

03장 - 함수와 함수형 프로그래밍

Copyright Note

- Book: Do it! 코틀린 프로그래밍 / 이지스퍼블리싱
 - Author: 황영덕 (Youngdeok Hwang; sean.ydhwang@gmail.com)
 - Update Date: 10-July-2019
 - Issue Date: 25-Nov-2017
 - Slide Revision #: rev02
 - Homepage: acaroom.net
 - Distributor: 이지스퍼블리싱 (담당: 박현규)
-
- Copyright© 2019 by acaroom.net All rights reserved.
 - ▬ All slides cannot be modified and copied without permission.
 - ▬ 모든 슬라이드의 내용은 허가없이 변경, 복사될 수 없습니다.

03 함수와 함수형 프로그래밍

03-1 함수 선언하고 호출하기

03-2 함수형 프로그래밍

03-3 고차 함수와 람다식

03-4 고차 함수와 람다식 활용

03-5 코틀린의 다양한 함수들

03-6 함수와 변수의 범위

함수란

❖ 함수의 선언



```
package chap03.section1

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

```
fun 함수 이름([변수 이름: 자료형, 변수 이름: 자료형..]): [반환값의 자료형] {
    표현식...
    [return 반환값]
}
```

함수를 간략하게 표현하기

❖ 일반적인 함수의 모습

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

함수를 간략하게 표현하기

❖ 간략화된 함수

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

함수를 간략하게 표현하기

❖ 반환 자료형 생략

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

코딩해 보세요! main()함수 선언하고 호출하기

❖ sumFunc.kt

```
package chap03.section1

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

fun main(){
    val result1 = sum(3, 2) // IntelliJ IDEA에서 매개변수(a: 3, b: 2)를 보여줌
    val result2 = sum(6, 7) // IntelliJ IDEA에서 매개변수(a: 6, b: 7)를 보여줌

    println(result1)
    println(result2)
}
```


코딩하기! 함수의 호출 원리 살펴보기

❖ MaxFunc.kt

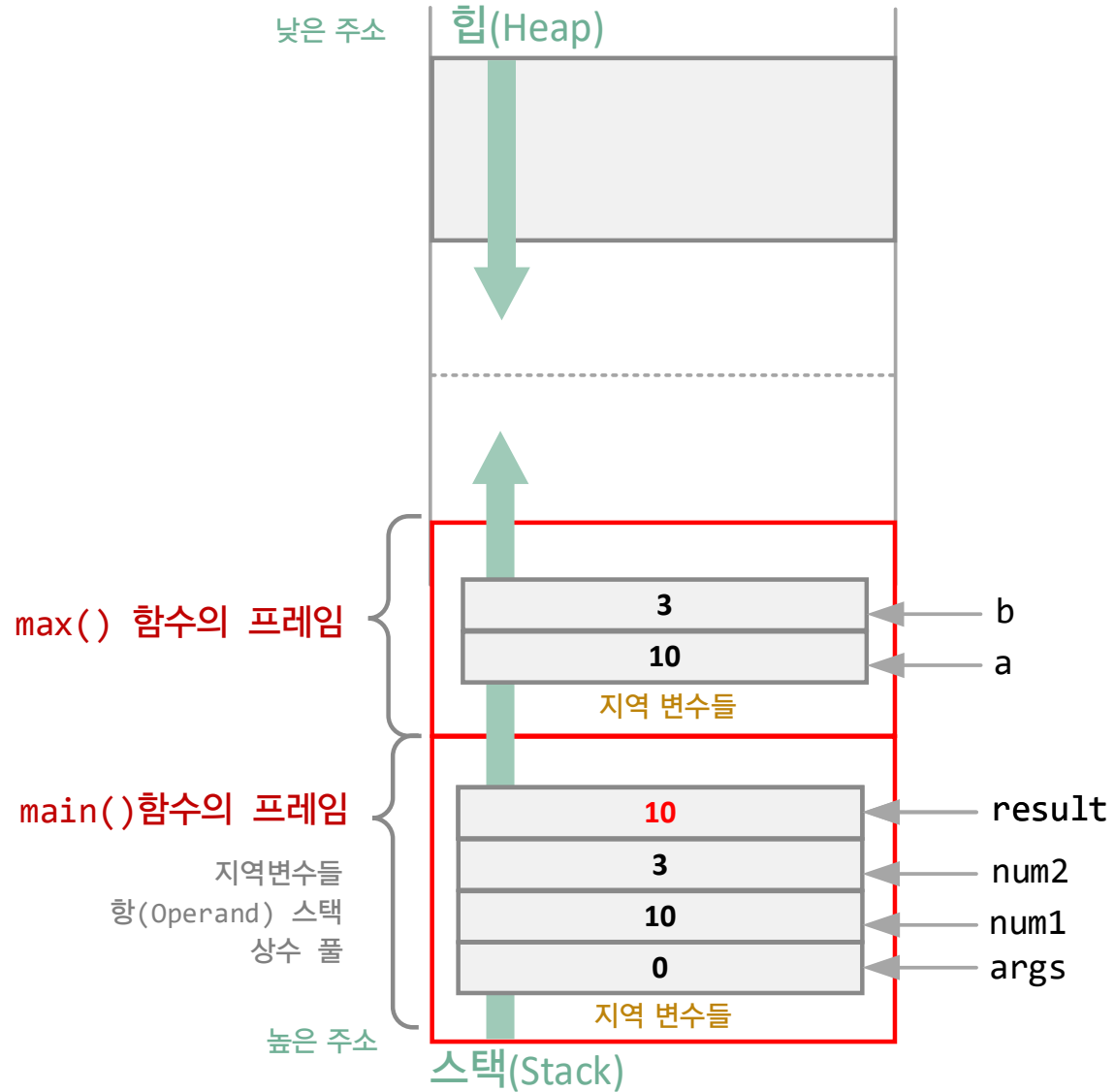
```
package chap03.section1

fun main() { // 최초의 스택 프레임
    val num1 = 10 // 임시 변수 혹은 지역 변수
    val num2 = 3  // 임시 변수 혹은 지역 변수
    val result: Int

    result = max(num1, num2) // 두 번째 스택 프레임
    println(result)
}

fun max(a: Int, b: Int) = if (a > b) a else b // a와 b는 max의 임시 변수
```

함수와 스택 프레임 이해하기



```
fun main() {  
    val num1 = 10  
    val num2 = 3  
    val result: Int  
  
    result = max(num1, num2)  
    println(result)  
}  
  
fun max(a: Int, b: Int) = if (a > b) a else b
```

반환값이 없는 함수

❖ Unit 형

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

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

매개변수의 기본값

❖ 일반적인 함수 사용

```
fun add(name: String, email: String) {  
    // name과 email을 회원 목록에 저장  
}
```

```
add("박현규", "default")  
add("박용규", "default")  
add("함진아", "default")  
...
```

```
fun add(name: String, email: String = "default") {  
    // name과 email을 회원 목록에 저장  
    // email의 기본값은 "default". 즉, email로 넘어오는 값이 없으면 자동으로 "default" 입력  
}  
...  
add("Youngdeok") // email 인자를 생략하여 호출(name에만 "Youngdeok"이 전달됨)
```

코딩해 보세요! 함수의 매개변수에 기본값 지정하기

❖ DefaultParameter.kt

```
package chap03.section1

fun main() {
    val name = "홍길동"
    val email = "hong@example.kr"

    add(name)
    add(name, email)
    add("둘리", "dooly@example.kr")
    defaultArgs()      // ① 100 + 200
    defaultArgs(200)   // ② 200 + 200
}

fun add(name: String, email: String = "default") {
    val output = "${name}님의 이메일은 ${email}입니다."
    println(output)
}

fun defaultArgs(x: Int = 100, y: Int = 200) {
    println(x + y)
}
```

코딩해 보세요! 매개변수 이름과 함께 함수 호출하기

❖ NamedParam.kt

```
package chap03.section2

fun main(args: Array<String>) {

    namedParam(x = 200, z = 100) // x, z의 이름과 함께 함수 호출(y는 기본값 사용)
    namedParam(z = 150) // z의 이름과 함께 함수 호출(x와 y는 기본 값으로 지정됨)

}

fun namedParam(x: Int = 100, y: Int = 200, z: Int) {
    println(x + y + z)
}
```

매개변수의 개수가 고정되지 않은 함수

❖ 가변적인 매개변수 받기

```
fun print3Numbers(num1: Int, num2: Int, num3: Int){  
    // num1, 2, 3을 각각 출력  
}  
fun print4Numbers(num1: Int, num2: Int, num3: Int, num4: Int){  
    // num1, 2, 3, 4를 각각 출력  
}
```

코딩해 보세요! 다양한 인자의 개수를 전달받는 함수

❖ VarargsTest.kt

```
package chap03.section1

fun main(args: Array<String>) {

    normalVarargs(1, 2, 3, 4) // 4개의 인자 구성
    normalVarargs(4, 5, 6)    // 3개의 인자 구성
}

fun normalVarargs(vararg counts: Int) {
    for (num in counts) {
        println("$num")
    }
    print("\n")
}
```


03 함수와 함수형 프로그래밍

03-1 함수 선언하고 호출하기

03-2 함수형 프로그래밍

03-3 고차 함수와 람다식

03-4 고차 함수와 람다식 활용

03-5 코틀린의 다양한 함수들

03-6 함수와 변수의 범위

함수형 프로그래밍

❖ 코틀린은 다중 패러다임 언어

- 함수형 프로그래밍(FP: Functional Programming)
- 객체 지향 프로그래밍(OOP: Object-Oriented Programming)

❖ 함수형 프로그래밍

- 코드 간략, 테스트나 재사용성 증가
- 람다식, 고차 함수를 사용해 구성
- 순수 함수

순수 함수

❖ 순수 함수(pure function) 이해

- 부작용(side-effect)이 없는 함수
 - 동일한 입력 인자에 대해서는 항상 같은 결과를 출력 혹은 반환 한다.
 - 값이 예측이 가능해 결정적(deterministic)이다.

// 순수 함수의 예

```
fun sum(a: Int, b: Int): Int {  
    return a + b // 동일한 인자인 a, b를 입력 받아 항상 a + b를 출력(부작용이 없음)  
}
```

- 순수 함수의 조건
 - 같은 인자에 대하여 항상 같은 값을 반환
 - 함수 외부의 어떤 상태도 바꾸지 않는다.

순수 함수

❖ 예) 그렇다면 순수 함수가 아닌 것은?

```
fun check() {  
    val test = User.grade() // check() 함수에 없는 외부의 User 객체를 사용  
    if (test != null) process(test) // 변수 test는 User.grade()의 실행 결과에 따라 달라짐  
}
```

```
const val global = 10  
  
fun main() {  
    val num1 = 10  
    val num2 = 3  
    val result = noPureFunction(num1, num2)  
    println(result)  
}  
  
fun noPureFunction(a: Int, b: Int): Int {  
    return a + b + global // 입력값과 무관하게 외부의 변수 사용  
}
```

순수 함수

❖ 순수 함수를 왜?

- 입력과 내용을 분리하고 모듈화 하므로 재사용성이 높아진다.
 - ▬ 여러가지 함수들과 조합해도 부작용이 없다.
- 특정 상태에 영향을 주지 않으므로 병행 작업 시 안전하다.
- 함수의 값을 추적하고 예측 할 수 있기때문에 테스트, 디버깅 등이 유리하다.

❖ 함수형 프로그래밍에 적용

- 함수를 매개변수, 인자에 혹은 반환값에 적용 (고차 함수)
- 함수를 변수나 데이터 구조에 저장
- 유연성 증가

람다식

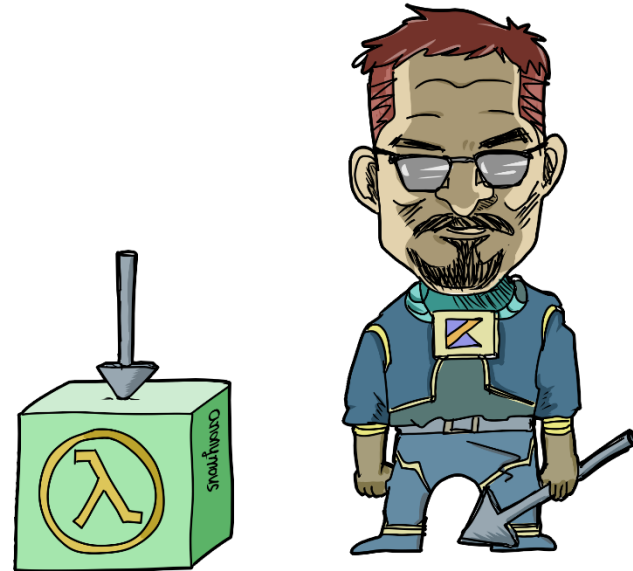
❖ 람다식(Lambda Expression)이란?

- 익명 함수의 하나의 형태로 이름 없이 사용 및 실행이 가능
- 람다 대수(Lambda calculus)로 부터 유래

```
{ x, y -> x + y } // 람다식의 예 (이름이 없는 함수 형태)
```

❖ 람다식의 이용

- 람다식은 고차 함수에서 인자로 넘기거나 결과값으로 반환 등을 할 수 있다.



일급 객체

❖ 일급 객체(First Class Citizen)란?

- 일급 객체는 함수의 인자로 전달할 수 있다.
- 일급 객체는 함수의 반환값에 사용할 수 있다.
- 일급 객체는 변수에 담을 수 있다.

❖ 코틀린에서 함수는 1급 객체로 다룸

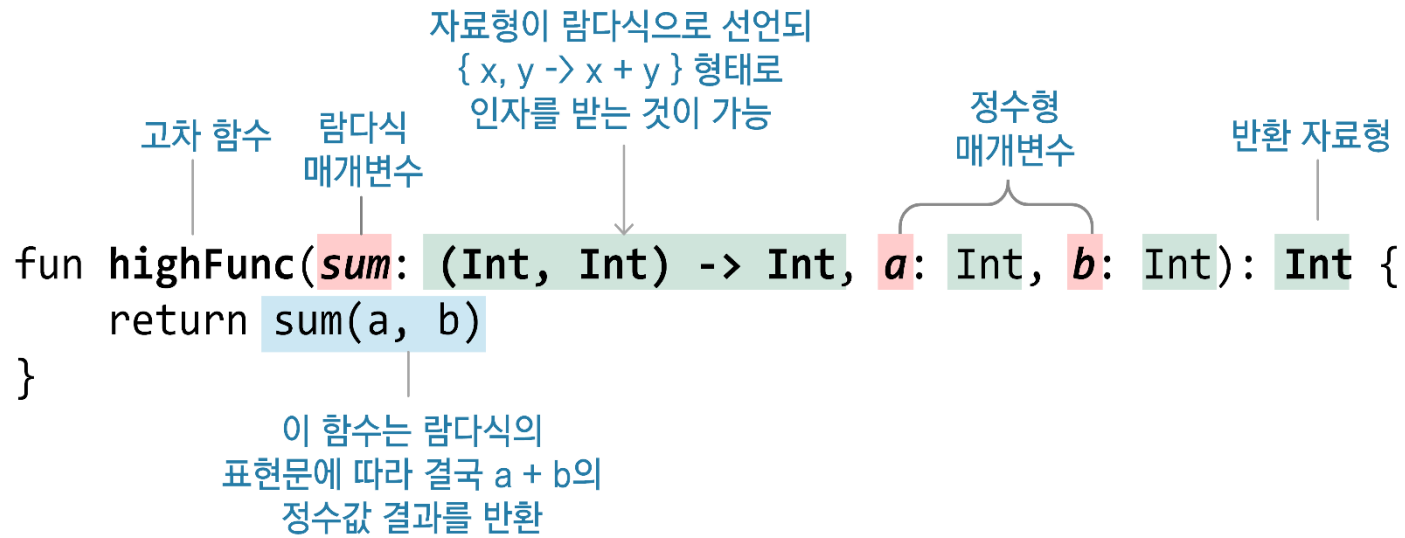
- 1급 함수라고도 한다.

고차 함수의 이해

❖ 고차 함수(high-order function)란?

```
fun main() {  
    println(highFunc({ x, y -> x + y }, 10, 20)) // 람다식 함수를 인자로 넘김  
}
```

```
fun highFunc(sum: (Int, Int) -> Int, a: Int, b: Int): Int = sum(a, b) // sum 매개변수는 함수
```



함수형 프로그래밍 왜?

❖ 왜 사용해야 하나요?

- 프로그램을 모듈화 해 디버깅하거나 테스트가 쉬움
- 간략한 표현식을 사용해 생산성이 높음
- 람다식과 고차함수를 사용하면서 다양한 함수 조합을 사용할 수 있음

❖ 정리

- 함수형 프로그래밍은 순수 함수를 조합해 상태 데이터 변경이나 부작용이 없는 루틴을 만들어 내며 이름 없는 함수 형태의 하나인 람다식을 이용해 함수를 변수처럼 매개변수, 인자, 반환값 등에 활용하는 고차 함수를 구성해 생산성을 높인 프로그래밍 방법

03 함수와 함수형 프로그래밍

03-1 함수 선언하고 호출하기

03-2 함수형 프로그래밍

03-3 고차 함수와 람다식

03-4 고차 함수와 람다식 활용

03-5 코틀린의 다양한 함수들

03-6 함수와 변수의 범위

고차 함수의 형태

- ❖ 일반 함수를 인자나 반환값으로 사용하는 고차 함수
 - 코딩해 보세요! 인자에 일반 함수 사용해 보기
 - FuncArgument.kt

```
package chap03.section3.funcargs

fun main() {

    val res1 = sum(3, 2) // 일반 인자
    val res2 = mul(sum(3,3), 3) // 인자에 함수를 사용

    println("res1: $res1, res2: $res2")
}

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

fun mul(a: Int, b: Int) = a * b
```

코딩해 보세요! 반환값에 일반 함수 사용해 보기

❖ FuncFunc.kt

```
package chap03.section3.funcfunc

fun main() {

    println("funcFunc: ${funcFunc()}")
}

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

fun funcFunc(): Int { // 함수의 반환값으로 함수 사용
    return sum(2, 2)
}
```

람다식의 구성

❖ 코딩해 보세요! 변수에 할당하는 람다식 함수 작성하기

■ HighOrderTest.kt

```
package chap03.section3

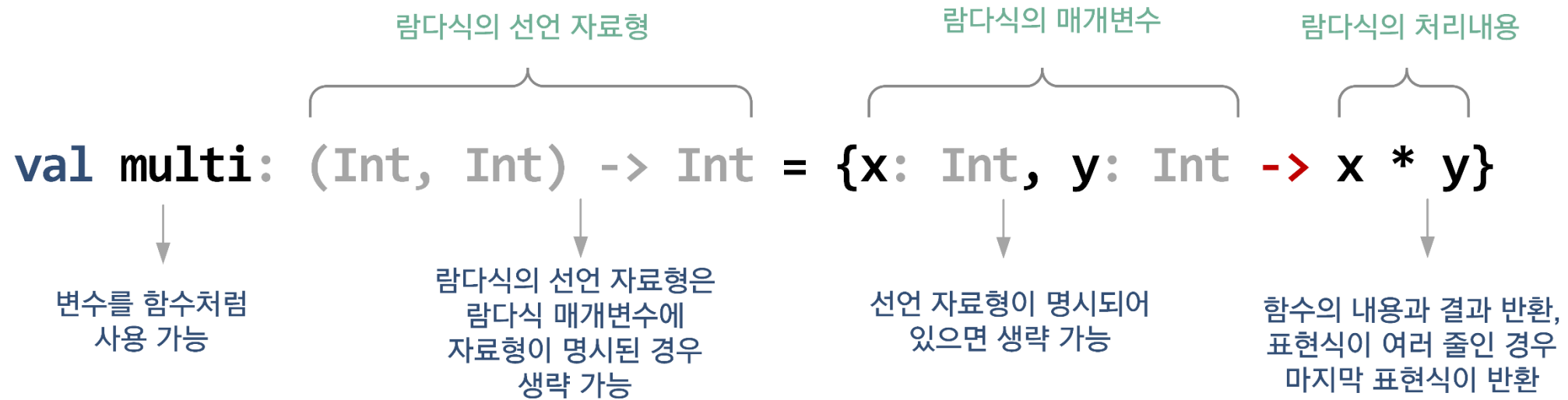
fun main() {

    var result: Int

    // 일반 변수에 람다식 할당
    val multi = {x: Int, y: Int -> x * y}
    // 람다식이 할당된 변수는 함수처럼 사용 가능
    result = multi(10, 20)
    println(result)
}
```

람다식의 구성

❖ 변수에 지정된 람다식



람다식의 구성

❖ 표현식이 2줄 이상일 때

```
val multi2: (Int, Int) -> Int = { x: Int, y: Int ->
    println("x * y")
    x * y // 마지막 표현식이 반환됨
}
```

❖ 자료형의 생략

```
val multi: (Int, Int) -> Int = {x: Int, y: Int -> x * y} // 생략되지 않은 전체 표현
val multi = {x: Int, y: Int -> x * y} // 선언 자료형 생략
val multi: (Int, Int) -> Int = {x, y -> x * y} // 람다식 매개변수 자료형의 생략
val multi = {x, y -> x * y} // 에러! 추론이 가능하지 않음
```

람다식의 구성

❖ 반환 자료형이 없거나 매개변수가 하나 있을 때

```
val greet: ()->Unit = { println("Hello World!") }  
val square: (Int)->Int = { x -> x * x }
```

❖ 람다식 안에 람다식이 있는 경우?

```
val nestedLambda: ()->()->Unit = { { println("nested") } }
```

❖ 선언부의 자료형 생략

```
val greet = { println("Hello World!") } // 추론 가능  
val square = { x: Int -> x * x } // 선언 부분을 생략하려면 x의 자료형을 명시해야 함  
val nestedLambda = { { println("nested") } } // 추론 가능
```


코딩해 보세요! 매개변수에 람다식 함수를 이용한 고차 함수

❖ HighOrderTest2.kt

```
package chap03.section3

fun main() {

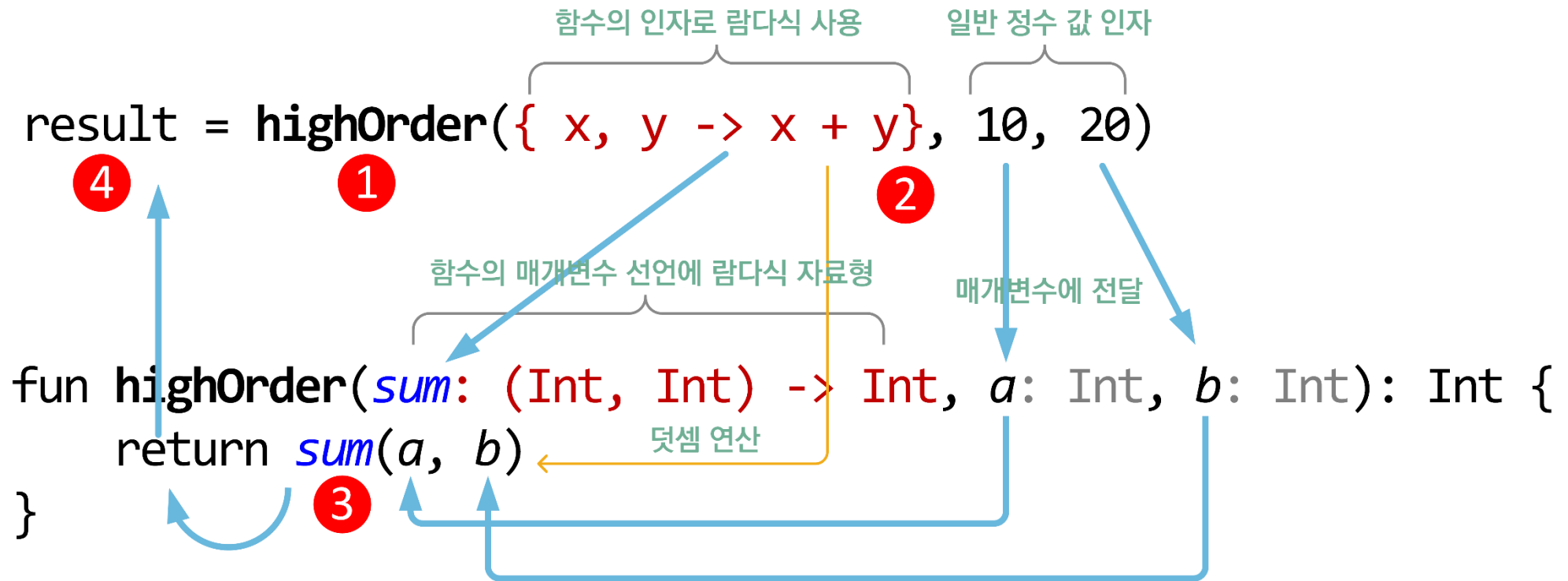
    var result: Int

    // 람다식을 매개변수와 인자로 사용한 함수
    result = highOrder({ x, y -> x + y }, 10, 20)
    println(result)
}

fun highOrder(sum: (Int, Int) -> Int, a: Int, b: Int): Int {
    return sum(a, b)
}
```

코딩해 보세요! 매개변수에 람다식 함수를 이용한 고차 함수

❖ 호출 동작



코딩해 보세요! 인자와 반환값이 없는 람다식 함수

❖ HighOrderTest3.kt

```
package chap03.section3

fun main() {

    // 반환값이 없는 람다식의 선언
    val out: () -> Unit = { println("Hello World!") }
    // 추론이 가능하므로 val out = { println("Hello World!") }와 같이 생략 가능
    // 람다식이 들어있는 변수를 다른 변수에 할당

    out() // 함수처럼 사용가능
    val new = out // 변수처럼 할당해 재사용 가능
    new()

}
```

람다식과 고차함수 호출하기

❖ 값에 의한 호출

- 함수가 인자로 전달될 경우
 - ▬ 람다식 함수는 값으로 처리되어 그 즉시 함수가 수행된 후 값을 전달

코딩해 보세요! 값에 의한 호출로 람다식 사용하기

❖ CallByValue.kt

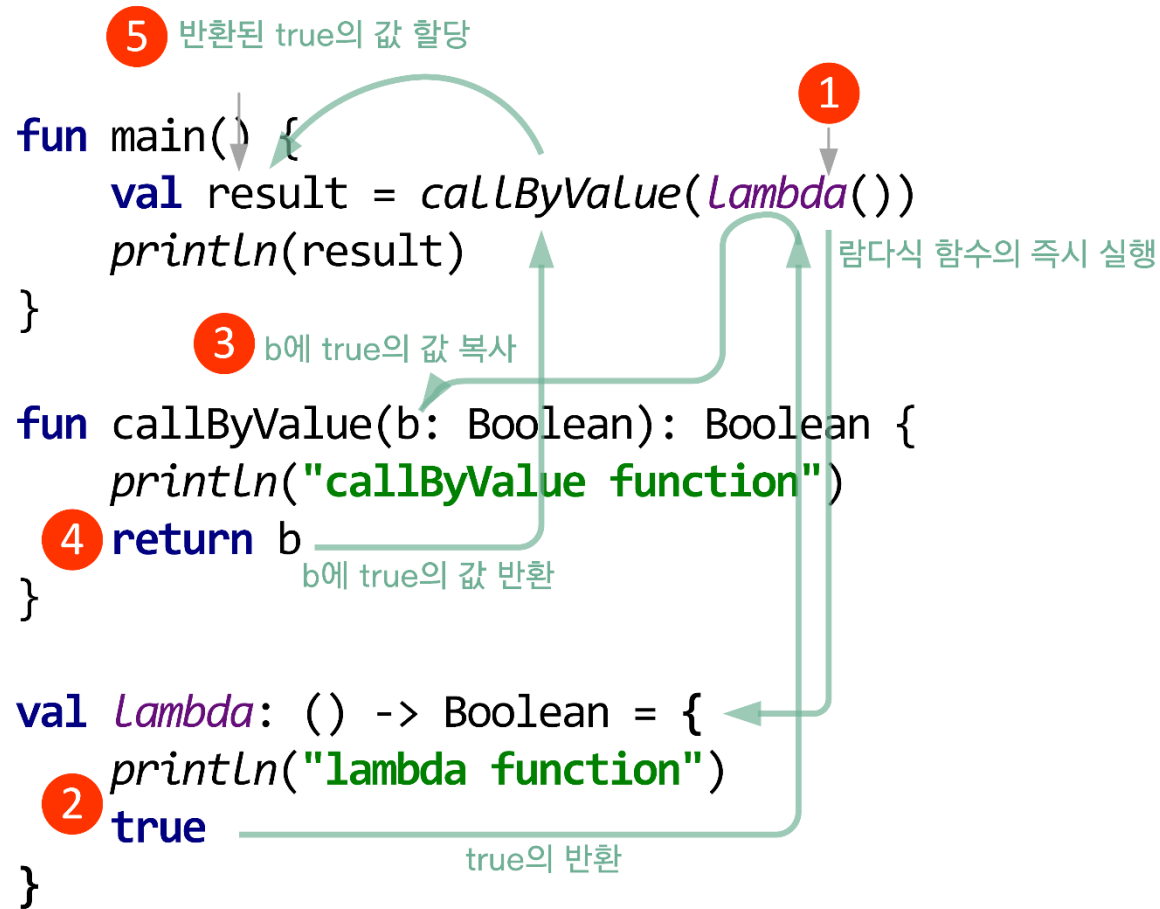
```
package chap03.section3

fun main() {
    val result = callByValue(lambda()) // 람다식 함수를 호출
    println(result)
}

fun callByValue(b: Boolean): Boolean { // 일반 변수 자료형으로 선언된 매개변수
    println("callByValue function")
    return b
}

val lambda: () -> Boolean = { // 람다 표현식이 두 줄이다
    println("lambda function")
    true // 마지막 표현식 문장의 결과가 반환
}
```

코딩해 보세요! 값에 의한 호출로 람다식 사용하기



코딩해 보세요! 람다식 이름을 사용한 호출 이용하기

❖ CallByName.kt

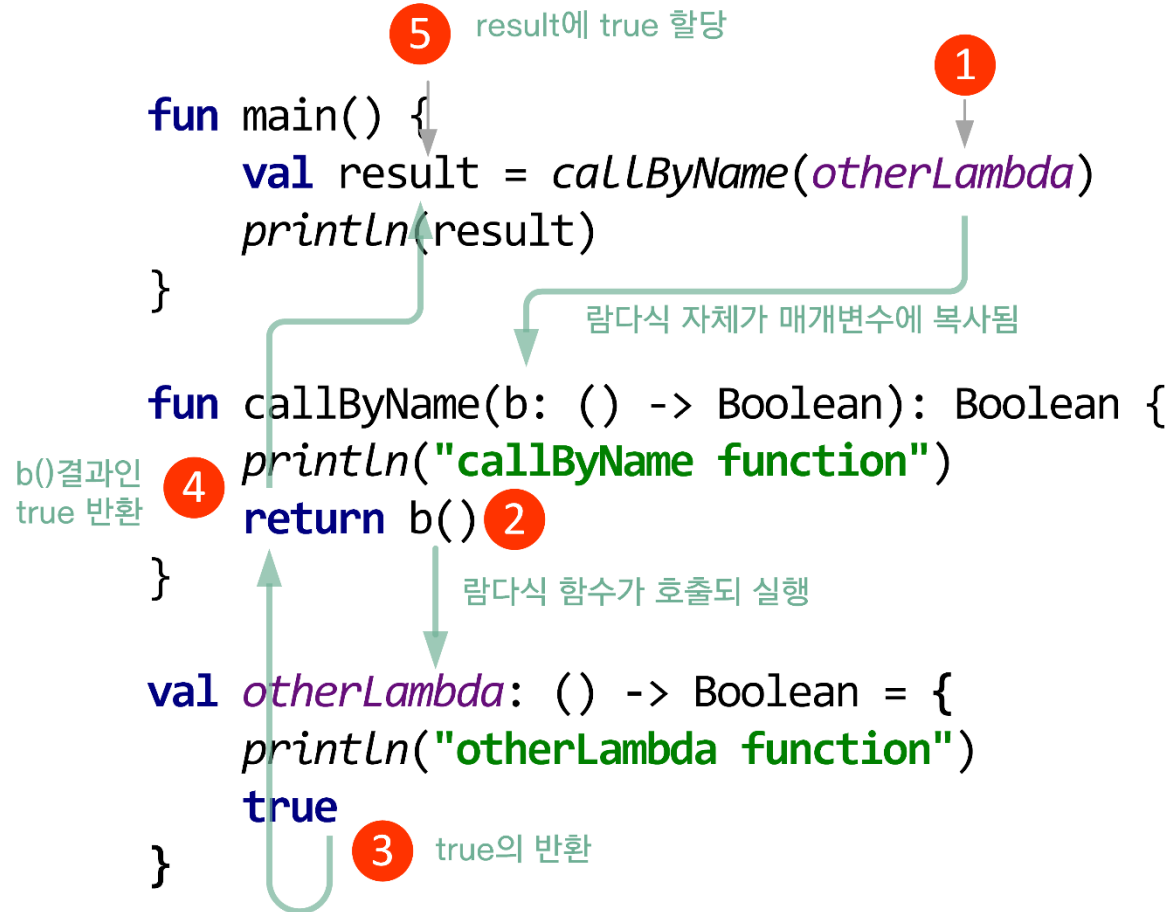
```
package chap03.section3

fun main() {
    val result = callByName(otherLambda) // 람다식 이름으로 호출
    println(result)
}

fun callByName(b: () -> Boolean): Boolean { // 람다식 함수 자료형으로 선언된 매개변수
    println("callByName function")
    return b()
}

val otherLambda: () -> Boolean = {
    println("otherLambda function")
    true
}
```

코딩해 보세요! 람다식 이름을 사용한 호출 이용하기



다른 함수의 참조에 의한 호출

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

```
funcParam(3, 2, sum) // 오류! sum은 람다식이 아님  
...  
fun funcParam(a: Int, b: Int, c: (Int, Int) -> Int): Int {  
    return c(a, b)  
}
```

```
funcParam(3, 2, ::sum)
```

코딩해 보세요! 참조에 의한 호출 방식으로 함수 호출하기

```
fun main() {  
    // 1. 인자와 반환값이 있는 함수  
    val res1 = funcParam(3, 2, ::sum)  
    println(res1)  
  
    // 2. 인자가 없는 함수  
    hello(::text) // 반환값이 없음  
  
    // 3. 일반 변수에 값처럼 할당  
    val likeLambda = ::sum  
    println(likeLambda(6,6))  
}
```

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

```
fun text(a: String, b: String) = "Hi! $a $b"
```

```
fun funcParam(a: Int, b: Int, c: (Int, Int) -> Int): Int {  
    return c(a, b)  
}
```

```
fun hello(body: (String, String) -> String): Unit {  
    println(body("Hello", "World"))  
}
```

FunctionReference.kt

람다식 함수의 매개변수

❖ 매개변수 개수에 따라 람다식을 구성하는 방법

- 매개변수가 없는 경우

```
package chap03.section3
```

ParamCount.kt

```
fun main() {  
    // 매개변수 없는 람다식 함수  
    noParam({ "Hello World!" })  
    noParam { "Hello World!" } // 위와 동일 결과, 소괄호 생략 가능  
}
```

```
// 매개변수가 없는 람다식 함수가 noParam 함수의 매개변수 out으로 지정됨  
fun noParam(out: () -> String) = println(out())
```

람다식 함수의 매개변수

❖ 매개변수 개수에 따라 람다식을 구성하는 방법

- 매개변수가 한 개인 경우

```
...
fun main() {
    // 매개변수 없는 람다식 함수
    ...
    // 매개변수가 하나 있는 람다식 함수
    oneParam({ a -> "Hello World! $a" })
    oneParam { a -> "Hello World! $a" } // 위와 동일 결과, 소괄호 생략 가능
    oneParam { "Hello World! $it" } // 위와 동일 결과, it으로 대체 가능
}
...
// 매개변수가 하나 있는 람다식 함수가 oneParam함수의 매개변수 out으로 지정됨
fun oneParam(out: (String) -> String) {
    println(out("OneParam"))
}
```

람다식 함수의 매개변수

❖ 매개변수 개수에 따라 람다식을 구성하는 방법

- 매개변수가 두 개 이상인 경우

```
...
fun main() {
    ...
    // 매개변수가 두 개 있는 람다식 함수
    moreParam { a, b -> "Hello World! $a $b"} // 매개변수명 생략 불가
    ...
}
// 매개변수가 두 개 있는 람다식 함수가 moreParam 함수의 매개변수로 지정됨
fun moreParam(out: (String, String) -> String) {
    println(out("OneParam", "TwoParam"))
}
```

람다식 함수의 매개변수

❖ 매개변수 개수에 따라 람다식을 구성하는 방법

- 매개변수를 생략하는 경우

```
moreParam { _, b -> "Hello World! $b"} // 첫 번째 문자열은 사용하지 않고 생략
```

람다식 함수의 매개변수

❖ 일반 매개변수와 람다식 매개변수를 같이 사용

```
...
fun main() {
    ...
    // 인자와 함께 사용하는 경우
    withArgs("Arg1", "Arg2", { a, b -> "Hello World! $a $b" }) // ①
    // withArgs()의 마지막 인자가 람다식인 경우 소괄호 바깥으로 분리 가능
    withArgs("Arg1", "Arg2") { a, b -> "Hello World! $a $b" } // ②
}
...
// withArgs함수는 일반 매개변수 2개를 포함, 람다식 함수를 마지막 매개변수로 가짐
fun withArgs(a: String, b: String, out: (String, String) -> String) {
    println(out(a, b))
}
```

두 개의 랴다식을 가진 함수의 사용

❖ TwoLambdaParam.kt

```
package chap03.section3

fun main() {
    twoLambda({ a, b -> "First $a $b" }, {"Second $it"})
    twoLambda({ a, b -> "First $a $b" }) {"Second $it"} // 위와 동일
}

fun twoLambda(first: (String, String) -> String, second: (String) -> String) {
    println(first("OneParam", "TwoParam"))
    println(second("OneParam"))
}
```

```
({첫 번째}, {두 번째})
({첫 번째}) {두 번째}
```

```
({첫 번째}, {두 번째}) {세 번째}
```


03 함수와 함수형 프로그래밍

03-1 함수 선언하고 호출하기

03-2 함수형 프로그래밍

03-3 고차 함수와 람다식

03-4 고차 함수와 람다식 활용

03-5 코틀린의 다양한 함수들

03-6 함수와 변수의 범위

동기화를 위한 코드 구현하기

❖ 동기화?

- 변경이 일어나면 안 되는 특정 코드를 보호하기 위한 잠금 기법
- 동기화로 보호되는 코드는 임계 영역(Critical Section)
- Lock을 활용해 임계 영역을 보호

❖ 자바의 Lock과 ReentrantLock

```
Lock lock = new ReentrantLock();
lock.lock(); // 잠금
try {
    // 보호할 임계 영역의 코드
    // 수행할 작업
} finally {
    lock.unlock(); // 해제
}
```

동기화를 위한 코드 구현하기

❖ ReentrantLock를 활용해 특정 함수 보호

- 고차함수를 이용해 구현

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

코딩해 보세요! 공유자원을 접근하는 코드를 보호하기

❖ LockHighOrder.kt

```
...
var sharable = 1 // 보호가 필요한 공유 자원

fun main() {
    val reLock = ReentrantLock()

    // ①, ②, ③ 표현식이 모두 동일
    lock(reLock, { criticalFunc() }) // ①
    lock(reLock) { criticalFunc() } // ②
    lock(reLock, ::criticalFunc) // ③

    println(sharable)
}

fun criticalFunc() {
    // 공유 자원 접근 코드 사용
    sharable += 1
}
...
```

네트워크 호출 구현하기

❖ 네트워크 처리 결과에 따른 콜백 함수 구현

■ 기존의 자바 코드

```
// 성공과 실패 콜백 함수를 위한 인터페이스 선언
public interface Callback {
    void onSuccess(ResultType result);
    void onError(Exception exception);
}

// networkCall의 선언
public void networkCall (Callback callback) {
    try{
        // 성공 시 onSuccess 콜백 함수 호출
        callback.onSuccess(myResult);
    }catch (e: Throwable) {
        // 실패 시 onError 콜백 함수 호출
        callback.onError(e);
    }
}

...

// networkCall의 사용 - 인자에서 인터페이스 구현을 익명 객체를 만들어 처리
networkCall(new Callback() {
    public void onSuccess(ResultType result) {
        // 네트워크 호출에 성공했을 때의 구현부
    }
    public void onError(Exception e){
        // 네트워크 호출에 실패했을 때의 구현부
    }
})
```



콜백(callback) 함수란 특정 이벤트가 발생하기까지 처리되지 않다가 이벤트가 발생하면 즉시 호출되어 처리되는 함수

네트워크 호출 구현하기

❖ 네트워크 처리 결과에 따른 콜백 함수 구현

■ 코틀린 코드

// 코틀린으로 만들어진 네트워크 호출 구현부

// ① 람다식 함수의 매개변수를 가진 networkCall()함수 선언

```
fun networkCall(onSuccess: (ResultType) -> Unit, onError: (Throwable) -> Unit) {  
    try {  
        onSuccess(myResult)  
    } catch (e: Throwable) {  
        onError(e)  
    }  
}
```

...

// ② networkCall()함수의 사용 - 인자 형식에 람다식을 사용

```
networkCall(result -> {  
    // 네트워크 호출에 성공했을 때 구현부  
}, error -> {  
    // 네트워크 호출에 실패했을 때 구현부  
});
```

03 함수와 함수형 프로그래밍

03-1 함수 선언하고 호출하기

03-2 함수형 프로그래밍

03-3 고차 함수와 람다식

03-4 고차 함수와 람다식 활용

03-5 코틀린의 다양한 함수들

03-6 함수와 변수의 범위

익명 함수

❖ 익명 함수(anonymous functions)

- 함수가 이름이 없는 것

```
fun (x: Int, y: Int): Int = x + y // 함수 이름이 생략된 익명 함수
```

```
val add: (Int, Int) -> Int = fun(x, y) = x + y // 익명 함수를 사용한 add 선언  
val result = add(10, 2) // add의 사용
```

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

```
val add = {x: Int, y: Int -> x + y } // 람다식과 매우 흡사
```



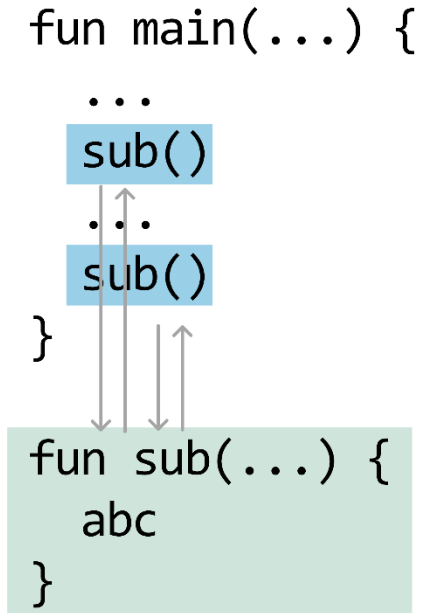
일반 익명 함수에서는 return, break, continue가 사용 가능하지만 람다식에서는 사용하기 어렵다. (라벨 표기법과 같이 사용해야 함)

인라인 함수

❖ 인라인(inline) 함수

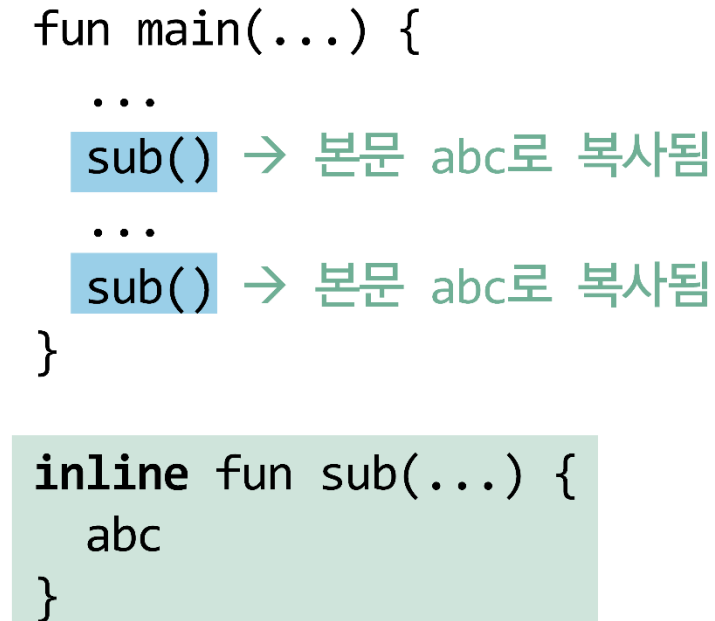
- 함수가 호출되는 곳에 내용을 모두 복사
- 함수의 분기 없이 처리 → 성능 증가

```
fun main(...) {  
    ...  
    sub()  
    ...  
    sub()  
}  
  
fun sub(...) {  
    abc  
}
```



The diagram illustrates a standard function call. In the `main` function, there are two calls to `sub()`. Arrows point from each `sub()` call down to the definition of `fun sub(...)` at the bottom, which contains the code `abc`. This represents a single copy of the function code being shared across multiple calls.

```
fun main(...) {  
    ...  
    sub() → 본문 abc로 복사됨  
    ...  
    sub() → 본문 abc로 복사됨  
}  
  
inline fun sub(...) {  
    abc  
}
```



The diagram illustrates an inlined function. In the `main` function, the two `sub()` calls are now followed by the text `→ 본문 abc로 복사됨` (copied from the body text `abc`). Below, the function definition is marked with the `inline` keyword: `inline fun sub(...)`. This shows that the function's body is copied directly into the call site, eliminating the need for a separate function definition and reducing branching.

코딩해 보세요! 인라인 함수 작성해 보기

❖ InlineFunction.kt

```
package chap03.section5

fun main() {
    // 인라인 함수 shortFunc의 내용이 복사되어 들어감
    shortFunc(3) { println("First call: $it") }
    shortFunc(5) { println("Second call: $it") }
}

inline fun shortFunc(a: Int, out: (Int) -> Unit) {
    println("Before calling out()")
    out(a)
    println("After calling out()")
}
```

인라인 함수의 제한과 인라인 금지

❖ 인라인 함수의 단점

- 코드가 복사되므로 내용이 많은 함수에 사용하면 코드가 늘어남

❖ `noinline` 키워드

- 일부 람다식 함수를 인라인 되지 않게 하기 위해

```
inline fun sub(out1: () -> Unit, noinline out2: () -> Unit) {
```

코딩해 보세요! `noinline`으로 람다식 함수의 인라인 막기

❖ NoinlineTest.kt

```
package chap03.section5.noinline

fun main() {
    shortFunc(3) { println("First call: $it") }
}

inline fun shortFunc(a: Int, noinline out: (Int) -> Unit) {
    println("Before calling out()")
    out(a)
    println("After calling out()")
}
```

인라인 함수와 비지역 반환

❖ 코딩해 보세요! return으로 빠져나오기 - LocalReturn.kt

- 비지역 반환(non-local return)

```
package chap03.section5.localreturn

fun main() {
    shortFunc(3) {
        println("First call: $it")
        return // ① 의도하지 않은 반환
    }
}

inline fun shortFunc(a: Int, out: (Int) -> Unit) {
    println("Before calling out()")
    out(a)
    println("After calling out()") // ② 이 문장은 실행되지 않음
}
```

비지역 반환의 금지

❖ 코딩해 보세요! crossinline으로 return 금지 - LocalReturnCrossinline.kt

```
package chap03.section5.crossinline

fun main() {
    shortFunc(3) {
        println("First call: $it")
        // return 사용 불가
    }
}

inline fun shortFunc(a: Int, crossinline out: (Int) -> Unit) {
    println("Before calling out()")
    nestedFunc { out(a) }
    println("After calling out()")
}

fun nestedFunc(body: () -> Unit) {
    body()
}
```

확장 함수

❖ 확장 함수(extension function)

- 클래스의 멤버 함수를 외부에서 더 추가할 수 있다.

```
fun 확장대상.함수명(매개변수, ...): 반환값 {  
    ...  
    return 값  
}
```

코딩해 보세요! String에 확장 함수 추가하기

❖ ExtensionFunction.kt

```
package chap03.section5

fun main() {
    val source = "Hello World!"
    val target = "Kotlin"
    println(source.getLongString(target))
}

// String을 확장해 getLongString 추가
fun String.getLongString(target: String): String =
    if (this.length > target.length) this else target
```

- this는 확장 대상에 있던 자리의 문자열인 source 객체를 나타냄
- 기존의 표준 라이브러리를 수정하지 않고도 확장

중위 함수

❖ 중위 표현법(infix notation)

- 클래스의 멤버 호출 시 사용하는 점(.)을 생략하고 함수 이름 뒤에 소괄호를 생략해 직관적인 이름을 사용할 수 있는 표현법

중위 함수의 조건

- 멤버 메서드 또는 확장 함수여야 합니다.
- 하나의 매개변수를 가져야 합니다.
- `infix` 키워드를 사용하여 정의합니다.

코딩해 보세요! 중위 함수를 이용해 연산자처럼 사용하기

❖ InfixFunction.kt

```
package chap03.section5

fun main() {
    // 일반 표현법
    //val multi = 3.multiply(10)

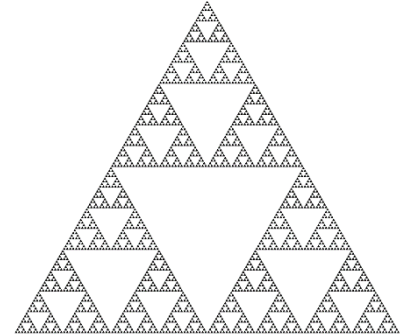
    // 중위 표현법
    val multi = 3 multiply 10
    println("multi: $multi")
}

// Int를 확장해서 multiply() 함수가 하나 더 추가되었음
infix fun Int.multiply(x: Int): Int { // infix로 선언되므로 중위 함수
    return this * x
}
```

꼬리 재귀 함수

❖ 재귀(recursion)란

- 자기 자신을 다시 참조
- 재귀 함수는 자기 자신을 계속 호출하는 특징



재귀 함수의 필수 조건

- 무한 호출에 빠지지 않도록 탈출 조건을 만들어 둔다.
- 스택 영역을 이용하므로 호출 횟수를 무리하게 많이 지정해 연산하지 않는다.
- 코드를 복잡하지 않게 한다.



무한 호출에 빠지는 경우 스택 오버플로(Stack Overflow) 가 발생할 수 있다.

꼬리 재귀 함수

❖ 꼬리 재귀 함수(tail recursive function)

- 스택에 계속 쌓이는 방식이 함수가 계속 실행되는 꼬리를 무는 형태
- 코틀린 고유의 `tailrec` 키워드를 사용해 선언

코딩해 보세요! 일반적인 factorial의 재귀 함수 만들기

❖ NormalFactorial.kt

```
package chap03.section5

fun main() {
    val number = 4
    val result: Long

    result = factorial(number)
    println("Factorial: $number -> $result")
}

fun factorial(n: Int): Long {
    return if (n == 1) n.toLong() else n * factorial(n-1)
}
```

```
factorial(4)
4*factorial(3)
4*(3*factorial(2))
4*(3*(2*factorial(1)))
4*(3*(2*1))
24
```

코딩해 보세요! 꼬리 재귀로 스택 오버플로 방지하기

❖ TailRecFactorial.kt

- 스택을 사용하지 않음

```
package chap03.section5.tailrec

fun main() {
    val number = 5
    println("Factorial: $number -> ${factorial(number)}")
}

tailrec fun factorial(n: Int, run: Int = 1): Long {
    return if (n == 1) run.toLong() else factorial(n-1, run*n)
}
```

피보나치 수열

❖ 피보나치 수열 재귀 함수

- 0, 1로 시작하여 n번째 수와 n+1번째 수의 합이 n+2번째 수가 되는 수열
 - 0, 1, 1, 2, 3, 5, 8...

```
fun fibonacci(n: Int, a: Long, b: Long): Long {  
    return if (n == 0) b else fibonacci(n-1, a+b, a)  
}
```

피보나치 수열

❖ 꼬리 재귀를 이용한 피보나치 수열

■ TailRecursionFibonacci.kt

```
package chap03.section5

import java.math.BigInteger

fun main() {
    val n = 100
    val first = BigInteger("0")
    val second = BigInteger("1")

    println(fibonacci(n, first, second))
}

// 꼬리 재귀 함수
tailrec fun fibonacci(n: Int, a: BigInteger, b: BigInteger): BigInteger {
    return if (n == 0) a else fibonacci(n-1, b, a+b)
}
```


03 함수와 함수형 프로그래밍

03-1 함수 선언하고 호출하기

03-2 함수형 프로그래밍

03-3 고차 함수와 람다식

03-4 고차 함수와 람다식 활용

03-5 코틀린의 다양한 함수들

03-6 함수와 변수의 범위



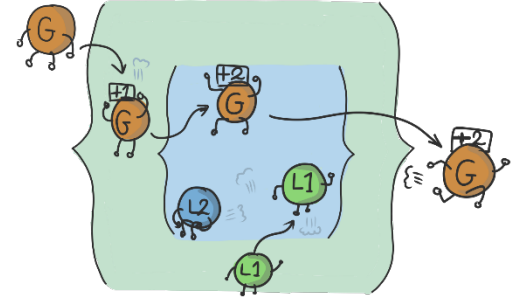
함수의 실행 블록

❖ 함수의 블록({ })

- 블록내에서 사용하는 변수 - 지역 변수(Local variable)

❖ 함수의 범위(Scope)

- 최상위 함수와 지역 함수



```
fun main() { // 최상위 레벨의 함수
    ...
    fun secondFunc(a: Int) { // 지역 함수 선언
        ...
    }
    userFunc(4) // 사용자 함수 사용 - 선언부의 위치에 상관 없이 사용
    secondFunc(2) // 지역 함수 사용 - 선언부가 먼저 나와야 사용 가능
}

fun userFunc(counts: Int) { // 사용자가 만든 최상위 레벨의 함수 선언
    ...
}
```

함수의 실행 블록

❖ 최상위 및 지역 함수의 사용 범위

- 최상위 함수는 `main()` 함수 전, 후 어디든 선언하고 코드 내에서 호출 가능
- 지역 함수는 먼저 선언되어야 그 함수를 호출할 수 있음

코딩해 보세요! 최상위 함수와 지역함수

❖ LocalFunctionRange.kt

```
package chap03.section6

fun a() = b() // 최상위 함수이므로 b()함수 선언 위치에 상관 없이 사용 가능
fun b() = println("b") // b()함수의 선언

fun c() {
    fun d() = e() // 오류! d()는 지역함수이며 e()의 이름을 모름
    fun e() = println("e")
}

fun main() {
    a() // 최상위 함수는 어디서든 호출될 수 있다.
    e() // 오류! c()함수에 정의된 e()는 c의 블록({ })을 벗어난 곳에서 사용할 수 없음
}
```

변수의 범위

❖ 전역 변수

- 최상위에 있는 변수로 프로그램이 실행되는 동안 삭제되지 않고 메모리에 유지
- 여러 요소가 동시에 접근하는 경우에 잘못된 동작을 유발할 수 있음
- 자주 사용되지 않는 전역 변수는 메모리 자원 낭비

❖ 지역변수

- 특정 코드 블록 내에서만 사용
- 블록 시작 시 임시로 사용되며 주로 스택 메모리를 사용

코딩해 보세요! 지역 변수와 전역 변수의 범위 - GlobalLocalVars.kt

```
var global = 10 // 전역 - 패키지의 모든 범위에 적용

fun main() {

    val local1 = 20 // main 블록 내에서만 유지
    val local2 = 21

    fun nestedFunc() {
        global += 1
        val local1 = 30 // func()함수 블록 내에서만 유지 (기존 local1이 가려짐)
        println("nestedFunc local1: $local1")
        println("nestedFunc local2: $local2") // 이 블록 바로 바깥의 main의 local2 사용
        println("nestedFunc global: $global")
    }
    nestedFunc()
    outsideFunc()
    println("main global: $global")
    println("main local1: $local1")
    println("main local2: $local2")
}

fun outsideFunc() {
    global += 1
    val outVal = "outside"
    println("outsideFunc global: $global")
    println("outsideFunc outVal: $outVal")
}
```