scalac

Jorge Vasquez

Introducción a la Programación con Efectos Funcionales usando ZIO

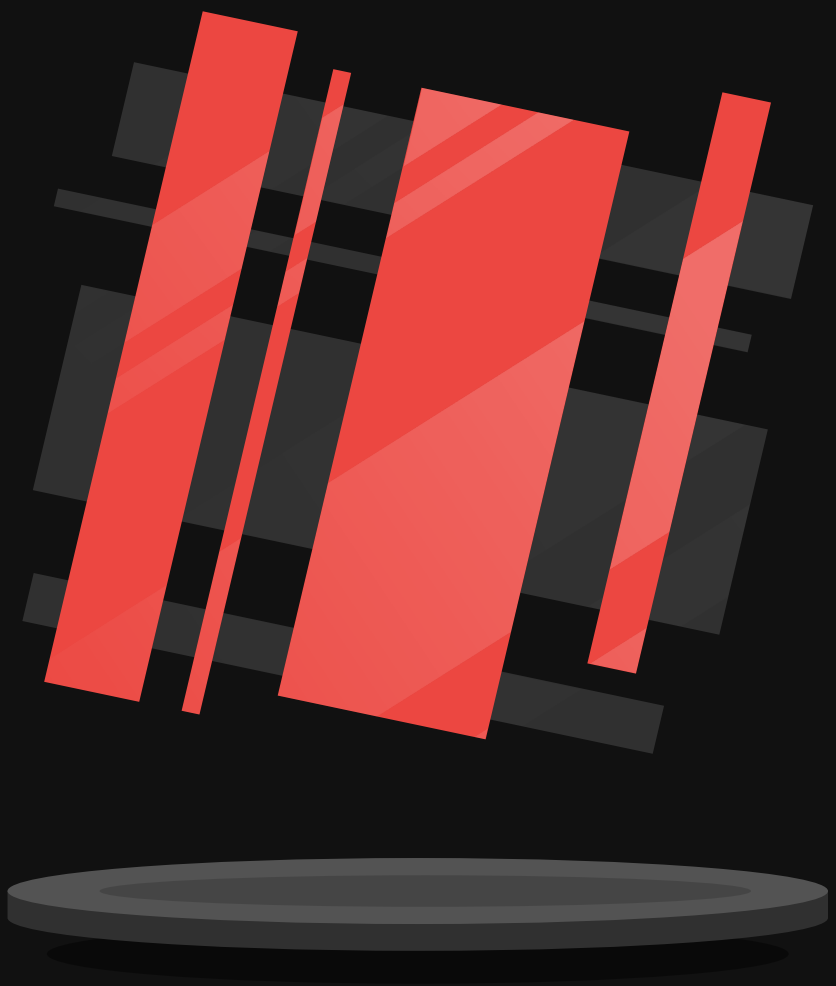

Tabla de contenidos

Tabla de contenidos

Introducción

El paradigma de programación funcional es muchas veces evitado por desarrolladores alrededor del mundo, quienes prefieren seguir desarrollando aplicaciones usando el paradigma de programación orientada a objetos, el cual es bien conocido y ampliamente usado. Este rechazo hacia la programación funcional ocurre ya sea por falta de conocimiento o porque los desarrolladores tienden a pensar que es algo difícil y útil solamente en ámbitos académicos, puesto que hay muchos conceptos nuevos usualmente explicados empleando muchos términos matemáticos. Nada más alejado de la realidad: la programación funcional ayuda a resolver muchos de los problemas que se presentan al desarrollar software, especialmente programas que requieren altos niveles de asincronicidad y concurrencia, y si es explicada adecuadamente no tiene por qué ser algo difícil de aprender.

El lenguaje Scala combina lo mejor de los mundos de la programación funcional y orientada a objetos, y provee completa interoperabilidad con el gran ecosistema de Java. Además, dentro del ecosistema de Scala, existen librerías como ZIO, que permiten a los desarrolladores ser altamente productivos y crear aplicaciones modernas, altamente performantes y concurrentes, aprovechando los principios de la programación funcional sin que ello represente una enorme barrera de entrada.

En este artículo trataremos los principios de la programación funcional, y a continuación veremos cómo, gracias a Scala y ZIO, podemos crear aplicaciones que resuelvan problemas del mundo real. Como ejemplo ilustrativo, implementaremos el juego del ahorcado.

Programación funcional 101

¿Qué es la programación funcional?

La **programación funcional** es un paradigma de programación, donde los programas son una composición de **funciones puras.** Esto representa una diferencia fundamental con la programación orientada a objetos, donde los programas son secuencias de instrucciones, que usualmente operan sobre algún estado mutable. Dichos programas son organizados usando funciones también, pero éstas no son funciones en el sentido matemático, porque no cumplen con algunas características fundamentales.

Una función pura debe ser total

Primeramente, **una función debe ser total**, eso significa que para cada entrada que se provea a la función debe haber una salida definida. Por ejemplo, la siguiente función para dividir dos enteros no es total:

```
def divide(a: Int, b: Int): Int = a / b
```

Para responder a la pregunta: ¿por qué esta función no es total? consideremos qué pasa si tratamos de dividir por cero:

```
divide(5, 0)

// java.lang.ArithmeticException: / by zero
```

La división por cero no está definida y Java maneja esto lanzando una excepción. Eso significa que la función `divide` no es total porque no retorna ninguna salida en el caso donde `b = 0`.

Algunas cosas importantes que podemos resaltar aquí:

- Si miramos el encabezado de esta función, podemos darnos cuenta de que dice una mentira: que para cada par de enteros `a` y `b`, esta función siempre retornará otro entero, lo cual ya hemos visto que no es cierto para todos los casos.
- Esto significa que cada vez que llamemos a la función `divide` en alguna parte de nuestra aplicación tendremos que ser muy cuidadosos, dado que nunca podemos estar completamente seguros de que la función vaya a retornar un entero.
- Si nos olvidamos de considerar el caso excepcional cuando llamemos a `divide`, al momento de ejecutar nuestra aplicación ésta lanzará excepciones y dejará de trabajar como debería. Desafortunadamente, el compilador nada podrá hacer para ayudarnos a evitar este tipo de situación, el código compilará y recién veremos los problemas en tiempo de ejecución.

Entonces, ¿cómo podemos solucionar este problema? Observemos esta definición alternativa de la función `divide`:

```
def divide(a: Int, b: Int): Option[Int] =
  if (b != 0) Some(a/b) else None
```

En este caso, la función `divide` retorna `Option[Int]` en vez de `Int`, de tal manera que cuando `b = 0`, la función retorna `None` en vez de lanzar una excepción. Así es como hemos transformado una función parcial en una función total, y gracias a eso tenemos algunos beneficios:

- El encabezado de la función claramente comunica que algunas entradas no son manejadas porque retorna un `Option[Int]`.
- Cuando llamemos a la función `divide` en alguna parte de nuestra aplicación, el compilador nos obligará a considerar el caso en el que el resultado no está definido. Si no lo hacemos, habrá errores de compilación, eso significa que el compilador nos podrá ayudar a evitar muchos *bugs* en tiempo de compilación y que no aparecerán en tiempo de ejecución.

Una función pura debe ser determinística y debe depender sólo de sus entradas

La segunda característica de una función es que debe ser **determinística y debe depender sólo de sus entradas**, eso significa que para cada entrada que se provea a la función siempre se debe obtener la misma salida, sin importar cuántas veces la función sea llamada. Por ejemplo, la siguiente función para generar enteros aleatorios no es determinística:

```
def generateRandomInt(): Int = (new scala.util.Random).nextInt
```

Para demostrar por qué esta función no es determinística, consideremos qué pasa la primera vez que llamamos a la función:

```
generateRandomInt() // Resultado: -272770531
```

Y luego, qué pasa si llamamos otra vez a la función:

```
generateRandomInt() // Resultado: 217937820
```

¡Obtenemos resultados diferentes! Claramente esta función no es determinística, y su encabezado es engañoso otra vez, porque sugiere que no depende de ninguna entrada para producir una salida, cuando en realidad hay una dependencia escondida hacia un objeto `scala.util.Random`. Esto podría causar problemas, porque nunca podemos estar realmente seguros de cómo la función `generateRandomInt` se va a comportar, haciéndola difícil de testear.

Ahora, veamos una definición alternativa. Para ello usaremos un generador de números aleatorios personalizado, basado en un ejemplo presentado en el libro Functional Programming in Scala:

```
final case class RNG(seed: Long) {
  def nextInt: (Int, RNG) = {
    val newSeed = (seed * 0x5DEECE66DL + 0xBL) & 0xFFFFFFFFFFFFL
    val nextRNG = RNG(newSeed)
    val n       = (newSeed >>> 16).toInt
    (n, nextRNG)
  }
}

def generateRandomInt(random: RNG): (Int, RNG) = random.nextInt
```

Esta nueva versión de la función `generateRandomInt` es determinística: no importa cuántas veces se la llame, siempre obtendremos la misma salida para las misma entrada y el encabezado ahora muestra claramente la dependencia hacia la variable `random`. Por ejemplo:

```
val random        = RNG(10)
val (n1, random1) = generateRandomInt(random) // n1 = 3847489, random1 = RNG(252149039181)
val (n2, random2) = generateRandomInt(random) // n2 = 3847489, random2 = RNG(252149039181)
```

Si queremos generar un nuevo entero, debemos proveer una entrada distinta:

```
val (n3, random3) = generateRandomInt(random2) // n3 = 1334288366, random3 = RNG(87443922374356)
```

Una función pura no debe tener efectos colaterales

Finalmente, una función no debe tener efectos colaterales. Algunos ejemplos de efectos colaterales son los siguientes:

- mutaciones de memoria,
- interacciones con el mundo exterior, tales como:
    - imprimir mensajes en la consola.
    - llamar a una API externa.
    - consultar a una base de datos.

Esto significa que una función pura sólo puede trabajar con valores inmutables y sólo puede retornar una salida para una entrada correspondiente, nada más.

Por ejemplo, la siguiente función `increment` no es pura porque trabaja con una variable mutable `a`:

```
var a = 0;
def increment(inc: Int): Int = {
  a += inc
  a
}
```

Y la siguiente función tampoco es pura porque imprime un mensaje en la consola:

```
def add(a: Int, b: Int): Int = {
  println(s"Adding two integers: $a and $b")
  a + b
}
```

¿Cuáles son las diferencias entre la programación funcional y la programación orientada a objetos?

La siguiente tabla resume las mayores diferencias entre la programación funcional y la programación orientada a objetos:

	Programación funcional	**Programación orientada a objetos**
Variables	inmutables	mutables
Modelo de programación	declarativa	imperativa
Se enfoca en...	“qué” se hace	“cómo” se hace
Paralelismo	adecuada para programación paralela	no adecuada para programación paralela
Efectos colaterales	Las funciones no pueden producir efectos colaterales	Los métodos pueden producir efectos colaterales
Iteraciones	recursividad	ciclos (`for`, `while`)
Estado de la aplicación	fluye a través de funciones puras	usualmente compartido por varios objetos
Elementos clave	valores inmutables y funciones	objetos y métodos

¿Cuáles son los beneficios de la programación funcional?

Por varios motivos, la programación funcional aún parece compleja para muchos. Pero, si miramos más de cerca sus beneficios, podemos cambiar nuestra forma de pensar.

Para empezar, adoptar este paradigma de programación nos ayuda a descomponer cada aplicación en piezas más pequeñas y simples que son confiables y fáciles de entender. Esto es porque un código fuente funcional es a menudo más **conciso**, **predecible** y **fácil de testear.** Pero, ¿cómo podemos asegurar esto?

- Dado que las funciones puras no dependen de ningún estado y dependen solamente de sus entradas, son mucho más fáciles de comprender. Es decir, para entender lo que una función hace, no necesitamos buscar otras partes del código que podrían afectar su funcionamiento, esto se conoce como **razonamiento local.**
- Dado que el código tiende a ser más **conciso**, también tiende a tener **menos *bugs*.**
- El proceso de *testing* y *debugging* se vuelve mucho más fácil con funciones que sólo reciben datos de entrada y producen salidas.
- Dado que las funciones puras son determinísticas, **las aplicaciones se comportan más predeciblemente.**
- La programación funcional nos permite **escribir programas paralelos correctos,** dado que no existe la posibilidad de tener estado mutable, por lo tanto **es imposible que se produzcan típicos problemas de concurrencia, como condiciones de carrera.**

Dado que Scala soporta el paradigma de programación funcional, estos beneficios también se aplican al lenguaje como tal. Como resultado, cada vez más compañías usan Scala, incluyendo gigantes como LinkedIn, Twitter y Netflix. Además, este año Scala se encuentra en el top 10 de los lenguajes que los desarrolladores quieren aprender.

Pero… ¡Las aplicaciones en la vida real necesitan ejecutar efectos colaterales!

Ahora que ya sabemos que la programación funcional trata de programar con **funciones puras**, y que las funciones puras **no pueden producir efectos colaterales**, surgen varias preguntas lógicas:

- ¿Cómo podemos entonces tener una aplicación escrita usando programación funcional, y que a la vez interactúe con servicios externos, como bases de datos o APIs de terceros? ¿No es eso algo contradictorio?
- ¿Significa esto que la programación funcional no se puede usar en aplicaciones reales, sino solamente en ámbitos académicos?

La respuesta a estas preguntas es la siguiente: **Sí, podemos usar programación funcional en aplicaciones reales, y no solamente en ámbitos académicos.** Para que nuestras aplicaciones sean capaces de interactuar con servicios externos, podemos hacer lo siguiente: en vez de escribir funciones que interactúan con el mundo exterior, escribimos **funciones que describan interacciones con el mundo exterior**, las cuales sean ejecutadas solamente en un punto específico de nuestra aplicación (llamado habitualmente como el **fin del mundo**) por ejemplo la función `main`.

Si lo pensamos detenidamente, **estas descripciones de interacciones con el mundo exterior son simplemente valores inmutables** que pueden servir como entradas y salidas de funciones puras, de esta forma no estaríamos violando un principio básico de la programación funcional: no producir efectos colaterales.

¿Y qué es el antes mencionado **fin del mundo**? Bueno, el fin del mundo es simplemente un punto específico de nuestra aplicación **donde el mundo funcional termina**, y donde esas descripciones de interacciones con el mundo exterior se ejecutan, usualmente tan tarde como sea posible, preferiblemente en el borde mismo de nuestra aplicación que es la función `main`. De esta forma logramos que toda nuestra aplicación esté escrita siguiendo el estilo funcional, pero que al mismo tiempo sea capaz de realizar tareas útiles.

Ahora que sabemos esto surge una nueva pregunta: ¿cómo podemos escribir nuestras aplicaciones de tal manera que todas nuestras funciones no ejecuten efectos colaterales, sino que solamente construyan descripciones de lo que se quiere hacer? Entonces, aquí es donde entra una librería muy poderosa que nos puede ayudar con esta tarea: **la librería ZIO.**

Introducción a la librería ZIO

¿Para qué sirve la librería ZIO?

ZIO es una librería que nos permite construir **aplicaciones modernas**, que sean **asíncronas, concurrentes, resilientes, eficientes, fáciles de entender y testear,** usando los principios de la programación funcional.

¿Por qué decimos que ZIO nos permite construir aplicaciones que sean **fáciles de entender y testear**? Porque nos ayuda a construir aplicaciones de cualquier complejidad en forma

incremental, a través de la combinación de descripciones de interacciones con el mundo exterior. Por cierto, dichas descripciones son llamadas **efectos funcionales.**

¿Por qué decimos que ZIO nos permite construir aplicaciones que sean **resilientes**? Porque ZIO aprovecha al máximo el sistema de tipado de Scala, de tal manera que puede capturar más *bugs* en tiempo de compilación, en vez de en tiempo de ejecución. Esto es genial porque, con sólo mirar el encabezado de una función, podemos saber:

- Si tiene **dependencias externas.**
- **Si puede fallar o no**, y además con qué tipo de errores puede fallar.
- **Si puede terminar exitosamente o no**, y además cuál es el tipo de dato que retorna al terminar.

Y finalmente, ¿por qué decimos que ZIO nos permite construir aplicaciones que sean **asíncronas y concurrentes**? Porque ZIO nos da superpoderes para trabajar con programación asíncrona y concurrente, usando un **modelo basado en fibras**, el cual es mucho más eficiente que un modelo basado en hilos. No entraremos en mucho detalle acerca de este aspecto en el presente artículo, sin embargo cabe mencionar que justamente en esta área es donde ZIO brilla, permitiéndonos construir aplicaciones realmente performantes.

El tipo de datos ZIO

El tipo de datos más importante en la librería ZIO (y también el bloque constructivo básico de cualquier aplicación basada en esta librería), es también llamado `ZIO`:

```
ZIO[-R, +E, +A]
```

El tipo de datos `ZIO` es un **efecto funcional**, lo que significa que es un valor inmutable que contiene una descripción de una serie de interacciones con el mundo exterior (consultas a bases de datos, llamadas a APIs de terceros, etc.). Un buen modelo mental del tipo de datos `ZIO` es el siguiente:

```
R => Either[E, A]
```

Eso significa que un efecto `ZIO`:

- Necesita un contexto de tipo `R` para ejecutarse (este contexto puede ser cualquier cosa: una conexión a una base de datos, un cliente REST, un objeto de configuración, etc.).
- Puede fallar con un error de tipo `E` o puede terminar exitosamente, retornando un valor de tipo `A`.

Alias comunes para el tipo de datos ZIO

Vale la pena mencionar que la librería ZIO provee algunos alias para el tipo de datos `ZIO`, los cuales son muy útiles para representar casos comunes:

- `Task[+A] = ZIO[Any, Throwable, A]`: Esto significa que un `Task[A]` es un efecto `ZIO` que:
  - No requiere un contexto para ejecutarse (esa es la razón por la que el tipo `R` es reemplazado por `Any`, lo cual significa que el efecto correrá sin importar qué contexto le proveamos)
  - Puede fallar con un `Throwable`.
  - Puede terminar exitosamente con un valor de tipo `A`.
- `UIO[+A] = ZIO[Any, Nothing, A]`: Esto significa que un `UIO[A]` es un efecto `ZIO` que:
  - No requiere un contexto para ejecutarse.
  - No puede fallar.
  - Puede terminar exitosamente con un valor de tipo `A`.
- `RIO[-R, +A] = ZIO[R, Throwable, A]`: Esto significa que un `RIO[R, A]` es un efecto `ZIO` que:
  - Requiere un contexto `R` para ejecutarse.
  - Puede fallar con un `Throwable`.
  - Puede terminar exitosamente con un valor de tipo `A`.
- `IO[+E, +A] = ZIO[Any, E, A]`: Esto significa que un `IO[E, A]` es un efecto `ZIO` que:
  - No requiere un contexto para ejecutarse.
  - Puede fallar con un error de tipo `E`.
  - Puede terminar exitosamente con un valor de tipo `A`.
- `URIO[-R, +A] = ZIO[R, Nothing, A]`: Esto significa que un `URIO[R, A]` es un efecto `ZIO` que:
  - Requiere un contexto `R` para ejecutarse.
  - No puede fallar.
  - Puede terminar exitosamente con un valor de tipo `A`.

Implementación del juego del ahorcado usando ZIO

A continuación implementaremos el juego del ahorcado usando ZIO, como ejemplo ilustrativo de cómo podemos desarrollar aplicaciones puramente funcionales en Scala con la ayuda de esta librería.

Como referencia, puedes ver el repositorio Github con el código completo en este link. Por cierto, la implementación del juego está basada en este repositorio creado por John De Goes, autor de la librería ZIO, donde puedes encontrar varios ejercicios interesantes.

Diseño y requerimientos

El juego del ahorcado consiste en tomar una palabra al azar y darle al jugador la posibilidad de que adivine letras hasta completar la palabra. Las reglas son las siguientes:

- El jugador tiene 6 intentos para arriesgar letras.
- Por cada letra incorrecta, se resta un intento.
- Cuando el jugador se queda sin intentos, pierde. Si adivina todas las letras, gana.

Entonces, nuestra implementación del juego del ahorcado debería funcionar de la siguiente manera:

- Al comenzar el juego, se deberá **solicitar al jugador su nombre,** el cual obviamente no debe ser vacío. Si fuera así, se debe mostrar un mensaje de error y volver a solicitar el nombre.
- A continuación, la aplicación deberá **seleccionar de forma aleatoria**, dentro de un diccionario de palabras predefinido, **la palabra que el jugador deberá adivinar.**
- Luego se deberá **mostrar por consola el estado inicial del juego**, que básicamente consiste en una horca, una serie de guiones que representan la cantidad de letras de la palabra a adivinar y las letras que ya fueron intentadas por el jugador, que obviamente será ninguna al principio del juego.
- Entonces, se deberá **pedir al jugador que adivine una letra y la escriba por consola.** Obviamente el caracter introducido debe ser una letra válida, sin importar si es mayúscula o minúscula:
    - **Si el caracter introducido es inválido**, se deberá mostrar un mensaje de error, y se deberá volver a solicitar una letra al jugador.
    - **Si el jugador introdujo una letra válida pero no aparece en la palabra**, pierde un intento, se muestra un mensaje adecuado por consola y se actualiza el estado del juego, añadiendo la letra recientemente intentada por el jugador a la lista de intentos, y dibujando la cabeza del ahorcado. Por cierto, las siguientes veces que el jugador se equivoque de letra se irá mostrando, en orden: el tronco, brazo derecho, brazo izquierdo, pierna derecha y pierna izquierda del ahorcado.
    - **Si el usuario introdujo una letra válida y ésta aparece en la palabra**, se muestra un mensaje adecuado por consola y se actualiza el estado del juego, añadiendo la letra recientemente intentada por el usuario a la lista de intentos, y descubriendo en la palabra oculta los lugares donde aparece la letra adivinada.
- El paso anterior se repite hasta que el usuario adivine la palabra completa o se quede sin intentos.

- **Si el jugador gana** se muestra un mensaje de felicitación.
- **Si el jugador pierde** se muestra un mensaje indicando cuál era la palabra a adivinar.

Creando la estructura base de la aplicación

Definiremos nuestra aplicación como un proyecto `sbt`, el archivo `build.sbt` contendrá las dependencias de nuestro proyecto:

```
val scalaVer = "2.13.4"

val zioVersion = "1.0.4"

lazy val compileDependencies = Seq(
  "dev.zio" %% "zio" % zioVersion
) map (_ % Compile)

lazy val settings = Seq(
  name := "zio-hangman",
  version := "1.0.0",
  scalaVersion := scalaVer,
  libraryDependencies ++= compileDependencies
)

lazy val root = (project in file("."))
  .settings(settings)
```

Como podemos ver, trabajaremos con `Scala 2.13.4` y con `ZIO 1.0.4`.

Creando el modelo de dominio, al estilo funcional

Primeramente, debemos definir nuestro modelo de dominio. Para ello definiremos algunas clases dentro del archivo `com/example/package.scala`.

Como primer paso, podemos definir una clase `Name` que represente el nombre del jugador:

```
final case class Name(name: String)
```

Como podemos ver, definimos `Name` como una *clase case* en vez de definirla simplemente como una clase normal. Usar *clases case* nos da muchos beneficios, como por ejemplo:

- **Inmutabilidad** por defecto.

- Se genera un método `apply` de forma automática, lo cual nos permite construir objetos sin necesidad de usar la palabra reservada `new`.
- Se genera un método `unapply` de forma automática, con lo cual podemos usar la clase en expresiones de reconocimiento de patrones (*pattern matching* en inglés).
- Se genera un método `copy` de forma automática, el cual nos permite realizar copias de objetos y actualizar ciertos campos al mismo tiempo.

Además, definimos la clase como `final` para que no pueda ser extendida por otras clases. Por otro lado, vemos que la clase `Name` simplemente encapsula un `String` que representa el nombre del usuario. Podríamos dejar este modelo así como está, sin embargo tiene un problema, y es que nos permite crear objetos `Name` con un `String` vacío, lo cual es algo indeseado. Entonces, para impedir esto, podemos aplicar una técnica de diseño funcional llamada **constructor inteligente (*smart constructor* en inglés)**, que simplemente consiste en definir un método en el objeto acompañante de la clase `Name` que nos permita hacer las validaciones respectivas antes de crear una instancia. Entonces, tendríamos algo así:

```
final case class Name(name: String)
object Name {
  def make(name: String): Option[Name] =
    if (!name.isEmpty) Some(Name(name)) else None
}
```

Como podemos ver, el método `make` corresponde a nuestro constructor inteligente, y éste verifica si el `String` recibido es vacío o no. Si no es vacío, devuelve un nuevo `Name`, pero encapsulado dentro de un `Some`, y si es vacío retorna un `None`. Lo importante a resaltar aquí es que el método `make` es una función pura, porque es total, determinística, la salida sólo depende del `String` de entrada y no produce efectos colaterales.

Ahora bien, hay algunas cosas que podemos mejorar en nuestra implementación. Por ejemplo, aún es posible llamar directamente al constructor de `Name` con `Strings` vacíos, para evitar esto podemos convertirlo en privado:

```
final case class private Name(name: String)
object Name {
  def make(name: String): Option[Name] =
    if (!name.isEmpty) Some(Name(name)) else None
}
```

De esta forma, la única forma de construir objetos `Name` es llamando al método `Name.make`, al menos eso es lo que esperamos, sin embargo aún podemos construir objetos `Name` con

`Strings` vacíos usando el método `Name.apply`. Por otro lado, hay un otro problema que debemos resolver, y es que gracias al método `copy` que es automáticamente generado para las *clases case*, después de haber creado un objeto `Name` válido usando `Name.make`, podríamos luego generar un objeto inválido generando una copia con un `String` vacío, nada nos impediría hacer eso. Para evitar esto, podemos definir la clase como `sealed abstract` en vez de `final`, con lo cual no se generará los métodos `apply` y `copy`:

```
sealed abstract case class Name private (name: String)
object Name {
  def make(name: String): Option[Name] =
    if (!name.isEmpty) Some(new Name(name) {}) else None
}
```

De esta forma, estamos completamente seguros que cualquier objeto `Name` en nuestra aplicación siempre será válido.

Similarmente, podemos definir una *clase case* llamada `Guess`, que represente una letra adivinada por el jugador:

```
sealed abstract case class Guess private (char: Char)
object Guess {
  def make(str: String): Option[Guess] =
    Some(str.toList).collect {
      case c :: Nil if c.isLetter => new Guess(c.toLower) {}
    }}
```

Como podemos ver, hemos usado la misma técnica de definir un constructor inteligente, el cual recibe un `String` introducido por el jugador, y verifica si solamente consiste de una letra. Si ése no es el caso (por ejemplo cuando el `String` introducido por el jugador consiste de varios caracteres o cuando introduce un número o un símbolo) se retorna `None`.

Luego, podemos definir otra *clase case* llamada `Word`, que represente una palabra a ser adivinada por el jugador:

```
sealed abstract case class Word private (word: String) {
  def contains(char: Char) = word.contains(char)
  val length: Int          = word.length
  def toList: List[Char]   = word.toList
  def toSet: Set[Char]     = word.toSet
}
object Word {
  def make(word: String): Option[Word] =
```

```
    if (!word.isEmpty && word.forall(_.isLetter)) Some(new Word(word.toLowerCase) {})
    else None
}
```

Otra vez tenemos un constructor inteligente que verifica que una palabra no sea vacía y que contenga solamente letras, no números ni símbolos. Adicionalmente esta clase contiene algunos métodos útiles (todos son funciones puras, por cierto):

- Saber si una palabra contiene un caracter determinado (`contains`).
- Obtener la longitud de una palabra (`length`).
- Obtener una lista con los caracteres de la palabra (`toList`).
- Obtener un `Set` con los caracteres de la palabra (`toSet`).

A continuación definimos otra *clase case* llamada `State`, la cual representa el estado interno de la aplicación, que incluye:

- El nombre del jugador (`name`).
- Las letras adivinadas hasta el momento (`guesses`).
- La palabra a adivinar (`word`)

```
sealed abstract case class State private (name: Name, guesses: Set[Guess], word: Word) {
  def failuresCount: Int            = (guesses.map(_.char) -- word.toSet).size
  def playerLost: Boolean           = failuresCount > 5
  def playerWon: Boolean            = (word.toSet -- guesses.map(_.char)).isEmpty
  def addGuess(guess: Guess): State = new State(name, guesses + guess, word) {}
}
object State {
  def initial(name: Name, word: Word): State = new State(name, Set.empty, word) {}
}
```

Similarmente, tenemos un constructor inteligente `State.initial`, que permite instanciar el estado inicial del juego con el nombre del jugador y la palabra que tiene que adivinar, obviamente con un `Set` vacío de letras adivinadas. Por otro lado, `State` incluye algunos métodos útiles:

- Obtener la cantidad de fallas del jugador (`failuresCount`)
- Saber si el jugador perdió (`playerLost`)
- Saber si el jugador ganó (`playerWon`)
- Añadir una letra al `Set` de letras adivinadas (`addGuess`)

Finalmente, definimos un modelo `GuessResult` que representa el resultado obtenido por el jugador después de adivinar una letra. Este resultado puede ser uno de los siguientes casos:

- El jugador ganó, porque adivinó la última letra que faltaba.

- El jugador perdió, porque usó el último intento que le quedaba.
- El jugador adivinó correctamente una letra, aunque todavía le faltan letras para ganar.
- El jugador adivinó incorrectamente una letra, aunque todavía le quedan más intentos.
- El jugador repitió una letra que ya había adivinado anteriormente, por lo tanto el estado del juego no cambia.

Esto lo podemos representar con una enumeración, de la siguiente forma:

```
sealed trait GuessResult
object GuessResult {
  case object Won       extends GuessResult
  case object Lost      extends GuessResult
  case object Correct   extends GuessResult
  case object Incorrect extends GuessResult
  case object Unchanged extends GuessResult
}
```

En este caso una enumeración tiene sentido pues `GuessResult` sólo puede tener un valor posible de entre los mencionados en la lista de opciones. Algunos detalles importantes aquí:

- Definimos una enumeración como un `sealed trait`.
- La palabra `sealed` es importante, pues eso asegura que todas las clases que pueden extender `GuessResult` están en el mismo archivo (`package.scala`) y que ninguna otra clase fuera de este archivo podrá extender `GuessResult`.
- La palabra `sealed` es importante además porque de esta forma el compilador nos puede ayudar cuando usamos reconocimiento de patrones (*pattern matching* en inglés) en instancias de `GuessResult`, advirtiéndonos si no hemos considerado todas las opciones posibles.
- Los valores posibles de `GuessResult` están definidos dentro de su objeto acompañante.

Creando el esqueleto de la aplicación

Ahora que ya hemos definido el modelo de dominio de nuestra aplicación, podemos comenzar con la implementación en el archivo `com/example/Hangman.scala`. Dentro de este archivo crearemos un objeto que implemente el trait `zio.App`, de esta manera:

```
import zio._

object Hangman extends App {
  def run(args: List[String]): ZIO[ZEnv, Nothing, ExitCode] = ???
}
```

Con sólo este pequeño pedazo de código podemos aprender varias cosas:

- Para trabajar con ZIO, sólo necesitamos incluir `import zio._`, lo cual nos proporcionará, entre otras cosas, acceso al tipo de datos `ZIO`.
- Toda aplicación ZIO debe implementar el trait `zio.App`, en vez del trait `App` de la librería estándar.
- El trait `zio.App` requiere que implementemos un método `run`, que es el punto de entrada de la aplicación, como podemos ver este método recibe una lista de argumentos como cualquier aplicación normal de Scala, y retorna un efecto funcional `ZIO`, que ya sabemos que solamente es una descripción de lo que nuestra aplicación debe hacer. Dicha descripción será traducida por el **entorno de ejecución de ZIO,** al momento de ejecutar la aplicación, en verdaderas interacciones con el mundo exterior, es decir efectos colaterales (lo interesante de esto es que nosotros como desarrolladores no necesitamos preocuparnos de cómo esto ocurre, ZIO se encarga de eso por nosotros). Por tanto, el método `run` sería el **fin del mundo funcional** para nuestra aplicación, y lo dejaremos sin implementar por el momento.

Funcionalidad para obtener el nombre del jugador por consola

Ahora que ya tenemos el esqueleto básico de nuestra aplicación, primeramente necesitamos escribir la funcionalidad para obtener el nombre del jugador por consola, para ello escribiremos una función auxiliar, dentro del objeto `Hangman`, que nos permita imprimir un mensaje cualquiera y a continuación solicite un texto al jugador:

```
import zio.console._

def getUserInput(message: String): ZIO[Console, IOException, String] = {
  putStrLn(message)
  getStrLn
}
```

Veamos con más detalle esta función:

- Para imprimir el mensaje provisto por consola, usamos la función `putStrLn` incluida en el módulo `zio.console`. Esta función retorna un efecto del tipo `ZIO[Console, Nothing, Unit]`, que es equivalente a `URIO[Console, Unit]`, lo que significa que es un efecto que para ejecutarse requiere del módulo `Console` (que es un módulo estándar provisto por ZIO), que no puede fallar y que retorna un valor de tipo `Unit`.
- Luego, para solicitar al usuario que introduzca un texto, usamos la función `getStrLn` incluida también en el módulo `zio.console`. Esta función retorna un efecto del tipo

`ZIO[Console, IOException, String]`, lo que significa que es un efecto que para ejecutarse requiere del módulo `Console`, que puede fallar con un error de tipo `IOException` y que retorna un valor de tipo `String`.

¡Y eso es todo! Bueno...en realidad hay un pequeño problema, y es que si llamáramos esta función desde el método `run`, nunca se mostraría un mensaje por consola y se pasaría directamente a solicitar un texto al usuario. ¿Qué es lo que está faltando entonces, si primero estamos llamando a `putStrLn` y luego a `getStrLn`? El problema es que tanto `putStrLn` como `getStrLn` retornan simples descripciones de interacciones con el mundo exterior (llamadas efectos funcionales), y si nos fijamos detenidamente: ¿estamos haciendo algo realmente con el efecto funcional retornado por `putStrLn`? La respuesta es que no, simplemente lo estamos descartando, y el único efecto que es retornado por nuestra función `getUserInput` es el efecto retornado por `getStrLn`. Es más o menos como si tuviéramos una función así:

```
def foo(): Int = {
  4
  3
}
```

¿Estamos haciendo algo con el `4` en esa función? ¡Nada! Simplemente es como si no estuviera ahí, y lo mismo pasa en nuestra función `getUserInput`, es como si la llamada a `putStrLn` no estuviera ahí.

Por lo tanto, tenemos que modificar la función `getUserInput` para que el efecto retornado por `putStrLn` sea realmente usado:

```
def getUserInput(message: String): ZIO[Console, IOException, String] =
  putStrLn(message).flatMap(_ => getStrLn)
```

Lo que hace esta nueva versión es retornar un efecto que combina en forma secuencial los efectos retornados por `putStrLn` y `getStrLn`, usando el operador `ZIO#flatMap`, el cual recibe una función donde:

- La entrada es el resultado del primer efecto (en este caso `putStrLn`, que retorna `Unit`).
- Retorna un nuevo efecto a ser ejecutado, en este caso `getStrLn`.
- Algo importante del funcionamiento de `ZIO#flatMap` es que, si el primer efecto falla, el segundo efecto no es ejecutado.

De esta forma, ya no estamos descartando el efecto producido por `putStrLn`.

Ahora bien, gracias a que el tipo de datos `ZIO` también ofrece un método `ZIO#map`, podemos escribir `getUserInput` usando una *comprensión for*, lo cual nos ayuda a visualizar el código en una forma más parecida a como luce un típico código imperativo:

```
def getUserInput(message: String): ZIO[Console, IOException, String] =
  for {
    _     <- putStrLn(message)
    input <- getStrLn
  } yield input
```

Esta implementación funciona perfectamente, sin embargo podemos escribirla de otra forma:

```
def getUserInput(message: String): ZIO[Console, IOException, String] =
  (putStrLn(message) <*> getStrLn).map(_._2)
```

En este caso, estamos usando otro operador para combinar dos efectos de manera secuencial, y es el operador `<*>` (que por cierto es equivalente al método `ZIO#zip`). Este operador, así como `ZIO#flatMap`, combina los resultados de dos efectos, con la diferencia de que el segundo efecto no necesita del resultado del primero para ejecutarse. El resultado de `<*>` es un efecto cuyo valor de retorno es una tupla, pero en este caso sólo necesitamos el valor de `getStrLn` (el resultado de `putStrLn` no nos interesa porque simplemente es un `Unit`), es por ello que ejecutamos `ZIO#map`, el cual es un método que nos permite transformar el resultado de un efecto, en este caso estamos obteniendo el segundo componente de la tupla retornada por `<*>`.

Una versión mucho más simplificada y equivalente a la anterior es la siguiente:

```
def getUserInput(message: String): ZIO[Console, IOException, String] =
  putStrLn(message) *> getStrLn
```

El operador `*>` (equivalente al método `ZIO#zipRight`), hace exactamente lo que hicimos en la anterior versión, pero de una forma mucho más resumida.

Y bueno, ahora que ya tenemos una función para obtener un texto por parte del jugador, podemos implementar un efecto funcional para obtener específicamente su nombre:

```
lazy val getName: ZIO[Console, IOException, Name] =
  for {
    input <- getUserInput("What's your name?")
    name <- Name.make(input) match {
              case Some(name) => ZIO.succeed(name)
              case None       => putStrLn("Invalid input. Please try again...") *> getName
            }
```

```
  } yield name
```

Como podemos ver:

- Primero se solicita al jugador que ingrese su nombre.
- Luego se intenta construir un objeto `Name`, usando el método `Name.make` que hemos definido anteriormente.
    - Si el nombre introducido es válido (es decir no es vacío) retornamos un efecto que termina exitosamente con el nombre correspondiente, usando el método `ZIO.succeed`.
    - Caso contrario, mostramos un mensaje de error por pantalla (usando `putStrLn`) y a continuación volvemos a solicitar el nombre, llamando a `getName` recursivamente. Esto significa que el efecto `getName` se ejecutará tantas veces como sea necesario, mientras el nombre del jugador sea inválido.

¡Y eso es todo! Sin embargo, podemos escribir una versión equivalente:

```
lazy val getName: ZIO[Console, IOException, Name] =
  for {
    input <- getUserInput("What's your name?")
    name  <- ZIO.fromOption(Name.make(input)) <> (putStrLn("Invalid input. Please try again...") *> getName)
  } yield name
```

En esta versión, el resultado de llamar a `Name.make`, que es de tipo `Option`, es convertido en un efecto `ZIO`, usando el método `ZIO.fromOption`, el cual retorna un efecto que termina exitosamente si el `Option` dado es un `Some`, y un efecto que falla si el `Option` dado es un `None`. Luego, estamos usando un nuevo operador `<>` (que es equivalente al método `ZIO#orElse`), que también nos permite combinar dos efectos de forma secuencial, pero de una forma algo diferente a lo que ocurre con `<*>`. La lógica de `<>` es la siguiente:

- Si el primer efecto es exitoso (en este caso si el nombre del jugador es válido), el segundo efecto no se ejecuta.
- Si el primer efecto falla (en este caso si el nombre del jugador es inválido), se ejecuta el segundo efecto (en este caso, se muestra un mensaje de error y a continuación se vuelve a llamar a `getName`).

Como vemos esta nueva versión de `getName` es algo más concisa, y nos quedaremos con ella.

Funcionalidad para escoger una palabra, en forma aleatoria, del diccionario

Veamos ahora cómo implementar la funcionalidad para escoger de forma aleatoria cuál es la palabra que el jugador tiene que adivinar. Para ello disponemos de un diccionario `words`, que

es simplemente una lista de palabras localizada en el archivo `com/example/package.scala`.

La implementación es la siguiente:

```
import zio.random._

lazy val chooseWord: URIO[Random, Word] =
  for {
    index <- nextIntBounded(words.length)
    word  <- ZIO.fromOption(words.lift(index).flatMap(Word.make)).orDieWith(_ => new Error("Boom!"))
  } yield word
```

Como podemos ver, estamos usando la función `nextIntBounded` del módulo `zio.random`. Esta función retorna un efecto que genera números enteros aleatorios entre 0 y un límite dado, en este caso el límite es la longitud del diccionario. Una vez que tenemos el entero aleatorio, obtenemos la palabra correspondiente del diccionario, con la expresión `words.lift(index).flatMap(Word.make)`, que retorna un `Option[Word]`, que es convertido a un efecto `ZIO` usando el método `ZIO.fromOption`, este efecto:

- Termina exitosamente si la palabra es encontrada en el diccionario para un determinado `index`, y si además dicha palabra no es vacía (condición verificada por `Word.make`).
- Caso contrario, falla.

Si lo pensamos detenidamente, ¿puede haber algún caso en que la obtención de alguna palabra del diccionario falle? No realmente, porque el efecto `chooseWord` nunca intentará obtener una palabra cuyo índice esté fuera del rango de la longitud del diccionario, y por otro lado todas las palabras del diccionario están predefinidas y no son vacías. Por tanto, podemos descartar sin ningún problema el caso erróneo, usando el método `ZIO#orDieWith`, el cual retorna un nuevo efecto que no puede fallar y, en caso de que sí fallara, eso significaría que hay algún **defecto** serio en nuestra aplicación (por ejemplo que alguna de nuestras palabras predefinidas sea vacía cuando no debería serlo) y por lo tanto ésta debería fallar inmediatamente con la excepción provista.

Al final, `chooseWord` es un efecto de tipo `URIO[Random, Word]`, lo que significa que:

- Para ejecutarse requiere del módulo `Random` (que es otro módulo estándar provisto por ZIO, así como lo es el módulo `Console`).
- No puede fallar.
- Termina exitosamente con un objeto de tipo `Word`.

Funcionalidad para mostrar por consola el estado del juego

Lo siguiente que necesitamos hacer es implementar la funcionalidad para mostrar por consola el estado del juego:

```
def renderState(state: State): URIO[Console, Unit] = {

  /*
     --------
     |      |
     |      0
     |     \|/
     |      |
     |     / \
     -

     f     n  c  t  o
     _  _  _  _  _  _  _
     Guesses: a, z, y, x
  */
  val hangman = ZIO(hangmanStages(state.failuresCount)).orDieWith(_ => new Error("Boom!"))
  val word =
    state.word.toList
      .map(c => if (state.guesses.map(_.char).contains(c)) s" $c " else "   ")
      .mkString

  val line    = List.fill(state.word.length)(" - ").mkString
  val guesses = s" Guesses: ${state.guesses.map(_.char).mkString(", ")}"

  hangman.flatMap { hangman =>
    putStrLn(
      s"""
       #$hangman
       #
       #$word
       #$line
       #
       #$guesses
       #
       #""".stripMargin('#')
    )
  }
}
```

No entraremos en mucho detalle de cómo esta función está implementada, más bien nos centraremos en una sola línea, que es quizás la más interesante:

```
val hangman = ZIO(hangmanStages(state.failuresCount)).orDie
```

Podemos ver que lo que hace esta línea es, primeramente, seleccionar cuál es la figura que se

deberá mostrar para representar al ahorcado, la cual es diferente de acuerdo a la cantidad de fallas que haya tenido el jugador (el archivo `com/example/package.scala` contiene una variable `hangmanStates` que consiste en una lista con las seis posibles figuras). Por ejemplo:

- `hangmanStates(0)` contiene el dibujo del ahorcado para `failuresCount=0`, es decir muestra solamente el dibujo de la horca sin el ahorcado
- `hangmanStates(1)` contiene el dibujo del ahorcado para `failuresCount=1`, es decir muestra el dibujo de la horca y la cabeza del ahorcado.

Ahora bien, la expresión `hangmanStages(state.failuresCount)` no es puramente funcional porque podría lanzar una excepción si es que `state.failuresCount` fuera mayor a 6. Entonces, como estamos trabajando con programación funcional no podemos permitir que nuestro código produzca efectos colaterales, como lanzar excepciones, es por ello que hemos encapsulado la anterior expresión dentro de `ZIO`, que en realidad es una llamada al método `ZIO.apply`, el cual permite construir un efecto funcional a partir de una expresión que produce efectos colaterales (también podemos usar el método equivalente `ZIO.effect`). Por cierto, el resultado de llamar a `ZIO.apply` es un efecto que puede fallar con un `Throwable`, pero de acuerdo al diseño de nuestra aplicación esperamos que en realidad nunca exista un fallo al intentar obtener el dibujo del ahorcado (puesto que `state.failuresCount` nunca debería ser mayor a 6), por tanto podemos descartar el caso erróneo llamando al método `ZIO#orDie`, el cual retorna un efecto que nunca falla (ya sabemos que si existiera alguna falla, sería en realidad un **defecto** y nuestra aplicación debería fallar inmediatamente). Por cierto, `ZIO#orDie` es muy similar a `ZIO#orDieWith`, pero sólo puede ser usado con efectos que fallan con un `Throwable`.

Funcionalidad para obtener una letra por parte del jugador

La funcionalidad para obtener una letra por parte del jugador es muy similar a cómo obtenemos su nombre, por lo tanto no entraremos en detalle:

```
lazy val getGuess: ZIO[Console, IOException, Guess] =
  for {
    input <- getUserInput("What's your next guess?")
    guess <- ZIO.fromOption(Guess.make(input)) <> (putStrLn("Invalid input. Please try again...") *> getGuess)
  } yield guess
```

Funcionalidad para analizar una letra introducida por parte del jugador

Esta funcionalidad es muy sencilla, lo único que hace es analizar el estado previo y el estado posterior a un intento del jugador, para ver si el jugador gana, pierde, adivinó correctamente una letra pero todavía no ganó el juego, adivinó incorrectamente una letra pero todavía no perdió el juego o si volvió a intentar una letra que ya había intentado previamente:

```
def analyzeNewGuess(oldState: State, newState: State, guess: Guess): GuessResult =
  if (oldState.guesses.contains(guess)) GuessResult.Unchanged
  else if (newState.playerWon) GuessResult.Won
  else if (newState.playerLost) GuessResult.Lost
  else if (oldState.word.contains(guess.char)) GuessResult.Correct
  else GuessResult.Incorrect
```

Implementación del bucle del juego

La implementación del bucle del juego usa las funcionalidades que hemos definido anteriormente:

```
def gameLoop(oldState: State): ZIO[Console, IOException, Unit] =
  for {
    guess       <- renderState(oldState) *> getGuess
    newState    = oldState.addGuess(guess)
    guessResult = analyzeNewGuess(oldState, newState, guess)
    _ <- guessResult match {
          case GuessResult.Won =>
            putStrLn(s"Congratulations ${newState.name.name}! You won!") *> renderState(newState)
          case GuessResult.Lost =>
            putStrLn(s"Sorry ${newState.name.name}! You Lost! Word was: ${newState.word.word}") *>
              renderState(newState)
          case GuessResult.Correct =>
            putStrLn(s"Good guess, ${newState.name.name}!") *> gameLoop(newState)
          case GuessResult.Incorrect =>
            putStrLn(s"Bad guess, ${newState.name.name}!") *> gameLoop(newState)
          case GuessResult.Unchanged =>
            putStrLn(s"${newState.name.name}, You've already tried that letter!") *> gameLoop(newState)
        }
  } yield ()
```

Como podemos ver, lo que hace la función `gameLoop` es lo siguiente:

- Muestra por consola el estado del juego y obtiene una letra por parte del jugador.
- Se actualiza el estado del juego con la nueva letra adivinada por el jugador.
- Se analiza la nueva letra adivinada por el jugador y:
    - Si el jugador ganó, se muestra un mensaje de felicitación y se muestra por última vez el estado del juego.
    - Si el jugador perdió, se muestra un mensaje indicando cuál era la palabra a adivinar y se muestra por última vez el estado del juego.
    - Si el jugador adivinó una letra correctamente pero aún no ganó el juego, se muestra un mensaje indicando que adivinó correctamente y se vuelve a llamar a `gameLoop`, con el estado actualizado del juego.

- Si el jugador adivinó una letra incorrectamente pero aún no perdió el juego, se muestra un mensaje indicando que adivinó incorrectamente y se vuelve a llamar a `gameLoop`, con el estado actualizado del juego.
- Si el jugador intenta una letra que ya había adivinado antes, se muestra un mensaje y se vuelve a llamar a `gameLoop`, con el estado actualizado del juego.

Al final, `gameLoop` retorna un efecto `ZIO` que depende del módulo `Console` (esto es obvio porque necesita leer texto por teclado y escribir por consola), puede fallar con un `IOException` o terminar exitosamente con un valor `Unit`.

Uniendo todas las piezas

Finalmente tenemos todas las piezas de nuestra aplicación, ahora lo único que resta es unirlas y llamarlas desde el método `run` que, como ya hemos mencionado previamente, es el punto de entrada de toda aplicación basada en ZIO:

```
def run(args: List[String]): ZIO[ZEnv, Nothing, ExitCode] =
 (for {
   name <- putStrLn("Welcome to ZIO Hangman!") *> getName
   word <- chooseWord
   _    <- gameLoop(State.initial(name, word))
 } yield ()).exitCode
```

Podemos ver que la lógica es muy sencilla:

- Se imprime un mensaje de bienvenida y se solicita el nombre del jugador.
- Se escoge una palabra de forma aleatoria para que el jugador la adivine.
- Se ejecuta el bucle del juego.

Hay algunos detalles importantes que explicar aquí, la expresión:

```
for {
   name <- putStrLn("Welcome to ZIO Hangman!") *> getName
   word <- chooseWord
   _    <- gameLoop(State.initial(name, word))
 } yield ()
```

Retorna un efecto del tipo `ZIO[Console with Random, IOException, Unit]`, y es interesante ver cómo ZIO sabe exactamente, todo el tiempo, de qué módulos depende un efecto funcional. En este caso el efecto depende del módulo `Console` y del módulo `Random`, lo cual es obvio porque nuestra aplicación requiere imprimir mensajes por pantalla, leer texto por

teclado y generar palabras aleatorias. Sin embargo, el método `run` requiere retornar un efecto del tipo `ZIO[ZEnv, Nothing, ExitCode]`. Es por ello que se necesita llamar el método `ZIO#exitCode`, el cual retornará un efecto del tipo `ZIO[Console with Random, Nothing, ExitCode]`, de la siguiente manera:

- Si el efecto original termina exitosamente, se retorna `ExitCode.success`
- Si el efecto original falla, se imprime el error por consola y se retorna `ExitCode.failure`

Ahora bien, si nos fijamos detenidamente, estamos retornando un efecto del tipo `ZIO[Console with Random, Nothing, ExitCode]`, pero `run` requiere retornar `ZIO[ZEnv, Nothing, ExitCode]`, ¿entonces por qué no hay ningún error de compilación? Para ello necesitamos entender qué significa `ZEnv`:

```
type ZEnv = Clock with Console with System with Random with Blocking
```

Podemos ver que `ZEnv` es solamente un alias que engloba todos los módulos estándar provistos por ZIO. Entonces, `run` básicamente espera retornar un efecto que requiera solamente los módulos provistos por ZIO, pero no necesariamente todos, y como el efecto que estamos intentando retornar requiere `Console with Random`, no hay ningún problema. Puede ser que te estés preguntando ahora: ¿acaso hay algún caso en el que tengamos efectos que requieran módulos que no son provistos por ZIO? Y la respuesta es que sí, porque nosotros podemos definir nuestros propios módulos. Y ahora te preguntarás, ¿qué hacemos en esos casos? La respuesta no la veremos en este artículo, pero si quieres saber más al respecto tenemos un *ebook* en inglés que explica cómo desarrollar aplicaciones modulares con ZIO, usando como ejemplo la implementación del juego del Tic-Tac-Toe.

¡Y eso es todo! Hemos terminado la implementación del juego del ahorcado, usando un estilo de programación puramente funcional, con la ayuda de la librería ZIO.

Conclusiones

En este artículo hemos podido ver cómo gracias a librerías como ZIO podemos implementar aplicaciones completas usando el paradigma de programación funcional, tan sólo escribiendo descripciones de interacciones con el mundo exterior (llamadas efectos funcionales) que pueden ser compuestas entre sí para formar descripciones más complejas. Hemos visto que ZIO permite crear, combinar y transformar efectos funcionales entre sí de diversas maneras, aunque ciertamente no hemos visto todas las posibilidades que ZIO ofrece, sin embargo espero que ello te sirva de impulso para continuar explorando el fascinante ecosistema que está siendo construido alrededor de esta librería.

Algo importante que no hemos hecho en este artículo es escribir tests unitarios para nuestra aplicación, y eso será precisamente el tema a tratar en un siguiente artículo, donde podremos ver cómo también es posible escribir tests unitarios, de forma puramente funcional, usando la librería ZIO Test.

Finalmente, si quieres aprender más sobre ZIO, puedes echarle un vistazo a los siguientes recursos (disponibles en inglés solamente):

- La Documentación Oficial de ZIO
- Artículos relacionados con ZIO en el Blog Oficial de Scalac

Referencias

- Repositorio Github para el presente artículo
- Artículo en inglés acerca de los Beneficios de la Programación Funcional, en el Blog de Scalac
- Libro en inglés: Functional Programming in Scala, por Paul Chiusano y Runar Bjarnason
- Artículos en inglés relacionados con ZIO, en el Blog Oficial de Scalac
- Ebook en inglés acerca de cómo desarrollar aplicaciones modulares con ZIO, por Jorge Vásquez
- Documentación Oficial de ZIO
- Repositorio Github con ejercicios para aprender ZIO, por John De Goes
- Top 10 de los lenguajes que los desarrolladores quieren aprender.