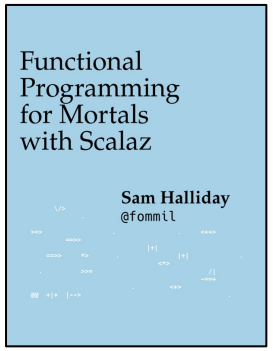
Abstracting over Execution with Higher Kinded Types

and how to remain Purely Functional

study aid for the introductory chapter of

Functional Programming for Mortals with Scalaz

the book by **Sam Halliday**



@fommil

supplemented with code from https://github.com/fommil/fpmortals

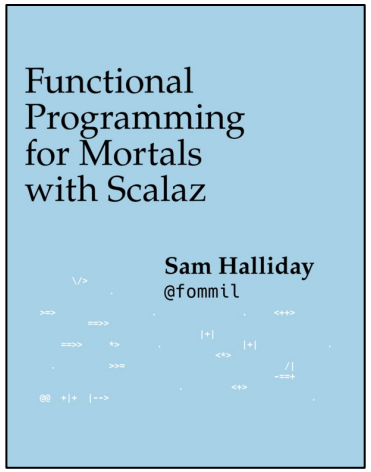
slides by  @philip_schwarz

We want to interact with the user over the command line interface. We can **read** what the user types and we can **write** a message to them.

```scala
trait TerminalSync {
  def read: String
  def write(t: String): Unit
}
```

```scala
trait TerminalAsync {
  def read: Future[String]
  def write(t: String): Future[Unit]
}
```

Sam Halliday



@fommil

How do we write **generic code** that does something as simple as **echo** the user's input **sync**hronously or **async**hronously depending on our **runtime implementation**?

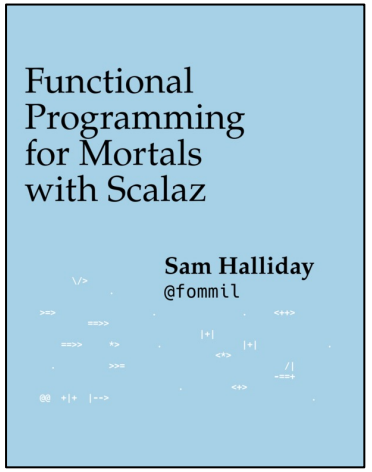
We can solve the problem with a **common parent** using the **higher kinded types** **Scala** language feature, which allow us to use a **type constructor** in our **type parameters**, which looks like C[_]. This is a way of saying that whatever **C** is, it must take a **type parameter**.
e.g. `List` is a **type constructor** because it takes a type (e.g. `Int`) and constructs a type (`List[Int]`).

```
trait Terminal[C[_]] {
  def read: C[String]
  def write(t: String): C[Unit]
}
```

There is this one weird trick we can use when we want to ignore the **type constructor**.
We can define a **type alias** to be equal to its parameter: `type Id[T] = T`.

We want to define `Terminal` for a type constructor **C**[_]. By defining `Now` to construct to its type parameter (like `Id`), we can implement a **common interface** for synchronous and asynchronous terminals.

Sam Halliday



@fommil

```
object TerminalSync extends Terminal[Now] {
  def read: String = StdIn.readLine
  def write(t: String): Unit = println(t)
}
```

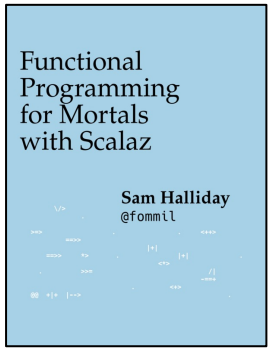
```
type Now[X] = X
```

We can think of **C** as a **context** because we say “in the context of executing Now” or “in the Future”

```
class TerminalAsync(implicit EC: ExecutionContext) extends Terminal[Future] {
  def read: Future[String] = Future { StdIn.readLine }
  def write(t: String): Future[Unit] = Future { println(t) }
}
```

Sam Halliday @fommil

```
trait Terminal[C[_]] {
  def read: C[String]
  def write(t: String): C[Unit]
}
```



We want to write **generic code** to **echo** the user's input **sync**hronously or **async**hronously depending on our **runtime implementation**, but **we know nothing about C and we can't do anything with a C[String]**.

```
/* Read (from a terminal) a string (in some context C, with some effect C)
   Then write the string (back to the terminal)
   Finally return the string (within its enclosing context/effect C) */
def echo[C[_]](t:Terminal[C]): C[String] = {
  val context: C[String] = t.read
  val text: String = /* ... extract the string contained in context C, but how? we know nothing about C ... */ ???
  t.write(text)
  context
}
```

@philip_schwarz

What we need is a kind of **execution environment** that lets us call a method returning C[T] and then be able to do something with the T, including calling another method on Terminal. We also need a way of wrapping a value as a C[_].

This signature works well

letting us write

```
trait Execution[C[_]] {
  def doAndThen[A, B](c: C[A])(f: A => C[B]): C[B]
  def create[B](b: B): C[B]
}
```

```
def echo[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  e.doAndThen(t.read) { text:String =>
    e.doAndThen(t.write(text)) { _:Unit =>
      e.create(text)
    }
  }
```

We can now share the **echo** implementation between synchronous and asynchronous codepaths. We can write a mock implementation of Terminal[Now] and use it in our tests without any timeouts. **Implementations of Execution[Now] and Execution[Future] are reusable by generic methods like echo**.

This **doAndThen** invocation extracts the text string (unboxes it) from the **context**/**effect C** returned by t.**read**, passes it to a function (the rest of the **echo** method's logic), and returns the result of that function, which is in the same **context**/**effect C**.

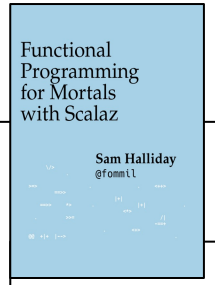
@philip_schwarz

This **doAndThen** invocation extracts the contents of a **context**/**effect C** (unboxes them) returned by t.**write** (in this case a `Unit` which we don't care about), passes it to a function (the rest of the **echo** method's logic), and returns the result of that function, which is in the same **context**/**effect C**.

create lifts a text string into a **context**/**effect C** (puts it in a box – boxes the string in **context**/**effect C**)

```
def echo[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  e.doAndThen(t.read) { text:String =>
    e.doAndThen(t.write(text)) { _:Unit =>
      e.create(text)
    }
  }
```

Sam Halliday @fommil




```
trait Terminal[C[_]] {
  def read: C[String]
  def write(t: String): C[Unit]
}
```

```
trait Execution[C[_]] {
  def doAndThen[A, B](c: C[A])(f: A => C[B]): C[B]
  def create[B](b: B): C[B]
}
```

running the echo method in the context of executing Now

```scala
trait Terminal[C[_]] {
  def read: C[String]
  def write(t: String): C[Unit]
}
```

```scala
trait Execution[C[_]] {
  def doAndThen[A, B](c: C[A])(f: A => C[B]): C[B]
  def create[B](b: B): C[B]
}
```

implementing `Terminal[Now]`

```scala
object TerminalSync extends Terminal[Now] {
  def read: Now[String] = StdIn.readLine
  def write(t: String): Now[Unit] = println(t)
}
```

```scala
implicit val executionNow = new Execution[Now] {
  def doAndThen[A, B](c: A)(f: A => Now[B]): Now[B] = f(c)
  def create[B](b: B): Now[B] = b
}
```

Implementing and instantiating `Execution[Now]`

```scala
implicit val terminalNow: Terminal[Now] = TerminalSync
```

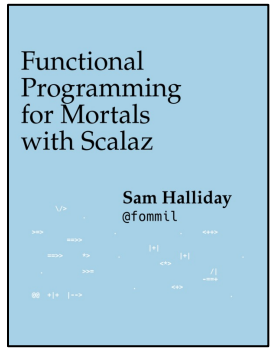
instantiating `Terminal[Now]`

```scala
type Now[X] = X
```

Implementations of `Terminal[Now]` and `Execution[Now]` are reusable by generic methods like **echo**

```scala
def echo[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  e.doAndThen(t.read) { text:String =>
    e.doAndThen(t.write(text)) { _:Unit =>
      e.create(text)
    }
  }
```

Sam Halliday @fommil




```scala
val input: Now[String] = echo[Now]
```

instantiating generic method `echo[C[_]]` for `Now` and invoking it with implicit `terminalNow:Terminal[Now]` and `executionNow:Execution[Now]`

same as previous slide but with Now[String] and Now[B] replaced with String and B since Now[X] = X

```
trait Terminal[C[_]] {
  def read: C[String]
  def write(t: String): C[Unit]
}
```

```
trait Execution[C[_]] {
  def doAndThen[A, B](c: C[A])(f: A => C[B]): C[B]
  def create[B](b: B): C[B]
}
```

implementing Terminal[Now]

```
object TerminalSync extends Terminal[Now] {
  def read: String = StdIn.readLine
  def write(t: String): Unit = println(t)
}
```

```
implicit val executionNow = new Execution[Now] {
  def doAndThen[A, B](c: A)(f: A => B): B = f(c)
  def create[B](b: B): B = b
}
```

Implementing and instantiating Execution[Now]

```
implicit val terminalNow: Terminal[Now] = TerminalSync
```

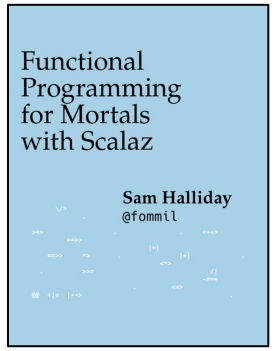
instantiating Terminal[Now]

```
type Now[X] = X
```

Implementations of Terminal[Now]]and Execution[Now] are reusable by generic methods like **echo**

```
def echo[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  e.doAndThen(t.read) { text:String =>
    e.doAndThen(t.write(text)) { _:Unit =>
      e.create(text)
    }
  }
```

Sam Halliday **@fommil**




```
val input: String = echo[Now]
```

instantiating generic method echo[C[_]] for Now and invoking it with implicit terminalNow :Terminal[Now] and executionNow:Execution[Now]

running the echo method in the Future

```scala
trait Terminal[C[_]] {
  def read: C[String]
  def write(t: String): C[Unit]
}
```

```scala
trait Execution[C[_]] {
  def doAndThen[A, B](c: C[A])(f: A => C[B]): C[B]
  def create[B](b: B): C[B]
}
```

implementing `Terminal[Future]`

```scala
object TerminalAsync extends Terminal[Future] {
  def read:Future[String]=Future{ StdIn.readLine }
  def write(t:String):Future[Unit]=Future{ println(t) }
}
```

Implementing and instantiating `Execution[Future]`

```scala
implicit val executionFuture = new Execution[Future] {
def doAndThen[A,B](c:Future[A])(f:A=>Future[B]):Future[B]=
  c flatMap f
def create[B](b:B):Future[B] =
  Future.successful(b)
}
```

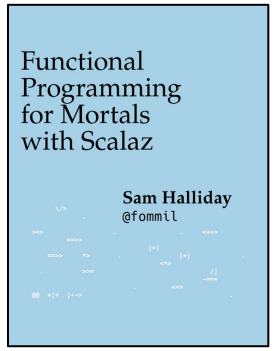
```scala
implicit val terminalFuture:Terminal[Future]=TerminalAsync
```

instantiating `Terminal[Future]`

Implementations of `Terminal[Future]` and `Execution[Future]` are reusable by generic methods like **echo**

```scala
def echo[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  e.doAndThen(t.read) { text:String =>
    e.doAndThen(t.write(text)) { _:Unit =>
      e.create(text)
    }
  }
```

Sam Halliday @fommil




```scala
val input: Future[String] = echo[Future]
```

instantiating generic method `echo[C[_]]` for `Future` and invoking it with implicit `terminalFuture:Terminal[Future]` and `executionFuture:Execution[Future]`

running a greet method in the context of executing Now

```
def greet[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  e.doAndThen(t.write(s"Hello, what is your name?")) { _ =>
    e.doAndThen(t.read) { name =>
      e.doAndThen(t.write(s"Nice to meet you, $name.")) { _ =>
        e.create(name)
      }
    }
  }

println("About to run greet[Now]")
val name: String = greet[Now]
println(s"The result was $name.")
```

@philip_schwarz

```
About to run greet[Now]
Hello, what is your name?
John
Nice to meet you, John.
The result was John
```

running the greet method in the Future

@philip_schwarz

```
def greet[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  e.doAndThen(t.write(s"Hello, what is your name?")) { _ =>
    e.doAndThen(t.read) { name =>
      e.doAndThen(t.write(s"Nice to meet you, $name.")) { _ =>
        e.create(name)
      }
    }
  }

println("About to run greet[Future]")
val futureName: Future[String] = greet[Future]
Try {
  Await.result(futureName, Duration(10,"seconds"))
} match {
  case Success(name)  => println(s"The result was $name")
  case Failure(e) => e match {
    case e:TimeoutException => println(s"Error: no name was entered within the allowed time window!")
    case _ => println(s"The following exception occurred running greet[Future]: $e")
  }
}
```

```
About to run greet[Future]
Hello, what is your name?
John
Nice to meet you, John.
The result was John
```

```
Hello, what is your name?
Error: no name was entered within the allowed time window!
```

In the previous section, we **abstracted over execution** and defined `echo[Id]` and `echo[Future]`.

We might reasonably expect that calling any `echo` will not perform any **side effects**, because it is **pure**. **However, if we use Future or Id as the execution context, our application will start listening to stdin**:

```
val futureEcho: Future[String] = echo[Future]
```

We have broken purity and are no longer writing FP code: `futureEcho` is the result of running `echo` once. Future **conflates the definition of a program with interpreting it (running it). As a result, applications built with** Future **are difficult to reason about**.

…

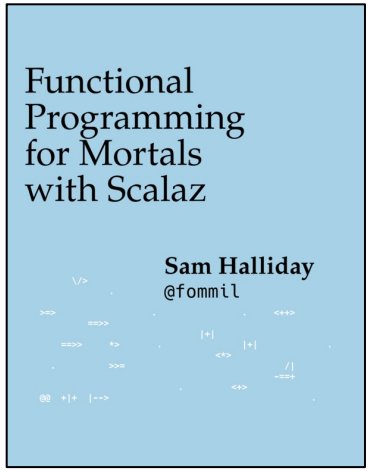
An expression is **referentially transparent** if it can be replaced with its corresponding value without changing the program's behaviour.

…

We cannot replace `echo[Future]` with a value, such as `val futureEcho`, since the pesky user will probably type something different the second time.

Sam Halliday

@fommil



Functional Programming is the act of **writing programs with pure functions**. Pure functions have three properties:

- **Total**: return a value for every possible input
- **Deterministic**: return the same value for the same input
- **Inculpable**: no (direct) interaction with the world or program state.

…

The kinds of things that break these properties are **side effects**…

We write **pure functions** by avoiding exceptions, and interacting with the world only through a safe **F**[_] execution context.

…

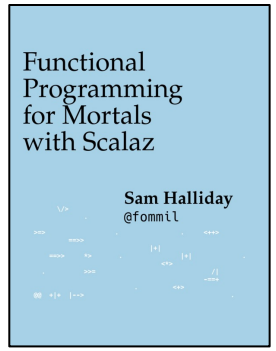
We can define a simple safe **F**[_] execution context which **lazily evaluates a thunk**.

```
final class IO[A] private (val interpret: () => A) {
  def map[B](f: A => B): IO[B]         = IO(f(interpret()))
  def flatMap[B](f: A => IO[B]): IO[B] = IO(f(interpret()).interpret())
}
object IO {
  def apply[A](a: =>A): IO[A] = new IO(() => a)
}
```

IO is just a data structure that references (potentially) impure code, it isn't actually running anything. We can implement `Terminal[IO]`

```
object TerminalIO extends Terminal[IO] {
  def read: IO[String]           = IO { StdIn.readLine }
  def write(t: String): IO[Unit] = IO { println(t) }
}
```

Sam Halliday @fommil



running the echo method using IO

```scala
final class IO[A] private (val interpret: () => A) {
  def map[B](f: A => B): IO[B]          = IO(f(interpret()))
  def flatMap[B](f: A => IO[B]): IO[B] = IO(f(interpret()).interpret())
}
object IO {
  def apply[A](a: =>A): IO[A] = new IO(() => a)
}
```

```scala
trait Terminal[C[_]] {
  def read: C[String]
  def write(t: String): C[Unit]
}
```

```scala
trait Execution[C[_]] {
  def doAndThen[A, B](c: C[A])(f: A => C[B]): C[B]
  def create[B](b: B): C[B]
}
```

```scala
object TerminalIO extends Terminal[IO] {
  def read: IO[String] =
    IO { StdIn.readLine }
  def write(t: String): IO[Unit] =
    IO { println(t) }
}
```

```scala
implicit val deferred: Execution[IO] = new Execution[IO] {
  def doAndThen[A, B](c: IO[A])(f: A => IO[B]): IO[B] =
    c flatMap f
  def create[B](b: B): IO[B] =
    IO(b)
}
```

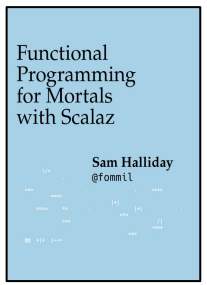
Implementing and instantiating `Execution[IO]`

```scala
implicit val io: Terminal[IO] = TerminalIO
```

instantiating `Terminal[IO]`

```scala
def echo[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  e.doAndThen(t.read) { text:String =>
    e.doAndThen(t.write(text)) { _:Unit =>
      e.create(text)
    }
  }
```

Sam Halliday @fommil




```scala
val delayed: IO[String] = echo[IO]
val input = delayed.interpret()
```

instantiating generic method `echo[C[_]]` for `IO` and invoking it with implicit `io:Terminal[IO]` and `deferred:Execution[IO]`

We can call `echo[IO]` to get back a value

```
val delayed: IO[String] = echo[IO]
```

This `val delayed` can be reused, it is just the definition of the work to be done. We can map the `String` and compose additional programs, much as we would map over a `Future`. `IO` keeps us honest that we are depending on some interaction with the world, but does not prevent us from accessing the output of that interaction.

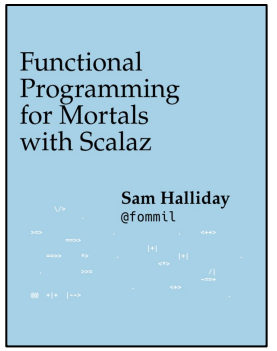
The **impure code** inside the `IO` is only evaluated when we `.interpret()` the value, which is an impure action

```
delayed.interpret()
```

An application composed of `IO` programs is only interpreted once, in the main method, which is also called the end of the world.

In this book, we expand on the concepts introduced in this chapter and show how to write maintainable, **pure functions**, that achieve your business's objectives.

Sam Halliday @fommil



running the greet method using IO

```
def greet[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  e.doAndThen(t.write(s"Hello, what is your name?")) { _ =>
    e.doAndThen(t.read) { name =>
      e.doAndThen(t.write(s"Nice to meet you, $name.")) { _ =>
        e.create(name)
      }
    }
  }

println("About to run greet[IO]")
val deferredName : IO[String] = greet[IO]
println("About to execute the result of greet[IO]")
val name: String = deferredName.interpret()
println(s"The result was $name.")
```

@philip_schwarz

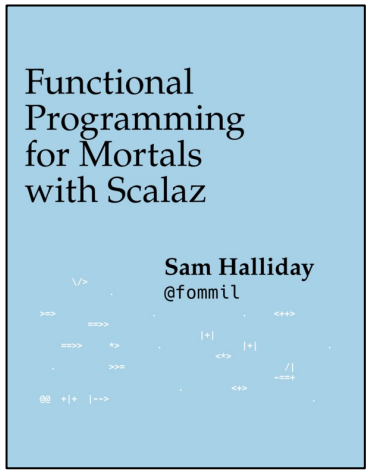
```
About to run greet[IO]
About to execute the result of greet[IO]
Hello, what is your name?
John
Nice to meet you, John.
The result was John.
```

```scala
def echo[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  e.doAndThen(t.read) { text:String =>
    e.doAndThen(t.write(text)) { _:Unit =>
      e.create(text)
    }
  }
```

The code for **echo** is horrible! Let's clean it up.

The `implicit class` **Scala** language feature gives **C** some methods. We'll call these methods **flatMap** and **map** for reasons that will become clearer in a moment. Each method takes an implicit `Execution[C]`, but this is nothing more than the **flatMap** and **map** that you're used to on **Seq**, **Option** and **Future**.

Sam Halliday



@fommil

```scala
implicit class Ops[A, C[_]](c: C[A]) {
  def flatMap[B](f: A => C[B])(implicit e: Execution[C]): C[B] =
    e.doAndThen(c)(f)
  def map[B](f: A => B)(implicit e: Execution[C]): C[B] =
    e.doAndThen(c)(f andThen e.create)
}
```

```scala
implicit val executionFuture = new Execution[Future] {
  def doAndThen[A,B](c:A)(f:A=>Future[B]):Future[B] = c flatMap f
  def create[B](b:B):Future[B] = Future.successful(b)
}
```

```scala
def echo[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  t.read.flatMap { in: String =>
    t.write(in).map { _: Unit =>
      in
    }
  }
```

scala.concurrent.Future

```
def flatMap[S](f: T => Future[S])(implicit executor: ExecutionContext): Future[S]
```

Creates a new future by applying a function to the successful result of this future, and returns the result of the function as the new future. If this future is completed with an exception then the new future will also contain this exception. Example:

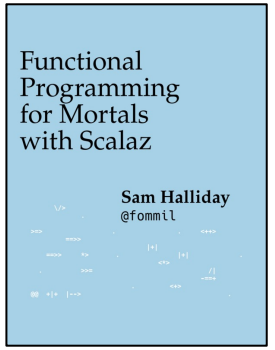
We can now reveal why we used **flatMap** as the method name: it lets us use a **for comprehension**, which is just syntax sugar over nested **flatMap** and **map**.

```
def echo[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  for {
    in <- t.read
     _ <- t.write(in)
  } yield in
```

Our **Execution** has the same signature as a trait in **scalaz** called **Monad**, except doAndThen is **flatMap** and create is **pure**. We say that **C** is **monadic** when there is an implicit **Monad**[**C**] available. In addition, **scalaz** has the **Id** type alias.

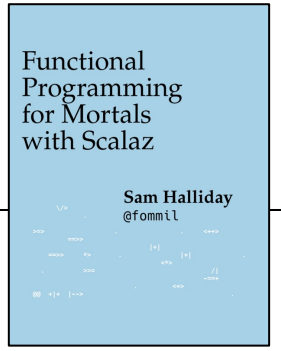
The takeaway is: **if we write methods that operate on monadic types, then we can write sequential code that abstracts over its execution context**. Here, we have shown an abstraction over **sync**hronous and **async**hronous execution but it can also be for the purpose of more rigorous error handling (where **C**[_] is **Either**[**Error**, _]), managing access to volatile state, performing I/O, or auditing of the session.

Sam Halliday @fommil



```scala
implicit class Ops[A, C[_]](c: C[A]) {
  def flatMap[B](f: A => C[B])(implicit e: Execution[C]): C[B] =
    e.doAndThen(c)(f)
  def map[B](f: A => B)(implicit e: Execution[C]): C[B] =
    e.doAndThen(c)(f andThen e.create)
}
```

Sam Halliday @fommil




```scala
implicit val executionNow: Execution[Now] = new Execution[Now] {
  def doAndThen[A, B](c: A)(f: A => B): B = f(c)
  def create[B](b: B): B                  = b
}

implicit def executionFuture (implicit EC: ExecutionContext): Execution[Future] = new Execution[Future] {
    def doAndThen[A, B](c: Future[A])(f: A => Future[B]): Future[B] = c flatMap f
    def create[B](b: B): Future[B] = Future.successful(b)
  }

implicit val deferred: Execution[IO] = new Execution[IO] {
  def doAndThen[A, B](c: IO[A])(f: A => IO[B]): IO[B] = c flatMap f
  def create[B](b: B): IO[B]                          = IO(b)
}
```

together with implicit class **Ops**, these implicit methods allow us to call `Terminal[C]` methods in a **for** comprehension.

```scala
def echo[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  for {
    in <- t.read
     _ <- t.write(in)
  } yield in
```

Running the greet method using Option

@philip_schwarz

```
class TerminalMaybe extends Terminal[Option] {
  def read: Option[String] = Some(StdIn.readLine).filter(_.trim != "")
  def write(t: String): Option[Unit] = Some(println(t))
}

implicit val maybeExecution: Execution[Option] = new Execution[Option] {
  def doAndThen[A, B](c: Option[A])(f: A => Option[B]): Option[B] = c flatMap f
  def create[B](b: B): Option[B] = Some(b)
}

implicit val maybeTerminal: Terminal[Option] = new TerminalMaybe

def greet[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  for {
        _ <- t.write(s"Hello, what is your name?")
    name <- t.read
        _ <- t.write(s"Nice to meet you, $name.")
  } yield name

println("About to run greet[Option]")
val maybeName: Option[String] = greet[Option]
maybeName match {
  case Some(name) => println(s"The result was $name.")
  case None => println("No name was entered!")
}
```

```
About to run greet[Option]
Hello, what is your name?
John
Nice to meet you, John.
The result was John.
```

```
About to run greet[Option]
Hello, what is your name?

No name was entered!
```

Running the greet method using Either

@philip_schwarz

```
type ValidatedName[A] = Either[String, A]

class TerminalValidated extends Terminal[ValidatedName] {
  def read: ValidatedName[String] = Right(StdIn.readLine).filterOrElse(!_.isEmpty,     "not supplied!")
                                                          .filterOrElse(_.head.isUpper, "not capitalised!")
                                                          .filterOrElse(_.length > 1,   "too short!")
  def write(t: String): ValidatedName[Unit] = Right(println(t))
}

implicit val validated: Execution[ValidatedName] = new Execution[ValidatedName] {
  def doAndThen[A, B](c: ValidatedName[A])(f: A => ValidatedName[B]): ValidatedName[B] = c flatMap f
  def create[B](b: B): ValidatedName[B] = Right(b)
}

implicit val validated: Terminal[ValidatedName] = new TerminalValidated

def greet[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  for {
        _ <- t.write(s"Hello, what is your name?")
     name <- t.read
        _ <- t.write(s"Nice to meet you, $name.")
  } yield name

println("About to run greet[ValidatedName]")
val validatedName: ValidatedName[String] = greet[ValidatedName]
validatedName match {
  case Right(name) => println(s"The result was $name.")
  case Left(error) => println(s"Invalid Name: $error.")
}
```

```
About to run greet[ValidatedName]
Hello, what is your name?
John
Nice to meet you, John.
The result was John.
```

```
About to run greet[ValidatedName]
Hello, what is your name?
john
Invalid Name: not capitalised!
```

Just for fun, running the **greet** method using **List**

@philip_schwarz

```
class TerminalMany extends Terminal[List] {
  def read: List[String] = StdIn.readLine.split(",").toList
  def write(t: String): List[Unit] = List(println(t))
}

implicit val manyExecution: Execution[List] = new Execution[List] {
  def doAndThen[A, B](c: List[A])(f: A => List[B]): List[B] = c flatMap f
  def create[B](b: B): List[B] = List(b)
}

implicit val manyTerminal: Terminal[List] = new TerminalMany

def greet[C[_]](implicit t: Terminal[C], e: Execution[C]): C[String] =
  for {
        _ <- t.write(s"Hello, what is your name?")
     name <- t.read
        _ <- t.write(s"Nice to meet you, $name.")
  } yield name

println("About to run greet[List]")
val names: List[String] = greet[List]
println(s"The result was $names.")
```

```
About to run greet[List]
Hello, what is your name?
John,Jane
Nice to meet you, John.
Nice to meet you, Jane.
The result was List(John, Jane).
```