v1.1.2

Clases y objetos

Kotlin

Paquetes y archivos

Paquetes

```
package foo.bar

fun baz() {}

class Goo {}

// ...
```

- La primera línea declara el paquete `foo.bar`
- Es solo un espacio de nombres, no tiene que coincidir con carpetas o archivos
- Si no se especifica ningún paquete, asume uno anónimo por defecto

Archivos

- No es obligatorio tener una clase por archivo
- Tampoco tiene que coincidir el nombre de la clase con el del archivo

Clases

Declaración

```
class Invoice {
}
```

Constructores

- Una clase en Kotlin puede tener un constructor primario y uno o más constructores secundarios
- El constructor primario es parte de la cabecera de la declaración de la clase
- No tiene cuerpo, usa un bloque de inicialización marcado con `init`

Constructor primario

```
class Person constructor(firstName: String) {
}
```

```
class Person(firstName: String) {
}
```

Acceso a los parámetros del constructor

```
class Customer(name: String) {
    init {
        logger.info("Customer initialized with value ${name}")
    }
}
```

```
class Customer(name: String) {
    val customerKey = name.toUpperCase()
}
```

Sintaxis abreviada

```
class Person(firstName: String, lastName: String, age: Int) {

    val nombre = firstName
    val apellidos = lastName
    var edad = age

}
```

```
class Person(val firstName: String, val lastName: String, var age: Int) {
    // ...
}
```

Modificadores

```
class Customer public @Inject constructor(name: String) { ... }
```

Constructor secundario (sin primario)

```
class Person {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

Constructor secundario (con primario)

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Constructor por defecto

- Si una clase no abstracta no declara constructor primario ni secundarios, tendrá un constructor autogenerado y sin parámetros

Instancia de una clase

```
val invoice = Invoice()

val customer = Customer("Joe Smith")
```

Miembros de una clase

- Constructores y bloques de inicialización
- Funciones
- Propiedades
- Clases anidadas e internas
- Declaraciones de objetos

Herencia

Herencia

- Todas las clases tienen un ancestro común llamado `Any`
- `Any` no equivale a `java.lang.Object`

Herencia

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

- Si hay constructor primario, usaremos sus parámetros para llamar al constructor de la clase base
- Si no lo hay, cada constructor secundario tiene que llamar a un constructor de la clase mediante `super` o delegar en otro constructor que lo haga

Herencia

```
class MyView : View {
    constructor(ctx: Context) : super(ctx)

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
}
```

Open

- La anotación `open` hace lo contrario que `final` en Java
- Por defecto las clases en Kotlin son finales, no se puede heredar de ellas salvo que las marquemos con `open`

Redefinir métodos

```
open class Base {
    open fun v() {}
    fun nv() {}
}

class Derived() : Base() {
    override fun v() {}
}
```

- Para poder redefinir un método, tiene que estar marcado como `open`

Redefinir métodos

```
open class AnotherDerived() : Base() {
    final override fun v() {}
}
```

- Un método marcado con `override` ya es `open`
- Para prohibir más modificaciones, lo marcamos con `final`

Redefinir propiedades

```
open class Foo {
    open val x: Int get { ... }
}

class Bar1 : Foo() {
    override val x: Int = ...
}
```

- Se permite redefinir una propiedad `val` con una `var`, pero no al revés (una propiedad `val` no proporciona setter)

Conflictos

```
open class A {
    open fun f() { print("A") }
    fun a() { print("a") }
}

interface B {
    fun f() { print("B") } // interface members
                          //are 'open' by default
    fun b() { print("b") }
}

class C() : A(), B {
    // The compiler requires f() to be overridden:
    override fun f() {
        super<A>.f() // call to A.f()
        super<B>.f() // call to B.f()
    }
}
```

- Si heredamos múltiples implementaciones, tenemos que proporcionar una propia seleccionando el tipo base al que llamamos usando `super<Base>`

Clases abstractas

```
open class Base {
    open fun f() {}
}

abstract class Derived : Base()
{
    override abstract fun f()
}
```

- No hace falta poner **open**

Propiedades

Declaración

```
class Address {
    var name: String = ...
    var street: String = ...
    var city: String = ...
    var state: String? = ...
    var zip: String = ...
}
```

Uso

```
fun copyAddress(address: Address): Address {
    val result = Address() // there's no 'new' keyword in Kotlin
    result.name = address.name // accessors are called
    result.street = address.street
    // ...
    return result
}
```

Sintaxis

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]
    [<getter>]
    [<setter>]
```

- La inicialización, el getter y el setter son opcionales
- El tipo es opcional si se puede deducir del inicializador o del tipo de retorno del getter

Propiedades

```
var initialized = 1 // has type Int, default getter and setter

val simple: Int? // has type Int, default getter, must be initialized
                 // in constructor
val inferredType = 1 // has type Int and a default getter

val isEmpty get() = this.size == 0  // has type Boolean
```

Métodos de acceso personalizados

```
val isEmpty: Boolean
    get() = this.size == 0


var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value)
            // parses the string and
            // assigns values to other
            // properties
    }
```

- Por convenio, el nombre del parámetro del setter es `value`, pero se puede modificar

Interfaces

Declaración

```
interface MyInterface {
    fun bar()
    fun foo() {
        // optional body
    }
}
```

- Un iterfaz no almacena estado
- Pueden tener propiedades pero han de ser abstractas o tener implementados los métodos de acceso

Implementación

```
class Child : MyInterface {
    override fun bar() {
        // body
    }
}
```

- Una clase u objeto puede implementar múltiples interfaces
- Se indican separados por comas

Propiedades

```
interface MyInterface {
    val prop: Int // abstract

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(prop)
    }
}

class Child : MyInterface {
    override val prop: Int = 29
}
```

Conflictos

```
interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }

    override fun bar() {
        super<B>.bar()
    }
}
```

- Si heredamos múltiples implementaciones, tenemos que proporcionar una propia seleccionando el tipo base al que llamamos usando `super<Base>`

Visibilidad

Modificadores

- private
- protected
- internal
- public

Paquetes

```kotlin
// file name: example.kt
package foo

fun baz() {}
class Bar {}
```

- Sin modificador, asume `public`, visible en todas partes
- `private`, solo visible en el archivo que lo contiene
- `internal`, visible a nivel de módulo
- `protected` no se puede usar a nivel de paquete

Paquetes

```kotlin
// file name: example.kt
package foo

private fun foo() {} // visible inside example.kt

public var bar: Int = 5 // property is visible everywhere
    private set         // setter is visible only in example.kt

internal val baz = 6    // visible inside the same module
```

Clases e interfaces

- `private`, visible sólo dentro de la clase
- `protected`, como `private` pero visible también a las subclases
- `internal`, cualquier cliente dentro del mismo módulo que vea la clase, ve sus miembros marcados como `internal`
- `public`, quien vea la clase que los declara, verá sus miembros marcados con `public`

Módulos

- Es un grupo de archivos de Kotlin que se compilan juntos, por ejemplo:
  - Un módulo de Intellij IDEA
  - Un proyecto de Maven o Gradle
  - Un conjunto de archivos compilados en una tarea de Ant

Clases de datos

Declaración

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

Clases de datos

- El compilador crea automáticamente:
  - El par `equals()` y `hashCode()`
  - Un `toString()` tipo `"User(name=John, age=42)"`
  - Funciones `componentN()` para cada propiedad, en orden
  - Función `copy()`

Función copy()

```
val jack = User(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

- Permite copiar los datos de un objeto a otro, modificando alguna propiedad y manteniendo el resto

Uso de *destructuring declarations*

```
val jane = User("Jane", 35)
val (name, age) = jane

println("$name, $age years of age") // prints "Jane, 35 years of age"
```

Genéricos

Declaración

```
class Box<T>(t: T) {
    var value = t
}
```

Uso

```
val box: Box<Int> = Box<Int>(1)


val anotherBox = Box(1) // 1 has type Int, so the compiler
                        // figures out that we are talking about Box<Int>
```

Enumeraciones

Declaración

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
```

Uso

```
val direccion = Direction.NORTH

when (direccion) {
    Direction.NORTH -> print("Vamos bien")
    Direction.EAST -> print("Un poco desviados")
    Direction.WEST -> print("Where the skies are blue")
    Direction.SOUTH -> print("No, no vamos bien")
}
```

Extensiones

Extensiones

- Permiten añadir nueva funcionalidad a una clase sin tener que heredar

Extensión de funciones

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' es la lista
    this[index1] = this[index2]
    this[index2] = tmp
}
```

Extensión de propiedades

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```