

Functions

함수

```
fun 함수(인수1:자료형1, 인수2:자료형2,...):반환자료형 {  
}
```

***인자 생략 가능, 반환값이 없는 경우 Unit 및 생략가능**

```
fun greet(str:String):Unit{  
    println(str)  
}
```

```
greet("Hello World!")
```

```
fun addNumbers(n1: Double, n2: Double): Int {  
    val sum = n1 + n2  
    val sumInteger = sum.toInt()  
    return sumInteger  
}
```

함수

- 단일 표현식과 같은 간단한 값을 반환하는 경우{ } 생략 가능

- return문 생략

```
fun getName(firstName: String, lastName: String): String  
= "$firstName $lastName"
```


- 리턴값의 타입도 생략 가능

```
fun getName(firstName: String, lastName: String) =  
"$firstName $lastName"
```

디폴트 매개변수

- 매개변수의 값을 입력하지 않으면 디폴트 값을 가지게 됨


```
fun main(args: Array<String>) {  
    foo('x', 2)  
}  
  
fun foo(letter: Char = 'a', number: Int = 15) {  
    ... ..  
    ... ..  
}
```



A diagram with two dotted arrows. One arrow starts from the string 'x' in the `foo('x', 2)` call and points to the `letter` parameter in the `foo` function signature. The other arrow starts from the number 2 and points to the `number` parameter in the `foo` function signature.

letter = 'x'	number = 2
--------------	------------

```
fun main(args: Array<String>) {  
    foo('y')  
}  
  
fun foo(letter: Char = 'a', number: Int = 15) {  
    ... ..  
    ... ..  
}
```



A diagram with one dotted arrow starting from the string 'y' in the `foo('y')` call and pointing to the `letter` parameter in the `foo` function signature.

letter = 'y'	number = 15
--------------	-------------

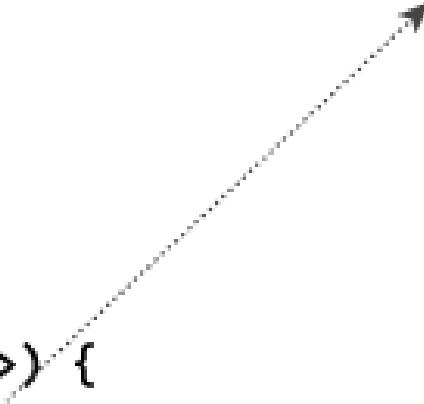
```
fun main(args: Array<String>) {  
    foo()  
}  
  
fun foo(letter: Char = 'a', number: Int = 15) {  
    ... ..  
    ... ..  
}
```

letter = 'a'	number = 15
--------------	-------------

디폴트 매개변수

- 매개변수의 이름을 사용하면 매개변수의 위치에 상관없이 사용 가능함

```
fun displayBorder(character: Char = '=', length: Int = 15) {  
    for (i in 1..length) {  
        print(character)  
    }  
}  
  
fun main(args: Array<String>) {  
    displayBorder(length = 5)  
}
```



가변길이 파라미터

- 가변 길이의 파라미터 `vararg`
 - 함수 내에 여러 개의 `vararg`를 선언 불가

```
fun displayStrings(vararg strings:String){  
    for(string in strings){  
        println(string)                displayStrings("one", "two", "three", "four")  
    }  
}
```

- 단일 파라미터들이 `vararg`보다 먼저 선언되어야 함

```
fun displayStrings(name:String, vararg strings:String){  
    for(string in strings){  
        println(string)  
    }  
}
```

람다 함수 (Lambda)

- Kotlin에서의 함수
 - 반환 자료형 생략 및 블록과 return 생략 가능

```
fun add(x:Int, y:Int): Int{  
    return x+y  
}  
println(add(2,5))
```

```
fun add(x:Int, y:Int) = x+y
```

- 람다함수 : 익명 함수를 간결하게 표현할 수 있는 방법

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

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

```
println(add(10,20))
```

람다 함수 (Lambda)

- 코틀린의 람다식

- 형식 : { 매개변수 -> 함수내용 }
- 람다 함수는 항상 { }로 감싸서 표현
- 인수 목록을 나열하고 -> 이후에 함수 내용이 위치함
- 인자는 ()로 감싸지 않음
- 인자는 형식 추론이 가능하므로 타입 생략 가능
- 함수 반환값은 함수 내용의 마지막 표현식

람다식 (Lambda)

- 코틀린 람다식

- 변수에 람다식을 저장하고, 변수를 일반 함수처럼 사용
 - 변수에 대입하지 않으면 이후 람다 함수를 사용할 수 없음
- 람다 함수 뒤에 ()를 추가하여 함수 호출
 - Run() 함수에 대입해도 바로 함수가 호출되어 실행
- 람다식이 유일한 인자일 경우 () 생략가능 함
- 람다식이 함수의 마지막 매개변수 인 경우 괄호 밖에 기술 (후행람다식)

```
{println("Hello")}()
```

```
run ({println("World")}) ➔ run {println("World")}
```

```
val product = items.fold(1) { acc, e -> acc * e }
```

SAM(Single Abstract Method) 변환

- 추상 메서드 하나를 인수로 사용할 때만 대신 함수 인수 전달

Java

```
changeBtn.setOnClickListener(new View.OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        changeBtn.setBackgroundColor(Color.CYAN);  
    }  
});
```

Kotlin

```
changeBtn.setOnClickListener {  
    view -> changeBtn.setBackgroundColor(Color.CYAN)  
}
```

```
changeBtn.setOnClickListener {  
    _ -> changeBtn.setBackgroundColor(Color.CYAN)  
}
```

인수를 참조하지 않는
경우 _ 또는 생략

```
changeBtn.setOnClickListener{  
    changeBtn.setBackgroundColor(Color.CYAN)  
}
```

```
changeBtn.setOnClickListener {  
    it.visibility = View.INVISIBLE  
}
```

람다식의 인수가 하나인 경우는
it (view) 로 인수 접근

SAM(Single Abstract Method) 변환

- 추상 메서드 하나를 인수로 사용할 때만 함수 인수 전달

Java

```
changeBtn.setOnTouchListener(new View.OnTouchListener() {  
    @Override  
    public boolean onTouch(View v, MotionEvent event) {  
        changeBtn.setBackgroundColor(Color.CYAN);  
        return false;  
    }  
});
```

Kotlin

```
changeBtn.setOnTouchListener{  
    v, event ->  
        changeBtn.setBackgroundColor(Color.CYAN)  
        false  
}
```

Function Type

- 함수 타입의 변수 선언
 - 함수 타입 변수에서는 리턴을 쓰지 않고, 마지막 줄이 반환됨
- 형식

```
val functionType1 : ()->Unit  
val functionType2 : (Int) -> Unit  
val functionType3 : (Int, String) -> String
```

```
functionType1 = { println("greenjoa") }  
functionType2 = { age -> println("나이는 $age") }  
functionType3 = { age, name ->  
    println("나이: $age, 이름: $name")  
    "나이: $age, 이름: $name"  
}
```

```
functionType1()  
functionType2(20)  
println(functionType3(20, "greenjoa"))
```

High-Order Function

- 함수의 인수로 함수나 람다식을 받거나 반환할 수 있는 함수

// 인수

```
fun highOderFunction1(func:()->Unit){  
    func()  
}
```

// 반환

```
fun highOderFunction2():()->Unit{  
    return { println("greenjoa") }  
}
```

// 인수 및 반환

```
fun highOderFunction3(func:()->Unit):()->Unit{  
    return func  
}
```

```
highOderFunction1 { println("hello") }  
highOderFunction2()  
highOderFunction3 { println("world") }()
```

확장 함수 (Extension function)

- 클래스에 새로운 함수를 추가
 - 기존 방식은 상속을 통해 새로운 클래스를 만들고, 함수 추가
 - 확장함수는 클래스 밖에서 정의된 클래스의 멤버 함수
 - 멤버 함수를 오버로딩한 경우 확장 함수가 호출됨
 - 예) String의 처음과 마지막 문자 삭제 함수
→ String 클래스에 존재하지 않는 함수

```
fun String.removeFirstLastChar():String =  
    this.substring(1, this.length - 1)
```

Receiver type

Receiver type

“HelloWorld”.removeFirstLastChar()

클래스

클래스

- 클래스 선언

```
class Person{  
  
}
```

- 객체 선언

- new 키워드는 사용하지 않음

```
val person = Person()
```

- 클래스 생성자

- Primary 생성자

- 매개변수들이 멤버 변수로 자동 추가됨

- Secondary 생성자

- 매개변수들이 멤버 변수로 추가되지 않음
 - 생성자 오버로딩의 개념으로 여러 개의 생성자 정의시 사용함
 - 반드시 Primary 생성자를 호출해야 함에 주의해야 함

클래스

- Primary 생성자
 - 빈 생성자를 생성하며, 코드를 포함할 수 없음
 - 매개변수는 자동으로 멤버 변수로 추가됨

```
class Person(val name:String){  
}
```

<java>

```
public class Person {  
    private final String name;  
    public Person(String name) {  
        this.name = name;  
    }  
}
```

- 초기화 블록 활용한 초기화

```
class Person constructor(val name:String){  
    init{  
        println(name)  
    }  
}
```

클래스

- Secondary 생성자
 - 생성자 오버로딩의 기능
 - 매개 변수들이 멤버 변수로 추가되지 않음
→ 매개변수에 **val / var**를 사용할 수 없음

```
class Person {  
    var name:String?=null  
    constructor(name:String){  
        this.name = name  
    }  
  
    fun printName():Unit{  
        println(name)  
    }  
}
```

```
val person = Person("greenjoa")  
person.printName()
```

클래스

- Secondary 생성자
 - **Primary 생성자가 존재할 경우, 반드시 primary 생성자를 호출해야 함**, 호출 안 할 경우 오류 발생함

```
class Person (val name :String){  
    var addr:String?=null  
    constructor(name:String, addr:String) :this(name){  
        this.addr = addr  
    }  
    fun printName():Unit{  
        println(name)  
    }  
    fun printAddr():Unit{  
        println(addr)  
    }  
}
```

```
val person = Person("greenjoa", "Seoul")  
person.printAddr()
```

클래스

- Secondary 생성자
 - 반드시 **primary** 생성자를 호출해야 함, 호출 안 할 경우 오류 발생함

```
class Person (val name :String){  
    var addr:String?=null  
    var tel:String?=null  
    constructor(name:String, addr:String) :this(name){  
        this.addr = addr  
    }  
    constructor(name:String, addr:String, tel:String):this(name, addr){  
        this.tel = tel  
    }  
}
```

클래스

- Primary 생성자에게 매개변수를 정의
 - 생성자에서 수행할 내용 없으면 {} 생략 가능

```
class Person (val name:String, val addr:String, val tel:String){  
    constructor(name:String, addr:String) :this(name, addr, "")  
    constructor(name:String):this(name, "", "")  
    constructor():this("", "", "")  
    ...  
}
```

```
val person1 = Person("greenjoa", "Seoul", "010-1234-1234")  
val person2 = Person("greenjoa", "Seoul")  
val person3 = Person("greenjoa")  
val person4 = Person()
```

디폴트 매개변수

- 함수의 매개변수에 디폴트 값을 지정

```
class Person (val name:String="noInfo",  
              val addr:String="noInfo",  
              val tel:String="noInfo"){  
  
}
```

```
val person1 = Person("greenjoa", "Seoul", "010-1234-1234")  
val person2 = Person("greenjoa", "Seoul")  
val person3 = Person("greenjoa")  
val person4 = Person()
```

커스텀 접근자

- 프로퍼티(멤버변수)를 접근하기 전에 수행하는 getter/setter 메서드를 자동으로 제공

```
class BankAccount(val accountNumber:Int, var accountBalance:Double){  
    val fees:Double = 25.00  
    var balanceLessFees:Double  
        get(){  
            return accountBalance - fees  
        }  
        set(value){  
            accountBalance += (value - fees)  
        }  
}  
  
val account = BankAccount(10,1000.0)  
account.balanceLessFees = 1000.0  
println(account.balanceLessFees)
```

접근 제한자

- 4개의 접근 제한자 존재함 (함수, 변수, 클래스, 인터페이스)
 - **Public** : 전체 공개, 생략하면 기본이 **public**
 - Private : 현재 파일 내에서만 사용 가능
 - 클래스 : 현재 클래스 or 인터페이스에서만 사용 가능
 - Protected : 해당 파일 내부에서만 사용 가능
 - 클래스 : private과 동일하지만, subclass에서 사용 가능
 - Internal : 같은 모듈 내에서만 사용 가능

```
private fun printName(name:String = "greenjoa"){  
    println(name)  
}
```

test1.kt

```
printName()
```

test2.kt **에러발생**

클래스의 상속

- 코틀린의 모든 기본 클래스는 상속이 불가능함
- 클래스 상속을 하려면 open 키워드를 추가해야 함
 - 단일 상속만 가능

```
open class Animal{  
}  
  
class Dog : Animal(){  
}
```

```
open class Animal(val name:String){  
}  
  
class Dog(name:String) : Animal(name){  
}  
  
class Dog : Animal {  
    constructor(name:String):super(name)  
}
```

추상 클래스

- 인스턴스화 할 수 없는 클래스
 - 추상 메소드는 상속을 통해 오버라이딩해 주어야 함
 - Open 키워드 생략

```
abstract class A{  
    abstract fun func()  
    fun func2(){  
    }  
}  
  
class B : A(){  
    override fun func() {  
        println("Hello")  
    }  
}
```

```
abstract class A{  
    abstract fun func()  
    open fun func2(){  
    }  
}  
  
class B : A(){  
    override fun func() {  
        println("Hello")  
    }  
  
    override fun func2() {  
        super.func2()  
        println("World")  
    }  
}
```

인터페이스

- 자바의 인터페이스와 동일한 기능을 수행함
 - 변수도 선언만 가능하며, 다중상속 가능

```
interface Runnable{  
    var type : Int  
    fun run()  
    fun fastRun() = println("빨리 달린다")  
}
```

```
class RunnableClass : Runnable{  
    override var type: Int = 0  
    override fun run() {  
        println("달린다")  
    }  
}
```

```
val test = RunnableClass()  
test.fastRun()  
test.run()  
test.type = 50  
println(test.type)
```

```
class RunnableClass : Runnable{  
    override var type: Int = 0  
    override fun run() {  
        println("달린다")  
    }  
    override fun fastRun() {  
        println("더 빨리 달린다")  
    }  
}
```

인터페이스

- 인터페이스와 클래스 다중 상속 가능하며, 순서 상관 없음

```
class RunnableClass : Runnable, A(){  
    override fun func() {  
        TODO("not implemented")  
    }  
  
    override var type: Int = 0  
    override fun run() {  
        println("달린다")  
    }  
    override fun fastRun() {  
        println("더 빨리 달린다")  
    }  
}
```

인터페이스 와 클래스 상속

- 인터페이스와 클래스 다중 상속 가능하며, 순서 상관 없음

```
class Dog : Runnable, Eatable, Animal(){
```

```
    override var type: Int = 0
```

```
    override fun run() {
```

```
        println("달린다")
```

```
    }
```

```
    override fun fastRun() {
```

```
        println("더 빨리 달린다")
```

```
    }
```

```
    override fun eat() {
```

```
        println("먹는다")
```

```
    }
```

```
}
```

Companion Object

- 자바에서의 static 변수 및 메소드 기능이 필요한 경우 사용

```
class Person {  
    fun callMe() = println("I'm called.")  
}
```

```
val p1 = Person()  
p1.callMe()
```

```
class Person {  
    companion object {  
        fun callMe() = println("I'm called.")  
    }  
}  
  
Person.callMe()
```

Object 클래스

- Singleton 패턴의 객체 정의
 - 하나의 instance를 가지는 클래스 선언

```
object Test {  
    private var a: Int = 0  
    var b: Int = 1  
  
    fun makeMe12(): Int {  
        a += 12  
        return a  
    }  
}
```

```
val result = Test.makeMe12()  
println("b = ${Test.b}")  
println("result = $result")
```

Object 클래스

- 익명의 객체를 선언시에도 사용
 - 인터페이스를 구현한 객체 생성시 사용

```
val obj = object : MouseAdapter() {  
    override fun mouseClicked(e: MouseEvent) {  
        // ...  
    }  
    override fun mouseEntered(e: MouseEvent) {  
        // ...  
    }  
}
```


Object 클래스

- 클래스의 멤버함수를 오버라이딩한 익명 객체

```
open class Person(name: String, age: Int) {  
    init {  
        println("name: $name, age: $age")  
    }  
    fun eat() = println("Eating food.")  
    fun talk() = println("Talking with people.")  
    open fun pray() = println("Praying god.")  
}  
  
val atheist = object : Person("greenjoa", 23) {  
    override fun pray() = println("I don't pray. I am an atheist.")  
}  
  
atheist.eat()  
atheist.talk()  
atheist.pray()
```

Data 클래스

- 데이터를 저장하는 구조의 클래스
 - equals, hashCode, copy, toString, set, get, componentN 자동생성

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

```
val jack = User("jack", 29)
println("name = ${jack.name}")
println("age = ${jack.age}")
```

```
val u1 = User("John", 29)
val u2 = u1.copy(name = "Randy")
println("u1: name = ${u1.name}, age = ${u1.age}")
println("u2: name = ${u2.name}, age = ${u2.age}")
```

```
val u3 = User("John", 29)
val (name, age) = u3
println("name = $name")
println("age = $age")
```

객체 분리 선언
→ u1.component1()
→ u1.component2()

Sealed 클래스

```
open class Expr
class Const(val value: Double) : Expr()
class Sum(val left: Expr, val right: Expr) : Expr()

fun eval(e: Expr): Double = when (e) {
    is Const -> e.value
    is Sum -> eval(e.right) + eval(e.left)
    else -> throw IllegalArgumentException("Unknown expression")
}
```

*else 일 경우 예외 발생

```
val obj1 :Expr = Const(10.2)
val obj2 :Expr = Sum(Const(10.2),Const(20.3))
println(eval(obj1))
println(eval(obj2))
```

Sealed 클래스

- Subclass의 생성 가능성을 제한
 - When 표현식에서 모든 sealed 클래스의 서브클래스를 처리
 - 같은 파일 내에 선언되어야 함
 - 자동 open 클래스
- ```
val obj1 = Expr.Const(10.2)
val obj2 = Expr.Sum(Expr.Const(10.0), Expr.Const(20.0))
println(eval(obj1))
println(eval(obj2))
```

```
sealed class Expr {
 class Const(val value: Double) : Expr()
 class Sum(val left: Expr, val right: Expr) : Expr()
 object NotANumber : Expr()
}
```

```
fun eval(e: Expr): Double = when (e) {
 is Expr.Const -> e.value
 is Expr.Sum -> eval(e.right) + eval(e.left)
 Expr.NotANumber -> java.lang.Double.NaN
}
```

# Collections

# 컬렉션 - List

- 리스트(List)
  - 같은 자료형의 데이터들을 순서대로 가짐
  - 중복 아이터를 가질 수 있고, 추가, 삭제, 수정이 용이함
- 리스트 생성
  - 읽기 전용 리스트 생성 `listOf()` 메서드 사용

```
val list = ArrayList<String>()
list.add("greenjoa")
```

```
val foods:List<String> = listOf("라면", "갈비", "밥")
```

```
val foods2 = listOf("라면", "갈비", "밥")
```

- 변경 가능한 리스트 생성 `mutableListOf()` 메서드 사용

```
val foods:MutableList<String> = mutableListOf("라면", "갈비", "밥")
```

```
val foods2 = mutableListOf("라면", "갈비", "밥")
```

```
foods.add("초밥")
```

```
foods.removeAt(0)
```

```
foods[1] = "부대찌개"
```

```
foods.set(1, "김치찌개")
```

```
val foods = mutableListOf<String>()
```

# 컬렉션 - Map

- 맵
  - 키와 값의 쌍으로 이루어진 자료구조
    - 키는 중복될 수 없는 자료구조

- 맵 생성

- 읽기 전용 맵

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
val value : Int = map.getValue("a")
println(value)
for((k,v) in map){
 println("$k -> $v")
}
```

```
val map = HashMap<String, String>()
map.put("item1", "greenjoa")
```

- 변경 가능한 맵

```
val citiesMap = mutableMapOf("한국" to "서울", "일본" to "동경", "중국" to "북경")
citiesMap["한국"] = "서울특별시" // 덮어쓰기
citiesMap["미국"] = "워싱턴" // 추가
for((k,v) in citiesMap){
 println("$k -> $v")
}
```

# 컬렉션 - Set

- 집합
  - 중복되지 않는 요소들로 구성된 자료구조
- 집합 생성
  - 읽기 전용 집합

```
val citySet = setOf("서울","수원","부산")
println(citySet.size)
println(citySet.contains("서울"))
```

```
val set = HashSet<String>()
set.add("서울")
```

- 변경 가능한 집합 생성

```
val citySet2 = mutableSetOf("서울","수원","부산")
citySet2.add("안양")
citySet2.add("안양")
citySet2.add("수원")
println(citySet2)
println(citySet2.intersect(citySet))
```



# 예제. 고객 정보 검색

아래의 파일에 있는 정보를 기반으로 고객 정보  
검색 서비스 만들기

<client.txt>

```
gdhong 홍길동 010-1111-2222 80
gdkim 김길동 010-3333-4444 90
gdlee 이길동 010-5555-6666 75
```

**수고하셨습니다.**