Effect Oriented Programming

A New Paradigm for Creating Predictable Systems

Bill Frasure, Bruce Eckel, and James Ward

Effect Oriented Programming

A New Paradigm for Creating Predictable Systems

Bill Frasure, Bruce Eckel, and James Ward

This book is for sale at http://leanpub.com/effect-oriented-programming

This version was published on 2024-05-23

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2021 - 2024 Bill Frasure, Bruce Eckel, and James Ward

Contents

Copyright

Edit This Chapter[1]

Effect Oriented Programming

By Bill Frasure, Bruce Eckel and James Ward

eBook ISBN 978-0-9818725-6-8

Print Book ISBN 978-0-9818725-7-5

The eBook ISBN covers the Leanpub eBook distribution in all formats, available through *www.EffectOrientedProgramming.com*.

Please purchase this book through www.EffectOrientedProgramming.com, to support its continued maintenance and updates.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, see EffectOrientedProgramming.com.

Created in Crested Butte, Colorado, USA.

Text printed in the United States

Ebook: Version 1.0, Month Year

First printing Month Year

[1] https://github.com/EffectOrientedProgramming/book/edit/main/Chapters/00_Copyright.md

Cover design by Daniel Will-Harris, www.Will-Harris.com[2]

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations are printed with initial capital letters or in all capitals.

The Scala trademark belongs to {{???}}. Java is a trademark or registered trademark of Oracle, Inc. in the United States and other countries. Windows is a registered trademark of Microsoft Corporation in the United States and other countries. All other product names and company names mentioned herein are the property of their respective owners.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Visit us at www.EffectOrientedProgramming.com.

Source Code

All the source code for this book is available as copyrighted freeware, distributed via GitHub[3]. To ensure you have the most current version, this is the official code distribution site. You may use this code in classroom and other educational situations as long as you cite this book as the source.

The primary goal of this copyright is to ensure that the source of the code is properly cited, and to prevent you from republishing the code without permission. (As long as this book is cited, using examples from the book in most media is generally not a problem.)

In each source-code file you find a reference to the following copyright notice:

[2] http://www.Will-Harris.com
[3] https://github.com/EffectOrientedProgramming/examples

// Copyright.txt
This computer source code is Copyright ©2021 MindView LLC.
All Rights Reserved.

Permission to use, copy, modify, and distribute this
computer source code (Source Code) and its documentation
without fee and without a written agreement for the
purposes set forth below is hereby granted, provided that
the above copyright notice, this paragraph and the
following five numbered paragraphs appear in all copies.

1. Permission is granted to compile the Source Code and to
include the compiled code, in executable format only, in
personal and commercial software programs.

2. Permission is granted to use the Source Code without
modification in classroom situations, including in
presentation materials, provided that the book "Effect Oriented
Programming" is cited as the origin.

3. Permission to incorporate the Source Code into printed
media may be obtained by contacting:

MindView LLC, PO Box 969, Crested Butte, CO 81224
MindViewInc@gmail.com

4. The Source Code and documentation are copyrighted by
MindView LLC. The Source code is provided without express
or implied warranty of any kind, including any implied
warranty of merchantability, fitness for a particular
purpose or non-infringement. MindView LLC does not
warrant that the operation of any program that includes the
Source Code will be uninterrupted or error-free. MindView
LLC makes no representation about the suitability of the
Source Code or of any software that includes the Source
Code for any purpose. The entire risk as to the quality
and performance of any program that includes the Source
Code is with the user of the Source Code. The user
understands that the Source Code was developed for research
and instructional purposes and is advised not to rely
exclusively for any reason on the Source Code or any
program that includes the Source Code. Should the Source

Code or any resulting software prove defective, the user assumes the cost of all necessary servicing, repair, or correction.

5. IN NO EVENT SHALL MINDVIEW LLC, OR ITS PUBLISHER BE LIABLE TO ANY PARTY UNDER ANY LEGAL THEORY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS, OR FOR PERSONAL INJURIES, ARISING OUT OF THE USE OF THIS SOURCE CODE AND ITS DOCUMENTATION, OR ARISING OUT OF THE INABILITY TO USE ANY RESULTING PROGRAM, EVEN IF MINDVIEW LLC, OR ITS PUBLISHER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. MINDVIEW LLC SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOURCE CODE AND DOCUMENTATION PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, WITHOUT ANY ACCOMPANYING SERVICES FROM MINDVIEW LLC, AND MINDVIEW LLC HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Please note that MindView LLC maintains a Web site which is the sole distribution point for electronic copies of the Source Code, where it is freely available under the terms stated above:

  https://github.com/EffectOrientedProgramming/examples

If you think you've found an error in the Source Code, please submit a correction at:
https://github.com/EffectOrientedProgramming/examples/issues

Preface

Edit This Chapter[4]

Effects are the unpredictable parts of systems. Traditional programs do not isolate these unpredictable parts, making it hard to manage them. *Effect Systems* partition the unpredictable parts and manage them separately from the predictable ones.

With Effect Systems, developers can more easily build systems that are reliable, resilient, and testable.

Effect Oriented Programming is a new paradigm for programming with Effect Systems.

Many bleeding-edge languages now have ways to manage Effects (e.g. OCaml, Unison, and Roc). But not every programming language has an Effect System. Some languages have built-in support for managing Effects while others have support through libraries.

Who is the book is for

This book focuses on the concepts of Effect Systems, rather than language and library specifics. Since Effect Systems are a new and emerging paradigm, we have limited choices. We use Scala 3 and an Effect System library called ZIO to convey the concepts of Effect Oriented Programming.

If you use Scala, you can use an Effect System, of which there are a number of options (ZIO, Cats Effects, and Kyo).

If you are not using Scala, the concepts of Effect Systems may only be useful when your language or a library supports them. However, learning the concepts now will help you prepare for that eventuality.

While Scala knowledge is not required to learn the concepts, this book assumes you are familiar with:

[4] https://github.com/EffectOrientedProgramming/book/edit/main/Chapters/01_Preface.md

- Functions
- Strong static typing
- Chaining operations on objects (`"asdf ".trim.length`)

Code examples

The code examples are available at: github.com/EffectOrientedProgramming/examples[5]

The code in this book uses a Scala 3 language syntax that might be unfamiliar, even to Scala developers. Since our focus is on the concepts of Effect Oriented Programming we've tried to make the code examples very readable, even on mobile devices. To accomplish this, when functions have single parameters we generally use Scala 3's Significant Indentation style. For example: `scala Console.printLine: "hello, world"` The parameter to the `ZIO.debug` function is specified on a new line instead of the usual parens (`ZIO.debug("hello, world")`) The colon (`:`) indicates that the function parameter will use the significant indentation syntax. For multi-parameter functions and in cases where the single parameter is very short and does not contain nested function calls, we use the traditional syntax: `scala Console.printLine(1)`

Acknowledgements

Kit Langton, for being a kindred spirit in software interests and inspiring contributor to the open source world.

Wyett Considine, for being an enthusiastic intern and initial audience.

Hali Frasure, for cooking so many dinners and facilitating our book nights generally.

[5] https://github.com/EffectOrientedProgramming/examples

Introduction to Effects

Edit This Chapter[6]

Systems built from unpredictable parts can have predictable behavior.

If you've been programming for a while, this sounds far-fetched or even impossible.

Most existing languages are built for rapid development. You create a system as quickly as possible, then begin isolating areas of failure, finding and fixing bugs until the system is tolerable and can be delivered. Throughout the lifetime of the system, bugs are regularly discovered and fixed. There is no realistic expectation that you will ever achieve a completely bug-free system, just one that seems to work well enough to meet the requirements. This is the reality programmers have come to accept.

If each piece of a traditional system is unpredictable, when you combine these pieces you get a multiplicative effect – the resulting parts are significantly less predictable than their component pieces.

What if we could change our thinking around the problem of building software systems? Imagine building small pieces that can each be reasoned about and made predictable. Now suppose there is a way to combine these predictable pieces to make larger parts that are just as predictable. Each time you combine smaller parts to create a larger part, the result inherits the predictability of its components. Instead of multiplying unpredictability, you maintain predictability. The resulting system is as predictable as any of its components.

This is what *Functional Programming* together with *Effect Systems* can achieve. This is what we want to teach you in this book.

The biggest impact on you as a programmer is the requirement for patience. With most languages, the first thing you want to do is figure out how to write "Hello, World!", then start accumulating the other language features as standalone concepts. In Functional Programming we start by examining the impact of each concept on

[6] https://github.com/EffectOrientedProgramming/book/edit/main/Chapters/02_Introduction_to_Effects.md

predictability. We then combine the smaller concepts, ensuring predictability at each step.

A predictable system isolates parts that are always the same (called pure functions) from the parts that are unpredictable (effects).

Dealing With Unpredictability

Any real program has to interact with things outside the programmer's control. All external systems are unpredictable.

Building systems that are predictable requires isolating and managing the unpredictable parts. An approach that programmers may use to handle this is to isolate the parts of the program which use external systems. By isolating them, programmers then have tools to handle the unpredictable parts in more predictable ways.

The interactions with external systems can be defined in terms of "Effects". Effects create a delineation between the parts of a program that interact with external systems and those that don't.

For example, a program that displays the current date requires something that actually knows the current date. The program needs to talk to an external system (maybe just the operating system) to get this information. These programs are unpredictable because the programmer has no control over what that external system will say or do.

What is an Effect?

An *Effect* is an interaction, often with an outside system, that is inherently unpredictable.

Anytime you run an Effect, you have changed the world and cannot go back.

If you 3D-print a figurine, you cannot reclaim that material. Once you send a Tweet, you can delete it but people might have already read it. Even if you provide database `DELETE` statements paired with `INSERT` statements, it must still be considered effectful. Another program might read your data before you delete it, or a database trigger might activate during an `INSERT`.

Once our program runs an Effect, the impact is out of our control.

We must assume that running an Effect modifies an external system.

As an example in real-life, just saying the words "You are getting a raise" creates an "effect" that may not be reversible.

Effects that only read data from external systems are also unpredictable. For example, getting a random number from the system is usually unpredictable.

In systems there are many types of Effects, like:

- Accepting user input
- Reading from a file
- Getting the current time from the system clock
- Generating a random number
- Displaying on the screen
- Writing to a file
- Mutating a variable
- Saving to a database

These can also have domain specific forms, like:

- Getting the current price of a stock
- Detecting the current from a pacemaker
- Checking the temperature of a nuclear reactor
- 3D printing a model
- Triggering an alarm
- Sensing slippage in an anti-lock braking system
- Stabilizing an airplane
- Detonating explosives

All of these are unpredictable.

Effect Systems Manage Unpredictability

Given that Effects are unpredictable, we can utilize operations from an Effect System to manage the unpredictability. Effect Systems are designed to make these operations easy. For example, any Effect can use a `timeout` to control the Effect's maximum duration.

Applying these operations starts to feel like a superpower.

Superpowers with Effects

Edit This Chapter[7]

{{ TODO: More consistent error messages }}

Once programs are defined in terms of Effects, we use operations from the Effect System to manage different aspects of unpredictability. Combining Effects with these operations feels like a superpower. The reason we call them "superpowers" is that the operations can be attached to **any** Effect. Operations can even be chained together.

Common operations like `timeout` are applicable to all Effects while some operations like `retry` are only applicable to a subset of Effects.

Ultimately this means we do not need to create bespoke operations for the many different Effects our system may have.

To illustrate this we will show a few examples of common operations applied to Effects. Let's start with the "happy path" where we save a user to a database (an Effect) and then gradually add superpowers.

To start with we save a user to a database:

```
val userName =
  "Morty"


val effect0 =
  saveUser:
    userName
```

Effects can be run as "main" programs, embedded in other programs, or in tests. To run an Effect with ZIO as a "main" program, we normally do this:

[7] https://github.com/EffectOrientedProgramming/book/edit/main/Chapters/03_Superpowers.md

```
object MyApp extends ZIOAppDefault:
  def run =
    effect0
```

In this book, to avoid the excess lines, we shorten this to: `scala def run = effect0 // Result: User saved`

By default, the `saveUser` Effect runs in the "happy path" so it will not fail.

We can explicitly specify the way in which this Effect will run by overriding the `bootstrap` value: "'scala override val bootstrap = happyPath

def run = effect0 // Result: User saved "'

Overriding the `bootstrap` value simulates failures in the following examples.

In real systems, assuming the "happy path" causes strange unhandled errors.

We can also run a scenario that causes failure: "'scala override val bootstrap = neverWorks

def run = effect0 // Log: **Database crashed!!** // Result: **Database crashed!!** "'

The program logs and returns the failure.

Superpower: Persevering Through Failure

Sometimes things work when you keep trying.
We can retry `effect0` with the `retryN` operation:

```
val effect1 =
  effect0.retryN(2)
```

The Effect with the retry behavior becomes a new Effect and can optionally be assigned to a `val` (as is done here).

Now we run the new Effect in a scenario that works on the third try: "'scala override val bootstrap = doesNotWorkInitially

def run = effect1 // Log: **Database crashed!!** // Log: **Database crashed!!** // Result: User saved "'

The output shows that running the Effect worked after two retries.

What If It Never Succeeds?

In the `neverWorks` scenario, the Effect fails its initial attempt and subsequent retries:

```
override val bootstrap =
  neverWorks

def run =
  effect1
// Log: **Database crashed!!**
// Log: **Database crashed!!**
// Log: **Database crashed!!**
// Result: **Database crashed!!**
```

After the failed retries, the program returns the error.

Superpower: Nice Error Messages

Let's define a new Effect that chains a nicer error onto the previously defined operations (the retries) using `orElseFail` which transforms any failure into a user-friendly error:

```
val effect2 =
  effect1.orElseFail:
    "ERROR: User could not be saved"
```

We altered the behavior without restructuring the original Effect. Running this new Effect in the `neverWorks` scenario will produce the error:

```
override val bootstrap =
  neverWorks

def run =
  effect2
// Log: **Database crashed!!**
// Log: **Database crashed!!**
// Log: **Database crashed!!**
// Result: ERROR: User could not be saved
```

The `orElseFail` is combined with the prior Effect that has the retry, creating another new Effect that has both error handling operations.

Superpower: Imposing Time Limits

Sometimes an Effect fails quickly, as we saw with retries. Sometimes an Effect taking too long is itself a failure. The `timeoutFail` operation can be chained to our previous Effect to specify a maximum time the Effect can run for, before producing an error:

```
val effect3 =
  effect2.timeoutFail("*** Save timed out ***"):
    5.seconds
```

If the effect does not complete within the time limit, it is canceled and returns our error message. Timeouts can be added to any Effect.

Running the new Effect in the `slow` scenario causes it to take longer than the time limit:

```
override val bootstrap =
  slow

def run =
  effect3
// Log: Interrupting slow request
// Result: *** Save timed out ***
```

The Effect took too long and produced the error.

Superpower: Fallback From Failure

In some cases there may be fallback behavior for failed Effects. One option is to use the `orElse` operation with a fallback Effect: `scala val effect4 = effect3.orElse: sendToManualQueue: userName`

`sendToManualQueue` represents something we can do when the user can't be saved.

Let's run the new Effect in the `neverWorks` scenario to ensure we reach the fallback:

```
override val bootstrap =
  neverWorks

def run =
  effect4
// Log: **Database crashed!!**
// Log: **Database crashed!!**
// Log: **Database crashed!!**
// Result: Please manually provision Morty
```

{{ todo: we are not seeing the expected number failures due to OurClock and timeoutFail }}

The retries do not succeed so the user is sent to the fallback Effect.

Superpower: Add Some Logging

Effects can be run concurrently and as an example, we can at the same time as the user is being saved, send an event to another system.

```
val effect5 =
  effect4.fireAndForget:
    logUserSignup
```

`fireAndForget` is a convenience method we defined (in hidden code) that makes it easy to run two effects in parallel and ignore any failures on the `logUserSignup` Effect.

```
override val bootstrap =
  happyPath

def run =
  effect5
// Log: Signup initiated for Morty
// Result: Please manually provision Morty
```

We run the effect again in the `HappyPath` scenario to demonstrate running the Effects in parallel.

We can add all sorts of custom behavior to our Effect type, and then invoke them regardless of error and result types.

Superpower: How Long Do Things Take?

For diagnostic information you can track timing:

```
val effect6 =
  effect5.timed


override val bootstrap =
  happyPath

def run =
  effect6
// Result: (PT5.032807209S,User saved)
```

We run the Effect in the "HappyPath" Scenario; now the timing information is packaged with the original output `String`.

Superpower: Maybe We Don't Want To Run Anything

Now that we have added all of these superpowers to our process, our lead engineer lets us known that a certain user should be prevented from using our system.

```
val effect7 =
  effect6.when(userName != "Morty")

override val bootstrap =
  happyPath

def run =
  effect7
// Result: None
```

We can add behavior to the end of our complex Effect, that prevents it from ever executing in the first place.

Many More Superpowers

{{ todo: make rendering in manuscript work }}

```
graph TD
  effect0 --retry--> effect1 --"orElseFail"--> effect2 --timeoutFail-->\
 effect3 --"orElse"--> effect4 --fireAndForget--> effect5 --timed--> ef\
fect6 --when--> effect7
```

These examples have shown only a glimpse into the superpowers we can add to **any** Effect. There are even more we will explore in the following chapters.

Deferred Execution Enables many of the Superpowers

If these effects were all executed immediately, we would not be able to freely tack on new behaviors. We cannot timeout something that might have already started running, or even worse - completed, before we get our hands on it. We cannot retry something if we are only holding on to the completed result. We cannot parallelize operations if they have already started single-threaded execution.

We need to be holding on to a value that represents something that *can* be run, but hasn't yet. If we have that, then our Effect System can freely add behavior before/after that value.

defer/run example

When we make a defer block, nothing inside of it will be executed yet. `scala val program = defer: Console.printLine("Hello").run val subject = "world" Console.printLine(subject).run`

The `.run` method is only available on our Effect values. We explicitly call `.run` whenever we want to sequence our effects. If we do not call `.run`, then we are just going to have an un-executed effect. We want this explicit control, so that we can manipulate our effects up until it is time to run them.

For example, we can repeat our un-executed effect:

```
val programManipulatingBeforeRun =
  defer:
    Console.printLine("Hello").repeatN(3).run
```

We *cannot* repeat our executed effect.

```
val programManipulatingBeforeRun =
  defer:
    Console.printLine("Hello").run.repeatN(3)
// error:
// value repeatN is not a member of Unit
// class Chapter198 extends mdoctools.ToRun:
//                                  ^
```

Note that these calls to `.run` are all within a `defer` block, so when `program` is defined, we still have not actually executed anything. We have described a program that knows the order in which to execute our individual effects *when the program is executed.*

```
val surroundedProgram =
  defer:
    Console.printLine("**Before**").run
    program.run
    Console.printLine("**After**").run
```

Even now, we have not executed anything. It is only when we pass our completed program over to the effect system that the program is executed.

```
def run = surroundedProgram
// Log: Signup initiated for Morty
// **Before**
// Hello
// world
// **After**
// Result: ()

// TODO Decide where to put this
val program =
  defer:
    println("hi").run
    (1 + 1).run
// error:
// value run is not a member of Unit.
// An extension method was tried, but could not be fully constructed:
//
//     run[R, E, A](println("hi"))
//
//     failed with:
//
//         Found:    Unit
//         Required: ZIO[Nothing, Any, Any]
//     println("hi").run
//     ^^^^^^^^^^^^^^^^
// error:
// value run is not a member of Int.
// An extension method was tried, but could not be fully constructed:
//
//     run[R, E, A](1.+(1))
//
//     failed with:
//
//         Found:    (2 : Int)
//         Required: ZIO[Nothing, Any, Any]
//     (1 + 1).run
//     ^^^^^^^^^^^
```

Explain the 2 versions of run and how they came to be

Interpreter

Configuration

Edit This Chapter[8]

1. Application startup uses the same tools that you utilize for the rest of your application

General/Historic discussion

One reason to modularize an application into "parts" is that the relationship between the parts can be expressed and also changed depending on the needs for a given execution path.
Typically, this approach to breaking things into parts and expressing what they need, is called "Dependency Injection."

… Why is it called "Dependency Injection" ? Avoid having to explicitly pass things down the call chain.

There is one way to express dependencies.

Let's consider an example: We want to write a function that fetches Accounts from a database The necessary parts might be a `DatabaseService` which provides database connections and a `UserService` which provides the access controls. By separating these dependencies our from the functionality of fetching accounts, tests can "fake" or "mock" the dependencies to simulate the actual dependency.

In the world of Java these dependent parts are usually expressed through annotations (e.g. `@Autowired` in Spring). But these approaches are "impure" (require mutability), often rely on runtime magic (e.g. reflection), and require everything that uses the annotations to be created through a Dependency Injection manager, complicating construction flow.
An alternative to this approach is to use "Constructor Injection" which avoids some of the pitfalls associated with "Field Injection" but doesn't resolve some of the

[8] https://github.com/EffectOrientedProgramming/book/edit/main/Chapters/04_Configuration.md

underlying issues, including the ability for dependencies to be expressed at compile time.

If instead functionality expressed its dependencies through the type system, the compiler could verify that the needed parts are in-fact available given a particular path of execution (e.g. main app, test suite one, test suite two).

What ZIO Provides Us.

With ZIO's approach to dependencies, you get many desirable characteristics at compile-time, using standard language features. Your services are defined as classes with constructor arguments, just as in any vanilla Scala application. No annotations that kick off impenetrable wiring logic outside your normal code.

For any given service in your application, you define what it needs in order to execute. Finally, when it is time to build your application, all of these pieces can be provided in one, flat space. Each component will automatically find its dependencies, and make itself available to other components that need it.

To aid further in understanding your application architecture, you can visualize the dependency graph with a single line.

You can also do things that simply are not possible in other approaches, such as sharing a single instance of a dependency across multiple test classes, or even multiple applications.

DI-Wow!

TODO Values to convey:

- Layer Graph
  - Cycles are a compile error
  - Visualization with Mermaid
  - test implementations
- Layer Resourcefulness
  - Layers can have setup & teardown (open & close)

```
// Explain private constructor approach
case class Dough():
  val letRise =
    ZIO.debug:
      "Dough is rising"

object Dough:
  val fresh =
    ZLayer.derive[Dough]
```

Step 1: Provide Dependency Layers to Effects

We must provide all required dependencies to an effect before you can run it.

```
def run =
  ZIO
    .serviceWithZIO[Dough]:
      dough => dough.letRise
    .provide:
      Dough.fresh
// Dough is rising
// Result: ()
```

Step 2: Unresolved Dependencies Are Compile Errors

If the dependency for an Effect isn't provided, we get a compile error:

TODO: Decide what to do about the compiler error differences between these approaches

TODO: Can we avoid the `.provide()` and still get a good compile error in mdoc

TODO: Strip `repl.MdocSession.MdocApp.` from output. Remove caret indicator from output.

```
ZIO
  .serviceWithZIO[Dough]:
    dough => dough.letRise
  .provide()
// error:
//
//
// ——— ZLAYER ERROR ——————————————————————————————————————————————\
—
//
//  Please provide a layer for the following type:
//
//    1. repl.MdocSession.MdocApp.Dough
//
// ——————————————————————————————————————————————————————————————\
—
//
//
//     ZIO
//     ^
```

Step 3: Dependencies can "automatically" assemble to fulfill the needs of an effect

The requirements for each ZIO operation are tracked and combined automatically.

```
case class Heat()

val oven =
  ZLayer.derive[Heat]
```

```
trait Bread

case class BreadHomeMade(
    heat: Heat,
    dough: Dough
) extends Bread

object Bread:
  val homemade =
    ZLayer.derive[BreadHomeMade]
```

Something around how like typical DI, the "graph" of dependencies gets resolved "for you" This typically happens in some completely new/custom phase, that does follow standard code paths. Dependencies on effects propagate to effects which use effects.

```
def run =
  ZIO
    .service[Bread]
    .provide(Bread.homemade, Dough.fresh, oven)
// Result: BreadHomeMade(Heat(),Dough())
```

Step 4: Different effects can require the same dependency

Eventually, we grow tired of eating plain `Bread` and decide to start making `Toast`. Both of these processes require `Heat`.

```
case class Toast(heat: Heat, bread: Bread)

object Toast:
  val make =
    ZLayer.derive[Toast]
```

It is possible to also use the oven to provide `Heat` to make the `Toast`.

The dependencies are tracked by their type. In this case both `Toast.make` and `Bread.homemade` require `Heat`.

Notice - Even though we provide the same dependencies in this example, oven is *also* required by `Toast.make`

```
def run =
  ZIO
    .service[Toast]
    .provide(
      Toast.make,
      Bread.homemade,
      Dough.fresh,
      oven
    )
// Result: Toast(Heat(),BreadHomeMade(Heat(),Dough()))
```

However, the oven uses a lot of energy to make `Toast`. It would be great if we can instead use our dedicated toaster!

```
val toaster =
  ZLayer.derive[Heat]

def run =
  ZIO
    .service[Heat]
    .provide:
      toaster
// Result: Heat()
```

Step 5: Dependencies must be fulfilled by unique types

```
ZIO
  .service[Toast]
  .provide(
    Toast.make,
    Dough.fresh,
    Bread.homemade,
    oven,
    toaster
  )
// error:
//
//
```

```
// ——— ZLAYER ERROR ———————————————————————————————————————————\
—
//
//  Ambiguous layers! I cannot decide which to use.
//  You have provided more than one layer for the following type:
//
//    repl.MdocSession.MdocApp.Heat is provided by:
//       1. oven
//       2. toaster
//
// ———————————————————————————————————————————————————————————————\
—
//
//
```

Unfortunately our program is now ambiguous. It cannot decide if we should be making `Toast` in the oven, `Bread` in the toaster, or any other combination.

Step 6: Providing Dependencies at Different Levels

This enables other effects that use them to provide their own dependencies of the same type

```
def run =
  ZIO
    .serviceWithZIO[Bread]:
      bread =>
        ZIO
          .service[Toast]
          .provide(
            Toast.make,
            toaster,
            ZLayer.succeed:
              bread
          )
    .provide(Bread.homemade, Dough.fresh, oven)
// Result: Toast(Heat(),BreadHomeMade(Heat(),Dough()))
```

Step 7: Effects can Construct Dependencies

```
case class BreadStoreBought() extends Bread

val buyBread =
  ZIO.succeed:
    BreadStoreBought()


val storeBought =
  ZLayer.fromZIO:
    buyBread


def run =
  ZIO
    .service[Bread]
    .provide:
      storeBought
// Result: BreadStoreBought()
```

Step 8: Dependencies can fail

Since dependencies can be built with effects, this means that they can fail.

```
def run =
  ZIO
    .service[Bread]
    .provide:
      Friend.bread(worksOnAttempt =
        3
      )
// Attempt 1: Error(Friend Unreachable)
// Result: Error(Friend Unreachable)
```

Step 9: Fallback Dependencies

```
def run =
  ZIO
    .service[Bread]
    .provide:
      Friend
        .bread(worksOnAttempt =
          3
        )
        .orElse:
          storeBought
// Attempt 1: Error(Friend Unreachable)
// Result: BreadStoreBought()
```

Step 10: Dependency Retries

```
def run =
  ZIO
    .service[Bread]
    .provide:
      Friend
        .bread(worksOnAttempt =
          3
        )
        .retry:
          Schedule.recurs:
            1
// Attempt 1: Error(Friend Unreachable)
// Attempt 2: Error(Friend Unreachable)
// Result: Error(Friend Unreachable)
```

```
def run =
  ZIO
    .service[Bread]
    .provide:
      Friend
        .bread(worksOnAttempt =
          3
        )
        .retry:
          Schedule.recurs:
            2
// Attempt 1: Error(Friend Unreachable)
// Attempt 2: Error(Friend Unreachable)
// Attempt 3: Succeeded
// Result: BreadFromFriend()
```

Step 11: Layer Retry + Fallback?

Maybe retry on the ZLayer eg. (BreadDough.rancid, Heat.brokenFor10Seconds)

```
def run =
  ZIO
    .service[Bread]
    .provide:
      Friend
        .bread(worksOnAttempt =
          3
        )
        .retry:
          Schedule.recurs:
            1
        .orElse:
          storeBought
// Attempt 1: Error(Friend Unreachable)
// Attempt 2: Error(Friend Unreachable)
// Result: BreadStoreBought()
```

Step 12: Externalize Config for Retries

Changing things based on the running environment.

- CLI Params
- Config Files
- Environment Variables

```
import zio.config.*
import zio.config.magnolia.deriveConfig
import zio.config.typesafe.*

case class RetryConfig(times: Int)

val configDescriptor: Config[RetryConfig] =
  deriveConfig[RetryConfig]

val configProvider =
  ConfigProvider.fromHoconString:
    "{ times: 2 }"

val config =
  ZLayer.fromZIO:
    read:
      configDescriptor.from:
        configProvider

def run =
  ZIO
    .serviceWithZIO[RetryConfig]:
      retryConfig =>
        ZIO
          .service[Bread]
          .provide:
            Friend
              .bread(worksOnAttempt =
                3
              )
              .retry:
                Schedule.recurs:
                  retryConfig.times
    .provide:
      config
// Attempt 1: Error(Friend Unreachable)
// Attempt 2: Error(Friend Unreachable)
// Attempt 3: Succeeded
// Result: BreadFromFriend()
```

Testing Effects

TODO: Bridge from dependency / configuration to here

TODO: Code that provides an "ideal friend" to our bread example Maybe as a 2 step process, first outside of a test, then in a test. Or a single step just in a test

Effects need access to external systems thus are unpredictable.
Tests are ideally predictable so how do we write tests for effects that are predictable?
With ZIO we can replace the external systems with predictable ones when running our tests.

With ZIO Test we can use predictable replacements for the standard systems effects (Clock, Random, Console, etc).

Random

An example of this is Random numbers. Randomness is inherently unpredictable. But in ZIO Test, without changing our Effects we can change the underlying systems with something predictable:

```
val coinToss =
  // TODO: This is the first place we use defer.
  // We need to deliberately, and explicitly,
  // introduce it.
  defer:
    if Random.nextBoolean.run then
      ZIO.debug("Heads").run
      ZIO
        .succeed:
          "Heads"
        .run
    else
      ZIO.debug("Tails").run
      ZIO
        .fail:
          "Tails"
        .run
```

```
val flipTen =
  defer:
    val numHeads =
      ZIO
        .collectAllSuccesses:
          List.fill(10):
            coinToss
        .run
        .size
    ZIO.debug(s"Num Heads = $numHeads").run
    numHeads

def run =
  flipTen
// Heads
// Heads
// Tails
// Tails
// Heads
// Heads
// Tails
// Heads
// Heads
// Tails
// Num Heads = 6
// Result: 6

def spec =
  test("flips 10 times"):
    defer:
      TestRandom
        .feedBooleans(true)
        .repeatN(9)
        .run
      assertTrue:
        flipTen.run == 10
// Heads
// Heads
// Heads
// Heads
// Heads
// Heads
```

```
// Heads
// Heads
// Heads
// Heads
// Num Heads = 10
// + flips 10 times
// Result: Summary(1,0,0,,PT0.066529S)


val rosencrantzCoinToss =
  coinToss.debug:
    "R"

val rosencrantzAndGuildensternAreDead =
  defer:
    ZIO
      .debug:
        "*Performance Begins*"
      .run
    rosencrantzCoinToss.repeatN(4).run

    ZIO
      .debug:
        "G: There is an art to building suspense."
      .run
    rosencrantzCoinToss.run

    ZIO
      .debug:
        "G: Though it can be done by luck alone."
      .run
    rosencrantzCoinToss.run

    ZIO
      .debug:
        "G: ...probability"
      .run
    rosencrantzCoinToss.run
```

```
def spec =
  test(
    "rosencrantzAndGuildensternAreDead finishes"
  ):
    defer:
      TestRandom
        .feedBooleans:
          true
        .repeatN:
          7
        .run
      rosencrantzAndGuildensternAreDead.run
      assertCompletes
// *Performance Begins*
// Heads
// R: Heads
// Heads
// R: Heads
// Heads
// ...
// R: Heads
// G: ...probability
// Heads
// R: Heads
// + rosencrantzAndGuildensternAreDead finishes
// Result: Summary(1,0,0,,PT0.039782S)

def spec =
  test("flaky plan"):
    defer:
      rosencrantzAndGuildensternAreDead.run
      assertCompletes
  @@ TestAspect.withLiveRandom @@
    TestAspect.flaky(Int.MaxValue)
// *Performance Begins*
// Tails
// <FAIL> R: Fail(Tails,Stack trace for thread "zio-fiber-1320237666":
//         at repl.MdocSession.MdocApp.coinToss(<input>:403)
//         at repl.MdocSession.MdocApp.rosencrantzCoinToss(<input>:470)
//         at repl.MdocSession.MdocApp.rosencrantzAndGuildensternAreDead(<inpu\
t>:475)
// ...
```

```
// R: Heads
// G: ...probability
// Heads
// R: Heads
// + flaky plan
// Result: Summary(1,0,0,,PT0.018375S)
```

The `Random` Effect uses an injected something which when running the ZIO uses the system's unpredictable random number generator. In ZIO Test the `Random` Effect uses a different something which can predictably generate "random" numbers. `TestRandom` provides a way to define what those numbers are. This example feeds in the `Ints` `1` and `2` so the first time we ask for a random number we get `1` and the second time we get `2`.

Anything an effect needs (from the system or the environment) can be substituted in tests for something predictable. For example, an effect that fetches users from a database can be simulated with a predictable set of users instead of having to setup a test database with predictable users.

When your program treats randomness as an effect, testing unusual scenarios becomes straightforward. You can preload "Random" data that will result in deterministic behavior. ZIO gives you built-in methods to support this.

Time

Even time can be simulated as using the clock is an effect.

```
val nightlyBatch =
  ZIO
    .sleep:
      24.hours
    .debug:
      "Parsing CSV"
```

```
def spec =
  test("batch runs after 24 hours"):
    val timeTravel =
      TestClock.adjust:
        24.hours

    defer:
      val fork =
        nightlyBatch.fork.run
      timeTravel.run
      fork.join.run

      assertCompletes
// Parsing CSV: ()
// + batch runs after 24 hours
// Result: Summary(1,0,0,,PT0.025136S)
```

The `race` is between `nightlyBatch` and `timeTravel`. It completes when the first Effect succeeds and cancels the losing Effect, using the Effect System's cancellation mechanism.

By default in ZIO Test, the clock does not change unless instructed to. Calling a time based effect like `timeout` would hang indefinitely with a warning like: "'

Warning: A test is using time, but is not advancing the test clock, which may result in the test hanging. Use TestClock.adjust to manually advance the time. "'

To test time based effects we need to `fork` those effects so that then we can adjust the clock. After adjusting the clock, we can then `join` the effect where in this case the timeout has then been reached causing the effect to return a `None`.

Using a simulated Clock means that we no longer rely on real-world time for time. So this example runs in milliseconds of real-world time instead of taking an actual 1 second to hit the timeout. This way our time-based tests run much more quickly since they are not based on actual system time. They are also more predictable as the time adjustments are fully controlled by the tests.

Targeting Error-Prone Time Bands

Using real-world time also can be error prone because effects may have unexpected results in certain time bands. For instance, if you have code that gets the time and it

happens to be 23:59:59, then after some operations that take a few seconds, you get some database records for the current day, those records may no longer be the day associated with previously received records. This scenario can be very hard to test for when using real-world time. When using a simulated clock in tests, you can write tests that adjust the clock to reliably reproduce the condition.

Errors

Edit This Chapter[9]

1. Why errors as values
2. Creating & Handling
    1. Flexible error types

Our program for this chapter

We want to show the user a page that shows the current temperature at their location
It will look like this

```
Temperature: 30 degrees
```

There are 2 error situations we need to handle:

- Network call to weather service fails.
- A fault in our GPS hardware

We want our program to always result in a sensible message to the user.

Throwing Exceptions

Throwing exceptions is one way indicate failure.

In a language that cannot `throw`, following the execution path is simple, following 2 basic rules:

- At a branch, execute first match
- Otherwise, Read everything:

[9] https://github.com/EffectOrientedProgramming/book/edit/main/Chapters/05_Errors.md

  - left-to-right
  - top-to-bottom,

Once you add `throw`, the world gets more complicated.

- Unless we `throw`, jumping through a different dimension

TODO Prose transition here

We have an existing `getTemperatureOrThrow` function that can fail in unspecified ways.

```
def render(value: String) =
  s"Temperature: $value"

def temperatureApp(): String =
  render:
    getTemperatureOrThrow()

def run =
  ZIO.attempt:
    temperatureApp()
// Result: Temperature: 35 degrees
```

On the happy path, everything looks as desired. If the network is unavailable, what is the behavior for the caller? If we don't make any attempt to handle our problem, the whole program blows up and shows the gory details to the user.

```
scenario =
  ErrorsScenario.NetworkError

def run =
  ZIO.succeed:
    temperatureApp()
// Result: Defect: NetworkException
```

Manual Error Discovery

Exceptions do not convey a contract in either direction.

If you have been burned in the past by functions that throw surprise exceptions , you might defensively catch `Exceptions` all over your program. For this program, it could look like:

```
def temperatureCatchingApp(): String =
  try
    render:
      getTemperatureOrThrow()
  catch
    case ex: Exception =>
      "Failure"


scenario =
  ErrorsScenario.NetworkError

def run =
  ZIO.succeed:
    temperatureCatchingApp()
// Result: Failure
```

We have improved the failure behavior significantly; is it sufficient for all cases? Imagine our network connection is stable, but we have a problem in our GPS hardware. In this situation, do we show the same message to the user? Ideally, we would show the user a distinct message for each scenario. The Network issue is transient, but the GPS problem is likely permanent.

```
def temperatureCatchingMoreApp(): String =
  try
    render:
      getTemperatureOrThrow()
  catch
    case ex: NetworkException =>
      "Network Unavailable"
    case ex: GpsFail =>
      "GPS Hardware Failure"
```

```
scenario =
  ErrorsScenario.NetworkError

def run =
  ZIO.succeed:
    temperatureCatchingMoreApp()
// Result: Network Unavailable

scenario =
  ErrorsScenario.GPSError

def run =
  ZIO.succeed:
    temperatureCatchingMoreApp()
// Result: GPS Hardware Failure
```

Wonderful! We have specific messages for all relevant error cases. However, this still suffers from downsides that become more painful as the codebase grows.

- We do not know if `temperatureApp` can fail
- Once we know it can fail, we must dig through the documentation or implementation to discover the different possibilities
- Because every function that is called by `temperatureApp` can call other functions, which can call other functions, and so on, we are never sure that we have found all the failure paths in our application

More Problems

Exceptions have other problems:

1. The only way to ensure your program won't crash is by testing it through all possible execution paths.
2. It is difficult or impossible to retry an operation if it fails.

Exceptions were a valiant attempt to produce a consistent error-reporting interface, and they are better than what came before. You just don't know what you're going to get when you use exceptions.

Error Handling with ZIO

ZIO enables more powerful, uniform error-handling.

```
override val bootstrap =
  errorsHappyPath

def run =
  getTemperature
// Result: Temperature: 35 degrees
```

Running the ZIO version without handling any errors “‘scala override val bootstrap = errorsNetworkError

def run = getTemperature // Result: repl.MdocSession$MdocApp$NetworkException “‘

This is not an error that we want to show the user. Instead, we want to handle all of our internal errors, and make sure that they result in a user-friendly error message.

{{ TODO: mdoc seems to have a bug and is not outputting the compiler warning }} `scala val bad = getTemperature.catchAll: case ex: NetworkException => ZIO.succeed: "Network Unavailable" //`

ZIO distinguishes itself here by alerting us that we have not caught all possible errors. The compiler prevents us from executing non-exhaustive blocks inside of a `catchAll`.

```
val temperatureAppComplete =
  getTemperature.catchAll:
    case ex: NetworkException =>
      ZIO.succeed:
        "Network Unavailable"
    case ex: GpsFail =>
      ZIO.succeed:
        "GPS Hardware Failure"

override val bootstrap =
  errorsGpsError

def run =
  temperatureAppComplete
// Result: GPS Hardware Failure
```

Now that we have handled all of our errors, we know we are showing the user a sensible message.

Further, this is tracked by the compiler, which will prevent us from invoking `.catchAll` again.

```
temperatureAppComplete.catchAll:
  case ex: Exception =>
    ZIO.succeed:
      "This cannot happen"
// error:
// This error handling operation assumes your effect can fail. However,\
 your effect has Nothing for the error type, which means it cannot fail\
, so there is no need to handle the failure. To find out which method y\
ou can use instead of this operation, please see the reference chart at\
: https://zio.dev/can_fail.
// I found:
//
//     CanFail.canFail[E](/* missing */summon[scala.util.NotGiven[E =:=\
 Nothing]])
//
// But no implicit values were found that match type scala.util.NotGive\
n[E =:= Nothing].
```

The compiler also ensures that we only call the following methods on effects that can fail:

- retry*
- orElse*
- mapError
- fold*
- merge
- refine*
- tapError*

Wrapping Legacy Code

If we are unable to re-write the fallible function, we can still wrap the call.

TODO Explain ZIO.attempt

```
val getTemperatureWrapped =
  ZIO.attempt:
    getTemperatureOrThrow()

val displayTemperatureZWrapped =
  getTemperatureWrapped.catchAll:
    case ex: NetworkException =>
      ZIO.succeed:
        "Network Unavailable"
    case ex: GpsFail =>
      ZIO.succeed:
        "GPS problem"

scenario =
  ErrorsScenario.HappyPath

def run =
  displayTemperatureZWrapped
// Result: 35 degrees

scenario =
  ErrorsScenario.NetworkError

def run =
  displayTemperatureZWrapped
// Result: Network Unavailable
```

This is decent, but does not provide the maximum possible guarantees. Look at what happens if we forget to handle one of our errors.

```
scenario =
  ErrorsScenario.GPSError

def run =
  getTemperatureWrapped.catchAll:
    case ex: NetworkException =>
      ZIO.succeed:
        "Network Unavailable"
// Result: Defect: GpsFail
```

The compiler does not catch this bug, and instead fails at runtime. Take extra care when interacting with legacy code , since we cannot automatically recognize these situations at compile time. We can provide a fallback case that will report anything we missed:

```
scenario =
  ErrorsScenario.GPSError

def run =
  getTemperatureWrapped.catchAll:
    case ex: NetworkException =>
      ZIO.succeed:
        "Network Unavailable"
    case other =>
      ZIO.succeed:
        "Unknown Error"
// Result: Unknown Error
```

This lets us avoid the most egregious gaps in functionality, but does not take full advantage of ZIO's type-safety.

ZIO super powers for errors

Prevents Overly-Defensive Programming

Anything can be wrapped with a try/catch. Things that produce exceptions don't need to be wrapped with trys.

You are forbidden from `retry`'ing effects that cannot fail.

```
ZIO
  .succeed(println("Always gonna work"))
  .retryN(100)
// error:
// This error handling operation assumes your effect can fail. However,\
 your effect has Nothing for the error type, which means it cannot fail\
, so there is no need to handle the failure. To find out which method y\
ou can use instead of this operation, please see the reference chart at\
: https://zio.dev/can_fail.
// I found:
//
//     CanFail.canFail[E](/* missing */summon[scala.util.NotGiven[E =:=\
 Nothing]])
//
// But no implicit values were found that match type scala.util.NotGive\
```

```
n[E =:= Nothing].
// ZIO
//    ^

ZIO
  .attempt(println("This might work"))
  .retryN(100)
```

Flexible error types

Collections of Error

eg `collectAllSuccesses`

Composability

Edit This Chapter[10]

An essential part of creating programs is the ability to combine small pieces into larger pieces.
We call this *composability*. This might seem so simple that it must be a solved problem.

Languages and libraries provide different ways to enable composability.

- Objects can be composed by putting objects inside other objects.
- Functions can be composed by calling functions within other functions.

These approaches do not address all aspects of composition. For example, you cannot compose functions using resources that need to be opened and closed. Issues that complicate composition include:

- errors
- async
- blocking
- managed resource
- cancellation
- either-ness
- environmental requirements

These concepts and their competing solutions will be expanded on and contrasted with ZIO throughout this chapter.

[10] https://github.com/EffectOrientedProgramming/book/edit/main/Chapters/06_Composability.md

Universal Composability with ZIO (All The Thing Example)

ZIOs compose in a way that covers all of these concerns. The methods for composability depend on the desired behavior.

When writing substantial, complex applications , you will encounter APIs that that return limited data types.

ZIO provides conversion methods that take these limited data types and turn them into its single, universally composable type.

Existing Code

We will utilize several pre-defined functions that leverage less-complete effect alternatives.

Future

```
import scala.concurrent.Future
```

The original asynchronous datatype in Scala has several undesirable characteristics:

- Cleanup is not guaranteed
- Start executing immediately
- Must all fail with Exception
- Needs `ExecutionContexts` passed everywhere

There is a function that returns a Future:

```
getHeadLine(???): Future[String]
```

TODO This is repetitive after listing the downsides above. By wrapping this in `ZIO.from`, it will:

- get the `ExecutionContext` it needs
- Defer execution of the code
- Let us attach finalizer behavior
- Give us the ability to customize the error type

```
def getHeadlineZ(scenario: Scenario) =
  ZIO
    .from:
      getHeadLine(scenario)
    .mapError:
      case _: Throwable =>
        Scenario.HeadlineNotAvailable()

def run =
  getHeadlineZ(Scenario.StockMarketHeadline())
// Result: stock market rising!
```

Now let's confirm the behavior when the headline is not available.

```
def run =
  getHeadlineZ(Scenario.HeadlineNotAvailable())
// Result: HeadlineNotAvailable()
```

Option

`Option` is the simplest of the alternate types you will encounter. It does not deal with asynchronicity, error types, or anything else. It merely indicates that a value might not be available.

- Execution is not deferred
- Cannot interrupt the code that is producing these values

```
val result: Option[String] =
  findTopicOfInterest:
    "content"
```

If you want to treat the case of a missing value as an error, you can again use `ZIO.from`: ZIO will convert `None` into a generic error type, giving you the opportunity to define a more specific error type.

```
def topicOfInterestZ(headline: String) =
  ZIO
    .from:
      findTopicOfInterest:
        headline
    .orElseFail:
      Scenario.NoInterestingTopic()

def run =
  topicOfInterestZ:
    "stock market rising!"
// Result: stock market

def run =
  topicOfInterestZ:
    "boring and inane content"
// Result: NoInterestingTopic()
```

Either

- Execution is not deferred
- Cannot interrupt the code that is producing these values

We have an existing function `wikiArticle` that checks for articles on a topic:

```
val wikiResult: Either[
  Scenario.NoWikiArticleAvailable,
  String
] =
  wikiArticle("stock market")

def wikiArticleZ(topic: String) =
  ZIO.from:
    wikiArticle:
      topic
```

```
def run =
  wikiArticleZ:
    "stock market"
// Wiki - articleFor(stock market)
// Result: detailed history of stock market
```

```
def run =
  wikiArticleZ:
    "barn"
// Wiki - articleFor(barn)
// Result: NoWikiArticleAvailable()
```

AutoCloseable

Java/Scala provide the `AutoCloseable` interface for defining finalizer behavior on objects. While this is a big improvement over manually managing this in ad-hoc ways, the static scoping of this mechanism makes it clunky to use.

TODO Decide whether to show nested files example to highlight this weakness

We have an existing function that produces an `AutoCloseable`.

```
val file: AutoCloseable =
  openFile()
```

Since `AutoCloseable` is a trait that can be implemented by arbitrary classes, we can't rely on `ZIO.from` to automatically manage this conversion for us. In this situation, we should use the explicit `ZIO.fromAutoCloseable` function.

```
val closeableFileZ =
  ZIO.fromAutoCloseable:
    ZIO.succeed:
      openFile()
```

Once we do this, the `ZIO` runtime will manage the lifecycle of this object via the `Scope` mechanism. TODO Link to docs for this?

Now we open a `File`, and check if it contains a topic of interest.

```
def run =
  defer:
    val file =
      closeableFileZ.run
    file.contains:
      "topicOfInterest"
// File - OPEN
// File - contains(topicOfInterest)
// File - CLOSE
// Result: false
```

Now we highlight the difference between the static scoping of `Using` or `ZIO.fromAutoCloseable`.

```
import scala.util.Using
import java.io.FileReader

Using(openFile()) {
  file1 =>
    Using(openFile()) {
      file2 =>
        // TODO Use reader1 and reader2
    }
}
```

```
def run =
  defer:
    val file1 =
      closeableFileZ.run
    val file2 =
      closeableFileZ.run
// File - OPEN
// File - OPEN
// File - CLOSE
// File - CLOSE
// Result: ()
```

Try

We continue using our `File`, but now we write to it. The existing API uses a `Try` to indicate success or failure.

```
val writeResult: Try[String] =
  openFile().write("asdf")

def writeToFileZ(file: File, content: String) =
  ZIO
    .from:
      file.write:
        content
    .mapError(
      _ => Scenario.DiskFull()
    )

def run =
  defer:
    val file =
      closeableFileZ.run
    writeToFileZ(file, "New data on topic").run
// File - OPEN
// File - write: New data on topic
// File - CLOSE
// Result: New data on topic
```

Functions that throw

```
openFile().summaryFor("asdf"): String

case class NoSummaryAvailable(topic: String)
def summaryForZ(
    file: File,
    // TODO Consider making a CloseableFileZ
    topic: String
) =
  ZIO
    .attempt:
      file.summaryFor(topic)
    .mapError(
      _ => NoSummaryAvailable(topic)
    )
```

TODO: - original function: File.summaryFor - wrap with ZIO - call zio version in AllTheThings

Downsides: - We cannot union these error possibilities and track them in the type system - Cannot attach behavior to deferred functions - do not put in place a contract

Slow, blocking functions

TODO Decide example functionality - AI analysis of news content?

TODO Prose about the long-running AI process here

```
// TODO Can we use silent instead of compile-only above?
summarize("some topic"): String
```

This gets interrupted, although it takes a big performance hit `scala def summarizeZ(article: String) = ZIO .attemptBlockingInterrupt: summarize(article) .onInterrupt: ZIO.debug("AI **INTERRUPTED**") .orDie // TODO Confirm we don't care about this case. .timeoutFail(Scenario.AITooSlow())(50.millis)`

- We can't indicate if they block or not
- Too many concurrent blocking operations can prevent progress of other operations
- Very difficult to manage
- Blocking performance varies wildly between environments

Sequencing

Another term for this form of composition is called `andThen` in Scala.

With ZIO you can use `zio-direct` to compose ZIOs sequentially with:

```
def run =
  defer:
    val topStory =
      findTopNewsStory.run
    textAlert(topStory).run
// Texting story: Battery Breakthrough
// Result: ()
```

Final Collective Criticism

Each of original approaches gives you benefits, but you can't easily assemble a program that utilizes all of them. They must be manually transformed into each other .

Instead of the best of all worlds, you get the pain of all worlds. eg `Closeable[Future[Either[Throwable, A]]]` The ordering of the nesting is significant, and not easily changed.

The number of combinations is something like: PairsIn(numberOfConcepts)

Fully Composed

Now that we have all of these well-defined effects, we can wield them in any combination and sequence we desire.

```
def researchHeadline(scenario: Scenario) =
  defer:
    val headline: String =
      getHeadlineZ(scenario).run

    val topic: String =
      topicOfInterestZ(headline).run

    val summaryFile: File =
      closeableFileZ.run

    val knownTopic: Boolean =
      summaryFile.contains:
        topic

    if (knownTopic)
      summaryForZ(summaryFile, topic).run
    else
      val wikiArticle: String =
        wikiArticleZ(topic).run

      val summary: String =
        summarizeZ(wikiArticle).run
```

```
      writeToFileZ(summaryFile, summary).run
      summary

def run =
  researchHeadline:
    Scenario.HeadlineNotAvailable()
// Result: HeadlineNotAvailable()

def run =
  researchHeadline:
    Scenario.SummaryReadThrows()
// File - OPEN
// File - contains(unicode)
// File - summaryFor(unicode)
// File - CLOSE
// Result: NoSummaryAvailable(unicode)

def run =
  researchHeadline:
    Scenario.NoWikiArticleAvailable()
// File - OPEN
// File - contains(barn)
// Wiki - articleFor(barn)
// File - CLOSE
// Result: NoWikiArticleAvailable()

def run =
  researchHeadline:
    Scenario.AITooSlow()
// File - OPEN
// File - contains(space)
// Wiki - articleFor(space)
// AI - summarize - start
// printing because our test clock is insane
// AI **INTERRUPTED**
// File - CLOSE
// Result: AITooSlow()
```

```
def run =
  researchHeadline:
    // TODO Handle inconsistency in this example
    // AI keeps timing out
    Scenario.DiskFull()
// File - OPEN
// File - contains(genome)
// Wiki - articleFor(genome)
// AI - summarize - start
// AI - summarize - end
// File - CLOSE
// Result: AITooSlow()
```

And finally, we see the longest, successful pathway through our application:

```
def run =
  researchHeadline:
    Scenario.StockMarketHeadline()
// File - OPEN
// File - contains(stock market)
// Wiki - articleFor(stock market)
// AI - summarize - start
// AI - summarize - end
// File - write: market is not rational
// File - CLOSE
// Result: market is not rational
```

Repeats

Repeating is a form of composability, because you are composing a program with itself

Injecting Behavior before/after/around

Graveyard candidates

Contract-based prose

Good contracts make good composability.

contracts are what makes composability work at scale our effects put in place contracts on how things can compose

Plain functions that return Unit TODO Incorporate to AllTheThings

{{TODO Decide if this section is worth keeping. If so, where?}}

`Unit` can be viewed as the bare minimum of effect tracking.

Consider a function

```
def saveInformation(info: String): Unit =
  ???
```

If we look only at the types, this function is an `String=>Unit`. `Unit` is the single, blunt tool to indicate effectful functions in plain Scala. When we see it, we know that *some* type of side-effect is being performed.

When a function returns `Unit`, we know that the only reason we are calling the function is to perform an effect. Alternatively, if there are no arguments to the function, then the input is `Unit`, indicating that an effect is used to *produce* the result.

Unfortunately, we can't do things like timeout/race/etc these functions. We can either execute them, or not, and that's about it, without resorting to additional tools for manipulating their execution.

State

Edit This Chapter[11]

1. Ref

Functional programmers often sing the praises of immutability. The advantages are real and numerous. However, it is easy to find situations that are intrinsically mutable.

- How many people are currently inside a building?
- How much fuel is in your car?
- How much money is in your bank account?

Rather than avoiding mutability entirely, we want to avoid unprincipled, unsafe mutability. If we codify and enumerate everything that we need from Mutability, then we can wield it safely. Required Operations:

- Update the value
- Read the current value

These are both effectful operations. In order to confidently use them, we need certain guarantees about the behavior:

- The underlying value cannot be changed during a read
- Multiple writes cannot happen concurrently, which would result in lost updates

Less obviously, we also need to create the Mutable reference itself. We are changing the world, by creating a space that we can manipulate.

Unreliable State

[11] https://github.com/EffectOrientedProgramming/book/edit/main/Chapters/07_State.md

```
val unreliableCounting =
  var counter =
    0
  val increment =
    ZIO.succeed:
      counter =
        counter + 1

  defer:
    ZIO
      .foreachParDiscard(Range(0, 100000)):
        _ => increment
      .run
    // It's not obvious to the reader why
    // we need to wrap counter in .succeed
    "Final count: " + ZIO.succeed(counter).run

def run =
  unreliableCounting
// Result: Final count: 99877
```

Due to the unpredictable nature of shared mutable state, we do not know exactly what the final count above is. Each time we publish a copy of this book, the code is re-executed and a different wrong result is generated. However, conflicts are extremely likely, so some of our writes get clobbered by others, and we end up with less than the expected 100,000. Ultimately, we lose information with this approach.

```
TODO Consider making a diagram parallel writes
```

Performing our side effects inside ZIO's does not magically make them safe. We need to fully embrace the ZIO components, utilizing `Ref` for correct mutation.

Reliable State

```
lazy val reliableCounting =
  def incrementCounter(counter: Ref[Int]) =
    counter.update:
      _ + 1

  defer:
    val counter =
      Ref.make(0).run
    ZIO
      .foreachParDiscard(Range(0, 100000)):
        _ =>
          incrementCounter:
            counter
      .run
    "Final count: " + counter.get.run

def run =
  reliableCounting
// Result: Final count: 100000
```

Now we can say with full confidence that our final count is 100000. Additionally, these updates happen *without blocking*. This is achieved through a strategy called "Compare & Swap", which we will not cover in detail. *TODO Link/reference supplemental reading*

Unreliable Effects

Although there are significant advantages; a basic `Ref` is not the solution for everything. We can only pass pure functions into `update`. The API of the plain Atomic `Ref` steers you in the right direction by not accepting `ZIO`s as parameters to any of its methods. To demonstrate why this restriction exists, we will deliberately undermine the system by sneaking in a side effect. First, we will create a helper function that imitates a long-running calculation.

```
def expensiveCalculation() =
  Thread.sleep:
    35
```

Our side effect will be a mock alert that is sent anytime our count is updated: `scala def sendNotification() = println: "Alert: updating count!"`

```
def update(counter: Ref[Int]) =
  counter.update:
    previousValue =>
      expensiveCalculation()
      sendNotification()
      previousValue + 1

def run =
  defer:
    val counter =
      Ref.make(0).run
    ZIO
      .foreachParDiscard(Range(0, 4)):
        _ => update(counter)
      .run
    val finalCount =
      counter.get.run
    s"Final count: $finalCount"
// Alert: updating count!
// Alert: updating count!
// Alert: updating count!
// Alert: updating count!
// Alert: updating count!
// Alert: updating count!
// Alert: updating count!
// Alert: updating count!
// Alert: updating count!
// Alert: updating count!
// Result: Final count: 4
```

What is going on?! Previously, we were losing updates because of unsafe mutability. Now, we have the opposite problem! We are sending far more alerts than intended, even though we can see that our final count is 4.

TODO This section will need significant attention and polish

Now we must consider the limitations of the "Compare & Swap" system. It achieves lock-free performance by letting each fiber freely make their updates, and then doing a last-second check to see if the underlying value changed during its update. If the value has not changed, the update is made. If it has changed, then the entire function that was passed into `update` is re-executed until it completes with a stable value. The higher the parallelism, or the longer the operation takes, the higher the likelihood of a compare-and-swap retry.

This retry behavior is safe with pure functions, which can be executed an arbitrary number of times. However, it is completely inappropriate for effects, which should only be executed a single time. For these situations, we need a specialized variation of `Ref`

Reliable Effects

`Ref.Synchronized` guarantees only a single execution of the `update` body and any of the effects contained inside. The only change required is replacing `Ref.make` with `Ref.Synchronized.make`

```
val sideEffectingUpdatesSync =
  defer:
    val counter =
      Ref.Synchronized.make(0).run
    ZIO
      .foreachParDiscard(Range(0, 4)):
        _ => update(counter)
      .run
    val finalCount =
      counter.get.run
    s"Final count: $finalCount"
```

```
def run =
  sideEffectingUpdatesSync
// Alert: updating count!
// Alert: updating count!
// Alert: updating count!
// Alert: updating count!
// Result: Final count: 4
```

Now we see exactly the number of alerts that we expected. This correctness comes with a cost though, as the name of this type implies. Each of your updates will run sequentially, despite initially launching them all in parallel. This is the only known way to avoid retries. Try to structure your code to minimize the coupling between effects and updates, and use this type only when necessary.

Reliability

Edit This Chapter[12]

For our purposes, A reliable system behaves predictably in normal circumstances as well as high loads or even hostile situations. If failures do occur, the system either recovers or shuts down in a well-defined manner.

Effects are the parts of your system that are unpredictable. When we talk about reliability in terms of effects, the goal is to mitigate these unpredictabilities. For example, if you make a request of a remote service, you don't know if the network is working or if that service is online. Also, the service might be under a heavy load and slow to respond. There are strategies to compensate for those issues without invasive restructuring. For example, we can attach fallback behavior: make a request to our preferred service, and if we don't get a response soon enough, make a request to a secondary service.

Traditional coding often requires extensive re-architecting to apply and adapt reliability strategies, and further rewriting if they fail. In a functional effect-based system, reliability strategies can be easily incorporated and modified. This chapter demonstrates components that enhance effect reliability.

Caching

Putting a cache in front of a service can resolve when a service is:

- Slow: the cache can speed up the response time.
- Brittle: the cache can provide a stable response and minimize the risk of overwhelming the resource.
- Expensive: the cache can reduce the number of calls to it, and thus reduce your operating cost.

[12] https://github.com/EffectOrientedProgramming/book/edit/main/Chapters/08_Reliability.md

To demonstrate, we will take one of the worst case scenarios that your service might encounter: the thundering herd problem. If you have a steady stream of requests coming in, any naive cache can store the result after the first request, and then be ready to serve it to all subsequent requests. However, it is possible that all the requests will arrive before the first one has been served and the value has been cached. In this case, a naive cache would cause each request to call the underlying slow/brittle/expensive service and then they would each update the cache with the identical value.

Here is what that looks like:

```
val thunderingHerdsScenario =
  defer:
    val popularService =
      ZIO.service[PopularService].run

    // All requests arrive at once
    ZIO
      .collectAllPar:
        List.fill(100):
          popularService.retrieve:
            Paths.get("awesomeMemes")
      .run

    val cloudStorage =
      ZIO.service[CloudStorage].run

    cloudStorage.invoice.run
```

We first show the uncached service:

```
val makePopularService =
  defer:
    val cloudStorage =
      ZIO.service[CloudStorage].run
    PopularService(cloudStorage.retrieve)
```

To construct a `PopularService`, we give it the effect that looks up content. In this version, it goes directly to the `CloudStorage` provider.

Suppose each request to our `CloudStorage` provider costs one dollar.

```
def run =
  thunderingHerdsScenario
    .provide(
      CloudStorage.live,
      ZLayer.fromZIO(makePopularService)
    )
// Result: Amount owed: $100
```

The invoice is 100 dollars because every single request reached our `CloudStorage` provider.

Now let's construct a `PopularService` that uses a cache:

```
val makeCachedPopularService =
  defer:
    val cloudStorage =
      ZIO.service[CloudStorage].run
    val cache =
      Cache
        .make(
          capacity =
            100,
          timeToLive =
            Duration.Infinity,
          lookup =
            Lookup(cloudStorage.retrieve)
        )
        .run

    PopularService(cache.get)
```

The only changes required are:

- Building the cache with sensible values
- Passing the `Cache.get` method to the `PopularService` constructor, instead of the bare `CloudStorage.retrieve` method

Now we run the same scenario with the cache in place:

```
def run =
  thunderingHerdsScenario.provide(
    CloudStorage.live,
    ZLayer.fromZIO(makeCachedPopularService)
  )
// Result: Amount owed: $1
```

The invoice is only 1 dollar, because only one request reached the `CloudStorage` provider. Wonderful! In practice, the savings will rarely be *this* extreme, but it is reassuring to know we can handle these situations with ease.

Staying under rate limits

Rate limits are a common way to structure agreements between services. In the worst case, going above this limit could overwhelm the service and make it crash. At the very least, you will be charged more for exceeding it.

TODO Show un-limited demo first?

Constructing a RateLimiter

Defining your rate limiter requires only the 2 pieces of information that should be codified in your service agreement:

```
$maxRequests / $interval
```

```
import nl.vroste.rezilience.RateLimiter

val makeRateLimiter =
  RateLimiter.make(
    max =
      1,
    interval =
      1.second
  )
```

```
// TODO explain timedSecondsDebug
def makeCalls(name: String) =
  expensiveApiCall
    .timedSecondsDebug:
      s"$name called API"
    .repeatN(2) // Repeats as fast as allowed
```

Now, we wrap our unrestricted logic with our `RateLimiter`. Even though the original code loops as fast the CPU allows, it will now adhere to our limit.

```
def run =
  defer:
    val rateLimiter =
      makeRateLimiter.run
    rateLimiter:
      makeCalls:
        "System"
    .timedSecondsDebug("Result")
      .run
// System called API [took 0s]
// System called API [took 0s]
// System called API [took 0s]
// Result [took 0s]
// Result: ()
```

Most impressively, we can use the same `RateLimiter` across our application. No matter the different users/features trying to hit the same resource, they will all be limited such that the entire application respects the rate limit.

```
def run =
  defer:
    val rateLimiter =
      makeRateLimiter.run
    val people =
      List("Bill", "Bruce", "James")

    ZIO
      .foreachPar(people):
        person =>
          rateLimiter:
            makeCalls(person)
      .timedSecondsDebug:
```

```
        "Total time"
      .run
// Bill called API [took 0s]
// Bill called API [took 0s]
// Bill called API [took 0s]
// James called API [took 0s]
// James called API [took 0s]
// James called API [took 0s]
// Bruce called API [took 0s]
// Bruce called API [took 0s]
// Bruce called API [took 0s]
// Total time [took 2s]
// Result: List((), (), ())
```

Constraining concurrent requests

If we want to ensure we don't accidentally DDOS a service, we can restrict the number of concurrent requests to it.

First, we demonstrate the unrestricted behavior:

```
def run =
  defer:
    val delicateResource =
      ZIO.service[DelicateResource].run
    ZIO
      .foreachPar(1 to 10):
        _ => delicateResource.request
      .as("All Requests Succeeded!")
      .run
  .provide(DelicateResource.live)
// Delicate Resource constructed.
// Do not make more than 3 concurrent requests!
// Current requests: : List(913)
// Current requests: : List(779, 913)
// Current requests: : List(765, 779, 913)
// Current requests: : List(746, 640, 765, 779, 913)
// Current requests: : List(274, 746, 640, 765, 779, 913)
// Current requests: : List(640, 765, 779, 913)
// Result: Crashed the server!!
```

We execute too many concurrent requests, and crash the server. To prevent this, we need a `Bulkhead`.

```
import nl.vroste.rezilience.Bulkhead
val makeOurBulkhead =
  Bulkhead.make(maxInFlightCalls =
    3
  )
```

Next, we wrap our original request with this `Bulkhead`.

```
def run =
  defer:
    val bulkhead =
      makeOurBulkhead.run

    val delicateResource =
      ZIO.service[DelicateResource].run
    ZIO
      .foreachPar(1 to 10):
        _ =>
          bulkhead:
            delicateResource.request
      .as("All Requests Succeeded")
      .run
  .provide(DelicateResource.live, Scope.default)
// Delicate Resource constructed.
// Do not make more than 3 concurrent requests!
// Current requests: : List(911)
// Current requests: : List(242, 911)
// Current requests: : List(9, 242, 911)
// Current requests: : List(868)
// Current requests: : List(983, 868)
// Current requests: : List(633, 983, 868)
// Current requests: : List(97)
// Current requests: : List(559, 97)
// Current requests: : List(331, 559, 97)
// Current requests: : List(409, 331, 559)
// Result: All Requests Succeeded
```

With this small adjustment, we now have a complex, concurrent guarantee.

Circuit Breaking

Often, when a request fails, it is reasonable to immediately retry. However, if we aggressively retry in an unrestricted way, we might actually make the problem worse by increasing the load on the struggling service. Ideally, we would allow some number of aggressive retries, but then start blocking additional requests until the service has a chance to recover.

In this scenario, we are going to repeat our call many times in quick succession.

```
val repeatSchedule =
  Schedule.recurs(140) &&
    Schedule.spaced(50.millis)
```

When unrestrained, the code will let all the requests through to the degraded service.

```
def run =
  defer:
    val numCalls =
      Ref.make[Int](0).run

    externalSystem(numCalls)
      .ignore
      .repeat(repeatSchedule)
      .run

    val made =
      numCalls.get.run

    s"Calls made: $made"
// Result: Calls made: 141
```

Now we will build our `CircuitBreaker`

```
import nl.vroste.rezilience.{
  CircuitBreaker,
  TrippingStrategy,
  Retry
}

val makeCircuitBreaker =
  CircuitBreaker.make(
    trippingStrategy =
      TrippingStrategy.failureCount(maxFailures =
        2
      ),
    resetPolicy =
      Retry.Schedules.common()
  )
```

Once again, the only thing that we need to do is wrap our original effect with the `CircuitBreaker`.

```
import CircuitBreaker.CircuitBreakerOpen
def run =
  defer:
    val cb =
      makeCircuitBreaker.run
    // TODO Can we move these Refs into the
    // external system?
    val numCalls =
      Ref.make[Int](0).run
    val numPrevented =
      Ref.make[Int](0).run
    val protectedCall =
      cb(externalSystem(numCalls)).catchSome:
        case CircuitBreakerOpen =>
          numPrevented.update(_ + 1)

    protectedCall
      .ignore
      .repeat(repeatSchedule)
      .run

    val prevented =
      numPrevented.get.run
```

```
    val made =
      numCalls.get.run
    s"Calls prevented: $prevented Calls made: $made"
// Result: Calls prevented: 75 Calls made: 66
```

{{TODO Fix output after `OurClock` changes}} Now we see that our code prevented the majority of the doomed calls to the external service.

Hedging

This technique snips off the most extreme end of the latency tail.

Determine the average response time for the top 50% of your requests. If you make a call that does not get a response within this average delay, make an additional, identical request. However, you do not give up on the 1st request, since it might respond immediately after sending the 2nd. Instead, you want to race them and use the first response you get

To be clear - this technique will not reduce the latency of the fastest requests at all. It only alleviates the pain of the slowest responses.

You have `1/n` chance of getting the worst case response time. This approach turns that into a `1/n^2` chance. The cost of this is only ~3% more total requests made. *Citations needed*

Further, if this is not enough to completely eliminate your extreme tail, you can employ the exact same technique once more. Then, you end up with `1/n^3` chance of getting that worst performance.

```
def run =
  defer:
    val contractBreaches =
      Ref.make(0).run

    ZIO
      .foreachPar(List.fill(50_000)(())):
        _ => // james still hates this
          defer:
            val hedged =
              logicThatSporadicallyLocksUp.race:
                logicThatSporadicallyLocksUp
                  .delay:
                    25.millis

            // TODO How do we make this demo more
            // obvious?
            // The request is returning the
            // hypothetical runtime, but that's
            // not clear from the code that will
            // be visible to the reader.
            val duration =
              hedged.run
            if (duration > 1.second)
              contractBreaches.update(_ + 1).run
      .run

    contractBreaches
      .get
      .debug("Contract Breaches")
      .run
// Contract Breaches: 0
// Result: 0
```

Restricting Time

Sometimes, it's not enough to simply track the time that a test takes. If you have specific Service Level Agreements (SLAs) that you need to meet, you want your tests to help ensure that you are meeting them. However, even if you don't have

contracts bearing down on you, there are still good reasons to ensure that your tests complete in a timely manner. Services like GitHub Actions will automatically cancel your build if it takes too long, but this only happens at a very coarse level. It simply kills the job and won't actually help you find the specific test responsible.

A common technique is to define a base test class for your project that all of your tests extend. In this class, you can set a default upper limit on test duration. When a test violates this limit, it will fail with a helpful error message.

This helps you to identify tests that have completely locked up, or are taking an unreasonable amount of time to complete.

For example, if you are running your tests in a CI/CD pipeline, you want to ensure that your tests complete quickly, so that you can get feedback as soon as possible. you can use `TestAspect.timeout` to ensure that your tests complete within a certain time frame.

Flakiness

Commonly, as a project grows, the supporting tests become more and more flaky. This can be caused by a number of factors:

- The code is using shared, live services Shared resources, such as a database or a file system, might be altered by other processes. These could be other tests in the project, or even unrelated processes running on the same machine.
- The code is not thread safe Other processes running simultaneously might alter the expected state of the system.
- Resource limitations A team of engineers might be able to successfully run the entire test suite on their personal machines. However, the CI/CD system might not have enough resources to run the tests triggered by everyone pushing to the repository. Your tests might be occasionally failing due to timeouts or lack of memory.

ZIO and ZLayer

Edit This Chapter[13]

Connect the DI & Errors chapter to how they are represented in the ZIO data type.

The way we get good compile errors is by having data types which "know" the ...

ZIO

We need an `Answer` about this scenario. The scenario requires things and could produce an error. "'

trait ZIO[Requirements, Error, Answer] "'

The `ZIO` trait is at the center of our Effect-oriented world.

```
trait ZIO[R, E, A]
```

A trait with 3 type parameters can be intimidating, but each one serves a distinct, important purpose.

R - Requirements

TODO

E - Errors

This parameter tells us how this operation might fail.

[13] https://github.com/EffectOrientedProgramming/book/edit/main/Chapters/09_ZIO_and_ZLayer.md

```
def parse(
    contents: String
): ZIO[Any, IllegalArgumentException, Unit] =
  ???
```

A - Answer

This is what our code will return if it completes successfully.

```
def defaultGreeting()
    : ZIO[Any, Nothing, String] =
  ???
```

ZLayer

R - Requirements

E - Errors ### A - Answer

Composing

Managing and wiring dependencies has been a perennial challenge in software development.

ZIO provides the `ZLayer` class to solve many of the problems in this space. If you pay the modest, consistent cost of constructing pieces of your application as `ZLayer`s, you will get benefits that scale with the complexity of your project. Consistent with `ZIO` itself, `ZLayer` has 3 type parameters that represent:

- What it needs from the environment
- How it can fail
- What it produces when successful.

With the same type parameters, and many of the same methods, you might be wondering why we even need a separate data type - why not just use `ZIO` itself for our dependencies? The environment type parameter for `ZLayer` maps directly to unique, singleton services in your application. The environment type parameter for `ZIO` might have *many* possible instances. `ZLayer` provides additional behaviors that are valuable specifically in this domain. Typically, you only want a single instance of a dependency to exist across your application. This may be to reduce memory/resource usage, or even to ensure basic correctness. `ZLayer` output values are shared maximally by default. They also build in scope management that will ensure resource cleanup in asynchronous, fallible situations.

ZIO <-> ZLayer

`ZLayer.fromZIO`

Appendix Running Effects

Edit This Chapter[14]

TODO: Make an appendix?

Building applications from scratch

One way to run ZIOs is to use a "main method" program (something you can start in the JVM). However, setting up the pieces needed for this is a bit cumbersome if done without helpers.

ZIOAppDefault

ZIO provides an easy way to do this with the `ZIOAppDefault` trait.

To use it create a new `object` that extends the `ZIOAppDefault` trait and implements the `run` method. That method returns a ZIO so you can now give it the example `ZIO.debug` data:

```
object HelloWorld extends zio.ZIOAppDefault:
  def run =
    ZIO.debug:
      "hello, world"
```

This can be run on the JVM in the same way as any other class that has a `static void main` method.

The `ZIOAppDefault` trait sets up the ZIO runtime which interprets ZIOs and provides some out-of-the-box functionality, and then runs the provided data in that runtime.

[14] https://github.com/EffectOrientedProgramming/book/edit/main/Chapters/10_Appendix_RunningEffects.md

If you are learning ZIO, you should start your exploration with `ZIOAppDefault`. It is the standard, simplest way to start executing your recipes.

For this book we shorten the definition for running ZIO Effects to: `scala def run = ZIO.debug: "hello, world" // hello, world // Result: ()`

```
// NOTE We cannot execute invoke main on this
// because it crashes mdoc in the CI process
object RunningZIOs extends ZIOAppDefault:
  def run =
    Console.printLine:
      "Hello World!"
```

You can provide arbitrary ZIO instances to the run method, as long as you have provided every piece of the environment. In other words, it can accept `ZIO[Any, _, _]`.

There is a more flexible `ZIOApp` that facilitates sharing layers between applications, but this is advanced and not necessary for most applications.

Testing code

ZIOSpecDefault

Similar to `ZIOAppDefault`, there is a `ZIOSpecDefault` that should be your starting point for testing ZIO applications. `ZIOSpecDefault` provides test-specific implementations built-in services, to make testing easier. When you run the same `ZIO` in these 2 contexts, the only thing that changes are the built-in services provided by the runtime.

TODO - Decide which scenario to test

```
object TestingZIOs extends ZIOSpecDefault:
  def spec =
    test("Hello Tests"):
      defer:
        ZIO.console.run
        assertTrue:
          Random.nextIntBounded(10).run > 10
```

For this book we can shorten the test definition to: `scala def spec = test("random is random"): defer: assertTrue: Random.nextIntBounded(10).run < 10 // + random is random // Result: Summary(1,0,0,,PT0.461185S)`

TODO Justify defer syntax over for-comp for multi-statement assertions I think this example completes the objective TODO Change this to a Console app, where the logic & testing is more visceral

```
def spec =
  test("random is still random"):
    defer:
      assertTrue:
        Random.nextIntBetween(0, 10).run <= 10 &&
        Random.nextIntBetween(10, 20).run <=
          20 &&
          Random.nextIntBetween(20, 30).run <= 30
// + random is still random
// Result: Summary(1,0,0,,PT0.083948S)
```

Consider a `Console` application:

```
val logic =
  defer:
    val username =
      Console
        .readLine:
          "Enter your name\n"
        .run
    Console
      .printLine:
        s"Hello $username"
      .run
  .orDie
```

If we try to run this code in the same way as most of the examples in this book, we encounter a problem.

```
object HelloWorldWithTimeout
    extends zio.ZIOAppDefault:
  def run =
    logic.timeout(1.second)
```

We cannot execute this code and render the results for the book because it requires interaction with a user. However, even if you are not trying to write demo code for a book, it is very limiting to need a user at the keyboard for your program to execute. Even for the smallest programs, it is slow, error-prone, and boring.

```
def spec =
  test("console works"):
    defer:
      TestConsole
        .feedLines:
          "Zeb"
        .run

      logic.run

      val capturedOutput: String =
        TestConsole.output.run.mkString
      val expectedOutput =
        s"""|Enter your name
            |Hello Zeb
            |""".stripMargin
      assertTrue:
        capturedOutput == expectedOutput
// - console works
//   Exception in thread "zio-fiber-1842749958" scala.NotImplementedErr\
or: an implementation is missing
//           at scala.Predef$.$qmark$qmark$qmark(Predef.scala:344)
//           at mdoctools.OurConsole.print(OurConsole.scala:14)
//           at zio.Console$.print$$anonfun$6(Console.scala:122)
//           at zio.ZIO$.consoleWith$$anonfun$1(ZIO.scala:3061)
//           at zio.FiberRef$unsafe$$anon$2.getWith$$anonfun$1(FiberRef.scala:\
474)
//           at repl.MdocSession.MdocApp.logic(<input>:83)
//           at zio.direct.ZioMonad.Success.$anon.flatMap(ZioMonad.scala:19)
```

```
//           at repl.MdocSession.MdocApp.logic(<input>:93)
//           at zio.direct.ZioMonad.Success.$anon.flatMap(ZioMonad.scala:19)
//           at repl.MdocSession.MdocApp.Chapter70Spec.spec(<input>:127)
// Result:
// - console works
//   Exception i
```

Interop with existing/legacy code via Unsafe

In some cases your ZIOs may need to be run outside a *main* program, for example when embedded into other programs. In this case you can use ZIO's `Unsafe` utility which is called `Unsafe` to indicate that the code may perform side effects.
To do the same `ZIO.debug` with `Unsafe` do:

```
val out =
  Unsafe.unsafe:
    implicit u: Unsafe =>
      Runtime
        .default
        .unsafe
        .run:
          ZIO.debug:
            "hello, world"
        .getOrThrowFiberFailure()
// hello, world
```

If needed you can even interop to Scala Futures through `Unsafe`, transforming the output of a ZIO into a Future.