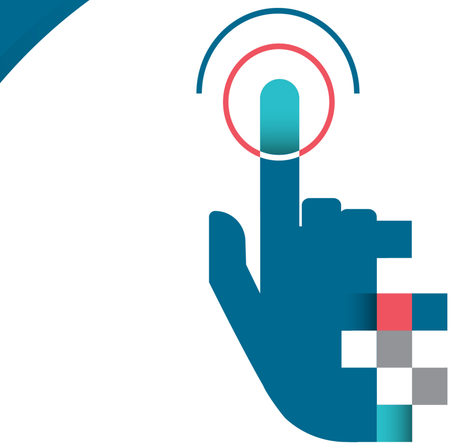
2019

1° ESCUELA DE ACTUALIZACIÓN EN TECNOLOGÍAS DE INFORMÁTICA

PROGRAMACIÓN EN KOTLIN PARA PLATAFORMAS JAVA Y ANDROID

LIC. EMMANUEL LAGARRIGUE LAZARTE

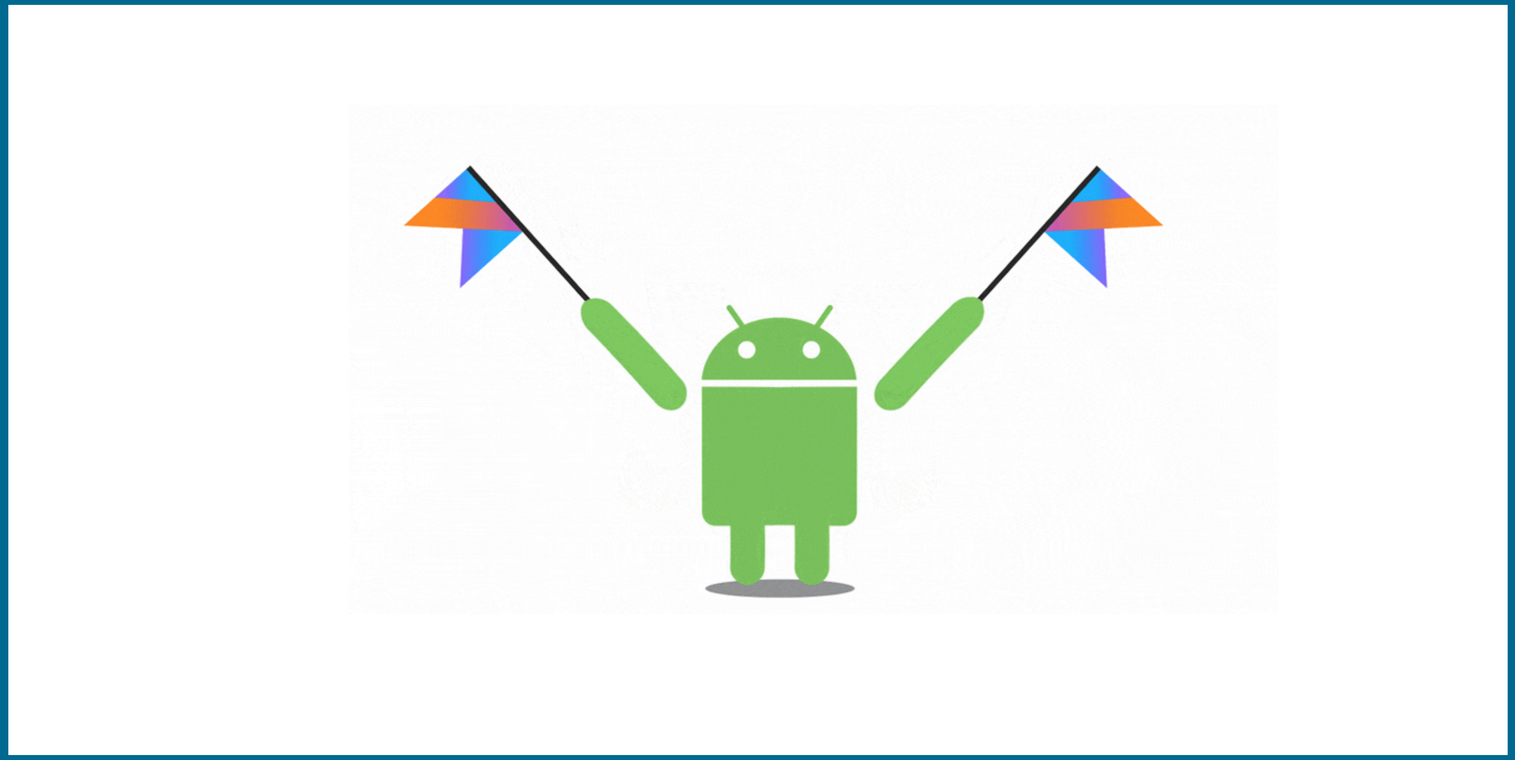


eati

DCIC DEPARTAMENTO DE CIENCIAS E INGENIERÍA DE LA COMPUTACIÓN

UNS UNIVERSIDAD NACIONAL DEL SUR

Funciones de Kotlin



Creando colecciones

```
val set = hashSetOf(1, 7, 53)

val list = arrayListOf(1, 7, 53)
val map = hashMapOf(1 to "one", 7 to "seven", 53 to "fifty-three")
```

Creando colecciones

```
>>> println(set.javaClass)
class java.util.HashSet

>>> println(list.javaClass)
class java.util.ArrayList

>>> println(map.javaClass)
class java.util.HashMap
```

javaClass is Kotlin's equivalent of Java's getClass().

Creando colecciones

```
>>> val strings = listOf("first", "second", "fourteenth")

>>> println(strings.last())
fourteenth

>>> val numbers = setOf(1, 14, 2)

>>> println(numbers.max())
14
```

Llamando funciones

```
fun <T> joinToString(
        collection: Collection<T>,
        separator: String,
        prefix: String,
        postfix: String
): String {

    val result = StringBuilder(prefix)

    for ((index, element) in collection.withIndex()) {
        if (index > 0) result.append(separator)
        result.append(element)
    }

    result.append(postfix)
    return result.toString()
}
```

Don't append a separator before the first element.

```
>>> val list = listOf(1, 2, 3)
>>> println(joinToString(list, "; ", "(", ")"))
(1; 2; 3)
```


Argumentos con nombre

```
joinToString(collection, " ", " ", ".")
```

```
/* Java */
joinToString(collection, /* separator */ " ",  /* prefix */ " ",
    /* postfix */ ".");
```

```
joinToString(collection, separator = " ", prefix = " ", postfix = ".")
```

Parámetros con valores por defecto

```
fun <T> joinToString(
        collection: Collection<T>,
        separator: String = ", ",
        prefix: String = "",
        postfix: String = ""
): String
```

Parameters with default values

Parámetros con valores por defecto

```
>>> joinToString(list, ", ", "", "")
1, 2, 3
>>> joinToString(list)
1, 2, 3
>>> joinToString(list, "; ")
1; 2; 3
```

Parámetros con valores por defecto

```
>>> joinToString(list, suffix = ";", prefix = "# ")
# 1, 2, 3;
```


Funciones Top-Level

```
package strings

fun joinToString(...): String { ... }
```

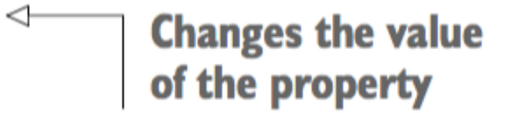
Funciones Top-Level

```
/* Java */
package strings;

public class JoinKt {
    public static String joinToString(...) { ... }
}
```

```
/* Java */
import strings.JoinKt;

...

JoinKt.joinToString(list, ", ", "", "");
```

Propiedades Top-Level

```
var opCount = 0

fun performOperation() {
    opCount++
    // ...
}
```




Propiedades Top-Level

```
const val UNIX_LINE_SEPARATOR = "\n"
```

This gets you the equivalent of the following Java code:

```
/* Java */
public static final String  UNIX_LINE_SEPARATOR = "\n";
```

Funciones de extensión


```
>>> println("Kotlin".lastChar())
n
```

Funciones de extensión

```
package strings

fun String.lastChar(): Char = get(length - 1)
```


Imports y Funciones de extensión

```
import strings.lastChar

val c = "Kotlin".lastChar()


import strings.lastChar as last

val c = "Kotlin".last()
```

Llamando a las Funciones de extensión desde java

```
/* Java */
char c = StringUtilKt.lastChar("Java");
```

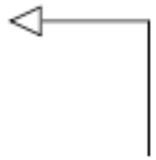
Colecciones: varags, infix calls, library support

Varargs: Funciones con cantidad arbitraria de argumentos

```
val list = listOf(2, 3, 5, 7, 11)
```

```
fun listOf<T>(vararg values: T): List<T> { ... }
```

```
fun main(args: Array<String>) {
    val list = listOf("args: ", *args)
    println(list)
}
```



Spread operator unpacks the array contents

Pares: infix calls y des-estructurando declaraciones

```
val map = mapOf(1 to "one", 7 to "seven", 53 to "fifty-three")
```

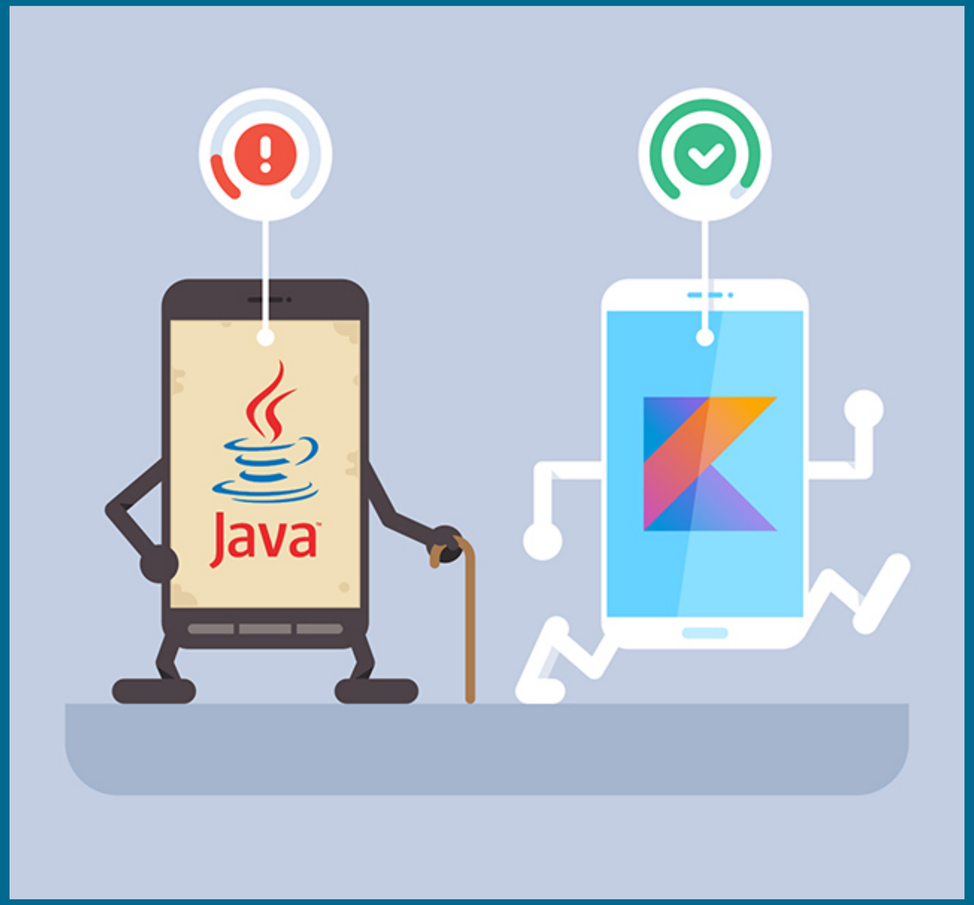
```
1.to("one")
1 to "one"
```

Calls the “to” function the regular way

Calls the “to” function using an infix notation

```
infix fun Any.to(other: Any) = Pair(this, other)
```

Clases, Objetos e Interfaces



Definiendo jerarquías de clases


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

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

Regular method declaration

Method with a default implementation


Interfaces en Kotlin

```
interface Focusable {
    fun setFocus(b: Boolean) =
        println("I ${if (b) "got" else "lost"} focus.")

    fun showOff() = println("I'm focusable!")
}
```

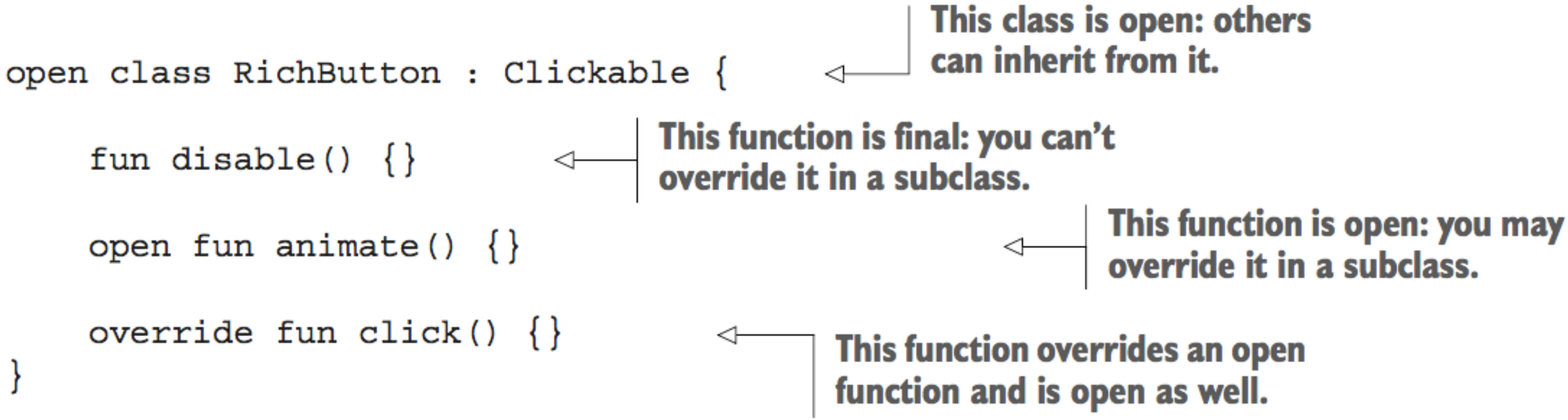
Interfaces en Kotlin

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

    override fun showOff() {
        super<Clickable>.showOff()
        super<Focusable>.showOff()
    }
}
```

You must provide an explicit implementation if more than one implementation for the same member is inherited.

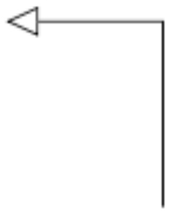
"super" qualified by the supertype name in angle brackets specifies the parent whose method you want to call.

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



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

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

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

Inner and nested classes

- Al igual que en Java, en Kotlin se pueden declarar clases dentro de una clase. La diferencia es q las clases anidadas no tienen acceso a la clase contenedora
- Una clase anidada en Kotlin sin modificadores explícitos es lo mismo que una clase anidada **static** en Java. Para que sea una clase anidada que contenga una referencia a la clase contenedora se debe usar el modificador **inner**.

Inner and nested classes

Class A declared within another class B	In Java	In Kotlin
Nested class (doesn't store a reference to an outer class)	static class A	class A
Inner class (stores a reference to an outer class)	class A	inner class A


Inner and nested classes

```
class Outer {
    inner class Inner {
        fun getOuterReference(): Outer = this@Outer
    }
}
```

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.

Constructores

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() {}
```

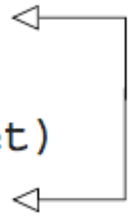
This class has a private constructor.

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) {
        // ...
    }
}
```



Calling superclass constructors

Constructores secundarios

```
class MyButton : View {
    constructor(ctx: Context): this(ctx, MY_STYLE) {
        // ...
    }

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

Delegates to another constructor of the class


Implementado interfaces

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

Implementado interfaces

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

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

class FacebookUser(val accountId: Int) : User {
    override val nickname = getFacebookName(accountId)
}

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

Primary constructor property

Custom getter

Property initializer

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

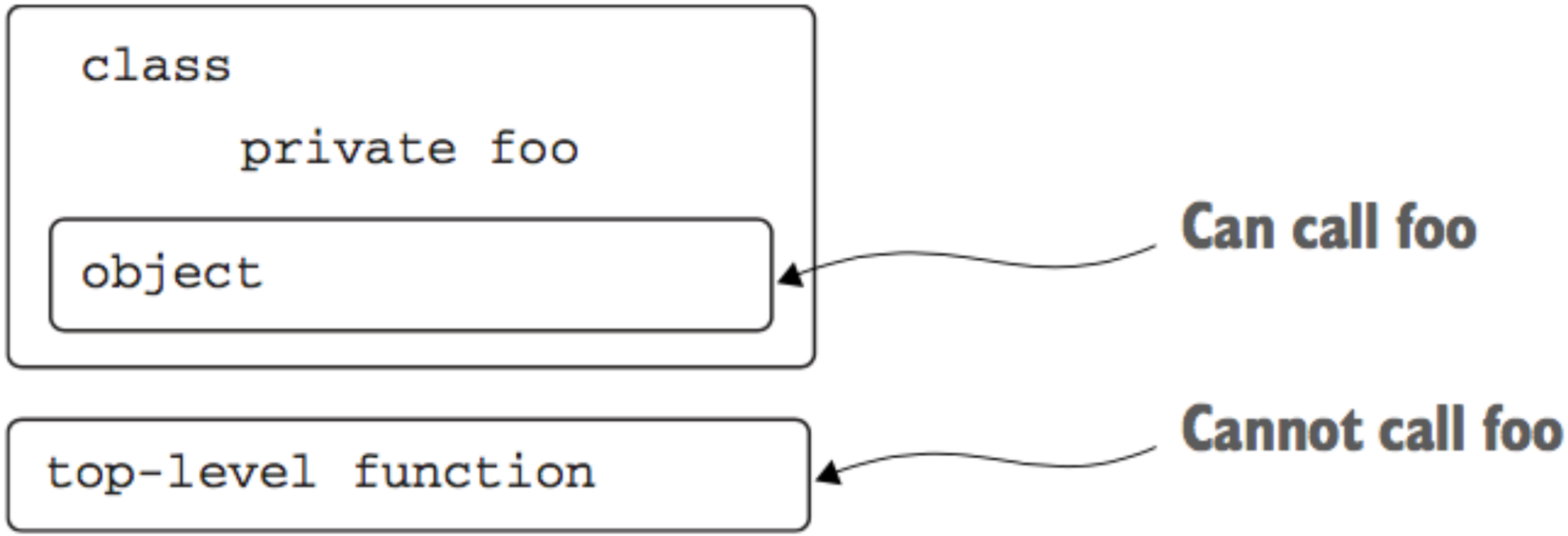
```
data class Person(val name: String) {
    object NameComparator : Comparator<Person> {
        override fun compare(p1: Person, p2: Person): Int =
            p1.name.compareTo(p2.name)
} }
```

```
>>> 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);
```

Companion Objects




Companion Objects

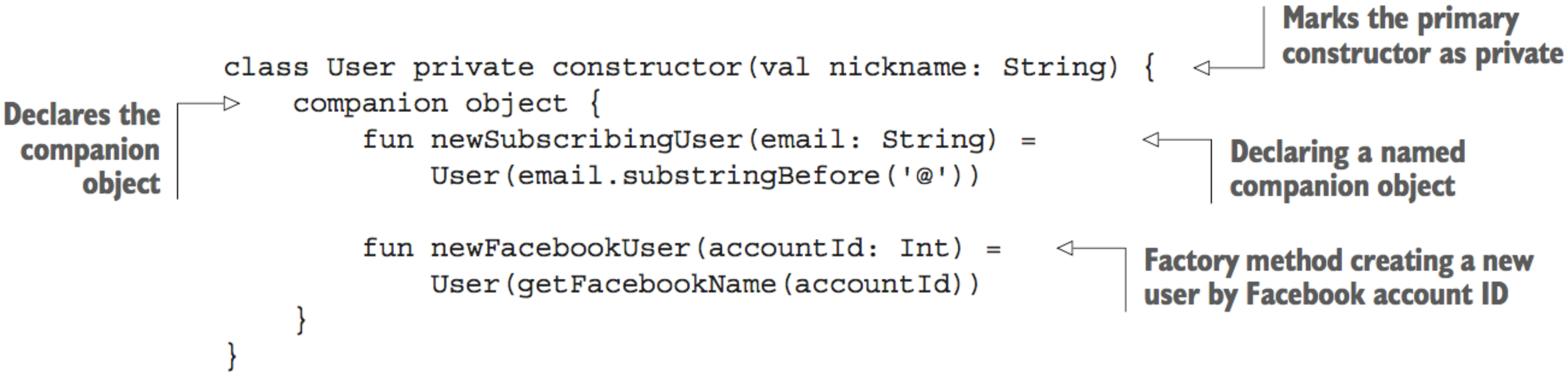
```
class A {
    companion object {
        fun bar() {
            println("Companion object called")
        }
    }
}

>>> A.bar()
Companion object called
```

Companion Objects

```
class User {
    val nickname: String

    constructor(email: String) {
        nickname = email.substringBefore('@')
    }

    constructor(facebookAccountId: Int) {
        nickname = getFacebookName(facebookAccountId)
    }
}
```

Secondary constructors

Companion Objects




```
>>> val subscribingUser = User.newSubscribingUser("bob@gmail.com")
>>> val facebookUser = User.newFacebookUser(4)

>>> println(subscribingUser.nickname)
bob
```