Introducción a Kotlin

emmanuel.lagarrigue@cs.uns.edu.ar

Clases, Objetos e Interfaces

Interfaces en Kotlin

```
interface Clickable {
    fun click()
}


class Button : Clickable {
    override fun click() = println("I was clicked")
}

>>> Button().click()
I was clicked
```

Interfaces en Kotlin

Regular method declaration

```
interface Clickable {
    fun click()
    fun showOff() = println("I'm clickable!")
}
```

Method with a default implementation

Modificadores de acceso: open, abstract y final (por defecto)

```
open class RichButton : Clickable {

    fun disable() {}

    open fun animate() {}

    override fun click() {}
}
```

This class is open: others can inherit from it.

This function is final: you can't override it in a subclass.

This function is open: you may override it in a subclass.

This function overrides an open function and is open as well.

open o final

```
open class RichButton : Clickable {
    final override fun click() {}
}
```

"final" isn't redundant here because "override" without "final" implies being open.

abstract

```
abstract class Animated {

    abstract fun animate()

    open fun stopAnimating() {
    }

    fun animateTwice() {
    }
}
```

This class is abstract: you can't create an instance of it.

This function is abstract: it doesn't have an implementation and must be overridden in subclasses.

Non-abstract functions in abstract classes aren't open by default but can be marked as open.

Modificadores de acceso en una clase

Modifier	Corresponding member	Comments
`final`	Can't be overridden	Used by default for class members
`open`	Can be overridden	Should be specified explicitly
`abstract`	Must be overridden	Can be used only in abstract classes; abstract members can't have an implementation
`override`	Overrides a member in a superclass or interface	Overridden member is open by default, if not marked `final`

Modificadores de visibilidad

Modifier	Class member	Top-level declaration
`public` (default)	Visible everywhere	Visible everywhere
`internal`	Visible in a module	Visible in a module
`protected`	Visible in subclasses	—
`private`	Visible in a class	Visible in a file

Clases selladas

```
interface Expr
class Num(val value: Int) : Expr
class Sum(val left: Expr, val right: Expr) : Expr

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

You have to check the "else" branch.

Clases selladas

```
sealed class Expr {
    class Num(val value: Int) : Expr()
    class Sum(val left: Expr, val right: Expr) : Expr()
}

fun eval(e: Expr): Int =
    when (e) {
        is Expr.Num -> e.value
        is Expr.Sum -> eval(e.right) + eval(e.left)
    }
```

Mark a base class as sealed ...

... and list all the possible subclasses as nested classes.

The "when" expression covers all possible cases, so no "else" branch is needed.

Constructor primario

```
class User(val nickname: String)
```

```
class User constructor(_nickname: String) {
    val nickname: String

    init {
        nickname = _nickname
    }
}
```

Primary constructor with one parameter

Initializer block

Constructor primario

```
class User(_nickname: String) {
    val nickname = _nickname
}
```

Primary constructor with one parameter

The property is initialized with the parameter.

```
class User(val nickname: String)
```

"val" means the corresponding property is generated for the constructor parameter.

```
class User(val nickname: String,
           val isSubscribed: Boolean = true)
```

Provides a default value for the constructor parameter

Constructor primario

```
>>> val alice = User("Alice")
>>> println(alice.isSubscribed)
true
>>> val bob = User("Bob", false)
>>> println(bob.isSubscribed)
false
>>> val carol = User("Carol", isSubscribed = false)
>>> println(carol.isSubscribed)
false
```

Uses the default value "true" for the isSubscribed parameter

You can specify all parameters according to declaration order.

You can explicitly specify names for some constructor arguments.

Constructor primario

```
open class User(val nickname: String) { ... }

class TwitterUser(nickname: String) : User(nickname) { ... }
```

Constructor primario

```
open class Button
```

The default constructor without arguments is generated.

```
class RadioButton: Button()
```

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

This class has a private constructor.

Constructores secundarios

```
open class View {
    constructor(ctx: Context) {
        // some code
    }
    constructor(ctx: Context, attr: AttributeSet) {
        // some code
    }
}
```

Constructores secundarios

```
class MyButton : View {
    constructor(ctx: Context)
        : super(ctx) {
        // ...
    }
    constructor(ctx: Context, attr: AttributeSet)
        :  super(ctx, attr) {
        // ...
    }
}
```

Implementado interfaces

```
interface User {
    val nickname: String
}
```

Implementado interfaces

```
class PrivateUser(override val nickname: String) : User      <- Primary constructor property

class SubscribingUser(val email: String) : User {
    override val nickname: String
        get() = email.substringBefore('@') )                  <- Custom getter
}

class FacebookUser(val accountId: Int) : User {
    override val nickname = getFacebookName(accountId)        <- Property initializer
}

>>> println(PrivateUser("test@kotlinlang.org").nickname)
test@kotlinlang.org
>>> println(SubscribingUser("test@kotlinlang.org").nickname)
test
```

Métodos generados por el compilador

Métodos de object

```
class Client(val name: String, val postalCode: Int)
```

toString()

```
class Client(val name: String, val postalCode: Int) {
    override fun toString() = "Client(name=$name, postalCode=$postalCode)"
}
```

```
>>> val client1 = Client("Alice", 342562)
>>> println(client1)
Client(name=Alice, postalCode=342562)
```

equals()

```
class Client(val name: String, val postalCode: Int) {
    override fun equals(other: Any?): Boolean {
        if (other == null || other !is Client)
            return false
        return name == other.name &&
               postalCode == other.postalCode
    }
    override fun toString() = "Client(name=$name, postalCode=$postalCode)"
}
```

Checks whether "other" is a Client

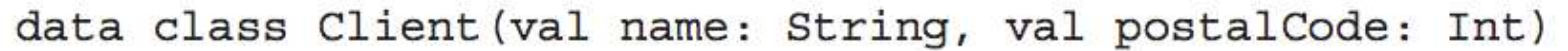
Checks whether the corresponding properties are equal

"Any" is the analogue of java.lang.Object: a superclass of all classes in Kotlin. The nullable type "Any?" means "other" can be null.

hashCode()

```
class Client(val name: String, val postalCode: Int) {
    ...
    override fun hashCode(): Int = name.hashCode() * 31 + postalCode
}
```

Data clases



```
data class Client(val name: String, val postalCode: Int)
```

object keyword

Singletons

```
object Payroll {
    val allEmployees = arrayListOf<Person>()

    fun calculateSalary() {
        for (person in allEmployees) {
            ...
        }
    }
}
```

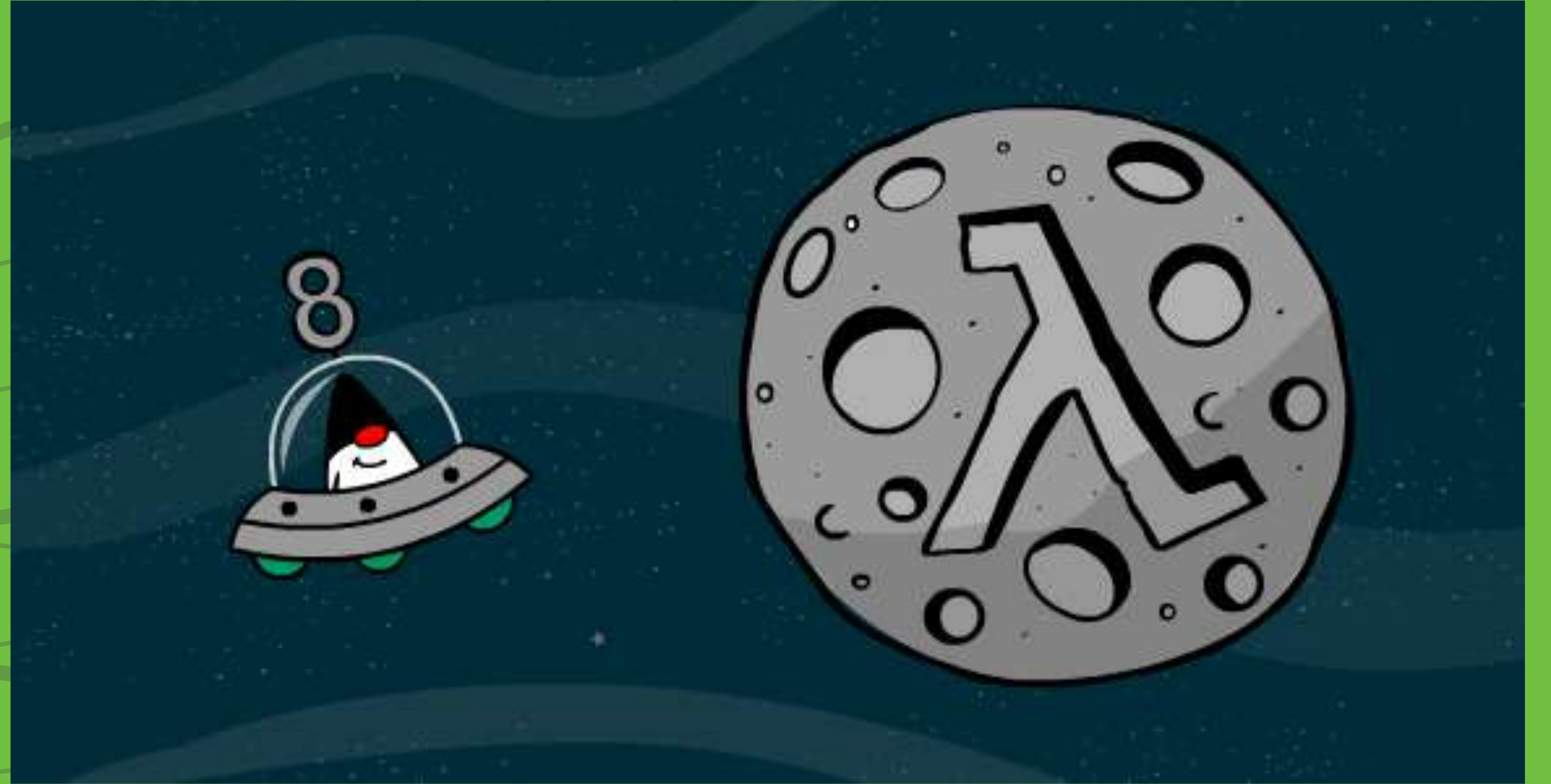
```
Payroll.allEmployees.add(Person(...))

Payroll.calculateSalary()
```

Singletons

```
data class Person(val name: String) {
    object NameComparator : Comparator<Person> {
        override fun compare(p1: Person, p2: Person): Int =
            p1.name.compareTo(p2.name)
} }

>>> val persons = listOf(Person("Bob"), Person("Alice"))
>>> println(persons.sortedWith(Person.NameComparator))
[Person(name=Alice), Person(name=Bob)]
```

Singletons

```
/* Java */
CaseInsensitiveFileComparator.INSTANCE.compare(file1, file2);
```



Programando con lambdas

Expresiones Lambda

```
/* Java */
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View view) {
        /* actions on click */
    }
});
```

Kotlin

```
button.setOnClickListener { /* actions on click */ }
```

Lambdas y colecciones

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

fun findTheOldest(people: List<Person>) {
    var maxAge = 0
    var theOldest: Person? = null
    for (person in people) {
        if (person.age > maxAge) {
            maxAge = person.age
            theOldest = person
        }
    }
    println(theOldest)
}
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> findTheOldest(people)
Person(name=Bob, age=31)
```

Stores the maximum age

Stores a person of the maximum age

If the next person is older than the current oldest person, changes the maximum

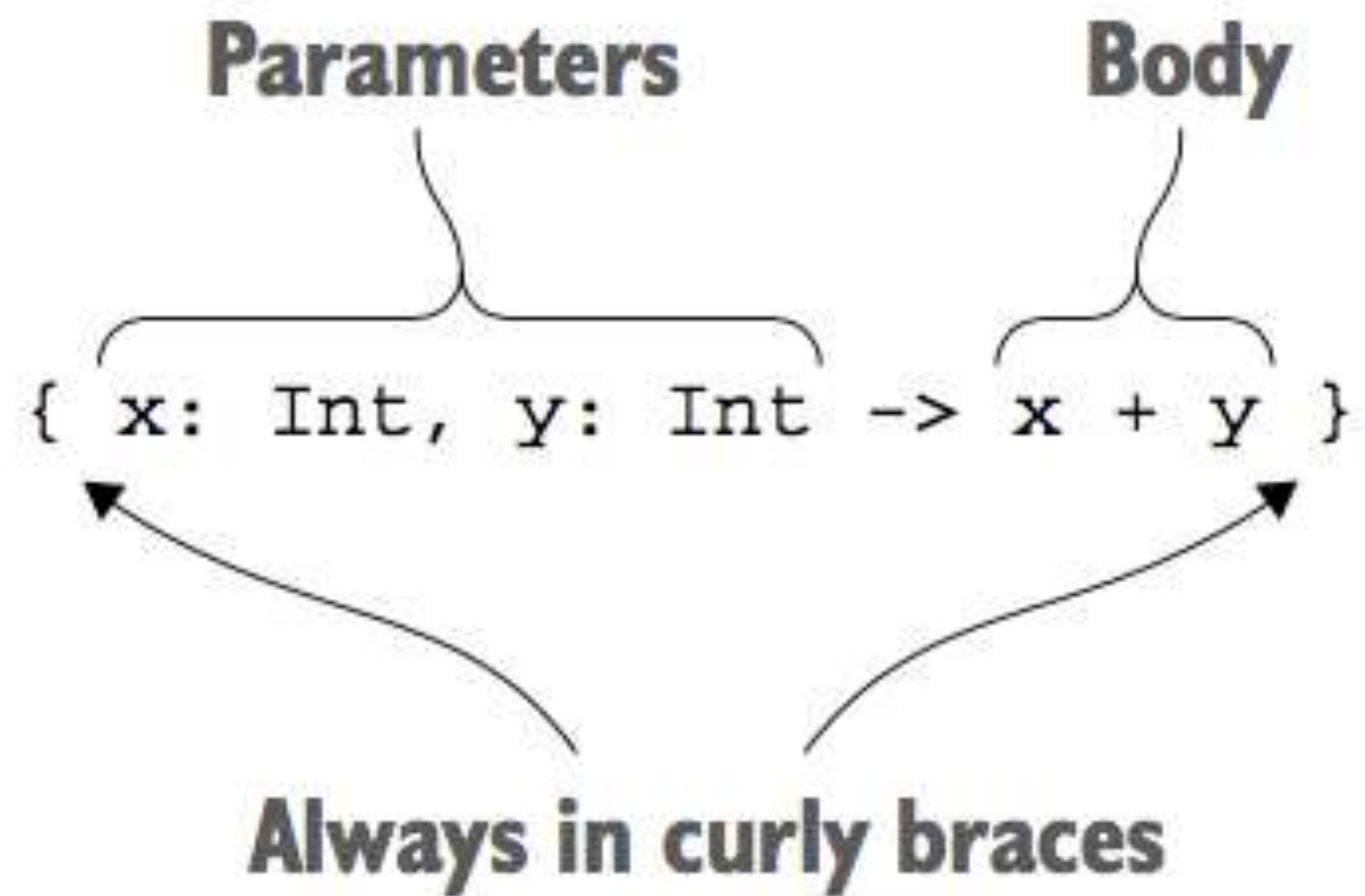
Lambdas y colecciones

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.maxBy { it.age })
Person(name=Bob, age=31)
```

Finds the maximum by comparing the ages

```
people.maxBy(Person::age)
```

Sintaxis



Sintaxis

```
>>> val sum = { x: Int, y: Int -> x + y }
>>> println(sum(1, 2))
3
```

Calls the lambda stored in a variable

Sintaxis

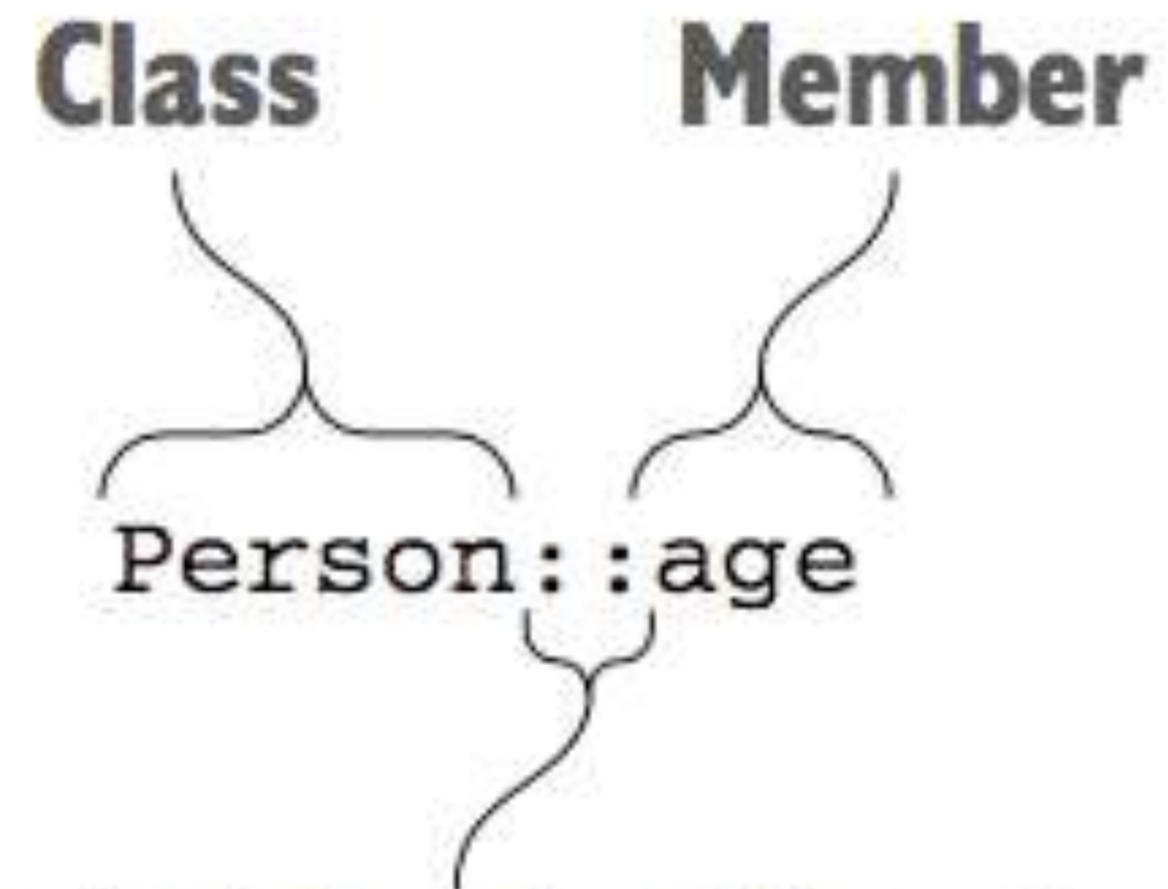
```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.maxBy { it.age })
Person(name=Bob, age=31)
```

```
people.maxBy({ p: Person -> p.age })
```

Sintaxis

```
people.maxBy({ p: Person -> p.age })

people.maxBy() { p: Person -> p.age }

people.maxBy { p -> p.age }

people.maxBy { it.age }

people.maxBy(Person::age)
```

Member references



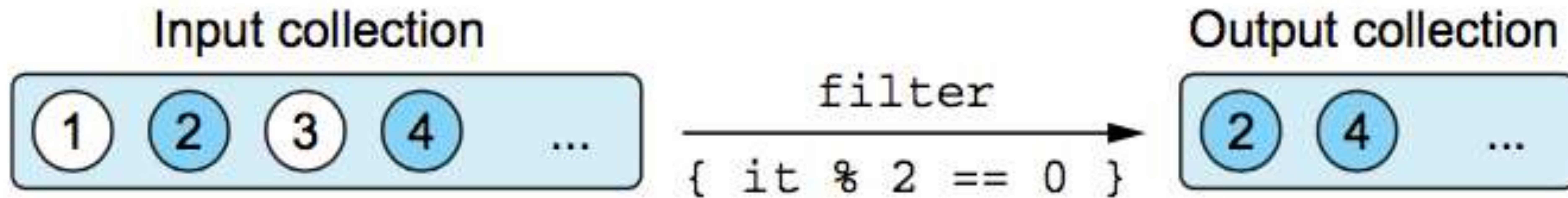
Functional APIs for collections

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

filter

```
>>> val list = listOf(1, 2, 3, 4)
>>> println(list.filter { it % 2 == 0 })
[2, 4]
```

Only even numbers remain.

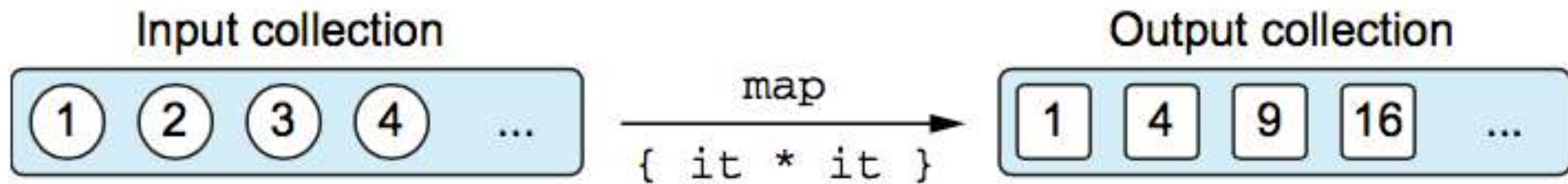


filter

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.filter { it.age > 30 })
[Person(name=Bob, age=31)]
```

map

```
>>> val list = listOf(1, 2, 3, 4)
>>> println(list.map { it * it })
[1, 4, 9, 16]
```



map

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.map { it.name })
[Alice, Bob]
```

```
people.map(Person::name)
```

filter y map

```
>>> people.filter { it.age > 30 }.map(Person::name)
[Bob]
```

```
>>> val numbers = mapOf(0 to "zero", 1 to "one")
>>> println(numbers.mapValues { it.value.toUpperCase() })
{0=ZERO, 1=ONE}
```

all, any, count, find

```
val canBeInClub27 = { p: Person -> p.age <= 27 }
```

all, any, count, find

```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.all(canBeInClub27))
false
```

all, any, count, find

```
>>> println(people.any(canBeInClub27))
true
```

all, any, count, find

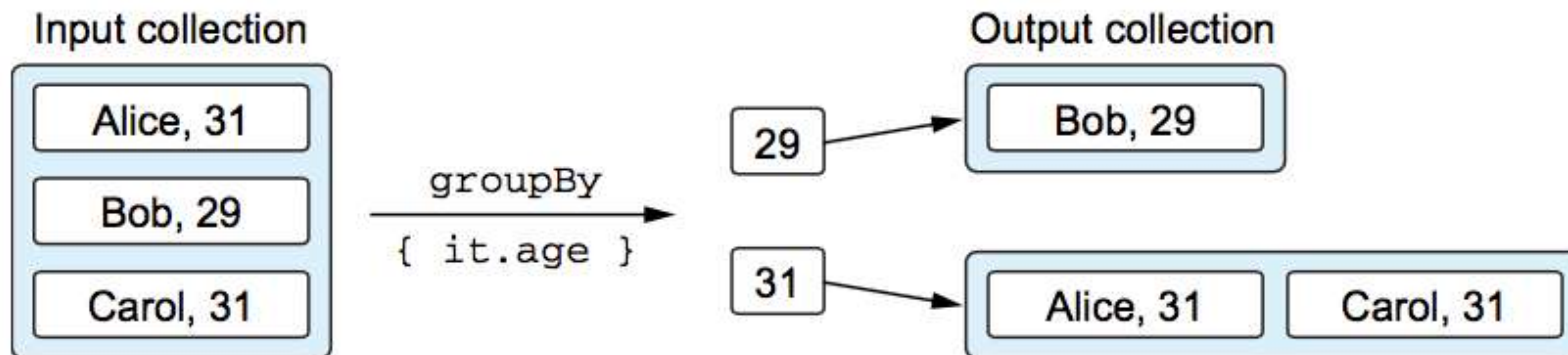
```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.count(canBeInClub27))
1
```

all, any, count, find

```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.find(canBeInClub27))
Person(name=Alice, age=27)
```

groupBy

```
>>> val people = listOf(Person("Alice", 31),
...             Person("Bob", 29), Person("Carol", 31))
>>> println(people.groupBy { it.age })

{29=[Person(name=Bob, age=29)],
 31=[Person(name=Alice, age=31), Person(name=Carol, age=31)]}
```



groupBy

```
>>> val list = listOf("a", "ab", "b")
>>> println(list.groupBy(String::first))
{a=[a, ab], b=[b]}
```

flatMap, flatten

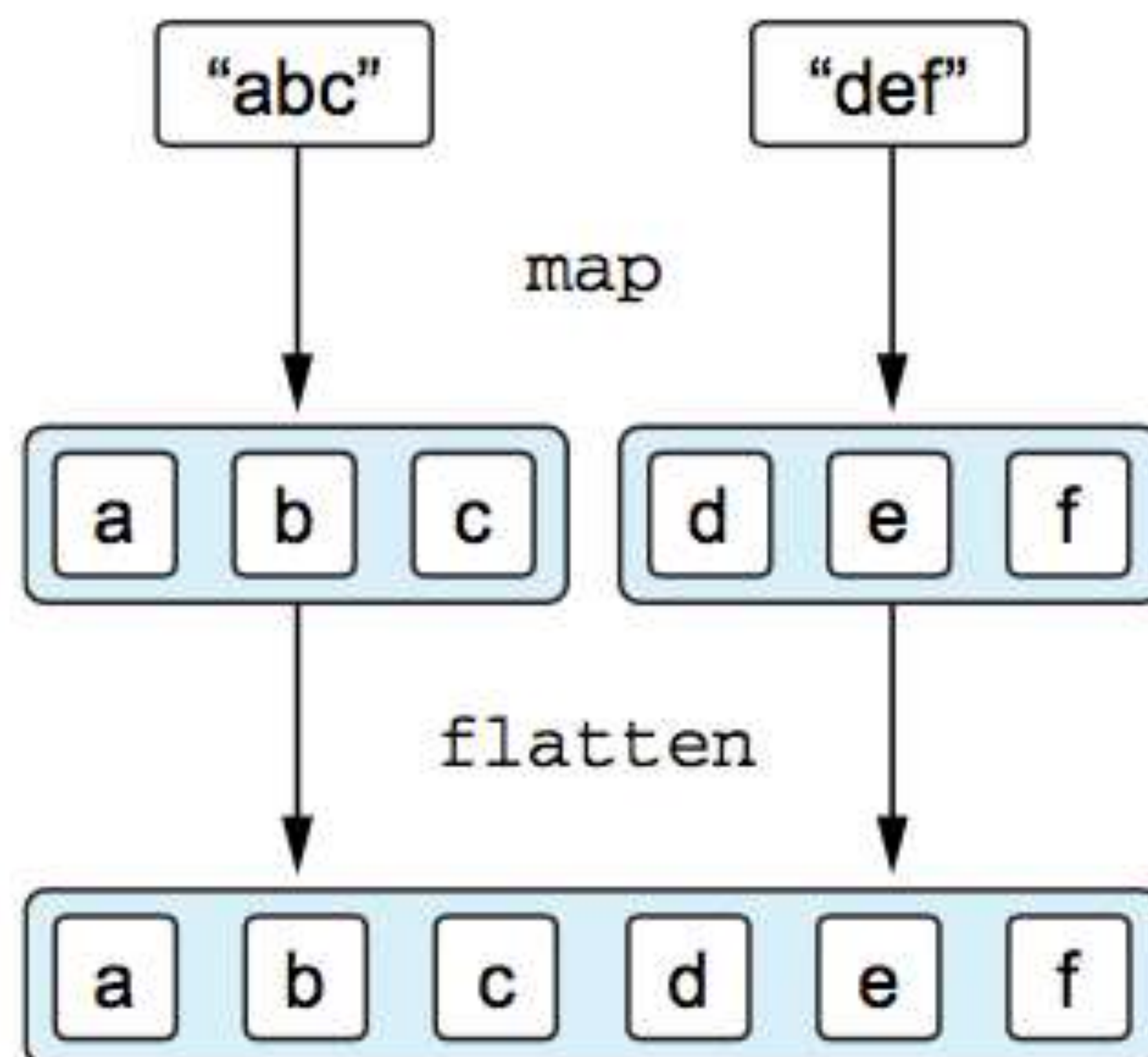
```
class Book(val title: String, val authors: List<String>)
```

```
books.flatMap { it.authors }.toSet()
```

Set of all authors who wrote books in the “books” collection

flatMap, flatten

```
>>> val strings = listOf("abc", "def")
>>> println(strings.flatMap { it.toList() })
[a, b, c, d, e, f]
```



flatMap, flatten

```
>>> val books = listOf(Book("Thursday Next", listOf("Jasper Fforde")),
...                    Book("Mort", listOf("Terry Pratchett")),
...                    Book("Good Omens", listOf("Terry Pratchett",
...                                              "Neil Gaiman")))
>>> println(books.flatMap { it.authors }.toSet())
[Jasper Fforde, Terry Pratchett, Neil Gaiman]
```

Usando interfaces java

```
/* Java */
public class Button {
    public void setOnClickListener(OnClickListener l) { ... }
}

/* Java */
public interface OnClickListener {
    void onClick(View v);
}
```

Usando interfaces java

```
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        ...
    }
}
```

```
button.setOnClickListener { view -> ... }
```

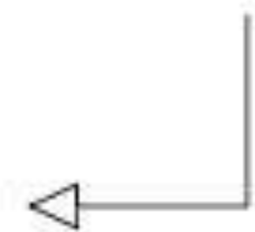
```
public interface OnClickListener {
    void onClick(View v);                ——→  { view -> ... }
}
```

Algunas funciones de Kotlin

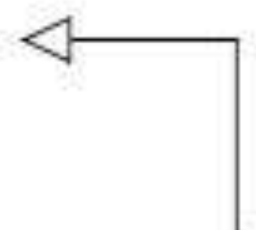
```
fun alphabet(): String {
    val result = StringBuilder()
    for (letter in 'A'..'Z') {
        result.append(letter)
    }
    result.append("\nNow I know the alphabet!")
    return result.toString()
}
>>> println(alphabet())
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Now I know the alphabet!
```

with

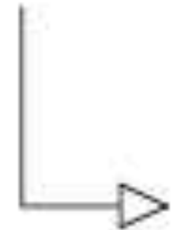
```
fun alphabet(): String {
    val stringBuilder = StringBuilder()
    return with(stringBuilder) {
        for (letter in 'A'..'Z') {
            this.append(letter)
        }
        append("\nNow I know the alphabet!")
        this.toString()
    }
}
```



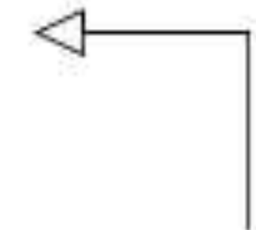
Specifies the receiver value on which you're calling the methods



Calls a method on the receiver value though an explicit "this"



Calls a method, omitting "this"



Returns a value from the lambda

apply

```
fun alphabet() = StringBuilder().apply {
    for (letter in 'A'..'Z') {
        append(letter)
    }
    append("\nNow I know the alphabet!")
}.toString()
```

Sistema de tipos de Kotlin

Nullability

Nullable types

```
/* Java */
int strLen(String s) {
    return s.length();
}
```

```
fun strLen(s: String) = s.length
```

```
>>> strLen(null)
ERROR: Null can not be a value of a non-null type String
```

Nullable types

```
fun strLenSafe(s: String?) = ...
```

Type? = Type or null

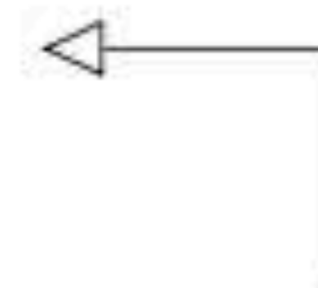
Nullable types

```
>>> fun strLenSafe(s: String?) = s.length()
ERROR: only safe (?.) or non-null asserted (!!.) calls are allowed
 on a nullable receiver of type kotlin.String?
```

```
>>> val x: String? = null
>>> var y: String = x
ERROR: Type mismatch: inferred type is String? but String was expected
```

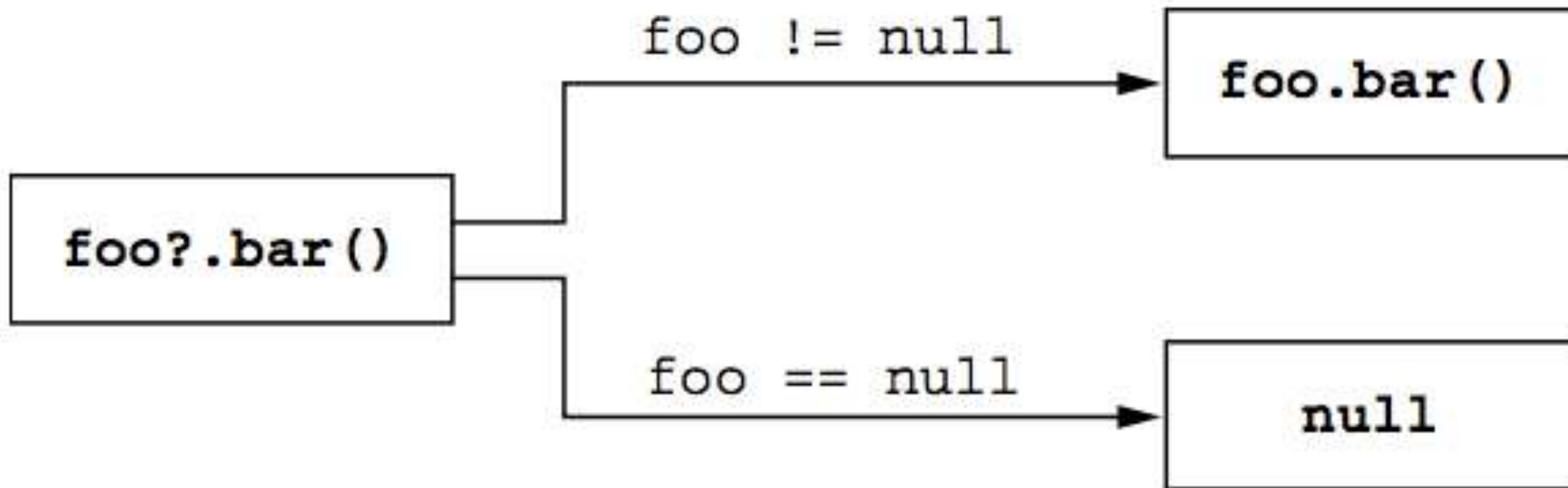
Nullable types

```
fun strLenSafe(s: String?): Int =
    if (s != null) s.length else 0

>>> val x: String? = null
>>> println(strLenSafe(x))
0
>>> println(strLenSafe("abc"))
3
```



By adding the check for null, the code now compiles.

Safe call operator “?.”



Safe call operator “?.”

```
fun printAllCaps(s: String?) {
    val allCaps: String? = s?.toUpperCase()
    println(allCaps)
}
```

allCaps may be null.

```
>>> printAllCaps("abc")
ABC
>>> printAllCaps(null)
null
```

Safe call operator “?.”

```
class Employee(val name: String, val manager: Employee?)

fun managerName(employee: Employee): String? = employee.manager?.name

>>> val ceo = Employee("Da Boss", null)
>>> val developer = Employee("Bob Smith", ceo)
>>> println(managerName(developer))
Da Boss
>>> println(managerName(ceo))
null
```

Safe call operator “?.”

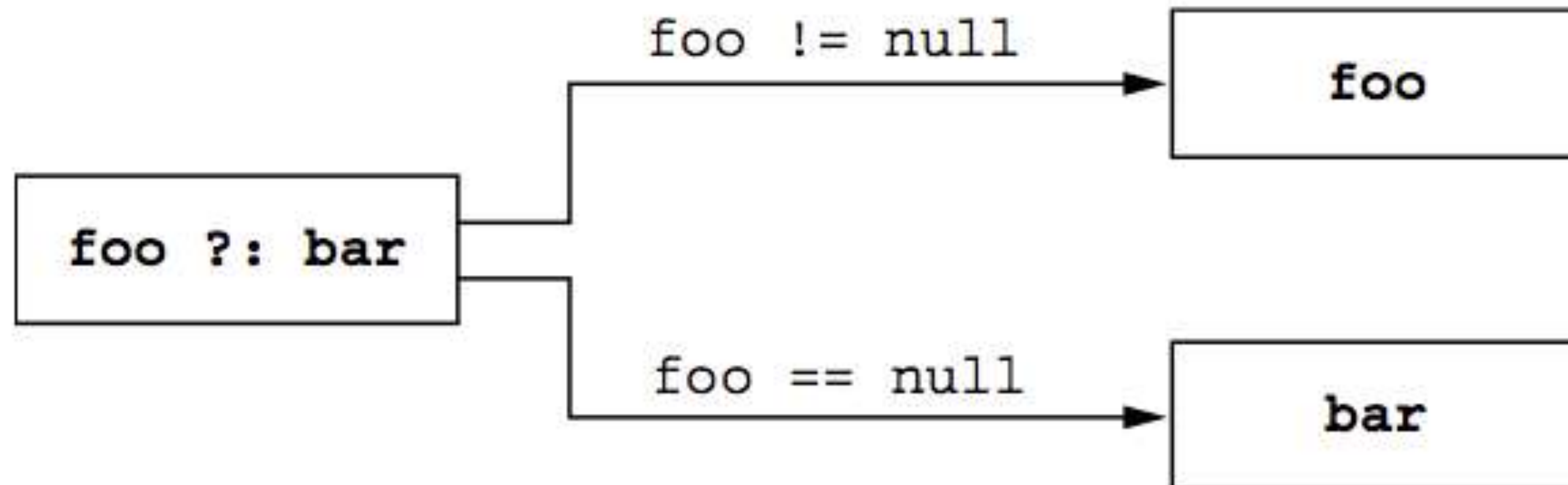
```
class Address(val streetAddress: String, val zipCode: Int,
              val city: String, val country: String)

class Company(val name: String, val address: Address?)

class Person(val name: String, val company: Company?)

fun Person.countryName(): String {
   val country = this.company?.address?.country
   return if (country != null) country else "Unknown"
}
>>> val person = Person("Dmitry", null)
>>> println(person.countryName())
Unknown
```

Several safe-call operators can be in a chain.

Elvis operator “?:”

```
fun foo(s: String?) {
    val t: String = s ?: ""
}
```

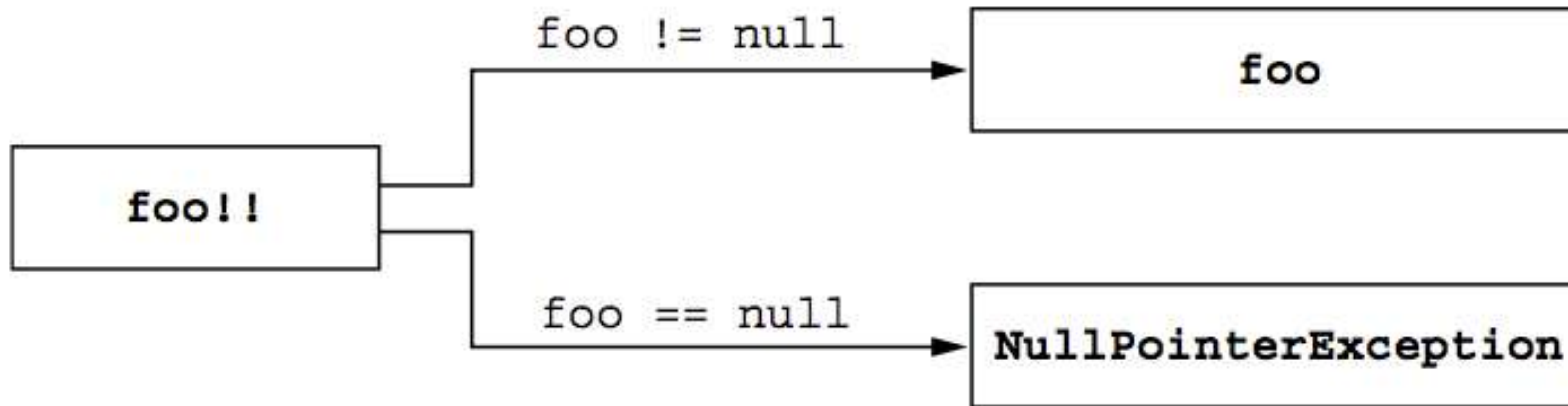
If “s” is null, the result is an empty string.



Elvis operator “?:”

```
fun strLenSafe(s: String?): Int = s?.length ?: 0

>>> println(strLenSafe("abc"))
3
>>> println(strLenSafe(null))
0


fun Person.countryName() =
    company?.address?.country ?: "Unknown"
```

Not-null assertions: “!!”



Not-null assertions: “!!”

```
class CopyRowAction(val list: JList<String>) : AbstractAction() {
    override fun isEnabled(): Boolean =
        list.selectedValue != null

    override fun actionPerformed(e: ActionEvent) {
        val value = list.selectedValue!!
        // copy value to clipboard
    }
}
```

actionPerformed is called only if isEnabled returns “true”.

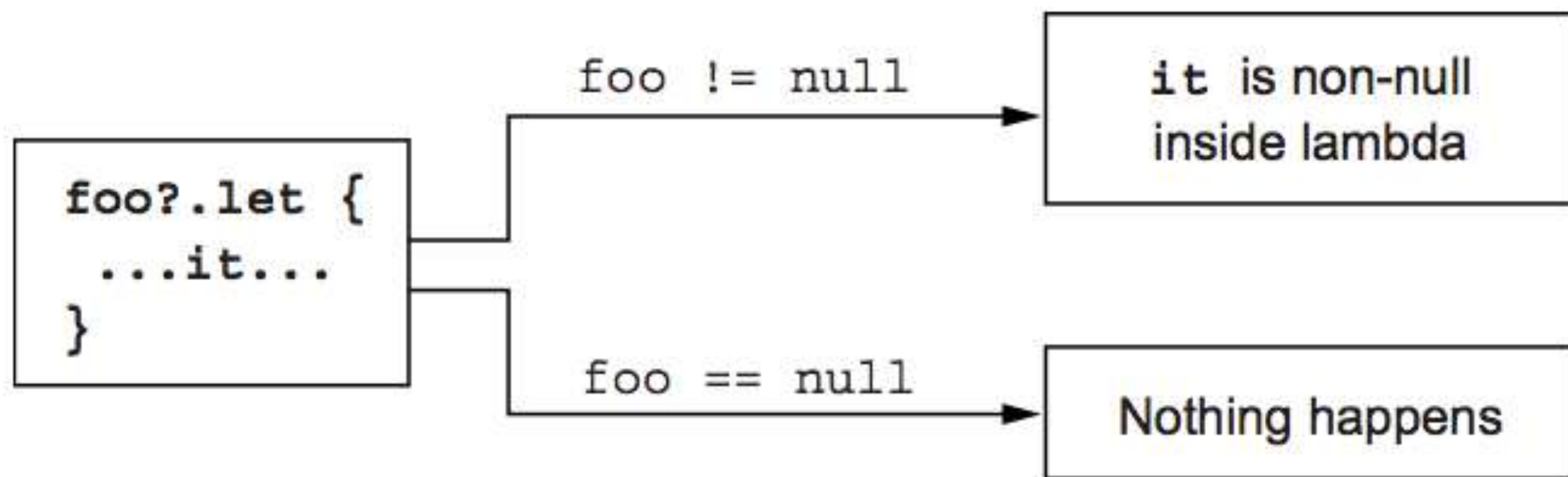
let

```
fun sendEmailTo(email: String) { /*...*/ }
```

```
>>> val email: String? = ...
>>> sendEmailTo(email)
ERROR: Type mismatch: inferred type is String? but String was expected
```

```
if (email != null) sendEmailTo(email)
```

let



let

```
email?.let { email -> sendEmailTo(email) }

email?.let { sendEmailTo(it) }
```

let

```
val person: Person? = getTheBestPersonInTheWorld()
if (person != null) sendEmailTo(person.email)
```

```
getTheBestPersonInTheWorld()?.let { sendEmailTo(it.email) }
```

Tipos primitivos y otros tipos básicos

```
val i: Int = 1
val list: List<Int> = listOf(1, 2, 3)
```

Tipos primitivos

- *Integer types*—`Byte`, `Short`, `Int`, `Long`
- *Floating-point number types*—`Float`, `Double`
- *Character type*—`Char`
- *Boolean type*—`Boolean`

Nullable primitive type

```
data class Person(val name: String,
                  val age: Int? = null) {

    fun isOlderThan(other: Person): Boolean? {
        if (age == null || other.age == null)
            return null
        return age > other.age
    }
}

>>> println(Person("Sam", 35).isOlderThan(Person("Amy", 42)))
false
>>> println(Person("Sam", 35).isOlderThan(Person("Jane")))
null
```

Conversiones de números

```
val i = 1
val l: Long = i
```

Error: type mismatch

```
val i = 1
val l: Long = i.toLong()
```

Tipos base “Any” and “Any?”

```
val answer: Any = 42
```

“Unit”: el “void” de Kotlin

```
fun f(): Unit { ... }

fun f() { ... }
```

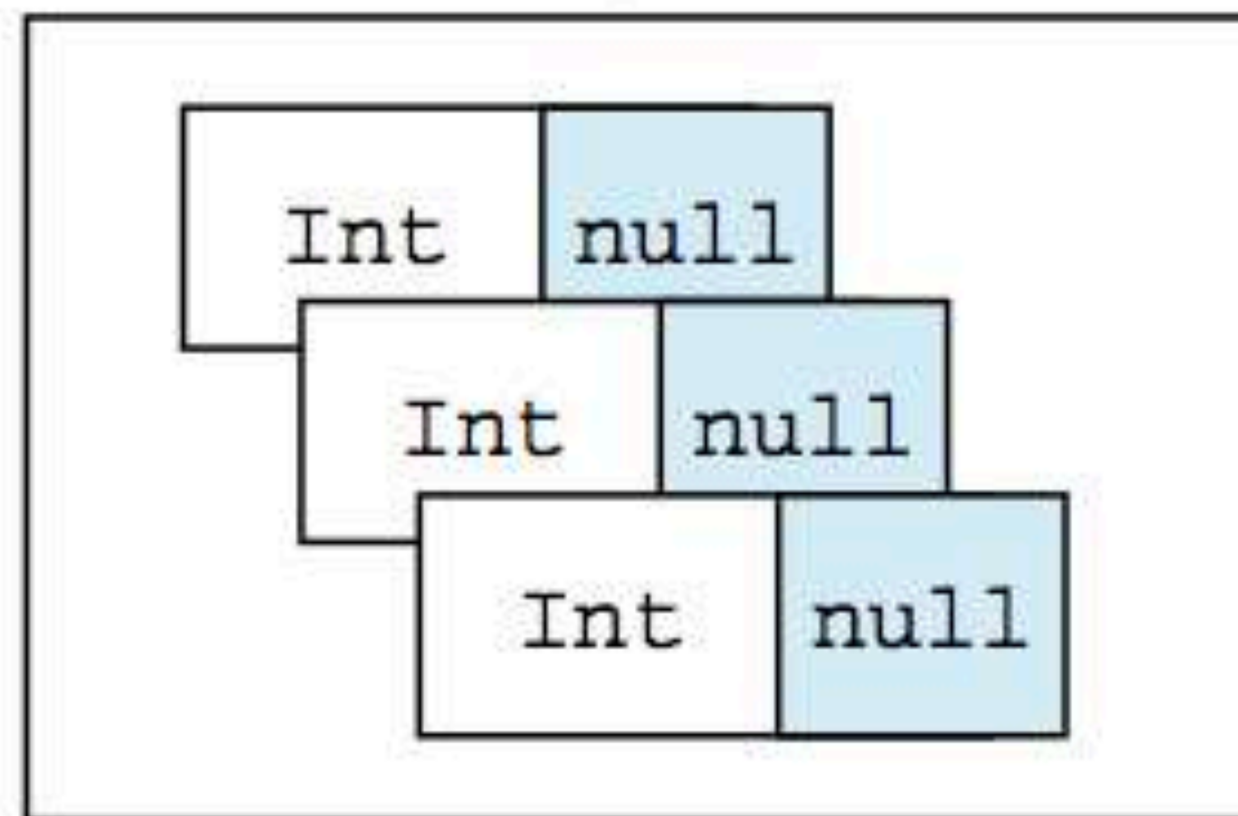
“Unit”: el “void” de Kotlin

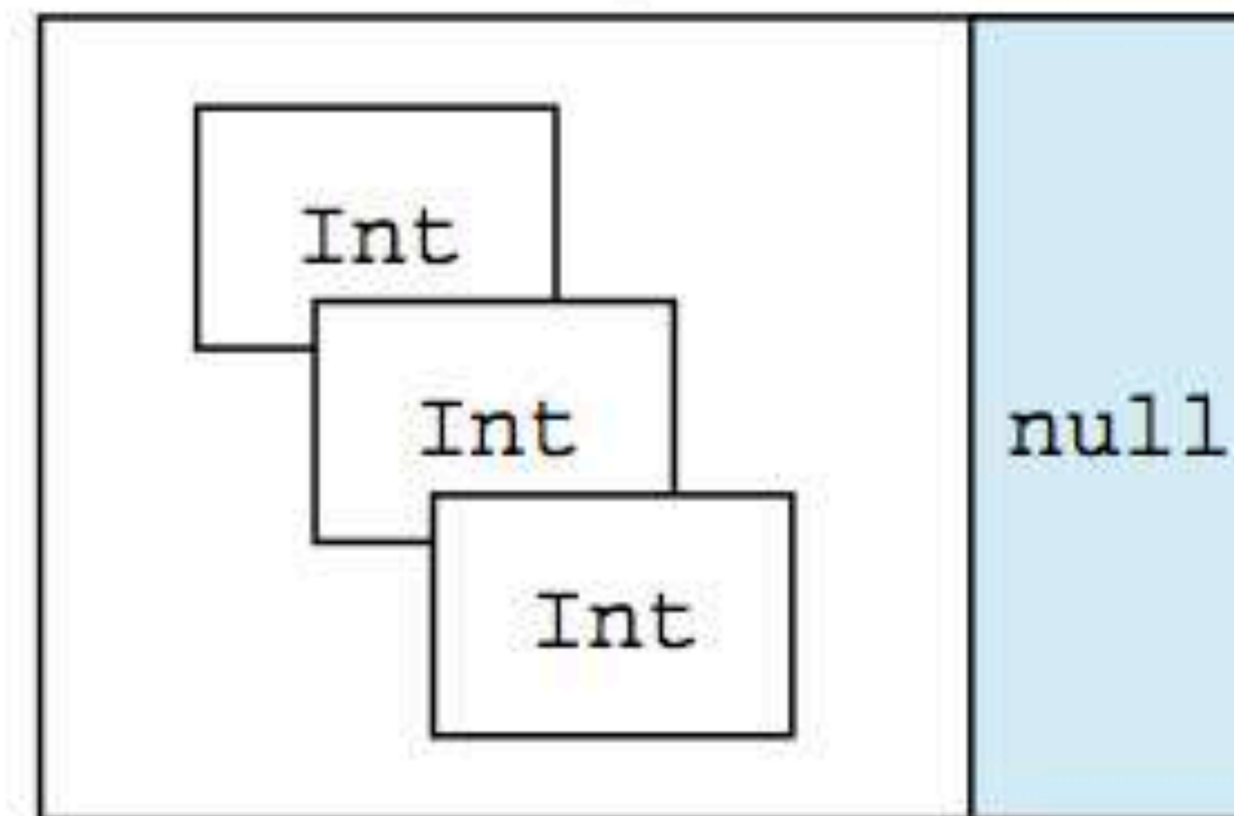
```
interface Processor<T> {
    fun process(): T
}

class NoResultProcessor : Processor<Unit> {
    override fun process() {
        // do stuff
    }
}
```

Returns Unit, but you omit the type specification

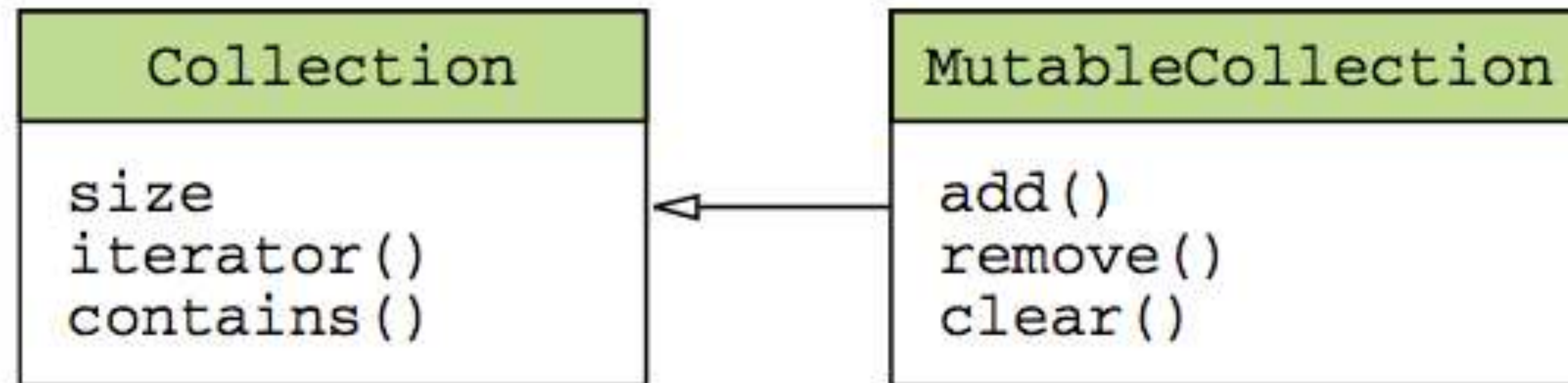
You don’t need an explicit return here.

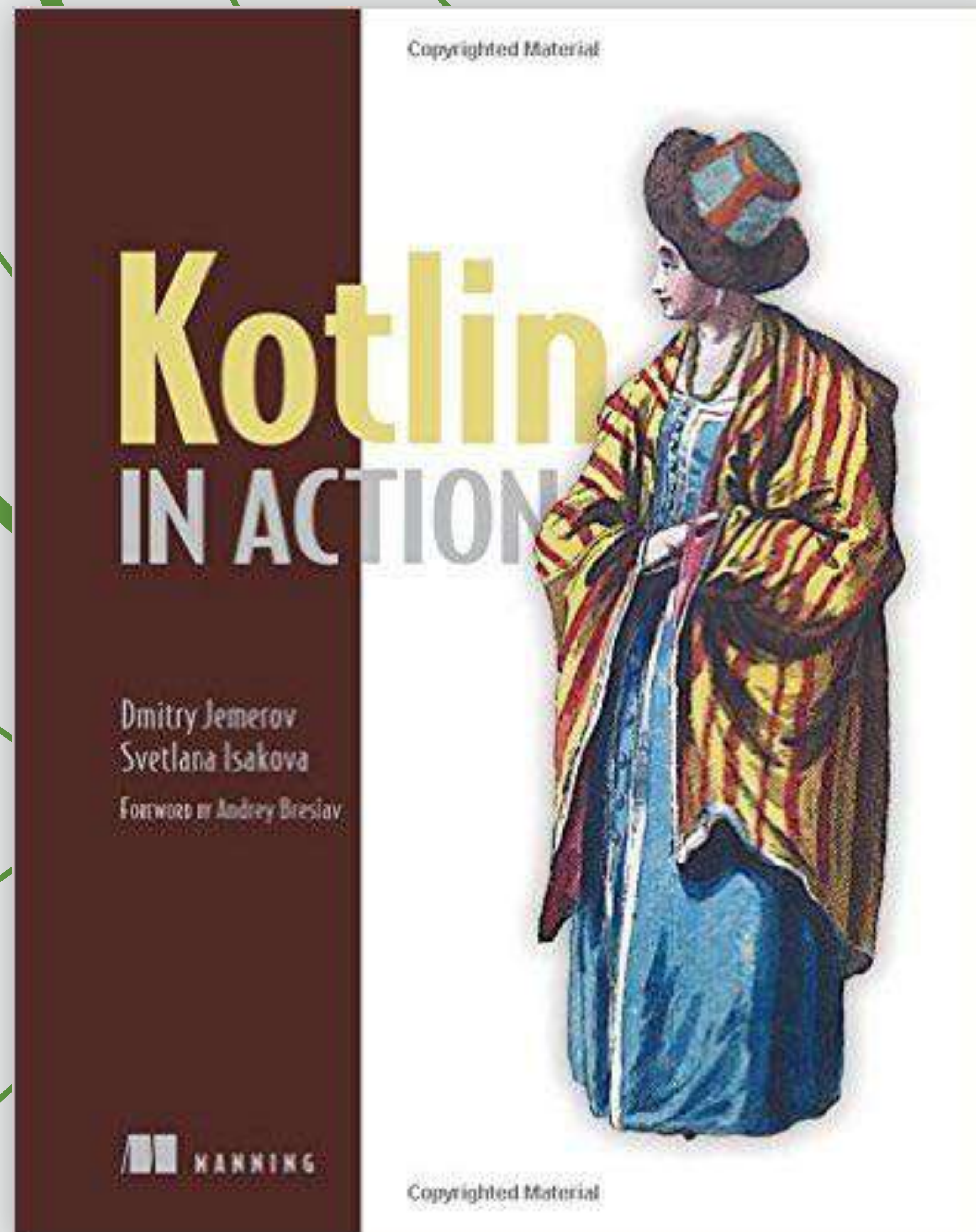
Nullability and collections

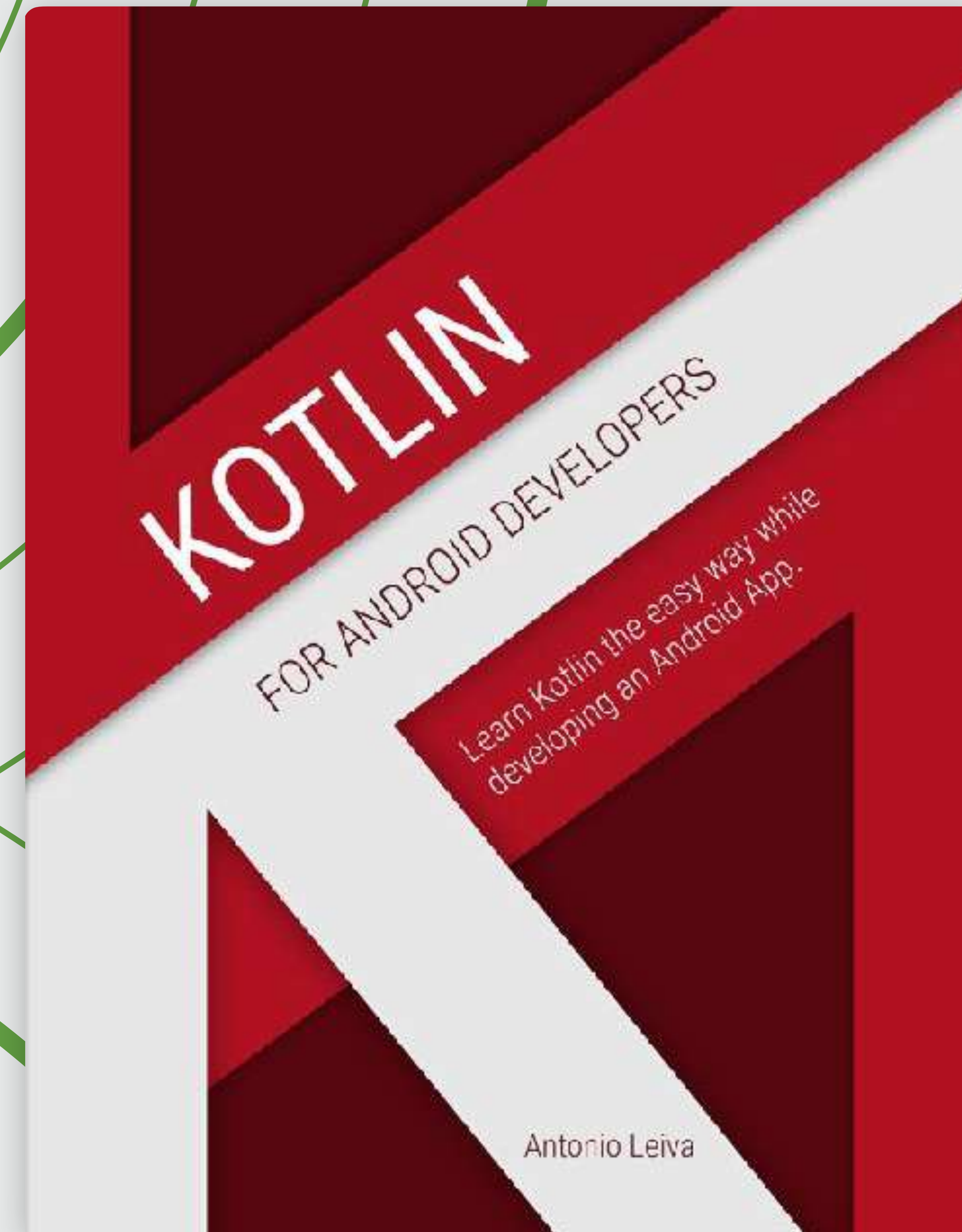




Read-only and mutable collections







https://kotlinlang.org