Instituto Tecnológico Del Valle De Oaxaca

Carrera: Ingeniería En Informática

Asignatura: Aplicaciones para móviles

Nombre Del Docente: RAMÍREZ SANTIAGO BENEDICTO

Semestre: 8° Grupo: B

Integrantes:

- José Antonio López Velasco - 18920032

19 de marzo de 2022

Contenido

Funciones y variables de elementos básicos

Echemos un vistazo al calentamiento de la declaración de la función principal en Kotlin

```
class CornerstoneApplication

fun main(args: Array<String>) {
    println("hello , kotlin!")
}
```

- **Palabra clave para declarar función: diversión**
- **La función se puede definir en la capa más externa del archivo, no es necesario colocarla en la clase**
- **Las matrices están representadas por la clase Array, que es diferente de Java. No hay una sintaxis especial para declarar tipos de matriz en Kotlin**
- **Use println para realizar la misma función que System.out.println en Java, pero más corto**
- **El punto y coma ";" se puede omitir al final de cada línea de código.**

función

Ahora declaremos una función completa con parámetros y valores de retorno.

```
fun max1(a: Int , b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}
```

A partir de esta declaración de función, podemos encontrar algunas características de Kotlin:

- **El nombre de la función sigue inmediatamente a la palabra clave divertida.**
- **El nombre del parámetro viene primero, el tipo de parámetro viene después, separado por ":"**

- **El tipo de valor de retorno de la función se escribe fuera de los corchetes en el lado derecho de la lista de parámetros, separados por ":"**

 Después de leer esta declaración de función, es posible que no crea que tenga nada especial, excepto que la parte de declaración de función es diferente de Java, pero parece que no hay mucha diferencia en general. ¿Podría preguntar sobre la simplicidad de la promesa de Kotlin? No se preocupe si Kotlin no cumple su promesa de simplicidad, no escribiré este artículo. Reescribamos esta función con las características de Kotlin:

```
fun max2(a: Int , b: Int): Int {
    return if(a > b) a else b
}
```

 Un poco más concisa que la función anterior, omití dos pares de corchetes "{}" y una declaración de retorno. A partir de la función anterior, probablemente pueda adivinar que la declaración return devuelve el resultado de `` `if (a> b) else b```. Tu conjetura es correcta, la respuesta es así, déjame explicarte a continuación.

Declaración (declaración) y expresión (expresión)

- **Las expresiones se valoran y se pueden utilizar como parte de otra expresión**
- **La declaración siempre rodea el elemento superior en su bloque de código y no produce un valor. Si en Kotlin es una expresión, en Java si es una declaración. Hay muchas otras estructuras de control en Kotlin que son expresiones, que encontraremos más adelante para una explicación. **

Las funciones de Kotlin pueden ser más concisas que la versión max2 de la función. Cuando digo esto, puedo escuchar a los desarrolladores de Kotlin decirme**you have my word!**

Cuerpo de la función de expresión

Dado que el cuerpo de nuestra función máxima está compuesto por una sola expresión, también podemos hacer que la función sea más concisa.

```
fun max3(a: Int , b: Int) = if (a > b) a else b
```

 Hemos omitido un par de llaves "{}" y una declaración de retorno y la declaración ": Int" que declara el valor de retorno. No hay necesidad de preocuparse por omitir el tipo de valor de retorno que causará errores, porque el compilador analizará la expresión como el cuerpo de la función y usará el tipo de expresión como el tipo de valor de retorno de la función, por lo que no necesita declararse explícitamente. El análisis de contexto del tipo generalmente se denomina inferencia de tipo inteligente.

variable

Palabras clave para declarar variables

- **val (value)** : Referencia inmutable. Las variables declaradas con val no se pueden volver a asignar después de la inicialización. Corresponde a las variables finales en Java.
- **var (variable)** : Referencia variable. El valor de esta variable se puede cambiar. Corresponde a variables no finales ordinarias en Java.
-

 Al declarar una variable en Kotlin, no es necesario especificar el tipo de datos de la variable, porque el compilador puede realizar inferencias de tipos en más contexto, por ejemplo:

```
1. val a = 66
2. val b = 7.5e6
```

 El compilador inferirá que el tipo de variable a es Int y el tipo de variable b es Double basado en 66 y 7.5e6. Por supuesto, el requisito previo es que se puedan utilizar 66 y 7 .5e6 para la derivación. Si no puede proporcionar información sobre el valor que se puede pagar a esta variable, el compilador no puede realizar la derivación de tipos.

Plantilla de cadena

 La plantilla de cadena es una nueva característica de Kotlin

```
fun hello(name: String) {
  println("hello $name")
}
```

 Usamos el nombre de la variable en la cadena, ya través de $ name, esto equivale a vincular la cadena con el signo + en Java. Cuando intenta utilizar el carácter $ en una cadena, necesita escapar de él, println ("\ $"). También puede citar expresiones o funciones más complejas como $ {name.length}

Clases y atributos

 Definir una clase en Kotlin usando la palabra clave class es lo mismo que en Java. Veamos primero una clase de Java

```
public class Person {
  private String name;

  public Person(String name) {
    this.name = name;
  }

  public String getName() {
    return this.name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

 Ah ... Respiré hondo después de escribir esta clase. ¿Cómo puedo decir que es una sensación muy molesta? Puedes ver que las clases de Java están llenas de códigos de plantilla, como la asignación de parámetros en los constructores. Funciones getter y setter para cada variable miembro. En Kotlin, estos códigos de visualización se pueden omitir.

```
class Person(var name: String)
```

 Muy bien, utilicé Kotlin para definir una clase Person, ¿es muy cómodo?

Atributos

 En Java, la combinación de un campo y sus accesos (funciones getter & setter) a menudo se denomina propiedad, y muchos frameworks se basan en gran medida en este concepto. En Kotlin, los atributos son la característica de lenguaje de primera clase, reemplazando completamente los campos y métodos de acceso.

```
class Person {
    val name: String
    var isMarried: Boolean
}
```

- **val atributo de solo lectura: genera un campo y un captador simple**
- **var se puede escribir y leer: genera un campo, un getter y un setter**

 Cuando se declara el atributo, se declara el descriptor de acceso correspondiente. La implementación predeterminada del descriptor de acceso es muy simple, así:

```
fun getName(): String {
    return name
}

fun setName(name: String) {
    this.name = name
}
```

 Si el descriptor de acceso simple predeterminado no puede satisfacer sus necesidades, también puede personalizar el descriptor de acceso: Aquí personalizamos el descriptor de acceso getter para la propiedad isSquare.

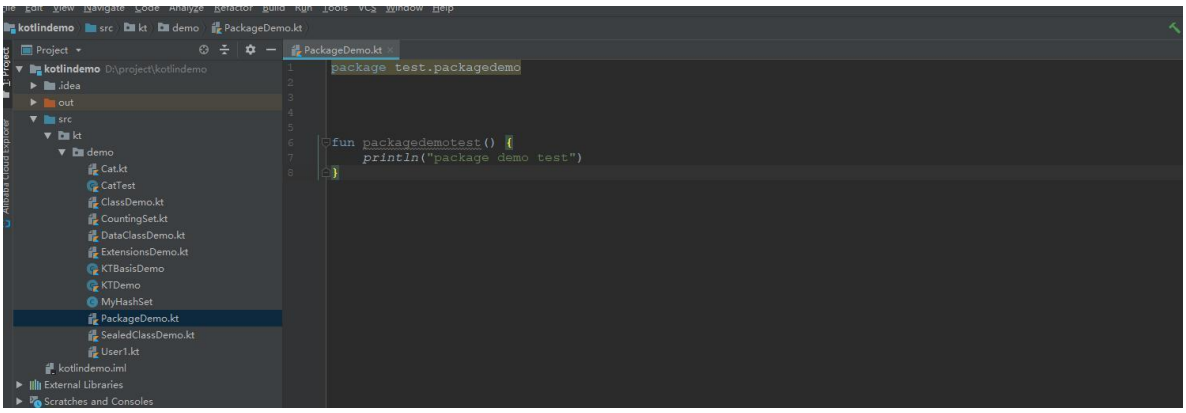
```
class Rectangle(val height: Int , val width: Int) {

    val isSquare: Boolean
        get() = height == width
}
```

Diseño del código fuente de Kotlin: directorios y paquetes

 Todas las clases en Java están organizadas en paquetes. Kotlin también tiene un concepto de paquete similar al de Java. Cada archivo de Kotlin puede comenzar con una declaración de paquete, y todas las declaraciones definidas en el archivo se colocarán en este paquete. Si las declaraciones definidas en otros archivos también tienen el mismo paquete, este archivo puede usarlas directamente; si los paquetes son diferentes, deben importarse antes de su uso. Al igual que Java, la declaración de importación se pasa a través de la palabra clave de importación. Kotlin no distingue entre clases y funciones, y permite importar cualquier tipo de declaración usando la palabra clave import.

 Existen regulaciones estrictas sobre el nombre del paquete y la estructura de carpetas en Java. Por ejemplo, la clase org.demo.Demo debe estar en la carpeta org / demo /. No existe tal restricción en Kotlin, y el nombre del paquete puede ser inconsistente con la estructura de carpetas en Kotlin.



Enum y cuando

Declare una enumeración en Kotlin:

```
1. enum class Color(val r: Int , val g: Int, val b: Int) {
2.   RED(255,0,0) , ORANGE(255,165,0) , YELLOW(255,255,0) ,
3.
```

```
    GREEN(0,255,0) , BLUE(0,0,255) , INDIGO(75,0,130) ,

    VIOLET(238,130,238);

    fun rgb() = (r * 256 + g) * 256 + b
}
```

 Enum es una palabra clave suave en Kotlin: tiene un significado especial solo cuando aparece antes de la palabra clave class, y puede usarse como un nombre normal en otros lugares.

** En el ejemplo anterior, el único lugar donde se debe usar un punto y coma ";" en Kotlin es que cuando se define cualquier método en una enumeración, se debe usar un punto y coma para separar la lista de constantes de enumeración de la definición de función. **

Usar cuando manejar valores de enumeración

when puede considerarse como un sustituto de la estructura de conmutador en Java, pero es más potente que el conmutador. Cuando es una expresión en Kotlin, es la misma expresión que se mencionó antes.

```
fun getChineseDesc(color: Color) = when (color) {
    ROJO -> "Rojo"
        NARANJA -> "Naranja"
        AMARILLO -> "Amarillo"
        VERDE -> "Verde"
        AZUL -> "Azul"
        INDIGO -> "Indigo"
        VIOLETA -> "Morado"
  }
```

 Este es un cuerpo de función de expresión. Este ejemplo demuestra la función de when como expresión.   Características de Kotlin cuando las expresiones:

- **No es necesario escribir una declaración de interrupción en la rama when, si la coincidencia es exitosa, la rama correspondiente se ejecutará más tarde**
- **Puede combinar varios valores en la misma rama, solo separe los valores con comas**

```
fun getWarmth(color: Color) = when (color) {
    RED , ORANGE , YELLOW -> "warm"
```

```
    GREEN -> "neutral"
    BLUE ,INDIGO , VIOLET -> "cold"
  }
```

- **Cualquier objeto se puede utilizar en la estructura when**

 La estructura when en Kotlin es mucho más poderosa que el switch en Java. Switch requiere el uso de constantes (enumeraciones, cadenas o literales numéricos) como condiciones de bifurcación. Y cuando permite utilizar cualquier objeto. En este ejemplo, la función setOf devuelve una instancia de colección Set.

```
fun mix(c1: Color , c2: Color) = when (setOf(c1 , c2)) {
    setOf(RED , YELLOW) -> ORANGE
    setOf(YELLOW , BLUE) -> GREEN
    setOf(BLUE , VIOLET) -> INDIGO
    else -> throw Exception("dirty color")
  }
```

- **when no puede tomar parámetros, si no se proporcionan parámetros to when, la condición de rama puede ser cualquier expresión booleana**

```
fun mixOptimized(c1: Color , c2: Color) = when {
    (c1 == RED && c2 == YELLOW) || (c1 == YELLOW && c2 == RED) -> ORANGE
    (c1 == YELLOW && c2 == BLUE) || (c1 == BLUE && c2 == YELLOW) -> GREEN
    (c1 == BLUE && c2 == VIOLET) || (c1 == VIOLET && c2 == BLUE) -> INDIGO
    else -> throw Exception("dirty color")
  }
```

Conversión de tipo inteligente: combinación de verificación y conversión de tipo

```
interface Expr

class Num(val value: Int) : Expr

class Sum(val left: Expr , val right: Expr) : Expr
fun eval(e: Expr): Int {
  if (e is Num) {
    val n = e as Num
    return n.value
  }

```

```
    if (e is Sum) {
        return eval(e.left) + eval(e.left)
    }

    throw IllegalArgumentException("Unknown expression")
}
```

 De hecho, no es necesario escribir e como Num en esta función, porque se puede inferir que la variable e es de tipo Num preguntando arriba y abajo, por lo que la API Num se puede usar directamente con la variable e en el bloque if. Está en Kotlin es similar a instanceOf en Java, pero en Java, se fuerza una conversión de tipo incluso si la verificación pasa.

```
fun eval(e: Expr): Int {
    if (e is Num) {
        return e.value
    }

    if (e is Sum) {
        return eval(e.left) + eval(e.left)
    }

    throw IllegalArgumentException("Unknown expression")
}
fun eval2(e: Expr): Int = when (e) {
    is Num -> e.value
    is Sum -> eval2(e.left) + eval2(e.left)
    else -> throw IllegalArgumentException("Unknown expression")
}
```

 La regla que se debe seguir en Kotlin es: la última expresión en el bloque de código es el resultado, por lo que la declaración de retorno se puede omitir

Iterando cosas: bucle while y bucle for

 El bucle while y el bucle do while en Kotlin son exactamente iguales que en Java, por lo que no hay nada de qué hablar, así que saltémoslo.

para números de iteración de bucle: intervalo y secuencia

 No hay un ciclo for de Java normal en Kotlin. Reemplazado por el concepto de intervalos. El siguiente código representa un intervalo de 1 a 10. El intervalo en Kotlin es inclusivo o cerrado, lo que significa que el segundo valor siempre es parte del intervalo. Lo más básico que puede hacer con los rangos de números enteros es recorrer todos los valores que contiene. Si puede iterar sobre todos los valores en un intervalo, dicho intervalo se llama secuencia.

```
val oneToTen = 1..10
```

```
for (i in 1..100) {
    println(i)
}
for (i in 100 downTo 1 step 2) {
    println(i)
}
```

 El siguiente ciclo comienza desde 0 y no contiene 100

```
for (i in 0 until 100) {
    println(i)
}
```

Colección iterativa

Iterar lista

```
val list = listOf("axc", "da", "dada", "3rff")
for (e in list) {
    println(e)
}
```

Iteración con superíndice, índice representa el superíndice del elemento y e representa el elemento. Este método de iteración también es aplicable a tipos de matrices.

```
val list = listOf("axc", "da", "dada", "3rff")
for ((index , e) in list.withIndex()) {
    println("index = $index , element = $e")
}
```

Mapa iterativo

```
val map = mapOf(1 to "One", 2 to "Two", 3 to "three")
for ((key , value) in map) {
```

```
    println("key = $key , value = $value")
}
```

Excepciones en Kotlin

 La forma básica de la declaración de manejo de excepciones en Kotlin es similar a la de Java, y la forma de lanzar excepciones no está listada. A diferencia de Java, la estructura de lanzamiento en Kotlin es una expresión y se puede usar como parte de otra expresión. Como muchos otros lenguajes JVM modernos, Kotlin no distingue entre excepciones marcadas y excepciones no marcadas, por lo que no es necesario usar la palabra clave throws para declarar qué excepciones se lanzarán en la función. No es necesario especificar la excepción lanzada por la función, y puede manejarse o no. Este diseño se basa en la decisión tomada por la práctica de usar excepciones en Java. La experiencia ha demostrado que estas reglas de Java a menudo conducen a una gran cantidad de código sin sentido que vuelve a generar o ignora las excepciones, y estas reglas no siempre pueden protegerlo de posibles errores.

 Try también se usa como expresión en Kotlin. Si un bloque de código try se ejecuta normalmente, la última expresión del bloque de código es el resultado. Si se detecta una excepción, la última expresión en el bloque de captura correspondiente es el resultado.

```
fun readNumber(reader: BufferedReader) {
    val number = try {
        Integer.parseInt(reader.readLine())
    } catch (e: NumberFormatException) {
        null
    }
}
```

 Después de aprender lo anterior, podrá utilizar Kotlin para el desarrollo y comprender el código escrito en Kotlin. Espero que Kotlin pueda mejorar la eficiencia de su trabajo, ahorrarle tiempo y ayudarlo a escribir un código más conciso, seguro y práctico.