> scalac

Jorge Vasquez

Introduction to Programming with ZIO Functional Effects

Table of content

Table of content

Introduction

The functional programming paradigm is often avoided by developers around the world, who prefer to continue developing applications using an object-oriented programming paradigm, which is better known and more widely used. This rejection of functional programming happens either due to a simple lack of knowledge or because developers tend to think that it is something hard and only useful in academic fields. This is probably because there are a lot of new concepts that are usually explained in a principled way, but full of mathematical jargon. Nothing could be further from the truth: functional programming helps us to solve many of the problems that arise when developing software, especially programs that require high levels of asynchronicity and concurrency, and if it is properly explained it does not have to be something difficult to learn.

Scala language combines the best of both the functional and object-oriented programming worlds, and provides complete interoperability with the vast Java ecosystem. In addition, within the Scala ecosystem, there are libraries such as ZIO, which allow developers to be highly productive and create modern, highly performant and concurrent applications, leveraging the power of functional programming - all without any huge barriers to entry.

In this article I will explain the principles of functional programming, and then demonstrate how, with the help of Scala and ZIO, we can create applications to solve real-world problems. As an illustrative example, we will implement a hangman game.

Functional programming 101

What is functional programming?

Functional programming is a programming paradigm, where programs are a composition of **pure functions.** This represents a fundamental difference to object-oriented programming, where programs are sequences of statements, usually operating in a mutable state. These are organized into so-called functions, but they are not functions in the mathematical sense, because they do not meet some fundamental characteristics.

A pure function must be total

Firstly, **a function must be total**, this means that for each input that is provided to the function there must be a defined output. For example, the following function dividing two integers is not total:

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

To answer why this function is not total, consider what will happen if we try to divide by zero:

```scala
divide(5, 0)

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

The division by zero is undefined and Java handles this by throwing an exception. That means the `divide` function is not total because it does not return any output in the case where `b = 0`.

Some important things we can highlight here:

- If we look at the signature of this function, we realize that it's telling a lie: that for each pair of integers *a* and b, this function will always return another integer, which we have already seen is not true for all cases.
- This means that every time we call the `divide` function in some part of our application we will have to be very careful, as we can never be completely sure that the function is going to return an integer.
- If we forget to consider the unhandled case when calling `divide`, then at runtime our application will throw exceptions and stop working as expected. Unfortunately, the compiler is not able to do anything to help us to avoid this type of situation, this code will compile and we will only see the problems at runtime.

So how can we fix this problem? Let's look at this alternative definition of the `divide` function:

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

In this case, the `divide` function returns `Option[Int]` instead of `Int`, so that when `b = 0`, the function returns None instead of throwing an exception. This is how we have transformed a partial function into a total function, and thanks to this we have some benefits:

- The function's signature is clearly communicating that some inputs are not being handled because it returns an `Option[Int]`.
- When we call the `divide` function in some part of our application, the compiler will force us to consider the case in which the result is not defined. If we don't, it will be a compile-time error, which means the compiler can help us to avoid many bugs, and they will not appear at runtime.

A pure function must be deterministic and must depend only on its inputs

The second characteristic of a function is that it must be **deterministic and must depend only on its inputs**. This means that for each input that is provided to the function, the same output must be returned, no matter how many times the function is called. For example, the following function for generating random integers is not deterministic:

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

To demonstrate why this function is not deterministic, let's consider what happens the first time we call the function:

```scala
generateRandomInt() // Result: -272770531
```

And then, what happens when we call the function again:

```scala
generateRandomInt() // Result: 217937820
```

We get different results! Clearly this function is not deterministic and its signature is misleading again, because it suggests that it does not depend on any input to produce an output, whereas in truth there is actually a hidden dependency on a `scala.util.Random` object. This may cause problems, because we can never really be sure how the `generateRandomInt` function is going to behave, making it difficult to test.

Now, let's have a look at an alternative definition. For this, we'll use a custom random number generator, based on an example from the Functional Programming in Scala book:

```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
```

This new version of the `generateRandomInt` function is deterministic: no matter how many times it is called, we will always get the same output for the same input and the signature now clearly states the dependency on the `random` variable. For example:

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

If we want to generate a new integer, we must provide a different input:

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

A pure function must not have side effects

Finally, a function must not have any side effects. Some examples of side effects are the following:

- memory mutations,
- interactions with the outside world, such as:
  - printing messages to the console,
  - calling an external API,
  - querying a database.

This means that a pure function can only work with immutable values and can only return an output for a corresponding input, nothing else.

For example, the following `increment` function is not pure because it works with a mutable variable *a*:

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

And the following function is not pure either because it prints a message in the console:

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

What are the differences between functional programming and object-oriented programming?

The following table summarizes the major differences between these two programming paradigms:

	Functional programming	**Object-oriented programming**
Variables	immutable	mutable
Programming model	declarative	imperative
Focuses on ...	“what” is done	“how” to do it
Parallellism	suitable for parallel programming	not suitable for parallel programming
Side effects	functions cannot produce side effects	methods can produce side effects
Iterations	recursion	loops (`for`, `while`)
Application state	flows through pure functions	usually shared by several objects
Key elements	immutable values and functions	objects and methods

What are the benefits of functional programming?

For various reasons, functional programming may still seem complex to many. But, if we take a closer look at the benefits, we can change our way of thinking.

For starters, embracing this programming paradigm helps us to break each application down into smaller, simpler pieces that are reliable and easy to understand. This is because a functional source code is often more **concise**, **predictable,** and **easier to test.** But how can we ensure this?

- Since pure functions do not rely on any state and instead depend only on their inputs, they are much easier to understand. That is, to understand what a function does, we do

not need to look for other parts of the code that could affect its operation. This is known as **local reasoning.**
- Their code tends to be more **concise**, which results in **fewer bugs.**
- The process of testing and debugging becomes much easier with functions that only receive input data and produce output.
- Since pure functions are deterministic, **applications behave more predictably.**
- Functional programming allows us to **write correct parallel programs,** since there is no possibility of having a mutable state, therefore **it is impossible for typical concurrency problems to occur, such as race conditions.**

Since Scala supports the functional programming paradigm, these benefits also apply to the language itself. As a result, more and more companies are using Scala, including giants such as LinkedIn, Twitter, and Netflix. In addition, this year Scala is one of the top 10 languages that developers want to learn.

But ... real life applications need to run side effects!

Now we know that functional programming is about programming with **pure functions**, and that pure functions **cannot produce side effects**, several logical questions arise:

- How can we then have an application written using functional programming, which at the same time interacts with external services, such as databases or third-party APIs? Isn't this somewhat contradictory?
- Does this mean that functional programming cannot be used in real applications, but only in academic settings?

The answer to these questions is the following: **Yes, we can use functional programming in real applications, and not just in academic settings.** For our applications to be able to interact with external services, we can do the following: instead of writing functions that interact with the outside world, we write **functions that describe interactions with the outside world**, which are executed only at a specific point in our application, (usually called the **end of the world**) for example the `main` function.

If we think about this carefully, **these descriptions of interactions with the outside world are simply immutable values** that can serve as inputs and outputs of pure functions, and in this way we would not be violating a basic principle of functional programming: do not produce side effects.

What is this aforementioned **end of the world**? Well, the end of the world is simply a specific point in our application **where the functional world ends**, and where the descriptions of interactions with the outside world are being run, usually as late as possible, preferably at the very edge of our program which is its `main` function. In this way, our entire application can be written following the functional style, but at the same time capable of performing useful tasks.

Now that we know all this, a new question arises: how can we write our applications in such a way that all our functions do not execute side effects, but only build descriptions of what we want to do? And this is where a very powerful library comes in, one that can help us with this task: Introducing **the ZIO library.**

Introduction to the ZIO Library

What is the ZIO Library for?

ZIO is a library that allows us to build **modern applications** that are **asynchronous, concurrent, resilient, efficient, easy to understand and to test,** using the principles of functional programming.

Why do we say that ZIO allows us to build applications that are **easy to understand and test**? Because it helps us to build applications of any complexity incrementally, through a combination of descriptions of interactions with the outside world. By the way, these descriptions are called **functional effects.**

Why do we say that ZIO allows us to build applications that are **resilient**? Because ZIO takes full advantage of the Scala type system, in such a way that it can catch more bugs at compile time, rather than at run time. This is great because, just by looking at the signature of a function, we can tell:

- If it has **external dependencies.**
- **If it can fail or not**, and also with what type of errors it can fail.
- **If it can finish successfully or not**, and also what the type of data is that returns when finishing.

And finally, why do we say that ZIO allows us to build applications that are **asynchronous and concurrent**? Because ZIO gives us the superpowers to work with asynchronous and concurrent programming, using a **fiber-based model**, which is much more efficient than a thread-based model. We will not go into much detail about this particular aspect in this article, however it is worth mentioning that it is precisely in this area that ZIO shines, allowing us to build really performant applications.

The ZIO data type

The most important data type in the ZIO library (and also the basic building block of any application based on this library), is also called `ZIO`:

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

The `ZIO` data type is a **functional effect**, which means that it is an immutable value that contains a description of a series of interactions with the outside world (database queries, calls to third-party APIs, etc.). A good mental model of the `ZIO` data type is the following:

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

This means that a `ZIO` effect:

- Needs a context of type `R` to run (this context can be anything: a connection to a database, a REST client , a configuration object, etc.).
- It may fail with an error of type `E` or it may complete successfully, returning a value of type `A`.

Common aliases for the ZIO data type

It's worth mentioning that ZIO provides some type aliases for the `ZIO` data type which are very useful when it comes to representing some common use cases:

- `Task[+A] = ZIO[Any, Throwable, A]:` This means a `Task[A]` is a `ZIO` effect that:
  - Doesn't require an environment to run (that's why the R type is replaced by `Any`, meaning the effect will run no matter what we provide to it as an environment)
  - Can fail with a `Throwable`
  - Can succeed with an `A`
- `UIO[+A] = ZIO[Any, Nothing, A]:` This means a `UIO[A]` is a ZIO effect that:
  - Doesn't require an environment to run.
  - Can't fail
  - Can succeed with an `A`
- `RIO[-R, +A] = ZIO[R, Throwable, A]:` This means a `RIO[R, A]` is a ZIO effect that:
  - Requires an environment `R` to run
  - Can fail with a `Throwable`
  - Can succeed with an `A`
- `IO[+E, +A] = ZIO[Any, E, A]:` This means a `IO[E, A]` is a ZIO effect that:
  - Doesn't require an environment to run.
  - Can fail with an `E`
  - Can succeed with an `A`

- `URIO[-R, +A] = ZIO[R, Nothing, A]`: This means a `URIO[R, A]` is a ZIO effect that:
  - Requires an environment R to run
  - Can't fail
  - Can succeed with an A

Implementing a Hangman game using ZIO

Next up, we will implement a Hangman game using ZIO, as an illustrative example of how we can develop purely functional applications in Scala with the help of this library.

For reference, you can see a Github repository with the complete code at this link. By the way, the implementation of the game is based on this repository created by John De Goes, author of the ZIO library, where you can find several interesting exercises.

Design and requirements

A Hangman game consists of taking a word at random and giving the player the possibility of guessing letters until the word is completed. The rules are as follows:

- The player has 6 attempts to guess letters.
- For each incorrect letter, one attempt is subtracted.
- When the player runs out of attempts, he loses. If you guess all the letters, you win.

So, our implementation of the Hangman game should work as follows:

- When starting the game, **the player should be asked for his name,** which obviously should not be empty. If so, an error message should be displayed and the name requested again.
- The application must then **randomly select**, within a predefined dictionary of words, **the word that the player must guess.**
- Then **the initial state of the game must be displayed on the console**, which basically consists of a gallows, a series of dashes that represent the number of letters in the word to be guessed and the letters that have already been tried by the player, which will obviously be zero at the beginning of the game.
- Then, **the player must be asked to guess a letter and write it on the console.** Obviously, the character entered must be a valid letter, regardless of whether it is uppercase or lowercase:
  - **If the character entered is invalid**, an error message should be displayed, and a letter should be requested from the player again.

    - **If the player has entered a valid letter but it does not appear in the word**, the player loses an attempt, an appropriate message is displayed on the console, and the game status is updated, adding the letter recently attempted by the player to the attempts list, and drawing the head of the hanged man. By the way, all of the subsequent times that the player makes a mistake, the following parts of the body will be shown in order: the trunk, right arm, left arm, right leg and left leg of the hanged man.
    - **If the user has entered a valid letter and it appears in the word**, an appropriate message is displayed on the console and the game status is updated, adding the letter recently attempted by the user to the list of attempts, and discovering in the hidden word the places where the guessed letter appears.
- The previous step is repeated until the user guesses the entire word or runs out of attempts.
    - **If the player wins,** a congratulatory message is displayed.
    - **If the player loses**, a message is displayed indicating what the word was to be guessed.

Creating the base structure of the application

We will define our application as a `sbt` project, the `build.sbt` file will contain the dependencies of our project:

```
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)
```

As you can see, we will work with `Scala 2.13.4` and with `ZIO 1.0.4`.

Creating the domain model, using a functional style

Firstly, we must define our domain model. For this we will define some classes within the file com/example/package.scala.

As a first step, we can define a Name class that represents the player's name:

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

As you can see, we define Name as a case class instead of simply defining it as a normal class. Using case classes gives us a lot of benefits, such as:

- **Immutability** by default.
- An apply method is generated automatically, which allows us to build objects without using the new keyword.
- An unapply method is generated automatically, which we can use in pattern matching expressions.
- A copy method is generated automatically, which allows us to make copies of objects and update certain fields at the same time.

Also, we define the class as final so that it cannot be extended by other classes. On the other hand, we see the Name class simply encapsulates a String that represents the user's name. We could leave this model as it is, however there's a problem. It allows us to create Name objects with an empty String, which is unwanted. So, to prevent this, we can apply a functional design technique called **smart constructor**, which simply consists of defining a method in the companion object of the Name class that allows us to do the respective validations before creating an instance. So, we would have something like this:

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

As you can see, the make method corresponds to our smart constructor, and it checks if the received String is empty or not. If it is not empty, it returns a new Name, but is encapsulated within a Some, and if it is empty it returns a None. The important thing to highlight here is that the make method is a pure function, because it is total, deterministic, the output only depends on the String input and it does not produce side effects.

Now, there are some things we can improve in our implementation. For example, it is still possible to directly call the Name constructor with empty Strings. To avoid this, we can make it private:

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

Thus, the only way to construct Name objects is by calling the Name.make method, at least that can be expected., However we are still able to construct Name objects with empty Strings by using the Name.apply method. Besides that, there is also another problem that we must solve, which is that thanks to the copy method that is automatically generated for case classes, after having created a valid Name object using Name.make, we could then generate an invalid object generating a copy with an empty String, and nothing would stop us from doing that. To avoid these issues, we can define the class as a sealed abstract instead of final, which means apply and copy won't be generated:

```
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
}
```

In this way, we are completely sure that any Name object in our application will always be valid.

Similarly, we can define a case class called Guess, which represents any letter guessed by the player:

```
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) {}
    }}
```

As you can see, we have used the same technique of defining a smart constructor, which receives a String entered by the player, and checks if it only consists of one letter. If that is not the case (for example when the String entered by the player consists of several characters or when entering a number or a symbol) None is returned.

Then we can define another case class called `Word`, which represents the word to be guessed by the player:

```scala
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
}
```

Again we have a smart constructor that verifies that a word is not empty and contains only letters, not numbers or symbols. Additionally this class contains some useful methods (they are all pure functions, by the way):

- To know if a word contains a certain character (`contains`).
- To get the length of a word (`length`).
- To get a `List` with the characters of the word (`toList`).
- To get a `Set` with the characters of the word (`toSet`).

Next, we define another case class called `State`, which represents the internal state of the application, which includes:

- The name of the player (`name`).
- The letters guessed so far (`guesses`).
- The word to guess (`word`)

```scala
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) {}
}
```

Similarly, we have a smart constructor `State.initial,` which allows us to instantiate the initial state of the game with the name of the player and the word that has to be guessed, obviously with an empty `Set` of guessed letters. On the other hand, `State` includes some useful methods:

- To get player's number of failures (`failuresCount`)

- To know if the player lost (`playerLost`)
- To know if the player won (`playerWonGuess`)
- To add a letter to the letter `Set` (`addGuess`)

Finally, we define a `GuessResult` model that represents the result obtained by the player after guessing a letter. This result can be one of the following:

- The player won, because he guessed the last missing letter.
- The player lost, because he used his last remaining attempt.
- The player correctly guessed a letter, although there are still missing letters to win.
- The player incorrectly guessed a letter, although he still has more attempts.
- The player repeated a letter that he had previously guessed, therefore the state of the game does not change.

We can represent this with an enumeration, as follows:

```scala
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
}
```

In this case an enumeration makes sense since `GuessResult` can only have one possible value from those mentioned in the list of options. Some important details here:

- We define an enumeration as a `sealed trait`.
- The word `sealed` is important, as it ensures that all classes that can extend `GuessResult` are in the same file (`package.scala`) and that no other class outside of this file will be able to extend `GuessResult.`
- The word `sealed` is also important because in this way the compiler can help us when we use pattern matching on instances of `GuessResult,` warning us if we have not considered all possible options.
- The possible values of `GuessResult` are defined within its companion object.

Creating the application skeleton

Now that we have defined the domain model of our application, we can begin with the implementation in the file `com/example/Hangman.scala.` Inside this file we will create an object that implements the `zio.App` trait, like this:

```scala
import zio._

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

With just this little piece of code we can learn several things:

- To work with ZIO, we only need to include `import zio._`, which will give us, among other things, access to the `ZIO` data type.
- Every ZIO application must implement the `zio.App` trait, instead of the `App` trait from the standard library.
- The `zio.App` trait requires that we implement a `run` method, which is the application's entry point, as you can see this method receives a list of arguments like any normal Scala application, and returns a `ZIO` functional effect, which we already know is only a description of what our application should do. This description will be translated by the **ZIO execution environment,** at the time of executing the application, in real interactions with the outside world, that is, side effects (the interesting thing about this is that we as developers do not need to worry about how this happens, ZIO takes care of all that for us). Therefore, the `run` method would be the **end of the functional world** for our application, and we will leave it unimplemented for now.

Functionality to obtain the name of the player by console

Now that we have the basic skeleton of our application, firstly we need to write a functionality to obtain the name of the player by the console, for this we will write an auxiliary function, inside the `Hangman` object, which allows us to print any message and then requests a text from the player:

```scala
import zio.console._

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

Let's see this function in more detail:

- To print the provided message on the console, we use the `putStrLn` function included in the `zio.console` module. This function returns an effect of the type `ZIO[Console, Nothing, Unit]`, which is equivalent to `URIO[Console, Unit]`, which means that it is an effect that requires the `Console` module to be executed (a standard module provided by ZIO), that cannot fail and that returns a value of type `Unit`.

- Then, to ask the user to enter a text, we use the `getStrLn` function also included in the `zio.console` module. This function returns an effect of type `ZIO[Console, IOException, String]`, which means that it is an effect that requires the `Console` module to be executed, which can fail with an error of type `IOException` and returns a value of type `String`.

And that's it! Well... there is actually a small problem, and that is that if we called this function from the `run` method, a message would never be displayed in the console and it would go directly to request a text from the user. What is missing then, if we call `putStrLn` first and then `getStrLn`? The problem is that both `putStrLn` and `getStrLn` return simple descriptions of interactions with the outside world (called functional effects), and if we look closely: are we really doing something with the functional effect returned by `putStrLn`? The answer is no, we are simply discarding it, and the only effect that is returned by our `getUserInput` function is the effect returned by `getStrLn`. It's more or less as if we had a function like this:

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

Are we doing something with the 4 in that function? Nothing! It's just as if it's not there, and the same thing happens in our `getUserInput` function, it's as if the `putStrLn` call wasn't there.

Therefore, we have to modify the `getUserInput` function so that the effect returned by `putStrLn` is actually used:

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

What this new version does is to return an effect that sequentially combines the effects returned by `putStrLn` and `getStrLn`, using the `ZIO#flatMap` operator, which receives a function where:

- The input is the result of the first effect (in this case `putStrLn`, which returns `Unit`).
- Returns a new effect to be executed, in this case `getStrLn`.
- Something important about how `ZIO#flatMap` works, is that if the first effect fails, the second effect is not executed.

In this way, we are no longer discarding out the effect produced by `putStrLn`.

Now, because the ZIO data type also offers a ZIO#map method, we can write getUserInput using a for comprehension, which helps us to visualize the code in a way that looks more like a typical imperative code:

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

This implementation works perfectly, however we can write it differently:

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

In this case, we are using another operator to combine two effects sequentially, and it is the <*> operator (which by the way is equivalent to the ZIO#zip method). This operator, like ZIO#flatMap, combines the results of two effects, with the difference that the second effect does not need the result of the first to be executed. The result of <*> is an effect whose return value is a tuple, but in this case we only need the value of getStrLn (the result of putStrLn does not interest us because it is simply a Unit), that is why we execute ZIO#map, which is a method that allows us to transform the result of an effect, in this case we are obtaining the second component of the tuple returned by <*>.

A much more simplified version equivalent to the previous one is the following:

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

The *> operator (equivalent to the ZIO#zipRight method) does exactly what we did in the previous version, but in a much more condensed way.

And well, now that we have a function to get a text from the player, we can implement a functional effect to specifically get their name:

```scala
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
```

As you can see:

- First the player is asked to enter his name.
- Then an attempt is made to build a `Name` object, using the method `Name.make` that we defined earlier.
    - If the entered name is valid (that is, it is not empty) we return an effect that ends successfully with the corresponding name, using the `ZIO.succeed` method.
    - Otherwise, we display an error message on the screen (using `putStrLn`) and then request the name again, calling `getName` recursively. This means that the `getName` effect will run as many times as necessary, as long as the player's name is invalid.

And that's it! However, we can write an equivalent version:

```scala
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
```

In this version, the result of calling `Name.make`, which is of type `Option`, is converted into a `ZIO` effect, using the `ZIO.fromOption` method, which returns an effect that ends successfully if the given `Option` is a `Some`, and an effect that fails if the given `Option` is `None`. Then we are using a new `<>` operator (which is equivalent to the `ZIO#orElse` method), which also allows us to combine two effects sequentially, but in a somewhat different way than `<*>`. The logic of `<>` is as follows:

- If the first effect is successful (in this case: if the player's name is valid), the second effect is not executed.
- If the first effect fails (in this case: if the player's name is invalid), the second effect is executed (in this case: an error message is displayed and then `getName` is called again).

As you can see, this new version of `getName` is somewhat more concise, and so we will stick with it.

Functionality to choose a word at random from the dictionary

Let's now see how to implement the functionality to randomly choose which word the player has to guess. For this we have a `words` dictionary, which is simply a list of words located in the file `com/example/package.scala`.

The implementation is as follows:

```scala
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
```

As you can see, we are using the `nextIntBounded` function of the `zio.random` module. This function returns an effect that generates random integers between 0 and a given limit, in this case the limit is the length of the dictionary. Once we have the random integer, we obtain the corresponding word from the dictionary, with the expression `words.lift(index).flatMap(Word.make)`, which returns an `Option[Word]`, which is converted to a `ZIO` effect using the `ZIO.fromOption` method, this effect:

- Ends successfully if the word is found in the dictionary for a certain `index`, and if this word is not empty (condition verified by `Word.make`).
- Otherwise, it fails.

If we think about it carefully, can there be any case where getting a word from the dictionary fails? Not really, because the `chooseWord` effect will never try to get a word whose index is outside the range of the dictionary length, and on the other hand all the words in the dictionary are predefined and not empty. Therefore, we can rule out the wrong case without any problem, using the `ZIO#orDieWith` method, which returns a new effect that cannot fail and, if it does fail, that would mean that there is some serious **defect** in our application (for example that some of our predefined words are empty when they should not be) and therefore it should fail immediately with the provided exception.

At the end, `chooseWord` is an effect with type `URIO[Random, Word]`, which means that:

- To run it requires `Random` the module(which is another standard module provided by ZIO, as is the `Console` module).
- It cannot fail.
- It successfully terminates with an object of type `Word`.

Functionality to show the game state by console

The next thing we need to do is to implement the functionality to show the game state by console:

```scala
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('#')
    )
  }
}
```

We will not go into too much detail about how this function is implemented, rather we will focus on a single line, which is perhaps the most interesting one:

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

We can see that what this line does, is to first of all, select which figure should be shown to represent the hangman, which is different according to the number of failures the player has had (the file `com/example/package.scala` contains a `hangmanStates` variable that consists of a list with the six possible figures). For example:

- `hangmanStates(0)` contains the hangman drawing for `failuresCount = 0`, that is, it shows only the drawing of the gallows without the hanged man
- `hangmanStates(1)` contains the hanged man drawing for `failuresCount = 1`, that is, it shows the drawing of the gallows and the head of the hanged man.

Now, the expression `hangmanStages(state.failuresCount)` is not purely functional because it could throw an exception if `state.failuresCount` is greater than 6. So, since we are working with functional programming, we cannot allow our code to produce side effects, such as throwing exceptions, that is why we have encapsulated the previous expression inside `ZIO`, which is actually a call to the method `ZIO.apply`, which allows us to build a functional effect from an expression that produces side effects (we can also use the equivalent method `ZIO.effect`). By the way, the result of calling `ZIO.apply` is an effect that can fail with a `Throwable`, but according to the design of our application we hope that there will never actually be a failure when trying to get the hanged man drawing (since `state.failuresCount` should never be greater than 6). Therefore we can rule out the wrong case by calling the `ZIO#orDie` method, which returns an effect that never fails (we already know that if there were any failure, it would actually be a **defect** and our application should fail immediately). By the way, `ZIO#orDie` is very similar to `ZIO#orDieWith`, but it can only be used with effects that fail with a `Throwable.`

Functionality to obtain a letter from the player

The functionality to obtain a letter from the player is very similar to how we obtain the name, therefore we will not go into too much detail:

```
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
```

Functionality to analyze a letter entered by the player

This functionality is very simple, all it does is to analyze the previous state and the state after a player's attempt, to see if the player wins, loses, has correctly guessed a letter but has not yet won the game, has incorrectly guessed a letter but has not yet lost the game, or if he has retried a letter that was previously tried:

```scala
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
```

Game loop implementation

The game loop implementation uses the functionalities that we defined earlier:

```scala
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 ()
```

As you can see, what the `gameLoop` function does is the following:

- Displays the status of the game by console and gets a letter from the player.
- The state of the game is updated with the new letter guessed by the player.
- The new letter guessed by the player is analyzed and:
    - If the player won, a congratulatory message is displayed and the game status is displayed for the last time.
    - If the player lost, a message is displayed indicating the word to guess and the state of the game is displayed for the last time.
    - If the player has guessed a letter correctly but has not yet won the game, a message is displayed stating that they guessed correctly and `gameLoop` is called again, with the updated state of the game.
    - If the player has guessed a letter incorrectly but has not yet lost the game, a message is displayed stating that they guessed incorrectly and `gameLoop` is called again, with the updated state of the game.

- ○ If the player tries a letter that they had guessed before, a message is displayed and `gameLoop` is called again, with the updated state of the game.

At the end, `gameLoop` returns a `ZIO` effect that depends on the `Console` module (this is obvious because it needs to read a text from and write to the console), it can fail with an `IOException` or end successfully with a `Unit` value.

Joining all the pieces

Finally, we have all the pieces of our application, and now the only thing that remains for us to dois to put them all together and call them from the `run` method which, as we have already mentioned , is the entry point of any ZIO-based application:

```
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
```

We can see the logic is very simple:

- A welcome message is printed and the player's name is requested.
- A word is chosen at random for the player to guess.
- The game loop runs.

There are some important details to explain here, for example, the expression:

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

Returns an effect of type `ZIO[Console with Random, IOException, Unit]`, and it's interesting to see how ZIO knows exactly, all the time, which modules a functional effect depends on. In this case the effect depends on the `Console` module and the `Random` module, which is obvious because our application requires printing messages to and reading from the console and generating random words. However, the `run` method requires returning an effect of type `ZIO[ZEnv, Nothing, ExitCode].` That is why we need to call the `ZIO#exitCode` method, which returns an effect of the type `ZIO[Console With Random,` `Nothing, ExitCode]`as follows:

- If the original effect ends successfully, `ExitCode.success` is returned.
- If the original effect fails, the error is printed by console and `ExitCode.failure` is returned

Now, if we look closely, we are returning an effect of the type `ZIO[Console with Random, Nothing, ExitCode]`, but `run` requires returning `ZIO[ZEnv, Nothing , ExitCode]`, so why is there no compilation error? For this we need to understand what `ZEnv` means:

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

We can see that `ZEnv` is just an alias that encompasses all of the standard modules provided by ZIO. So, `run` basically expects an effect to be returned that requires only the modules provided by ZIO, but not necessarily all, and since the effect we are trying to return requires `Console with Random`, there is no problem. You may be now wondering : is there a case where we have effects that require modules that are not provided by ZIO? And the answer is yes, because we can define our own modules. And now you may be wondering about what to do in these cases, you won't find the answer in this article, but if you want to know more about this, we have an ebook that explains how to develop modular applications with ZIO, using an implementation of a Tic-Tac-Toe game as an example.

And that's it! We have finished the implementation of a Hangman game, using a purely functional programming style, with the help of the ZIO library.

Conclusions

In this article we have been able to see how, thanks to libraries such as ZIO, we can implement complete applications using the functional programming paradigm, just by writing descriptions of interactions with the outside world (called functional effects) that can be combined with each other to form more complex descriptions. We have seen that ZIO allows us to create, combine and transform functional effects with each other in various ways, although we certainly have not seen all the possibilities that ZIO offers. However, I hope this will serve as a boost to continue exploring the fascinating ecosystem that is being built around this library.

Something important that we have not done in this article is to write unit tests for our application, and that will be precisely the topic to be discussed in a following article, where we will see how it is also possible to write unit tests, in a purely functional way, using the ZIO Test library.

Finally, if you want to learn more about ZIO, you can take a look at the following resources:

- The Official ZIO Documentation
- Articles related to ZIO on the Official Scalac Blog

References

- Github repository for this article
- Article about the Benefits of Functional Programming, in the Scalac Blog
- Functional Programming in Scala, by Paul Chiusano and Runar Bjarnason
- Articles related to ZIO, in the Official Scalac Blog
- Ebook about how to develop modular applications with ZIO, by Jorge Vásquez
- Official ZIO documentation
- Github repository with exercises to learn ZIO, by John De Goes
- Top 10 languages developers want to learn

scalac

Scale fast with Scalac

We’re a team of 120 developers ready to develop your solution. We’ve worked with over 80 companies around the world.

Click the button to find out more about our consulting and development solutions.

Find out more