈 의 공개 API로 간주한다. 다른 모듈에서 호출할 수 있으며 이 때에도 호출 위치에 인라인한다.

이는 인라인 함수를 선언한 모듈을 변경한 뒤에 호출하는 모듈을 재컴파일하지 않아서 바이너리 호환성이 깨지는 문제를 발생할 위험이 있다.

이렇게 모듈의 **비** -공개 API를 변경해서 발생하는 비호환성 위험을 제거하기 위해 공개 API 인라인 함수가 비-공개 API 선언(`private` 이나 `internal` 선언과 그 선언의 몸체에 있는 부분)을 사용하는 것을 허용하지 않는다.

`internal` 선언에 `@PublishedApi` 애노테이션을 붙이면 공개 API 인라인 함수에서 그것을 사용할 수 있다. `internal` 인라인 함수에 `@PublishedApi` 를 붙이면, 그것의 몸체 또한 public인 것처럼 검사한다.

코루틴

⚠ 코틀린 1.1에서 코루틴은 *실험* 버전이다. 자세한 내용은 아래 를 참고한다.

일부 API는 길게 실행되는 오퍼레이션을(네트워크 IO, 파일 IO, CPU나 GPU에 집중된 작업 등) 시작하고, 끝날 때까지 호출자가 멈출 것을 요구한다. 코루틴은 쓰레드 차단을 피하고 더 싸고 더 제어가능한 오퍼레이션으로 대체하는 방법을 제공한다. 그 오퍼레이션은 코루틴의 *서스펜션(suspension)* 이다.

코루틴은 비동기의 복잡함을 라이브러리에 넣어서 비동기 프로그래밍을 단순하게 한다. 코루틴을 사용하면 프로그램 로직을 *순차적으로* 표현할 수 있는데 하위 라이브러리는 이를 비동기로 해석한다.

라이브러리가 사용자 코드의 관련 부분을 콜백으로 감싸고, 관련 이벤트를 구독하고, 다른 쓰레드로 실행 스케줄을 잡지만(심지어 다른 머신에서), 코드는 마치 순차적으로 실행하는 것처럼 단순한 형태를 유지한다.

다른 언어에서 사용할 수 있는 다양한 비동기 기법을 코틀린의 코루틴을 이용해서 라이브러리로 구현할 수 있다. 이런 라이브러리에는 C#과 ECMAScript에서 온 `async / await` , Go에서 온 채널 과 `select` , C#과 파이썬에서 온 제너레이터/ `yield` 가 있다. 그런 구성 요소를 제공하는 라이브러리에 대한 설명은 아래 를 참고한다.

블로킹 vs 연기

기본적으로 코루틴은 *쓰레드 블로킹* 없이 *연기* 할 수 있는 계산이다. 쓰레드 블로킹은 비싸다. 특히 부하가 높은 상황에서는 상대적으로 작은 개수의 쓰레드만 실질적으로 사용되기 때문에 그 쓰레드 중 하나를 블록킹하면 일부 중요한 작업을 지연시키게 된다.

반면에 코루틴 서스펜션은 비용이 거의 없다. 컨텍스트 스위치나 OS와 관련된 다른 것이 필요없다. 그 기반하에 사용자 라이브러리는 높은 수준으로 서스펜션을 제어할 수 있다. 우리는 라이브러리 작성자로서 우리 요구에 따라 어떤 서스펜션이 발생하고 최적화/로그/가로채기할지 결정할 수 있다.

또 다른 차이점은 코루틴은 임의의 명령어를 연기할 수 없다는 것이다. 코루틴은 *서스펜션 포인트* 라고 불리는 곳만 연기할 수 있다. 서스펜션 포인트는 특별히 표시한 함수를 호출하는 지점이다.

서스펜딩 함수(suspending function)

서스펜션은 `suspend` 라는 특별한 수식어를 붙인 함수를 호출할 때 발생한다:

```
suspend fun doSomething(foo: Foo): Bar {
    ...
}
```

이런 함수를 *서스펜딩 함수(suspending function)* 라고 부르는데, 이 함수를 호출하면 코루틴을 연기하기 때문이다(라이브러리는 함수 호출에 대한 결과가 이미 사용가능하면, 서스펜션 없이 진행할지 여부를 결정할 수 있다). 서스펜딩 함수는 일반 함수와 같은 방법으로 파라미터와 리턴 값을 가질 수 있다. 하지만 서스펜딩 함수는 코루틴이나 다른 서스펜딩 함수에서만 호출할 수 있다. 사실 코루틴을 시작하려면, 적어도 한 개의 서스펜딩 함수가 있어야 하며 보통 익명 함수를 사용한다(즉, 서스펜딩 람다). 예제를 살펴보자. 다음은 단순화한 `async()` 함수이다(`kotlinx.coroutines` 라이브러리).

```
fun <T> async(block: suspend () -> T)
```

여기서 `async()` 는 일반 함수인데(서스펜딩 함수가 아님), `block` 파라미터는 `suspend` 수식어를 가진 `suspend () -> T` 함수 타입이다. 람다를 `async()` 에 전달하면, 그것은 *서스펜딩 람다* 가 되고, 그 람다에서 서스펜딩 함수를 호출할 수 있다:

```
async {
    doSomething(foo)
    ...
}
```

주의: 현재, 서스펜딩 함수 타입은 상위 타입으로 사용할 수 없고, 익명 서스펜딩 함수는 지원하지 않는다.

공통점을 계속 보면, `await()` 는 서스펜딩 함수일 수 있다(그래서 `aync {}` 블록 안에서 호출가능함), 이는 특정 계산이 끝나고 그 결과를 리턴할 때까지 코루틴을 연기한다:

```
async {
    ...
    val result = computation.await()
    ...
}
```

`async/await` 함수가 실제로 어떻게 동작하는지에 대한 내용은 여기 를 참고한다.

서스펜딩 함수인 `await()` 와 `doSomething()` 은 `main()` 과 같은 일반 함수에서 호출할 수 없다:

```
fun main(args: Array<String>) {
    doSomething() // 에러: 코루틴 컨텍스트가 아닌 곳에서 서스펜딩 함수를 호출
}
```

서스펜딩 함수는 가상(virtual)일 수 있고, 그 함수를 오버라이딩할 때는 `suspend` 수식어를 명시해야 한다:

```
interface Base {
    suspend fun foo()
}

class Derived: Base {
    override suspend fun foo() { ... }
}
```

`@RestrictsSuspension` 애노테이션

확장 함수(그리고 람다)도 일반 함수처럼 `suspend` 를 붙일 수 있다. 이는 사용자가 확장할 수 있는 다른 API와 DSL 의 생성을 가능하게 한다. 어떤 경우에 라이브러리 제작자는 사용자가 서스펜딩 코루틴의 *새로운 방법* 을 추가하는 것을 금지해야 할 때가 있다.

이럴 때 `@RestrictsSuspension` 애노테이션을 사용한다. 리시버 클래스나 인터페이스 `R` 에 이 애노테이션을 붙이면, 모든 서스펜딩 확장은 `R` 의 멤버나 `R` 에 대한 다른 확장으로 위임해야 한다. 확장이 무한하게 다른 확장에 위임할 수 없기 때문에(프로그램이 끝나지 않는다), 이는 모든 서스펜션이 `R` 멤버의 호출을 통해서 일어나는 것을 보장하며, 이를 통해 라이브러리 제작자는 완전히 제어할 수 있다.

이는 모든 서스펜션을 라이브러리에 종속된 방식으로 처리하는 상대적으로 *드문* 경우이다. 예를 들어 아래 에서 설명하는 `buildSequence()` 함수로 제너레이터를 구현할 때, 코루틴의 서스펜딩 호출이 다른 함수가 아닌 반드시 `yield()` 나 `yieldAll()` 로 끝나야 할 필요가 있다. 왜냐면 `SequenceBuilder` 는 `@RestrictsSuspension` 을 달았기 때문이다.

```
@RestrictsSuspension
public abstract class SequenceBuilder<in T> {
    ...
}
```

깃헙에서 소스를 참고하자.

코루틴의 내부 작동

여기서 코루틴이 내부적으로 어떻게 동작하는지 완벽하게 설명하진 않겠지만, 대략 어떤 식으로 진행되는지 감을 잡는 것은 매우 중요하다. 코루틴은 완전히 컴파일 기법으로 구현하며(VM이나 OS의 지원이 필요없다), 서스펜션은 코드 변환을 통해 작동한다. 기본적으로 모든 서스펜딩 함수는(최적화를 적용하지만 여기서는 이에 대해선 언급하지 않는다) 서스펜딩 호출에 대한 상태를 갖는 상태 머신으로 변환된다. 서스펜션 직전에 컴파일러가 생성한 클래스의 필드에 다음 상태와 관련된 로컬 변수를 함께 저장한다. 코루틴을 재개할 때, 로컬 변수를 복원하고 서스펜션 이후의 상태에서 상태 머신을 진행한다.

연기된 코루틴은 연기한 상태와 로컬 변수를 유지하는 객체에 전달해서 저장할 수 있다. 그런 객체의 타입이 `Continuation` 이며 여기서 설명한 전반적인 코드 변환은 고전적인 Continuation-passing style 에 해당한다. 결과적으로 서스펜딩 함수는 실제 구현에서 `Continuation` 의 추가 파라미터를 갖는다.

코루틴의 자세한 동작 방식은 설계 문서 를 참고한다.

(C#이나 ECMAScript 2016와 같은) 다른 언어의 async/await의 설명은 여기 설명과 관련이 있지만, 다른 언어가 구현한 언어 특징은 코틀린 코루틴만큼 일반적이지 않다.

코루틴의 실험 상태

코루틴 설계는 실험 단계 로 앞으로 바뀔 수 있다. 코틀린 1.1에서 코루틴을 컴파일할 때 기본적으로 *The feature "coroutines" is experimental* 이라는 경고를 출력한다. 이 경고를 없애고 싶으면 opt-in 플래그 를 지정하면 된다. 최종적으로 설계가 끝나고 실험 상태에서 승격되면 마지막 API를 `kotlin.coroutines` 로 옮기고, 하위 호환성을 위해 실험용 패키지는 (아마도 별도 아티팩트로 분리해서) 유지할 것이다.

중요 사항 : 라이브러리 제작자는 같은 규칙을 따를 것을 권한다. 코루틴 기반 API를 노출하는 패키지에는 "experimental"(예 `com.example.experimental`)을 추가해서 라이브러리의 바이너리 호환성을 유지하자. 최종 API를 릴리즈할 때 다음 과정을 따른다:

- 모든 API를 `com.example` 에 복사한다(experimental 접미사없이),
- 하위 호환성을 위해 experimental 패키지를 유지한다.

이는 라이브러리 사용자의 마이그레이션 이슈를 최소화할 것이다.

표준 API

코루틴에는 세 가지 주요 요소가 있다:

— 언어 지원(예, 앞에서 설명한 서스펜딩 함수)

— 코틀린 표준 라이브러리의 저수준 핵심 API

— 사용자 코드에서 직접 사용할 상위수준 API

저수준 API: `kotlin.coroutines`

저수준 API는 상대적으로 작고 고수준 라이브러리를 작성하는 것 외에 다른 용도로 사용하면 안 된다. 저수준 API는 두 개의 주요 패키지로 구성된다:

— `kotlin.coroutines.experimental` 는 다음과 같은 주요 타입과 기본 요소를 가짐

  — `createCoroutine()`

  — `startCoroutine()`

  — `suspendCoroutine()`

— `kotlin.coroutines.experimental.intrinsics suspendCoroutineOrReturn` 과 같은 저수준의 본질적인 요소 가짐

이들 API에 대한 자세한 사용법은 여기 를 참고한다.

`kotlin.coroutines` 의 제너레이터 API

`kotlin.coroutines.experimental` 의 유일한 "어플리케이션 수준" 함수는 다음과 같다:

— `buildSequence()`

— `buildIterator()`

이들 함수는 시퀀스와 관련이 있어서 `kotlin-stdlib` 에 들어가 있다. 사실 이들 함수는(여기서는 `buildSequence()` 로 제한할 수 있다) _제너레이터_를 구현해서 lazy 시퀀스를 싼 비용으로 구축할 수 있는 방법을 제공한다.

```
    val fibonacciSeq = buildSequence {
        var a = 0
        var b = 1

        yield(1)

        while (true) {
            yield(a + b)

            val tmp = a + b
            a = b
            b = tmp
        }
    }
```

이는 지연된, 잠재적으로 무한인 피보나치 시퀀스를 만든다. 이 시퀀스는 `yield()` 함수를 호출해서 연속된 피보나치 숫자를 생성하는 코루틴을 만든다. 그 시퀀스를 반복할 때 이터레이터의 각 단계마다 다음 숫자를 생성하는 코루틴의 다른 부분을 실행한다. 그래서 이 시퀀스로부터 유한한 숫자 목록을 취할 수 있다. 예를 들어 `fibonacciSeq.take(8).toList()` 의 결과는 `[1, 1, 2, 3, 5, 8, 13, 21]` 이 된다. 그리고 코루틴은 구현이 가벼워서 실제로 쓸모 있게 만든다.

이런 시퀀스가 실제로 지연되는지 보기 위해, `buildSequence()` 호출 안에 디버그 문구를 출력해보자:

```
    val lazySeq = buildSequence {
        print("START ")
        for (i in 1..5) {
            yield(i)
            print("STEP ")
        }
        print("END")
    }

    // 시퀀스의 처음 세 개 요소를 출력
    lazySeq.take(3).forEach { print("$it ") }
```

위 코드를 실행하면 처음 세 개의 요소를 출력한다. 생성 루프에서 `STEP` 과 숫자가 교차로 출력된다. 이는 계산이 실제로 지연됨을 의미한다. `1` 을 출력하기 위해 `START` 출력과 함께 첫 번째 `yield(i)` 까지만 실행한다. 그리고 `2` 를 출력하기 위해 다음 `yield(i)` 까지 진행하고 `STEP` 을 출력한다. `3` 도 같으며 다음 `STEP` 을(또한 `END` 를) 출력하지 않는다. 왜냐면 시퀀스의 다음 요소를 요청하지 않았기 때문이다.

한 번에 값의 콜렉션(또는 시퀀스)를 생성하려면, `yieldAll()` 함수를 사용한다:

```
    val lazySeq = buildSequence {
        yield(0)
        yieldAll(1..10)
    }

    lazySeq.forEach { print("$it ") }
```

`buildIterator()` 는 `buildSequence()` 와 비슷하게 동작하는데, lazy 이터레이터를 리턴한다.

`SequenceBuilder` 클래스(이는 위에서 설명한 `@RestrictsSuspension` 애노테이션을 가능하게 한다)에 대한 서스펜딩 확장을 작성하면 `buildSequence()` 에 커스텀 생성 로직을 추가할 수 있다.

```
suspend fun SequenceBuilder<Int>.yieldIfOdd(x: Int) {
    if (x % 2 != 0) yield(x)
}

val lazySeq = buildSequence {
    for (i in 1..10) yieldIfOdd(i)
}
```

다른 고수준 API: `kotlinx.coroutines`

오직 코루틴과 관련된 핵심 API는 코루틴 표준 라이브러리에서만 가능하다. 이는 거의 모든 코루틴 기반 라이브러리가 사용하는 핵심 프리미티브와 인터페이스로 구성된다.

코루틴에 기반한 대부분의 어플리케이션 수준 API는 별도 라이브러리인 `kotlinx.coroutines` 로 릴리즈된다. 이 라이브러리는 다음을 다룬다:

- `kotlinx-coroutines-core` 를 이용한 플랫폼에 무관한 비동기 프로그래밍
  - 이 모듈은 Go와 같은 채널을 포함하며 `select` 와 다른 편리한 프리미티브를 지원한다.
  - 여기 에서 이 라이브러리에 대한 종합적인 안내를 확인할 수 있다.
- JDK 8의 `CompletableFuture` 에 기반한 API: `kotlinx-coroutines-jdk8`
- 자바 7 또는 상위 버전의 API에 기반한 논블로킹 IO (NIO): `kotlinx-coroutines-nio`
- Swing 지원(`kotlinx-coroutines-swing`)과 JavaFx 지원(`kotlinx-coroutines-javafx`)
- RxJava 지원: `kotlinx-coroutines-rx`

이 라이브러리는 공통의 작업을 쉽게 만들어주는 편리한 API를 제공하고, 코루틴 기반 라이브러리를 작성하는 방법을 보여주는 완전한 예제도 제공한다.

여러 특징

분리 선언(Destructuring Declarations)

때때로 다음 예처럼 객체를 여러 변수에 분리해서 할당하는 것이 편리할 때가 있다.

```
val (name, age) = person
```

이런 구문을 _분리 선언(destructuring declaration)_이라고 부른다. 분리 선언은 한 번에 여러 변수를 생성한다. `name` 과 `age` 의 두 변수를 선언했고 각자 따로 사용할 수 있다:

```
println(name)
println(age)
```

분리 선언은 다음과 같은 코드로 컴파일된다:

```
val name = person.component1()
val age = person.component2()
```

`component1()` 과 `component2()` 함수는 코틀린에서 광범위하게 사용하는 *관례 규칙(principle of conventions)* 예다(`+` 와 `*` , `for` -루프 등의 연산자를 보자). 필요한 개수의 component 함수를 호출할 수만 있으면 무엇이든 분리 선언의 오른쪽에 위치할 수 있다. 물론 `component3()` 과 `component4()` 등이 존재할 수 있다.

`componentN()` 함수에 `operator` 키워드를 붙여야 분리 선언에서 그 함수를 사용할 수 있다.

분리 선언은 또한 다음과 같이 `for` 루프에서도 동작한다:

```
for ((a, b) in collection) { ... }
```

변수 `a` 와 `b` 는 콜렉션 요소의 `component1()` 과 `component2()` 함수가 리턴한 값을 구한다.

예제: 함수에서 두 값 리턴하기

함수에서 두 값을 리턴하고 싶다고 하자. 예를 들어 어떤 종류의 결과 객체와 상태를 리턴해야 한다고 가정하자. 코틀린에서 이를 간략하게 처리하는 방법은 *데이터 클래스* 를 선언하고 그 클래스의 인스턴스를 리턴하는 것이다:

```
data class Result(val result: Int, val status: Status)
fun function(...): Result {
    // computations

    return Result(result, status)
}

// 이제 이 함수를 사용한다
val (result, status) = function(...)
```

데이터 클래스는 자동으로 `componentN()` 함수를 선언하므로 여기서 분리 선언이 작동한다.

주의 : 위 `function()` 에서 표준 클래스 `Pair` 를 사용한 `Pair<Int, Status>` 를 리턴해도 된다. 하지만 데이터에 적당한 이름을 붙이는게 더 나을 때가 있다.

예제: 분리 선언과 맵

맵을 순회하는 가장 나은 방법은 아마도 다음일 것이다:

```
for ((key, value) in map) {
    // 키와 값으로 무엇을 함
}
```

이게 작동하려면 다음을 충족해야 한다

— 맵에서 값의 시퀀스로 제공하는 `iterator()` 함수를 제공한다.

— 각 요소를 `component1()` 과 `component2()` 함수를 제공하는 Pair로 제공한다.

그리고 사실 표준 라이브러리는 그런 확장을 제공한다:

```
operator fun <K, V> Map<K, V>.iterator(): Iterator<Map.Entry<K, V>> = entrySet().iterator()
operator fun <K, V> Map.Entry<K, V>.component1() = getKey()
operator fun <K, V> Map.Entry<K, V>.component2() = getValue()
```

따라서 맵을 사용한(또한 데이터 클래스 인스턴스의 콜렉션을 사용한) `for` 루프에서 분린 선언을 자유롭게 사용할 수 있다.

사용하지 않는 변수를 위한 밑줄 (1.1부터)

분리 선언에서 변수가 필요 없으면 이름 대신에 밑줄을 사용할 수 있다:

```
val (_, status) = getResult()
```

이 방식으로 생략한 component에 대해 `componentN()` 연산자 함수를 호출하지 않는다.

람다에서의 분리 선언 (1.1부터)

람다 파라미터에도 분린 선언 구문을 사용할 수 있다. 람다가 `Pair` 타입의 파라미터를 가지면(또는 `Map.Entry` , 또는 적당한 `componentN` 함수를 가진 다른 타입), 괄호 안에 Pair 타입 파라미터를 넣는 대신 분리한 새 파라미터를 사용할 수 있다:

```
map.mapValues { entry -> "${entry.value}!" }
map.mapValues { (key, value) -> "$value!" }
```

두 파라미터를 선언하는 것과 한 개 파라미터 대신에 분리한 쌍을 선언하는 것의 차이를 보자.

```
{ a -> ... } // 한 개 파라미터
{ a, b -> ... } // 두 개 파라미터
{ (a, b) -> ... } // 분리한 쌍
{ (a, b), c -> ... } // 분리한 쌍과 다른 파라미터
```

분리한 파라미터의 컴포넌트를 사용하지 않으면, 이름 대신에 밑줄로 대체할 수 있다:

```
map.mapValues { (_, value) -> "$value!" }
```

분리한 파라미터 전체 타입이나 일부 컴포넌트의 타입을 개별적으로 지정할 수 있다:

```
map.mapValues { (_, value): Map.Entry<Int, String> -> "$value!" }

map.mapValues { (_, value: String) -> "$value!" }
```

콜렉션: List, Set, Map

많은 언어와 달리 코틀린은 변경 가능한 콜렉션과 불변 콜렉션을 구분한다(리스트, 집합, 맵 등). 콜렉션을 수정할 수 있는지 정확하게 제어하는 것은 버그를 줄이고 좋은 API를 설계할 때 유용하다.

수정 가능한 콜렉션의 읽기 전용 _뷰_와 실제로 불변인 콜렉션의 차이는 미리 이해하는 것이 중요하다. 둘다 생성하기 쉽지만 타입 시스템은 이 차이를 표현하지 않는다. 따라서 (만약 관련이 있다면) 그것을 추적하는 것은 당신의 몫이다.

코틀린의 `List<out T>` 타입은 `size` , `get` 과 같은 읽기 전용 오퍼레이션을 제공하는 인터페이스이다. 자바처럼 `Iterable<T>` 를 상속한 `Collection<T>` 을 상속한다. 리스트를 수정하는 메서드는 `MutableList<T>` 에 있다. 이 패턴은 또한 `Set<out T>/MutableSet<T>` 과 `Map<K, out V>/MutableMap<K, V>` 에도 적용된다.

아래 코드는 리스트와 집합의 기본 사용법을 보여준다:

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
val readOnlyView: List<Int> = numbers
println(numbers)        // "[1, 2, 3]" 출력
numbers.add(4)
println(readOnlyView)   // prints "[1, 2, 3, 4]"
readOnlyView.clear()    // -> 컴파일 안 된다

val strings = hashSetOf("a", "b", "c", "c")
assert(strings.size == 3)
```

코틀린은 리스트나 집합을 생성하기 위한 전용 구문 구성 요소가 없다. `listOf()` , `mutableListOf()` , `setOf()` , `mutableSetOf()` 와 같은 표준 라이브러리에 있는 메서드를 사용해서 생성한다.

성능이 중요하지 않은 코드에서는 간단한 이디엄인 `mapOf(a to b, c to d)` 를 사용해서 맵을 생성할 수 있다.

`readOnlyView` 변수는 같은 리스트를 가리키고 하부 리스트를 변경할 때 바뀐다는 것에 주목하자. 만약 리스트에 존재하는 유일한 참조가 읽기 전용 타입이면, 콜렉션이 완전히 불변이라고 간주할 수 있다. 그런 콜렉션을 만드는 간단한 방법은 다음과 같다:

```
val items = listOf(1, 2, 3)
```

현재 `listOf` 메서드는 배열 리스트를 사용해서 구현하는데, 앞으로는 리스트가 바뀌지 않는다는 사실을 이용해서 메모리에 더 효율적인 완전히 불변인 콜렉션 타입을 리턴하도록 구현할 것이다.

읽기 전용 타입은 공변이다. 이는 Rectangle이 Shape를 상속한다고 가정할 때 `List<Rectangle>` 를 `List<Shape>` 에 할당할 수 있다는 것을 의미한다. 변경가능 콜렉션 타입에서는 이를 허용하지 않는다. 왜냐면 런타임에 실패가 발생할 수 있기 때문이다.

때때로 특정 시점의 콜렉션에 대해 변경되지 않는 것을 보장하는 스냅샷을 호출자에게 리턴하고 싶을 때가 있다:

```
class Controller {
    private val _items = mutableListOf<String>()
    val items: List<String> get() = _items.toList()
}
```

`toList` 확장 메서드는 단지 리스트 항목을 중복해서 갖는 리스트를 생성하며, 리턴한 리스트가 결코 바뀌지 않는다는 것을 보장한다.

리스트나 맵은 유용하게 쓸 수 있는 다양한 확장 메서드를 제공하는데, 이들 메서드에 익숙해지면 좋다:

```
val items = listOf(1, 2, 3, 4)
items.first() == 1
items.last() == 4
items.filter { it % 2 == 0 }   // [2, 4]를 리턴

val rwList = mutableListOf(1, 2, 3)
rwList.requireNoNulls()        // [1, 2, 3]을 리턴
if (rwList.none { it > 6 }) println("No items above 6")  // "No items above 6" 출력
val item = rwList.firstOrNull()
```

… 또한 sort, zip, fold, reduce 등 있었으면 하는 유틸리티도 제공한다.

맵은 다음 패턴을 따른다. 다음과 같이 쉽게 생성하고 접근할 수 있다:

```
val readWriteMap = hashMapOf("foo" to 1, "bar" to 2)
println(readWriteMap["foo"])  // "1"을 출력
val snapshot: Map<String, Int> = HashMap(readWriteMap)
```

범위

범위(range) 식은 연산자 형식이 `..` 인 `rangeTo` 함수로 구성한다. `..` 은 `in` 과 `!in` 을 보완한다. 모든 비교가능한 타입에 대해 범위를 정의할 수 있는데, 정수 기본 타입의 경우 구현을 최적화한다. 다음은 범위의 사용 예이다:

```
if (i in 1..10) { // 1 <= i && i <= 10과 동일
    println(i)
}
```

정수 타입 범위(`IntRange` , `LongRange` , `CharRange`)는 반복할 수 있는 추가 특징이 있다. 컴파일러는 추가 오버헤드없이 정수 타입 범위를 동일한 자바의 인덱스 기반 `for` -루프로 변환한다:

```
for (i in 1..4) print(i) // "1234" 출력

for (i in 4..1) print(i) // 아무것도 출력하지 않음
```

숫자를 역순으로 반복하려면? 간단하다. 표준 라이브리에 있는 `downTo()` 함수를 사용하면 된다:

```
for (i in 4 downTo 1) print(i) // "4321" 출력
```

1씩 간격이 아닌 다른 간격으로 숫자를 반복할 수 있나? 물론 가능하다. `step()` 을 사용하면 된다:

```
for (i in 1..4 step 2) print(i) // "13" 출력

for (i in 4 downTo 1 step 2) print(i) // "42" 출력
```

마지막 요소를 포함하지 않는 범위를 생성하려면 `until` 함수를 사용한다:

```
for (i in 1 until 10) { // i in [1, 10), 10은 제외
    println(i)
}
```

동작 방식

범위는 라이브러리의 공통 인터페이스인 `ClosedRange<T>` 를 구현한다.

`ClosedRange<T>` 는 , 비교가능한 타입에 정의한 것으로 수학적 의미로 닫힌 구간을 표시한다. 이는 두 개의 끝점인 `start` 와 `endInclusive` 을 갖는다. 이 두 점은 범위에 포함된다. 주요 연산자인 `contains` 는 `in` / `!in` 연산자에서 주로 사용한다.

정수 타입 Progression(`IntProgression` , `LongProgression` , `CharProgression`)은 산술적 증가를 표현한다. Progression은 `first` 요소, `last` , 0이 아닌 `step` 으로 정의한다. 첫 번째 요소는 `first` 이고 이어지는 요소는 앞 요소에 `step` 을 더한 값이 된다. `last` 요소는 Progression이 비어서 더 이상 없을 때까지 반복을 통해 도달한다.

Progression은 `Iterable<N>` 의 하위 타입으로 `N` 은 각각 `Int` , `Long` 또는 `Char` 이다. 따라서 Progression을 `for` 나 `map` , `fileter` 와 같은 함수에서 사용할 수 있다. `Progression` 에 대한 반복은 자바/자바스크립트의 인덱스 기반 `for` -루프와 동일하다:

```
for (int i = first; i != last; i += step) {
  // ...
}
```

정수 타입에서 `..` 연산자는 `ClosedRange<T>` 와 `*Progression` 를 구현한 객체를 생성한다. 예를 들어, `IntRange` 는 `ClosedRange<Int>` 를 구현했고, `IntProgression` 를 상속하므로 `IntProgression` 에 정의한 모든 오퍼레이션을 `IntRange` 에서 사용가능하다.. `downTo()` 와 `step()` 함수의 결과는 항상 `*Progression` 이다.

Progression의 컴페니언 오브젝트가 제공하는 `fromClosedRange` 함수를 사용해서 Progression을 생성한다:

```
IntProgression.fromClosedRange(start, end, step)
```

Progression의 `last` 요소는 양수 `step` 에 대해서는 `end` 값보다 크지 않은 최댓값을 계산해서 구하고 음수 `step` 에 대해서는 `end` 보다 작지 않은 최솟값을 계산해서 구한다. 계산 결과로 얻은 값은 `(last - first) % step == 0` 이 된다.

유틸리티 함수

`rangeTo()`

정수 타입의 `rangeTo()` 연산자는 단순히 `*Range` 생성자를 호출한다:

```
class Int {
    //...
    operator fun rangeTo(other: Long): LongRange = LongRange(this, other)
    //...
    operator fun rangeTo(other: Int): IntRange = IntRange(this, other)
    //...
}
```

실수 타입(`Double` , `Float`)은 `rangeTo` 연산자가 없고, 대신 범용 `Comparable` 를 위해 표준 라이브러리가 제공하는 연산자를 사용한다:

```
    public operator fun <T: Comparable<T>> T.rangeTo(that: T): ClosedRange<T>
```

이 함수가 리턴하는 범위는 반복에서 사용할 수 없다.

`downTo()`

모든 정수 타입 쌍에 대해 `downTo()` 확장 함수를 정의했다. 다음의 두 예를 보자:

```
fun Long.downTo(other: Int): LongProgression {
    return LongProgression.fromClosedRange(this, other.toLong(), -1L)
}

fun Byte.downTo(other: Int): IntProgression {
    return IntProgression.fromClosedRange(this.toInt(), other, -1)
}
```

`reversed()`

`reversed()` 확장 함수는 각 `*Progression` 클래스에 존재하며 역순의 Progression을 리턴한다:

```
fun IntProgression.reversed(): IntProgression {
    return IntProgression.fromClosedRange(last, first, -step)
}
```

`step()`

`*Progression` 클래스를 위한 `step()` 확장 함수를 제공한다, 이 함수는 `step` 값(함수 파라미터)을 갖는 Progression을 리턴한다. 증분 값은 항상 양수여야 하므로, 이 함수는 반복의 방향을 바꿀 수 없다:

```
fun IntProgression.step(step: Int): IntProgression {
    if (step <= 0) throw IllegalArgumentException("Step must be positive, was: $step")
    return IntProgression.fromClosedRange(first, last, if (this.step > 0) step else -step)
}

fun CharProgression.step(step: Int): CharProgression {
    if (step <= 0) throw IllegalArgumentException("Step must be positive, was: $step")
    return CharProgression.fromClosedRange(first, last, if (this.step > 0) step else -step)
}
```

`(last - first) % step == 0` 규칙을 지키기 때문에 리턴한 Progression의 `last` 값은 원래 Progression의 `last` 값과 다를 수 있다:

```
(1..12 step 2).last == 11  // progression with values [1, 3, 5, 7, 9, 11]
(1..12 step 3).last == 10  // progression with values [1, 4, 7, 10]
(1..12 step 4).last == 9   // progression with values [1, 5, 9]
```

타입 검사와 변환: 'is'와 'as'

`is` 와 `!is` 연산자

`is` 연산자를 사용하면 런타임에 객체가 특정 타입을 따르는지 검사할 수 있다. `!is` 연산자는 반대를 검사한다:

```
if (obj is String) {
    print(obj.length)
}

if (obj !is String) { // !(obj is String)와 동일
    print("Not a String")
}
else {
    print(obj.length)
}
```

스마트 타입 변환

코틀린에서는 많은 경우 명시적인 타입 변환 연산이 필요 없다. 왜냐면 컴파일러가 불변 값에 대해 `is` -검사와 명시적 타입 변환 을 추적해서 자동으로 필요한 곳에 (안전한) 타입 변환을 넣어주기 때문이다:

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x를 자동으로 String으로 타입 변환
    }
}
```

!is 검사를 통해 리턴할 경우 컴파일러는 타입 변환을 안전하게 할 수 있다는 것을 알고, 자동으로 타입 변환을 추가한다:

```
    if (x !is String) return
    print(x.length) // x를 자동으로 String으로 타입 변환
```

또한 `&&` 와 `||` 의 오른쪽에도 자동 타입 변환을 추가한다:

```
    // `||`의 오른쪽에서 x를 자동으로 String으로 타입 변환
    if (x !is String || x.length == 0) return

    // `&&`의 오른쪽에서 x를 자동으로 String으로 타입 변환
    if (x is String && x.length > 0) {
        print(x.length) // x를 자등으로 String으로 타입 변환
    }
```

이런 _스마트 타입 변환_은 `when` -식 과 `while` -루프 에서도 동작한다:

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.length + 1)
    is IntArray -> print(x.sum())
}
```

변수 검사와 사용 사이에 변수가 바뀌지 않는다는 것을 컴파일러가 확신할 수 없으면 스마트 타입 변환이 동작하지 않는다. 구체적으로 다음 규칙에 따라 스마트 타입 변환을 적용한다:

- `val` 로컬 변수 - 항상 적용
- `val` 프로퍼티 - 프로퍼티가 private이거나 internal이거나 프로퍼티를 선언한 같은 모듈에서 검사를 한 경우에 적용. 커스텀 getter를 가진 프로퍼티나 open 프로퍼티에는 스마트 변환이 적용되지 않음
- `var` 로컬 변수 - 변수 검사와 사용 사이에 수정이 없고 변수를 캡처한 람다가 변수를 수정하지 않으면 적용
- `var` 프로퍼티- 적용하지 않음 (왜냐면 다른 코드가 언제든지 변수를 수정할 수 있기 때문).

"불안전" 타입 변환 연산자

보통 타입 변환 연산자는 타입 변환이 불가능하면 익셉션을 발생한다. 따라서 이를 *불안전* 하다고 한다. 코틀린에서 불안전 타입 변환은 중위 연산자인 `as` 로 수행한다(연산자 우선순위 를 보자):

```
val x: String = y as String
```

x는 null이 가능 하지 않기 때문에 `null` 을 `String` 타입으로 변환할 수 없다. 예를 들어, `y` 가 null이면 위 코드는 익셉션을 던진다. 자바의 변환 세만틱에 맞추려면, 다음과 같이 변환 연산자의 오른쪽에 null 가능 타입이 와야 한다:

```
val x: String? = y as String?
```

"안전한" (nullable) 타입 변환 연산자

익셉션이 발생하는 것을 피하려면 *안전한* 타입 변환 연산자인 `as?` 를 사용할 수 있다. 이 연산자는 실패시 `null` 을 리턴한다.

```
val x: String? = y as? String
```

`as?` 의 오른쪽이 null이 아닌 `String` 타입이지만 타입 변환 결과는 null 가능 타입이다.

this 식

현재 _리시버_를 지정하려면 `this` 식을 사용한다.

— 클래스의 멤버에서 `this` 는 그 클래스의 현재 객체를 참조한다.

— 확장 함수나 리시버를 가진 함수 리터럴에서 `this` 는 점의 왼쪽 편에 전달한 *리시버* 파라미터를 지정한다.

`this` 에 한정자가 없으면 _가장 안쪽을 둘러썬 범위_를 참조한다. 다른 범위의 `this` 를 참조하려면 _라벨 한정자_를 사용한다.

한정된 `this`

외부 범위(클래스나 확장 함수, 또는 라벨이 있는 리시버를 가진 함수 리터럴)에서 `this` 에 접근하려면 `this@label` 형식을 사용한다. 여기서 `@label` 은 `this` 가 속한 범위를 의미하는 라벨이다:

```
class A { // 암묵적으로 @A 라벨 사용
    inner class B { // 암묵적으로 @B 라벨 사용
        fun Int.foo() { // 암묵적으로 @foo 라벨 사용
            val a = this@A // A의 this
            val b = this@B // B의 this

            val c = this // foo()의 리시버인 Int
            val c1 = this@foo // foo()'의 리시버인 Int

            val funLit = lambda@ fun String.() {
                val d = this // funLit의 리시버
            }


            val funLit2 = { s: String ->
                // 둘러싼 람다식에 리시버가 없으므로
                // foo()의 리시버
                val d1 = this
            }
        }
    }
}
```

동등성(Equality)

코틀린에는 두 가지 종류의 동등성이 있다:

— 참조 동등성 (두 참조가 동일 객체를 가리킴)

— 구조 동등성 (`equals()` 로 검사).

참조 동등

참조 동등은 `===` 오퍼레이션으로 검사한다(반대는 `!==` 로 검사). `a === b` 는 `a` 와 `b` 가 동일 객체를 가리키는 경우에만 true이다.

구조 동등

구조 동등은 `==` 오퍼레이션으로 검사한다(반대는 `!=` 로 검사). `a == b` 와 같은 식을 다음과 같이 바꾼다.

```
a?.equals(b) ?: (b === null)
```

`a` 가 `null` 이 아니면 `equals(Any?)` 함수를 호출하고, `a` 가 `null` 이면 `b` 가 `null` 인지 검사한다. `null` 을 직접 비교할 때에는 코드를 최적화할 필요가 없다. `a == null` 을 자동으로 `a === null` 로 바꾼다.

부동 소수점 숫자의 동등

동등 비교의 피연산자가 정적으로 (null 가능이거나 아닌) `Float` 나 `Double` 임을 알 수 있으면, 부동 소수점 계산을 위한 IEEE 754 표준에 따라 검사한다.

이 외의 경우 구조 동등을 사용한다. 이는 표준을 따르지 않으므로 `NaN` 는 그 자신과 같고, `-0.0` 은 `0.0` 과 같지 않다.

부동 소수점 비교 를 참고한다.

연산자 오버로딩

코틀린은 타입에 대해 미리 정의한 연산자의 구현을 제공할 수 있다. 이 연산자는 `+` 나 `*` 과 같이 부호 표기가 고정되어 있고 정해진 우선순위 를 갖는다. 연산자를 구현하려면 정해진 이름을 가진 멤버 함수 나 확장 함수 를 해당 타입에 제공한다. 이 타입은 이항 연산자의 왼쪽 타입이고 단항 연산자의 인자 타입이다. 연산자를 오버로딩한 함수는 `operator` 수식어를 붙여야 한다.

이어서 다른 연산자를 위한 연산자 오버로딩을 규정하는 관례를 살펴보자.

단항 연산자

단항 접두 연사자

식	변환
`+a`	`a.unaryPlus()`
`-a`	`a.unaryMinus()`
`!a`	`a.not()`

이 표는 컴파일러가 다음 절차에 따라 처리한다는 것을 보여준다(예로 `+a` 를 사용했다):

— `a` 의 타입을 결정한다. 타입을 `T` 라고 해 보자.

— 리시버 `T` 에 대해 파라미터가 없고 `operator` 이고 이름이 `unaryPlus()` 인 멤버 함수나 확장 함수를 찾는다.

— 함수가 없거나 모호하면 컴파일 에러가 발생한다.

— 함수가 존재하고 리턴 타입이 `R` 이면 `+a` 식의 타입은 `R` 이다.

이 오퍼레이션뿐만 아니라 다른 오퍼레이션은 기본 타입 에 대해 최적화를 하므로 함수 호출에 따른 오버헤드가 발생하지 않는다.

다음 예는 단항 빼기 연산자를 오버라이딩하는 방법을 보여준다:

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

val point = Point(10, 20)
println(-point)  // "(-10, -20)" 출력
```

증가와 감소

식	변환
`a++`	`a.inc()` + 아래 참고
`a--`	`a.dec()` + 아래 참고

`inc()` 와 `dec()` 함수는 `++` 나 `--` 오퍼레이션을 사용하는 변수에 할당할 값을 리턴해야 한다. 두 함수는 `inc` 나 `dec` 가 불린 객체를 수정하면 안 된다.

컴파일러는 다음 절차에 따라 *접미사* 형식의 연산자를 처리한다(예로 `a++` 를 사용했다).

— `a` 의 타입을 결정한다. `T` 라고 해 보자.

— `T` 타입의 리시버에 적용할 수 있는 파라미터 없고 `operator` 가 붙은 `inc()` 함수를 찾는다.

— 함수의 리턴 타입이 `T` 의 하위타입인지 검사한다.

계산 식의 효과는 다음과 같다:

— `a` 의 초기 값을 임시 저장소 `a0` 에 보관한다.

— `a.inc()` 의 결과를 `a` 에 할당한다.

— `a0` 를 식의 결과로 리턴한다.

`a--` 처리 과정도 완전히 같다.

`++a` 나 `--a` 와 같은 *접두사* 형식도 동일 방식으로 동작하고, 효과는 다음과 같다:

— `a.inc()` 를 `a` 에 할당한다.

— `a` 의 새 값을 식의 결과로 리턴한다.

이항 연산자

수치 연산자

식	변환
`a + b`	`a.plus(b)`
`a - b`	`a.minus(b)`
`a * b`	`a.times(b)`
`a / b`	`a.div(b)`
`a % b`	`a.rem(b)` , `a.mod(b)` (deprecated)
`a..b`	`a.rangeTo(b)`

이 표의 연산자에 대해 컴파일러는 *변환* 칼럼의 식으로 변환만 한다.

코틀린 1.1부터 `rem` 연산자를 지원한다. 코틀린 1.0의 `mod` 연산자는 1.1에서 디프리케이트되었다.

예제

아래 Counter 클래스 예제는 주어진 값으로 시작하고 오버로딩한 `+` 연산자를 사용해서 증가시킬 수 있다.

```
data class Counter(val dayIndex: Int) {
    operator fun plus(increment: Int): Counter {
        return Counter(dayIndex + increment)
    }
}
```

'In' 연산자

식	변환
`a in b`	`b.contains(a)`
`a !in b`	`!b.contains(a)`

`in` 과 `!in` 의 절차는 동일하다. 결과만 반대이다.

인덱스 기반 접근 연산자

식	변환
`a[i]`	`a.get(i)`
`a[i, j]`	`a.get(i, j)`
`a[i_1, ..., i_n]`	`a.get(i_1, ..., i_n)`
`a[i] = b`	`a.set(i, b)`
`a[i, j] = b`	`a.set(i, j, b)`
`a[i_1, ..., i_n] = b`	`a.set(i_1, ..., i_n, b)`

대괄호는 각각 알맞은 개수의 인자를 사용한 `get` 과 `set` 호출로 바뀐다.

호출 연산자

식	변환
`a()`	`a.invoke()`
`a(i)`	`a.invoke(i)`
`a(i, j)`	`a.invoke(i, j)`
`a(i_1, ..., i_n)`	`a.invoke(i_1, ..., i_n)`

증강 할당(Augmented assignments)

식	변환
`a += b`	`a.plusAssign(b)`
`a -= b`	`a.minusAssign(b)`
`a *= b`	`a.timesAssign(b)`
`a /= b`	`a.divAssign(b)`
`a %= b`	`a.remAssign(b)` , `a.modAssign(b)` (deprecated)

위 할당 오퍼레이션에 대해, 예를 들어 `a += b` 는 다음과 단계를 수행한다:

— 오른쪽 칼럼의 함수가 사용가능하면

— 대응하는 이항 함수(예, `plusAssign()` 의 경우 `plus()` 함수)도 사용가능하면 에러 발생(애매모호함 때문)

— 리턴 타입이 `Unit` 인지 확인하고 그렇지 않으면 에러

— `a.plusAssign(b)` 에 대한 코드 생성

— 사용가능하지 않으면, `a = a + b` 코드 생성을 시도한다(이는 타입 검사를 포함한다. `a + b` 의 타입은 `a` 의 상위타입이어야 한다)

주의 : 코틀린에서 할당은 식이 *아니다* .

동등과 비동등 연산자

식	변환
`a == b`	`a?.equals(b) ?: (b === null)`
`a != b`	`!(a?.equals(b) ?: (b === null))`

주의 : `===` 와 `!==` (동일성 검사)는 오버로딩할 수 없어서 두 연산자를 위한 변환은 제공하지 않는다.

`==` 오퍼레이션은 특수하다. 이 식은 `null` 을 선별하기 위해 복잡한 식으로 변환된다. `null == null` 은 항상 true이고 null이 아닌 `x` 에 대해 `x == null` 은 항상 false이며, `x.equals()` 를 호출하지 않는다.

비교 연산자

식	변환
`a > b`	`a.compareTo(b) > 0`
`a < b`	`a.compareTo(b) < 0`
`a >= b`	`a.compareTo(b) >= 0`
`a <= b`	`a.compareTo(b) <= 0`

모든 비교연산자는 `compareTo` 함수 호출로 바뀌고 `compareTo` 함수는 `Int` 를 리턴해야 한다.

프로퍼티 위임 연산자

`provideDelegate` , `getValue` , `setValue` 연산자 함수에 대한 내용은 위임 프로퍼티 에서 설명한다.

이름 가진 함수를 위한 중위 호출

중의 함수 호출 을 사용해서 커스텀 중위 연산자를 흉내낼 수 있다.

Null 안정성

Null 가능 타입과 Not-null 타입

코틀린 타입 시스템은 10억 달러 실수 라고 알려진, 코드에서 null을 참조하는 위험을 없애는 데 도움을 준다.

자바를 포함한 많은 프로그래밍 언어에서 공통으로 발견되는 실수 중 하나는 null 레퍼런스의 멤버를 참조해서 null 참조 익셉션이 발생하는 것이다. 자바에서는 `NullPointerException`, 줄여서 NPE가 이에 해당한다.

코틀린 타입 시스템은 코드에서 `NullPointerException`을 제거하는데 도움을 준다. NPE는 다음 상황에서만 가능하다:

— `throw NullPointerException()`를 명시적으로 실행

— 뒤에서 설명한 `!!` 연산자 사용

— NPE를 발생하는 외부의 자바 코드 사용

— 초기화와 관련해서 데이터의 일관성이 깨졌을 때(생성자 어딘가에서 초기화되지 않은 *this* 를 사용할 수 있음)

코틀린 타입 시스템은 `null` 을 가질 수 있는 레퍼런스(nullable 레퍼런스)와 가질 수 없는 레퍼런스(non-null 레퍼런스)를 구분한다. 예를 들어, `String` 타입 변수는 `null` 을 가질 수 없다:

```
var a: String = "abc"
a = null // 컴파일 에러
```

null을 가질 수 있으려면 `String?` 와 같이 변수를 null 가능 String으로 선언해야 한다:

```
var b: String? = "abc"
b = null // ok
```

이제 `a` 의 프로퍼티에 접근하거나 메서드를 호출할 때 NPE가 발생하지 않고 안전하게 사용할 수 있다는 것을 보장한다:

```
val l = a.length
```

하지만, `b` 의 동일한 프로퍼티에 접근시도를 하면 안전하지 않으므로 컴파일러 에러가 발생한다:

```
val l = b.length // 에러: 변수 'b'는 null일 수 있다
```

하지만, 여전히 그 프로퍼티에 접근해야 한다면, 이를 위한 몇 가지 방법이 있다.

조건으로 `null` 검사하기

첫 번째로 `b` 가 `null` 인지 명시적으로 검사하고 두 옵션을 따로 처리할 수 있다:

```
val l = if (b != null) b.length else -1
```

컴파일러는 검사 결과 정보를 추적해서 `if` 안에서 `length` 를 호출할 수 있도록 한다. 더 복잡한 조건도 지원한다:

```
if (b != null && b.length > 0) {
    print("String of length ${b.length}")
} else {
    print("Empty string")
}
```

이 코드는 `b` 가 불변인 경우에만 작동한다(예를 들어, null 검사와 사용 사이에서 수정할 수 없는 로컬 변수이거나 또는 지원 필드가 있고 오버라이딩할 수 없는 `val` 멤버인 경우). 왜냐면, 그렇지 않을 경우 검사 이후에 `b` 를 `null` 로 바꿀 수 있기 때문이다.

안전한 호출

두 번째는 안전 호출 연산자인 `?.` 를 사용하는 것이다:

```
b?.length
```

이 코드는 `b` 가 null이 아니면 `b.length` 를 리턴하고 그렇지 않으면 `null` 을 리턴한다. 이 식의 타입은 `Int?` 이다.

안전 호출은 체인에서 유용하다. Employee 타입의 Bob이 Department에 속할(또는 속하지 않을) 수 있고, department가 또 다른 Employee를 department의 head로 가질 수 있다고 할 때, 존재할 수도 있는 Bod의 department의 head를 구하는 코드를 다음과 같이 작성할 수 있다:

```
bob?.department?.head?.name
```

이 체인은 프로퍼티 중 하나라도 null이면 `null` 을 리턴한다.

non-null 값에만 특정 오퍼레이션을 수행하고 싶다면 안전 호출 연산자를 `let` 과 함께 사용하면 된다:

```
val listWithNulls: List<String?> = listOf("A", null)
for (item in listWithNulls) {
     item?.let { println(it) } // A를 출력하고 null을 무시
}
```

엘비스 연산자

null 가능 레퍼런스인 `r` 에 대해, "`r` 이 null이 아니면 그것을 사용하고 그렇지 않으면 다른 not-null 값인 `x` 를 사용한다"는 코드를 다음과 같이 작성할 수 있다:

```
val l: Int = if (b != null) b.length else -1
```

완전한 `if` -식 외에, 엘비스 연산자인 `?:` 를 사용해서 다음과 같이 표현할 수도 있다:

```
val l = b?.length ?: -1
```

`?:` 의 왼쪽 식이 null이 아니면, 엘비스 연산자는 그것을 리턴하고, 그렇지 않으면 오른쪽 식을 리턴한다. 오른쪽 식은 왼쪽이 null일 경우에만 평가한다.

코틀린에서 `throw` 와 `return` 은 식이므로, 엘비스 연산자의 오른쪽에 둘을 사용할 수 있다. 이는 함수 인자를 검사할 때 매우 유용하다:

```
fun foo(node: Node): String? {
    val parent = node.getParent() ?: return null
    val name = node.getName() ?: throw IllegalArgumentException("name expected")
    // ...
}
```

`!!` 연산자

`!!` 연산자는 NPE를 선호나는 개발자를 위한 세 번째 옵션이다. `b!!` 라고 작성하면 `b` 가 null이 아니면 b를 리턴하고(예를 들어, `String`), null이면 NPE를 발생한다:

```
val l = b!!.length
```

따라서 NPE를 원한다면 이 연산자를 사용하면 된다. 하지만, NPE는 명시적으로 사용해야 하며 뜻밖의 코드에서 나타나지 않도록 한다.

안전한 타입 변환

일반 타입 변환은 객체가 지정한 타입이 아니면 `ClassCastException` 을 발생한다. 타입 변환 방법을 위한 다른 방법은 타입 변환에 실패하면 `null` 을 리턴하는 안전한 타입 변환을 사용하는 것이다:

```
val aInt: Int? = a as? Int
```

null 가능 타입의 콜렉션

null 가능 타입의 요소를 갖는 콜렉션에서 null이 아닌 요소를 걸러내고 싶다면, `filterNotNull` 를 사용한다:

```
val nullableList: List<Int?> = listOf(1, 2, null, 4)
val intList: List<Int> = nullableList.filterNotNull()
```

익셉션

익셉션 클래스

코틀린의 모든 익셉션 클래스는 `Throwable` 클래스의 자식 클래스이다. 모든 익셉션은 메시지, 스택 트레이스, 선택적인 원인을 포함한다.

익셉션 객체를 던지려면, `throw` -식을 사용한다:

```
throw MyException("Hi There!")
```

익셉션을 잡으려면 `try` -식을 사용한다:

```
try {
    // 어떤 코드
}
catch (e: SomeException) {
    // 익셉션 처리
}
finally {
    // 선택적인 finally 블록
}
```

`catch` 블록은 없거나 여러 개 올 수 있다. `finally` 블록은 생략할 수 있다. 하지만, 최소 한 개의 `catch` 블록이나 `finally` 블록은 존재해야 한다:

try는 식이다

`try` 는 식이어서 값을 리턴할 수 있다:

```
val a: Int? = try { parseInt(input) } catch (e: NumberFormatException) { null }
```

`try` -식이 리턴한 값은 `try` 블록의 마지막 식이거나 `catch` 블록(또는 블록들의)의 마지막 식이다. `finally` 블록의 내용은 식의 결과에 영향을 주지 않는다.

Checked 익셉션

코틀린에는 Checked 익셉션이 없다. 여기에는 여러 이유가 있는데, 간단한 예로 살펴보자.

다음은 JDK의 `StringBuilder` 클래스가 구현한 인터페이스 예이다.

```
Appendable append(CharSequence csq) throws IOException;
```

이 시그너처는 무엇을 말할까? 이것은 어딘가에(`StringBuilder` , 어떤 종류의 로그, 콘솔 등) 문자열을 추가할 때마다 `IOExceptions` 을 잡아야 한다고 말한다. 왜? 그것이 IO 연산을 할 수도 있기 때문이다(`Writer` 도 `Appendable` 를 구현하고 있다). 이는 모든 곳에 다음과 같이 익셉션을 무시하는 코드를 만든다:

```
try {
    log.append(message)
}
catch (IOException e) {
    // 안전해야 함
}
```

이는 좋은 게 아니다. 이에 대한 내용은 Effective Java , Item 65: *Don't ignore exceptions* 를 참고하자.

Bruce Eckel은 Does Java need Checked Exceptions? 에서 다음과 같이 말했다:

> 작은 프로그램에서의 실험 결과 익셉션 규약을 강제하는 것이 개발 생산성과 코드 품질을 향상시킨다는 결론을 얻었다. 하지만, 대형 소프트웨어 프로젝트에서 경험한 결과는 다른 결론을 말한다. 생산성이 떨어지고 코드 품질에서 거의 향상이 이루어지지 않았다.

다음은 이와 관련된 다른 논의이다:

- Java's checked exceptions were a mistake (Rod Waldhoff)
- The Trouble with Checked Exceptions (Anders Hejlsberg)

Nothing 타입

코틀린에서 `throw` 는 식이므로, 식을 사용할 수 있다. 예를 들어, 엘비스 식에서 사용할 수 있다:

```
val s = person.name ?: throw IllegalArgumentException("Name required")
```

`throw` 식의 타입은 특수 타입인 `Nothing` 이다. 이 타입은 값을 갖지 않으면 도달할 수 없는 코드를 표시하는 용도로 사용한다. 코드에서 리턴하지 않는 함수를 표시할 때 `Nothing` 을 사용할 수 있다:

```
fun fail(message: String): Nothing {
    throw IllegalArgumentException(message)
}
```

이 함수를 호출하면, 컴파일러는 호출 이후 실행이 계속되지 않는 것을 알 것이다:

```
val s = person.name ?: fail("Name required")
println(s)     // 이 시점에 's'가 초기화된 것을 안다
```

Nothing 타입을 만나는 또 다른 경우는 타입 추론이다. Nothing의 null 가능 타입인 `Nothing?` 은 정확하게 `null` 한 가지 값만 가진다. `null` 을 사용해서 추론할 타입의 값을 초기화하면 더 구체적인 타입을 결정할 수 있는 다른 정보가 없기 때문에 컴파일러는 `Nothing?` 타입으로 유추한다:

```
val x = null           // 'x'는 `Nothing?` 타입을 가짐
val l = listOf(null)   // 'l'은 `List<Nothing?>` 타입을 가짐
```

자바 상호운용성

익센션의 자바 상호운용성에 대한 내용은 자바 상호운용 에서 확인할 수 있다.

애노테이션

애노테이션 선언

애노테이션은 코드에 메타데이터를 붙이는 방법이다. 클래스 앞에 `annotation` 수식어를 붙여서 애노테이션을 선언한다:

```
annotation class Fancy
```

애노테이션 클래스에 메타-애노테이션을 붙여서 애노테이션에 추가 속성을 지정할 수 있다:

- `@Target` 은 애노테이션을 어떤 요소에(클래스, 함수, 프로퍼티, 식 등) 적용할 수 있는지 지정한다.
- `@Retention` 은 애노테이션을 컴파일한 클래스 파일에 보관할지, 런타임에 리플렉션을 통해서 접근할 수 있는지를 지정한다. (기본값은 둘 다 true이다.)
- `@Repeatable` 은 한 요소의 같은 애노테이션을 여러 번 적용하는 것을 허용한다.
- `@MustBeDocumented` 은 애노테이션이 공개 API에 속하며 생성한 API 문서에 클래스나 메서드 시그너처를 포함시켜야 함을 지정한다.

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
annotation class Fancy
```

사용법

```
@Fancy class Foo {
    @Fancy fun baz(@Fancy foo: Int): Int {
        return (@Fancy 1)
    }
}
```

클래스의 주요 생성자에 애노테이션이 필요하면, `constructor` 키워드를 생성자 선언에 추가하고 그 앞에 애노테이션을 추가해야 한다:

```
class Foo @Inject constructor(dependency: MyDependency) {
    // ...
}
```

프로퍼티 접근자에도 애노테이션을 붙일 수 있다:

```
class Foo {
    var x: MyDependency? = null
        @Inject set
}
```

생성자

애노테이션은 파라미터가 있는 생성자를 가질 수 있다.

```
annotation class Special(val why: String)

@Special("example") class Foo {}
```

허용하는 파라미터 타입은 다음과 같다:

- 자바 기본 타입에 대응하는 타입(Int, Long 등)
- 문자열
- 클래스 (`Foo::class`)
- 열거 타입
- 다른 애노테이션
- 위에서 열거한 타입의 배열

추가 파라미터는 null 가능 타입일 수 없는데, 왜냐면 JVM은 애노테이션 속성의 값으로 `null` 을 저장하는 것을 지원하지 않기 때문이다.

다른 애노테이션의 파라미터로 애노테이션을 사용할 경우, 해당 애노테이션의 이름에 @ 문자를 접두어로 붙이지 않는다.

```
annotation class ReplaceWith(val expression: String)

annotation class Deprecated(
        val message: String,
        val replaceWith: ReplaceWith = ReplaceWith(""))

@Deprecated("This function is deprecated, use === instead", ReplaceWith("this === other"))
```

애노테이션의 인자로 클래스를 사용하고 싶다면, 코틀린 클래스(KClass)를 사용한다. 코틀린 컴파일러는 이를 자동으로 자바 클래스로 변환하므로, 자바 코드에서 애노테이션과 인자를 사용할 수 있다.

```
import kotlin.reflect.KClass

annotation class Ann(val arg1: KClass<*>, val arg2: KClass<out Any?>)

@Ann(String::class, Int::class) class MyClass
```

람다

람다에도 애노테이션을 사용할 수 있다. 람다의 몸체에 생성한 `invoke()` 메서드에 적용된다. 병렬 제어를 위해 애노테이션을 사용하는 Quasar 와 같은 프레임워크에서 이를 유용하게 사용한다.

```
annotation class Suspendable

val f = @Suspendable { Fiber.sleep(10) }
```

애노테이션 사용 위치 대상

프로퍼티나 주요 생성자 파라미터에 애노테이션을 적용할 때, 코틀린 요소에서 여러 자바 요소가 생성될 수 있고 따라서 생성된 자바 바이트코드에는 애노테이션이 여러 위치에 붙을 수 있다. 애노테이션을 정확하게 어떻게 생성할지 지정하려면 다음 구문을 사용한다:

```
class Example(@field:Ann val foo,    // 자바 필드에 적용
              @get:Ann val bar,      // 자바 getter에 적용
              @param:Ann val quux)   // 자바 생성자 파라미터에 적용
```

파일에 애노테이션을 적용하기 위해 같은 구문을 사용할 수 있다. 이를 위해 파일의 최상단에 패키지 디렉티브 이전에 또는 기본 패키지면 모든 임포트 이전에 적용 대상으로 `file` 을 가진 애노테이션을 붙인다:

```
@file:JvmName("Foo")

package org.jetbrains.demo
```

동일 대상에 여러 애노테이션을 붙일 경우 대상 뒤의 대괄호에 모든 애노테이션을 위치시켜서 대상을 반복하는 것을 피할 수 있다:

```
class Example {
     @set:[Inject VisibleForTesting]
     var collaborator: Collaborator
}
```

지원하는 사용 위치 대상은 다음과 같다:

- `file`
- `property` (이 대상을 가진 애노테이션은 자바에는 보이지 않는다)
- `field`
- `get` (프로퍼티 getter)
- `set` (프로퍼티 setter)
- `receiver` (확장 함수나 프로퍼티의 리시버 파라미터)
- `param` (생성자 파라미터)

— `setparam` (프로퍼티 setter 파라미터)
— `delegate` (위임 프로퍼티의 위임 인스턴스를 보관하는 필드)

확장 함수의 리시버 파라미터에 애노테이션을 붙이려면 다음 구문을 사용한다:

```
fun @receiver:Fancy String.myExtension() { }
```

사용 위치 대상을 지정하지 않으면, 사용할 애노테이션의 `@Target` 애노테이션에 따라 대상을 선택한다. 적용 가능한 대상이 여러 개면, 다음 목록에서 먼저 적용가능한 대상을 사용한다:

— `param`
— `property`
— `field`

자바 애노테이션

자바 애노테이션은 코틀린과 100% 호환된다:

```
import org.junit.Test
import org.junit.Assert.*
import org.junit.Rule
import org.junit.rules.*

class Tests {
    // 프로퍼티 getter에 @Rule 애노테이션을 적용
    @get:Rule val tempFolder = TemporaryFolder()

    @Test fun simple() {
        val f = tempFolder.newFile()
        assertEquals(42, getTheAnswer())
    }
}
```

자바에서 작성한 애노테이션은 파라미터의 순서를 정의하지 않기 때문에 인자를 전달하기 위해 일반적인 함수 호출 구문을 사용할 수 없다. 대신 이름을 가진 인자 구문을 사용해야 한다:

```
// Java
public @interface Ann {
    int intValue();
    String stringValue();
}
```

```
// Kotlin
@Ann(intValue = 1, stringValue = "abc") class C
```

자바와 같이 `value` 파라미터는 특별하다. `value` 파라미터의 값은 이름 없이 지정할 수 있다:

```
// Java
public @interface AnnWithValue {
    String value();
}
```

```
// Kotlin
@AnnWithValue("abc") class C
```

자바의 `value` 인자가 배열 타입이면, 코틀린에서 `vararg` 파라미터가 된다:

```
// Java
public @interface AnnWithArrayValue {
    String[] value();
}
```

```kotlin
// Kotlin
@AnnWithArrayValue("abc", "foo", "bar") class C
```

배열 타입을 가진 다른 인자의 경우 `arrayOf` 를 직접 사용해야 한다:

```java
// Java
public @interface AnnWithArrayMethod {
    String[] names();
}
```

```kotlin
// Kotlin
@AnnWithArrayMethod(names = arrayOf("abc", "foo", "bar")) class C
```

애노테이션 인스턴스의 값은 코틀린 코드에 프로퍼티로 노출된다:

```java
// Java
public @interface Ann {
    int value();
}
```

```kotlin
// Kotlin
fun foo(ann: Ann) {
    val i = ann.value
}
```

리플렉션

리플렉션은 란타임에 프로그램의 구조를 볼 수 있게 해주는 라이브러리와 언어 특징이다. 코틀린에서 함수와 프로퍼티는 1급이며, 이 둘의 내부를 알아내는 것은(예, 런타임에 이름, 프로퍼티 타입이나 함수를 알아내는 것) 함수형 또는 리액티브 방식을 간단하게 사용하는 것과 밀접하게 관련있다.

> ⚠ 자바에서 리플렉션 특징을 사용하는데 필요한 런타임 컴포넌트는 별도 JAR 파일(`kotlin-reflect.jar`)로 제공한다. 이를 통해 리플렉션 특징을 사용하지 않는 어플리케이션이 필요로 하는 런타임 라이브러리의 크기를 줄였다. 리플렉션을 사용한다면 그 jar 파일을 프로젝트의 클래스패스에 추가해야 한다.

클래스 레퍼런스

가장 기본적인 리플렉션 특징은 코틀린 클래스에 대한 런타임 레퍼런스를 구하는 것이다. 정적으로 알려진 코틀린 클래스에 대한 레퍼런스를 얻으려면 *class 리터럴* 구문을 사용하면 된다:

```
val c = MyClass::class
```

레퍼런스의 값은 KClass 타입이다.

코틀린 클래스 레퍼런스는 자바 클래스 레퍼런스와 같지 않다. 자바 클래스 레퍼런스를 구하려면 `KClass` 인스턴스의 `.java` 프로퍼티를 사용한다.

객체에 묶은(Bound) 클래스 레퍼런스 (1.1부터)

리시버로 객체를 사용하면 `::class` 구문을 통해 특정 객체의 클래스에 대한 레퍼런스를 구할 수 있다:

```
val widget: Widget = ...
assert(widget is GoodWidget) { "Bad widget: ${widget::class.qualifiedName}" }
```

예를 들어, 위 코드는 리시버 식의 타입(`Widget`)에 상관없이 `GoodWidget` 또는 `BadWidget` 인스턴스의 클래스에 대한 레퍼런스를 구한다.

함수 레퍼런스

다음과 같이 이름 가진 함수를 선언했다고 하자:

```
fun isOdd(x: Int) = x % 2 != 0
```

`isOdd(5)` 와 같이 쉽게 함수를 바로 호출할 수 있지만, 또한 함수를 값으로 다른 함수에 전달할 수 있다. 이를 위해 `::` 연산자를 사용한다:

```
val numbers = listOf(1, 2, 3)
println(numbers.filter(::isOdd)) // [1, 3] 출력
```

여기서 `::isOdd` 는 함수 타입 `(Int) -> Boolean` 의 값이다.

`::` 는 문백을 통해 원하는 타입을 알 수 있을 때, 오버로딩한 함수에서도 사용할 수 있다. 다음 예를 보자:

```
fun isOdd(x: Int) = x % 2 != 0
fun isOdd(s: String) = s == "brillig" || s == "slithy" || s == "tove"

val numbers = listOf(1, 2, 3)
println(numbers.filter(::isOdd)) // isOdd(x: Int)를 참조
```

또는 변수에 명시적으로 타입을 지정해서 메서드 레퍼런스를 저장할 때 필요한 문맥을 제공할 수 있다:

```
val predicate: (String) -> Boolean = ::isOdd   // isOdd(x: String)를 참고
```

클래스 멤버나 확장 함수를 사용해야 한다면 한정자를 붙여야 한다. 예를 들어, `String::toCharArray` 는 `String` : `String.() -> CharArray` 타입을 위한 확장 함수를 제공한다.

예: 함수 조합

다음 함수를 보자:

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}
```

이 함수는 파라미터로 전달한 두 함수를 조합한다(`compose(f, g) = f(g(*))`). 이제 이 함수에 호출가능한 레퍼런스를 적용할 수 있다:

```
fun length(s: String) = s.length

val oddLength = compose(::isOdd, ::length)
val strings = listOf("a", "ab", "abc")

println(strings.filter(oddLength)) // "[a, abc]" 출력
```

프로퍼티 참조

코틀린에서 1급 객체로서 프로퍼티에 접근할 때에도 `::` 연산자를 사용한다:

```
var x = 1

fun main(args: Array<String>) {
    println(::x.get()) // "1" 출력
    ::x.set(2)
    println(x)         // "2" 출력
}
```

`::x` 식은 `KProperty<Int>` 타입의 프로퍼티 객체로 평가하는데 이는 `get()` 을 사용해서 프로퍼티의 값을 읽거나 `name` 프로퍼티를 사용해서 프로퍼티 이름을 가져올 수 있도록 해 준다. 더 자세한 정보는 KProperty 클래스 문서 를 참고한다.

수정 가능한 프로퍼티, 예를 들어 `var y =` 에 대해 `::y` 는 `set()` 함수를 가진 KMutableProperty<Int> 타입의 값을 리턴한다.

파라미터가 없는 함수가 필요한 곳에 프로퍼티 레퍼런스를 사용할 수 있다:

```
val strs = listOf("a", "bc", "def")
println(strs.map(String::length)) // [1, 2, 3] 출력
```

클래스 멤버인 프로퍼티에 접근할 때에는 클래스로 한정한다:

```
class A(val p: Int)

fun main(args: Array<String>) {
    val prop = A::p
    println(prop.get(A(1))) // "1" 출력
}
```

확장 프로퍼티의 경우:

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

fun main(args: Array<String>) {
    println(String::lastChar.get("abc")) // "c" 출력
}
```

자바 리플렉션과 상호운영성

자바 플랫폼에서, 표준 라이브러리는 자바 리플렉션 객체와의 매핑을 제공하는 리플렉션 클래스 확장을 포함한다(`kotlin.reflect.jvm` 패키지 참고). 예를 들어, 지원 필드를 찾거나 코틀린 프로퍼티에 대한 getter로 동작하는 자바 메서드를 찾으려면, 다음과 같은 코드를 사용할 수 있다:

```
import kotlin.reflect.jvm.*

class A(val p: Int)

fun main(args: Array<String>) {
    println(A::p.javaGetter) // "public final int A.getP()" 출력
    println(A::p.javaField)  // "private final int A.p" 출력
}
```

자바 클래스에 대응한 코틀린 클래스를 얻으려면, `.kotlin` 확장 프로퍼티를 사용한다:

```
fun getKClass(o: Any): KClass<Any> = o.javaClass.kotlin
```

생성자 레퍼런스

메서드나 프로퍼티처럼 생성자도 참조할 수 있다. 생성자와 같은 파라미터를 갖고 해당 타입의 객체를 리턴하는 함수 타입 객체가 필요한 곳에 생성자 레퍼런스를 사용할 수 있다. `::` 연산자에 클래스 이름을 붙여서 생성자 레퍼런스를 구한다. 다음 함수를 보자. 이 함수는 파라미터가 없고 리턴 타입이 `Foo` 인 함수 파라미터를 갖고 있다:

```
class Foo

fun function(factory: () -> Foo) {
    val x: Foo = factory()
}
```

Foo 클래스의 인자없는 생성자인 `::Foo` 를 사용하면 다음과 같이 간단하게 생성자를 호출할 수 있다:

```
function(::Foo)
```

객체에 묶은(Bound) 함수 레퍼런스와 프로퍼티 레퍼런스 (1.1부터)

특정 객체의 인스턴스 메서드를 참조할 수 있다:

```
val numberRegex = "\\d+".toRegex()
println(numberRegex.matches("29")) // "true" 출력

val isNumber = numberRegex::matches
println(isNumber("29")) // "true" 출력
```

`matches` 메서드를 직접 호출하는 대신, 그 메서드에 대한 레퍼런스를 저장하고 있다. 이 레퍼런스는 그 리서버에 묶인다. 위 예제처럼 직접 레퍼런스를 호출할 수 있고 또는 해당 함수 타입이 필요한 곳에 사용할 수 있다:

```
val strings = listOf("abc", "124", "a70")
println(strings.filter(numberRegex::matches)) // "[124]" 출력
```

객체에 묶은 타입과 그에 대응하는 클래스 레퍼런스를 비교해보자. 객체에 묶은 호출가능한 레퍼런스는 리시버를 갖는다. 그래서 리시버 타입 파라미터가 필요 없다:

```
val isNumber: (CharSequence) -> Boolean = numberRegex::matches

val matches: (Regex, CharSequence) -> Boolean = Regex::matches
```

프로퍼티 레퍼런스도 묶을 수 있다:

```
val prop = "abc"::length
println(prop.get())   // prints "3"
```

타입에 안전한 빌더

그루비 커뮤니티에서 빌더 개념은 유명하다. 빌더는 반정도 선언적인 방식으로 데이터를 정의할 수 있도록 한다. XML 생성 , UI 컴포넌트 배치 , 3D 장면 설명 등이 빌더의 좋은 예이다.

코틀린은 다양한 용도로 사용할 수 있는 *타입에 안전한* 빌더를 허용한다. 심지어 이 빌더는 그루비 자체에서 만든 동적 타입 구현보다 더 매력적이다.

코틀린은 타입에 안전한 빌더로 하지 못하는 경우를 위해 동적 타입 빌더도 지원한다.

타입에 안전한 빌더 예제

다음 코드를 보자:

```
import com.example.html.* // 아래 선언을 보자

fun result(args: Array<String>) =
    html {
        head {
            title {+"XML encoding with Kotlin"}
        }
        body {
            h1 {+"XML encoding with Kotlin"}
            p  {+"this format can be used as an alternative markup to XML"}

            // 속성과 텍스트 내용을 가진 요소
            a(href = "http://kotlinlang.org") {+"Kotlin"}

            // 혼합 내용
            p {
                +"This is some"
                b {+"mixed"}
                +"text. For more see the"
                a(href = "http://kotlinlang.org") {+"Kotlin"}
                +"project"
            }
            p {+"some text"}

            // 생성한 내용
            p {
                for (arg in args)
                    +arg
            }
        }
    }
```

이 코드는 완전히 올바른 코틀린 코드이다. 이 코드를 여기 에서 실행해 볼 수 있다(브라우저에서 수정하고 실행해보자).

동작 방식

타입에 안전한 빌더를 구현하는 메커니즘을 살펴보자. 먼저 생성하고 싶은 모델을 정의해야 한다. 이 예에서는 HTML 태그를 위한 모델이 필요하다. 약간의 클래스로 쉽게 모델을 정의할 수 있다. 예를 들어, `<html>` 태그를 위한 클래스는 `HTML` 이다. 이는 `<head>` 와 `<body>` 와 같은 자식을 정의한다. (아래 클래스 선언을 참고한다.)

이제 코드에서 다음과 같이 호출할 수 있는 이유를 살펴보자:

```
html {
 // ...
}
```

`html` 은 실제로는 인자로 람다 식 을 받는 함수 호출이다. 임 함수는 다음과 같이 정의되어 있다:

```
fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

이 함수는 `init` 이란 이름의 파라미터를 가지며, 이 파라미터 자체는 함수이다. 이 함수의 타입은 `HTML.() -> Unit` 인데, 이는 *리시버를 가진 함수 타입_이다. 이는 함수에* `HTML` *(_리시버)* 타입의 인스턴스를 전달해야 하고 함수 안에서 그 인스턴스의 멤버를 호출할 수 있다는 것을 의미한다. `this` 키워드로 리시버에 접근할 수 있다:

```
html {
    this.head { /* ... */ }
    this.body { /* ... */ }
}
```

(`head` 와 `body` 는 `HTML` 의 멤버 함수이다)

`this` 를 생략할 수 있고, 이미 빌더처럼 보이는 것을 얻었다:

```
html {
    head { /* ... */ }
    body { /* ... */ }
}
```

그러면 이것은 무엇을 호출할까? 위에서 정의한 `html` 함수의 몸체를 보자. 함수는 `HTML` 인스턴스를 생성하고 인자로 전달받은 함수를 호출해서 그 인스턴스를 초기화하고(이 예에서는 `HTML` 인스턴스의 `head` 와 `body` 를 호출한다), 인스턴스를 리턴한다. 이것이 정확하게 빌더가 해야 하는 일이다.

`HTML` 의 `head` 와 `body` 함수도 `html` 과 유사하게 정의한다. 유일한 차이점은 두 함수는 둘러싼 `HTML` 인스턴스의 `children` 콜렉션에 생성한 인스턴스를 추가한다는 것이다:

```
fun head(init: Head.() -> Unit) : Head {
    val head = Head()
    head.init()
    children.add(head)
    return head
}

fun body(init: Body.() -> Unit) : Body {
    val body = Body()
    body.init()
    children.add(body)
    return body
}
```

실제로 이 두 함수는 같은 것을 하므로, 지네릭 버전인 `initTag` 를 만들 수 있다:

```
protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
    tag.init()
    children.add(tag)
    return tag
}
```

이제 두 함수가 매우 단순해진다:

```
fun head(init: Head.() -> Unit) = initTag(Head(), init)

fun body(init: Body.() -> Unit) = initTag(Body(), init)
```

그리고, `<head>` 와 `<body>` 태그를 생성하기 위해 두 함수를 사용할 수 있다.

여기서 논의할 만한 다른 한 가지는 어떻게 몸체에 텍스트를 넣느냐는 것이다. 예제 코드에서는 다음과 같이 했다:

```
html {
    head {
        title {+"XML encoding with Kotlin"}
    }
    // ...
}
```

기본적으로 태그 몸체에 단지 문자열을 넣었다. 하지만, 문자열 앞에 `+` 가 붙어 있는데, 이는 접두사 `unaryPlus()` 오퍼레이션을 호출하는 함수 호출이다. 이 오퍼레이션은 실제로 `TagWithText` 추상 클래스(`Title` 의 부모)의 멤버인 `unaryPlus()` 확장 함수로 정의되어 있다.

```
fun String.unaryPlus() {
    children.add(TextElement(this))
}
```

그래서 접두사 `+` 는 문자열을 `TextElement` 인스턴스로 감싸고, `children` 콜렉션에 추가하며, 따라서 태그 트리의 알맞은 부분이 된다.

지금까지 보여준 코드는 빌더 예제의 상단에서 임포트한 `com.example.html` 패키지에 정의했다. 마지막 절에 이 패키지 전체 코드를 실었다.

범위 제어: @DslMarker (1.1부터)

DSL을 사용할 때, 그 문백에서 호출할 수 있는 함수가 너무 많아질 수 있는 문제가 발생할 수 있다. 람다 안에서 모든 사용가능한 암묵적인 리서버의 메서드를 호출할 수 있고, 그 결과로 다른 `head` 안에 `head` 를 넣는 것고 같이 일관성이 깨지는 결과를 얻을 수 있다:

```
html {
    head {
        head {} // 금지해야 한다
    }
    // ...
}
```

이 예에서는 가장 가까운 암묵적인 리시버인 `this@head` 의 멤버만 가능해야 한다. `head()` 는 외부 리서버인 `this@html` 의 멤버이므로, `head()` 를 호출하는 것은 막아야 한다.

이 문제를 해결하기 위해, 코틀린 1.1은 리시버 범위를 제어하기 위한 특별한 메커니즘을 추가했다. 컴파일러가 범위 제어를 시작하도록 하려면, DSL에서 사용할 모든 리시버 타입에 동일한 마커 애노테이션을 붙여야 한다. 예를 들어, HTML 빌더의 경우 `@HTMLTagMarker` 애노테이션을 선언한다:

```
@DslMarker
annotation class HtmlTagMarker
```

`@DslMarker` 애노테이션이 붙은 애노테이션 클래스를 DSL 마커라고 부른다.

DSL에서 모든 태그 클래스는 같은 상위 클래스인 `Tag` 를 상속한다. 그 상위클래스에만 `@HtmlTagMarker` 를 붙여도 충분하며, 그 뒤로 코틀린 컴파일러는 모든 상속한 클래스를 `@HtmlTagMarker` 로 다룬다:

```
@HtmlTagMarker
abstract class Tag(val name: String) { ... }
```

`HTML` 나 `Head` 클래스에 `@HtmlTagMarker` 를 붙이면 안 되는데, 그 이유는 상위클래스에 이미 붙였기 때문이다:

```
class HTML() : Tag("html") { ... }
class Head() : Tag("head") { ... }
```

이 애노테이션을 추가하면, 코틀린 컴파일러는 암묵적 리시버가 동일 DSL의 어느 부분인지 알 수 있게 되고, 가장 가까운 리시버의 멤버만 호출할 수 있게 만들 수 있다::

```
html {
    head {
        head { } // 에러: 외부 리시버의 멤버
    }
    // ...
}
```

외부 리시버의 멤버를 호출하는 것은 여전히 가능하지만, 이를 하려면 명시적으로 리시버를 지정해야 한다:

```
html {
    head {
        this@html.head { } // 가능
    }
    // ...
}
```

`com.example.html` 패키지 전체 설명

이 절은 `com.example.html` 패키지를 어떻게 정의했는지 보여준다(위 예제에서 사용한 요소만 설명). 이 예는 HTML 트리를 생성한다. 이 예제는 식 함수 와 리시버를 가진 람다 를 많이 사용한다.

`@DslMarker` 애노테이션은 코틀린 1.1부터 가능하다.

```
package com.example.html

interface Element {
    fun render(builder: StringBuilder, indent: String)
}

class TextElement(val text: String) : Element {
    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent$text\n")
    }
}

@DslMarker
annotation class HtmlTagMarker

@HtmlTagMarker
abstract class Tag(val name: String) : Element {
    val children = arrayListOf<Element>()
    val attributes = hashMapOf<String, String>()

    protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
        tag.init()
        children.add(tag)
        return tag
    }

    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent<$name${renderAttributes()}>\n")
        for (c in children) {
            c.render(builder, indent + "  ")
        }
        builder.append("$indent</$name>\n")
    }

    private fun renderAttributes(): String {
        val builder = StringBuilder()
        for ((attr, value) in attributes) {
            builder.append(" $attr=\"$value\"")
        }
        return builder.toString()
    }

    override fun toString(): String {
        val builder = StringBuilder()
        render(builder, "")
        return builder.toString()
    }
}

abstract class TagWithText(name: String) : Tag(name) {
    operator fun String.unaryPlus() {
        children.add(TextElement(this))
    }
}

class HTML : TagWithText("html") {
    fun head(init: Head.() -> Unit) = initTag(Head(), init)

    fun body(init: Body.() -> Unit) = initTag(Body(), init)
}

class Head : TagWithText("head") {
    fun title(init: Title.() -> Unit) = initTag(Title(), init)
}

class Title : TagWithText("title")
```

```
abstract class BodyTag(name: String) : TagWithText(name) {
    fun b(init: B.() -> Unit) = initTag(B(), init)
    fun p(init: P.() -> Unit) = initTag(P(), init)
    fun h1(init: H1.() -> Unit) = initTag(H1(), init)
    fun a(href: String, init: A.() -> Unit) {
        val a = initTag(A(), init)
        a.href = href
    }
}

class Body : BodyTag("body")
class B : BodyTag("b")
class P : BodyTag("p")
class H1 : BodyTag("h1")

class A : BodyTag("a") {
    var href: String
        get() = attributes["href"]!!
        set(value) {
            attributes["href"] = value
        }
}

fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}
```

타입 별칭

타입 별칭은 타입을 위한 다른 이름을 제공한다. 타입 이름이 너무 길면 더 짧은 이름을 추가해서 그 이름을 대신 사용할 수 있다.

이는 긴 지네릭 타입 이름을 짧게 만들 때 유용하다. 예를 들어, 콜렉션 타입을 축약할 때 사용한다.

```
typealias NodeSet = Set<Network.Node>

typealias FileTable<K> = MutableMap<K, MutableList<File>>
```

함수 타입을 위한 다른 별칭을 제공할 수 있다:

```
typealias MyHandler = (Int, String, Any) -> Unit

typealias Predicate<T> = (T) -> Boolean
```

내부와 중첩 클래스를 위한 새 이름을 가질 수 있다:

```
class A {
    inner class Inner
}
class B {
    inner class Inner
}

typealias AInner = A.Inner
typealias BInner = B.Inner
```

타입 별칭이 새 타입을 추가하는 것은 아니다. 대응하는 기저 타입은 동일하다. 코드에 `typealias Predicate<T>` 를 추가하고 `Predicate<Int>` 를 사용하면 코틀린 컴파일러는 이를 `(Int) -> Boolean` 로 확장한다. 그래서 일반 함수 타입이 필요한 곳에 별칭 타입 변수를 전달할 수 있으며 반대도 가능하다:

```
typealias Predicate<T> = (T) -> Boolean

fun foo(p: Predicate<Int>) = p(42)

fun main(args: Array<String>) {
    val f: (Int) -> Boolean = { it > 0 }
    println(foo(f)) // "true" 출력

    val p: Predicate<Int> = { it > 0 }
    println(listOf(1, -2).filter(p)) // "[1]" 출력
}
```

레퍼런스

키워드와 연산자

Hard 키워드

다음 토큰은 항상 키워드로 해석하며 식별자로 사용할 수 없다:

- `as`
  - 타입 변환 에 사용
  - 임포트 별칭 지정
- `as?` : 안전한 타입 변환 에 사용
- `break` : 루프 실행을 종료
- `class` : 클래스 선언
- `continue` : 가장 가까운 루프의 다음 단계를 진행
- `do` : do/while 루프 시작 (사후조건을 가진 루프)
- `else` : if 식 에서 조건이 false일 때 실행하는 브랜치 정의
- `false` : 불리언 타입 의 `false` 값
- `for` : for 루프 를 시작
- `fun` : 함수 선언
- `if` : if 식 시작
- `in`
  - for 루프 에서 반복할 객체를 지정
  - 콜렉션이나 'contains' 메서드를 정의한 다른 엔티티, 범위 에 값이 속하는지 검사하기 위한 중위 연산자로 사용
  - when 식 에서 같은 목적으로 사용
  - 반공변 으로 타입 파라미터 지정
- `!in`
  - 콜렉션이나 'contains' 메서드를 정의한 다른 엔티티, 범위 에 값이 속하지 않는지 검사하기 위한 중위 연산자로 사용
  - when 식 에서 같은 목적으로 사용
- `interface` : 인터페이스
- `is`
  - 값이 특정 타입인지 검사
  - when 식 에서 같은 목적으로 사용
- `!is`
  - 값이 특정 타입이 아닌지 검사
  - when 식 에서 같은 목적으로 사용
- `null` : 어떤 객체도 가리키지 않는 객체 레퍼런스를 표현하는 상수
- `object` : 클래스와 그것으 인스턴스를 동시에 선언
- `package` : 현재 파일을 위한 패키지 를 지정
- `return` 가장 가깝게 둘러싼 함수나 임의 함수에서 리턴
- `super`
  - 메서드나 프로퍼티의 상위클래스 구현을 참조
  - 보조 생성자에서 상위클래스의 생성자를 호출
- `this`

  - 현재 리시버 를 참조
  - 보조 생성자에서 같은 클래스의 다른 생성자를 호출
- `throw` : 익셉션을 발생
- `true` : 불리언 타입 의 'true' 값을 지정
- `try` : 익셉션 처리 블록을 시작
- `typealias` : 타입 별칭 을 선언
- `val` : 읽기 전용 프로퍼티 나 로컬 변수 를 선언
- `var` 수정 가능 프로퍼티 나 로컬 변수 를 선언
- `when` : when 식 시작(주어진 브랜치 중 하나 실행)
- `while` : while 루프 (전위조건을 가진 루프)

Soft 키워드

다음 토큰은 적용 가능한 문맥에서는 키워드로 쓰이고, 다른 문맥에서는 식별자로 사용할 수 있다:

- `by`
  - 인터페이스 구현을 다른 객체에 위임
  - 프로퍼티를 위한 접근자 구현을 다른 객체에 위임
- `catch` : 특정 익셉션 타입을 처리하는 블록 시작
- `constructor` : 주요 또는 보조 생성자 선언
- `delegate` : 애노테이션 사용 위치 대상 으로 사용
- `dynamic` : 코틀린/JS 코드에서 동적 타입 을 참조
- `field` : 애노테이션 사용 위치 대상 으로 사용
- `file` : 애노테이션 사용 위치 대상 으로 사용
- `finally` : try 블록이 끝날 때 항상 실행하는 블록 시작
- `get`
  - 프로퍼티의 getter 선언
  - 애노테이션 사용 위치 대상 으로 사용
- `import` : 현재 파일에 다른 패키지의 선언을 임포트
- `init` : 초기화 블록 시작
- `param` : 애노테이션 사용 위치 대상 으로 사용
- `property` : 애노테이션 사용 위치 대상 으로 사용
- `receiver` : 애노테이션 사용 위치 대상 으로 사용
- `set`
  - 프로퍼티 setter 선언
  - 애노테이션 사용 위치 대상 으로 사용
- `setparam` : 애노테이션 사용 위치 대상 으로 사용
- `where` : 지네릭 타입 파라미터에 대한 제약 지정

수식어 키워드

다음 토큰은 선언의 수식어 목록에서 키워드로 사용하며, 다른 문맥에서는 식별자로 사용할 수 있다:

- `abstract` : 클래스나 멤버를 추상 으로 표시
- `annotation` : 애노테이션 클래스 선언
- `companion` : 컴패니언 오브젝트 선언
- `const` : 컴파일 타임 상수 로 프로퍼티 표시
- `crossinline` : 인라인 함수에 전달한 람다에서 비-로컬 리턴 금지
- `data` : 컴파일러가 클래스를 위한 canonical 멤버를 생성하도록 지시
- `enum` : 열거형 선언
- `external` : 선언을 코틀린이 아닌 구현으로 표시(JNI 로 접근 가능하거나 자바스크립트 에 구현한 것)

- `final` : 멤버 오버라이딩 금지
- `infix` : 중위 표기법 함수 호출 허용
- `inline` : 호출 위치에서 전달한 함수와 람다를 인라인하도록 컴파일러에게 말함
- `inner` : 중첩 클래스 에서 외부 클래스의 인스턴스 참조를 허용
- `internal` : 선언을 현재 모듈에 보이도록 표시
- `lateinit` : 생성자 밖에서 non-null 프로퍼티를 초기화하는 것을 허용
- `noinline` : 인라인 함수에 전달한 람다를 인라인하지 않음
- `open` : 클래스 상속 또는 멤버 오버라이딩 을 허용
- `operator` : 함수를 연사자 오버로딩이나 컨벤션 구현으로 표시
- `out` : 타입 파라미터를 공변 으로 표시
- `override` : 상위클래스 멤버의 오버라이드 로 멤버를 표시
- `private` : 선언을 현재 클래스나 파일에 보이도록 표시
- `protected` : 선언을 현재 클래스와 그 클래스의 하위클래스에 보이도록 표시
- `public` : 선언을 모든 곳에 보이도록 표시
- `reified` : 인라인 함수의 타입 파라미터를 런타임에서 접근가능하게 표시
- `sealed` : sealed 클래스 선언 (제한된 하위클래스를 갖는 클래스)
- `suspend` : 함수나 람다를 서스펜딩으로 표시(코루틴 으로 사용가능)
- `tailrec` : 꼬리 재귀 함수로 표시 (컴파일러가 재귀를 반복으로 바꾸도록 허용)
- `vararg` : allows 파라미터를 가변 인자로 전달할 수 있게 허용

특수 식별자

컴파일러는 특정한 문맥에서 다음 식별자를 정의하며, 다른 문백에서는 일반 식별자로 사용할 수 있다:

- `field` : 프로퍼티 접근자에서 프로퍼티의 지원 필드 를 참조하기 위해 사용
- `it` : 람다 안에서 명시적 선언 없이 파라미터에 접근하기 위해 사용

연산자와 특수 문자

코틀린은 다음 연산자와 특수 문자를 지원한다:

- `+` , `-` , `*` , `/` , `%` - 수치 연산자
  - `*` : 가변 인자에 배열을 전달 할 때에도 사용
- `=`
  - 할당 연산자
  - 파라미터의 기본 값 을 지정할 때 사용
- `+=` , `-=` , `*=` , `/=` , `%=` - augmented 할당 연산자
- `++` , `--` - 증가 감소 연산자
- `&&` , `||` , `!` - 논리 'and', 'or', 'not' 연산자 (비트 연산자의 경우, 대응하는 중의 연산자 를 사용)
- `==` , `!=` - 동등비교 연산자 (기본 타입이 아닌 경우 `equals()` 호출로 변환)
- `===` , `!==` - 참조 동일비교 연산자
- `<` , `>` , `<=` , `>=` - 비교 연산자 (기본 타입이 아닌 경우 `compareTo()` 호출로 변환)
- `[` , `]` - 인덱스기반 접근 연산자 (`get` 과 `set` 호출로 변환)
- `!!` : 식이 null이 아님을 단언한다
- `?.` : 안전한 호출 수행 (리시버가 null이 아니면 메서드를 호출하거나 프로퍼티에 접근)
- `?:` : 왼쪽 값이 null이면 오른쪽 값을 취한다(엘비스 연산자)
- `::` : 멤버 레퍼런스 나 클래스 레퍼런스 생성
- `..` : 범위 생성
- `:` : 선언에서 타입과 이름을 구분
- `?` : 타입을 null 가능 으로 표시
- `->`
  - 람다 식 의 파라미터와 몸체를 분리

    - 함수 타입 에서 파라미터와 리턴 타입 선언을 분리
    - when 식 브랜치에서 조건과 몸체를 분리
- `@`
    - 애노테이션 시작
    - 루프 라벨 시작 또는 참조
    - 람다 라벨 시작 또는 참조
    - 외부 범위에서 'this' 식 을 참조
    - 외부 상위클래스 를 참조
- `;` : 같은 줄의 여러 문장을 구분
- `$` : 문자열 템플릿 에서 변수나 식을 참조
- `_`
    - 람다 식 에서 사용하지 않는 파라미터 대체
    - 분리 선언 에서 사용하지 않는 파라미터 대체

호환성

이 문서는 코틀린의 하위시스템과 다른 버전 간에 보장하는 호환성을 설명한다.

호환성 용어

주어진 두 코틀린 버전(예, 1.2와 1.1.5)에 대해 한 버전으로 작성한 코드를 다른 버전에서 사용할 수 있나? 호환성은 이 질문에 답하는 것을 의미한다. 아래 목록은 다른 버전 간의 호환성 모드를 설명한다. 버전 숫자가 작으면 (비록 큰 버전 숫자보다 작은 버전이 나중에 나왔어도) 오래된 버전이다. "오래된 버전(Older Version)"에 OV를 사용하고 "새로운 버전(Newer Version)"에 대해 NV를 사용한다.

- **C** - 완전환 호환성(**C** ompatibility)
  - 언어
    - 구문 변화 없음 (*모듈 버그* *)
    - 신규 경고/힌트를 추가하거나 제거할 수 있음
  - API (`kotlin-stdlib-*` , `kotlin-reflect-*`)
    - API 변경 없음
    - `WARNING` 수준의 디프리케이션을 추가하거나 제거할 수 있음
  - 바이너리 (ABI)
    - 런타임: 바이너리를 교환해서 사용할 수 있음
    - 컴파일: 바이너리를 교환해서 사용할 수 있음
- **BCLA** - 언어 API에 대한 하위 호환(**B** ackward **C** ompatibility for the **L** anguage and **A** PI)
  - 언어
    - OV에서 디프리케이션한 구문을 NV에서 제거할 수 있음
    - 이 외에 OV에 호환되는 모든 코드는 NV에 호환(modulo bugs*)
    - NV에 새 구문을 추가할 수 있음
    - OV의 제약이 NV에서 완화될 수 있음
    - 신규 경고/힌트를 추가하거나 제거할 수 있음
  - API (`kotlin-stdlib-*` , `kotlin-reflect-*`)
    - 새 API를 추가할 수 있음
    - `WARNING` 수준의 디프리케이션을 추가하거나 제거할 수 있음
    - `WARNING` 수준의 디프리케이션이 NV에서 `ERROR` 나 `HIDDEN` 으로 등급이 올라갈 수 있음
- **BCB** - 바이너리에 대한 하위 호환(**B** ackward **C** ompatibility for **B** inaries)
  - 바이너리 (ABI)
    - 런타임: OV 바이너리가 동작하는 모든 곳에 NV-바이너리를 사용할 수 있음
    - NV 컴파일러: OV 바이너리에 대한 컴파일 가능한 코드는 NV 바이너리에 대해 호환
    - OV 컴파일러는 NV 바이너리를 수용하지 않을 수 있음 (예, 새로운 언어 특징이 API가 존재하는 경우)
- **BC** - 완전한 하위 호환(**B** ackward **C** ompatibility)
  - BC = BCLA & BCB
- **EXP** - 실험 특징
  - 아래 참고
- **NO** - 호환성 보장하지 않음
  - 마이그레이션이 잘 되도록 최선을 다하지만, 어떤 보장도 하지 않음
  - 모든 비호환 하위시스템에 대해 개별적으로 마이그레이션 계획

---

* 변화 없음 *modulo bugs* 는 중요한 버그가 발견되면(예, 컴파일러 진단이나 다른 곳에서) 버그를 고쳤을 때 호환성을 깨는 변경이 발생할 수도 있으므로 그런 변경에 매우 주의한다는 것을 의미한다.

코틀린 릴리즈 호환성

JVM을 위한 코틀린 :

— 패치 버전 업데이트 (예 1.1.X)는 완전히 호환

— 마이너 버전 업데이트 (예 1.X)는 하위 호환

Kotlin	1.0	1.0.X	1.1	1.1.X	…	2.0
1.0	-	C	BC	BC	…	?
1.0.X	C	-	BC	BC	…	?
1.1	BC	BC	-	C	…	?
1.1.X	BC	BC	C	-	…	?
…	…	…	…	…	…	…
2.0	?	?	?	?	…	-

JS를 위한 코틀린 : 코틀린 1.1부터, 패치와 마이너 버전 업데이트에 대해 언어와 API(BCLA)에 대한 하위 호환성을 제공하지만, BCB는 제공하지 않는다.

Kotlin	1.0.X	1.1	1.1.X	…	2.0
1.0.X	-	EXP	EXP	…	EXP
1.1	EXP	-	BCLA	…	?
1.1.X	EXP	BCLA	-	…	?
…	…	…	…	…	…
2.0	EXP	?	?	…	-

코틀린 스크립트 : 패치와 마이너 버전 업데이트에 대해 언어와 API(BCLA)에 대한 하위 호환성을 제공하지만, BCB는 제공하지 않는다.

플랫폼 간 호환성

여러 플랫폼(JVM/안드로이드, 자바스크립트와 앞으로 지원할 네이티브 플랫폼)에서 코틀린을 사용할 수 있다. 모든 플랫폼은 자신만의 특성을 갖고 있어서(예, 자바스크립트는 정수가 없다), 언어에 알맞게 맞춰야 한다. 우리의 목적은 많은 희생없이 합리적인 코드 이식성을 제공하는 것이다.

모든 플랫폼은 특정한 언어 확장(JVM의 플랫폼 타입과 자바스크립트의 동적 타입)이나 제약(예, JVM에서의 일부 오버로딩 관련 제약)을 갖지만, 핵심 언어는 동일하게 유지한다.

표준 라이브러리는 모든 플랫폼에서 가능한 핵심 API를 제공하며, 이 API가 모든 플랫폼에서 동일하게 동작하도록 만들기 위해 노력했다. 이와 함께, 표준 라이브러리는 플랫폼에 특화된 확장(예, JVM을 위한 `java.io` 와 자바스크립트를 위한 `js()`)과 추가로 동일하게 호출할 수 있지만 다르게 동작하는 일부 API(예, JVM과 자바스크립트의 정규 표현식)를 제공한다.

실험 특징

코틀린 1.1의 코루틴과 같은 실험 특징은 아래 표시한 호환성 모드에서 제외된다. 이런 특징을 컴파일러 경고없이 사용하려면 사전 동의(opt-in)를 요구한다. 실험 특징은 피치 버전 업데이트에 대해서는 하위 호환성을 유지하지만, 마이너 버전 업데이트에서는 어떤 호환성도 보장하지 않는다(가능하면 마이그레이션 도구를 제공할 것이다);

Kotlin	1.1	1.1.X	1.2	1.2.X
1.1	-	BC	NO	NO
1.1.X	BC	-	NO	NO
1.2	NO	NO	-	BC
1.2.X	NO	NO	BC	-

EAP 빌드

EAP(Early Access Preview)를 특수 채널로 제공하는데 이 채널을 통해 커뮤니티의 얼리 어댑터가 시도해보고 피드백을 줄 수 있다. EAP 빌드는 (각 릴리즈마다 알맞게 호환성을 유지하게 위해 노력은 하겠지만) 어떤 호환성 보장도 하지 않는다. EAP 빌드에 대한 품질 기대 수준은 정식 릴리즈보다 매우 낮다. 베타 빌드도 이 부류에 속한다.

중요 사항 : 컴파일러의 릴리즈 빌드는 1.x의 EAP 빌드(예, 1.1.0-eap-X)로 컴파일한 모든 바이너리를 거부한다. 안정된 버전을 릴리즈한 이후에 이전 버전으로 컴파일한 어떤 코드도 유지하길 원치 않기 때문이다. 패치 버전의 EAP(예, 1.1.3-eap-X)는 이와 상관없으며 이 EAP는 안정한 ABI를 사용해서 빌드를 생성할 수 있다.

호환성 모드

큰 팀이 새 버전으로 마이그레이션을 할 때, 일부 개발자가 이미 업데이트했는데 다른 개발자는 하지 않아서, 어떤 시점에 "깨진 상태(inconsistent state)"가 될 수 있다. 다른 개발자가 컴파일할 수 없는 코드를 작성하고 커밋하는 것을 방지하기 위해, 다음 명령행 스위치를 제공한다(IDE와 그레이들 / 메이븐 에서도 사용 가능하다).

— `-language-version X.Y` - 코틀린 언어 X.Y 버전에 대한 위한 호환성 모드. 이후에 나온 모든 언어 특징에 대해 에러를 발생한다.

— `-api-version X.Y` - 코틀린 API X.Y 버전에 대한 호환성 모드. 코틀린 표준 라이브러리에서 신규 API를 사용하는 모든 코드에 대해(컴파일러가 생성한 코드 포함) 에러를 발생한다.

바이너리 호환성 경고

NV 코틀린 컴파일러를 사용하는데 클래스패스에 OV 표준 라이브러리나 OV 리플레션 라이브러리가 존재하면, 프로젝트를 잘못 설정했다는 신호일 수 있다. 컴파일이나 런타임에 원치 않는 문제가 발생하는 것을 막으려면, 의존을 NV로 업데이트하거나 API 버전/언어 버전 인자를 명시적으로 지정할 것을 권한다. 그렇지 않을 경우 컴파일러는 뭔가 잘못될 수 있다는 것을 탐지하고 경고를 발생한다.

예를 들어, OV = 1.0 이고 NV = 1.1 이면, 다음 경고 중 하나를 볼 수 있다:

```
Runtime JAR files in the classpath have the version 1.0, which is older than the API version 1.1.
Consider using the runtime of version 1.1, or pass '-api-version 1.0' explicitly to restrict the
available APIs to the runtime of version 1.0.
```

이는 1.0 버전의 표준 또는 리플렉션 라이브러리로 코틀린 1.1 컴파일러를 사용한다는 것을 의미한다. 이 경고는 몇 가지 방법으로 처리할 수 있다:

- 1.1 표준 라이브러리의 API나 그 API에 의존한 언어 특징을 사용하길 원한다면, 의존 버전을 1.1로 올려야 한다.
- 1.0 표준 라이브러리와 호환되도록 코드를 유지하고 싶다면, `-api-version 1.0` 를 전달한다.
- 코틀린은 1.1로 업그레이드는 하지만 새 언어 특징은 사용할 수 없다면(예, 일부 팀원이 아직 업그레이드 전인 경우), 모든 API와 언어 특징을 1.0으로 제한하기 위해 `-language-version 1.0` 를 전달할 수 있다.

```
Runtime JAR files in the classpath should have the same version. These files were found in the classpath:
    kotlin-reflect.jar (version 1.0)
    kotlin-stdlib.jar (version 1.1)
Consider providing an explicit dependency on kotlin-reflect 1.1 to prevent strange errors
Some runtime JAR files in the classpath have an incompatible version. Consider removing them from the
classpath
```

이 메시지는 다른 버전의 라이브러리에 대한 의존이 존재함을 의미한다. 예를 들어, 1.1 표준 라이브러리와 1.0 리플렉션 라이브러리에 의존이 존재하는 경우이다. 런타임에 발생하는 미묘한 에러를 막으려면, 모든 코틀린 라이브러리가 같은 버전을 사용할 것을 권한다. 이 예의 경우 1.1 리플렉션 라이브러리를 의존에 명시적으로 추가할 것을 고려한다.

```
Some JAR files in the classpath have the Kotlin Runtime library bundled into them.
This may cause difficult to debug problems if there's a different version of the Kotlin Runtime library in
the classpath.
Consider removing these libraries from the classpath
```

이 메시지는 클래스패스에 코틀린 표준 라이브러리에 의존하지 않는 라이브러리가 그레이들/메이븐 의존으로 존재하는데, 그 라이브러리가 표준 라이브러리를 포함한다는 것을(예, _번들_한 경우) 의미한다.
표준 빌드 도구는 이런 라이브러리를 코틀린 표준 라이브러리의 인스턴스로 고려하지 않고, 따라서 의존 버전 분석 메커니즘에 적용을 받지 않으며, 결과적으로 클래스패스에 동일 라이브러리의 여러 버전을 가질 수 있다. 그래서 이런 라이브러리는 문제를 발생할 수 있다. 라이브러리 제작자에 연락해서 그레이들/메이븐 의존을 대신 사용해달라고 제안하는 것을 고려한다.

자바 상호운용

코틀린에서 자바 호출하기

코틀린은 자바와의 상호운용성을 염두에 두고 설계했다. 기존 자바 코드를 코틀린에서 자연스럽게 호출할 수 있고, 자바 코드에서도 비교적 부드럽게 코틀린 코드를 사용할 수 있다. 이 절에서는 코틀린에서 자바 코드를 호출하는 것에 대해 자세히 살펴본다.

아무 문제 없이 거의 모든 자바 코드를 사용할 수 있다:

```
import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // 자바 콜렉션에 대해 'for'-루프 동작
    for (item in source) {
        list.add(item)
    }
    // 연산자 규칙 또한 동작:
    for (i in 0..source.size - 1) {
        list[i] = source[i] // get과 set 호출됨
    }
}
```

Getters와 Setters

getter와 setter를 위한 자바 규칙(이름이 `get` 으로 시작하는 인자 없는 메서드와 이름이 `set` 으로 시작하고 한 개 인자를 갖는 메서드)을 따르는 메서드는 코틀린에서 프로퍼티로 표현된다. `Boolean` 접근 메서드(getter 이름이 `is` 로 시작하고 setter 이름은 `set` 으로 시작)는 getter 메서드와 같은 이름을 갖는 프로퍼티로 표현한다.

예제:

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) {  // getFirstDayOfWeek() 호출
        calendar.firstDayOfWeek = Calendar.MONDAY      // setFirstDayOfWeek() 호출
    }
    if (!calendar.isLenient) {                         // isLenient() 호출
        calendar.isLenient = true                      // setLenient() 호출
    }
}
```

자바 클래스에 setter만 존재하면 코틀린에서 프로퍼티로 보이지 않는다. 왜냐면 현재 코틀린은 쓰기 전용 프로퍼티를 지원하지 않기 때문이다.

void를 리턴하는 메서드

void를 리턴하는 자바 메서드를 코틀린에서 호출하면 `Unit` 을 리턴한다. 만약 리턴 값을 사용하면, 값 자체를 미리 알 수 있기 때문에(`Unit`) 코틀린 컴파일러는 호출 위치에 값을 할당한다.

코틀린에서 키워드인 자바 식별자 처리

`in` , `object` , `is` 등 일부 코틀린 키워드는 자바에서 유효한 식별자이다. 자바 라이브러리가 코틀린 키워드를 메서드 이름으로 사용할 경우, 역따옴표를 탈출 문자로 사용해서 메서드를 호출할 수 있다:

```
foo.`is`(bar)
```

null 안정성과 플랫폼 타입

자바에서 모든 레퍼런스는 `null` 일 수 있는데, 이는 자바에서 온 객체에 대해 코틀린의 엄격한 null 안정성 요구를 비실용적으로 만든다. 코틀린에서는 *플랫폼 타입* 이라고 불리는 자바 선언 타입으로 별도 처리한다. 그 타입에 대한 null 검사를 완화해서 안정성 보장을 자바와 동일하게 한다(아래 참고)

다음 예를 보자:

```
val list = ArrayList<String>() // non-null (생성자 결과)
list.add("Item")
val size = list.size // non-null (기본 int)
val item = list[0] // 플랫폼 타입 유추 (일반 자바 객체)
```

플랫폼 타입 변수의 메서드를 호출할 때, 코틀린은 컴파일 시점에 null 가능성 에러를 발생하지 않는다. 대신 코틀린이 null이 전파되는 것을 막기 위해 생성한 단언이나 널포인터 익셉션 때문에 런타임에 호출이 실패할 수 있다.

```
item.substring(1) // 허용, 단 item이 null이면 익셉션이 발생
```

플랫폼 타입은 *non-denotable* 인데, 이는 언어에서 그 타입을 바로 사용할 수 없다는 것을 의미한다. 플랫폼 값을 코틀린 변수에 할당하면, 타입 추론에 의존하거나(위 예에서 `item` 은 추론한 플랫폼 타입을 가짐) 또는 기대하는 타입을 선택할 수 있다(null 가능과 non-null 타입 둘 다 허용).

```
val nullable: String? = item // 허용, 항상 동작
val notNull: String = item // 허용, 런타임에 실패 가능
```

non-null 타입을 선택하면 컴파일러는 할당 위치에 단언을 생성(emit)한다. 이는 코틀린의 non-null 변수가 null 값을 갖는 것을 막아준다. 단언은 또한 non-null 값을 기대하는 코틀린 함수에 플랫폼 값을 전달할 때에도 생성된다. 전반적으로 컴파일러는 null이 프로그램에 퍼지는 것을 막기 위해 최선을 다한다(지네릭 때문에 완전히 없애는 것은 불가능하다).

플랫폼 타입을 위한 표기

위에서 언급한 것처럼, 플랫폼 타입을 프로그램에서 직접 언급할 수 없기 때문에 언어에 플랫폼 타입을 위한 구문이 없다. 그럼에도 불구하고 컴파일러와 IDE는 때때로 플랫폼 타입을 표시해야 할 때가 있으며(에러 메시지에서나 파라미터 정보 등), 따라서 이를 위한 기억용 표기를 제공한다:

— `T!` 은 " `T` 또는 `T?` "를 의미한다.

— `(Mutable)Collection<T>!` 은 " `T` 의 자바 콜렉션이 수정가능하거나 불변일 수 있고, null 가능이거나 non-null일 수 있다"는 것을 의미한다.

— `Array<(out) T>!` 은 " `T` 의(또는 `T` 의 하위타입의) 자바 배열이 null 가능이거나 아닐 수 있다"를 의미한다.

널가능성 애노테이션

널가능성(nullability) 애노테이션을 가진 자바 타입은 플랫폼 타입이 아닌 실제 null 가능 또는 not-null 코틀린 타입으로 표현한다. 컴파일러는 다음을 포함한 여러 널가능성 애노테이션을 지원한다:

— JetBrains (`@Nullable` and `@NotNull` from the `org.jetbrains.annotations` package)

— Android (`com.android.annotations` and `android.support.annotations`)

— JSR-305 (`javax.annotation`)

— FindBugs (`edu.umd.cs.findbugs.annotations`)

— Eclipse (`org.eclipse.jdt.annotation`)

— Lombok (`lombok.NonNull`).

전체 목록은 코틀린 컴파일러 소스 코드 에서 찾을 수 있다.

JSR-305 지원

JSR-305 에 정의된 `@Nonnull` 애노테이션은 자바 타입의 널가능성을 표시하기 위해 지원한다.

`@Nonnull(when = ...)` 값이 `When.ALWAYS` 이면, 애노테이션이 붙은 타입을 non-null로 다룬다. `When.MAYBE` 와 `When.NEVER` 는 null 가능 타입을 표시한다. `When.UNKNOWN` 은 타입을 플랫폼 타입 으로 강제한다.

라이브러리를 JSR-305 애노테이션에 기대어 컴파일할 수 있지만, 라이브러리리 사용자를 위한 컴파일 의존에 애노테이션 아티팩트(예, `jsr305.jar`)를 포함시킬 필요는 없다. 코틀린 컴파일러는 클래스패스에 존재하는 JSR-305 애노테이션을 읽을 수 있다.

코틀린 1.1.50 부터, 커스텀 널가능성 한정자(KEEP-79) 를 지원한다(아래 참고).

타입 한정자 별명 (1.1.50부터)

애노테이션 타입에 `@TypeQualifierNickname`과 JSR-305 `@Nonnull`(또는 이것의 다른 이름인 `@CheckForNull`와 같은 것)을 함께 붙이면, 정확한 널가능성을 검색하기 위해 그 애노테이션 타입 자체를 사용할 수 있고, 적용한 널가능성 애노테이션과 동일한 의미를 갖는다:

```
@TypeQualifierNickname
@Nonnull(when = When.ALWAYS)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNonnull {
}

@TypeQualifierNickname
@CheckForNull // 다른 타입 한정 별칭에 대한 별칭
@Retention(RetentionPolicy.RUNTIME)
public @interface MyNullable {
}

interface A {
    @MyNullable String foo(@MyNonnull String x); // `fun foo(x: String): String?`로 보임
    String bar(List<@MyNonnull String> x);       // `fun bar(x: List<String>!): String!`로 보임
}
```

타입 한정자 디폴트 (1.1.50부터)

`@TypeQualifierDefault`는 새로운 애노테이션 도입을 허용하는데, 그 애노테이션을 적용할 때 애노테이션을 붙인 요소의 범위 안에서 기본 널가능성을 정의한다.

그런 애노테이션 타입은 `@Nonnull`(또는 닉네임)과 한 개 이상의 `ElementType` 값을 갖는 `@TypeQualifierDefault(...)` 애노테이션을 붙여야 한다:

- `ElementType.METHOD` : 메서드 리턴 타입 대상
- `ElementType.PARAMETER` : 밸류 파라미터 대상
- `ElementType.FIELD` : 필드 대상

타입 자체에 널가능성 애노테이션이 없을 때 기본 널가능성을 사용하며, 타입 용도와 일치하는 `ElementType`을 가진 @TypeQualifierDefault을 붙인 가장 안쪽을 둘러싼 요소가 기본 널가능성을 결정한다.

```
@Nonnull
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER})
public @interface NonNullApi {
}

@Nonnull(when = When.MAYBE)
@TypeQualifierDefault({ElementType.METHOD, ElementType.PARAMETER})
public @interface NullableApi {
}

@NullableApi
interface A {
    String foo(String x); // fun foo(x: String?): String?

    @NotNullApi // 인터페이스로부터 기본을 오버라이딩
    String bar(String x, @Nullable String y); // fun bar(x: String, y: String?): String

    // The type of `x` 파라미터 타입은 플랫폼으로 남는다.
    // 이유는 널가능성 애노테이션 값을 UNKNOWN으로 표시했기 때문이다.
    String qux(@Nonnull(when = When.UNKNOWN) String x); // fun baz(x: String!): String?
}
```

패키지 수준의 기본 널가능성 또한 지원한다:

```
// FILE: test/package-info.java
@NonNullApi // 'test' 패키지에 있는 모든 타입의 기본 널가능성을 non-nullable로 선언한다
package test;
```

컴파일러 설정

다음 값 중 하나를 갖는 `-Xjsr305` 컴파일러 플래그를 추가해서 JSR-305 검사를 설정할 수 있다:

— `-Xjsr305=strict` 은 JSR-305 애노테이션을 일반 널가능성 애노테이션으로 동작하게 만들고, 애노테이션 타입을 알맞게 사용하지 않으면 에러를 발생한다.

— `-Xjsr305=warn` 은 알맞지 않은 사용에 대해 에러 대신 컴파일 경고를 낸다.

— `-Xjsr305=ignore` 는 컴파일러가 JSR-305 널가능성 애노테이션을 완전히 무시하게 한다.

코틀린 1.1.50+/1.2 버전의 기본 동작은 `-Xjsr305=warn` 과 같다. `strict` 값은 실험적으로만 고려해야 한다(향후에 더 다양한 검사를 추가할 것이다).

매핑된 타입

코틀린은 일부 자바 타입을 특별하게 다룬다. 그런 타입은 자바 타입을 "그대로"로 로딩하지 않고 대응하는 코틀린 타입으로 _매핑_한다. 매핑은 컴파일 시점에만 중요하며 런타임 표현은 바뀌지 않고 유지한다. 자바 기본 타입은 대응하는 코틀린 타입으로 매핑된다(플랫폼 타입 을 염두한다).

자바 타입	코틀린 타입
`byte`	`kotlin.Byte`
`short`	`kotlin.Short`
`int`	`kotlin.Int`
`long`	`kotlin.Long`
`char`	`kotlin.Char`
`float`	`kotlin.Float`
`double`	`kotlin.Double`
`boolean`	`kotlin.Boolean`

일부 기본 타입이 아닌 내장 클래스도 매핑한다:

자바 타입	코틀린 타입
`java.lang.Object`	`kotlin.Any!`
`java.lang.Cloneable`	`kotlin.Cloneable!`
`java.lang.Comparable`	`kotlin.Comparable!`
`java.lang.Enum`	`kotlin.Enum!`
`java.lang.Annotation`	`kotlin.Annotation!`
`java.lang.Deprecated`	`kotlin.Deprecated!`
`java.lang.CharSequence`	`kotlin.CharSequence!`
`java.lang.String`	`kotlin.String!`
`java.lang.Number`	`kotlin.Number!`
`java.lang.Throwable`	`kotlin.Throwable!`

자바의 박싱된 기본 타입은 null 가능 코틀린 타입으로 매핑한다:

자바 타입	코틀린 타입
`java.lang.Byte`	`kotlin.Byte?`
`java.lang.Short`	`kotlin.Short?`
`java.lang.Integer`	`kotlin.Int?`
`java.lang.Long`	`kotlin.Long?`
`java.lang.Character`	`kotlin.Char?`
`java.lang.Float`	`kotlin.Float?`
`java.lang.Double`	`kotlin.Double?`
`java.lang.Boolean`	`kotlin.Boolean?`

타입 파라미터로 박싱된 기본 타입을 사용하면 플랫폼 타입으로 매핑된다. 예를 들어 `List<java.lang.Integer>` 는 코틀린에서 `List<Int!>` 가 된다.

콜렉션 타입은 코틀린에서 읽기 전용 타입과 변경가능 타입이 존재하므로 자바 콜렉션을 다음과 같이 변경한다(이 표에서 모든 코틀린 타입은 `kotlin.collections` 패키지에 위치한다):

자바 타입	코틀린 읽기 전용 타입	코틀린 수정가능 타입	로딩된 플랫폼 타입
`Iterator<T>`	`Iterator<T>`	`MutableIterator<T>`	`(Mutable)Iterator<T>!`
`Iterable<T>`	`Iterable<T>`	`MutableIterable<T>`	`(Mutable)Iterable<T>!`
`Collection<T>`	`Collection<T>`	`MutableCollection<T>`	`(Mutable)Collection<T>!`

자바 타입	코틀린 읽기 전용 타입	코틀린 수정가능 타입	로딩된 플랫폼 타입
List<T>	List<T>	MutableList<T>	(Mutable)List<T>!
ListIterator<T>	ListIterator<T>	MutableListIterator<T>	(Mutable)ListIterator<T>!
Map<K, V>	Map<K, V>	MutableMap<K, V>	(Mutable)Map<K, V>!
Map.Entry<K, V>	Map.Entry<K, V>	MutableMap.MutableEntry<K,V>	(Mutable)Map.(Mutable)Entry<K, V>!

자바 배열은 아래 에서 언급한 것처럼 매핑된다:

Java type	Kotlin type
int[]	kotlin.IntArray!
String[]	kotlin.Array<(out) String>!

코틀린에서의 자바 지네릭

코틀린 지네릭은 자바 지네릭과 조금 다르다(지네릭 을 보자). 자바 타입을 코틀린에 임포트할 때 일부 변환을 수행한다:

- 자바의 와일드카드는 타입 프로젝션으로 변환
  - `Foo<? extends Bar>` 는 `Foo<out Bar!>!` 가 됨
  - `Foo<? super Bar>` 는 `Foo<in Bar!>!` 가 됨
- 자바 원시 타입은 스타 프로젝션으로 변환
  - `List` 는 `List<*>!` 가 된다 (예를 들어, `List<out Any?>!`)

자바와 마찬가지로 코틀린 지네릭도 런타임에 유지되지 않는다. 예를 들어, 객체는 생성자에 전달된 실제 타입 인자에 대한 정보를 담지 않으므로 `ArrayList<Integer>()` 와 `ArrayList<Character>()` 를 구분할 수 없다. 이는 지네릭을 고려한 `is` 를 수행하기 어렵게 만든다. 코틀린은 오직 스타 프로젝션인 지네릭 타입에 대한 `is` 만 허용한다:

```
if (a is List<Int>) // 에러: 실제로 Int 리스트인지 확인할 수 없다
// 하지만
if (a is List<*>) // OK: List 내용에 대한 보장을 하지 않음
```

자바 배열

코틀린 배열은 자바와 달리 무공변이어서 발생 가능한 런타임 실패를 방지한다. 상위타입 배열 파라미터를 사용하는 코틀린 메서드에서 하위타입 배열을 전달하는 것을 금지한다. 자바 메서드의 경우 이를 허용한다(`Array<(out) String>!` 형식의 [플랫폼 타입).

박싱/언박싱 오퍼레이션 비용을 없애기 위해 자바 플랫폼의 기본 데이터 타입을 가진 배열을 사용한다. 코틀린은 이런 구현 상세를 감추기 때문에 자바 코드를 사용하려면 다른 방법이 필요하다. 이런 경우를 다루기 위해 모든 기본 타입 배열을 위한 특수 클래스(`IntArray` , `DoubleArray` , `CharArray` 등)를 제공하고 있다. 특수 클래스는 `Array` 클래스와 상관없으며 최대 성능을 위해 자바의 기본타입 배열로 컴파일된다.

다음과 같이 int 타입 배열인 indices를 받는 자바 메서드가 있다고 하자:

```
public class JavaArrayExample {

    public void removeIndices(int[] indices) {
        // 여기 코드 ...
    }
}
```

다음과 같이 코틀린에서 기본타입 값을 갖는 배열을 전달할 수 있다:

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndices(array)  // 메서드에 int[] 전달
```

JVM 바이트 코드로 컴파일할 때, 컴파일러는 배열에 대한 접근을 최적화해서 오버헤드가 발생하지 않는다:

```
val array = arrayOf(1, 2, 3, 4)
array[x] = array[x] * 2 // get()과 set()에 대한 실제 호출이 없음
for (x in array) { // 이터레이터 만들지 않음
    print(x)
}
```

인덱스를 갖고 조작할 때 조차도 오버헤드가 발생하지 않는다:

```
for (i in array.indices) { // 이터레이터 만들지 않음
    array[i] += 2
}
```

마지막으로 `in` 검사도 오베헤드가 없다::

```
if (i in array.indices) { // (i >= 0 && i < array.size)와 동일
    print(array[i])
}
```

자바 가변인자

자바 클래스는 종종 가변 인자를 가진 메서드 선언을 사용한다(varars):

```
public class JavaArrayExample {

    public void removeIndicesVarArg(int... indices) {
        // 코드 사용
    }
}
```

이 경우 `IntArray` 에 `*` 연산자(펼침 연산자)를 사용해서 배열을 전달할 수 있다:

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndicesVarArg(*array)
```

현재 varargs로 선언한 메서드에 `null` 을 전달할 수는 없다.

연산자

자바는 연산자 구문을 사용하기에 알맞은 메서드 표시 방법이 없이 때문에, 코틀린은 연산자 오버로딩과 다른 규칙(`invoke()` 등)에 맞는 이름과 시그너처를 가진 자바 메서드를 허용한다. 중위 호출 구문을 사용해서 자바 메서드를 호출하는 것은 허용하지 않는다.

Checked 익셉션

코틀린에서 모든 익셉션은 unchecked다. 이는 컴파일러가 익셉션 처리(catch)를 강제하지 않음을 의미한다. 따라서 checked 익셉션을 선언한 자바 메서드를 호출할 때 코틀린은 어떤 것도 강제하지 않는다.

```
fun render(list: List<*>, to: Appendable) {
    for (item in list) {
        to.append(item.toString()) // 자바는 IOException을 catch할 것을 요구한다
    }
}
```

Object 메서드

자바 타입을 코틀린으로 임포트할 때, `java.lang.Object` 타입의 모든 레퍼런스는 `Any` 로 바뀐다. `Any` 는 플랫폼에 종속되지 않기 때문에, 멤버로 `toString()` , `hashCode()` , `equals()` 만 선언하고 있다. `java.lang.Object` 의 다른 멤버를 사용하려면 확장 함수 를 사용한다.

wait()/notify()

Effective Java Item 69는 `wait()` 와 `notify()` 보다 병행 유틸리티를 쓰라고 제안한다. 그래서 `Any` 타입 레퍼런스에는 이 두 메서드를 사용할 수 없다. 실제로 이 두 메서드를 호출하고 싶다면 `java.lang.Object` 로 변환해서 호출할 수는 있다:

```
(foo as java.lang.Object).wait()
```

getClass()

객체의 자바 클래스를 구하려면, 클래스 레퍼런스 의 `java` 확장 프로퍼티를 사용한다:

```
val fooClass = foo::class.java
```

위 코드는 코틀린 1.1부터 지원하는 객체에 묶인 클래스 레퍼런스 로, 대신 `javaClass` 확장 프로퍼티를 사용할 수도 있다:

```
val fooClass = foo.javaClass
```

clone()

`clone()` 을 오버라이딩하려면, `kotlin.Cloneable` 를 확장해야 한다:

```
class Example : Cloneable {
    override fun clone(): Any { ... }
}
```

Effective Java , Item 11: *Override clone judiciously* 를 잊지 말자.

finalize()

`finalize()` 를 오버라이딩 하려면, `override` 키워드 없이 메서드를 선언하면 된다:

```
class C {
    protected fun finalize() {
        // finalization logic
    }
}
```

자바 규칙에 따라, `finalize()` 는 `private` 이면 안 된다.

자바 클래스를 상속

코틀린에서 최대 한 개의 자바 클래스를 상속할 수 있다(자바 인터페이스는 필요한 만큼 많이 상속 가능).

정적 멤버에 접근

자바 클래스의 정적 멤버는 그 클래스를 위한 "컴페니언 오브젝트"로 구성한다. 그 "컴페니언 오브젝트"를 값으로 전달할 수 없지만, 멤버에 접근할 수는 있다:

```
if (Character.isLetter(a)) {
    // ...
}
```

자바 리플렉션

자바 리플렉션은 코틀린 클래스에 동작하며, 반대로도 동작한다. 위에서 언급한 것처럼 `instance::class.java` , `ClassName::class.java` , `instance.javaClass` 를 사용해서 자바 리플렉션에 필요한 `java.lang.Class` 를 구할 수 있다.

자바 getter/setter 메서드 또는 코틀린 프로퍼티를 위한 지원 필드 구하기, 자바 필드를 위한 `KProperty` 구하기, `KFunction` 을 위한 자바 메서드나 생성자 구하기 또는 그 반대 구하기 등을 지원한다.

SAM 변환

자바 8과 마찬가지로, 코틀린은 SAM 변환을 지원한다. 이는 코틀린 함수 리터럴을 한 개의 추상 메서드를 가진 자바 인터페이스 구현으로 자동으로 변환한다는 것을 뜻한다. 코틀린 함수의 파라미터 타입이 인터페이스 메서드의 파라미터 타입과 일치하면 된다.

SAM 인터페이스의 인스턴스를 생성할 때 이를 사용할 수 있다:

```
val runnable = Runnable { println("This runs in a runnable") }
```

…그리고 메서드 호출에서 사용할 수 있다:

```
val executor = ThreadPoolExecutor()
// Java signature: void execute(Runnable command)
executor.execute { println("This runs in a thread pool") }
```

자바 클래스가 함수형 인터페이스를 인자로 받는 메서드를 여러 개 가지고 있다면, 어댑터 함수를 사용해서 호출할 메서드를 선택할 수 있다. 어댑터 함수는 람다를 특정한 SAM 타입으로 변환한다. 컴파일러는 또한 필요할 때 그런 어댑터 함수를 생성한다:

```
executor.execute(Runnable { println("This runs in a thread pool") })
```

SAM 변환은 인터페이스에만 동작한다. 추상 클래스가 단지 1개의 추상 메서드만 갖고 있다 해서 추상 클래스에는 동작하지 않는다.

이런 특징은 자바 상호운용에서만 동작한다. 코틀린은 적당한 함수 타입이 있기 때문에, 함수를 코틀린 인터페이스 구현으로 자동으로 변환해주는 기능이 불필요하며 그래서 지원하지 않는다.

코틀린에서 JNI 사용하기

네이티브(C나 C++) 코드로 구현한 함수를 선언하려면 `external` 수식어를 지정해야 한다:

```
external fun foo(x: Int): Double
```

프로시저의 나머지는 자바와 똑같게 동작한다.

자바에서 코틀린 호출하기

자바에서 쉽게 코틀린 코드를 호출할 수 있다.

프로퍼티

코틀린 프로퍼티는 다음 자바 요소로 컴파일된다:

- `get` 접두사가 앞에 붙은 계산된 이름을 가진 getter 메서드
- `set` 접두사가 앞에 붙은 계산된 이름을 가진 setter 메서드 (`var` 프로퍼티에만 적용)
- 프로퍼티 이름과 같은 이름을 가진 private 필드 (지원 필드를 가진 프로퍼티에만 적용)

예를 들어, `var firstName: String` 코드는 다음의 자바 선언으로 컴파일된다:

```
private String firstName;

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}
```

프로퍼티 이름이 `is` 로 시작하면, 다른 이름 매핑 규칙을 사용한다. getter 이름으로 프로퍼티 이름을 그대로 사용하고, setter 이름은 `is` 대신에 `set` 을 사용한다. 예를 들어, `isOpen` 프로퍼티의 경우 getter는 `isOpen` 이고 setter는 `setOpen()` 이 된다. 이 규칙은 `Boolean` 뿐만 아니라 모든 타입의 프로퍼티에 적용된다.

패키지 수준 함수

`example.kt` 파일의 `org.foo.bar` 안에 선언한 모든 함수(확장 함수 포함)와 프로퍼티는 `org.foo.bar.ExampleKt` 라 불리는 자바 클래스의 정적 메서드로 컴파일된다.

```
// example.kt
package demo

class Foo

fun bar() {
}
```

```
// 자바
new demo.Foo();
demo.ExampleKt.bar();
```

`@JvmName` 애노테이션을 사용해서 생성할 자바 클래스 이름을 바꿀 수 있다:

```
@file:JvmName("DemoUtils")

package demo

class Foo

fun bar() {
}
```

```
// 자바
new demo.Foo();
demo.DemoUtils.bar();
```

같은 자바 클래스 이름을 생성하는 파일이 여러 개 존재하면(같은 패키지와 같은 이름 또는 같은 @JvmName 애노테이션) 보통 에러이다. 하지만, 컴파일러는 이를 위한 파사드 클래스를 만드는 기능을 제공한다. 이 파사드 클래스는 같은 이름을 가진 모든 파일의 모든 선언을 포함하며, 지정한 이름을 갖는다. 이 파사드 클래스를 생성하려면 모든 파일에 @JvmMultifileClass 애노테이션을 사용하면 된다.

```
// oldutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package demo

fun foo() {
}
```

```
// newutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package demo

fun bar() {
}
```

```
// 자바
demo.Utils.foo();
demo.Utils.bar();
```

인스턴스 필드

코틀린 프로퍼티를 자바 필드로 노출하고 싶다면, 프로퍼티에 `@JvmField` 애노테이션을 붙이면 된다. 이 필드는 해당 프로퍼티와 동일한 가시성을 갖는다. 프로퍼티가 지원 필드를 갖고, private이 아니고, `open` 이나 `override` 나 `const` 수식어를 갖지 않고, 위임 프로퍼티가 아니면, `@JvmField` 애노테이션을 붙일 수 없다.

```
class C(id: String) {
    @JvmField val ID = id
}
```

```
// 자바
class JavaClient {
    public String getID(C c) {
        return c.ID;
    }
}
```

초기화 지연 프로퍼티 또한 필드로 노출할 수 있다. 필드는 `lateinit` 프로퍼티의 setter와 동일한 가시성을 갖는다.

정적 필드

이름 가진 오브젝트나 컴페니언 오브젝트에 선언한 코틀린 프로퍼티는 이름 가진 오브젝트나 컴페니언 오브젝트를 포함하는 클래스에 정적 지원 필드를 갖는다.

보통 이 필드는 private이지만 다음 중 하나로 노출할 수 있다:

- `@JvmField` 애노테이션
- `lateinit` 수식어
- `const` 수식어

이 프로퍼티에 `@JvmField` 를 붙이면, 프로퍼티와 같은 가시성을 가진 정적 필드로 만든다.

```
class Key(val value: Int) {
    companion object {
        @JvmField
        val COMPARATOR: Comparator<Key> = compareBy<Key> { it.value }
    }
}
```

```
// 자바
Key.COMPARATOR.compare(key1, key2);
// Key 클래스에 public static final 필드
```

객체나 컴패니언 오브젝트의 초기화 지연 프로퍼티는 프로퍼티 setter와 같은 가시성을 갖는 정적 지원 필드를 갖는다.

```
object Singleton {
    lateinit var provider: Provider
}
```

```
// 자바
Singleton.provider = new Provider();
// Singleton 클래스에 public static이고 final이 아닌 필드
```

(클래스나 최상위 수준에 선언한) `const` 프로퍼티는 자바의 정적 필드로 바뀐다:

```
// file example.kt

object Obj {
    const val CONST = 1
}

class C {
    companion object {
        const val VERSION = 9
    }
}

const val MAX = 239
```

자바:

```
int c = Obj.CONST;
int d = ExampleKt.MAX;
int v = C.VERSION;
```

정적 메서드

위에서 말한 것처럼, 코틀린은 패키지 수준 함수를 정적 메서드로 표현한다. 코틀린은 또한 이름가진 오브젝트나 컴패니언 오브젝트에 정의한 함수에 `@JvmStatic` 을 붙이면 정적 메서드를 생성한다. 이 애노테이션을 사용하면 컴파일러는 객체를 둘러싼 클래스에 정적 메서드를 생성하고 객체 자체에 인스턴스 메서드를 생성한다. 다음은 예이다:

```
class C {
    companion object {
        @JvmStatic fun foo() {}
        fun bar() {}
    }
}
```

이제 `foo()` 는 자바에서 정적 메서드이고, `bar()` 는 아니다:

```
C.foo(); // 동작
C.bar(); // 에러: 정적 메서드 아님
C.Companion.foo(); // 인스턴스 메서드 유지
C.Companion.bar(); // 오직 이렇게만 동작
```

이름 가진 오브젝트도 동일하다:

```
object Obj {
    @JvmStatic fun foo() {}
    fun bar() {}
}
```

자바:

```
Obj.foo(); //동작
Obj.bar(); // 에러
Obj.INSTANCE.bar(); // 동작, 싱글톤 인스턴스를 통해 호출
Obj.INSTANCE.foo(); // 역시 동작
```

`@JvmStatic` 애노테이션을 오브젝트나 컴패니언 오브젝트의 프로퍼티에도 적용할 수 있다. 이는 getter와 setter 메서드를 오브젝트나 컴패니언 오브젝트를 감싼 클래스의 정적 멤버로 만든다.

가시성

코틀린 가시성은 다음 규칙에 따라 자바로 매핑된다:

- `private` 멤버는 `private` 멤버로 컴파일
- `private` 최상위 선언은 패키지-로컬 선언으로 컴파일
- `protected` 는 `protected` 로 유지 (자바는 같은 패키지의 다른 클래스에서 protected 멤버에 접근하는 것을 허용하나 코틀린은 그렇지 않다. 따라서, 자바 클래스가 코드에 대한 더 넓은 접근을 갖는다.)
- `internal` 선언은 자바에서 `public` 이 됨. `internal` 클래스의 이름을 변형해서 자바에서 우연히 사용하는 것을 어렵게 하고 코틀린 규칙에 따라 서로 볼 수 없는 동일 시그너처를 가진 멤버를 오버로딩할 수 있게 한다.
- `public` 은 `public` 으로 남음

KClass

`KClass` 타입의 파라미터를 갖는 코틀린 메서드를 호출해야 할 때가 있다. `Class` 에서 `KClass` 로의 자동 변환은 없기에 `Class<T>.kotlin` 확장 프로퍼티와 동일한 메서드를 호출해서 수동으로 구해야 한다.

```
kotlin.jvm.JvmClassMappingKt.getKotlinClass(MainView.class)
```

@JvmName으로 시그너처 깨짐 처리

코틀린에서 이름 가진 함수가 바이트코드에서 다른 JVM 이름을 갖도록 해야 할 때가 있다. 가장 두드러진 예는 *타입 제거* 때문에 발생한다:

```
fun List<String>.filterValid(): List<String>
fun List<Int>.filterValid(): List<Int>
```

두 함수를 함께 정의할 수 없는데, 그 이유는 JVM 시그너처가 `filterValid(Ljava/util/List;)Ljava/util/List;` 로 같기 때문이다. 정말로 코틀린에서 같은 이름을 가진 함수가 필요하다면 한 함수(또는 두 함수)에 다른 이름을 인자로 갖는 `@JvmName` 을 붙이면 된다:

```
fun List<String>.filterValid(): List<String>

@JvmName("filterValidInt")
fun List<Int>.filterValid(): List<Int>
```

위 예에서 코틀린은 `filterValid` 이름으로 두 함수를 접근할 수 있지만, 자바에서는 `filterValid` 와 `filterValidInt` 이름을 사용한다.

같은 트릭을 `getX()` 함수와 `x` 프로퍼티를 함께 사용해야 할 때 적용할 수 있다:

```
val x: Int
    @JvmName("getX_prop")
    get() = 15

fun getX() = 10
```

오버로드 생성

보통 디폴트 파라미터 값을 갖는 코틀린 함수를 작성하면, 자바에서는 모든 파라미터가 필요한 전체 시그너처로 보인다. 자바에 여러 오버로드 메서드를 노출하고 싶다면 `@JvmOverloads` 애노테이션을 사용하면 된다.

애노테이션은 생성자와 정적 메서드 등에도 동작한다. 인터페이스에 정의한 메서드를 포함한 추상 메서드에는 사용할 수 없다.

```
class Foo @JvmOverloads constructor(x: Int, y: Double = 0.0) {
    @JvmOverloads fun f(a: String, b: Int = 0, c: String = "abc") {
        ...
    }
}
```

기본 값을 가진 모든 파라미터에 대해, 추가로 한 개의 오버로드 메서드를 생성한다. 추가 메서드는 파라미터 목록에서 기본 값을 가진 파라미터와 그 파라미터 목록의 오른쪽에 있는 모든 파라미터를 제거한 나머지 파라미터를 갖는다. 위 예의 경우 다음을 생성한다:

```
// 생성자:
Foo(int x, double y)
Foo(int x)

// 메서드
void f(String a, int b, String c) { }
void f(String a, int b) { }
void f(String a) { }
```

보조 생성자 에서 설명한 것처럼, 클래스가 모든 생성자 파라미터에 대해 기본 값을 가지면, 인자 없는 public 생성자를 생성한다. 이는 `@JvmOverloads` 를 지정하지 않아도 동작한다.

Checked 익셉션

앞서 언급한 것처럼, 코틀린에는 checked 익셉션이 없다. 그래서 보통 코틀린 함수의 자바 시그너처는 발생할 익셉션을 선언하지 않는다. 아래와 같은 코틀린 함수가 있다고 하자.

```
// example.kt
package demo

fun foo() {
    throw IOException()
}
```

다음과 같이 자바 코드에서 이 함수를 호출하고 익셉션을 catch 하면,

```
// Java
try {
  demo.Example.foo();
}
catch (IOException e) { // 에러: foo()는 throws 목록에 IOException을 선언하지 않았음
  // ...
}
```

자바 컴파일러가 에러를 발생한다. 왜냐면 `foo()` 가 `IOException` 을 선언하지 않았기 때문이다. 이 문제를 해결하려면 코틀린에 `@Throws` 애노테이션을 사용한다:

```
@Throws(IOException::class)
fun foo() {
    throw IOException()
}
```

Null-안정성

자바에서 코틀린 함수를 호출할 때 non-null 파라미터에 `null` 을 전달하는 것을 막을 수 없다. 이것이 코틀린이 non-null을 기대하는 모든 public 함수에 대해 런타임 검사를 생성하는 이유이다. 이 방법은 자바 코드에서 즉시 `NullPointerException` 을 발생하게 만든다.

변성 지네릭

코틀린 클래스에서 선언 위치 변성 을 사용할 때, 자바 코드에 어떻게 보여줄지에 대한 두 가지 선택이 있다. 선언 위치 변성을 사용하는 클래스와 두 함수 예를 보자:

```
class Box<out T>(val value: T)

interface Base
class Derived : Base

fun boxDerived(value: Derived): Box<Derived> = Box(value)
fun unboxBase(box: Box<Base>): Base = box.value
```

이 함수를 자바 코드로 해석하는 순진한 방법은 다음과 같이 하는 것이다:

```
Box<Derived> boxDerived(Derived value) { ... }
Base unboxBase(Box<Base> box) { ... }
```

문제는 코틀린은 `unboxBase(boxDerived("s"))` 가 가능하지만, 자바는 불가능하다는 것이다. 왜냐면, 자바에서 `Box` 클래스는 `T` 파라미터에 대해 *무공변* 이며, 그래서 `Box<Derived>` 는 `Box<Base>` 의 하위타입이 아니기 때문이다. 자바에서 이것을 사용하려면 `unboxBase` 를 다음과 같이 정의해야 한다:

```
Base unboxBase(Box<? extends Base> box) { ... }
```

여기서 사용 위치 변성으로 선언 위치 변성을 흉내내기 위해 *와일드카드 타입* (`? extends Base`)을 사용했다. 자바는 사용 위치 변성만 있기 때문이다.

코틀린 API가 자바에서 동작할 수 있도록 타입 파라미터가 메서드의 *파라미터로 쓰일* 때 공변으로 정의한 `Box` 에 대한 `Box<Super>` 와 `Box<? extends Super>` 를 생성한다(또는 반공변으로 정의한 `Foo` 에 대한 `Foo<? super Bar>` 를 생성). 타입 파라미터가 리턴 값일 때, 와일카드를 생성하지 않는다. 왜냐면, 자바 클라이언트가 이를 처리할 수 있기 때문이다(그리고, 리턴 타입에서 와일드카드는 보통의 자바 코딩 스타일에 반한다). 따라서 예제의 함수를 실제로 다음과 같이 해석한다:

```
// 리턴 타입 - 와일드카드 없음
Box<Derived> boxDerived(Derived value) { ... }

// 파라미터 - 와일드카드
Base unboxBase(Box<? extends Base> box) { ... }
```

주의: 인자 타입이 final이면 와일드카드를 생성할 위치가 없고, 따라서 `Box<String>` 은 그것이 취하는 위치가 무엇이냐에 상관없이 항상 `Box<String>` 이다.

기본 생성되지 않는 타입에 대해 와일드카드가 필요하면 `@JvmWildcard` 주석을 사용한다:

```
fun boxDerived(value: Derived): Box<@JvmWildcard Derived> = Box(value)
// 다음으로 해석
// Box<? extends Derived> boxDerived(Derived value) { ... }
```

반면에 와일드카드를 생성하지 않기를 원하면 `@JvmSuppressWildcards` 를 사용한다:

```
fun unboxBase(box: Box<@JvmSuppressWildcards Base>): Base = box.value
// 다음으로 해석
// Base unboxBase(Box<Base> box) { ... }
```

주의: `@JvmSuppressWildcards` 는 개별 타입 인자에 적용할 수 있을 뿐만 아니라 함수나 클래스에 적용할 수 있다. 클래스나 함수에 적용하면 그 안에 있는 모든 와일드카드를 생성하지 않는다.

Nothing 타입의 해석

`Nothing` 타입은 자바에 알맞은 비슷한 것이 없다. 사실 `java.lang.Void` 를 포함한 모든 자바 레퍼런스 타입은 값으로 `null` 을 가질 수 있는데 `Nothing` 은 그것 조차 안 된다. 따라서, 이 타입을 자바에서 정확하게 표현할 수 없다. 이것이 코틀린이 `Nothing` 타입의 인자를 사용할 때 raw 타입을 생성하는 이유이다.

```
fun emptyList(): List<Nothing> = listOf()
// 다음으로 해석
// List emptyList() { ... }
```

도구

코틀린 코드 문서화

코틀린 코드를 문서화하는데 사용하는 언어를(자바의 JavaDoc과 같은 언어) **KDoc** 이라 부른다. KDoc은 블록 태그를 위한 JavaDoc의 구문(코틀린에 특정한 구성 요소를 지원하기 위해 확장 추가)과 인라인 마크업을 위한 마크다운을 합쳤다.

문서 생성

코틀린 문서 생성 도구는 Dokka 이다. 명령어 사용법은 Dokka README 를 참고한다.

Dokka는 그레이들, 메이븐, 앤트 플러그인을 지원하므로 빌드 과정에 문서 생성을 통합할 수 있다.

KDoc 구문

JavaDoc처럼 KDoc 주석은 `/**` 로 시작해서 `*/` 로 끝난다. 주석의 각 줄은 별표로 시작할 수 있으며 별표는 주석 내용에 포함되지 않는다.

관례상, 문서 텍스트의 첫 번째 단락은 요소의 요약 내용이 오고 다음 문장부터 상세 설명이 온다.

모든 블록 태그는 새 줄에 작성하고 `@` 글자로 시작한다.

다음은 KDoc을 이용해서 클래스를 문서화한 예이다:

```
/**
 * *members*의 그룹
 *
 * 이 클래스는 유용한 로직이 없다. 단지 문서화 예제일 뿐이다.
 *
 * @param T 이 그룹의 멤버 타입
 * @property name 이 그룹의 이름
 * @constructor 빈 그룹을 생성한다.
 */
class Group<T>(val name: String) {
    /**
     * 이 그룹에 [member]를 추가한다.
     * @return 그룹의 새 크기
     */
    fun add(member: T): Int { ... }
}
```

블록 태그

현재 KDoc은 다음 블록 태그를 지원한다:

`@param <name>`

클래스, 프로퍼티, 함수의 타입 파라미터나 함수의 값 파라미터를 문서화한다. 설명에서 파라미터 이름을 더 잘 구분하는 것을 선호하면, 대괄호로 파라미터 이름을 감싸면 된다. 다음 두 구문은 동일하다:

```
@param name description.
@param[name] description.
```

`@return`

함수의 리턴 값을 문서화한다.

`@constructor`

클래스의 주요 생성자를 문서화한다.

`@receiver`

확장 함수의 리시버를 문서화한다.

`@property <name>`

특정 이름을 가진 클래스의 프로퍼티를 문서화한다. 주요 생성자에 선언한 프로퍼티를 문서화할 때 이 태그를 사용할 수 있다. 프로퍼티 선언 앞에 바로 문서 주석을 넣는 것이 어색할 수 있다.

`@throws <class>` , `@exception <class>`

메서드가 발생할 수 있는 익셉션을 문서화한다. 코틀린에는 checked 익셉션이 없으므로 모든 발생 가능한 익셉션을 문서화한다고 기대하지 않는다. 하지만, 클래스 사용자에게 유용한 정보를 제공하고 싶을 때 이 태그를 사용할 수 있다.

`@sample <identifier>`

지정한 이름의 한정자를 가진 함수의 몸체를 현재 요소를 위한 문서화에 삽입한다. 요소의 사용 방법 예를 보여주기 위한 용도로 사용할 수 있다.

`@see <identifier>`

문서의 **See Also** 블록에 지정한 클래스나 메서드에 대한 링크를 추가한다.

`@author`

문서화할 요소의 작성자를 지정한다.

`@since`

문서화한 요소를 추가한 소프트웨어의 버전을 지정한다.

`@suppress`

생성할 문서에서 요소를 제외한다. 모듈의 공식 API에 속하지 않지만 외부에서 여전히 접근할 수 있는 요소에 사용할 수 있다.

⚠ KDoc은 `@deprecated` 태그를 지원하지 않으며, 대신 `@Deprecated` 애노테이션을 사용한다.

인라인 마크업

KDoc은 인라인 마크업을 위해 일반 마크다운 구문을 사용한다. 코드에서 다른 요로의 링크를 위한 간결한 구문을 추가로 확장했다.

요소에 대한 링크

다른 요소(클래스, 메서드, 파라미터의 프로퍼티)를 링크하려면, 단순히 대괄호에 이름을 넣으면 된다:

```
Use the method [foo] for this purpose.
```

링크에 커스텀 라벨을 지정하고 싶으면 마크다운의 참조-방식 구문을 사용한다:

```
Use [this method][foo] for this purpose.
```

링크에 이름을 한정해서 사용할 수 있다. JavaDoc과 달리 한정한 이름은 메서드 이름 시작 전에 컴포넌트를 구분하기 위해
항상 점(.) 글자를 사용한다.

```
Use [kotlin.reflect.KClass.properties] to enumerate the properties of the class.
```

문서화할 요소 안에서 사용할 이름과 같은 규칙을 사용해서 링크의 이름을 찾아낸다. 특히, 이는 현재 파일에 임포트한 이름을 가진 경우 KDoc 주석에서 그 이름을 사용할 때 완전한 이름을 사용할 필요가 없다는 것을 의미한다.

KDoc에는 오버로딩한 이름을 알아내기 위한 구문이 없다. 코틀린 문서 생성 도구는 같은 페이지의 모든 오버로딩된 함수를 위한 문서화를 추가하므로, 링크에 특정한 오버로딩된 함수를 지정하지 않아도 링크가 동작한다.

모듈과 패키지 문서화

모듈 전체와 그 모듈의 모든 패키지에 대한 문서화는 별도 마크다운 파일로 제공한다. 그 파일 경로는 `-include` 명령행 파라미터를 사용해서 Dokka에 전달하고, 앤트 메이븐, 그레이들 플러그인에서 해당하는 파라미터로 전달한다.

그 파일 안에는 전체 모듈과 개별 패키지에 대한 문서화를 대응하는 첫 번째 레벨 헤딩으로 추가한다. 모듈을 위한 헤딩 텍스트는 "Module `<모듈명>` " 형식이어야 하고, 패키지의 경우는 "Package `<완전한 패키지 이름>` "이여야 한다.

다음은 예제이다.

```
# Module kotlin-demo

The module shows the Dokka syntax usage.

# Package org.jetbrains.kotlin.demo

Contains assorted useful stuff.

## Level 2 heading

Text after this heading is also part of documentation for `org.jetbrains.kotlin.demo`

# Package org.jetbrains.kotlin.demo2

Useful stuff in another package.
```

코틀린 애노테이션 처리 도구 사용하기

코틀린 플러그인은 _Dagger_나 _DBFlow_와 같은 애노테이션 처리기를 지원한다. 애노테이션 처리기를 코틀린 클래스에 사용하려면 `kotlin-kapt` 플러그인을 적용한다.

그레이들 설정

```
apply plugin: 'kotlin-kapt'
```

또는, 코틀린 1.1.1부터 DSL 플러그인을 사용해서 애노테이션 처리기를 적용할 수 있다:

```
plugins {
    id "org.jetbrains.kotlin.kapt" version "1.1.51"
}
```

`dependencies` 블록에 `kapt` 설정을 사용해서 각각의 의존을 추가한다:

```
dependencies {
    kapt 'groupId:artifactId:version'
}
```

이미 android-apt 플러그인을 사용한다면 `build.gradle` 파일에서 그 설정을 제거하고 `apt` 설정을 `kapt` 로 대체한다. 프로젝트가 자바 클래스를 포함한다면, `kapt` 가 자바 클래스도 처리한다.

`androidTest` 나 `test` 소스에 대해 애노테이션 처리기를 사용하려면 각 `kapt` 설정에 대해 `kaptAndroidTest` 와 `kaptTest` 이름을 사용한다. `kaptAndroidTest` 와 `kaptTest` 는 `kapt` 를 확장했으므로 `kapt` 의존을 그대로 적용할 수 있고, 제품 소스와 테스트 소스에 사용할 수 있다.

일부 애노테이션 처리기는(`AutoFactory` 와 같은) 선언 시그너처의 정확한 타입에 의존한다. 기본적으로 Kapt는 모든 알 수 없는 타입(생성된 클래스의 타입 포함)을 `NonExistentClass` 로 대체하는데, 이 기능이 다르게 동작하게 바꿀 수 있다. `build.gradle` 파일에 다음 플래그를 추가하면 스텁의 에러 타입 추론이 가능해진다:

```
kapt {
    correctErrorTypes = true
}
```

이 옵션은 실험 단계이므로 기본으로 비활성화되어 있다.

메이븐 설정 (코틀린 1.1.2부터)

`compile` 전에 kotlin-maven-plugin의 `kapt` goal을 실행하는 설정을 추가한다:

```
<execution>
    <id>kapt</id>
    <goals>
        <goal>kapt</goal>
    </goals>
    <configuration>
        <sourceDirs>
            <sourceDir>src/main/kotlin</sourceDir>
            <sourceDir>src/main/java</sourceDir>
        </sourceDirs>
        <annotationProcessorPaths>
            <!-- Specify your annotation processors here. -->
            <annotationProcessorPath>
                <groupId>com.google.dagger</groupId>
                <artifactId>dagger-compiler</artifactId>
                <version>2.9</version>
            </annotationProcessorPath>
        </annotationProcessorPaths>
    </configuration>
</execution>
```

코틀린 예제 리포지토리 에서 코틀린, 메이븐, Dagger의 사용 예를 보여주는 완전한 예제 프로젝트를 찾을 수 있다.

인텔리J IDEA 자체의 빌드 시스템은 아직 kapt를 지원하지 않는다. 애노테이션 처리를 다시 실행하고 싶다면 “Maven Projects” 툴바에서 빌드를 실행한다.

그레이들 사용하기

그레이들에서 코틀린을 빌드하려면 *kotlin-gradle* 플러그인을 설정 하고 프로젝트에 그 플러그인을 적용 하고, *kotlin-stdlib* 의존 을 추가해야 한다. 인텔리J IDEA의 Tools | Kotlin | Configure Kotlin in Project 메뉴를 실행하면 자동으로 이를 처리해준다.

플러그인과 버전

`kotlin-gradle-plugin` 은 코틀린 코드와 모듈을 컴파일한다.

사용할 코틀린 버전은 보통 `kotlin_version` 프로퍼티로 정의한다:

```
buildscript {
    ext.kotlin_version = '1.1.51'

    repositories {
        mavenCentral()
    }

    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

코틀린 그레이들 플러그인 1.1.1과 그레이들 플러그인 DSL 을 사용하면 코틀린 버전을 설정할 필요가 없다.

JVM 대상

JVM을 대상으로 하려면 코틀린 플러그인을 적용해야 한다:

```
apply plugin: "kotlin"
```

또는 코틀린 1.1.1부터 그레이들 플러그인 DSL 로 플러그인을 적용할 수 있다:

```
plugins {
    id "org.jetbrains.kotlin.jvm" version "1.1.51"
}
```

이 블록에서 `version` 은 리터럴이어야 하며, 다른 빌드 스크립트로부터 적용할 수 없다.

같은 폴더나 다른 폴더에 코틀린 소스와 자바 소스를 함께 사용할 수 있다. 기본 관례는 다른 폴더를 사용하는 것이다:

```
project
    - src
        - main (root)
            - kotlin
            - java
```

기본 관례를 사용하지 않으려면 해당 *sourceSets* 프로퍼티를 수정해야 한다:

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
    main.java.srcDirs += 'src/main/myJava'
}
```

자바스크립트 대상

자바스크립트를 대상으로 할 때에는 다른 플러그인을 적용해야 한다:

```
apply plugin: "kotlin2js"
```

이 플러그인은 코틀린 파일에만 작동하므로, (같은 프로젝트에서 자바 파일을 포함한다면) 코틀린과 자바 파일을 구분할 것을 권한다. JVM을 대상으로 하고 기본 관례를 사용하지 않으면, *sourceSets* 을 사용해서 소스 폴더를 지정해야 한다.

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
}
```

플러그인은 자바스크립트 파일을 출력할 뿐만 아니라 바이너리 디스크립터를 가진 JS 파일을 추가로 생성한다. 다른 코틀린 모듈에서 의존할 수 있는 재사용가능한 라이브러리를 빌드하려면 이 파일이 필요하며, 변환 결과에 함께 배포해야 한다. `kotlinOptions.metaInfo` 옵션으로 파일 생성을 설정한다:

```
compileKotlin2Js {
    kotlinOptions.metaInfo = true
}
```

안드로이드 대상

안드로이드의 그레이들 모델은 일반 그레이들 모델과 약간 다르다. 그래서 안드로이드 코틀린으로 작성한 안드로이드 프로젝트를 빌드하려면, *kotlin* 플러그인 대신 *kotlin-android* 플러그인이 필요하다.

```
buildscript {
    ext.kotlin_version = '1.1.51'

    ...

    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
```

표준 라이브러리 의존 도 설정해야 한다.

안드로이드 스튜디오

안드로이드 스튜디오를 사용하면 android 설정에 다음을 추가해야 한다:

```
android {
  ...

  sourceSets {
    main.java.srcDirs += 'src/main/kotlin'
  }
}
```

이 설정은 안드로이드 스튜디오가 코틀린 디렉토리를 소스 루트로 인식하도록 해서 프로젝트 모델을 IDE로 로딩할 때 올바르게 인식하도록 한다. 또는 보통 `src/main/java` 에 위치한 자바 소스 디렉토리에 코틀린 클래스를 위치시킬 수도 있다.

의존 설정

위에서 보여준 `kotlin-gradle-plugin` 의존 외에 코틀린 표준 라이브러리를 의존에 추가해야 한다.

```
repositories {
    mavenCentral()
}

dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib"
}
```

자바 스크립트를 대상으로 하면 대신 `compile "org.jetbrains.kotlin:kotlin-stdlib-js"` 를 사용한다.

JDK 7이나 JDK 8이 대상이라면 코틀린 표준 라이브러리의 확장 버전을 사용할 수 있다. 확장 버전은 JDK 새 버전에 들어간 API를 위해 추가한 확장 함수를 포함한다. `kotlin-stdlib` 대신 다음 의존 중 하나를 사용한다:

```
compile "org.jetbrains.kotlin:kotlin-stdlib-jre7"
compile "org.jetbrains.kotlin:kotlin-stdlib-jre8"
```

코틀린 리플렉션 이나 테스팅 도구를 사용하면 다음 의존을 추가로 설정한다:

```
compile "org.jetbrains.kotlin:kotlin-reflect"
testCompile "org.jetbrains.kotlin:kotlin-test"
testCompile "org.jetbrains.kotlin:kotlin-test-junit"
```

코틀린 1.1.2부터 `org.jetbrains.kotlin` 그룹에 속한 의존은 적용한 플러그인에서 가져온 버전을 기본으로 사용한다. `compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"` 과 같이 완전한 의존 표기를 사용해서 버전을 수동으로 제공할 수 있다.

애노테이션 처리

코틀린 애노테이션 처리 도구 (`kapt`) 설명을 참고한다.

증분 컴파일

코틀린은 그레이들에서 선택적인 증분 컴파일을 지원한다. 증분 컴파일은 빌드 간 소스 파일의 변경을 추적해서 변경에 영향을 받는 파일만 컴파일한다.

코틀린 1.1.1부터 기본으로 증분 컴파일을 활성화한다.

기본 설정을 변경하는 몇 가지 방법이 있다:

1. `gradle.properties` 나 `local.properties` 파일에 `kotlin.incremental=true` 또는 `kotlin.incremental=false` 설정을 추가한다.
2. 그레이들 명령행 파라미터로 `-Pkotlin.incremental=true` 또는 `-Pkotlin.incremental=false` 를 추가한다. 이 경우 각 빌드마다 파라미터를 추가해야 하며 증분 컴파일을 사용하지 않은 빌드가 존재하면 증분 캐시를 무효로한다.

증분 컴파일을 활성화하면, 빌드 로그에서 다음 경고 문구를 보게 된다:

```
Using kotlin incremental compilation
```

첫 번째 빌드는 증분이 아니다.

코루틴 지원

코루틴 지원은 코틀린 1.1에서 실험 특징이므로, 프로젝트에서 코루틴을 사용하면 코틀린 컴파일러는 경고를 발생한다. 경고를 끄려면, `build.gradle` 에 다음 블록을 추가한다:

```
kotlin {
    experimental {
        coroutines 'enable'
    }
}
```

모듈 이름

빌드로 생성한 코틀린 모듈은 프로젝트의 `archivesBaseName` 프로퍼티의 이름을 따른다. 프로젝트가 `lib` 나 `jvm` 과 같이 하위 프로젝트들에 공통인 넓은 의미의 이름을 사용하면, 모듈과 관련된 코틀린 출력 파일은(`*.kotlin_module`) 같은 이름을 갖는 외부 모듈의 출력 파일과 충돌할 수 있다. 이는 프로젝트를 한 개의 아카이브(예, APK)로 패키징할 때 문제를 발생시킨다.

이를 피하려면, 수동으로 유일한 `archivesBaseName` 을 설정해야 한다:

```
archivesBaseName = 'myExampleProject_lib'
```

컴파일러 옵션

컴파일러 옵션을 추가로 설정하려면 코틀린 컴파일 태스크의 `kotlinOptions` 프로퍼티를 사용한다.

JVM 대상일 때, 제품을 위한 태스크는 `compileKotlin` 이고 테스트 코드를 위한 태스크는 `compileTestKotlin` 이다. 커스텀 소스 집합을 위한 태스크는 `compile<Name>Kotlin` 패턴에 따라 호출한다.

안드로이드 프로젝트에서 태스크의 이름은 build variant 를 포함하며, `compile<BuildVariant>Kotlin` 패턴을 따른다(예, `compileDebugKotlin` , `compileReleaseUnitTestKotlin`).

자바스크립트 대싱일 때, 각각 `compileKotlin2Js` 와 `compileTestKotlin2Js` 태스크를 부르고, 커스텀 소스 집합은 `compile<Name>Kotlin2Js` 태스크를 부른다.

단일 태스크를 설정하려면 그 이름을 사용한다. 다음은 예이다:

```
compileKotlin {
    kotlinOptions.suppressWarnings = true
}

compileKotlin {
    kotlinOptions {
        suppressWarnings = true
    }
}
```

프로젝트의 모든 코틀린 컴파일 태스크를 설정하는 것도 가능하다:

```
tasks.withType(org.jetbrains.kotlin.gradle.tasks.KotlinCompile).all {
    kotlinOptions {
        // ...
    }
}
```

그레이들 태스크를 위한 전체 목록은 다음과 같다:

JVM과 JS의 공통 속성

이름	설명	가능한 값	기본 값
`apiVersion`	지정한 버전의 번들 라이브러리에서만 선언 사용을 허용함	"1.0", "1.1"	"1.1"
`languageVersion`	소스 호환성을 지정한 언어 버전으로 지정함	"1.0", "1.1"	"1.1"
`suppressWarnings`	경고를 생성하지 않음		false
`verbose`	Enable verbose 로깅 출력을 활성화		false
`freeCompilerArgs`	추가 컴파일러 인자 목록		[]

JVM 전용 속성

이름	설명	가능한 값	기본 값
`javaParameters`	메서드 파라미터에 대해 자바 1.8 리플렉션을 위한 메타데이터 생성		false
`jdkHome`	클래스패스로 포함할 JDK 홈 디렉톨 경로(기본 JAVA_HOME과 다른 경우)		
`jvmTarget`	생성할 JVM 바이트코의 대상 버전(1.6 또는 1.8), 기본은 1.6	"1.6", "1.8"	"1.6"
`noJdk`	자바 런타임을 클래스패스에 포함시키지 않음		false
`noReflect`	코틀린 리플렉션 구현을 클래스패스에 포함시키지 않음		true
`noStdlib`	코틀린 런타임을 클래스패스에 포함시키지 않음		true

JS 전용 속성

이름	설명	가능한 값	기본 값
`friendModulesDisabled`	internal 선언 추출을 비활성화		false
`main`	메인 함수를 불러야 할지 여부	"call", "noCall"	"call"
`metaInfo`	메타데이털르 가진 .meta.js와 .kjsm 파일 생성 여부. 라이브러리를 만들 때 사용함		true
`moduleKind`	컴파일러가 생성할 모듈의 종류	"plain", "amd", "commonjs", "umd"	"plain"
`noStdlib`	번들한 코틀린 표준 라이브러리를 사용하지 않을지 여부		true
`outputFile`	출력 파일 경로		
`sourceMap`	소스맵 생성 여부		false
`sourceMapEmbedSources`	소스 파일을 소스맵에 삽입할지 여부	"never", "always", "inlining"	"inlining"
`sourceMapPrefix`	소스맵에 경로에 대한 접두사		
`target`	JS 파일을 생성할 때 사용할 ECMA 버전 지정	"v5"	"v5"
`typedArrays`	기본 타입 배열을 JS 타입 배열로 변환할지 여부		false

문서 생성

코틀린 프로젝트의 문서를 생성하려면 Dokka 를 사용한다. 설정 명령어는 Dokka README 문서를 참고한다. Dokka는 언어를 함께 사용하는 프로젝트를 지원하며 표준 JavaDoc을 포함한 다양한 형식으로 출력할 수 있다.

OSGi

OSGi 지원은 코틀린 OSGi 페이지 를 참고한다.

예제

다음 예제는 그레이들 플러그인을 다양한 방법으로 설정하는 예를 보여준다:

- 코틀린
- 자바와 코틀린 혼합
- 안드로이드
- 자바스크립트

메이븐 사용하기

플러그인과 버전

kotlin-maven-plugin 은 코틀린 소스와 모듈을 컴파일한다. 현재는 메이븐 V3만 지원한다.

kotlin.version 프로퍼티로 사용할 코틀린 버전을 정의한다:

```
<properties>
    <kotlin.version>1.1.51</kotlin.version>
</properties>
```

의존

코틀린은 어플리케이션이 사용할 수 있는 확장 표준 라이브러리를 제공한다. 다음 의존을 pom 파일에 설정한다:

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-stdlib</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

JDK 7이나 JDK 8이 대상이라면 코틀린 표준 라이브러리의 확장 버전을 사용할 수 있다. 확장 버전은 JDK 새 버전에 들어간 API를 위해 추가한 확장 함수를 포함한다. `kotlin-stdlib` 대신 사용할 JDK 버전에 따라 `kotlin-stdlib-jre7` 또는 `kotlin-stdlib-jre8` 을 의존 중 하나를 사용한다.

코틀린 리플렉션 이나 테스팅 도구를 사용할 경우 리플렉션 라이브러리를 위한 아티팩트 ID는 `kotlin-reflect` 이고 테스트 라이브러리를 위한 아티팩트 ID는 `kotlin-test` 와 `kotlin-test-junit` 이다.

코틀리 소스코드만 컴파일하기

소스 코드를 컴파일하려면 태그에 소스 디렉토리를 지정한다:

```
<build>
    <sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>
    <testSourceDirectory>${project.basedir}/src/test/kotlin</testSourceDirectory>
</build>
```

소스를 컴파일할 때 코틀린 메이븐 플러그인을 참조해야 한다:

```
<build>
    <plugins>
        <plugin>
            <artifactId>kotlin-maven-plugin</artifactId>
            <groupId>org.jetbrains.kotlin</groupId>
            <version>${kotlin.version}</version>

            <executions>
                <execution>
                    <id>compile</id>
                    <goals> <goal>compile</goal> </goals>
                </execution>

                <execution>
                    <id>test-compile</id>
                    <goals> <goal>test-compile</goal> </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

코틀린과 자바 소스 컴파일

자바와 코틀린을 함께 사용하는 코드를 컴파일하려면 자바 컴파일러보다 코틀린 컴파일러를 먼저 호출해야 한다. 메이븐 관점에서는 다음과 같이 pom.xml 파일에서 maven-compiler-plugin 위에 코틀린 플러그인을 위치시켜서 maven-compiler-plugin보다 kotlin-maven-plugin을 먼저 실행해야 함을 의미한다:

```
<build>
    <plugins>
        <plugin>
            <artifactId>kotlin-maven-plugin</artifactId>
            <groupId>org.jetbrains.kotlin</groupId>
            <version>${kotlin.version}</version>
            <executions>
                <execution>
                    <id>compile</id>
                    <goals> <goal>compile</goal> </goals>
                    <configuration>
                        <sourceDirs>
                            <sourceDir>${project.basedir}/src/main/kotlin</sourceDir>
                            <sourceDir>${project.basedir}/src/main/java</sourceDir>
                        </sourceDirs>
                    </configuration>
                </execution>
                <execution>
                    <id>test-compile</id>
                    <goals> <goal>test-compile</goal> </goals>
                    <configuration>
                        <sourceDirs>
                            <sourceDir>${project.basedir}/src/test/kotlin</sourceDir>
                            <sourceDir>${project.basedir}/src/test/java</sourceDir>
                        </sourceDirs>
                    </configuration>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.5.1</version>
            <executions>
                <!-- Replacing default-compile as it is treated specially by maven -->
                <execution>
                    <id>default-compile</id>
                    <phase>none</phase>
                </execution>
                <!-- Replacing default-testCompile as it is treated specially by maven -->
                <execution>
                    <id>default-testCompile</id>
                    <phase>none</phase>
                </execution>
                <execution>
                    <id>java-compile</id>
                    <phase>compile</phase>
                    <goals> <goal>compile</goal> </goals>
                </execution>
                <execution>
                    <id>java-test-compile</id>
                    <phase>test-compile</phase>
                    <goals> <goal>testCompile</goal> </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

증분 컴파일

빌드를 더 빠르게 하기 위해, 메이븐을 위한 증분 컴파일을 활성화할 수 있다(코틀린 1.1.2부터 지원). 증분 컴파일을 하려면 `kotlin.compiler.incremental` 프로퍼티를 정의한다:

```
<properties>
    <kotlin.compiler.incremental>true</kotlin.compiler.incremental>
</properties>
```

대신, `-Dkotlin.compiler.incremental=true` 옵션을 사용해서 빌드를 실행해도 된다.

애노테이션 처리

코틀린 애노테이션 처리 도구 (`kapt`) 설명을 참고한다.

Jar 파일

모듈의 코드만 포함하는 작은 Jar 파일을 만들려면 메이븐 pom.xml 파일의 `build->plugins` 에 다음 설정을 포함한다. `main.class` 프로퍼티는 코틀린이나 자바 메인 클래스를 가리킨다.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>2.6</version>
    <configuration>
        <archive>
            <manifest>
                <addClasspath>true</addClasspath>
                <mainClass>${main.class}</mainClass>
            </manifest>
        </archive>
    </configuration>
</plugin>
```

독립(Self-contained) Jar 파일

모듈의 코드와 의존을 포함한 독립 Jar 파일을 만드려면 메이븐 pom.xml 파일의 `build->plugins` 에 다음 설정을 포함한다. `main.class` 프로퍼티는 코틀린이나 자바 메인 클래스를 가리킨다.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>2.6</version>
    <executions>
        <execution>
            <id>make-assembly</id>
            <phase>package</phase>
            <goals> <goal>single</goal> </goals>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>${main.class}</mainClass>
                    </manifest>
                </archive>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
            </configuration>
        </execution>
    </executions>
</plugin>
```

이 독립 jar 파일을 JRE에 전달하면 어플리케이션을 실행할 수 있다:

```
java -jar target/mymodule-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

자바스크립트 대상

자바스크립트 코드를 컴파일하려면 `compile` 실행의 goal로 `js` 와 `test-js` 를 사용해야 한다:

```
<plugin>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-plugin</artifactId>
    <version>${kotlin.version}</version>
    <executions>
        <execution>
            <id>compile</id>
            <phase>compile</phase>
            <goals>
                <goal>js</goal>
            </goals>
        </execution>
        <execution>
            <id>test-compile</id>
            <phase>test-compile</phase>
            <goals>
                <goal>test-js</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

또한 표준 라이브러리 의존을 변경해야 한다:

```
<groupId>org.jetbrains.kotlin</groupId>
<artifactId>kotlin-stdlib-js</artifactId>
<version>${kotlin.version}</version>
```

단위 테스팅을 지원하려면 `kotlin-test-js` 를 의존으로 추가한다.

더 많은 정보는 Getting Started with Kotlin and JavaScript with Maven 튜토리얼을 참고한다.

컴파일러 옵션 지정

컴파일러를 위한 추가 옵션을 메이븐 노드의 `` 요소에 태그로 지정할 수 있다:

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>
    <executions>...</executions>
    <configuration>
        <nowarn>true</nowarn>  <!-- 경고 비활성화 -->
    </configuration>
</plugin>
```

프로퍼티를 통해 다양한 옵션을 설정할 수 있다:

```
<project ...>
    <properties>
        <kotlin.compiler.languageVersion>1.0</kotlin.compiler.languageVersion>
    </properties>
</project>
```

다음 속성을 지원한다:

Attributes common for JVM and JS

이름	프로퍼티 이름	설명	가능한 값	기본 값
nowarn		경고를 생성하지 않음	true, false	false
languageVersion	kotlin.compiler.languageVersion	소스 호환성을 지정한 언어 버전으로 지정함	"1.0", "1.1"	"1.1"
apiVersion	kotlin.compiler.apiVersion	지정한 버전의 번들 라이브러리에서만 선언 사용을 허용함	"1.0", "1.1"	"1.1"
sourceDirs		컴파일할 소스 파일을 포함한 폴더목록		프로젝트 소스 루트
compilerPlugins		활성화함 컴파일러 플러그인		[]

이름	프로퍼티 이름	설명	가능한 값	기본 값
pluginOptions		컴파일러 플러그인을 위한 옵션		[]
args		추가 컴파일러 인자		[]

JVM 전용 속성

이름	프로퍼티 이름	설명	가능한 값	기본 값
jvmTarget	kotlin.compiler.jvmTarget	생성할 JVM 바이트코드의 대상 버전	"1.6", "1.8"	"1.6"
jdkHome	kotlin.compiler.jdkHome	클래스패스로 포함할 JDK 홈 디렉톨 경로(기본 JAVA_HOME과 다른 경우)		

JS 전용 속성

이름	프로퍼티 이름	설명	가능한 값	기본 값
outputFile		출력 파일 경로		
metaInfo		메타데이털르 가진 .meta.js와 .kjsm 파일 생성 여부. 라이브러리를 만들 때 사용함	true, false	true
sourceMap		소스맵 생성 여부	true, false	false
sourceMapEmbedSources		소스 파일을 소스맵에 삽입할지 여부	"never", "always", "inlining"	"inlining"
sourceMapPrefix		소스맵에 경로에 대한 접두사		
moduleKind		컴파일러가 생성할 모듈의 종류	"plain", "amd", "commonjs", "umd"	"plain"

문서 생성

표준 JavaDoc 생성 플러그인(`maven-javadoc-plugin`)은 코틀린 코드를 지원하지 않는다. 코틀린 프로젝트를 위한 문서를 생성하려면 Dokka 를 사용한다. 설정 명령어는 Dokka README 문서를 참고한다. Dokka는 언어를 함께 사용하는 프로젝트를 지원하며 표준 JavaDoc을 포함한 다양한 형식으로 출력할 수 있다.

OSGi

OSGi 지원은 코틀린 OSGi 페이지 를 참고한다.

예제

메이븐 예제 프로젝트는 깃헙 리포지토리에서 바로 다운로드 받을 수 있다.

앤트 사용하기

앤트 태스크 얻기

코틀린은 앤트에서 사용할 세 개 태스크를 제공한다:

— kotlinc: JVM 대상 코틀린 컴파일러

— kotlin2js: 자바스크립트 대상 코틀린 컴파일러

— withKotlin: 표준 *javac* 앤트 태스크를 사용할 때 코틀린 파일을 컴파일하기 위한 태스크

이 태스크는 *kotlin-ant.jar* 라이브러리에 정의되어 있다. 이 파일은 코틀린 컴파일러 의 *lib* 폴더에 있다.

코틀린 소스만 있는 경우의 JVM 대상

프로젝트가 코틀린 소스 코드로만 구성되어 있을 경우, 프로젝트를 컴파일하는 가장 쉬운 방법은 *kotlinc* 태스크를 사용하는 것이다:

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlinc src="hello.kt" output="hello.jar"/>
    </target>
</project>
```

${kotlin.lib}는 코틀린 표준 컴파일러 압축을 푼 폴더를 가리킨다.

코틀린 소스만 있고 여러 루트를 가진 경우의 JVM 대상

프로젝트에 소스 루트가 여러 개인 경우, 경로를 정의하기 위해 요소로 *src* 를 사용한다:

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlinc output="hello.jar">
            <src path="root1"/>
            <src path="root2"/>
        </kotlinc>
    </target>
</project>
```

코틀린과 자바 소스를 가진 경우의 JVM 대상

프로젝트가 코틀린과 자바 소스 코드로 구성되어 있으면, *kotlinc* 를 사용할 수 있지만 태스크 파라미터 중복을 피하기 위해 *withKotlin* 태스크를 사용할 것을 권한다:

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <delete dir="classes" failonerror="false"/>
        <mkdir dir="classes"/>
        <javac destdir="classes" includeAntRuntime="false" srcdir="src">
            <withKotlin/>
        </javac>
        <jar destfile="hello.jar">
            <fileset dir="classes"/>
        </jar>
    </target>
</project>
```

`<withKotlin>` 을 위한 추가 명령행 인자를 지정하려면 중첩한 `<compilerArg>` 파라미터를 사용한다. 사용할 수 있는 전체 인자 목록은 `kotlinc -help` 로 확인할 수 있다. `moduleName` 속성을 사용해서 컴파일할 모듈의 이름을 지정할 수 있다:

```
<withKotlin moduleName="myModule">
    <compilerarg value="-no-stdlib"/>
</withKotlin>
```

단일 소스 폴더를 가진 자바 스크립트 대상

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlin2js src="root1" output="out.js"/>
    </target>
</project>
```

접두사, 접미사, 소스맵 옵션을 가진 자바스크립트 대상

```
<project name="Ant Task Test" default="build">
    <taskdef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <kotlin2js src="root1" output="out.js" outputPrefix="prefix" outputPostfix="postfix"
sourcemap="true"/>
    </target>
</project>
```

단일 소스 폴더와 metaInfo 옵션을 가진 자바스크립트 대상

코틀린/자바스크립트 라이브러리로 변환 결과를 배포하고 싶을 때 `metaInfo` 옵션이 유용하다. `metaInfo` 를 true로 설정하면, 컴파일 하는 동안 바이너리 메타데이터를 가진 JS 파일을 추가로 생성한다. 변환 결과와 함께 이 파일을 배포해야 한다:

```
<project name="Ant Task Test" default="build">
    <typedef resource="org/jetbrains/kotlin/ant/antlib.xml" classpath="${kotlin.lib}/kotlin-ant.jar"/>

    <target name="build">
        <!-- out.meta.js will be created, which contains binary metadata -->
        <kotlin2js src="root1" output="out.js" metaInfo="true"/>
    </target>
</project>
```

레퍼런스

요소와 속성의 전체 목록은 아래와 같다:

kotlinc와 kotlin2js를 위한 공통 속성

이름	설명	필수	기본 값
`src`	컴파일 할 코틀린 소스 파일이나 폴더	Yes	
`nowarn`	모든 컴파일 경고를 감춤	No	false
`noStdlib`	클래스패스에 코틀린 표준 라이브러리를 포함하지 않음	No	false
`failOnError`	컴파일하는 동안 에러를 발견하면 빌드를 실패함	No	true

kotlinc 속성

이름	설명	필수	기본 값
`output`	출력 위치 디렉토리 또는 .jar 파일 이름	Yes	
`classpath`	컴파일 클래스패스	No	
`classpathref`	컴파일 클래스패스 레퍼런스	No	
`includeRuntime`	`output` 이 .jar 파일이면 jar 파일에 코틀린 런타임 라이브러리를 포함	No	true
`moduleName`	컴파일 할 모듈의 이름	No	프로젝트 또는 (지정한 경우) 대상의 이름

kotlin2js 속성

이름	설명	필수
`output`	출력 파일	Yes
`libraries`	코틀린 라이브러리 경로	No
`outputPrefix`	생성할 자바스크립트 파일에 사용할 접두사	No
`outputSuffix`	생성할 자바스크립트 파일에 사용할 접미사	No
`sourcemap`	소스맵 파일을 생성할 위치	No
`metaInfo`	바이너리 디스크립터를 가진 메타데이터 파일을 생성할 위치	No
`main`	컴파일러가 생성한 코드가 메인 함수를 호출할지 여부	No

코틀린과 OSGi

코틀린 OSGi 지원을 활성화하려면 일반 코틀린 라이브러리 대신 `kotlin-osgi-bundle` 를 포함해야 한다. `kotlin-osgi-bundle` 이 `kotlin-runtime` , `kotlin-stdlib` , `kotlin-reflect` 를 포함하므로 이 세 의존을 제거한다. 또한, 외부 코틀린 라이브러리를 포함할 경우 주의해야 한다. 대부분 일반적인 코틀린 의존은 OSGi를 지원하지 않으므로 그것을 사용하지 말고 프로젝트에서 제거해야 한다.

메이븐

메이븐 프로젝트에 코틀린 OSGi 번들을 포함하려면 다음 설정을 사용한다:

```
    <dependencies>
         <dependency>
             <groupId>org.jetbrains.kotlin</groupId>
             <artifactId>kotlin-osgi-bundle</artifactId>
             <version>${kotlin.version}</version>
         </dependency>
     </dependencies>
```

외부 라이브러리에서 표준 라이브러리를 제외하려면 다음 설정을 사용한다("*" 표시 제외는 메이븐 3에서만 동작한다):

```
         <dependency>
             <groupId>some.group.id</groupId>
             <artifactId>some.library</artifactId>
             <version>some.library.version</version>

             <exclusions>
                 <exclusion>
                     <groupId>org.jetbrains.kotlin</groupId>
                     <artifactId>*</artifactId>
                 </exclusion>
             </exclusions>
         </dependency>
```

그레이들

다음 설정으로 그레이들 프로젝트에 `kotlin-osgi-bundle` 를 추가한다:

```
compile "org.jetbrains.kotlin:kotlin-osgi-bundle:$kotlinVersion"
```

의존 전이로 가져오는 기본 코틀린 라이브러리를 제외하려면 다음 접근을 사용한다:

```
dependencies {
 compile (
   [group: 'some.group.id', name: 'some.library', version: 'someversion'],
   .....) {
  exclude group: 'org.jetbrains.kotlin'
}
```

FAQ

왜 필요한 메니페스트 옵션을 모든 코틀린 라이브러리에 추가하지 않았나

비록 OSGi 지원을 제공하는 가장 선호하는 방법임에도 불구하고 "패키지 분리" 이슈 때문에 지금은 할 수 없다. 이 문제는 쉽게 없앨 수 없고 그런 큰 변화는 현재 계획에 없다. `Require-Bundle` 특징이 있지만 최선의 옵션이 아니며 사용을 권하지 않는다. 그래서 OSGi를 위한 아티팩트를 구분하기로 결정했다.

컴파일러 플러그인

all-open 컴파일러 플러그인

코틀린은 기본적으로 클래스와 멤버가 `final` 이다. 이는 `open` 인 클래스를 요구하는 스프링 AOP와 같은 프레임워크나 라이브러리를 사용할 때 불편을 준다. `all-open` 컴파일러 플러그인은 그런 프레임워크의 요구에 코틀린을 맞추고 `open` 키워드를 직접 사용하지 않아도 클래스에 특정 애노테이션을 붙이고 멤버를 open으로 만든다. 예를 들어 스프링을 사용할 때 모든 클래스를 open으로 만들 필요가 없다. 단지 클래스에 `@Configuration` 이나 `@Service` 와 같은 특정 애노테이션만 붙이면 된다. `all-open` 플러그인은 이 애노테이션을 지정하는 것을 허용한다.

그레이들, 메이븐, IDE를 위한 all-open 플러그인을 제공한다. 스프링의 경우 `kotlin-spring` 컴파일러 플러그인을 사용한다(아래 참고).

all-open 플러그인을 사용하는 방법

`build.gradle` 에 플러그인을 추가한다:

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"
    }
}

apply plugin: "kotlin-allopen"
```

또는 그레이들 플러그인 DSL을 사용한다면 `plugins` 블록에 all-open을 추가한다:

```
plugins {
  id "org.jetbrains.kotlin.plugin.allopen" version "1.1.51"
}
```

그리고 클래스를 open으로 만들 애노테이션을 지정한다:

```
allOpen {
    annotation("com.my.Annotation")
}
```

클래스(또는 그 상위타입)에 `com.my.Annotation` 을 붙이면, 클래스 그 자체와 클래스의 모든 멤버가 open이 된다.

또한 메타-애노테이션에도 동작한다:

```
@com.my.Annotation
annotation class MyFrameworkAnnotation

@MyFrameworkAnnotation
class MyClass // 모드 open이 된다
```

`MyFrameworkAnnotation` 에 `com.my.Annotation` 을 붙였으므로 이 애노테이션 또한 클래스를 open으로 만든다.

다음은 메이븐에서 all-open을 사용하는 방법이다:

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <!-- 또는 스프링 지원은 "spring" -->
            <plugin>all-open</plugin>
        </compilerPlugins>

        <pluginOptions>
            <!-- 각 줄에 애노테이션이 위치 -->
            <option>all-open:annotation=com.my.Annotation</option>
            <option>all-open:annotation=com.their.AnotherAnnotation</option>
        </pluginOptions>
    </configuration>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-allopen</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

kotlin-spring 컴파일러 플러그인 Kotlin-spring compiler plugin

스프링 애노테이션을 직접 지정할 필요는 없다. `kotlin-spring` 플러그인을 사용하면 자동으로 스프링의 요구에 맞게 all-open 플러그인을 설정한다:

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-allopen:$kotlin_version"
    }
}

apply plugin: "kotlin-spring"
```

또는 그레이들 플러그인 DSL을 사용한다:

```
plugins {
  id "org.jetbrains.kotlin.plugin.spring" version "1.1.51"
}
```

메이븐 예제는 위와 유사하다.

플러그인은 `@Component` , `@Async` , `@Transactional` , `@Cacheable` 애노테이션을 지정한다. 메타 애노테이션 지원 덕에 `@Configuration` , `@Controller` , `@RestController` , `@Service` , `@Repository` 를 붙인 클래스는 자동으로 open이 된다. 왜냐면 이들 애노테이션이 `@Component` 를 가진 메타 애노테이션이기 때문이다.

물론, 한 프로젝트에서 `kotlin-allopen` 과 `kotlin-spring` 을 사용할 수 있다. start.spring.io 를 사용하면 `kotlin-spring` 플러그인이 기본으로 활성화된다.

no-arg 컴파일러 플러그인

no-arg 플러그인은 지정한 애노테이션을 가진 클래스에 인자 없는 생성자를 추가로 생성한다. 생성한 생성자는 조작했기(synthetic) 때문에 자바나 코틀린에서 직접 호출할 수 없지만 리플렉션으로 호출할 수는 있다. 코틀린이나 자바 관점에서 `data` 클래스는 인자 없는 생성자가 없지만 JPA에서 `data` 클래스의 인스턴스를 생성할 수 있다(아래의 `kotlin-jpa` 설명을 보자).

no-arg 플러그인을 사용하는 방법

사용법은 all-open과 매우 유사하다. 플러그인을 추가하고 인자 없는 생성자를 생성하고 싶은 애노테이션 목록을 지정한다.

다음은 그레이들에서의 no-arg 사용법이다.

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-noarg:$kotlin_version"
    }
}

apply plugin: "kotlin-noarg"
```

또는 그레이들 플러그인 DSL을 사용한다:

```
plugins {
  id "org.jetbrains.kotlin.plugin.noarg" version "1.1.51"
}
```

인자 없는 생성자를 생성할 애노테이션 타입을 지정한다:

```
noArg {
    annotation("com.my.Annotation")
}
```

플러그인이 생성하는 인자 없는 생성자가 초기화 로직을 수행하도록 하려면 `invokeInitializers` 옵션을 활성화한다. 코틀린 1.1.3-2부터 앞으로 해결해야 할 KT-18667 와 KT-18668 때문에 기본으로 비활성화되었다:

```
noArg {
    invokeInitializers = true
}
```

다음은 메이븐에서의 no-arg 사용법이다:

```
<plugin>
    <artifactId>kotlin-maven-plugin</artifactId>
    <groupId>org.jetbrains.kotlin</groupId>
    <version>${kotlin.version}</version>

    <configuration>
        <compilerPlugins>
            <!-- 또는 JPA를 지원하려면 "jpa" 사용 -->
            <plugin>no-arg</plugin>
        </compilerPlugins>

        <pluginOptions>
            <option>no-arg:annotation=com.my.Annotation</option>
            <!-- 인조 생성자에서 인스턴스 초기화를 호출  -->
            <!-- <option>no-arg:invokeInitializers=true</option> -->
        </pluginOptions>
    </configuration>

    <dependencies>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-maven-noarg</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

kotlin-jpa 컴파일러 플러그인

이 플러그인은 인자 없는 생성자를 생성해야 할 마커로 `@Entity` 와 `@Embeddable` 을 사용한다. 그레이들에서 플러그인을 추가하는 방법은 다음과 같다:

```
buildscript {
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-noarg:$kotlin_version"
    }
}

apply plugin: "kotlin-jpa"
```

또는 그레이들 플러그인 DSL을 사용해도 된다:

```
plugins {
  id "org.jetbrains.kotlin.plugin.jpa" version "1.1.51"
}
```

메이븐 예제는 위 설정과 유사하다.

FAQ

FAQ

코틀린은 무엇인가?

코틀린은 OSS 정적 타입 프로그래밍 언어로서, JVM, 안드로이드, 자바스크립트, 네이티비를 대상으로 한다. 코틀린은 JetBrains 가 개발한다. 2010년에 프로젝트를 시작했고 초기부터 오픈 소스였다. 첫 번째 공식 1.0 버전은 2016년 2월에 나왔다.

현재 코틀린 버전은?

현재 릴리즈한 버전은 1.1.51로 September 29, 2017에 공개했다.

코틀린은 무료인가?

맞다. 코틀린은 무료이며, 무료였고, 앞으로도 무료일 것이다. 아파치 2.0 라이선스이며 깃헙에서 소스 코드를 구할 수 있다.

코틀린은 객체 지향 언어인가? 함수형 언어인가?

코틀린은 객체 지향과 함수형의 구성 요소를 모두 갖고 있다. OO와 FP 방식 또는 두 방식을 섞어서 사용할 수 있다. 고차 함수, 함수 타입과 람다와 같은 특징을 1급으로 지원하므로 함수형 프로그래밍을 하거나 알아보고 싶다면 코틀린이 훌륭한 선택이다.

코틀린이 자바 프로그래밍 언어에 비해 주는 장점은 무엇인가?

코틀린은 더 간결하다. 대략적인 측정 결과 코드 줄 수를 거의 40% 줄여준다. 또한 타입에 더 안전하다. 예를 들어, null 불가 타입은 어플리케이션에서 NPE를 줄여준다 스마트 변환, 고차 함수, 확장 함수, 리시버를 가진 람다를 포함한 다른 특징은 표현력 좋은 코드를 작성하는 능력을 제공하며 또한 DSL 작성을 촉진한다.

코틀린은 자바 프로그래밍 언어와 호환되나?

그렇다. 코틀린은 자바 프로그래밍 언어와 100% 호환되며, 기존 코드베이스를 코틀린에 알맞게 사용할 수 있도록 하는 것에 중점을 뒀다. 쉽게 자바에서 코틀린 코드를 호출할 수 있고 코틀린에서 자바 코드를 호출할 수 있다. 이는 적은 위험으로 매우 쉽게 코틀린을 적용할 수 있게 해준다. 또한 자바를 코틀린으로 자동 변환해주는 도구를 IDE에 탑재해서 기존 코드를 간단하게 마이그레이션할 수 있다.

코틀린을 어디에 쓸 수 있나?

서버사이드, 클라이언트 사이드 웹과 안드로이드 등 모든 종류의 개발에서 코틀린을 사용할 수 있고, 임베디드 시스템, 맥OS, iOS와 같은 다른 플랫폼을 지원하기 위한 코틀린/네이티브도 현재 작업 중에 있다. 어디에 사용하고 있는지 몇 가지만 말해보면, 모바일과 서버 사이드 어플리케이션, 자바 스크립트와 JavaFX를 위한 클라이언트 사이드, 데이터 과학에 코틀린을 사용하고 있다.

안드로이드 개발에 코틀린을 사용할 수 있나?

그렇다. 안드로이드는 1급 언어로 코틀린을 지원한다. 이미 Basecamp, Pinterest 등 수 백여개의 안드로이드 어플리케이션이 코틀린을 사용하고 있다. 안드로이드 개발 관련 자료 에서 더 많은 정보를 확인할 수 있다.

서버 사이드 개발에 코틀린을 사용할 수 있다?

그렇다. 코틀린은 JVM과 100% 호환되며, 스프링 부트, vert.x 또는 JSF와 같은 기존 프레임워크를 사용할 수 있다. 추가로 Ktor 와 같은 코틀린으로 작성한 전용 프레임워크도 있다. 서버 사이드 개발 관련 자료 에서 더 많은 정보를 확인할 수 있다.

웹 개발에 코틀린을 사용할 수 있나?

그렇다. 백엔드 웹에서 사용할 수 있을 뿐만 아니라, 클라이언트 사이드 웹에 코틀린/JS를 사용할 수 있다. 많이 쓰이는 자바스크립트 라이브러리에 대한 정적 타입을 얻기 위해 코틀린은 DefinitelyTyped 의 정의를 사용할 수 있고, AMD나 CommonJS 같은 기존의 모듈 시스템과 호환된다. 클라이언트 사이드 개발 관련 자료 에서 더 많은 정보를 확인할 수 있다.

데스크톱 개발에 코틀린을 사용할 수 있나?

그렇다. JavaFx, 스윙과 같은 자바 UI 프레임워크를 사용할 수 있다. 추가로 TornadoFX 와 같은 코틀린 전용 프레임워크가 존재한다.

네이티브 개발에 코틀린을 사용할 수 있나?

현재 코틀린/네이티브를 작업 중 이다. 코틀린/네이티브는 코틀린을 JVM 없이 실행할 수 있는 네이티브 코드로 컴파일한다. 테크놀러지 프리뷰 릴리즈가 있는데 아직 제품으로 준비된 상태는 아니며, 1.0 버전에서는 모든 플랫폼을 지원할 예정은 아니다. 코틀린/네이브티 소개 블로그 포스트 에서 더 많은 정보를 확인할 수 있다.

코틀린을 지원하는 IDE는?

인텔리J IDEA , 안드로이드 스튜디오 , 이클립스 , 넷빈즈 를 포함한 주요 자바 IDE에서 코틀린을 지원한다. 또한 명령행 컴파일러 를 사용해서 컴파일하고 어플리케이션을 실행할 수 있다.

코틀린을 지원하는 빌드 도구는?

JVM 영역에서는 그레이들 , 메이븐 , 앤트 , 코발트 등 주요 빌드 도구를 지원한다. 클라이언트 사이드 자바스크립트 대상으로 사용가능한 빌드 도구도 있다.

코틀린은 무엇으로 컴파일되나?

JVM을 대상으로 할 때, 코틀린은 자바와 호환되는 바이트코드를 생성한다. 자바스크립를 대상으로 할 때, 코틀린을 ES5.1로 트랜스파일하며 AMD와 CommonJS를 포함한 모듈 시스템에 호환되는 코드를 생성한다. 네이티브를 대상으로 할 때, 코틀린은 (LLVM을 통해) 플랫폼 전용 코드를 생성할 것이다.

코틀린은 자바 6만 대상으로 할 수 있나?

아니다. 코틀린은 자바 6이나 자바 8에 호환되는 바이트코드 생성을 선택할 수 있다. 더 높은 버전의 플랫폼에 대해 더 최적화된 바이트 코드를 생성한다.

코틀린은 어렵나?

코틀린은 자바, C#, 자바스크립트, 스칼라, 그루비와 같은 기존 언어에서 영향을 받았다. 코틀린을 배우기 쉽게 하려고 노력했으며, 코틀린을 시작해서 몇 일이면 쉽게 코틀린을 읽고 작성할 수 있을 것이다. 코틀린의 특징을 배우고 몇 가지 고급 특징을 배우는데는 좀 더 걸리겠지만, 전반적으로 코틀린은 복잡한 언어가 아니다.

어떤 회사에서 코틀린을 사용하나?

코틀린을 회사는 너무 많아서 나열할 수 없다. 블로그나 깃헙 리포지토리, 발표를 통해 코틀린 사용을 공개적으로 선언한 회사에는 Square , Pinterest , Basecamp 가 있다.

누가 코틀린을 개발하나?

코틀린은 주로 젯브레인의 엔지니어 팀이 개발한다(현재 40명 이상 참여). 언어 설계를 이끄는 사람은 Andrey Breslav 이다. 핵심 팀 외에 깃헙에 100명 이상의 외부 공헌자가 존재한다.

어디서 코틀린을 더 배울 수 있나?

가장 좋은 출발점은 코틀린 웹사이트 이다. 거기서 컴파일러를 다운로드 받을 수 있고, 온라인으로 시도 할 수 있고, 또한 레퍼런스 문서 와 튜토리얼 , 자료를 구할 수 있다.

코틀린에 대한 책이 있나?

코틀린 팀 멤버인 Dmitry Jemerov와 Svetlana Isakova가 쓴 Kotlin in Action , 안드로이드 개발자를 대상으로 한 Kotlin for Android Developers 등 이미 코틀린에 대한 책이 여러 개 있다.

코틀린을 배울 수 있는 온라인 강좌가 존재하나?

Kevin Jones의 Pluralsight Kotlin Course , Hadi Hariri의 O'Reilly Course , Peter Sommerhoff의 Udemy Kotlin Course 등 몇 개의 코틀린 강좌가 존재한다. 또한 유튜브와 비메오에서 많은 Kotlin talks 를 볼 수 있다.

코틀린 커뮤니티가 있나?

있다. 코틀린은 매우 활발한 커뮤니티를 갖고 있다. 코틀린 포럼 , 스택오버플로우 , 에서 코틀린 개발자를 만날 수 있고, 코틀린 슬랙 (2017년 5월 기준 거의 7000명의 멤) 에서 더 활발하게 활동하고 있다.

코틀린 이벤트가 있나?

있다. 코틀린에만 초점을 둔 많은 사용자 그룹과 미트업이 있다. 웹 사이트의 목록 에서 찾을 수 있다. 또한 전세계적으로 커뮤니티가 주도하는 코틀린 나이트 이벤트가 열린다.

코틀린 컨퍼런스가 있나?

있다. 첫 번째 공식 KotlinConf 이 2017년 11월 2-3일에 샌프란시스코에서 열린다. 또한 전세계적으로 다른 컨퍼런스에서 코틀린을 다루고 있는 중이다. 웹 사이트에서 예정된 발표 목록을 확인할 수 있다.

소셜 미디어에 코틀린이 있나?

있다. 가장 활발히 활동하는 소셜미디어는 트위터 계정 이다. 구글+ 그룹 도 있다.

다른 온라인 코틀린 자료는?

커뮤니티가 멤버가 만든 코틀린 다이제스트 , 뉴스레터 , 팟캐스트 등의 사이트를 온라인 자료 에서 확인할 수 있다.

코틀린 HD 로고는 어디서 구할 수 있나?

여기 에서 로고를 다운로드할 수 있다. 아카이브에 포함된 `guidelines.pdf` 의 간단한 규칙을 따라주길 바란다.

자바 프로그래밍 언어와 비교

코틀린에서 해결한 몇 가지 자바 이슈

코틀린은 자바에서 겪는 몇 가지 이슈를 고쳤다:

- 널 레퍼런스는 타입 시스템으로 제어 함
- raw 타입 없음
- 코틀린 배열은 공변 임
- 자바의 SAM 변환과 달리 코틀린에는 제대로 된 함수 타입 이 있음
- 와일드카드 없는 사용 위치 변성
- 코틀린에는 checked 익셉션 이 없음

자바에는 있는데 코틀린에는 없는 것

- Checked 익셉션
- 클래스가 아닌 기본 타입
- 정적 멤버
- private 아닌 필드
- 와일드카드 타입

자바에는 없는데 코틀린에 있는 것

- 람다 식 + 인라인 함수 = 더 효율적인 커스텀 제어 구조
- 확장 함수
- Null-안정성
- 스마트 변환
- 문자열 템플릿
- 프로퍼티
- 주요 생성자
- 1급 위임
- 변수와 프로퍼티 타입에 대한 타입 추론
- 싱글톤
- 선언 위치 변성과 타입 프로젝션
- 범위 식
- 연산자 오버로딩
- 컴패니언 오브젝트
- 데이터 클래스
- 읽기 전용과 변경 가능 콜렉션의 인터페이스 분리
- 코루틴