SECOND EDITION

PRACTICAL FP IN SCALA

A HANDS-ON APPROACH

GABRIEL VOLPE

Practical FP in Scala

A hands-on approach

Gabriel Volpe

September 13, 2021

Second Edition

Contents

Preface

Scala is a hybrid language that mixes both the Object-Oriented Programming (OOP) and Functional Programming (FP) paradigms. This allows you to get up-and-running pretty quickly without knowing the language in detail. Over time, as you learn more, you are hopefully going to appreciate what makes Scala great: its *functional building blocks*.

Pattern matching, folds, recursion, higher-order functions, etc. If you decide to continue down this road, you will discover the functional subset of the community and its great ecosystem of libraries.

Sooner rather than later, you will come across the Cats[1] library and its remarkable documentation. You might even start using it in your projects! Once you get familiar with the power of typeclasses such as `Functor`, `Monad`, and `Traverse`, I am sure you will love it.

As you evolve into a functional programmer, you will learn about functional effects and referential transparency. You might as well start using the popular `IO` Monad present in Cats Effect[2] and other similar libraries.

One day you will need to process a lot of data that doesn't fit into memory; a suitable solution to this engineering problem is streaming. While searching for a valuable candidate, you might stumble upon a purely functional streaming library: Fs2[3]. You will quickly learn that it is also a magnificent library for control flow.

A requirement to build a RESTful API[4] will more likely come down your way early on in your career. Http4s[5] leverages the power of Cats Effect and Fs2 so you can focus on shipping features while remaining on functional land.

You might decide to adopt a message broker as a communication protocol between microservices and to distribute data. You name it: `Kafka`, `Pulsar`, `RabbitMQ`, to mention a few. Each of these wonderful technologies has a library that can fulfill every client's needs.

[1] https://typelevel.org/cats
[2] https://typelevel.org/cats-effect
[3] https://fs2.io
[4] https://restfulapi.net/
[5] https://http4s.org/

Unless you have taken the *stateless* train, you will need a database or a cache as well. Whether it is `PostgreSQL`, `ElasticSearch`, or `Redis`, the Scala FP ecosystem of libraries has got your back.

So far so good! There seems to be a wide set of tools available to write a complete purely functional application and finally ditch the enterprise framework.

At this point, you find yourself in a situation where many programmers that are enthusiastic about functional programming find themselves: needing to **deliver business value in a time-constrained manner**.

Answering this and many other fundamental questions are the aims of this book. Even if at times it wouldn't give you a straightforward answer, it will show you the way. It will give you choices and hopefully enlighten you.

Throughout the following chapters, we will develop a shopping cart application that tackles system design from different angles. We will architect our system, making both sound business and technical decisions at every step, using the best possible techniques I am knowledgeable of at this moment.

Acknowledgments

One can only dream of starting writing a book and making it over the finish line. Yet, I managed to do this twice! Though, this would have been an impossible task without the help of many people that had supported me over time, as well as many open-source and free software I consider indispensable.

I am beyond excited and can only be thankful to all of you.

People

I consider myself incredibly lucky to have had all these great human beings influencing the content of this book one way or another. This humble piece of work is dedicated:

- To my beloved partner Alicja for her endless support in life.
- To my friend John Regan for his invaluable feedback, which has raised the bar on my writing skills to the next level.
- To the talented @impurepics[1], author of the book's cover.
- To Jakub Kozłowski for reviewing almost every pull request of the book and the Shopping Cart application.
- To my OSS friends, Fabio Labella, Frank Thomas, Luka Jacobowitz, Michael Pilquist, Oleg Nizhnik, Olivier Mélois, Piotr Gawryś, Rob Norris, and Ross A. Baker, both for their priceless advice and for proofreading some of the drafts.
- To the +1500 early readers who supported my work for the extra motivation and the early feedback.
- To all the amazing volunteers that have provided incredible reviews in this second edition: Adianto Wibisono, Barış Yüksel, Bartłomiej Szwej, Bjørn Madsen, Mark Mynsted, Pavels Sisojevs, and Sinan Pehlivanoglu.

Last but not least, this edition is dedicated to all the people that make the Typelevel ecosystem as great as it is nowadays, especially to the maintainers and contributors of my two favorite Scala libraries: Cats Effect and Fs2. This book wouldn't exist without all of your work! #ScalaThankYou

Although the book was thoroughly reviewed, I am the sole responsible for all of the opinionated sentences, and any remaining mistakes are only mine.

[1] https://twitter.com/impurepics

Software

As a grateful open-source software contributor, this section is dedicated to all the free tools that have made this book possible.

- NeoVim[2]: my all-time favorite text editor, used to write this book as well as to code the Shopping Cart application.
- Pandoc[3]: a universal document converter written in Haskell, used to generate PDFs and ePub files.
- LaTeX[4]: a high-quality typesetting system to produce technical and scientific documentation, as well as books.

[2]https://neovim.io/
[3]https://pandoc.org/
[4]https://www.latex-project.org/

Fonts

This book's main font is Latin Modern Roman[5], distributed under The GUST Font License (GFL)[6]. Other fonts in use are listed below.

- JetBrainsMono[7] for code snippets, available under the SIL Open Font License 1.1[8]
- Linux Libertine[9] for some Unicode characters, licensed under the GNU General Public License version 2.0 (GPLv2)[10] and the SIL Open Font License[11].

[5] https://tug.org/FontCatalogue/latinmodernroman/
[6] https://www.ctan.org/license/gfl
[7] https://www.jetbrains.com/lp/mono/
[8] https://github.com/JetBrains/JetBrainsMono/blob/master/OFL.txt
[9] https://sourceforge.net/projects/linuxlibertine/
[10] https://opensource.org/licenses/gpl-2.0.php
[11] https://scripts.sil.org/cms/scripts/page.php?item_id=OFL

Dependency versions

At the moment of writing, all the standalone examples use Scala 2.13.5 and **sbt** 1.5.3, as well as the following dependencies defined in this minimal **build.sbt**[1] file.

```
ThisBuild / scalaVersion := "2.13.5"

lazy val root = (project in file("."))
  .settings(
    name := "minimal",
    libraryDependencies ++= Seq(
      compilerPlugin(
        "org.typelevel" %% "kind-projector" % "0.12.0"
          cross CrossVersion.full
      ),
      "org.typelevel" %% "cats-core"       % "2.6.1",
      "org.typelevel" %% "cats-effect"     % "3.1.1",
      "org.typelevel" %% "cats-mtl"        % "1.2.1",
      "co.fs2"        %% "fs2-core"        % "3.0.3",
      "dev.optics"    %% "monocle-core"    % "3.0.0",
      "dev.optics"    %% "monocle-macro"   % "3.0.0",
      "io.estatico"   %% "newtype"         % "0.4.4",
      "eu.timepit"    %% "refined"         % "0.9.25",
      "eu.timepit"    %% "refined-cats"    % "0.9.25",
      "tf.tofu" %% "derevo-cats"           % "0.12.5",
      "tf.tofu" %% "derevo-cats-tagless"   % "0.12.5",
      "tf.tofu" %% "derevo-circe-magnolia" % "0.12.5",
      "tf.tofu" %% "tofu-core-higher-kind" % "0.10.2"
    ),
    scalacOptions ++= Seq(
      "-Ymacro-annotations", "-Wconf:cat=unused:info"
    )
```

The **sbt-tpolecat** plugin is also necessary. Here is a minimal **plugins.sbt** file.

```
addSbtPlugin("io.github.davidgregory084" % "sbt-tpolecat" % "0.1.17")
```

[1] https://gist.github.com/gvolpe/04b31a5caa875f8f16bcd1d12b72face

Please note that Scala Steward[2] keeps on updating the project's dependencies on a daily basis, which may not reflect the versions described in this book.

[2]https://github.com/fthomas/scala-steward

Prerequisites

This book is considered intermediate to advanced. Familiarity with functional programming concepts and basic FP libraries such as Cats and Cats Effect will be of tremendous help even though I will do my best to be as clear and concise as I can.

Both Scala with Cats[1] and Essential Effects[2], in that order, are excellent books to learn these concepts. The official documentation of Cats Effect[3] is also a great resource.

The following list details the topics required to understand this book.

- Higher-Kinded Types (HKTs)[4].
- Typeclasses[5].
- IO Monad[6].
- Referential Transparency[7].

Unfortunately, these topics are quite lengthy to be explained in this book, so readers are expected to be acquainted with them. However, some examples will be included, and this might be all you need.

If the requirements feel overwhelming, it is not because the entire book is difficult, but rather because some specific parts might be. You can try to read it, and if at some point you get stuck, you can skip that section. You could also make a pause, go to read about these resources, and then continue where you left off.

Remember that we are going to develop an application together, which will help you learn a lot, even if you haven't employed these techniques and libraries before.

[1] https://underscore.io/books/scala-with-cats/
[2] https://essentialeffects.dev/
[3] https://typelevel.org/cats-effect/
[4] https://typelevel.org/blog/2016/08/21/hkts-moving-forward.html
[5] https://typelevel.org/cats/typeclasses.html
[6] https://typelevel.org/blog/2017/05/02/io-monad-for-cats.html
[7] https://en.wikipedia.org/wiki/Referential_transparency

How to read this book

For conciseness, most of the imports and some datatype definitions are elided from the book, so it is recommended to read it by following along the two Scala projects that supplement it.

- pfps-examples[1]: Standalone examples.
- pfps-shopping-cart[2]: Shopping cart application.

The first project includes self-contained examples that demonstrate some features or techniques explained independently.

The latter contains the source code of the full-fledged application that we will develop in the next ten chapters, including a test suite and deployment instructions.

Bear in mind that the presented Shopping Cart application only acts as a guideline. To get a better learning experience, readers are encouraged to write their own application from scratch; **getting your hands dirty is the best way to learn**.

There is also a Gitter channel[3] where you are welcome to ask any kind of questions related to the book or functional programming in general.

[1] https://github.com/gvolpe/pfps-examples
[2] https://github.com/gvolpe/pfps-shopping-cart
[3] https://gitter.im/pfp-scala/community

Conventions used in this book

Colored boxes might indicate either notes, tips, or warnings.

Notes

`A note on what's being discussed`

Tips

`A tip about a particular topic`

Warning

`Claim or decision based on the author's opinion`

If you are reading this on Kindle, you won't see colors, unfortunately.

Chapter 1: Best practices

Before we get to analyzing the business requirements and writing the application, we are going to explore some design patterns and best practices. A few well-known; others not so standard and biased towards my preferences.

These will more likely appear at least once in the application we will develop, so you can think of this chapter as a preparation for what's to come.

Strongly-typed functions

One of the most significant benefits of functional programming is that it lets us reason about functions by looking at their type signature. Yet, the truth is that these are commonly created by us, imperfect humans, who often end up with weakly-typed functions.

For instance, let's look at the following function.

```
def lookup(username: String, email: String): F[Option[User]]
```

Do you see any problems with it? Let's see how we can use it.

```
$ lookup("aeinstein@research.com", "aeinstein")
$ lookup("aeinstein", "123")
$ lookup("", "")
```

See the issue? It is not only easy to confuse the order of the parameters but it is also straightforward to feed our function with invalid data! So what can we do about it? We could make this better by introducing *value classes*.

Value classes

In vanilla Scala, we can wrap a single field and extend the `AnyVal` abstract class to avoid some runtime costs. Here is how we can define value classes for `username` and `email`.

```
case class Username(val value: String) extends AnyVal
case class Email(val value: String) extends AnyVal
```

Now we can re-define our function using these types.

```
def lookup(username: Username, email: Email): F[Option[User]]
```

Notice that we can no longer confuse the order of the parameters.

```
$ lookup(Username("aeinstein"), Email("aeinstein@research.com"))
```

Or can we?

```
$ lookup(Username("aeinstein@research.com"), Email("aeinstein"))
$ lookup(Username("aeinstein"), Email("123"))
$ lookup(Username(""), Email(""))
```

Fine, we are doing this on purpose. However, in a statically-typed language, we would expect the compiler to help prevent this but it cannot due to lack of information. A way to communicate our intentions to the compiler is to make the case class constructors private only expose *smart constructors*.

```scala
case class Username private(val value: String) extends AnyVal
case class Email private(val value: String) extends AnyVal

def mkUsername(value: String): Option[Username] =
  (value.nonEmpty).guard[Option].as(Username(value))

def mkEmail(value: String): Option[Email] =
  (value.contains("@")).guard[Option].as(Email(value))
```

Smart constructors are functions such as `mkUsername` and `mkEmail`, which take a raw value and return an optional validated one. The optionality can be denoted using types such as `Option`, `Either`, `Validated`, or any other higher-kinded type.

So let's pretend that these functions validate the raw values properly and give us back some valid data. We can now use them in the following way.

```scala
(
  mkUsername("aeinstein"),
  mkEmail("aeinstein@research.com")
).mapN {
  case (username, email) ⇒ lookup(username, email)
}
```

But guess what? We can still do wrong...

```scala
(
  mkUsername("aeinstein"),
  mkEmail("aeinstein@research.com")
).mapN {
  case (username, email) ⇒
    lookup(username.copy(value = ""), email)
}
```

Unfortunately, we are still using case classes, which means the `copy` method is still there. A proper way to finally get around this issue is to use `sealed abstract case class`es.

```scala
sealed abstract case class Username(value: String)
sealed abstract case class Email(value: String)
```

Or `sealed abstract class`es, where we need to add the `val` keyword to make `value` accessible from the outside.

```scala
sealed abstract class Username(val value: String)
sealed abstract class Email(val value: String)
```

Having this encoding in combination with smart constructors will mitigate the issue at the cost of boilerplate and more memory allocation.

Newtypes

Value classes are fine in most cases, but we haven't talked about their limitations and performance issues. In many cases, Scala needs to allocate extra memory when using value classes, as described in the article Value classes and universal traits[1]. Quoting the relevant part:

> A value class is actually instantiated when:
>
> - a value class is treated as another type.
> - a value class is assigned to an array.
> - doing runtime type tests, such as pattern matching.

The language cannot guarantee that these primitive type wrappers won't actually allocate more memory, in addition to the pitfalls described in the previous section.

Thus, my recommendation is to avoid value classes and sealed abstract classes completely and instead use the Newtype[2] library, which gives us *zero-cost wrappers* with no runtime overhead.

This is how we can define our data using newtypes.

```
import io.estatico.newtype.macros._

@newtype case class Username(value: String)
@newtype case class Email(value: String)
```

It uses macros, for which we need the macro paradise compiler plugin in Scala versions below 2.13.0, and only an extra compiler flag `-Ymacro-annotations` in versions 2.13.0 and above.

Despite eliminating the extra allocation issue and removing the `copy` method, notice how we can still trigger the functionality incorrectly.

```
Email("foo")
```

This means that smart constructors are still needed to avoid invalid data.

Notes

```
Newtypes do not solve validation; they are just zero-cost wrappers
```

Newtypes can also be constructed using the `coerce` method in the following way.

[1] https://docs.scala-lang.org/overviews/core/value-classes.html
[2] https://github.com/estatico/scala-newtype

```
import io.estatico.newtype.ops._

"foo".coerce[Email]
```

Though, this is considered an *anti-pattern*, and its use is highly discouraged when we know the concrete type. The only reason for its existence is so we can write polymorphic code for newtypes, which we will rarely ever need.

Refinement types

We have seen how newtypes help us tremendously in our strongly-typed functions quest. Nevertheless, it requires smart constructors to validate input data, which adds boilerplate and leaves us with a bittersweet feeling. Still, do not give up hope as we have one last card to play: refinement types, provided by the Refined[3] library.

Refinement types allow us to validate data at compile time as well as at runtime. Let's see an example.

```
import eu.timepit.refined.types.string.NonEmptyString

def lookup(username: NonEmptyString): F[Option[User]]
```

We are saying that a valid username is any non-empty string; though, we could also say that a valid username is any string containing the letter 'g', in which case, we would need to define a custom refinement type instead of using a built-in one like `NonEmptyString`. The following example demonstrates how we can do this.

```
import eu.timepit.refined.api.Refined
import eu.timepit.refined.collection.Contains

type Username = String Refined Contains['g']

def lookup(username: Username): F[Option[User]]
```

By saying that it should contain a letter 'g' (using string literals), we are also implying that it should be non-empty. If we try to pass some invalid arguments, we are going to get a compiler error.

```
import eu.timepit.refined.auto._

$ lookup("")          // error
$ lookup("aeinstein") // error
$ lookup("csagan")    // compiles
```

[3] https://github.com/fthomas/refined

Refinement types are great and let us define custom validation rules. Though, in many cases, a simple rule applies to many possible types. For example, a `NonEmptyString` applies to almost all our inputs. In such cases, we can combine forces and use Refined and Newtype together!

```
@newtype case class Brand(value: NonEmptyString)
@newtype case class Category(value: NonEmptyString)

val brand: Brand = Brand("foo")
```

These two types share the same validation rule, so we use refinement types, but since they represent different concepts, we create a newtype for each of them. This combination is ever so powerful that I couldn't recommend it enough.

Another feature that makes the Refined library very appealing is its integration with multiple libraries such as Circe[4], Doobie[5], and Monocle[6], to name a few. Having support for these third-party libraries means that we don't need to write custom refinement types to integrate with them as the most common ones are provided out of the box.

Runtime validation

Up until now, we have seen how refinement types help us validate data at compile time, as well as combining them together with newtypes. Yet, we haven't talked much about runtime validation, which is something we need in the real world.

Almost every application needs to deal with runtime validation. For example, we can not possibly know what values we are going to receive from HTTP requests or any other service, so compile-time validation is not an option here.

Refined gives us a generic function for this purpose, which is roughly defined as follows.

```
def refineV[P]: RefinePartiallyApplied[P] =
  new RefinePartiallyApplied[P]

final class RefinePartiallyApplied[P] {
  def apply[T](t: T)(
    implicit v: Validate[T, P]
  ): Either[String, Refined[T, P]]
}
```

It is not a coincidence that the type parameter is named `P`, which stands for predicate.

In the following example, pretend `str` represents an actual runtime value.

[4] https://github.com/circe/circe
[5] https://github.com/tpolecat/doobie
[6] https://github.com/optics-dev/Monocle

```scala
import eu.timepit.refined._

val str: String = "some runtime value"

val res: Either[String, NonEmptyString] =
  refineV[NonEmpty](str)
```

Most refinement types provide a convenient `from` method, which take the raw value and returns a validated one or an error message. For example, the following example is equivalent to the one above.

```scala
val res: Either[String, NonEmptyString] =
  NonEmptyString.from(str)
```

It also helps with type inference so it is recommended to use `from` over the generic `refineV`. We can add the same feature to any custom refinement type too.

```scala
import eu.timepit.refined.api.RefinedTypeOps
import eu.timepit.refined.numeric.Greater

type GTFive = Int Refined Greater[5]
object GTFive extends RefinedTypeOps[GTFive, Int]

val number: Int = 33

val res: Either[String, GTFive] = GTFive.from(number)
```

Summarizing, Refined lets us perform runtime validation via `Either`, which forms a Monad. This means validation is done sequentially. It would fail on the first error encountered during multiple value validation. In such cases, it is usually a better choice to go for `cats.data.Validated`, which is similar to `Either`, except it only forms an `Applicative`.

In practical terms, this means it can validate data simultaneously and accumulate errors instead of validating data sequentially and failing fast on the first encountered error. A common type for such purpose is `ValidatedNel[E, A]`, which is an alias for `Validated[NonEmptyList[E], A]`. We can convert those refinement results to this type via the `toValidatedNel` extension method.

```scala
case class MyType(a: NonEmptyString, b: GTFive)

def validate(a: String, b: Int): ValidatedNel[String, MyType] =
  (
    NonEmptyString.from(a).toValidatedNel,
    GTFive.from(b).toValidatedNel
  ).mapN(MyType.apply)
```

Evaluating this function with `a = ""` and `b = 3` yields the following result.

```
Invalid(
  NonEmptyList(Predicate isEmpty() did not fail.,
  Predicate failed: (3 > 5).)
)
```

We could get to the same result via `toEitherNel` + `parMapN` instead.

```
def validate(a: String, b: Int): EitherNel[String, MyType] =
  (
    NonEmptyString.from(a).toEitherNel,
    GTFive.from(b).toEitherNel
  ).parMapN(MyType.apply)
```

Evaluating this function with the previous inputs yields a similar result.

```
Left(
  NonEmptyList(Predicate isEmpty() did not fail.,
  Predicate failed: (3 > 5).)
)
```

Except it returns `Left` instead of `Invalid`.

Behind the scenes, what makes this work is the `cats.Parallel` instance for `Either` and `Validated`, which abstracts over monads which support parallel composition via some related `Applicative`.

```
implicit def ev[E: Semigroup]: Parallel.Aux[Either[E, *], Validated[E, *]] =
    new Parallel[Either[E, *]] { ... }
```

We are able to accumulate errors because of the `Semigroup` constraint on `E`. In our examples, `E = String`.

Furthermore, since we generally use newtypes together with refinement types, there is something else to consider. Let's look at the following `Person` domain model.

```
type UserNameR = NonEmptyString
object UserNameR extends RefinedTypeOps[UserNameR, String]

type NameR = NonEmptyString
object NameR extends RefinedTypeOps[NameR, String]

type EmailR = String Refined Contains['@']
object EmailR extends RefinedTypeOps[EmailR, String]

@newtype case class UserName(value: UserNameR)
```

```scala
@newtype case class Name(value: NameR)
@newtype case class Email(value: EmailR)

case class Person(
    username: UserName,
    name: Name,
    email: Email
)
```

To perform validation, we will need an extra `map` to lift the refinement type into our newtype, in addition to `toEitherNel`. E.g.

```scala
def mkPerson(
  u: String,
  n: String,
  e: String
): EitherNel[String, Person] =
  (
    UserNameR.from(u).toEitherNel.map(UserName.apply),
    NameR.from(n).toEitherNel.map(Name.apply),
    EmailR.from(e).toEitherNel.map(Email.apply)
  ).parMapN(Person.apply)
```

It gets the job done at the cost of being repetitive and maybe a bit boilerplatey. Now what if I told you this pattern can be abstracted away and reduced down to this?

```scala
import NewtypeRefinedOps._

def mkPerson(
  u: String,
  n: String,
  e: String
): EitherNel[String, Person] =
  (
    validate[UserName](u),
    validate[Name](n),
    validate[Email](e)
  ).parMapN(Person.apply)
```

Interesting, isn't it? This is one of the exceptional cases where I think resorting to the infamous Coercible[7] typeclass, from the Newtype library, is more than acceptable.

```scala
object NewtypeRefinedOps {
  import io.estatico.newtype.Coercible
```

[7] https://github.com/estatico/scala-newtype#coercible

```
  import io.estatico.newtype.ops._

  final class NewtypeRefinedPartiallyApplied[A] {
    def apply[T, P](raw: T)(implicit
        c: Coercible[Refined[T, P], A],
        v: Validate[T, P]
    ): EitherNel[String, A] =
      refineV[P](raw).toEitherNel.map(_.coerce[A])
  }

  def validate[A]: NewtypeRefinedPartiallyApplied[A] =
    new NewtypeRefinedPartiallyApplied[A]
}
```

We could also make it work as an extension method of the raw value, though, this requires two method calls instead.

```
(
  u.as[UserName].validate,
  n.as[Name].validate,
  e.as[Email].validate
).parMapN(Person.apply)
```

You can refer to the source code for the implementation. We will skip it because it is very similar to the **`validate`** function we've seen above.

I hope it was enough to convince you of the benefits of this ever powerful duo! Throughout the development of the shopping cart application, we will get acquainted with this technique, as it will be ubiquitous.

Encapsulating state

Mostly every application needs to thread some kind of state, and in functional Scala, we have great tools to manage it properly. Whether we use `MonadState`, `StateT`, `MVar`, or `Ref`, we can write good software by following some design guidelines.

One of the best approaches to managing state is to encapsulate state in the interpreter and only expose an abstract interface with the functionality the user needs.

> **Tips**
>
> `Our interface should know nothing about state`

By doing so, we control exactly how the users interact with state. Conversely, if we use something like `MonadState[F, AppState]` or `Ref[F, AppState]` directly, functions can potentially access and modify the entire state of the application at any given time (unless used together with *classy lenses*, which are a bit more advanced and less obvious than using plain old interfaces).

In-memory counter

Let's say we need an in-memory counter that needs to be accessed and modified by other components. Here is what our interface could look like.

```
trait Counter[F[_]] {
  def incr: F[Unit]
  def get: F[Int]
}
```

It has a higher-kinded type `F[_]`, representing an abstract effect, which most of the time ends up being `IO`, but it could really be any other concrete type that fits the shape.

Next, we need to define an interpreter in the companion object of our interface, in this case using a `Ref`. We will talk more about it in the next section.

```
import cats.Functor
import cats.effect.kernel.Ref
import cats.syntax.functor._

object Counter {
  def make[F[_]: Functor: Ref.Make]: F[Counter[F]] =
    Ref.of[F, Int](0).map { ref ⇒
      new Counter[F] {
        def incr: F[Unit] = ref.update(_ + 1)
        def get: F[Int]   = ref.get
```

```
      }
    }
}
```

By exposing a smart constructor as above, we make it impossible for the `Ref` to be accessed outside of it. This has a fundamental reason: *state shall not leak*. If we had instead, taken the `Ref` as an argument to our smart constructor, it could be potentially misused in other places.

Furthermore, since the creation of a `Ref` is effectful (it allocates a mutable reference), the constructor returns `F[Counter[F]]`, which needs to be `flatMap`ed at call site to create and access the inner `Counter[F]`.

> **Tips**
>
> `Remember that a new Counter will be created on every flatMap call`

Moreover, notice the typeclass constraints used in the interpreter: `Functor` and `Ref.Make`. This is all we need, though, both constraints could be subsumed by a single `Sync` constraint, if we wanted that. However, it is preferred to avoid hard constraints that enable FFI (Foreign Function Interface), i.e. side-effects.

We could also create the interpreter as a class, e.g. `LiveCounter`, instead of doing it via an *anonymous class* in the smart constructor. This is how it was done in the first edition of this book but my preferences have shifted towards the former over time. See below.

```
object LiveCounter {
  def make[F[_]: Sync]: F[Counter[F]] =
    Ref.of[F, Int](0).map(new LiveCounter[F](_))
}

class LiveCounter[F[_]] private (
    ref: Ref[F, Int]
) extends Counter[F] {
  def incr: F[Unit] = ref.update(_ + 1)
  def get: F[Int]   = ref.get
}
```

This is up to you; go with the one you favour the most and be consistent about it. However, be aware that in such cases, we need to make the interpreter's constructor *private*. Otherwise, we would be allowing its construction with arbitrary instances of a `Ref` constructed somewhere else.

Moving on, it's worth highlighting that other programs will interact with this counter solely via its interface. E.g.