Functions and Lambdas

Functions

함수 선언

```
fun double(x: Int): Int {
}
```

- fun 으로 함수 선언
- (파라미터 이름: 타입)
- 함수명(): 타입

중위 표기법(Infix notation)

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

// 중위 표기법을 사용해서 함수 호출
1 shl 2 // 1.shl(2)
1 + 2 // 1.+(2)
```

- 멤버 함수이거나 확장(extention) 함수인 경우,
- 파라미터가 하나인 경우,
- infix로 명시된 경우
  마침표와 괄호를 생략해서 사용 가능하다.

기본 인자(dafualt arguments)

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

read(byteArray) // off와 len은 기본값으로 적용됨
```

- 파라미터에 위와 같이 기본값을 설정하면 함수 사용시 인자를 안넣어도 된다.

```
open class A {
    open fun foo(i: Int = 10) { ... }
}
class B : A() {
    override fun foo(i: Int) { ... }  // no default value allowed
}

// 사용 예
val b = B()
b.foo() // ok
b.foo(2) // ok
```

- 상속받아서 함수를 override 한 경우 기본 값을 정의할 수 없다. (하면 오류남)

이름이 있는 인자(named arguments)

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

// java에서 일반적으로 사용시
reformat(str, true, true, false, '_')

// 이름을 사용하면 가독성이 좋아진다.
reformat(str,
        normalizeCase = true,
        upperCaseFirstLetter = true,
        divideByCamelHumps = false,
        wordSeparator = '_'
        )

// 기본인자와 같이 사용시 더욱 간결해진다.
reformat(str, wordSeparator = '_')
```

- 자바의 byte 코드는 파라미터의 이름을 유지하지 않기 때문에 자바 함수를 사용할때 이름이 있는 인자처럼 사용하지 않도록 유의

Unit 리턴 함수

```
fun printHello(name: String?): Unit { // Unit
  ...
  // `return Unit` 또는 `return` 생략 가능
}

// Unit타입도 생략 가능
fun printHello(name: String?) {
  ...
}
```

- 자바의 void와 유사

단일 표현 함수(Single-Expression functions)

```
// 중괄호 생략한 경우
fun double(x: Int): Int = x * 2

// 컴파일러가 타입추론도 가능하면 리턴타입 생략 가능
fun double(x: Int) = x * 2
```

명시적 리턴 타입(Explicit return types)

- Kotiln은 기본적으로 블록(중괄호)으로 정의된 함수인 경우 리턴 타입을 반드시 명시해야 한다.
- 블록 본문은 복잡한 제어 흐름을 가질 수 있고, 리턴 타입이 명백하지 않기 때문이다.

가변인자 (Varargs=variable arguments)

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

- 자바에서는 `(String... str)` 으로 가변인자를 표현했던 것에 비해 코틀린에서는 파라미터 앞에 `vararg` 을 붙임을써 표현한다.
- 코드에서 ts를 Array로 인식한다.

```
val a = arrayOf(1, 2, 3) // n개로 추가 가능
val list = asList(-1, 0, *a, 4) // '*' 표시로 배열을 가변인자로 전달
```

- 배열을 가변인자로 넣고 싶을 땐 `*` 를 앞에 넣어준다.
- 파라미터가 n개 인 경우 오직 하나의 파라미터만 가변인자 `vararg` 로 지정할 수 있다.

- 가변인자가 마지막 파라미터가 아닐 경우 가변인자 뒤에 파라미터는 named 파라미터로 사용해야 한다.

함수 범위

최상위 함수(top-level or package-level function)

- 코틀린에서는 함수를 파일에 최상위 레벨로 정의할 수 있다.
- java에서 Util성 public static 함수라고 생각하면 된다.
- 자바, C#, 스칼라처럼 함수를 정의하기 위해 일부러 클래스를 만들 필요가 없다.

```
// test.kt (파일명은 아무렇게나 해도 된다.)
@file:JvmName("Utils") // 컴파일후에는 Utils라는 클래스 생성
package test

fun foo() {
}
```

- 코틀린 파일을 생성 (클래스가 아님)
- 패키지 위에 `@file:JvmName("이름")` 으로 정의

```
test.foo() // 이렇게 아무곳에서나 사용 가능
```

지역 함수(local function)

- 함수 안에서 함수를 정의할 수 있다.

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

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

- 상위 함수범위 안에서 지역함수는 지역변수에 접근이 가능
- 아래와 같이 변경이 가능하다.

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>() // visited가 밖으로 빠짐
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return // 같은 local 범위 안에 있으므로 접근 가능
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0]) // Hashset() 이 없어짐
}
```

멤버 함수

- 일반적 함수

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

```
Sample().foo() // java에서는 new Sample().foo();
```

제너릭 함수

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

- 함수 앞에 `<T>` 사용해서 제너릭 함수로 정의

인리인(Inline) 함수

- 다다음장에..

확장(Extension) 함수

- 이미 배웠음..

고차(Higher-Order) 함수와 람다

- 다음장에..

꼬리 재귀 함수

```
// 요건 자바
private fun findFixPoint(): Double {
    var x = 1.0
    while (true) {
        val y = Math.cos(x)
        if (x == y) return y
        x = y
    }
}
// 요건 코틀린
tailrec fun findFixPoint(x: Double = 1.0): Double
        = if (x == Math.cos(x)) x else findFixPoint(Math.cos(x))
```

- 코틀린은 재귀함수를 스택오버플로우의 리스크가 없는 루프로 바꾸는 알고리즘을 허락한다.
- 함수 앞에 `tailrec` (tail recursion)을 붙이고 컴파일러가 최적화가 가능한 조건이 맞는다고 판단하면 효율적인 루프 기반으로 바뀐다.
- 반드시 마지막에 자기 자신을 호출해야 하며, try/catch/finally 구역에서는 사용하지 못한다.
- JVM backend 에서만 지원

Higher-Order Functions and Lambdas

고차 함수(Higher-Order Functions)

- 고차함수는 파라미터로 함수를 받거나 함수를 리턴하는 함수이다.

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

- 위 함수는 Lock 오브젝트를 받아서 `lock()` 을 요청하고 파라미터로 받은 `body()` 함수 실행 후 `unlock()` 을 하는 함수이다.
- `() -> T` 는 파라미터가 없으며 리턴타입이 T인 함수를 나타낸다.

```
// 사용 예제1
fun toBeSynchronized() = sharedResource.operation()
val result = lock(lock, ::toBeSynchronized) // '::'은 Other의 Reflection 챕터에서 다룰 예정

// 사용 예제2
val result = lock(lock, { sharedResource.operation() })

// 사용 예제3
lock (lock) {
    sharedResource.operation()
}
```

- 우리가 썼던 map()이란 함수가 바로 고차함수이다.

```
// map() 함수 구현부
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
}
// 사용 예제
val doubled = ints.map { value -> value * 2 }
```

it: 단일 파라미터의 implicit 이름

```
// 예제1 함수 리터럴의 파라미터가 하나라면 -> 를 포함해서 생략 가능하다.
ints.map { it * 2 }

// 예제2 LINQ-스타일(Language Integrated Query / SQL문처럼 쿼리식을 추가하여 확장하는 스타일)
strings.filter { it.length == 5 }.sortBy { it }.map { it.toUpperCase() }
```

사용되지 않은 변수의 언더스코어(Underscore for unused variables) (since 1.1)

- 변수를 정의하고 사용하지 않으면 `unused variables` 이라는 warning이 발생한다.
- 파라미터가 하나인 경우 `_`로 대체 가능하기 때문에 `unused variables`라 부르는 것 같다.
- 람다 파라미터가 사용되지 않은 경우 `_`를 사용가능하다.

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

람다에서의 비구조화?(Destructuring in Lambdas) (since 1.1)

- Other 파트에서..

인라인 함수

- 다음장에..

람다 표현식과 익명 함수(Lambda Expressions and Anonymous Function)

- '함수 리터럴'로 선언 없이 바로 사용 가능하다.

```
// { a, b -> a.length < b.length }를 람다가 아닌 일반적인 방식으로 선언시
fun compare(a: String, b: String): Boolean = a.length < b.length

// '함수 리터럴'을 사용한 예제, 파라미터에 함수를 바로 사용함
max(strings, { a, b -> a.length < b.length })
```

함수 타입

- 위에서 보여준 `max()`의 정의

```
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)) // 여기서 less() 다시 쓰임
            max = it
    return max
}
```

- 함수를 파라미터로 받으려면 파라미터를 함수로 만들면 된다.
- `less` 파라미터 타입은 `(T, T) -> Boolean`이다.
- 파라미터의 의미를 명시하고 싶으면 이름을 붙여도 된다.

```
less: (x: T, y: T) -> Boolean
```

- 함수 타입 자체가 nullable 일 경우 아래와 같이 작성한다.

```
var sum: ((Int, Int) -> Int)? = null
```

람다 표현식 구문

- 람다 표현식의 전체 구문은 아래와 같다.

```
val sum = { x: Int, y: Int -> x + y }
```

- 괄호 `{}` 안에서 타입을 생략할 수 있는데 그럴경우 아래와 같이 작성하면 된다.

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

- 파라미터가 한개인 경우 생략 가능하며 `it` 을 사용한다.

```
ints.filter { it > 0 } // 이 리터럴 타입은 '(it: Int) -> Boolean' 이다.
```

익명 함수

람다 함수에서 명시적으로 리턴 타입을 정의하고 싶으면 '익명함수'를 사용하면 된다.

```
// 식
fun(x: Int, y: Int): Int = x + y

// 또는 블럭
fun(x: Int, y: Int): Int {
  return x + y
}

// 일반 함수와 동일하게 파라미터 타입도 유추 가능
ints.filter(fun(item) = item > 0)
```

- 블록 함수가 있는 익명함수는 `return` 을 꼭 명시해야 한다.
- 익명함수에서는 파라미터를 항상 괄호 `()` 안에서 전달한다.
  괄호 밖에서 허용하는 약식 구문은 람다식에서만 가능
- 람다 식과 익명 함수의 또 다른 차이점은 비-로컬 리턴(non-local return)(다음장에 자세히..)
  값의 동작 이다.
  레이블이없는 return 문은 항상 `fun` 키워드로 선언 된 함수에서 반환된다.
  즉, 람다 식 내부의 반환은 둘러싼 함수에서 반환되는 반면, 익명 함수 내부의 반환은 익명 함수 자체에서 반환된다.

클로져

- 익명함수나 람다식에서도 클로져(즉, 외부 범위에 선언된 변수)에 접근이 가능하다.
- 자바와 다르게(?) 수정도 가능

```
var sum = 0
ints.filter { it > 0 }.forEach {
    sum += it
}
print(sum)
```

리시버를 갖는 함수 리터럴

- 코틀린은 지정된 리시버 객체(`.` 기호 앞에 전달되는 객체, 함수 및 프로퍼티에 바로 접근할 수 있도록 할 객체)를 이용하여 함수 리터럴을 호출하는 기능을 제공한다.
- 추가적인 수식어구 없이 호출이 가능하다.
- 함수의 본문 내부에 있는 리시버 객체의 멤버에 액세스 할 수 있는 확장 함수와 유사하다.

```
// 리시버가 있는 함수 유형
sum : Int.(other: Int) -> Int
// 사용시
1.sum(2)
// 익명함수로 사용시 리시버 타입을 직접 지정할 수 있음
// 나중에 사용해야하는 경우 유용
val sum = fun Int.(other: Int): Int = this + other
```

- 문맥에서 리시버 타입을 유추할 수 있다면 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(무슨말인지..?)
    return html
}
html {       // 리시버를 가진 람다(?) 시작
    body()   // 리시버 객체의 함수 호출(위에서 HTML클래스에 정의된 body() 함수)
}
```

인라인 함수(inline function)

- 고차함수를 사용하면 메모리 할당과 버추얼 호출로 런타임에 어느정도 불이익이 발생한다.
- 이런 경우 람다식을 인라인해서 부하를 제거할 수 있다.

```
fun lock(l: Any, work: () -> Unit)  {
    l.lock()
    try {
      work()
    } finally {
      l.unlock()
    }
}

// 사용시
lock(l) { foo() }
```

- 위의 코드처럼 작성했을 경우 `{ foo() }`자체도 객체이기 때문에 메모리에 올라가 `lock()` 함수를 호출할 때 비용이 발생한다.
- 반면 `inline` 키워드를 앞에 붙였을 경우에는 아래와 같이 컴파일러에서 코드를 생성하여 완성시킨다.

```
inline fun lock(l: Any, work: () -> Unit)  {
    l.lock()
    try {
        work()
    } finally {
        l.unlock()
    }
}

lock(l) { foo() }  // 이 코드를 컴파일러가 아래와 같이 바꿔준다
//-----------------------
l.lock()
try {
    foo()
} finally {
    l.unlock()
}
//-----------------------
```

- `inline` 제한자가 함수 자체와 함수에 전달하는 람다에 영향을 주고, 해당 람다가 인라인 된다.
- 인라인시 생성된 코드가 커질 수 있지만, 합리적인 방법으로 처리하면 루프 내부의 거대(megamorphic) 호출 구역에서 효가 있다.
- 일반적으로 filter, map 같은 함수에 inline이 적용된다.

```
// filter 원문
public inline fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T> {
    return filterTo(ArrayList<T>(), predicate)
}

// map 원문
public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
    return mapTo(ArrayList<R>(collectionSizeOrDefault(10)), transform)
}
```

noinline

- 인라인 함수에 전달된 다수의 람다 파라미터중 일부 람다만 인라인되길 원하면 `noinline` 제한자로 지정하면 된다.

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) {
    // ...
}
```

- 인라인 함수 정의시 인라인 가능한 함수 파라미터가 없고 `reified` 타입(아래에 나옴) 파라미터가 없으면, 컴파일러는 `inline` 함수로써 이점이 없다는 경고를 발생시킨다.

비-로컬 리턴(Non-local returns)

- 코틀린에서 이름을 가진 함수나 익명함수에서 나가려면 `return`만 사용할 수 있다.
- 람다에서 나가려면 label(`@` 사용해서 리턴했던 것)을 사용해야 한다.
- 람다 안에서는 단순 `return`을 허용하지 않는데, 람다는 둘러싼 함수를 리턴할 수 없기 때문

```
fun foo() {
    일반적 람다 함수 {
        return // ERROR: 여기서 foo()를 리턴할 수 없다.
    }
}

fun foo() {
    인라인 함수 {
        return // OK: 람다가 전달되어 인라인된 함수라면 리턴도 인라인 가능해서 허용됨 (??)
    }
}
```

- 위에처럼 람다에 위치하지만 둘러싼 함수를 나가는 리턴을 비-로컬 리턴(non-local return)이라고 한다. 일반적으로 루프를 사용하는 경우에 아래와 같이 확인할 수 있다.

```
fun hasZeros(ints: List<Int>): Boolean {
    ints.forEach {
        if (it == 0) return true // returns from hasZeros
    }
    return false
}
```

- 일부 인라인 함수는 파라미터로 전달받은 람다를 호출할 때 함수 몸체에서 직접 호출하지 않고 다른 실행 컨텍스트를 통해(예, 로컬 객체나 중첩 함수) 호출해야 할 때가 있다. 이 경우 람다 안에서 비-로컬 흐름을 제어할 수 없다. 이를 지정하려면 람다 파라미터에 `crossinline` 제한자를 붙이면 된다.

```
inline fun f(crossinline body: () -> Unit) {
    val f = object: Runnable {
        override fun run() = body()
    }
    // ...
}
```

- 인라인된 람다에서 break와 continue는 아직 사용할 수 없는데, 앞으로 지원할 계획이다.

Reified(구체화된?) 타입 파라미터

- 때때로 파라미터로 전달한 타입에 접근해야 할 때가 있다.
- 아래 코드는 노드가 특정 타입을 가졌는지 확인하기 위해 트리를 탐색하고 리플렉션을 사용하는 예이다.

```
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?
}
```

- 사용시 코드가 아름답지 않다.

```
treeNode.findParentOfType(MyTreeNode::class.java)
```

- 아래와 같이 사용하기를 원한다.

```
treeNode.findParentOfType<MyTreeNode>()
```

- 이렇게 할 수 있도록 인라인 함수는 reified 타입 파라미터를 지원한다. 이를 사용한 코드는 다음과 같다.

```
inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p.parent
    }
    return p as T?
}
```

- 타입 파라미터에 `reified` 제한자를 적용하면 마치 클래스처럼 타입 파라미터에 접근할 수 있다.
- 인라인 함수이므로 !is 나 as 와 같은 일반 연산자가 동작한다. 또한 앞서 언급한 `myTree.findParentOfType<MyTreeNodeType>()` 처럼 호출할 수 있다.
- 일반 함수는 `reified` 파라미터를 가질 수 없다.
- 런타임 표현을 갖지 않는 타입(`reified` 타입 파라미터가 아니거나 `Nothing`과 같은 가공 타입)은 `reified` 타입 파라미터를 위한 인자로 사용할 수 없다.

인라인 프로퍼티 (since 1.1)

- val 또는 var의 getter, setter 그리고 프로퍼티 자체에 `inline`을 붙일 수 있는데, 어떻게 쓰는지 잘 모르겠다.

```
val foo: Foo
    inline get() = Foo()

var bar: Bar
    get() = ...
    inline set(v) { ... }
```

```
// 아래처럼 변수 앞에 사용하면 getter, setter에 자동으로 붙음
inline var bar: Bar
    get() = ...
    set(v) { ... }
```

- accessors 와 backing field를 지원하지 않은 프로퍼티에 inline을 붙임으로써 지원 가능하다.
- 사용 참고사이트

코루틴(Coroutines) (since 1.1)

`코루틴은 1.1에서 실험적이다.`

- 코루틴을 사용하고 싶으면 `build.gradle` 파일에 다음 코드가 필요하다.

```
apply plugin: 'kotlin'

// 요기
kotlin {
  experimental {
    coroutines 'enable'
  }
}
```

- 일부 api(네트워크 IO, 파일 IO, CPU- or GPU-집약적 작업 등)는 완료될 때 까지 호출자를 차단하도록 요구한다. 코루틴은 스레드를 차단하지 않고 더 저렴하고 더 제어 가능한 작업을 제공한다.
- 코루틴은 복자합게 만드는 비동기 프로그래밍을 단순화 시킨다.
- 라이브러리는 사용자 코드의 관련 부분을 콜백으로 래핑하고, 관련 이벤트를 subscribe하고, 다른 스레드 (또는 다른 시스템)에서 실행을 스케줄링 할 수 있다.

Blocking vs Suspending

- 일반적으로 부하가 큰 로드에서 스레드를 차단하면 상대적으로 적은 수의 스레드가 동작하기 때문에 비용이 많이 든다. 또한 지연될 수 있다.
- 반면 코루틴은 비용이 들지 않는다. cotext 변환이나 다른 os의 개입이 필요없다.

- 사용자 라이브러리에 의해 서스펜션을 컨트롤 할 수 있다.(?)
- 또한 무작위 명령에 의해 중단될 수 없으며, 구체적으로 마킹된 지점에서만 중단이 가능하다.

Suspending functions(중단 함수?)

```
suspend fun doSomething(foo: Foo): Bar {
    ...
}
```

- 위의 함수를 호출하면 코루틴이 멈출수 있으므로 중단함수라고 한다.
- 중단함수는 일반함수처럼 사용할 수 있지만 코루틴 및 다른 중단함수에서만 사용할 수 있다.
- 코루틴을 시작하려면 적어도 하나의 중단함수가 있어야 하고, 일반적으로 익명(중단 람다)이다.(?)

```
fun <T> async(block: suspend () -> T)
```

- 위의 async함수는 일반적인 함수이나, block 파라미터는 `supend () -> T`라는 중지 함수 타입을 갖는다.
- `async()`에 람다를 전달하면 이것은 중지람다(suspending lambda)라고 하며 아래와 같이 사용가능하다.

```
async {
    doSomething(foo)
    ...
}
```

- 아래 함수에서 `await()`함수는 일부 계산이 끝나고 결과를 리턴할때까지 코루틴을 중단하는 중단함수(그래서 `async{}` 안에서 사용 가능)일 수 있다.

```
async {
    ...
    val result = computation.await()
    ...
}
```

- 아래와 같이 중단함수인 `await()`과 `doSometing()`함수는 일반 함수안에서 사용할 수 없다.

```
fun main(args: Array<String>) {
    doSomething() // ERROR: 코루틴이 아닌 컨텍스트에서 중지함수 호출 안됨
}
```

- 또한 중지 함수는 인터페이스의 함수일 수 있으며, override 할 때에도 `suspend`를 명시해야 한다.

```
interface Base {
    suspend fun foo()
}

class Derived: Base {
    override suspend fun foo() { ... }
}
```

- 아래는 일반적인 예

```
// runs the code in the background thread pool
fun asyncOverlay() = async(CommonPool) {
    // start two async operations
    val original = asyncLoadImage("original")
    val overlay = asyncLoadImage("overlay")
    // and then apply overlay to both results
    applyOverlay(original.await(), overlay.await())
}

// launches new coroutine in UI context
launch(UI) {
    // wait for async overlay to complete
    val image = asyncOverlay().await()
    // and then show it in UI
    showImage(image)
}
```

@RestrictsSuspension 주석

- 사용자가 확장할 수 있는 DSLs(Other의 Type-Safe Builders 챕터에서 등장)이나 API를 만들 수 있는데 라이브러리는 사용자가 코루틴을 일시 중단하는 새로운 방법을 찾지 못하도록 해야한다.
- 이 때 `@RestrictsSuspension` 주석을 사용한다.
- 모든 중단이 특수한 방식으로 라이브러리에서 처리되는 드문(rare) 경우에 적합하다.
- 예를 들어, 아래에서 설명하는 `buildSequence()` 함수를 통해 생성기를 구현할 때 코루틴의 일시 중단 호출이 `yield()` 또는 `yieldAll()`을 호출하고 다른 함수는 호출하지 않도록해야 한다.

```
val fibonacciSeq = buildSequence { // buildSequence 구현시 SequenceBuilder가 동작함
    var a = 0
    var b = 1

    yield(1)

    while (true) {
        yield(a + b)

        val tmp = a + b
        a = b
        b = tmp
    }
}

// 사용시
println(fibonacciSeq.take(8).toList())
// 결과 = [1, 1, 2, 3, 5, 8, 13, 21]
```

- 이것이 `SequenceBuilder`가 `@RestrictsSuspension`으로 주석 된 이유이다.

```
@RestrictsSuspension
public abstract class SequenceBuilder<in T> {
    ...
}
```

코루틴의 내부 동작 방식

- 내부 동작을 자세하게 설명하진 않지만 감을 잡는게 중요하다. (뭐라고?)
- 코루틴은 컴파일 기술을 통해 완벽하게 구현되며(VM 또는 OS 차원의 지원 필요 없음), 코드 변환을 통해 중지가 동작된다.
- 모든 중지 함수는 중지가 호출되는 상태 시스템으로 변환된다.(?)
- 중지 직전의 상태는 로컬 변수와 함께 컴파일러의 생성된 클래스 필드에 저장된다.
- 해당 코루틴을 다시 시작할 때 로컬 변수가 복원되고 중단 직후 상태에서 진행된다.

- 중지된 코루틴은 객체로 저장되는데, 이러한 객체의 유형을 `Continuation`이라 하며, 전체 코드 변환은 classical한 Continuation-passing style 에 해당한다.(??)

- 코틀린의 코루틴에 회의적인 포스팅
    - 결론은 rxJava의 Observables이 코루틴보다 더 좋다는..