* ${}를 사용하여 print 문 텍스트의 변수와 계산 값을 둘러쌉니다. 예를 들어 ${age}에서 age가 변수입니다.
* val 키워드와 이름을 사용하여 변수를 만듭니다. 설정이 완료되면 변경할 수 없습니다. 등호를 사용하여 변수에 값을 할당합니다. 값의 예로는 텍스트와 숫자가 있습니다.
* String은 "Hello"와 같이 따옴표로 묶인 텍스트입니다.
* Int는 양의 정수 또는 음의 정수(예: 0, 23, -1024)입니다.
* 함수에서 사용할 인수 한 개 이상을 함수에 전달할 수 있습니다. 예:fun printCakeBottom(age:Int, layers:Int) {}
* repeat() {} 문을 사용하여 일련의 명령어를 여러 번 반복합니다. 예를 들면 repeat (23) { print("%") } 또는 repeat (layers) { print("@@@@@@@@@@") }이 있습니다.
* *루프*는 명령어를 여러 번 반복하는 명령어입니다. repeat() 문은 루프의 예입니다.
* 루프를 중첩할 수 있습니다. 즉, 루프 내에 루프를 배치할 수 있습니다. 예를 들어 repeat() 문 내에 repeat() 문을 만들어 케이크 층을 만들 때처럼 여러 행에 걸쳐 기호를 여러 번 출력할 수 있습니다.

**함수 인수 사용 요약:** 함수에 인수를 사용하려면 다음 세 가지 작업을 실행해야 합니다.

* 함수 정의에 인수와 유형을 추가합니다. printBorder(border: String)
* 함수 내에서 인수를 사용합니다. println(border)
* 함수 호출 시 인수를 제공합니다. printBorder(border)

[6. 요약](https://developer.android.com/codelabs/basic-android-kotlin-training-conditional-dice-roll-behavior?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-four%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-conditional-dice-roll-behavior#5)

* if 문을 사용하여 일부 명령을 실행하는 조건을 설정합니다. 예를 들어 사용자가 행운의 숫자를 굴리면 승리 메시지를 출력합니다.
* Boolean 데이터 유형에는 true 및 false라는 값이 있고 의사 결정에 사용할 수 있습니다.
* 초과(>), 미만(<), 같음(==) 등의 연산자를 사용하여 값을 비교합니다.
* else if 문 체인을 사용하여 여러 조건을 설정합니다. 예를 들어 가능한 각 주사위 굴리기에 다른 메시지를 출력합니다.
* 조건 체인 끝에 else 문을 사용하여 명시적으로 커버되지 않을 수 있는 사례를 포착합니다. 6면 주사위 사례를 커버하면 else 문이 8면 주사위로 굴린 값인 숫자 7과 8을 포착합니다.
* 값 비교에 따라 코드를 실행하는 간단한 형식으로 when 문을 사용합니다.

**if-else의 일반 형식**

**if (*condition-is-true*) {**

***execute-this-code***

**} else if (*condition-is-true*) {**

***execute-this-code***

**} else {**

***execute-this-code***

**}**

**when 문**

**when (*variable*) {**

**matches-value -> *execute-this-code***

**matches-value -> *execute-this-code***

**...**

**}**

[7. 요약](https://developer.android.com/codelabs/basic-android-kotlin-training-classes-and-inheritance?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-2-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-classes-and-inheritance#6)

이 Codelab을 통해 학습한 내용은 다음과 같습니다.

* 하위 클래스가 상위 클래스에서 기능을 상속받는 클래스 트리인 클래스 계층 구조를 만드는 방법. 속성과 함수가 서브클래스에 상속됩니다.
* 일부 기능을 서브클래스에서 구현하도록 남기는 abstract 클래스를 만드는 방법. 따라서 abstract 클래스는 인스턴스화할 수 없습니다.
* abstract 클래스의 서브클래스를 만드는 방법
* override 키워드를 사용하여 서브클래스의 속성과 함수를 재정의하는 방법
* super 키워드를 사용하여 상위 클래스의 함수와 속성을 참조하는 방법
* 서브클래스로 분류할 수 있도록 클래스를 open으로 만드는 방법
* 속성을 private으로 만들어 클래스 내에서만 사용할 수 있도록 하는 방법
* with 구문을 사용하여 동일한 객체 인스턴스에서 여러 호출을 실행하는 방법
* kotlin.math 라이브러리에서 기능을 가져오는 방법
* /\*\*  
  \* Program that implements classes for different kinds of dwellings.  
  \* Shows how to:  
  \* Create class hierarchy, variables and functions with inheritance,  
  \* abstract class, overriding, and private vs. public variables.  
  \*/  
    
  import kotlin.math.PI  
  import kotlin.math.sqrt  
    
  fun main() {  
     val squareCabin = SquareCabin(6, 50.0)  
     val roundHut = RoundHut(3, 10.0)  
     val roundTower = RoundTower(4, 15.5)  
    
     with(squareCabin) {  
         println("\nSquare Cabin\n============")  
         println("Capacity: ${capacity}")  
         println("Material: ${buildingMaterial}")  
         println("Floor area: ${floorArea()}")  
     }  
    
     with(roundHut) {  
         println("\nRound Hut\n=========")  
         println("Material: ${buildingMaterial}")  
         println("Capacity: ${capacity}")  
         println("Floor area: ${floorArea()}")  
         println("Has room? ${hasRoom()}")  
         getRoom()  
         println("Has room? ${hasRoom()}")  
         getRoom()  
         println("Carpet size: ${calculateMaxCarpetSize()}")  
     }  
    
     with(roundTower) {  
         println("\nRound Tower\n==========")  
         println("Material: ${buildingMaterial}")  
         println("Capacity: ${capacity}")  
         println("Floor area: ${floorArea()}")  
         println("Carpet size: ${calculateMaxCarpetSize()}")  
     }  
  }  
    
  /\*\*  
  \* Defines properties common to all dwellings.  
  \* All dwellings have floorspace,  
  \* but its calculation is specific to the subclass.  
  \* Checking and getting a room are implemented here  
  \* because they are the same for all Dwelling subclasses.  
  \*  
  \* @param residents Current number of residents  
  \*/  
  abstract class Dwelling(private var residents: Int) {  
     abstract val buildingMaterial: String  
     abstract val capacity: Int  
    
     /\*\*  
      \* Calculates the floor area of the dwelling.  
      \* Implemented by subclasses where shape is determined.  
      \*  
      \* @return floor area  
      \*/  
     abstract fun floorArea(): Double  
    
     /\*\*  
      \* Checks whether there is room for another resident.  
      \*  
      \* @return true if room available, false otherwise  
      \*/  
     fun hasRoom(): Boolean {  
         return residents < capacity  
     }  
    
     /\*\*  
      \* Compares the capacity to the number of residents and  
      \* if capacity is larger than number of residents,  
      \* add resident by increasing the number of residents.  
      \* Print the result.  
      \*/  
     fun getRoom() {  
         if (capacity > residents) {  
             residents++  
             println("You got a room!")  
         } else {  
             println("Sorry, at capacity and no rooms left.")  
         }  
     }  
    
     }  
    
  /\*\*  
  \* A square cabin dwelling.  
  \*  
  \*  @param residents Current number of residents  
  \*  @param length Length  
  \*/  
  class SquareCabin(residents: Int, val length: Double) : Dwelling(residents) {  
     override val buildingMaterial = "Wood"  
     override val capacity = 6  
    
     /\*\*  
      \* Calculates floor area for a square dwelling.  
      \*  
      \* @return floor area  
      \*/  
     override fun floorArea(): Double {  
         return length \* length  
     }  
    
  }  
    
  /\*\*  
  \* Dwelling with a circular floorspace  
  \*  
  \* @param residents Current number of residents  
  \* @param radius Radius  
  \*/  
  open class RoundHut(  
         val residents: Int, val radius: Double) : Dwelling(residents) {  
    
     override val buildingMaterial = "Straw"  
     override val capacity = 4  
    
     /\*\*  
      \* Calculates floor area for a round dwelling.  
      \*  
      \* @return floor area  
      \*/  
     override fun floorArea(): Double {  
         return PI \* radius \* radius  
     }  
    
     /\*\*  
      \*  Calculates the max length for a square carpet  
      \*  that fits the circular floor.  
      \*  
      \* @return length of carpet  
      \*/  
     fun calculateMaxCarpetSize(): Double {  
         val diameter = 2 \* radius  
         return sqrt(diameter \* diameter / 2)  
     }  
    
  }  
    
  /\*\*  
  \* Round tower with multiple stories.  
  \*  
  \* @param residents Current number of residents  
  \* @param radius Radius  
  \* @param floors Number of stories  
  \*/  
  class RoundTower(  
         residents: Int,  
         radius: Double,  
         val floors: Int = 2) : RoundHut(residents, radius) {  
    
     override val buildingMaterial = "Stone"  
    
     // Capacity depends on the number of floors.  
     override val capacity = floors \* 4  
    
     /\*\*  
      \* Calculates the total floor area for a tower dwelling  
      \* with multiple stories.  
      \*  
      \* @return floor area  
      \*/  
     override fun floorArea(): Double {  
         return super.floorArea() \* floors  
     }  
  }

[8. 요약](https://developer.android.com/codelabs/basic-android-kotlin-training-lists?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-2-pathway-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-lists#7)

Kotlin에서는 Kotlin 표준 라이브러리를 통해 데이터 컬렉션을 더 쉽게 관리하고 조작할 수 있는 기능을 제공합니다. 컬렉션은 동일한 데이터 유형의 여러 객체로 정의할 수 있습니다. Kotlin에는 목록, 집합, 지도와 같은 다양한 기본 컬렉션 유형이 있습니다. 이 Codelab에서는 목록에 특히 중점을 두었고 향후 Codelab에서 집합과 지도를 자세히 알아봅니다.

* 목록은 특정 유형 요소의 정렬된 컬렉션입니다(예: Strings. 목록).
* 색인은 요소의 위치를 나타내는 정수 위치입니다(예: myList[2]).
* 목록에서 첫 번째 요소는 색인 0(예: myList[0])에 있고 마지막 요소는 myList.size-1(예: myList[myList.size-1] 또는 myList.last())에 있습니다.
* 두 가지 목록 유형은 다음과 같습니다. List, MutableList.
* List는 읽기 전용으로, 초기화가 완료되면 수정할 수 없습니다. 그러나 원본을 변경하지 않고 새 목록을 반환하는 sorted() 및 reversed()와 같은 작업을 적용할 수 있습니다.
* MutableList는 요소를 추가하거나 삭제, 수정하는 등 만든 후에 수정할 수 있습니다.
* addAll()을 사용하여 항목 목록을 변경 가능한 목록에 추가할 수 있습니다.
* while 루프를 사용하여 표현식이 false로 평가될 때까지 코드 블록을 실행하고 루프를 종료합니다.

while (expression) {

`// While the expression is true, execute this code block`

}

* for 루프를 사용하여 목록의 모든 항목을 반복합니다.

for (item in myList) {

`// Execute this code block for each element of the list`

}

* vararg 수정자를 사용하면 가변적인 인수 수를 함수나 생성자에 전달할 수 있습니다.

open class Item(val name: String, val price: Int)

class Noodles : Item("Noodles", 10) {

override fun toString(): String {

return name

}

}

class Vegetables(vararg val toppings: String) : Item("Vegetables", 5) {

override fun toString(): String {

if (toppings.isEmpty()) {

return "$name Chef's Choice"

} else {

return name + " " + toppings.joinToString()

}

}

}

class Order(val orderNumber: Int) {

private val itemList = mutableListOf<Item>()

fun addItem(newItem: Item): Order {

itemList.add(newItem)

return this

}

fun addAll(newItems: List<Item>): Order {

itemList.addAll(newItems)

return this

}

fun print() {

println("Order #${orderNumber}")

var total = 0

for (item in itemList) {

println("${item}: $${item.price}")

total += item.price

}

println("Total: $${total}")

}

}

fun main() {

val ordersList = mutableListOf<Order>()

// Add an item to an order

val order1 = Order(1)

order1.addItem(Noodles())

ordersList.add(order1)

// Add multiple items individually

val order2 = Order(2)

order2.addItem(Noodles())

order2.addItem(Vegetables())

ordersList.add(order2)

// Add a list of items at one time

val order3 = Order(3)

val items = listOf(Noodles(), Vegetables("Carrots", "Beans", "Celery"))

order3.addAll(items)

ordersList.add(order3)

// Use builder pattern

val order4 = Order(4)

.addItem(Noodles())

.addItem(Vegetables("Cabbage", "Onion"))

ordersList.add(order4)

// Create and add order directly

ordersList.add(

Order(5)

.addItem(Noodles())

.addItem(Noodles())

.addItem(Vegetables("Spinach"))

)

// Print out each order

for (order in ordersList) {

order.print()

println()

}

}

함수 유형

이러한 유형의 동작을 사용할 수 있도록 Kotlin에서는 *함수 유형*이라는 것이 제공됩니다. 입력 매개변수와 반환 값을 기반으로 특정 유형의 함수를 정의할 수 있습니다. 다음과 같은 형식으로 표시됩니다.

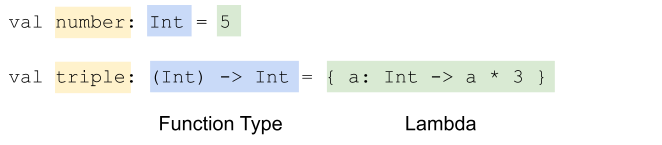
함수 유형의 예: (Int) -> Int

위 유형의 함수는 Int 유형의 매개변수를 사용하고 Int 유형의 값을 반환해야 합니다. 함수 유형 표기에서 괄호 안에 매개변수를 나열합니다(매개변수가 여러 개 있는 경우 쉼표로 구분). 다음에 화살표 ->를 배치한 후 반환 유형을 나열합니다.

이 기준을 충족하는 함수 유형은 무엇일까요? 아래와 같이 정수 입력 값에 3을 곱하는 람다 표현식이 있을 수 있습니다. 람다 표현식 구문의 경우 매개변수가 먼저 나오고(빨간색 상자로 강조표시됨) 함수 화살표가 나온 다음에 함수 본문(보라색 상자로 강조표시됨)이 나옵니다. 람다의 마지막 표현식은 반환 값입니다.



아래 다이어그램과 같이 람다를 변수에 저장할 수도 있습니다. 구문은 Int와 같은 기본 데이터 유형의 변수를 선언하는 방법과 유사합니다. 변수 이름(노란색 상자), 변수 유형(파란색 상자), 변수 값(녹색 상자)을 살펴봅니다. triple 변수는 함수를 저장합니다. 함수 유형이 (Int) -> Int이고 값은 람다 표현식 { a: Int -> a \* 3}입니다.

1. 플레이그라운드에서 다음 코드를 사용해봅니다. 여기에 숫자(예: 5)를 전달하여 triple 함수를 정의하고 호출합니다. 

**참고:** 단일 매개변수가 있는 람다는 일반적이므로 Kotlin은 약식 방법을 제공합니다. Kotlin은 단일 매개변수가 있는 람다의 매개변수에 특수 식별자 [**it**](https://kotlinlang.org/docs/reference/lambdas.html#it-implicit-name-of-a-single-parameter)을 사용합니다.

## 고차 함수

이제 Kotlin의 매우 유연한 함수 조작 방법을 확인하기 시작했으므로 또 다른 강력한 개념인 고차 함수에 관해 알아보겠습니다. 이는 함수(이 경우에는 람다)를 다른 함수로 전달하거나 다른 함수에서 함수를 반환하는 것을 의미합니다.

map, filter, forEach 함수는 모두 매개변수로 함수를 사용했으므로 고차 함수의 예입니다. 이 filter 고차 함수에 전달되는 람다에서는 단일 매개변수와 화살표 기호를 생략하고 it 매개변수도 사용할 수 있습니다.

forEach

peopleAges에 항목을 출력하고 사람의 이름과 나이를 예를 들어 "Fred is 31, Ann is 23,..." 등으로 포함한다고 가정하겠습니다. 이전 Codelab에서 for 루프를 배웠으므로 for (people in peopleAges) { ... }를 사용하여 루프를 작성할 수 있습니다.

하지만 컬렉션의 모든 객체를 열거하는 것은 일반적인 작업이므로, Kotlin에서는 자동으로 모든 항목을 탐색한 후 항목별로 작업을 실행하는 [forEach()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/for-each.html" \t "_blank)를 제공합니다.

1. 플레이그라운드에서 println() 뒤에 다음 코드를 추가합니다.

peopleAges.forEach { print("${it.key} is ${it.value}, ") }

for 루프와 비슷하지만 약간 더 컴팩트합니다. forEach는 현재 항목의 변수를 지정하는 대신 특수 식별자 it을 사용합니다.

forEach() 메서드를 호출할 때 괄호를 추가할 필요가 없었습니다. 중괄호 {} 안에 코드를 전달하기만 하면 됩니다.

1. 프로그램을 실행하고 추가 결과를 살펴봅니다.

Fred is 31, Ann is 23, Barbara is 42, Joe is 51,

원하는 것과 매우 흡사하지만 끝에 쉼표가 추가되어 있습니다.

컬렉션을 문자열로 변환하는 것은 일반적인 작업이며, 끝에 구분 기호가 추가되는 것도 일반적인 문제입니다. 이 문제를 처리하는 방법은 이후 단계에서 다룹니다.

map

[map()](https://kotlinlang.org/docs/reference/collection-transformations.html#mapping) 함수(위에서 설명한 맵 또는 사전 컬렉션과 혼동해서는 안 됨)는 컬렉션의 각 항목에 변환을 적용합니다.

1. 프로그램에서 forEach 문을 다음 줄로 바꿉니다.

println(peopleAges.map { "${it.key} is ${it.value}" }.joinToString(", ") )

1. 프로그램을 실행하고 추가 결과를 살펴봅니다.

Fred is 31, Ann is 23, Barbara is 42, Joe is 51

쉼표가 추가되지 않은, 올바른 출력을 얻었습니다. 한 줄에 다양한 사항이 있으므로 자세히 살펴보겠습니다.

* peopleAges.map은 peopleAges의 각 항목에 변환을 적용하고 변환된 항목으로 이루어진 새 컬렉션을 만듭니다.
* 중괄호 {} 안에 있는 부분은 각 항목에 적용할 변환을 정의합니다. 키-값 쌍을 가져와서 문자열로 변환합니다. 예를 들어 <Fred, 31>을 Fred is 31로 변환됩니다.
* joinToString(", ")은 변환된 컬렉션의 각 항목을 문자열에 추가하고 ,로 구분하며 마지막 항목에는 기호를 추가하지 않습니다.
* 이 모든 과정이 이전 Codelab에서 함수 호출 및 속성 액세스에서 실행한 것처럼 점 연산자(.)로 결합됩니다.

filter

컬렉션의 또 다른 작업은 특정 조건과 일치하는 항목을 찾는 것입니다. [filter()](https://kotlinlang.org/docs/reference/collection-filtering.html) 함수는 컬렉션에서 표현식을 기반으로 일치하는 항목을 반환합니다.

1. println() 뒤에 다음 줄을 추가합니다.

val filteredNames = peopleAges.filter { it.key.length < 4 }  
println(filteredNames)

filter 호출에는 괄호가 필요 없으며, it은 목록의 현재 항목을 나타냅니다.

1. 프로그램을 실행하고 추가 결과를 살펴봅니다.

{Ann=23, Joe=51}

이 경우 표현식은 키(String)의 길이를 가져와서 4글자 미만인지 여부를 확인합니다. 즉, 이름이 4글자 미만인 항목이 일치하여 새 컬렉션에 추가됩니다.

맵에 필터를 적용할 때 반환되는 유형은 새 맵(LinkedHashMap)입니다. 새 맵에서 추가로 처리하거나 새 맵을 목록 같은 다른 유형의 컬렉션으로 변환할 수 있습니다.

sortedWith()

문자열 목록을 정렬하려면 내장된 sorted() 메서드를 컬렉션에 사용하면 됩니다. 그러나 문자열 길이를 기준으로 목록을 정렬하려면 두 문자열의 길이를 가져와 비교하는 코드를 작성해야 합니다. Kotlin에서는 람다를 sortedWith() 메서드에 전달하여 작성할 수 있습니다.

**참고:** 정렬하기 위해 두 객체를 비교할 때 규칙은 첫 번째 객체가 두 번째 객체보다 작으면 0보다 작은 값을 반환하고 첫 번째 객체가 두 번째 객체보다 크면 0보다 큰 값을 반환하는 것입니다.

1. 플레이그라운드에서 이름 목록을 만들고 다음 코드를 사용하여 이름을 기준으로 정렬해 출력합니다.

fun main() {  
    val peopleNames = listOf("Fred", "Ann", "Barbara", "Joe")  
    println(peopleNames.sorted())  
}

1. 이제 람다를 sortedWith() 함수에 전달하여 이름의 길이를 기준으로 정렬된 목록을 출력합니다. 람다는 동일한 유형의 두 매개변수를 사용하고 Int를 반환해야 합니다. main() 함수의 println() 문 뒤에 다음 코드 줄을 추가합니다.

println(peopleNames.sortedWith { str1: String, str2: String -> str1.length - str2.length })

## [6. 요약](https://developer.android.com/codelabs/basic-android-kotlin-training-collections?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-3-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-collections#5)

* 컬렉션은 관련 항목의 그룹입니다. (mapOf, setOf, listOf)
* 컬렉션은 변경 가능하거나 불가능합니다.
* 컬렉션은 순서가 지정되거나 지정되지 않을 수 있습니다.
* 컬렉션에 고유 항목이 필요하거나 중복 항목이 허용될 수 있습니다.
* Kotlin은 목록, 집합, 맵을 포함한 다양한 종류의 컬렉션을 지원합니다.
* Kotlin은 컬렉션을 처리하고 변환하기 위한 forEach, map, filter, sorted 등의 많은 함수를 제공합니다.
* 람다는 이름이 없으며 곧바로 표현식으로 전달할 수 있는 함수입니다. 예를 들면 다음과 같습니다. { a: Int -> a \* 3 }
* 고차 함수는 함수를 다른 함수로 전달하거나 다른 함수에서 함수를 반환하는 것을 의미합니다.

? !!

private var \_binding: FragmentLetterListBinding? = null

null을 허용하므로 \_binding 속성(예: \_binding?.someView)에 액세스할 때마다 null 안전을 위해 ?를 포함해야 합니다. 그러나 null 값 하나 때문에 코드에 물음표를 다수 포함해야 한다는 의미는 아닙니다.

액세스할 때 값이 null이 아님을 확신하는 경우 유형 이름에 !!를 추가할 수 있습니다. 그런 다음 ? 연산자 없이 다른 속성처럼 액세스할 수 있습니다.

**참고:** **!!**를 사용하여 변수를 null을 허용하는 것으로 만들 때 변수가 null이 아님을 아는(**\_binding**이 **onCreateView()**에서 할당된 후 값을 보유하는 것을 아는 것처럼) 위치 한두 군데에서만 사용하는 것이 좋습니다. 이런 식으로 null을 허용하는 값에 액세스하는 것은 위험하며 비정상 종료가 발생할 수 있으므로 최소한으로 사용합니다.

private val binding get() = \_binding!!

### **Kotlin 속성 위임**

Kotlin에는 각 변경 가능한(var) 속성에 자동으로 생성된 기본 getter 함수와 setter 함수가 있습니다. 값을 할당하거나 속성 값을 읽을 때 setter 및 getter 함수가 호출됩니다.

읽기 전용 속성(val)의 경우 변경 가능한 속성과 약간 다릅니다. 기본적으로 getter 함수만 생성됩니다. 읽기 전용 속성의 값을 읽을 때 이 getter 함수가 호출됩니다.

Kotlin에서 속성 위임을 사용하면 getter-setter 책임을 다른 클래스에 넘길 수 있습니다.

이 클래스(대리자 클래스라고 함)는 속성의 getter 및 setter 함수를 제공하고 변경사항을 처리합니다.

대리자 속성은 다음과 같이 by 절 및 대리자 클래스 인스턴스를 사용하여 정의됩니다.

// Syntax for property delegation  
var <property-name> : <property-type> by <delegate-class>()

Array는 List와 비슷하지만 초기화될 때 고정 크기를 가집니다. Array는 크기를 확장하거나 축소할 수 없는(크기를 조절하려면 배열을 복사해야 함) 반면, List에는 add() 함수와 remove() 함수가 있어 크기를 늘리고 줄일 수 있습니다.

.setNegativeButton(getString(R.string.exit)) { \_, \_ ->  
        exitGame()  
    }  
    .setPositiveButton(getString(R.string.play\_again)) { \_, \_ ->  
        restartGame()  
    }

이 구문은 setNegativeButton(getString(R.string.exit), { \_, \_ -> exitGame()})의 약식 표현입니다. 여기서 setNegativeButton() 메서드는 두 매개변수, 즉 String과 함수 DialogInterface.OnClickListener()(람다로 표현 가능)를 사용합니다. 전달되는 마지막 인수가 함수이면 괄호 *바깥에* 람다 표현식을 배치할 수 있습니다. 이를 [*후행 람다 구문*](https://kotlinlang.org/docs/reference/lambdas.html#passing-a-lambda-to-the-last-parameter)이라고 합니다. 람다를 괄호 안에 배치하거나 바깥에 배치하여 코드를 작성하는 방법이 모두 허용됩니다. setPositiveButton 함수의 경우도 마찬가지입니다.

### **apply 범위 함수**

*apply*는 Kotlin 표준 라이브러리의 [범위 함수](https://kotlinlang.org/docs/reference/scope-functions.html)입니다. 이 함수는 객체의 컨텍스트 내에서 코드 블록을 실행하며, 임시 범위를 형성합니다. 그러면 이 범위에서 이름을 사용하지 않고 객체에 액세스할 수 있습니다. apply의 일반적인 사용 사례는 객체를 구성하는 것입니다. 이 함수 호출은 '객체에 다음 할당 적용'으로 읽을 수 있습니다.

**예:**

clark.apply {  
    firstName = "Clark"  
    lastName = "James"  
    age = 18  
}  
  
// The equivalent code without apply scope function would look like the following.  
  
clark.firstName = "Clark"  
clark.lastName = "James"  
clark.age = 18

## [4. Kotlin의 코루틴](https://developer.android.com/codelabs/basic-android-kotlin-training-introduction-coroutines?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-4-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-introduction-coroutines#3)

백그라운드 작업을 위한 스레드를 직접 만들고 사용하는 것은 Android에서 이루어지지만 Kotlin은 동시 실행을 더 유연하고 쉽게 관리할 수 있는 코루틴도 제공합니다.

코루틴의 주요 기능 중 하나는 상태를 저장하여 중단했다가 재개할 수 있다는 것입니다. 코루틴은 실행되거나 실행되지 않을 수 있습니다.

Kotlin의 코루틴 구현은 멀티태스킹을 지원하는 여러 기능을 추가합니다.

|  |  |
| --- | --- |
| Job | 취소 가능한 작업 단위(예: launch() 함수로 만든 작업 단위)입니다. |
| CoroutineScope | launch() 및 async()와 같은 새 코루틴을 만드는 데 사용되는 함수는 CoroutineScope를 확장합니다. |
| Dispatcher | 코루틴이 사용할 스레드를 결정합니다. Main 디스패처는 항상 기본 스레드에서 코루틴을 실행하지만 Default나 IO, Unconfined와 같은 디스패처는 다른 스레드를 사용합니다. |

import kotlinx.coroutines.\*  
  
fun main() {  
    repeat(3) {  
        GlobalScope.launch {  
            println("Hi from ${Thread.currentThread()}")  
        }  
    }  
}

GlobalScope는 앱이 실행되는 한 내부의 코루틴이 실행되도록 허용합니다. 기본 스레드에 관해 언급했던 이유로 인해 예 코드 외부에서는 권장되지 않습니다. 앱에서 코루틴을 사용할 때는 다른 범위가 사용됩니다.

launch() 함수는 취소 가능한 Job 객체에 래핑된 닫힌 코드에서 코루틴을 만듭니다. launch()는 반환 값이 코루틴의 범위 밖에서 필요하지 않을 때 사용됩니다.

fun CoroutineScope.launch {  
    context: CoroutineContext = EmptyCoroutineContext,  
    start: CoroutineStart = CoroutineStart.DEFAULT,  
    block: suspend CoroutineScope.() -> Unit  
}

실제로 개발자가 실행을 위해 전달한 코드 블록은 suspend 키워드로 표시

## runBlocking

새 코루틴을 시작하고 완료될 때까지 현재 스레드를 차단. 주로 기본 함수와 테스트에서 차단 코드와 비차단 코드 사이를 연결하는 데 사용.

Kotlin에는 launch와 유사한 async 함수가 있습니다. async() 함수는 다음과 같이 정의됩니다.

Fun CoroutineScope.async() {  
    context: CoroutineContext = EmptyCoroutineContext,  
    start: CoroutineStart = CoroutineStart.DEFAULT,  
    block: suspend CoroutineScope.() -> T  
}: Deferred<T>

Deferred 유형의 값을 반환합니다. Deferred는 미래 값 참조를 보유할 수 있는 취소 가능한 Job입니다. Deferred를 사용하면 즉시 값을 반환하는 것처럼 함수를 계속 호출할 수 있습니다. 현재 코드 줄이 Deferred의 출력을 기다리도록 하려면 코드 줄에서 await()를 호출하면 됩니다. 그러면 원시 값이 반환됩니다.

import kotlinx.coroutines.\*  
import java.time.LocalDateTime  
import java.time.format.DateTimeFormatter  
  
val formatter = DateTimeFormatter.ISO\_LOCAL\_TIME  
val time = { formatter.format(LocalDateTime.now()) }  
  
suspend fun getValue(): Double {  
    println("entering getValue() at ${time()}")  
    delay(3000)  
    println("leaving getValue() at ${time()}")  
    return Math.random()  
}

fun main() {  
    runBlocking {  
        val num1 = async { getValue() }  
        val num2 = async { getValue() }  
        println("result of num1 + num2 is ${num1.await() + num2.await()}")  
    }  
}

함수가 또 다른 suspend 함수를 호출하면 언제든지 그 함수는 suspend 함수여야 합니다. 함수가 suspend 함수를 호출하지 않으면 그 자체가 suspend 함수가 아니어도 됩니다.

### **객체 선언**

Kotlin에서 [객체 선언](https://kotlinlang.org/docs/reference/object-declarations.html#object-declarations)은 싱글톤 객체를 선언하는 데 사용됩니다. [싱글톤 패턴](https://en.wikipedia.org/wiki/Singleton_pattern)은 객체의 인스턴스가 하나만 생성되도록 보장하며 이 객체의 전역 액세스 포인트 하나를 가집니다. 객체 선언의 초기화는 스레드로부터 안전하며 처음 액세스할 때 실행됩니다. 객체 선언에는 항상 object 키워드 뒤에 이름이 있습니다.

// Object declaration  
object DataProviderManager {  
    fun registerDataProvider(provider: DataProvider) {  
        // ...  
    }  
​  
    val allDataProviders: Collection<DataProvider>  
        get() = // ...  
}  
  
// To refer to the object, use its name directly.  
DataProviderManager.registerDataProvider(...)

### **let 범위 함수**

객체의 컨텍스트 내에서 코드 블록을 실행할 수 있음

let은 호출 체인의 결과에서 함수 하나 이상을 호출하는 데 사용됩니다.

let 함수는 안전 호출 연산자( ?.)와 함께 객체에서 null 안전 연산을 실행하는 데 사용됩니다. 이 경우 let 코드 블록은 객체가 null이 아닌 경우에만 실행됩니다.