



UD2.3 – Fundamentos de Kotlin

2º CFGS
Desarrollo de Aplicaciones Multiplataforma
2023-24

1.- Programación Orientada a Objetos

Como se indicó al principio de la unidad, Kotlin busca ser conciso y evitar escribir demasiado código.

Esta característica es muy visible cuando se desarrollan clases.

Se va a estudiar tanto la manera más tradicional de codificar clases como en Java y la manera aconsejada por Kotlin.

1.- Programación Orientada a Objetos

Para crear clases se usa la palabra reservada **class** como en Java.

Kotlin permite que en un mismo archivo **.kt** se definan más de una clase públicas (esto no se permite en Java).

Si en un archivo **.kt** solo se define una clase, entonces el nombre del archivo debe ser el nombre de la clase.

Si en un archivo **.kt** se definen varias clases, entonces se debe buscar un nombre representativo.

1.- Programación Orientada a Objetos

En Kotlin todas las clases hereda de la clase **Any** (en Java es Object).

La clase **Any** define tres métodos que heredarán sus hijas:

equals → indica si un objeto es igual a otro

hashCode → devuelve el código hash de un objeto

toString → devuelve la representación en String de un objeto

1.- Programación Orientada a Objetos

Ejemplo de definición de clase en Kotlin:

```
class Person {  
}
```

Kotlin es tan conciso que una clase sin cuerpo se puede definir así:

```
class Person
```

Aun sin cuerpo se puede instanciar un objeto de esta clase:

```
val morty = Person()
```

1.- Programación Orientada a Objetos

Se puede observar que la estructura de definición de una clase es similar a la usada en Java.

Los **getters** y **setters** se deben definir tras cada propiedad.

El constructor se define con la palabra **constructor**.

Su uso también es similar a Java:

```
val ps5 = Product( name: "PS5", price: 499.99)
ps5.name = "Play Statation 5"
println(ps5.name)
```

```
class Product {
    var name: String
        get() {
            return field
        }
        set(name: String) {
            field = name
        }
    var price: Double
        get() {
            return field
        }
        set(price: Double) {
            field = price
        }

    constructor(name: String, price: Double) {
        this.name = name
        this.price = price
    }
}
```

1.- Programación Orientada a Objetos

Si no se indica lo contrario las clases, propiedades y métodos **por defecto son públicas** (public).

Existen los modificadores de visibilidad:

public

private

protected

internal → visible en el mismo módulo (paquete)

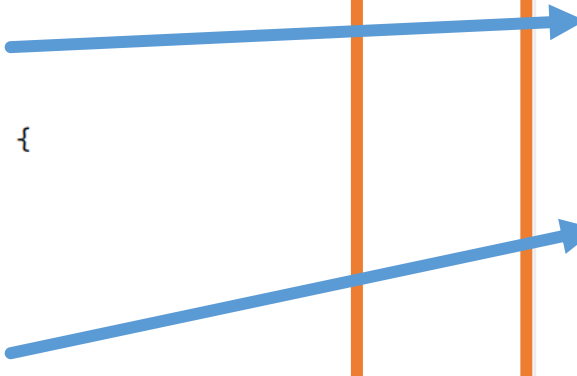
```
class Product {  
    var name: String  
        get() {  
            return field  
        }  
        set(name: String) {  
            field = name  
        }  
    var price: Double  
        get() {  
            return field  
        }  
        set(price: Double) {  
            field = price  
        }  
  
    constructor(name: String, price: Double) {  
        this.name = name  
        this.price = price  
    }  
}
```

1.- Programación Orientada a Objetos

Kotlin permite ahorrar código usando funciones de expresión (en una línea):

```
class Product {  
    var name: String  
    get() {  
        return field  
    }  
    set(name: String) {  
        field = name  
    }  
    var price: Double  
    get() {  
        return field  
    }  
    set(price: Double) {  
        field = price  
    }  
  
    constructor(name: String, price: Double) {  
        this.name = name  
        this.price = price  
    }  
}
```

```
class Product {  
    var name: String  
    get() = field  
    set(name: String) {  
        field = name  
    }  
    var price: Double  
    get() = field  
    set(price: Double) {  
        field = price  
    }  
  
    constructor(name: String, price: Double) {  
        this.name = name  
        this.price = price  
    }  
}
```

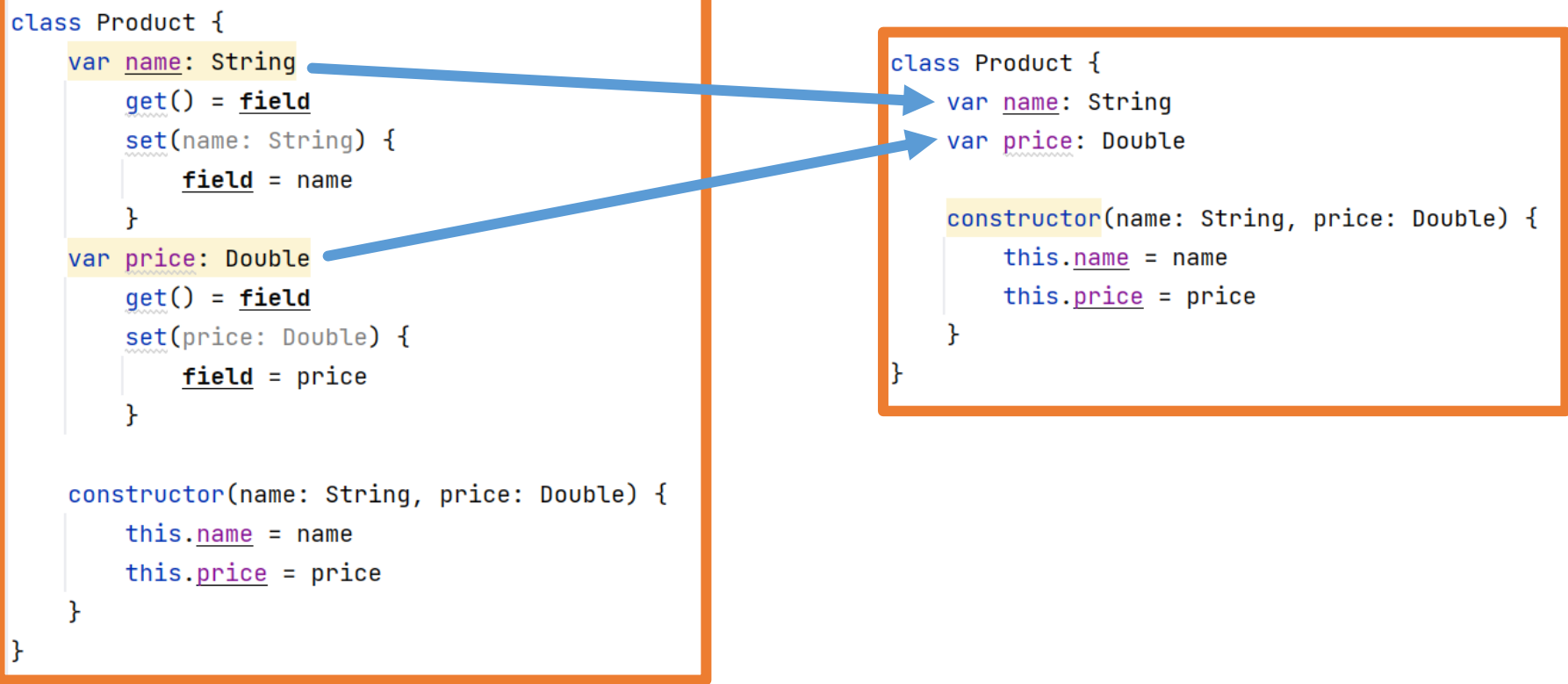


1.- Programación Orientada a Objetos

Kotlin permite ahorrar código ya que los getters y setters son implícitos:

```
class Product {  
    var name: String  
        get() = field  
        set(name: String) {  
            field = name  
        }  
    var price: Double  
        get() = field  
        set(price: Double) {  
            field = price  
        }  
  
    constructor(name: String, price: Double) {  
        this.name = name  
        this.price = price  
    }  
}
```

```
class Product {  
    var name: String  
    var price: Double  
  
    constructor(name: String, price: Double) {  
        this.name = name  
        this.price = price  
    }  
}
```

The diagram illustrates the simplification of Kotlin code by removing explicit getters and setters. Two blue arrows originate from the left box: one points from the 'name' property's getter and setter block to the 'name' property declaration in the right box, and the other points from the 'price' property's getter and setter block to the 'price' property declaration in the right box. This visualizes how Kotlin's implicit getters and setters allow for more concise code.

1.- Programación Orientada a Objetos

Kotlin permite ahorrar código incorporando las propiedades en la definición de la clase haciendo que el **constructor sea implícito**:

```
class Product {  
    var name: String  
    var price: Double  
  
    constructor(name: String, price: Double) {  
        this.name = name  
        this.price = price  
    }  
}
```



```
class Product(var name: String, var price: Double)
```

1.- Programación Orientada a Objetos

Kotlin permite ahorrar código:

```
class Product {  
    var name: String  
    get() {  
        return field  
    }  
    set(name: String) {  
        field = name  
    }  
    var price: Double  
    get() {  
        return field  
    }  
    set(price: Double) {  
        field = price  
    }  
  
    constructor(name: String, price: Double) {  
        this.name = name  
        this.price = price  
    }  
}
```



```
class Product(var name: String, var price: Double)
```

1.- Programación Orientada a Objetos

Con esta estructura se tiene el constructor, los getters y los setters

```
class Product(var name: String, var price: Double)
```

Si no se van a realizar acciones específicas en el constructor, los getters o los setters, esta sintaxis permite ahorrar mucho código.

Utilizando esta sintaxis este constructor se denomina **constructor primario**.

1.- Programación Orientada a Objetos

Los constructores que se incluyen en el cuerpo de la clase se denominan **constructores secundarios**.

Los constructores se declaran con la palabra reservada **constructor**.

Existe la sobrecarga de métodos como se puede ver con el constructor.

Si una propiedad no aparece en algún constructor deberá tener un valor asignado por defecto.

```
val game = Product(name: "GTA VI", price: 70.0)
val controller = Product(name: "Dual Sense")
```

```
class Product {
    var name: String
    var price: Double = -1.0

    constructor(name: String, price: Double) {
        this.name = name
        this.price = price
    }

    constructor(name: String) {
        this.name = name
    }

    // constructor(name: String) : this(name, -1.0)
}
```

```
class Product(var name: String, var price: Double) {
    constructor(name: String) : this(name, price: -1.0)
}
```

```
class Product(var name: String) {
    var price: Double = -1.0

    constructor(name: String, price: Double) : this(name) {
        this.price = price
    }
}
```

1.- Programación Orientada a Objetos

Si se usan combinados el constructor primario con constructores secundarios, se debe usar la palabra **this** en los constructores secundarios.

Mediante **this** se llama al constructor primario (equivalente a super en Java).

```
class Product {  
    var name: String  
    var price: Double = -1.0  
  
    constructor(name: String, price: Double) {  
        this.name = name  
        this.price = price  
    }  
  
    constructor(name: String) {  
        this.name = name  
    }  
  
    //    constructor(name: String) : this(name, -1.0)  
}
```

```
class Product(var name: String, var price: Double) {  
    constructor(name: String) : this(name, price: -1.0)  
}
```

```
class Product(var name: String) {  
    var price: Double = -1.0  
  
    constructor(name: String, price: Double) : this(name) {  
        this.price = price  
    }  
}
```

1.- Programación Orientada a Objetos

Puede no haber constructores.

En ese caso en la declaración de las propiedades se debe asignar un valor.

```
class Product {  
    var name: String = ""  
    var price: Double = -1.0  
}
```

```
val game = Product()  
game.name = "GTA VI"  
println("Se vende ${game.name}")
```

1.- Programación Orientada a Objetos

Si se usa el constructor primario pero se requiere realizar acciones sobre las propiedades se debe usar un bloque de instrucciones **init**.

```
class Product(var name: String, var price: Double) {  
    init {  
        name = name.uppercase()  
    }  
}
```

```
val product = Product( name: "Nintendo Switch", price: 350.0)  
println("Se vende ${product.name}")  
// Mostrará: Se vende NINTENDO SWITH
```


1.- Programación Orientada a Objetos

Se pueden definir funciones miembro (métodos) en las clases.

```
class Product(var name: String, var price: Double) {  
    fun finalPrice(percentage: Double) = price - price * percentage / 100  
}
```

La misma función definida de manera extendida:

```
class Product(var name: String, var price: Double) {  
    fun finalPrice(percentage: Double): Double {  
        return price - price * percentage / 100  
    }  
}
```

1.- Programación Orientada a Objetos

Si se sobrescribe una función heredada (ya existe en la clase madre), se debe añadir la palabra **override** en la definición de la función.

```
class Product(var name: String, var price: Double) {  
    override fun toString(): String {  
        return "$name: $price €"  
    }  
}
```

1.- Programación Orientada a Objetos

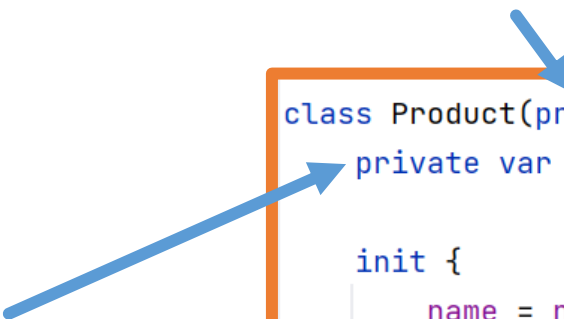
Evidentemente se puede combinar todo lo anterior:

- Constructor primario
- init para realizar acciones en el constructor primario
- Constructor secundario
- getters y setters implícitos
- Sobrecarga de funciones heredadas

```
class Product(var name: String) {  
    var price: Double = -1.0  
  
    init {  
        name = name.uppercase()  
    }  
  
    constructor(name: String, price: Double): this(name) {  
        this.price = price  
    }  
  
    override fun toString(): String {  
        return "$name: $price €"  
    }  
}
```

1.- Programación Orientada a Objetos

Si se quiere cambiar el tipo de acceso de las propiedades de la clase simplemente se debe indicar delante de su declaración.



```
class Product(private var name: String) {  
    private var price: Double = -1.0  
  
    init {  
        name = name.uppercase()  
    }  
  
    constructor(name: String, price: Double): this(name) {  
        this.price = price  
    }  
  
    override fun toString(): String {  
        return "$name: $price €"  
    }  
}
```


1.- Programación Orientada a Objetos

Al ser implícitos, **los getters y los setters tendrán el mismo tipo de acceso que se defina para las propiedades**, y esto **no se puede cambiar**.

Este funcionamiento puede ser un problema ya que si se quiere dotar a las clases de la encapsulación (característica típica de la POO) no se tendrá acceso a las propiedades cuando se usen los objetos de la clase.

```
class Product(private var name: String) {  
    private var price: Double = -1.0  
  
    init {  
        name = name.uppercase()  
    }  
  
    constructor(name: String, price: Double): this(name) {  
        this.price = price  
    }  
  
    override fun toString(): String {  
        return "$name: $price €"  
    }  
}
```

```
val product = Product(name: "Test")  
product.name = "PS5"  
product.price = 495.5
```

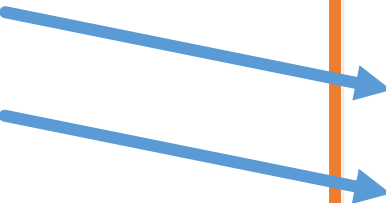


1.- Programación Orientada a Objetos

Para solucionar este problema se pueden crear métodos públicos para acceder a las propiedades privadas.

Aunque esta solución va en contra de la filosofía de Kotlin de ahorrar código.

```
class Product(private var name: String) {  
    private var price: Double = -1.0  
  
    init {  
        name = name.uppercase()  
    }  
  
    constructor(name: String, price: Double): this(name) {  
        this.price = price  
    }  
  
    fun getName(): String {  
        return name  
    }  
  
    fun setName(name: String) {  
        this.name = name  
    }  
  
    override fun toString(): String {  
        return "$name: $price €"  
    }  
}
```



1.- Programación Orientada a Objetos

Si una propiedad es pública se pueden definir el get y el set de manera privada para así proteger el acceso.

En el ejemplo se puede ver que el get seguirá público pero el set no.

Si se necesita cambiar el valor de la propiedad se necesitará definir una función para ello como en la página anterior setName y setPrice.

```
class Product(name: String, price: Double) {  
    var name: String = name  
        private set  
    var price: Double = price  
        private set  
  
    override fun toString(): String {  
        return "$name: $price €"  
    }  
}
```

1.- Programación Orientada a Objetos

Como ya se ha visto anteriormente, una vez declarada una clase su uso es igual que en Java.

El acceso a las propiedades y las funciones se realiza mediante la notación de punto:

```
val product = Product( name: "PS5", price: 495.50)  
println("${product.name}: ${product.finalPrice( average: 10.0)} €")
```


2.- Enum class en Kotlin

Las **enum class** permiten definir un conjunto de constantes relacionadas entre sí.

Es una manera segura de disponer de los valores sin temor a errores.

```
enum class Direction {  
    NORTH, SOUTH, WEST, EAST  
}  
  
enum class Color(val rgb: Int) {  
    RED( rgb: 0xFF0000),  
    GREEN( rgb: 0x00FF00),  
    BLUE( rgb: 0x0000FF)  
}
```

```
val direction = Direction.EAST  
println(direction)           // EAST  
println(direction.name)      // EAST  
println(direction.ordinal)    // 3 (posición)  
  
val color = Color.RED  
println(color)                // RED  
println(color.name)           // RED  
println(color.rgb)            // 16711680 (en decimal)  
println(color.rgb.toHexString()) // 00ff0000 (en hexadecimal)  
println(color.ordinal)        // 0 (posición)
```

3.- Data class en Kotlin

Las **data class** son una manera de modelar los datos y la finalidad de sus objetos es simplemente almacenar datos.

Son clases muy simples que solo contienen atributos.

Se utilizan para interactuar con otras clases o con API's externas.

Para declarar una clase de este tipo se usa la palabra **data**.

```
data class Manga(var name: String, var type: String, var volumes: Int)
```

3.- Data class en Kotlin

Kotlin provee a las data class de una serie de **funciones de utilizad**:

- equals(): permite comparar dos objetos de la clase.
- hashCode(): código hash (este código se usa en la función anterior).
- copy(): permite realizar una copia del objeto.
- toString(): genera un string legible con los datos del objeto.
- component1(), component2()...: get a cada propiedad del objeto en su orden de declaración.

3.- Data class en Kotlin

```
data class Manga(var name: String, var type: String, var volumes: Int)
```

```
val dragonBall = Manga(name: "Dragon Ball", type: "Shonen", volumes: 42)
val deathNote = Manga(name: "Death Note", type: "Shonen", volumes: 13)
val naruto = Manga(name: "Naruto", type: "Shonen", volumes: 72)

println(dragonBall)
println(deathNote)
if (dragonBall.equals(naruto)) println("Son el mismo manga.")
else println("Son mangas diferentes.")
```

Salida:

```
Manga(name=Dragon Ball, type=Shonen, volumes=42)
Manga(name=Death Note, type=Shonen, volumes=13)
Son mangas diferentes.
```

3.- Data class en Kotlin

Las propiedades que no aparezcan en el constructor primario no se beneficiarán de las características de las data class.

Por ejemplo, al usar toString solo aparecerán las propiedades que estén en el constructor primario.

Mediante la función copy se pueden copiar objetos completos y durante la copia se pueden cambiar valores de las propiedades.

```
var db = dragonBall.copy()  
var dbz = dragonBall.copy(nombre="DBZ")
```

3.- Data class en Kotlin

Los objetos de una data class se pueden **deconstruir** de manera que se puede extraer el valor de sus propiedades a variables.

Si hay algún valor que no se quiera se debe poner el carácter `_` en su lugar.

Si los valores que no se quieren están al final de la lista basta con no ponerlos.

```
val dragonBall = Manga( name: "Dragon Ball", type: "Shonen", volumes: 42)
val (name, type, volumes) = dragonBall
println("Manga: $name ($type)")
println("Tomos: $volumes")
```

Salida:

```
Manga: Dragon Ball (Shonen)
Tomos: 42
```

```
val (name) = dragonBall
```

```
val (name, type) = dragonBall
```

```
val (name, _, volumes) = dragonBall
```

4.- Funciones de extensión

Kotlin permite **extender la funcionalidad** de las clases existentes ya sean del sistema o propias sin uso de la herencia incluso si la clase existente es final.

Las funciones de extensión se definen fuera de la definición de la clase por ello **solo se podrán utilizar en el ámbito en el que se definan**.

Aunque la función de extensión **se define fuera de la clase** es como si se hubiera **definido desde dentro** comportándose como un método más de la clase, pudiendo hacer uso tanto de las propiedades y los métodos si los hay aunque sean privados.

En el apartado de las funciones ya se vio un ejemplo:

```
// Se añade una función a la clase Int  
fun Int.isOdd(): Boolean {  
    return this % 2 != 0  
}
```

4.- Funciones de extensión

Estas funciones se definen como otras funciones pero usando el nombre de la clase que se quiere extender.

Función de extensión de una clase existente:

```
fun Int.isOdd(): Boolean {  
    return this % 2 != 0  
}
```



```
fun Int.isOdd() = this % 2 != 0
```

Función de extensión de una clase propia

```
class Product(var name: String, var price: Double)  
  
fun Product.sale(salePrice: Double): Double {  
    return if (price > salePrice) salePrice else price  
}
```


4.- Funciones de extensión

Un uso típico de las funciones de extensión se da con las colecciones.

Gracias a una función de extensión se puede realizar una acción sobre cada uno de los elementos de la colección.

```
fun main() {  
    val numbers = mutableListOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
    println(numbers.addition(quantity: 2))  
}  
  
fun MutableList<Int>.addition(quantity: Int): MutableList<Int> {  
    val numbers = mutableListOf<Int>()  
    for (i in this) numbers.add(i + quantity)  
    return numbers  
}
```

Salida:

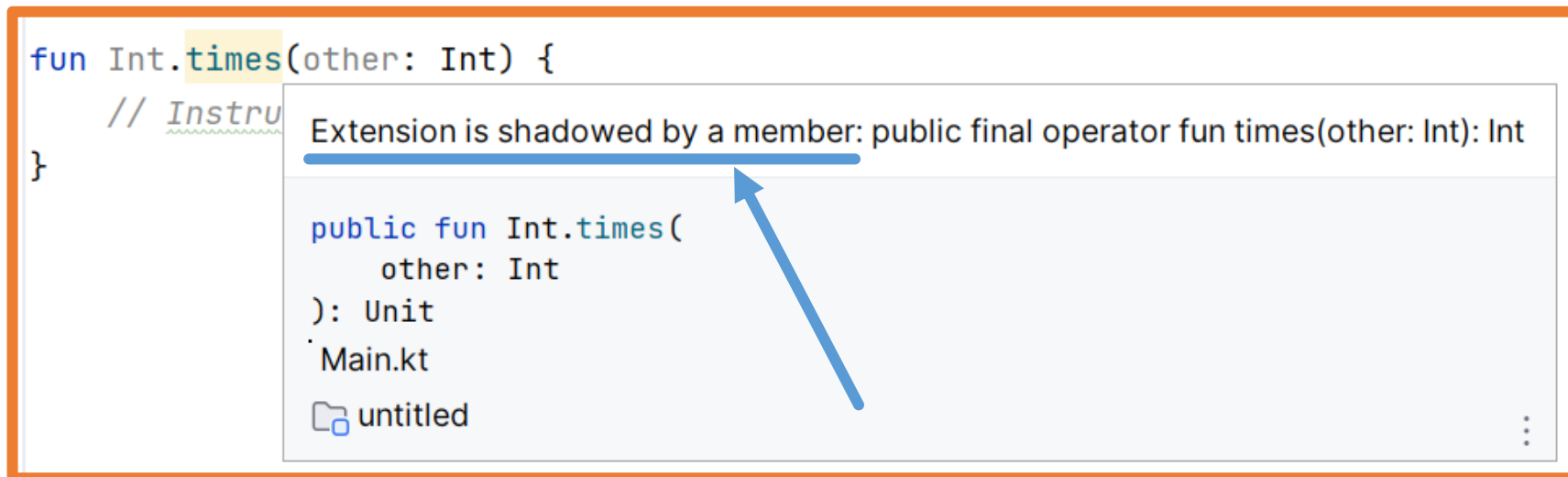
```
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

4.- Funciones de extensión

Una función de extensión **no puede sobre escribir a una función miembro.**

Una función de extensión **sí que puede sobrecargar a una función miembro.**

Aunque no se muestre un error, si se intenta sobrescribir una función miembro mediante una función de extensión, en ejecución siempre se llamará a la función miembro.



```
fun Int.times(other: Int) {  
    // Instru  
}
```

Extension is shadowed by a member: public final operator fun times(other: Int): Int

```
public fun Int.times(  
    other: Int  
): Unit  
Main.kt  
untitled
```

The screenshot shows a code editor with a Kotlin extension function `fun Int.times(other: Int) {` and a comment `// Instru`. A warning message is displayed: `Extension is shadowed by a member: public final operator fun times(other: Int): Int`. Below the warning, the member function `public fun Int.times(other: Int): Unit` is shown. A blue arrow points from the warning message to the member function. The file name `Main.kt` and a folder icon labeled `untitled` are visible at the bottom.

5.- Singleton

En Kotlin se pueden crear **singleton** que son **objetos únicos en su clase**, esto significa que no podrá haber más instancias de esa clase.

```
object Author {  
    var name = "Álex Torres"  
    var company = "2 DAM"  
    var date = Date()  
  
    override fun toString():String {  
        return """"$name ($company)  
                |$date""".trimMargin()  
    }  
}
```

Se le pueden añadir funciones de extensión como con cualquier otra clase.

Si se declaran de manera global en el archivo .kt se podrán añadir funciones de extensión como con cualquier otra clase.

Práctica

Actividad

Clase de película

Actividad

Dados

6.- Herencia

En **Kotlin por defecto todas las clases son finales**, lo cual significa que no puede haber herencia si no se indica explícitamente.

Para indicar que una clase puede ser súper clase se debe usar la palabra reservada **open**.

Si se usa el constructor primario y los getters y setters por defecto:

```
open class Product(var name: String, var price: Double)

class Monitor(name: String, price: Double): Product(name, price)
```

Con este código se tiene:

- La clase Product con el constructor, los getter y los setters
- La clase Monitor con su constructor que llama al constructor de la clase padre, y hereda los getter y los setters de la clase padre.

6.- Herencia

Se pueden añadir propiedades y métodos nuevos.

```
open class Product(var name: String, var price: Double)

class Monitor(name: String, price: Double): Product(name, price) {
    var size: Int = 0

    constructor(name: String, price: Double, size: Int): this(name, price)

    override fun toString(): String {
        return "$name ($size): $price €"
    }
}
```

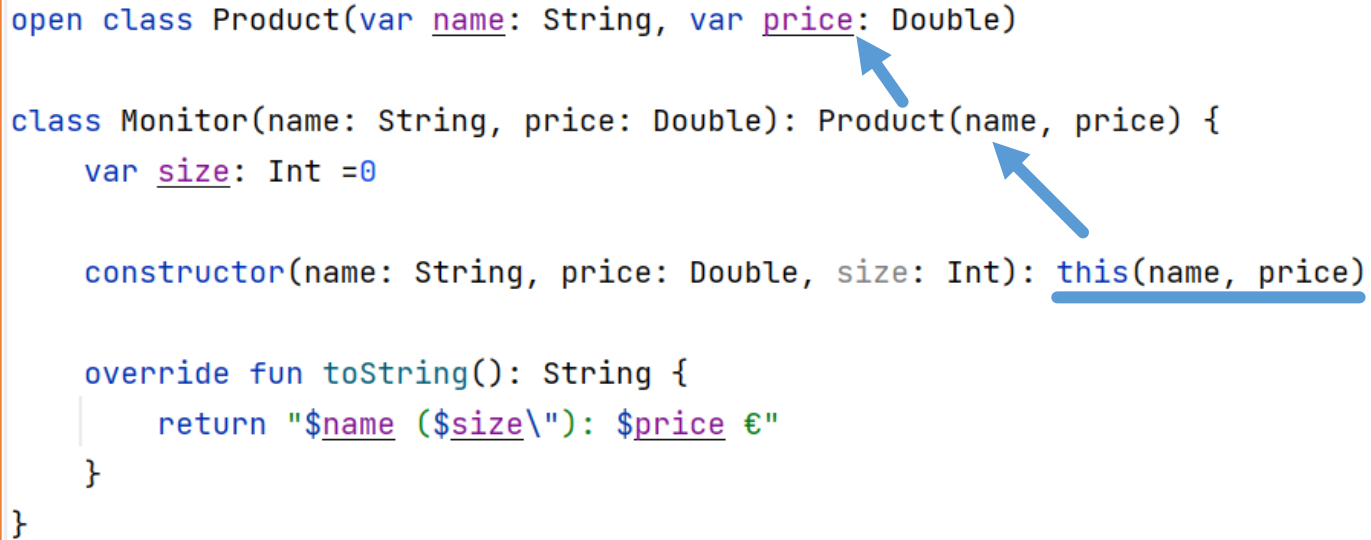
6.- Herencia

```
open class Product(var name: String, var price: Double)

class Monitor(name: String, price: Double): Product(name, price) {
    var size: Int = 0

    constructor(name: String, price: Double, size: Int): this(name, price)

    override fun toString(): String {
        return "$name ($size\"): $price €"
    }
}
```



En la clase hija automáticamente se crea **el constructor primario que llamará al constructor de la clase madre.**

Además, se ha creado un constructor secundario que mediante **this** llamará al constructor primario.

7.- Clases abstractas

Para declarar una clase abstracta se usa la palabra **abstract** delante de la palabra **class**.

De una clase abstracta no se pueden instanciar objetos.

```
abstract class Product(var name: String, var trademark: String) {
    abstract var price: Double
    abstract fun powerOn()
    abstract fun powerOff()
    fun information() {
        println("$name ($trademark): $price €")
    }
}

class Monitor(name: String, trademark: String, var color: String, override var price: Double) : Product(name, trademark) {
    override fun powerOn() {
        println("Encendido")
    }

    override fun powerOff() {
        println("Apagado")
    }
}
```