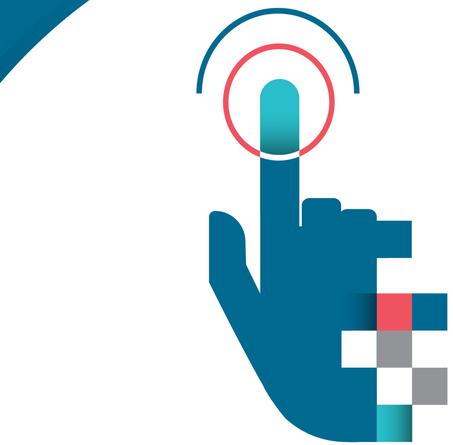
2019

1° ESCUELA DE ACTUALIZACIÓN EN TECNOLOGÍAS DE INFORMÁTICA

PROGRAMACIÓN EN KOTLIN PARA PLATAFORMAS JAVA Y ANDROID

LIC. EMMANUEL LAGARRIGUE LAZARTE

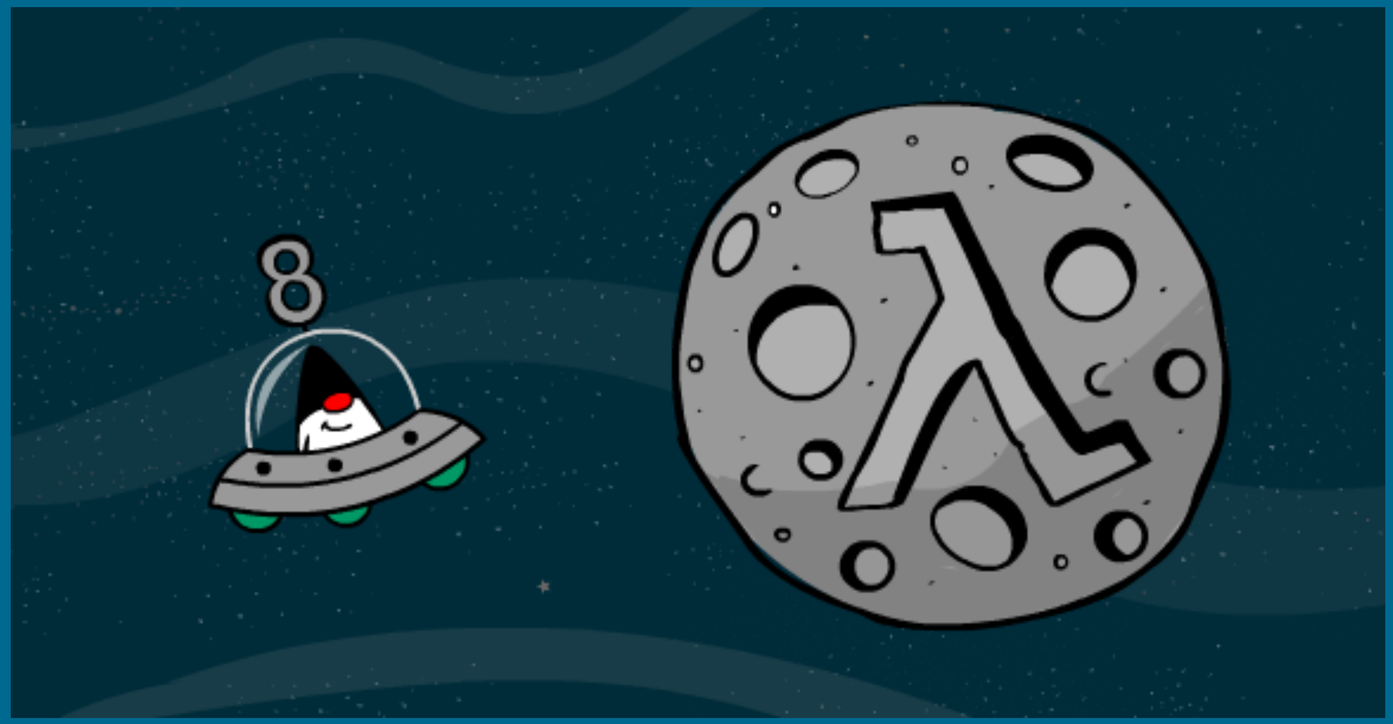


eati

DCIC DEPARTAMENTO DE CIENCIAS E INGENIERÍA DE LA COMPUTACIÓN

UNS UNIVERSIDAD NACIONAL DEL SUR

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

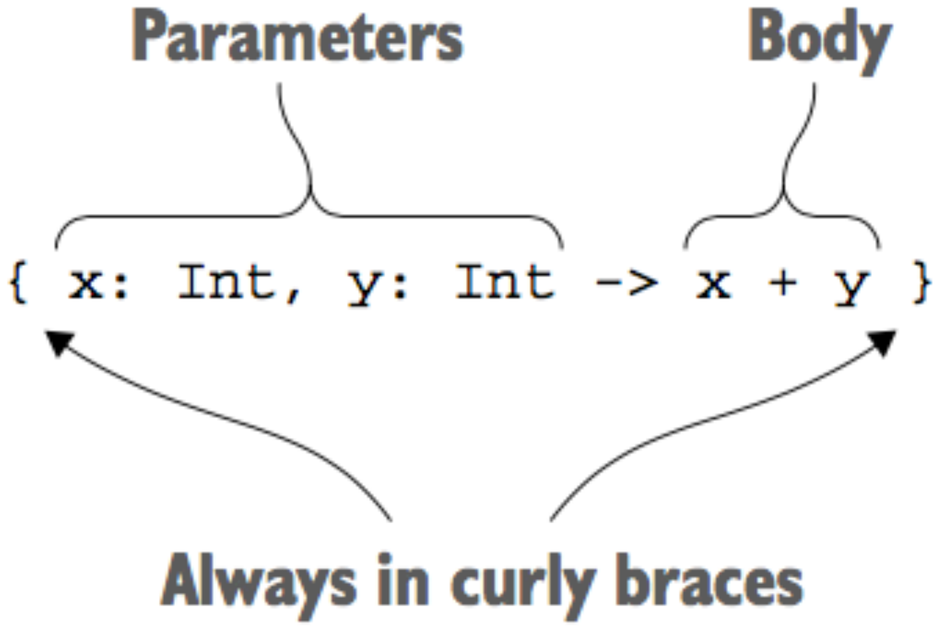
Lambdas y colecciones

```
>>> 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)
```

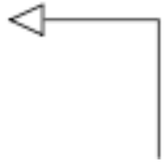
Sintaxis





Sintaxis

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



Calls the lambda stored in a variable



Sintaxis

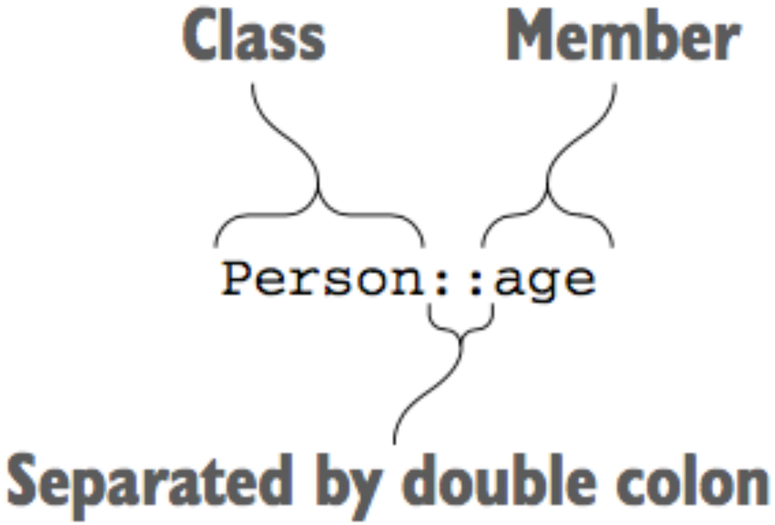
```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.maxBy { it.age })
Person(name=Bob, age=31)
```

```
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)
```

forEach

```
fun printMessagesWithPrefix(messages: Collection<String>, prefix: String) {
    messages.forEach {
        println("$prefix $it")
    }
}

>>> val errors = listOf("403 Forbidden", "404 Not Found")
>>> printMessagesWithPrefix(errors, "Error:")
Error: 403 Forbidden
Error: 404 Not Found
```

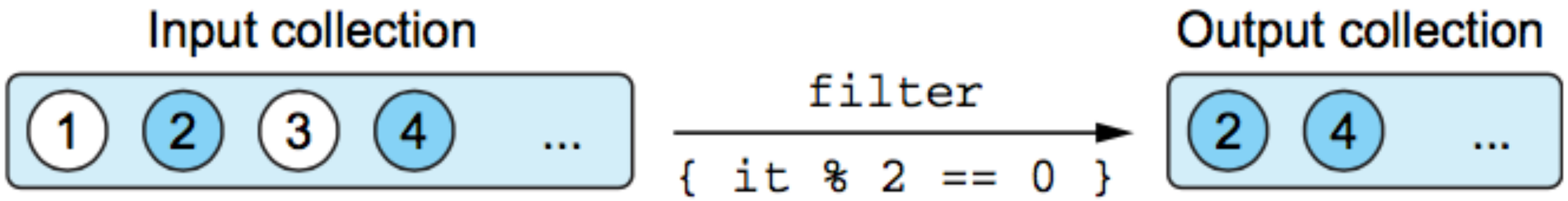
Accesses the "prefix" parameter in the lambda

Takes as an argument a lambda specifying what to do with each element

filter

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

Only even numbers remain.

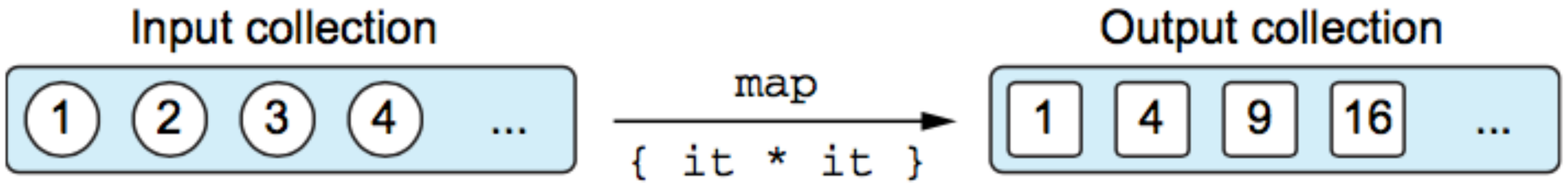



filter

```
>>> 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

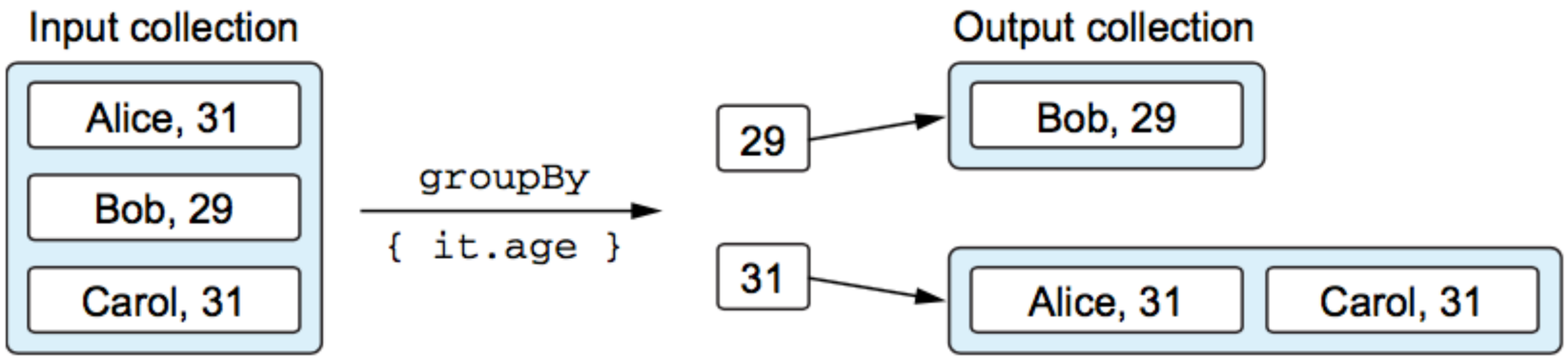
```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.count(canBeInClub27))
1
```


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

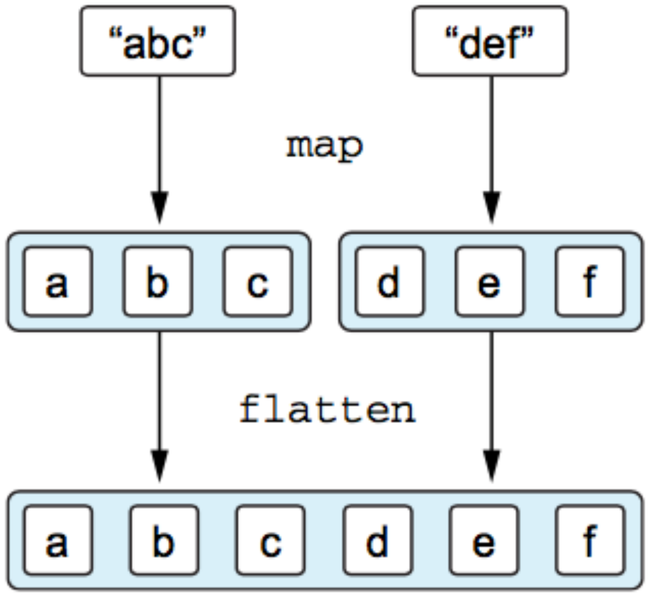
```
class Book(val title: String, val authors: List<String>)
```

```
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

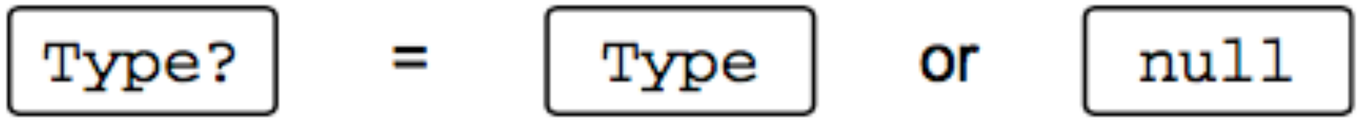
```
/* Java */
int strLen(String s) {
    return s.length();
}
```

```
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?) = ...
```



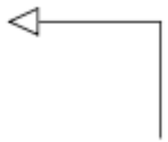

Nullable types

```
>>> 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
```

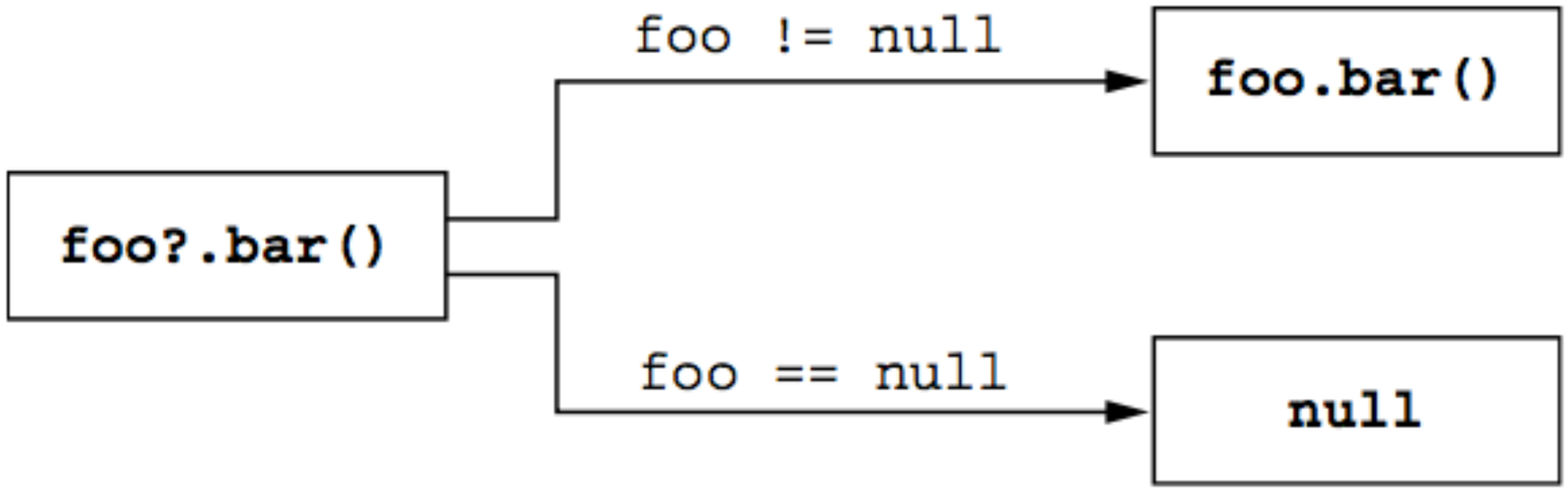
Nullable types

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

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



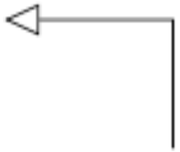
By adding the check for null, the code now compiles.

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 “?.”

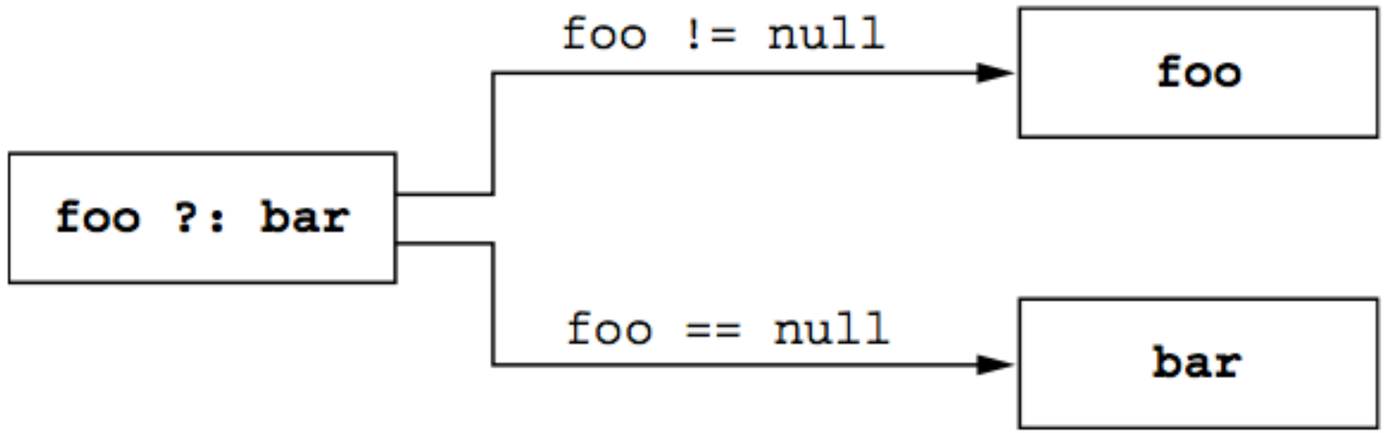
```
class Address(val streetAddress: String, val zipCode: Int,
              val city: String, val country: String)

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.

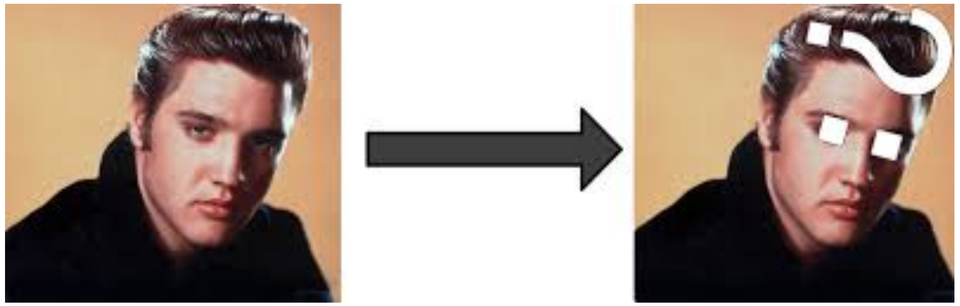
Elvis operator “?:”

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

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

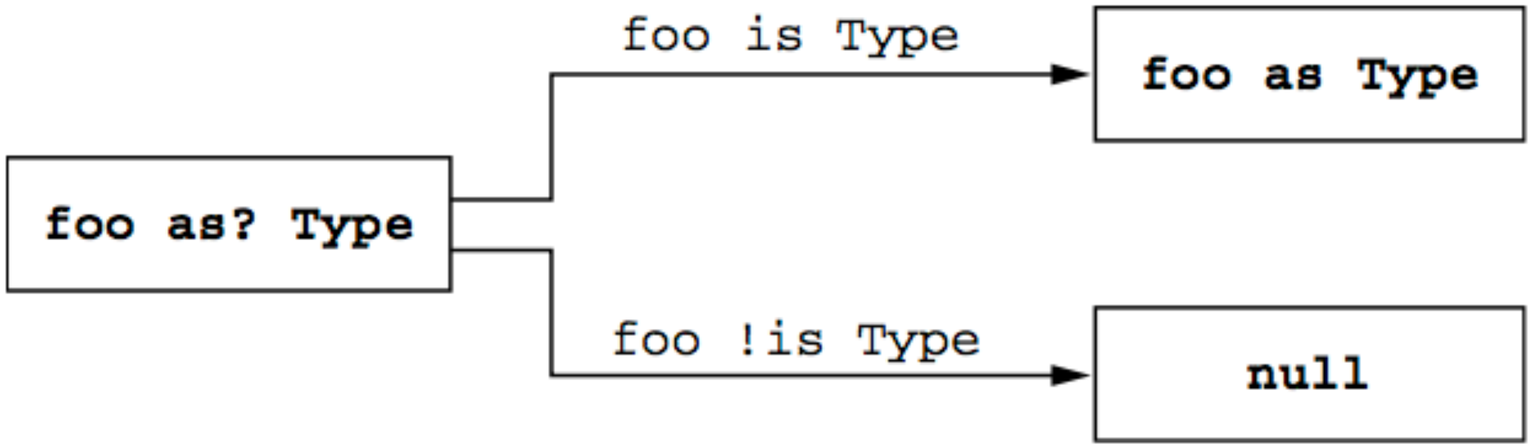


Elvis operator “?:”



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

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


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

Casts seguros: “as?”



Casts seguros: “as?”

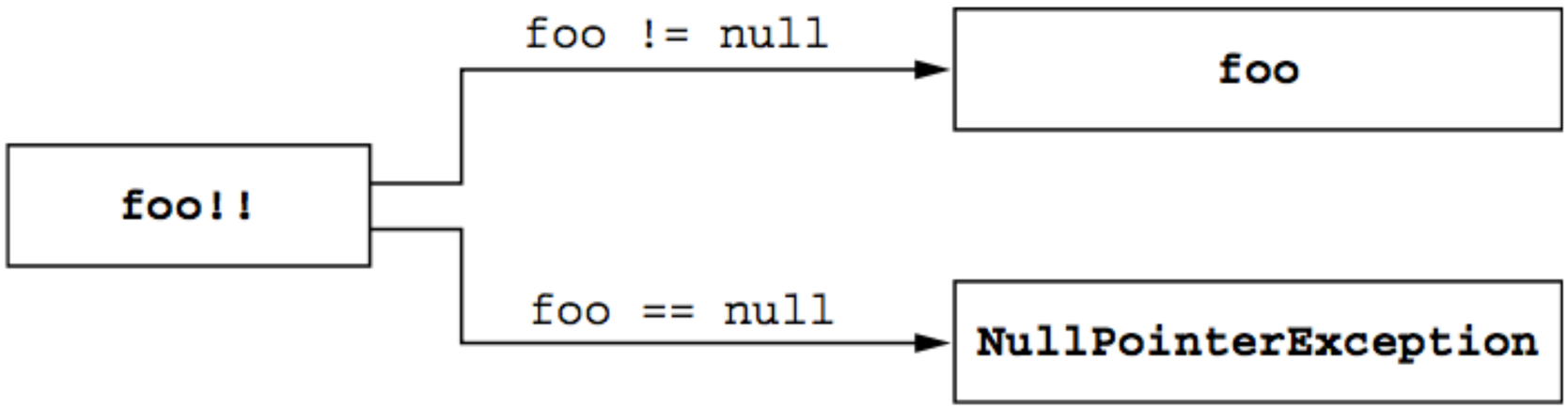
```
class Person(val firstName: String, val lastName: String) {
    override fun equals(o: Any?): Boolean {
        val otherPerson = o as? Person ?: return false

        return otherPerson.firstName == firstName &&
                otherPerson.lastName == lastName
    }

    override fun hashCode(): Int =
        firstName.hashCode() * 37 + lastName.hashCode()
}

>>> val p1 = Person("Dmitry", "Jemerov")
>>> val p2 = Person("Dmitry", "Jemerov")
>>> println(p1 == p2)
true
>>> println(p1.equals(42))
false
```

Checks the type and returns false if no match

After the safe cast, the variable otherPerson is smart-cast to the Person type.

The == operator calls the “equals” method.

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”.

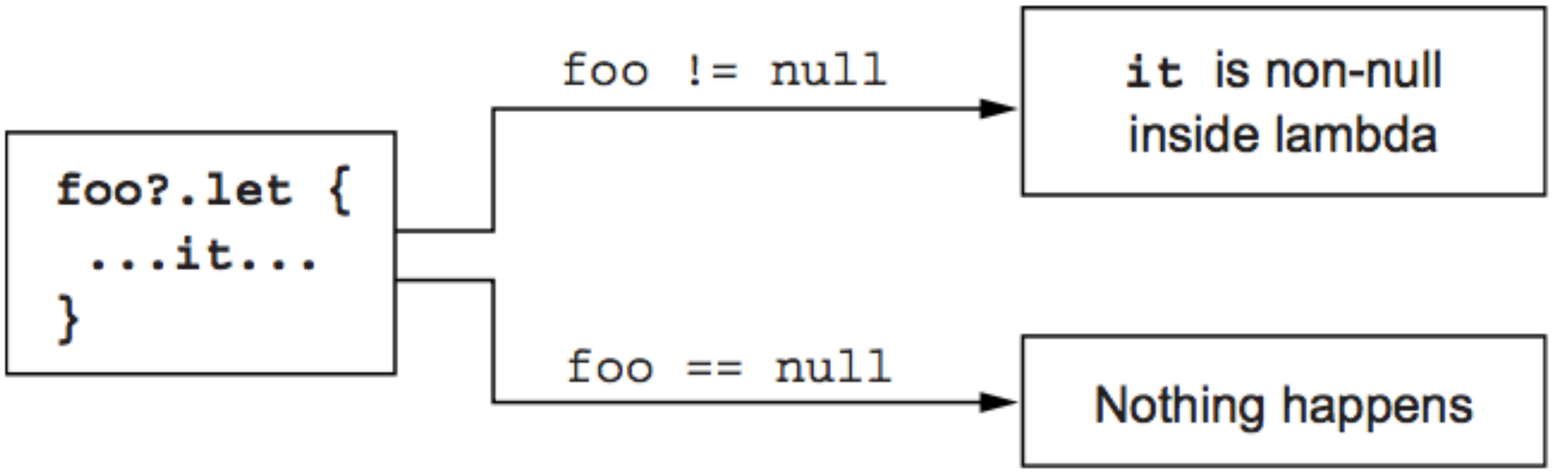

let

```
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


foo?.let {
...it...
}
foo != null
it is non-null
inside lambda
foo == null
Nothing happens


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() { ... }
```

Explicit Unit declaration is omitted

“Unit”: el “void” de Kotlin

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

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

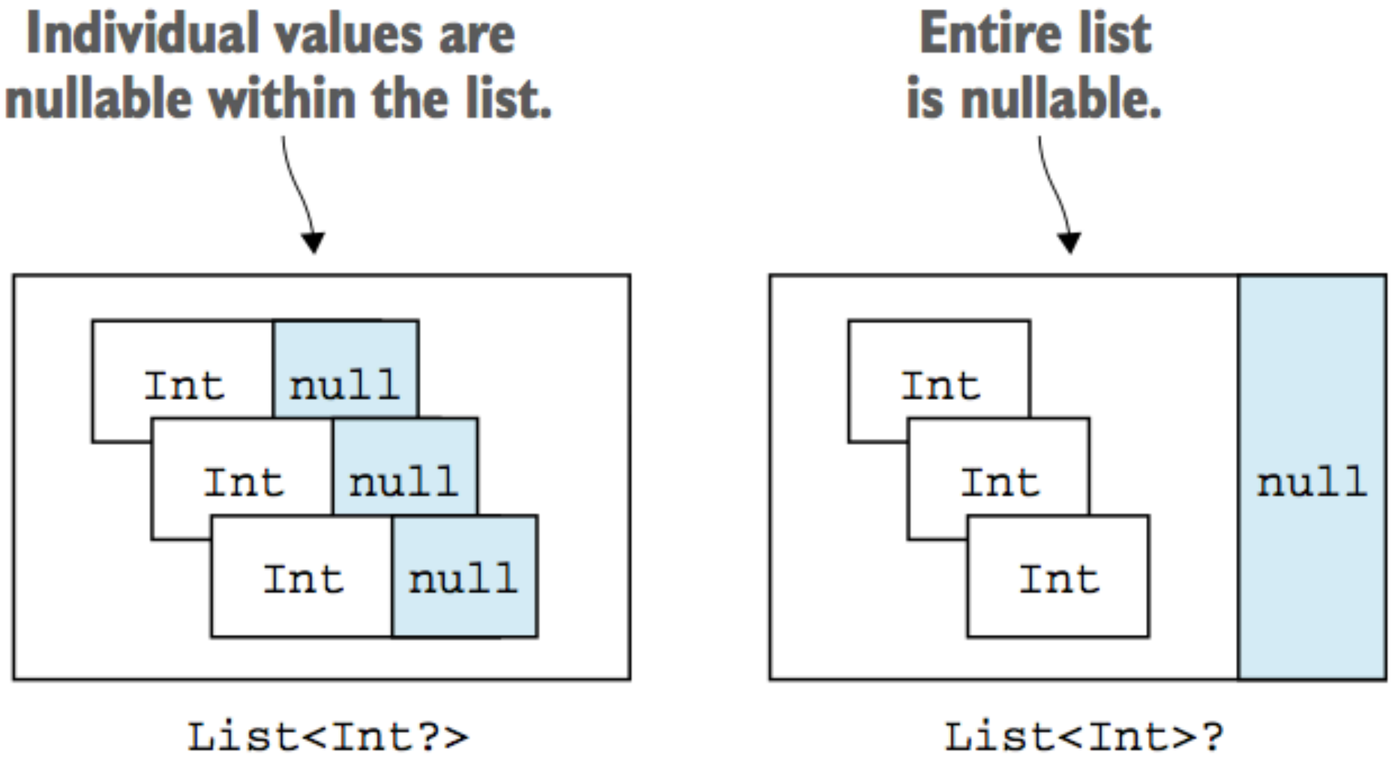
Returns Unit, but you omit the type specification

You don’t need an explicit return here.


Tipo Nada

```
fun fail(message: String): Nothing {
    throw IllegalStateException(message)
}
```

Nullability and collections



Read-only and mutable collections