
KOTLIN
www.postparaprogramadores.com

Contenido

Kotlin - Descripción general

Kotlin es un nuevo lenguaje de programación de código abierto como Java, JavaScript, etc. Es un lenguaje de alto nivel fuertemente tipado estáticamente que combina partes funcionales y técnicas en un mismo lugar. Actualmente, Kotlin apunta a Java y JavaScript. Se ejecuta en JVM.

Kotlin está influenciado por otros lenguajes de programación como Java, Scala, Groovy, Gosu, etc. La sintaxis de Kotlin puede no ser exactamente similar a JAVA, sin embargo, internamente Kotlin depende de la biblioteca Java Class existente para producir resultados maravillosos para los programadores. . Kotlin proporciona interoperabilidad, seguridad de código y claridad a los desarrolladores de todo el mundo.

Ventajas y desventajas

Las siguientes son algunas de las ventajas de usar Kotlin para el desarrollo de su aplicación.

Lenguaje fácil : Kotlin es un lenguaje funcional y muy fácil de aprender. La sintaxis es bastante similar a Java, por lo tanto, es muy fácil de recordar. Kotlin es más expresivo, lo que hace que su código sea más legible y comprensible.

Conciso : Kotlin se basa en JVM y es un lenguaje funcional. Por lo tanto, reduce muchos códigos de placa de caldera utilizados en otros lenguajes de programación.

Tiempo de ejecución y rendimiento : mejor rendimiento y menor tiempo de ejecución.

Interoperabilidad : Kotlin es lo suficientemente maduro como para construir una aplicación interoperable de una manera menos compleja.

Nuevo : Kotlin es un nuevo lenguaje que brinda a los desarrolladores un nuevo comienzo. No es un reemplazo de Java, aunque está desarrollado sobre JVM. Se acepta como el primer idioma oficial del desarrollo de Android. Kotlin se puede definir como - Kotlin = JAVA + nuevas características adicionales actualizadas.

Las siguientes son algunas de las desventajas de Kotlin.

Declaración de espacio de nombres : Kotlin permite a los desarrolladores declarar las funciones en el nivel superior. Sin embargo, cada vez que se declara la misma función en muchos lugares de su aplicación, es difícil entender qué función se llama.

Sin declaración estática : Kotlin no tiene un modificador de manejo estático habitual como Java, lo que puede causar algún problema al desarrollador Java convencional.

Descarga más libros de programación GRATIS click aquí

Kotlin - Configuración del entorno

Sin embargo, si aún desea utilizar Kotlin sin conexión en su sistema local, debe ejecutar los siguientes pasos para configurar su espacio de trabajo local.

Paso 1 : instalación de Java 8.

Kotlin se ejecuta en JVM, por lo tanto. es realmente necesario usar JDK 8 para su desarrollo local de Kotlin. Consulte el sitio web oficial de Oracle para descargar e instalar JDK 8 o una versión anterior. Es posible que tenga que establecer la variable de entorno para JAVA para que pueda funcionar correctamente. Para verificar su instalación en el sistema operativo Windows, presione “java –version” en el símbolo del sistema y, como resultado, le mostrará la versión java instalada en su sistema.

Paso 2 - Instalación de IDE.

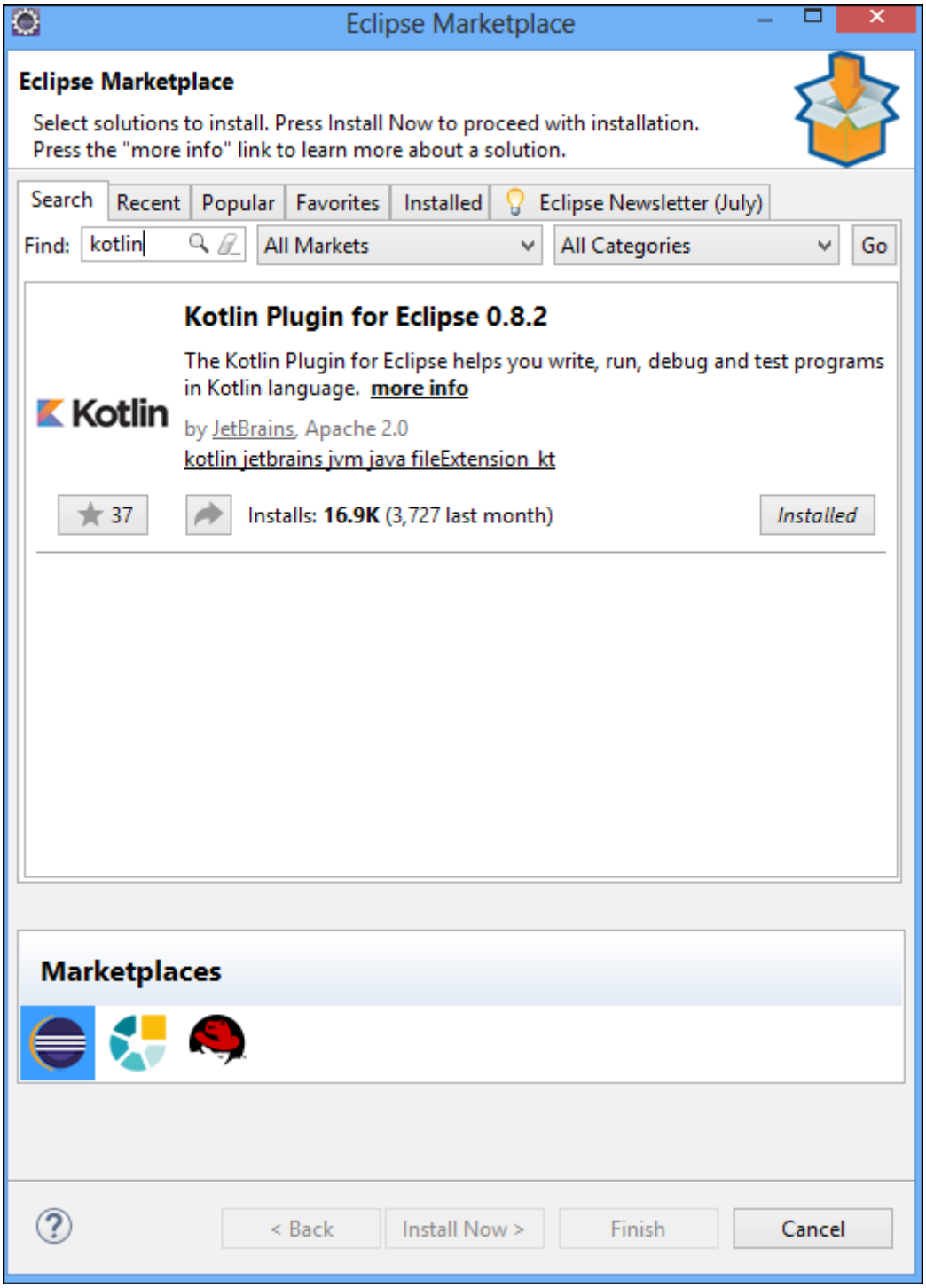
Hay varios IDE disponibles en Internet. Puede usar cualquiera de su elección. Puede encontrar el enlace de descarga de diferentes IDE en la siguiente tabla.

Nombre IDE	Enlace de instalación
NetBeans	https://netbeans.org/downloads/
Eclipse	https://www.eclipse.org/downloads/
Intellij	https://www.jetbrains.com/idea/download/#section = windows

Siempre se recomienda utilizar la versión de software reciente para sacar la máxima facilidad de ella.

Paso 3 : configuración de Eclipse.

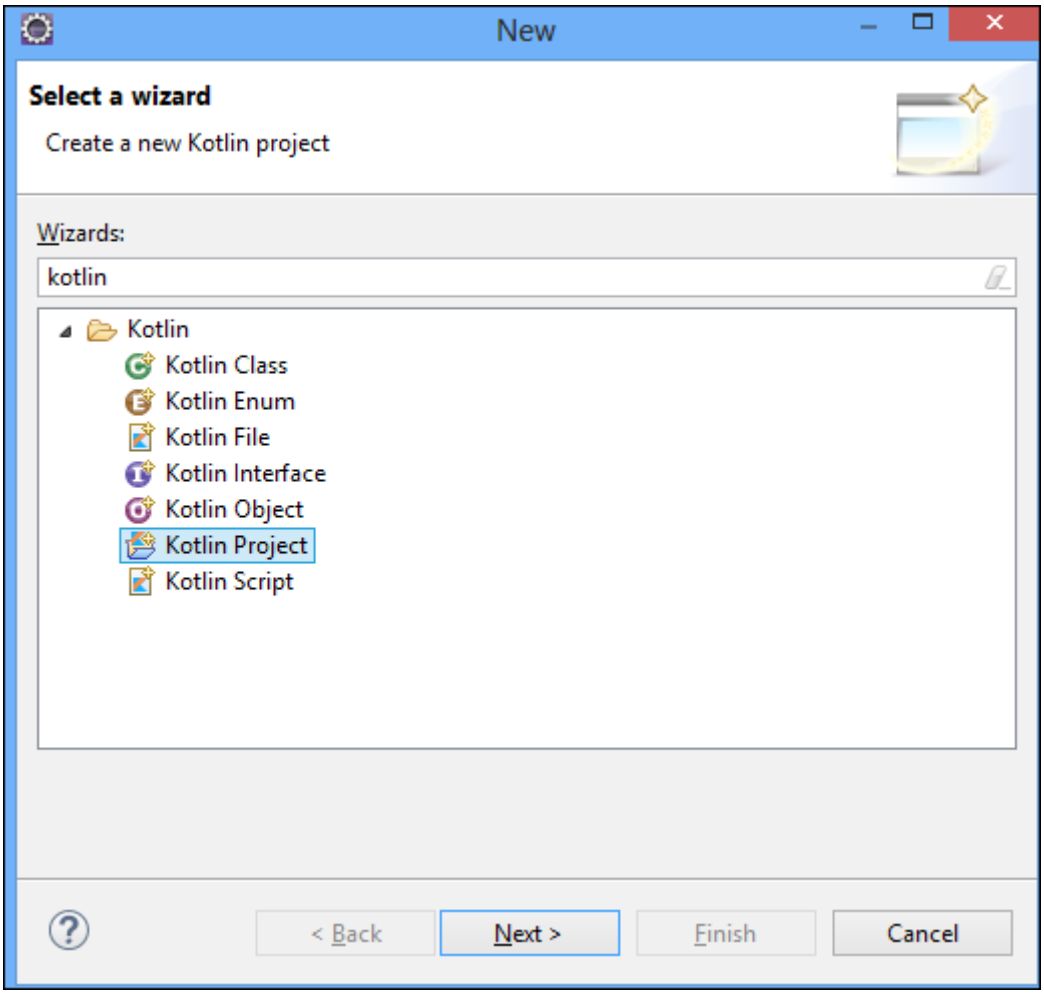
Abra Eclipse y vaya a "Eclipse Market Place". Encontrará la siguiente pantalla.



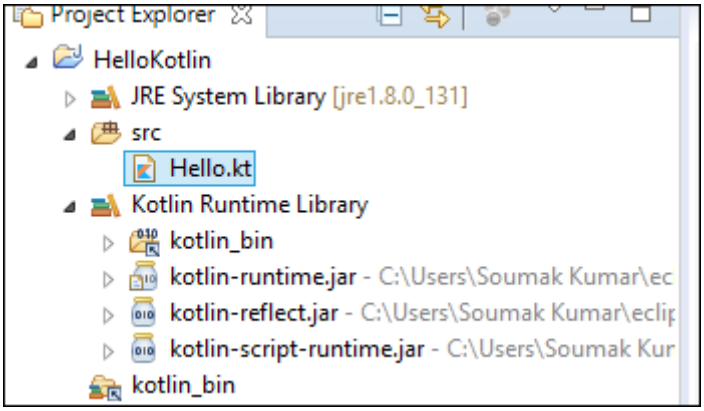
Busque Kotlin en el cuadro de búsqueda e instálelo en su sistema local. Puede tomar algún tiempo dependiendo de la velocidad de internet. Es posible que deba reiniciar su Eclipse, una vez que se haya instalado correctamente.

Paso 4 - Proyecto Kotlin.

Una vez que Eclipse se reinicie con éxito y se instale Kotlin, podrá crear un proyecto de Kotlin sobre la marcha. Vaya a **Archivo** → **Nuevo** → **Otros** y seleccione "Proyecto Kotlin" de la lista.



Una vez que se realiza la configuración del proyecto, puede crear un archivo Kotlin en la carpeta "SRC". Haga clic izquierdo en la carpeta "Src" y presione "nuevo". Obtendrá una opción para el archivo Kotlin, de lo contrario, tendrá que buscar entre los "otros". Una vez que se crea el nuevo archivo, el directorio de su proyecto tendrá el siguiente aspecto.



Su entorno de desarrollo está listo ahora. Continúe y agregue el siguiente código en el archivo "Hello.kt".

```
fun main(args: Array<String>) {
   println("Hello, World!")
```

```
}
```

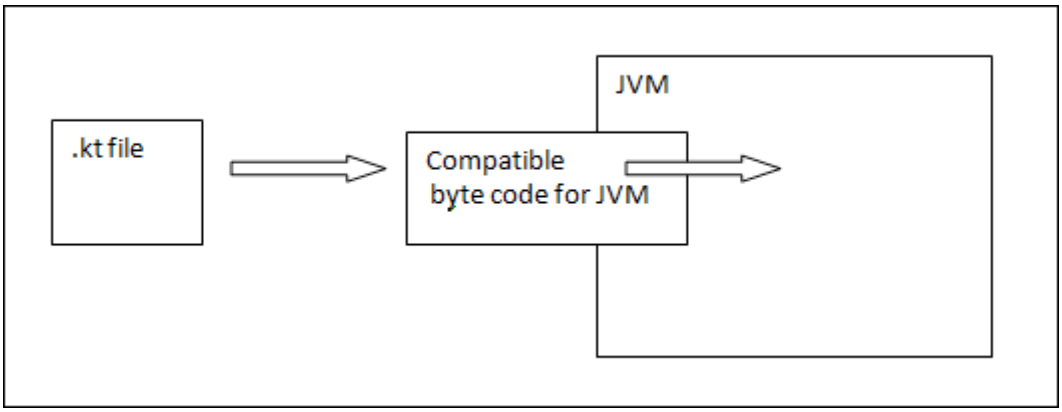
Ejecútelo como una aplicación Kotlin y vea el resultado en la consola como se muestra en la siguiente captura de pantalla. Para una mejor comprensión y disponibilidad, utilizaremos nuestra herramienta de codificación.

```
Hello, World!
```

Kotlin - Arquitectura

Kotlin es un lenguaje de programación y tiene su propia arquitectura para asignar memoria y producir una salida de calidad para el usuario final. Los siguientes son los diferentes escenarios en los que el compilador de Kotlin funcionará de manera diferente, siempre que esté dirigido a otro tipo de lenguajes diferentes, como Java y JavaScript.

El compilador de Kotlin crea un código de bytes y ese código de bytes puede ejecutarse en la JVM, que es exactamente igual al código de bytes generado por el archivo **.class de** Java . Siempre que se ejecute un archivo codificado de dos bytes en la JVM, pueden comunicarse entre sí y así es como se establece una característica interoperable en Kotlin para Java.



Cada vez que se dirige a Kotlin JavaScript, el compilador Kotlin convierte la **.kt** archivo en ES5.1 y genera un código compatible con JavaScript. El compilador Kotlin es capaz de crear códigos compatibles con la plataforma a través de LLVM.

Kotlin - Tipos básicos

En este capítulo, aprenderemos sobre los tipos de datos básicos disponibles en el lenguaje de programación Kotlin.

Números

La representación de números en Kotlin es bastante similar a Java, sin embargo, Kotlin no permite la conversión interna de diferentes tipos de datos. La siguiente tabla enumera diferentes longitudes variables para diferentes números.

Type	Size
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

En el siguiente ejemplo, veremos cómo funciona Kotlin con diferentes tipos de datos. Ingrese el siguiente conjunto de código en nuestro campo de codificación.

```
fun main(args: Array<String>) {
   val a: Int = 10000
   val d: Double = 100.00
   val f: Float = 100.00f
   val l: Long = 1000000004
   val s: Short = 10
   val b: Byte = 1

   println("Your Int Value is "+a);
   println("Your Double  Value is "+d);
   println("Your Float Value is "+f);
   println("Your Long Value is "+l);
   println("Your Short Value is "+s);
   println("Your Byte Value is "+b);
}
```

Cuando ejecuta el código anterior en el campo de codificación, generará el siguiente resultado en la consola web.

```
Your Int Value is 10000
Your Double  Value is 100.0
Your Float Value is 100.0
Your Long Value is 1000000004
Your Short Value is 10
Your Byte Value is 1
```

Caracteres

Kotlin representa el personaje usando **char** . El carácter debe declararse en una comilla simple como **'c'** . Ingrese el siguiente código en nuestro campo de codificación y vea cómo Kotlin interpreta la variable de caracteres. La variable de caracteres no puede declararse como las variables numéricas. La variable Kotlin se puede declarar de dos maneras: una con **“var”** y otra con **“val”** .

```
fun main(args: Array<String>) {
   val letter: Char    // defining a variable
   letter = 'A'        // Assigning a value to it
   println("$letter")
}
```

El código anterior generará el siguiente resultado en la ventana de resultados del navegador.

```
A
```

Booleano

Boolean es muy simple como otros lenguajes de programación. Solo tenemos dos valores para Boolean: verdadero o falso. En el siguiente ejemplo, veremos cómo Kotlin interpreta el booleano.

```
fun main(args: Array<String>) {
   val letter: Boolean   // defining a variable
   letter = true         // Assinging a value to it
   println("Your character value is "+"$letter")
}
```

El código anterior generará el siguiente resultado en el navegador.

```
Your character value is true
```

Instrumentos de cuerda

Las cadenas son matrices de caracteres. Al igual que Java, son inmutables por naturaleza. Tenemos dos tipos de cadenas disponibles en Kotlin: una se llama **cadena sin procesar** y otra se llama **cadena escapada** . En el siguiente ejemplo, haremos uso de estas cadenas.

```
fun main(args: Array<String>) {
   var rawString :String  = "I am Raw String!"
   val escapedString : String  = "I am escaped String!\n"

   println("Hello!"+escapedString)
   println("Hey!!"+rawString)
}
```

El ejemplo anterior de String escapado permite proporcionar espacio de línea adicional después de la primera instrucción de impresión. A continuación se mostrará el resultado en el navegador.

```
Hello!I am escaped String!

Hey!!I am Raw String!
```

Matrices

Las matrices son una colección de datos homogéneos. Al igual que Java, Kotlin admite matrices de diferentes tipos de datos. En el siguiente ejemplo, haremos uso de diferentes matrices.

```
fun main(args: Array<String>) {
   val numbers: IntArray = intArrayOf(1, 2, 3, 4, 5)
   println("Hey!! I am array Example"+numbers[2])
}
```

El fragmento de código anterior produce el siguiente resultado. La indexación de la matriz es similar a otros lenguajes de programación. Aquí, estamos buscando un segundo índice, cuyo valor es "3".

```
Hey!! I am array Example3
```

Colecciones

La recopilación es una parte muy importante de la estructura de datos, lo que facilita el desarrollo de software para los ingenieros. Kotlin tiene dos tipos de colección: una es una **colección inmutable** (que significa listas, mapas y conjuntos que no pueden ser editables) y otra es una **colección mutable** (este tipo de colección es editable). Es muy importante tener en cuenta el tipo de colección utilizada en su aplicación, ya que el sistema Kotlin no representa ninguna diferencia específica en ellas.

```
fun main(args: Array<String>) {
   val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
//mutable List
   val readOnlyView: List<Int> = numbers                    //
immutable list
   println("my mutable list--"+numbers)         // prints "[1,
2, 3]"
   numbers.add(4)
   println("my mutable list after addition --"+numbers)
// prints "[1, 2, 3, 4]"
   println(readOnlyView)
   readOnlyView.clear()     // ⇒ does not compile
// gives error
}
```

El código anterior generará el siguiente resultado en el navegador. Da un error cuando intentamos borrar la lista mutable de colección.

```
main.kt:9:18: error: unresolved reference: clear
   readOnlyView.clear()    // -> does not compile
                  ^
```

En la colección, Kotlin proporciona algunos métodos útiles como **first (), last (), filter ()** , etc. Todos estos métodos son autodescriptivos y fáciles de implementar. Además, Kotlin sigue la misma estructura, como Java, mientras implementa la recopilación. Usted es libre de implementar cualquier colección de su elección, como Mapa y Conjunto.

En el siguiente ejemplo, hemos implementado Map and Set utilizando diferentes métodos integrados.

```
fun main(args: Array<String>) {
   val items = listOf(1, 2, 3, 4)
   println("First Element of our list----"+items.first())
   println("Last Element of our list----"+items.last())
   println("Even Numbers of our List----"+items.
      filter { it % 2 = = 0 })   // returns [2, 4]

   val readWriteMap = hashMapOf("foo" to 1, "bar" to 2)
   println(readWriteMap["foo"])  // prints "1"

   val strings = hashSetOf("a", "b", "c", "c")
   println("My Set Values are"+strings)
}
```

El fragmento de código anterior produce el siguiente resultado en el navegador.

```
First Element of our list----1
Last Element of our list----4
Even Numbers of our List----[2, 4]
1
My Set Values are[a, b, c]
```

Rangos

Los rangos son otra característica única de Kotlin. Al igual que Haskell, proporciona un operador que lo ayuda a recorrer un rango. Internamente, se implementa utilizando **rangeTo ()** y su forma de operador es **(..)** .

En el siguiente ejemplo, veremos cómo Kotlin interpreta este operador de rango.

```
fun main(args: Array<String>) {
   val i:Int  = 2
   for (j in 1..4)
   print(j) // prints "1234"
```

```
    if (i in 1..10) { // equivalent of 1 < = i && i < = 10
       println("we found your number --"+i)
    }
}
```

El fragmento de código anterior produce el siguiente resultado en el navegador.

```
1234we found your number --2
```

Kotlin - Control de flujo

En el capítulo anterior hemos aprendido sobre los diferentes tipos de tipos de datos disponibles en el sistema Kotlin. En este capítulo, discutiremos los diferentes tipos de mecanismos de control de flujo disponibles en Kotlin.

Si - de lo contrario

Kotlin es un lenguaje funcional, por lo tanto, como todo lenguaje funcional en Kotlin, **"si"** es una expresión, no es una palabra clave. La expresión **"si"** devolverá un valor siempre que sea necesario. Al igual que otros lenguajes de programación, el bloque **"if-else"** se utiliza como operador de verificación condicional inicial. En el siguiente ejemplo, compararemos dos variables y proporcionaremos la salida requerida en consecuencia.

```
fun main(args: Array<String>) {
   val a:Int = 5
   val b:Int = 2
   var max: Int

   if (a > b) {
      max = a
   } else {
      max = b
   }
   print("Maximum of a or b is " +max)

   // As expression
   // val max = if (a > b) a else b
}
```

El fragmento de código anterior produce el siguiente resultado como resultado en el navegador. Nuestro ejemplo también contiene otra línea de código, que describe cómo usar la declaración **"If"** como expresión.

```
Maximum of a or b is 5
```

Uso de cuando

Si está familiarizado con otros lenguajes de programación, es posible que haya oído hablar del término declaración de cambio, que es básicamente un operador condicional cuando se pueden aplicar múltiples condiciones en una

variable en particular. **El** operador **"cuándo"** compara el valor de la variable con las condiciones de la rama. Si cumple la condición de bifurcación, ejecutará la declaración dentro de ese alcance. En el siguiente ejemplo, aprenderemos más sobre "cuándo" en Kotlin.

```
fun main(args: Array<String>) {
   val x:Int = 5
   when (x) {
      1 -> print("x = = 1")
      2 -> print("x = = 2")

      else -> { // Note the block
         print("x is neither 1 nor 2")
      }
   }
}
```

El fragmento de código anterior produce el siguiente resultado en el navegador.

```
x is neither 1 nor 2
```

En el ejemplo anterior, el compilador de Kotlin hace coincidir el valor de **x** con las ramas dadas. Si no coincide con ninguna de las ramas, ejecutará la parte else. Prácticamente, cuando es equivalente a bloque múltiple if. Kotlin proporciona otra flexibilidad al desarrollador, donde el desarrollador puede proporcionar múltiples controles en la misma línea al proporcionar "," dentro de los controles. Modifiquemos el ejemplo anterior de la siguiente manera.

```
fun main(args: Array<String>) {
   val x:Int = 5
   when (x) {
      1,2 -> print(" Value of X either 1,2")

      else -> { // Note the block
         print("x is neither 1 nor 2")
      }
   }
}
```

Ejecute lo mismo en el navegador, lo que generará el siguiente resultado en el navegador.

```
x is neither 1 nor 2
```

En bucle

Loop es una invención que proporciona la flexibilidad para iterar a través de cualquier tipo de estructura de datos. Al igual que otros lenguajes de programación, Kotlin también proporciona muchos tipos de metodología de bucle, sin embargo, entre ellos, **"For"** es el más exitoso. La implementación y

el uso de For loop es conceptualmente similar a Java for loop. El siguiente ejemplo muestra cómo podemos usar lo mismo en ejemplos de la vida real.

```
fun main(args: Array<String>) {
   val items = listOf(1, 2, 3, 4)
   for (i in items) println("values of the array"+i)
}
```

En el fragmento de código anterior, hemos declarado una lista nombrada como "elementos" y usando for loop estamos iterando a través de esa lista definida e imprimiendo su valor en el navegador. A continuación se muestra la salida.

```
values of the array1
values of the array2
values of the array3
values of the array4
```

El siguiente es otro ejemplo de código, en el que estamos utilizando alguna función de biblioteca para hacer que nuestro desarrollo funcione más fácil que nunca.

```
fun main(args: Array<String>) {
   val items = listOf(1, 22, 83, 4)

   for ((index, value) in items.withIndex()) {
      println("the element at $index is $value")
   }
}
```

Una vez que compilamos y ejecutamos el código anterior en nuestro campo de codificación, producirá el siguiente resultado en el navegador.

```
the element at 0 is 1
the element at 1 is 22
the element at 2 is 83
the element at 3 is 4
```

Bucle While y Bucle Do-While

While y Do-While funcionan exactamente de manera similar a como lo hacen en otros lenguajes de programación. La única diferencia entre estos dos bucles es que, en el caso del bucle Do-while, la condición se probará al final del bucle. El siguiente ejemplo muestra el uso del **bucle While** .

```
fun main(args: Array<String>) {
   var x:Int = 0
   println("Example of While Loop--")

   while(x< = 10) {
      println(x)
```

```
        x++
    }
}
```

El fragmento de código anterior produce el siguiente resultado en el navegador.

```
Example of While Loop--
0
1
2
3
4
5
6
7
8
9
10
```

Kotlin también tiene otro ciclo llamado ciclo Do-While, donde el cuerpo del ciclo se ejecutará una vez, solo entonces se comprobará la condición. El siguiente ejemplo muestra el uso del **bucle Do-while** .

```
fun main(args: Array<String>) {
    var x:Int = 0
    do {
        x = x + 10
        println("I am inside Do block---"+x)
    } while(x <= 50)
}
```

El fragmento de código anterior produce el siguiente resultado en el navegador. En el código anterior, el compilador de Kotlin ejecutará el bloque DO, luego irá a verificar la condición mientras se bloquea.

```
I am inside Do block---10
I am inside Do block---20
I am inside Do block---30
I am inside Do block---40
I am inside Do block---50
I am inside Do block---60
```

Uso de retorno, descanso, continuar

Si está familiarizado con algún lenguaje de programación, debe tener una idea de las diferentes palabras clave que nos ayudan a implementar un buen flujo de control en la aplicación. Las siguientes son las diferentes palabras clave que se pueden usar para controlar los bucles o cualquier otro tipo de flujo de control.

Retorno : Retorno es una palabra clave que devuelve algún valor a la función de llamada desde la función llamada. En el siguiente ejemplo,

implementaremos este escenario utilizando nuestro campo de codificación Kotlin.

```
fun main(args: Array<String>) {
   var x:Int = 10
   println("The value of X is--"+doubleMe(x))
}
fun doubleMe(x:Int):Int {
   return 2*x;
}
```

En el fragmento de código anterior, estamos llamando a otra función y multiplicando la entrada con 2, y devolviendo el valor resultante a la función llamada que es nuestra función principal. Kotlin define la función de una manera diferente que veremos en un capítulo posterior. Por ahora, es suficiente entender que el código anterior generará la siguiente salida en el navegador.

```
The value of X is--20
```

Continuar y romper - Continuar y romper son la parte más vital de un problema lógico. La palabra clave "break" termina el flujo del controlador si alguna condición ha fallado y "continuar" hace lo contrario. Toda esta operación ocurre con visibilidad inmediata. Kotlin es más inteligente que otros lenguajes de programación, en donde el desarrollador puede aplicar más de una etiqueta como visibilidad. El siguiente código muestra cómo estamos implementando esta etiqueta en Kotlin.

```
fun main(args: Array<String>) {
   println("Example of Break and Continue")
   myLabel@ for(x in 1..10) { // appling the custom label
      if(x = = 5) {
         println("I am inside if block with value"+x+"\n--
hence it will close the operation")
         break@myLabel //specifing the label
      } else {
         println("I am inside else block with value"+x)
         continue@myLabel
      }
   }
}
```

El fragmento de código anterior produce el siguiente resultado en el navegador.

```
Example of Break and Continue
I am inside else block with value1
I am inside else block with value2
I am inside else block with value3
I am inside else block with value4
I am inside if block with value5
-- hence it will close the operation
```

Como puede ver, el controlador continúa el ciclo hasta que, a menos que el valor de **x** sea 5. Una vez que el valor de **x** alcanza 5, comienza a ejecutar el bloque if y una vez que se alcanza la declaración de interrupción, todo el flujo de control termina el programa. ejecución.

Kotlin - Clase y objeto

En este capítulo, aprenderemos los conceptos básicos de la Programación Orientada a Objetos (OOP) usando Kotlin. Aprenderemos sobre la clase y su objeto y cómo jugar con ese objeto. Por definición de OOP, una clase es un plano de una entidad en tiempo de ejecución y el objeto es su estado, que incluye tanto su comportamiento como su estado. En Kotlin, la declaración de clase consiste en un encabezado de clase y un cuerpo de clase rodeado de llaves, similar a Java.

```
Class myClass { // class Header

   // class Body
}
```

Al igual que Java, Kotlin también permite crear varios objetos de una clase y usted es libre de incluir sus miembros y funciones de clase. Podemos controlar la visibilidad de las variables de los miembros de la clase usando diferentes palabras clave que aprenderemos en el Capítulo 10 - Control de visibilidad. En el siguiente ejemplo, crearemos una clase y su objeto a través del cual accederemos a diferentes miembros de datos de esa clase.

```
class myClass {
   // property (data member)
   private var name: String = "Tutorials.point"

   // member function
   fun printMe() {
      print("You are at the best Learning website Named-
"+name)
   }
}
fun main(args: Array<String>) {
   val obj = myClass() // create obj object of myClass class
   obj.printMe()
}
```

El código anterior generará el siguiente resultado en el navegador, donde llamaremos a printMe () de myClass usando su propio objeto.

```
You are at the best Learning website Named- Tutorials.point
```

Clase anidada

Por definición, cuando una clase se ha creado dentro de otra clase, se llama como una clase anidada. En Kotlin, la clase anidada es estática por defecto,

por lo tanto, se puede acceder sin crear ningún objeto de esa clase. En el siguiente ejemplo, veremos cómo Kotlin interpreta nuestra clase anidada.

```
fun main(args: Array<String>) {
   val demo = Outer.Nested().foo() // calling nested class
method
   print(demo)
}
class Outer {
   class Nested {
      fun foo() = "Welcome to The Postparaprogramadores.com"
   }
}
```

El código anterior generará el siguiente resultado en el navegador.

```
Welcome to The Postparaprogramadores.com
```

Clase interior

Cuando una clase anidada se marca como "interna", se llamará como una clase interna. El miembro de datos de la clase externa puede acceder a una clase interna. En el siguiente ejemplo, accederemos al miembro de datos de la clase externa.

```
fun main(args: Array<String>) {
   val demo = Outer().Nested().foo() // calling nested class
method
   print(demo)
}
class Outer {
   private val welcomeMessage: String = "Welcome to the
Postparaprogramadores.com"
   inner class Nested {
      fun foo() = welcomeMessage
   }
}
```

El código anterior generará el siguiente resultado en el navegador, donde llamaremos a la clase anidada utilizando el constructor predeterminado proporcionado por los compiladores de Kotlin en el momento de la compilación.

```
Welcome to the Postparaprogramadores.com
```

Clase interna anónima

La clase interna anónima es un concepto bastante bueno que hace que la vida de un programador sea muy fácil. Cada vez que implementamos una interfaz, el concepto de bloqueo interno anónimo aparece. El concepto de crear un objeto de interfaz utilizando la referencia de objeto de tiempo de ejecución se

conoce como clase anónima. En el siguiente ejemplo, crearemos una interfaz y crearemos un objeto de esa interfaz utilizando el mecanismo de clase interna anónima.

```
fun main(args: Array<String>) {
   var programmer :Human = object:Human // creating an
instance of the interface {
      override fun think() { // overriding the think method
         print("I am an example of Anonymous Inner Class ")
      }
   }
   programmer.think()
}
interface Human {
   fun think()
}
```

El código anterior generará el siguiente resultado en el navegador.

```
I am an example of Anonymous Inner Class
```

Alias de tipo

Los alias de tipo son propiedad del compilador Kotlin. Proporciona la flexibilidad de crear un nuevo nombre de un tipo existente, no crea un nuevo tipo. Si el nombre del tipo es demasiado largo, puede introducir fácilmente un nombre más corto y usarlo para uso futuro. Los alias de tipo son realmente útiles para el tipo complejo. En la última versión, Kotlin revocó la compatibilidad con los alias de tipo, sin embargo, si está utilizando una versión anterior de Kotlin, puede haberla utilizado de la siguiente manera:

```
typealias NodeSet = Set<Network.Node>
typealias FileTable<K> = MutableMap<K, MutableList<File>>
```

Kotlin - Constructores

En este capítulo, aprenderemos sobre los constructores en Kotlin. Kotlin tiene dos tipos de constructor: uno es el **constructor primario** y el otro es el **constructor secundario** . Una clase de Kotlin puede tener un constructor primario y uno o más constructores secundarios. El constructor Java inicializa las variables miembro, sin embargo, en Kotlin el constructor primario inicializa la clase, mientras que el constructor secundario ayuda a incluir algo de lógica adicional mientras se inicializa la misma. El constructor primario se puede declarar a nivel de encabezado de clase como se muestra en el siguiente ejemplo.

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

En el ejemplo anterior, hemos declarado el constructor primario dentro del paréntesis. Entre los dos campos, el nombre es de solo lectura, ya que se

declara como "val", mientras que la edad del campo se puede editar. En el siguiente ejemplo, utilizaremos el constructor primario.

```
fun main(args: Array<String>) {
   val person1 = Person("Postparaprogramadores.com", 15)
   println("First Name = ${person1.firstName}")
   println("Age = ${person1.age}")
}
class Person(val firstName: String, var age: Int) {
}
```

El código anterior inicializará automáticamente las dos variables y proporcionará el siguiente resultado en el navegador.

```
First Name = Postparaprogramadores.com
Age = 15
```

Como se mencionó anteriormente, Kotlin permite crear uno o más constructores secundarios para su clase. Este constructor secundario se crea utilizando la palabra clave "constructor". Se requiere siempre que desee crear más de un constructor en Kotlin o cuando desee incluir más lógica en el constructor primario y no puede hacerlo porque otra clase puede llamar al constructor primario. Eche un vistazo al siguiente ejemplo, donde hemos creado un constructor secundario y estamos usando el ejemplo anterior para implementar el mismo.

```
fun main(args: Array<String>) {
   val HUman = HUman("Postparaprogramadores.com", 25)
   print("${HUman.message}"+"${HUman.firstName}"+
      "Welcome to the example of Secondary  constructor, Your
Age is-${HUman.age}")
}
class HUman(val firstName: String, var age: Int) {
   val message:String  = "Hey!!!"
      constructor(name : String , age :Int ,message
:String):this(name,age) {
   }
}
```

Nota - Se puede crear cualquier cantidad de constructores secundarios, sin embargo, todos esos constructores deben llamar al constructor primario directa o indirectamente.

El código anterior generará el siguiente resultado en el navegador.

```
Hey!!! Postparaprogramadores.comWelcome to the example of
Secondary  constructor, Your Age is- 25
```

Kotlin - Herencia

En este capítulo, aprenderemos sobre la herencia. Por definición, todos sabemos que la herencia significa acumular algunas propiedades de la clase madre en la clase hija. En Kotlin, la clase base se nombra como "Any", que es la superclase de la clase predeterminada 'any' declarada en Kotlin. Al igual

que todos los demás OOPS, Kotlin también proporciona esta funcionalidad utilizando una palabra clave conocida como **":"** .

Todo en Kotlin es, por defecto, final, por lo tanto, necesitamos usar la palabra clave "abrir" delante de la declaración de clase para que se pueda heredar. Eche un vistazo al siguiente ejemplo de herencia.

```
import java.util.Arrays

open class ABC {
   fun think () {
      print("Hey!! i am thiking ")
   }
}
class BCD: ABC(){ // inheritence happend using default
constructor
}

fun main(args: Array<String>) {
   var  a = BCD()
   a.think()
}
```

El código anterior generará el siguiente resultado en el navegador.

```
Hey!! i am thiking
```

Ahora, ¿qué pasa si queremos anular el método think () en la clase secundaria. Luego, debemos considerar el siguiente ejemplo donde estamos creando dos clases y anular una de sus funciones en la clase secundaria.

```
import java.util.Arrays

open class ABC {
   open fun think () {
      print("Hey!! i am thinking ")
   }
}
class BCD: ABC() { // inheritance happens using default
constructor
   override fun think() {
      print("I Am from Child")
   }
}
fun main(args: Array<String>) {
   var  a = BCD()
   a.think()
}
```

El fragmento de código anterior llamará al método heredado de la clase secundaria y generará el siguiente resultado en el navegador. Al igual que Java, Kotlin tampoco permite múltiples herencias.

I Am from Child

Kotlin - Interfaz

En este capítulo, aprenderemos sobre la interfaz en Kotlin. En Kotlin, la interfaz funciona exactamente de manera similar a Java 8, lo que significa que pueden contener la implementación de métodos y la declaración de métodos abstractos. Una clase puede implementar una interfaz para utilizar su funcionalidad definida. Ya hemos introducido un ejemplo con una interfaz en el Capítulo 6 - sección "clase interna anónima". En este capítulo, aprenderemos más al respecto. La palabra clave "interfaz" se utiliza para definir una interfaz en Kotlin como se muestra en el siguiente fragmento de código.

```
interface ExampleInterface {
   var myVar: String     // abstract property
   fun absMethod()       // abstract method
   fun sayHello() = "Hello there" // method with default
implementation
}
```

En el ejemplo anterior, hemos creado una interfaz llamada "EjemploInterfaz" y dentro de ella tenemos un par de propiedades y métodos abstractos todos juntos. Mire la función llamada "sayHello ()", que es un método implementado.

En el siguiente ejemplo, implementaremos la interfaz anterior en una clase.

```
interface ExampleInterface  {
   var myVar: Int             // abstract property
   fun absMethod():String     // abstract method

   fun hello() {
      println("Hello there, Welcome to
Postparaprogramadores.Com!")
   }
}
class InterfaceImp : ExampleInterface {
   override var myVar: Int = 25
   override fun absMethod() = "Happy Learning "
}
fun main(args: Array<String>) {
   val obj = InterfaceImp()
   println("My Variable Value is = ${obj.myVar}")
   print("Calling hello(): ")
   obj.hello()

   print("Message from the Website-- ")
   println(obj.absMethod())
}
```

El código anterior generará el siguiente resultado en el navegador.

```
Calling hello(): Hello there, Welcome to
Postparaprogramadores.Com!
Message from the Website-- Happy Learning
```

Como se mencionó anteriormente, Kotlin no admite múltiples herencias, sin embargo, se puede lograr lo mismo implementando más de dos interfaces a la vez.

En el siguiente ejemplo, crearemos dos interfaces y luego implementaremos ambas interfaces en una clase.

```
interface A {
   fun printMe() {
      println(" method of interface A")
   }
}
interface B  {
   fun printMeToo() {
      println("I am another Method from interface B")
   }
}

// implements two interfaces A and B
class multipleInterfaceExample: A, B

fun main(args: Array<String>) {
   val obj = multipleInterfaceExample()
   obj.printMe()
   obj.printMeToo()
}
```

En el ejemplo anterior, hemos creado dos interfaces de muestra A, B y en la clase llamada "multipleInterfaceExample" hemos implementado dos interfaces declaradas anteriormente. El código anterior generará el siguiente resultado en el navegador.

```
method of interface A
I am another Method from interface B
```

Kotlin - Control de visibilidad

En este capítulo, aprenderemos sobre los diferentes modificadores disponibles en el idioma Kotlin. **El modificador de acceso** se utiliza para restringir el uso de las variables, métodos y clases utilizados en la aplicación. Al igual que otros lenguajes de programación OOP, este modificador es aplicable en múltiples lugares, como en el encabezado de la clase o en la declaración del método. Hay cuatro modificadores de acceso disponibles en Kotlin.

Privado

Las clases, métodos y paquetes se pueden declarar con un modificador privado. Una vez que algo se declara como privado, será accesible dentro de su alcance inmediato. Por ejemplo, se puede acceder a un paquete privado dentro de ese archivo específico. Una clase o interfaz privada solo puede ser accesible por sus miembros de datos, etc.

```
private class privateExample {
```

```
    private val i = 1
    private val doSomething() {
    }
}
```

En el ejemplo anterior, la clase **"privateExample"** y la variable i solo pueden ser accesibles en el mismo archivo Kotlin, donde se menciona ya que todas se declaran como privadas en el bloque de declaración.

Protegido

Protegido es otro modificador de acceso para Kotlin, que actualmente no está disponible para la declaración de nivel superior, ya que ningún paquete puede protegerse. Una clase o interfaz protegida es visible solo para su subclase.

```
class A() {
    protected val i = 1
}
class B : A() {
    fun getValue() : Int {
        return i
    }
}
```

En el ejemplo anterior, la variable **"i"** se declara protegida, por lo tanto, solo es visible para su subclase.

Interno

Internal es un modificador recientemente agregado introducido en Kotlin. Si algo está marcado como interno, entonces ese campo específico estará en el campo interno. Un paquete interno es visible solo dentro del módulo bajo el cual se implementa. Una interfaz de clase interna solo es visible por otra clase presente dentro del mismo paquete o módulo. En el siguiente ejemplo, veremos cómo implementar un método interno.

```
class internalExample {
    internal val i = 1
    internal fun doSomething() {
    }
}
```

En el ejemplo anterior, el método llamado "doSomething" y la variable se menciona como interna, por lo tanto, estos dos campos solo pueden ser accesibles dentro del paquete bajo el cual se declara.

Público

Se puede acceder al modificador público desde cualquier parte del espacio de trabajo del proyecto. Si no se especifica ningún modificador de acceso, de forma predeterminada estará en el ámbito público. En todos nuestros ejemplos

anteriores, no hemos mencionado ningún modificador, por lo tanto, todos están en el ámbito público. El siguiente es un ejemplo para comprender más sobre cómo declarar una variable o método público.

```
class publicExample {
   val i = 1
   fun doSomething() {
   }
}
```

En el ejemplo anterior, no hemos mencionado ningún modificador, por lo tanto, todos estos métodos y variables son públicos por defecto.

Kotlin - Extensión

En este capítulo, aprenderemos sobre otra nueva característica de Kotlin llamada "Extensión". Usando la extensión, podremos agregar o eliminar algunas funciones del método incluso sin heredarlas o modificarlas. Las extensiones se resuelven estadísticamente. En realidad, no modifica la clase existente, pero crea una función invocable que se puede llamar con una operación de punto.

Extensión de la función

En la extensión de la función, Kotlin permite definir un método fuera de la clase principal. En el siguiente ejemplo, veremos cómo se implementa la extensión a nivel funcional.

```
class Alien {
   var skills : String = "null"

   fun printMySkills() {
      print(skills)
   }
}
fun main(args: Array<String>) {
   var  a1 = Alien()
   a1.skills = "JAVA"
   //a1.printMySkills()

   var  a2 = Alien()
   a2.skills = "SQL"
   //a2.printMySkills()

   var  a3 = Alien()
   a3.skills = a1.addMySkills(a2)
   a3.printMySkills()
}
fun Alien.addMySkills(a:Alien):String{
   var a4 = Alien()
   a4.skills = this.skills + " " +a.skills
   return a4.skills
```

```
}
```

En el ejemplo anterior, no tenemos ningún método dentro de la clase "Alien" llamada "addMySkills ()", sin embargo, todavía estamos implementando el mismo método en otro lugar fuera de la clase. Esta es la magia de la extensión.

El código anterior generará el siguiente resultado en el navegador.

```
JAVA SQL
```

Extensión de objeto

Kotlin proporciona otro mecanismo para implementar la funcionalidad estática de Java. Esto se puede lograr usando la palabra clave "objeto complementario". Usando este mecanismo, podemos crear un objeto de una clase dentro de un método de fábrica y luego podemos llamar a ese método usando la referencia del nombre de la clase. En el siguiente ejemplo, crearemos un "objeto complementario".

```
fun main(args: Array<String>) {
   println("Heyyy!!!"+A.show())
}
class A {
   companion object {
      fun show():String {
         return("You are learning Kotlin from
Postparaprogramadores.com")
      }
   }
}
```

El código anterior generará el siguiente resultado en el navegador.

```
Heyyy!!! You are learning Kotlin from
Postparaprogramadores.com
```

El ejemplo anterior parece estático en Java, sin embargo, en tiempo real estamos creando un objeto como una variable miembro de esa misma clase. Es por eso que también se incluye en la propiedad de extensión y se puede llamar alternativamente como una extensión de objeto. Básicamente, está extendiendo el objeto de la misma clase para usar algunas de las funciones miembro.

Kotlin - Clases de datos

En este capítulo, aprenderemos más sobre las clases de datos del lenguaje de programación Kotlin. Una clase se puede marcar como una clase de datos siempre que se marque como "datos". Este tipo de clase se puede utilizar para mantener separados los datos básicos. Aparte de esto, no proporciona ninguna otra funcionalidad.

Todas las clases de datos deben tener un constructor primario y todo el constructor primario debe tener al menos un parámetro. Siempre que una clase se marque como datos, podemos usar algunas de las funciones incorporadas de esa clase de datos como "toString ()", "hashCode ()", etc. Cualquier clase de datos no puede tener un modificador como abstracto y abierto o interno. La clase de datos también se puede extender a otras clases. En el siguiente ejemplo, crearemos una clase de datos.

```
fun main(args: Array<String>) {
   val book: Book = Book("Kotlin", "TutorialPoint.com", 5)
   println("Name of the Book is--"+book.name) // "Kotlin"
   println("Puclisher Name--"+book.publisher) //
"TutorialPoint.com"
   println("Review of the book is--"+book.reviewScore) // 5
   book.reviewScore = 7
   println("Printing all the info all together--
"+book.toString())
   //using inbuilt function of the data class

   println("Example of the hashCode function--
"+book.hashCode())
}

data class Book(val name: String, val publisher: String, var
reviewScore: Int)
```

El código anterior generará el siguiente resultado en el navegador, donde hemos creado una clase de datos para contener algunos de los datos, y desde la función principal hemos accedido a todos sus miembros de datos.

```
Name of the Book is--"Kotlin"
Puclisher Name--"TutorialPoint.com"
Review of the book is--5
Printing all the info all together--(name-Kotlin, publisher-
TutorialPoint.com, reviewScore-7)
Example of the hashCode function---1753517245
```

Kotlin - Clase sellada

En este capítulo, aprenderemos sobre otro tipo de clase llamada clase "Sellada". Este tipo de clase se utiliza para representar una jerarquía de clases restringida. Sellado permite a los desarrolladores mantener un tipo de datos de un tipo predefinido. Para hacer una clase sellada, necesitamos usar la palabra clave "sellado" como un modificador de esa clase. Una clase sellada puede tener su propia subclase, pero todas esas subclases deben declararse dentro del mismo archivo Kotlin junto con la clase sellada. En el siguiente ejemplo, veremos cómo usar una clase sellada.

```
sealed class MyExample {
   class OP1 : MyExample() // MyExmaple class can be of two
types only
   class OP2 : MyExample()
```

```
}
fun main(args: Array<String>) {
   val obj: MyExample = MyExample.OP2()

   val output = when (obj) { // defining the object of the
class depending on the inuputs
      is MyExample.OP1 -> "Option One has been chosen"
      is MyExample.OP2 -> "option Two has been chosen"
   }

   println(output)
}
```

En el ejemplo anterior, tenemos una clase sellada llamada "MyExample", que puede ser de dos tipos solamente: uno es "OP1" y otro es "OP2". En la clase principal, estamos creando un objeto en nuestra clase y asignando su tipo en tiempo de ejecución. Ahora, como esta clase "MyExample" está sellada, podemos aplicar la cláusula "when" en tiempo de ejecución para implementar la salida final.

En la clase sellada, no necesitamos usar ninguna declaración innecesaria "else" para completar el código. El código anterior generará el siguiente resultado en el navegador.

```
option Two has been chosen
```

Kotlin - Genéricos

Al igual que Java, Kotlin proporciona un orden superior de escritura variable denominado Genéricos. En este capítulo, aprenderemos cómo Kotlin implementa Generics y cómo, como desarrollador, podemos usar esas funcionalidades proporcionadas dentro de la biblioteca de genéricos. En cuanto a la implementación, los genéricos son bastante similares a Java, pero el desarrollador de Kotlin ha introducido dos nuevas palabras clave **"fuera"** y **"dentro"** para que los códigos de Kotlin sean más legibles y fáciles para el desarrollador.

En Kotlin, una clase y un tipo son conceptos totalmente diferentes. Según el ejemplo, List es una clase en Kotlin, mientras que List <String> es un tipo en Kotlin. El siguiente ejemplo muestra cómo se implementan los genéricos en Kotlin.

```
fun main(args: Array<String>) {
   val integer: Int = 1
   val number: Number = integer
   print(number)
}
```

En el código anterior, hemos declarado un "entero" y luego hemos asignado esa variable a una variable numérica. Esto es posible porque "Int" es una subclase de la clase Number, por lo tanto, la conversión de tipo ocurre automáticamente en tiempo de ejecución y produce la salida como "1".

Aprendamos algo más sobre genéricos en Kotlin. Es mejor optar por el tipo de datos genéricos siempre que no estemos seguros del tipo de datos que vamos

a utilizar en la aplicación. En general, en Kotlin los genéricos se definen por **<T>** donde "T" significa plantilla, que puede ser determinada dinámicamente por el cumplidor de Kotlin. En el siguiente ejemplo, veremos cómo usar tipos de datos genéricos en el lenguaje de programación Kotlin.

```
fun main(args: Array<String>) {
   var objet = genericsExample<String>("JAVA")
   var objet1 = genericsExample<Int>(10)
}
class genericsExample<T>(input:T) {
   init {
      println("I am getting called with the value "+input)
   }
}
```

En el fragmento de código anterior, estamos creando una clase con tipo de retorno genérico, que se representa como **<T>** . Eche un vistazo al método principal, donde hemos definido dinámicamente su valor en la ejecución al probar el tipo de valor, al crear el objeto de esta clase. Así es como el compilador Kotlin interpreta los genéricos. Obtendremos el siguiente resultado en el navegador, una vez que ejecutemos este código en nuestro campo de codificación.

```
I am getting called with the value JAVA
I am getting called with the value 10
```

Cuando queremos asignar el tipo genérico a cualquiera de sus súper tipos, entonces necesitamos usar la palabra clave "out", y cuando queremos asignar el tipo genérico a cualquiera de sus subtipos, entonces necesitamos usar "in" palabra clave. En el siguiente ejemplo, usaremos la palabra clave "out". Del mismo modo, puede intentar usar la palabra clave "in".

```
fun main(args: Array<String>) {
   var objet1 = genericsExample<Int>(10)
   var object2 = genericsExample<Double>(10.00)
   println(objet1)
   println(object2)
}
class genericsExample<out T>(input:T) {
   init {
      println("I am getting called with the value "+input)
   }
}
```

El código anterior producirá el siguiente resultado en el navegador.

```
I am getting called with the value 10
I am getting called with the value 10.0
genericsExample@28d93b30
genericsExample@1b6d3586
```

Kotlin admite el patrón de diseño de **"delegación"** al introducir una nueva palabra clave **"por"** . Usando esta metodología de palabra clave o delegación, Kotlin permite que la clase derivada acceda a todos los métodos públicos implementados de una interfaz a través de un objeto específico. El siguiente ejemplo demuestra cómo sucede esto en Kotlin.

```
interface Base {
   fun printMe() //abstract method
}
class BaseImpl(val x: Int) : Base {
   override fun printMe() { println(x) }   //implementation
of the method
}
class Derived(b: Base) : Base by b  // delegating the public
method on the object b

fun main(args: Array<String>) {
   val b = BaseImpl(10)
   Derived(b).printMe() // prints 10 :: accessing the
printMe() method
}
```

En el ejemplo, tenemos una interfaz "Base" con su método abstracto llamado "printme ()". En la clase BaseImpl, estamos implementando este "printme ()" y luego desde otra clase estamos usando esta implementación usando la palabra clave "by".

El código anterior generará el siguiente resultado en el navegador.

```
10
```

Delegación de propiedad

En la sección anterior, hemos aprendido sobre el patrón de diseño de delegación usando la palabra clave "by". En esta sección, aprenderemos sobre la delegación de propiedades utilizando algunos métodos estándar mencionados en la biblioteca de Kotlin.

Delegar significa pasar la responsabilidad a otra clase o método. Cuando una propiedad ya está declarada en algunos lugares, deberíamos reutilizar el mismo código para inicializarla. En los siguientes ejemplos, utilizaremos alguna metodología de delegación estándar proporcionada por Kotlin y alguna función de biblioteca estándar mientras implementamos la delegación en nuestros ejemplos.

Usando Lazy ()

Lazy es una función lambda que toma una propiedad como entrada y, a cambio, proporciona una instancia de **Lazy <T>** , donde <T> es básicamente el tipo de propiedades que está utilizando. Echemos un vistazo a lo siguiente para comprender cómo funciona.

```
val myVar: String by lazy {
   "Hello"
}
fun main(args: Array<String>) {
   println(myVar +" My dear friend")
}
```

En el fragmento de código anterior, estamos pasando una variable "myVar" a la función Lazy, que a cambio asigna el valor a su objeto y lo devuelve a la función principal. A continuación se muestra la salida en el navegador.

```
Hello My dear friend
```

Delegetion.Observable ()

Observable () toma dos argumentos para inicializar el objeto y devuelve el mismo a la función llamada. En el siguiente ejemplo, veremos cómo usar el método Observable () para implementar la delegación.

```
import kotlin.properties.Delegates
class User {
   var name: String by Delegates.observable("Welcome to
Postparaprogramadores.com") {
      prop, old, new ->
      println("$old -> $new")
   }
}
fun main(args: Array<String>) {
   val user = User()
   user.name = "first"
   user.name = "second"
}
```

El código anterior generará el siguiente resultado en el navegador.

```
first -> second
```

En general, la sintaxis es la expresión después de que se delega la palabra clave "by". Los métodos **get () y set ()** de la variable **p** se delegarán a sus métodos **getValue ()** y **setValue ()** definidos en la clase Delegate.

```
class Example {
   var p: String by Delegate()
}
```

Para el fragmento de código anterior, la siguiente es la clase de delegado que necesitamos generar para asignar el valor en la variable **p** .

```
class Delegate {
   operator fun getValue(thisRef: Any?, property:
KProperty<*>): String {
      return "$thisRef, thank you for delegating
'${property.name}' to me!"
```

```
    }
    operator fun setValue(thisRef: Any?, property:
KProperty<*>, value: String) {
        println("$value has been assigned to '${property.name}
in $thisRef.'")
    }
}
```

Durante la lectura, se llamará al método getValue () y al establecer la variable, se llamará al método setValue ().

Kotlin - Funciones

Kotlin es un lenguaje de tipo estático, por lo tanto, las funciones juegan un gran papel en él. Estamos bastante familiarizados con la función, ya que estamos usando la función en todos los ejemplos. La función se declara con la palabra clave "diversión". Al igual que cualquier otra OOP, también necesita un tipo de retorno y una lista de argumentos de opción.

En el siguiente ejemplo, estamos definiendo una función llamada MyFunction y desde la función principal estamos llamando a esta función y pasando algún argumento.

```
fun main(args: Array<String>) {
    println(MyFunction("postparaprogramadores.com"))
}
fun MyFunction(x: String): String {
    var c:String  = "Hey!! Welcome To ---"
    return (c+x)
}
```

El código anterior generará el siguiente resultado en el navegador.

```
Hey!! Welcome To ---postparaprogramadores.com
```

La función debe declararse de la siguiente manera:

```
fun <nameOfFunction>(<argument>:<argumentType>):<ReturnType>
```

Los siguientes son algunos de los diferentes tipos de funciones disponibles en Kotlin.

Función lambda

Lambda es una función de alto nivel que reduce drásticamente el código de la placa de la caldera al declarar una función y definirla. Kotlin te permite definir tu propia lambda. En Kotlin, puede declarar su lambda y pasar esa lambda a una función.

Eche un vistazo al siguiente ejemplo.

```
fun main(args: Array<String>) {
    val mylambda :(String)->Unit  = {s:String->print(s)}
```

```
    val v:String = "Postparaprogramadores.com"
    mylambda(v)
}
```

En el código anterior, hemos creado nuestra propia lambda conocida como "mylambda" y hemos pasado una variable a esta lambda, que es de tipo String y contiene un valor "Postparaprogramadores.com".

El código anterior generará el siguiente resultado en el navegador.

```
Postparaprogramadores.com
```

Función en línea

El ejemplo anterior muestra lo básico de la expresión lambda que podemos usar en la aplicación Kotlin. Ahora, podemos pasar un lambda a otra función para obtener nuestra salida, lo que hace que la función de llamada sea una función en línea.

Eche un vistazo al siguiente ejemplo.

```
fun main(args: Array<String>) {
    val mylambda:(String)->Unit  = {s:String->print(s)}
    val v:String = "Postparaprogramadores.com"
    myFun(v,mylambda) //passing lambda as a parameter of
another function
}
fun myFun(a :String, action: (String)->Unit) { //passing
lambda
    print("Heyyy!!!")
    action(a)// call to lambda function
}
```

El código anterior generará el siguiente resultado en el navegador. Usando la función en línea, hemos pasado una lambda como parámetro. Cualquier otra función puede convertirse en una función en línea utilizando la palabra clave "en línea".

```
Heyyy!!!Postparaprogramadores.com
```

Kotlin - Declaraciones de desestructuración

Kotlin contiene muchas características de otros lenguajes de programación. Le permite declarar múltiples variables a la vez. Esta técnica se llama declaración de desestructuración.

La siguiente es la sintaxis básica de la declaración de desestructuración.

```
val (name, age) = person
```

En la sintaxis anterior, hemos creado un objeto y los hemos definido todos juntos en una sola declaración. Más tarde, podemos usarlos de la siguiente manera.

```
println(name)
```

```
println(age)
```

Ahora, veamos cómo podemos usar lo mismo en nuestra aplicación de la vida real. Considere el siguiente ejemplo donde estamos creando una clase de Estudiante con algunos atributos y luego los usaremos para imprimir los valores de los objetos.

```
fun main(args: Array<String>) {
   val s = Student("Postparaprogramadores.com","Kotlin")
   val (name,subject) = s
   println("You are learning "+subject+" from "+name)
}
data class Student( val a :String,val b: String ){
   var name:String = a
   var subject:String = b
}
```

El código anterior generará el siguiente resultado en el navegador.

```
You are learning Kotlin from Postparaprogramadores.com
```

Kotlin - Manejo de excepciones

El manejo de excepciones es una parte muy importante de un lenguaje de programación. Esta técnica restringe nuestra aplicación de generar la salida incorrecta en tiempo de ejecución. En este capítulo, aprenderemos cómo manejar la excepción de tiempo de ejecución en Kotlin. Las excepciones en Kotlin son bastante similares a las excepciones en Java. Todas las excepciones son descendientes de la clase "Throwable". El siguiente ejemplo muestra cómo usar la técnica de manejo de excepciones en Kotlin.

```
fun main(args: Array<String>) {
   try {
      val myVar:Int = 12;
      val v:String = "Postparaprogramadores.com";
      v.toInt();
   } catch(e:Exception) {
      e.printStackTrace();
   } finally {
      println("Exception Handeling in Kotlin");
   }
}
```

En el fragmento de código anterior, hemos declarado una Cadena y luego hemos vinculado esa cadena al entero, que en realidad es una excepción en tiempo de ejecución. Por lo tanto, obtendremos el siguiente resultado en el navegador.

```
val myVar:Int = 12;
Exception Handeling in Kotlin
```

Nota - Al igual que Java, Kotlin también ejecuta el bloque finalmente después de ejecutar el bloque catch.