Haoyi's Programming Blog (..)

About (../post/HelloWorldBlog.html) Resume (https://lihaoyi.github.io/Resume/) Github (https://github.com/li

Twitter (https://twitter.com/li_haoyi) Subscribe (http://eepurl.com/c3A5Tz) Talks (../post/TalksIveGiven.html)

Strategic Scala Style: Principle of Least Power

Posted 2016-02-14 (https://github.com/lihaoyi/blog/commit/00f45a23193e4545471a1ce4d3737ee67f657e79)

← Code Reviewing My Earliest Surviving Program (CodeReviewingMyEarliestSurvivingProgram.html)Talks I've Given → (TalksIveGiven.html)

The Scala language is large and complex, and it provides a variety of tools that a developer can use to do the same thing in a variety of ways. Given the range of possible solutions to every problem, how can a developer choose which one should be used? This is the first in a series of blog posts aiming to provide style guidelines at a "strategic" level. Above the level of "how much whitespace should I use" or camelCase vs PascalCase, it should help a developer working with the Scala language choose from the buffet of possible solutions.

About Strategic Scala Style

These guidelines are based on my own experience working on open- and closed-source projects in Scala. Despite that, they all follow from a coherent set of fundamental principles, which hopefully will provide justification and beyond the normal "I prefer you prefer" nature of these discussions.

These guidelines will all assume that you already know most of Scala's language features and what can be done with them, and will focus purely on how to choose between them when picking your solution. It sticks purely to "Vanilla Scala" and its standard library features/APIs: you will not find anything regarding e.g. Akka or Scalaz in here.

No doubt people coming from "Monadic" or "Reactive" or "Type-level" or "Scala.js" camps (i.e. basically everyone) would disagree with some guidelines. Nevertheless, hopefully the over-all document is still broadly applicable enough that disagreements become

> "I think these points should be re-ordered"

or

> "Here's another technique that could be slotted in here"

Rather than

> "Everything is wrong and this is terrible"

You may agree or disagree with any of these; let me know in the comments below!

Quick Reference: Principle of Least Power

Here are the principles behind this set of guidelines:

- **Philosophy: Principle of Least Power**
  1. *Complexity is your Enemy*
  2. *Don't Fear Refactoring*
  3. *Don't Over Engineer*

Here is a listing of all the guidelines at a glance

- **Immutability & Mutability**
  1. *Use immutability as far as possible*
  2. *Unless you are actually modeling mutable things*
  3. *Or for performance*
  4. *Even then, scope it as tightly as possible*
  5. *Don't use double-mutability, you probably don't need it*
  6. *Don't use Event-Sourcing/CQRS unless you know what you're doing*
- **Published Interfaces**
  1. *The simplest interface to a package is static method with standard types*
  2. *Next, static methods that take/return custom types, presumably with methods*
  3. *Next, is needing the user to instantiate some classes*
  4. *Lastly, is needing the user to inherit from some classes*
- **Data Types**
  1. *Use built-in primitives, collection & combinations of them when possible*
  2. *Use opaque functions when you just need a single callback or factory*
  3. *Use a simple `case class` if you want to bundle multiple things together*
  4. *Use a `sealed trait` if you want to pass multiple different things*
  5. *Use an opaque `class` or `trait` as a last resort*
- **Error Handling**
  1. *If you know there is only one thing that can go wrong, use an Option*
  2. *If you know there is multiple things that can go wrong, use a Simple Sealed Trait*

    3. *If you don't know what can go wrong, use exceptions*
    4. *(Almost) Never set error flags*
- **Asynchronous Return Types**
    1. *The simplest case, return `T`*
    2. *For asynchronous results, return `Future[T]`*
    3. *Only pass in callbacks e.g. `onSuccess: T => Unit` as a last resort*
- **Dependency Injection**
    1. *First, hard-code the dependency*
    2. *Second, pass it as a parameter to the method which needs it*
    3. *Third, inject it into multiple methods by passing it into a containing class*
    4. *Fourth, if your containing class is split into traits in multiple files, use abstract members*
    5. *Fifth, make the method parameter implicit*
    6. *If all else fails, use "Dynamic Variables" aka global-mutable-state*
    7. *Don't use setter injection*

Philosophy: Principle of Least Power

In this context, this principle can be applied as follows:

> Given a choice of solutions, pick the least powerful solution capable of solving your problem

This is not immediately obvious. Developers put effort into trying to create powerful, flexible solutions. However, a powerful, flexible solution that can do anything is the most difficult to analyze, while a restricted solution that does a few things, and in fact *can only do a few things*, is straightforward for someone to inspect, analyze and manipulate later.

The origin of the rule is with regard to programming languages, i.e.

> Given a choice of languages, choose the least powerful language capable of solving your problem

And is justified by much the same reasoning I provided above.

Why is this principle applicable to programming in the Scala language? You could easily imagine the opposite principle applying:

> Given the choice of solutions, pick the most powerful solution capable of solving your problem

This would mean, for example, that solutions like meta-programming would be preferred over other more "basic" solutions. Not because meta-programming is advanced, but because it often allows tremendous flexibility to achieve absolutely anything in a very small amount of code. Why is that a bad thing?

Arguably, this is bad for a few reasons:

1. When programming in Scala, Complexity is your Enemy
2. Scala is statically typed, so you don't need to fear refactoring
3. Thus you don't need to over-engineer in anticipation of future work; just refactor when necessary

Complexity is your Enemy

The most common complaints from developers using Scala is that code is confusing and "hard to read" and "complicated", and that the compiler is slow. I'm not going to talk about compilation speed because it's often outside your control, but given the other complaint, making code "easier to read" and "less complicated" should be a priority to a developer working in the language.

This is not the case in every language! In Python or Ruby for example, people often call it "easy to read" or "executable pseudocode" referring to how the code looks exactly like you would imagine from sketchy it out on the whiteboard. The chief complaints center around and refactorability/maintainability and runtime performance. When programming in Python or Ruby an experienced programmer works to improve the refactorability of the code, e.g writing loads of unit tests to catch type errors, far more than you would in a statically-typed language like Java or Scala. That's not "better" or "worse" than writing fewer unit tests in Java or Scala, just "different" in order to accommodate the different constraints and problems the language presents you with.

Coming back to Scala, a developer should put in extra effort be making your code "easier to read" and "less complicated". That isn't the case in every language, but it is what you should do to in order to mitigate this weakness of the Scala programming language. Luckily, Scala provides other tools that help.

Don't Fear Refactoring

In dynamically-typed languages like Python and Ruby, and even other slightly-weaker-statically-typed languages like Java or C, trivial refactorings are often pretty difficult or scary. For example, renaming a field or method in a large Python codebase is difficult as you have no way to assure yourself you properly updated all callsites. Even in Java, widespread use of casting and reflection means you could easily add/remove/modify a class, compile everything successfully, and still have a `ClassCastException` or `InvocationTargetException` pop up at runtime. To combat this, you often program slightly pre-emptively: even in a small codebase, you will often pass arguments into a function you're not yet using, or pass in "more than you need", e.g. passing in the a whole object rather than just the single method/callback necessary. This is often subtle and subconscious, but the goal is usually to try and avoid the need for refactoring later: if you need to do more things, you can do so with minimal changes to the existing code.

In the Scala programming language, you should not fear refactoring. You have the compiler to guide you, from trivial changes like plumbing extra arguments into a function, to more involved restructurings of your codebase. There are still points of danger around things like `.toString` or `==`, but they're few enough that they become a "slightly annoying nuisances" rather than the "show-stopping hurdle" they present to many refactorings in dynamic languages.

Don't Over-Engineer

Thus, in Scala, you should avoid over-engineering things in anticipation of future need.

- If your function only needs a single method from an object, and aren't sure if it will need other things later, pass in that method rather than the whole object so you *can't* use other things.
- If you're not sure whether a one-use-helper method will need to be re-used in future, nest it inside the method that uses it so it *can't* be immediately re-used.

This may seem counter-intuitive, but it serves a purpose: by enforcing that all these currently-un-needed things *can't* take place, you are limiting the set of things that *can* be done with your code. By doing this we forcefully narrow the interface between different parts of your codebase as much as possible. This provides benefits:

- You get more freedom on both sides of the interface to evolve independently! If you pass a single method into a function, instead of the entire complex object, it becomes trivial to swap out the function later.
- You can better reason about what bits of your code are interacting with what other bits: if you pass a single method into a function rather than the whole object, you now can see *immediately* that that function only uses that one call, whereas previously you would need to dig through the sources to see where it was used.

In effect, by not pre-emptively over-engineering, we are trading off *ease-of-making-edits* to *ease-of-reading-and-understanding*. However, if we believe the earlier points that Scala's weakness is the difficulty in understanding complex code, and that it's strength is the ease of doing refactorings, this is a reasonable trade-of: we mitigate the problem that Scala is bad (complexity) at by leaning on what it's good at (refactoring). That is the principle under which the following guidelines arise.

Immutability & Mutability

When deciding between immutability and mutability...

1. *Use immutability as far as possible*
2. *Unless you are actually modeling mutable things*
3. *Or for performance*
4. *Even then, scope it as tightly as possible*
5. *Don't use double-mutability, you probably don't need it*
6. *Don't use Event-Sourcing/CQRS unless you know what you're doing*

Immutable things don't change, and things changing when they shouldn't is a common source of bugs. If you're not sure whether something will need to change later, leave it immutable as a `val` or `collection.Seq` and make the jump to `var` or `mutable.Buffer` later when necessary.

Immutability By Default

This is OK

```
val x = if(myBoolean) expr1 else expr
```

This is not OK:

```
var x: ExprType = null
if(myBoolean) x = expr1 else x = expr
```

If something can be straightforwardly expressed in an immutable style, do it.

Mutability For Mutable Things

In general, if you are actually modeling something which changes over time, using mutable `var`s or collections like `mutable.Buffer`s is fine. e.g. in a video game, you may have:

```
class Item{ ... }

class Player(var health: Int = 100, val items: mutable.Buffer[Item] = mutable.Buffer.empty)

val player = new Player()
```

Where we are actually modeling something whose `health` and `items` can change over time. This is fine. In theory, you could use fancy techniques like Event-Sourcing or CQRS to model this "immutably". In practice, modeling mutable things with mutable state is fine.

In contrast, this is not ok:

```
class Item{ ... }

class Player(var health: Int = 100, var items: mutable.Buffer[Item] = null)

val player = new Player()

player.items = mutable.Buffer.empty[Items]
```

Here, we are initializing the `items` variable to `null` and then initializing it again "properly" with an empty list sometime later. This is a very common pattern in Java and other languages, and is not ok: if you forget to mutate `player.items` before using it, or more-likely forget that some method you are using is using `player.items` before it gets set, it will blow up now or (worse) much later with a `NullPointerException`.

Here is a real example, taken from the Scala Parallel Collections library (http://docs.scala-lang.org/overviews/parallel-collections/configuration.html#task-support), which violates this principle:

```
scala> import scala.collection.parallel._
import scala.collection.parallel._

scala> val pc = mutable.ParArray(1, 2, 3)
pc: scala.collection.parallel.mutable.ParArray[Int] = ParArray(1, 2, 3)

scala> pc.tasksupport = new ForkJoinTaskSupport(new scala.concurrent.forkjoin.ForkJoinPool(2))

scala> pc map { _ + 1 }
res0: scala.collection.parallel.mutable.ParArray[Int] = ParArray(2, 3, 4)
```

As you can see, it uses the mutable `.tasksupport` attribute to configure how the parallel `map` operation is run. This is bad: it could easily have been passed in as a parameter to `map`, whether explicitly or as an implicit. As `tasksupport` does not model any actually-mutable value, using a mutable `var` just to initialize it is definitely bad style and who-ever wrote it should feel bad.

In general, if the thing you are modeling changes over time, it is ok to use mutability. If the thing you are modeling does not, and you are just using mutability as part of some initialization process, you should reconsider.

Mutability For Perf

This is Ok

```
val fibs = mutable.Buffer(1, 1)
while(fibs.length < 100){
  fibs.append(fibs(fibs.length-1) + fibs(fibs.length-2))
}
```

Very often, mutable code ends up being an order-of-magnitude faster than the immutable version. This is even more pronounced when implementing Algorithms, as the most common, fast algorithms in books like CLRS are done in a mutable fashion. If having a small amount of mutability can increase your performance ten- or hundred-fold, that can let you simplify the *rest* of your code by doing away with parallelism, caching, batching, and all sorts of knotty things. Don't be afraid to make that tradeoff.

Limit the Scope of Mutability

This:

```
def getFibs(n: Int): Seq[Int] = {
  val fibs = mutable.Buffer(1, 1)
  while(fibs.length < n){
    fibs.append(fibs(fibs.length-1) + fibs(fibs.length-2))
  }
  fibs
}
```

is better than

```

def getFibs(n: Int, fibs: mutable.Buffer[Int]): Unit = {
  fibs.clear()
  fibs.append(1)
  fibs.append(1)
  while(fibs.length < n){
    fibs.append(fibs(fibs.length-1) + fibs(fibs.length-2))
  }
}
```

Even if you've decided that you're going to introduce mutability in some part of your code, don't let it leak everywhere unnecessarily! Ideally it's all encapsulated within a single function, in which case to the outside world it looks identical to the same function implemented with immutable internals.

Note that sometimes you *do* need mutability to leak across function, class or module boundaries. For example, if you need performance, the second example above *is* faster than the first, and reduces allocations and thus garbage-collection pressure. However, default to the first example above unless you are 100% sure you need the perf.

No Double-Mutability

In Java, this is bad, but very common

```
ArrayList<Int> myList = new java.util.ArrayList<Int>()
```

In Scala, the equivalent bad code is

```
var myList = mutable.Buffer[Int]
```

However, you *almost never* need both the container to be mutable, as well as the variable holding the container to be mutable! The better Java code is

```
final ArrayList<Int> myList = new java.util.ArrayList<Int>()
```

In Scala:

```
val myList = mutable.Buffer[Int]
```

Or

```
var myList = Vector[Int]
```

Whether you want it to be a mutable `var` holding an immutable collection or a immutable `val` holding a mutable collection is debatable, but you basically never want it to be doubly-mutable

Event-Sourcing/CQRS

Often, you'll hear people talking about techniques that make mutable state-changing code "pure" or "immutable", by instead storing a mostly-immutable, append-only log of events. This has many of the benefits of immutability in that even as new events change things, the old events are still there, and you can query the "state" of the system in any point in time by re-playing the events up to that point. This is something you cannot do when using mutable `var` s or collections like `mutable.Buffer` , and the technique is used to great effect in many places e.g.

- Database, where the transaction logs allow "streaming" replication, and isolation between transactions.
- Video-games, where storing the input-log allows you to re-play everything that happened during a session e.g. for future viewing

These techniques can both be used in-memory, or with persisted to disk, or even with a full database/datastore holding the append-only log. A full explanation on how these techniques work is beyond the scope of this document.

In general, if you want the benefits that these techniques bring: re-playability, isolation, streaming replication, then by all means use these techniques. In general, though, it is probably an over-kill for most use cases and should not be used by default unless you know you want those benefits.

Published Interfaces

When defining an interface to a package someone else will use...

1. *The simplest interface to a package is static method with standard types*
2. *Next, static methods that take/return custom types, presumably with methods*
3. *Next, is needing the user to instantiate some classes*
4. *Lastly, is needing the user to inherit from some classes*

I use the term "Published Interfaces" as distinct from a "normal" Java `interface` or Scala `trait` . A published interface is a larger-scale interface between moderately large sections of a program, comprising of multiple `class` es, `object` s: the interface presented to a developer by an entire `package` or `.jar` .

Static Methods Only

Below is the best kind of published interface you can ask for

```
object TheirCode{
  /**
   * Takes a set of `T`s and a function which defines what other `T`s`
   * can be reached from the outgoing edges from each `T``
   *
   * Returns the set of strongly connected components (each component
   * being a Set[T])
   *
   * O(N + E) in the number of nodes N and total number of edges E
   */
  def stronglyConnectedComponents[T](nodes: Set[T],
                                     edges: T => Set[T]): Set[Set[T]] = {
    ... 500 lines of crazy code ...
  }
}

object MyCode{
  import TheirCode._
  stronglyConnectedComponents(..., ...)
}
```

It encapsulates a non-trivial algorithm, that uses tons of mutable state internally, and you probably would not be able to come up with yourself.

Maybe it's using Tarjan's algorithm? Maybe it's using the Double-Stack algorithm? To the consumer of this interface, you don't care: you know you can feed in a `Set[T]` and a `T => Set[T]` of outgoing edges, and it will spit out the `Set[Set[T]]` of all the strongly connected components. You see it has 500 lines of crazy algorithmic code, but you don't need to care about any of that. The 2-line signature, and 5-line doc-comment, are all you need to know as far as using this in your own code is concerned.

Of course, you can't *always* present a super-simple single-static-function only-dealing-with-known-types interface to your code: your code may simply be doing more than one-thing, and may need to be configured in more ways than can be stuffed into a single function's arguments. This sort of dead-simple "interface with only static-function dealing with known-types" is something to aspire to.

Instantiating Custom Types

Below is a not-as-good interface

```
object TheirCode{
  trait Frag{
    def render: String
  }
  // HTML constructors
  def div(children: Frag*): Frag
  def p(children: Frag*): Frag
  def h1(children: Frag*): Frag
  ...
  implicit def stringFrag(s: String): Frag
}

object MyCode{
  import TheirCode._
  val frag = div(
    h1("Hello World"),
    p("I am a paragraph")
  )
  frag.render // <div><h1>Hello World</h1><p>I am a paragraph</p></div>
}
```

Here, the interface of `TheirCode` starts making demands of the developer: you can't just call "one function" and get what you want, now you need to learn what a `Frag` is, which of the static methods `TheirCode` exposes returns `Frags` , and how to convert your own stuff into `Frag` s via the implicit conversion. Lastly, you also have to know what *you* can do with a `Frag` : in this case you can call `render` to turn it into a `String` . And *only then* can a developer do useful things with the library! After all, a developer trying to use your library isn't thinking

> I want to learn how to construct `Frag` s and operate on them

They're thinking

> I want to generate HTML strings and put some of my strings in there.

Of course, a developer will *have* to learn how to do all this `Frag` stuff before they can do what they want, but the less they need to learn the better.

This example interface isn't terribly complex, but it is certainly more complex than the previous `stronglyConnectedComponents` example! This also isn't all-or-nothing: you can introduce more- or less- custom types and constructors for your users to learn, and it gets correspondingly harding for an outsider to figure out how to use your code.

Again, it is not always possible to make your interface simpler than this, and I have published lots of code that uses this style of interface. Nevertheless, it is better than needing to inherit from classes to get the job done...

Inheriting From Classes

Forcing users of your API to inherit from classes or traits should be the last resort. That's not to say that APIs designed around inheritence are un-usable: they've been used in the Java world for ages. Nevertheless, if you have a choice between exposing a few static functions, exposing some classes/types the developer will have to work with, and forcing the developer to inherit from your classes/traits, inheritence should be last on the list of options and only used as a last resort.

Data Types

When picking what sort of type to use for some value...

1. *Use built-in primitives, collection & combinations of them when possible*
2. *Use opaque functions when you just need a single callback or factory*
3. *Use a simple `case class` if you want to bundle multiple things together*
4. *Use a `sealed trait` if you want to pass multiple different things*
5. *Use an opaque `class` or `trait` as a last resort*

This applies whether that value is a method parameter, class parameter, field, or method return type. In general, using the "simplest" thing for each use case means that whoever is looking at the code later can make stronger assumptions about how that value.

If I see a primitive or built-in collection, I know exactly it contains. If I see an opaque function, I know the only thing that can be done is to call it. If I see a simple `case class` , I can be relatively confident it's a dumb struct. If it's a `sealed trait` , it could be one of multiple dumb-structs. If it's a custom hand-rolled `class` or `trait` , all bets are off: it could be anything!

By starting from the simplest types and making your way down the list in increasing-power only when necessary, you are respecting the principle of least power and sending signals to an API user about how your value is going to be used.

Built-ins

Where possible, you should always use built-in primitives and collections. While they are all flawed in various ways, they are well known and "boring", and someone looking at how to use your interface knows they're just interacting through already-known data-types. Standard `Int` s, `String` s, `Seq` s and `Option` s and combinations should be used instead of your own custom versions of the same concepts.

If you find yourself passing an object to a method that only accesses a single field on that object, consider just passing that field directly. e.g. this:

```
class Foo(val x: Int, val s: String, val d: Double){
  ... more things ...
}

def handle(foo: Foo) = {
  ... foo.x ... // Only one usage of foo
}

val foo = new Foo(123, "hellol", 1.23)

handle(foo)
```

May be better re-written as

```
class Foo(val x: Int, val s: String, val d: Double)

def handle(x: Int) = {
  ... x ... // Only one usage of x
}

val foo = new Foo(123, "hellol", 1.23)

handle(foo.x)
```

This makes it clear that the `handle` doesn't *actually* need the entirety of `Foo`, which contains both `x: Int` and `s: String` and possibly other things. It only needs the single integer `x`. Furthermore, if we want to re-use `handle` in other parts of our codebase or exercise it in our unit tests, we won't need to go through the trouble of instantiating an entire `Foo` just to pass in a single `Int` and throw the rest away.

Similarly, if you are returning an object of whom only a single field is used, just return that field. If you need the rest of the object later, you can then refactor to make it available.

Functions

More complex than built-ins is the opaque function; whether it's `Function0[R]` / `() => R`, `Function1[T1, R]` / `T1 => R`, or one of the higher numbered `FunctionN` s. The only thing you can do with these types is call them with arguments to get their return value.

Taking and returning functions can substitute for many cases where in Java-land you would be taking and returning single-abstract-method interfaces: e.g. Runnable (https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html), Comparator[T] (https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) can be substituted by a `() => Unit` and `(T, T) => Int` in Scala-land.

In general, if you find yourself passing an object to a method that only calls a single method, consider making that method take a `FunctionN` and passing in a lambda. e.g. this

```
class Exponentiator(val x: Int){
   ... some methods ...
  def exponentiate(d: Double): Double = math.pow(d, x)
   ... more methods ...
}

def handle(exp: Exponentiator) = {
  val myDouble = ...
  ... exp.exponentiate(myDouble) ... // Only one usage of foo
}

val myExp = new Exponentiator(2)

handle(myExp)
```

May be better re-written as

```
class Exponentiator(val x: Int){
   ... some methods ...
  def exponentiate(d: Double): Double = math.pow(d, x)
   ... more methods ...
}

def handle(op: Double => Double) = {
  val myDouble = ...
  ... op(myDouble) ... // Only one usage of foo
}

val exp = new Exponentiator(2)

handle(exp.exponentiate)
```

Again, this transformation makes it more obvious which parts of `exp` the `handle` function is using, making it easier to use `handle` elsewhere, or stub out the logic in a unit test.

Simple Case Classes

Also more complex than built-ins, but in a different way than Functions, is case classes. These let you bundle multiple things together ad-hoc and assign names to the individual components. The advantage of doing it in a "simple" `case class` rather than in your own ad-hoc class is that everyone who sees your `case class` knows immediately what to expect: constructor, accessors, pattern-matching, hashCode, toString, equality, all nice things to have when dealing with dumb-struct-style objects.

As an example, if you find yourself writing code that looks like this

```
class Foo(_x: Int, _y: Int){
  def x = _x
  def y = _y
  def hypotenus = math.sqrt(x * x + y * y)
  override def hashCode = x + y // ad-hoc custom hash function
  override def equals(other: Any) = other match{
    case f: Foo => f.x == x && f.y == y
    case _ => false
  }
}
```

Consider instead writing

```
case class Foo(x: Int, y: Int){
  def hypotenus = math.sqrt(x * x + y * y)
}
```

This serves multiple purposes:

- Saves on verbosity
- Reduces the number of places we can go wrong (e.g. in our custom ad-hoc hash function above)
- Makes our dumb-struct behave like everyone else's dumb-structs

In general, there is no bright line between dumb-struct classes and more involved, ad-hoc classes: dumb-structs can and often do contain small amounts of logic (e.g. the `hypotenus` method above), and ad-hoc classes can and do contain small amounts of "dumb" data. Nevertheless, if the main purpose of a class is as a "dumb" data-structure rather than a repository for code, consider making it a `case class` .

Sealed Traits

If you need to pass one of several things, and each of those things is a "dumb-struct", consider using a `sealed trait` . This tells the future maintainer that the set of things that can live in this type is finite and fixed: they never need to worry about "new" classes coming along and inheriting from that interface/trait and needing to be able to handle it.

Ad-hoc Classes

These are the most powerful of all the types that can be ascribed to a value. When looking at built-ins, functions or `case classes` , a developer using your code knows at a glance roughly what to expect. When looking at an ad-hoc `class` or `trait` , they don't know anything at all about how the class will be used, and will have to actually read the documentation or dig through the source code.

Furthermore, the fact that ad-hoc classes are so ad-hoc means you lose concrete features that simpler data-types provide. For example:

- Built-ins and primitives are all trivially serializable, without reflection, using implicits (e.g. using spray-json (https://github.com/spray/spray-json))
- Case classes and sealed-traits are all almost-trivially serializable, without reflection, using a few macros (e.g. using spray-json-shapeless (https://github.com/fommil/spray-json-shapeless) or uPickle (http://lihaoyi.github.io/upickle-pprint/upickle/))
- Built-ins, primitives, and case-classes all provide automatic, structural `==` equality, `.hashCode` , and a meaningful `.toString` .
- Built-ins and case-classes all allow destructuring pattern matching, which is often very convenient

These are all things you lose when you start using ad-hoc classes. In exchange for the flexibility they offer, you lose a whole lot of features!

This isn't to say ad-hoc classes or traits are *bad*; in fact, most Scala programs have tons of them, and Java programs are basically 100% ad-hoc classes! What this means is that, where possible, you should try to avoid using an ad-hoc class in favor of something simpler: a Simple Case Class, Sealed Traits, Functions or Built-ins. Again, this applies anywhere a type an be seen: as a method argument, as a return type, as the type of a variable or value, whether local-to-a-method or belonging to an object or class.

Error Handling

1. *If you know there is only one thing that can go wrong, use an Option*
2. *If you know there is multiple things that can go wrong, use a Simple Sealed Trait*
3. *If you don't know what can go wrong, use exceptions*
4. *(Almost) Never set error flags*

In general, `Option` s, simple sealed traits, and exceptions are the most common way of dealing with errors. It is likely that different APIs you end up using will be using a mish-mash of these techniques. In general, it makes sense to wrap these mixed errors as soon as possible to put them in some consistent format: e.g. if you decide to use a sealed trait, wrap all known exceptions in a `try` - `catch` and convert the `Some` / `None` of `Option` returning functions into the relevant subclasses of your sealed trait. If you decide to use exceptions, use `Option.getOrElse(throw new MyCustomException)` to convert any `Option` s into better-named exceptions.

Option

This is the simplest possible mechanism for returning errors from a function: you either get a `Some` containing the result, or you get a `None` containing nothing.

This works great in simple cases, for example `Map[K, V]#get(k: K)` returns `Option[T]` because there's really only one thing that could have gone wrong: the key doesn't exist. There's nothing to say about *why* the key doesn't exist, and there's no other thing that could have gone wrong. This is the ideal use case for an `Option` . You can also us `for` -comprehensions or `map` / `flatMap` to chain them and easily propagate failures.

The downside of using `Option` s is that it doesn't give you any place to put the more information about the error. If you need to distinguish between multiple failures, e.g. to display error messages to a user, you need something more powerful. You could return `Option[Option[T]]` or even `Option[Option[Option[T]]]` to deal with a small fixed set of errors, but it probably makes more sense to define your own custom sealed trait to use as a return value.

Simple Sealed Trait

When `Option[T]` isn't flexible enough, you may fall back to defining a custom sealed trait to represent errors. For example:

```
sealed trait Result
object Result{
  case class Success(value: String) extends Result
  case class Failure(msg: String) extends Result
  case class Error(msg: String) extends Result
}

def functionMayFail: Result
```

This way, who-ever calls `functionMayFail` will see that it has three possible cases: `Success` , `Failure` and `Error` , each of which has their own results associated with it. You can define your own `map` / `flatMap` so you can use it in `for` -comprehensions to propagate the various failure states.

Exceptions

Exceptions are controversial, but they have their place for things that can "almost never" go wrong, but still do: `/ by zero` , `ClassNotFound` , `NoSuchFileException` , `StackOverflowError` , `OutOfMemoryError` keyboard interruption, etc..

In general, it is often unfeasible to make everything that could fail with one of these errors return an Option or some kind of ADT: every expression in your codebase can potentially `StackOverflow` or `OutOfMemory` or `ClassNotFound` ! Much arithmetic is choke full of `/` calls, and it is unreasonable (both from the boilerplate and from the performance-cost) to wrap every one of those in an `Option` . So what happens when something goes wrong?

You could decide that you want the program to halt if any of these cases appear, and try to debug it. And that's exactly what exceptions do if you don't catch them: they propagate up the stack and exit your program with a message and stack trace to help debug what went wrong. Perfect!

You could also decide you want to run some code after the problem occurs, e.g. to report the error for later debugging somewhere else, and to perhaps keep the process going since despite the unexpected failure the "next" task it's doing should be unaffected. And that's where you catch exceptions and proceed!

For example, if you are writing a large and complex sub-system that accomplishes one task, it is often reasonable to assume that

1. There will be bugs that causes a whole range of "impossible" errors to occur
2. When they occur, you want to report them somehow with diagnostics
3. You want the rest of the program to continue anyway despite these bugs

This is the case where you could consider placing a large `try` - `catch` around calls to that subsystem, logging the exception, and moving on. In theory you *could* place fine-grained `try` - `catch` es around every individual possibly-failure point and convert them all to `Option` s or your own `sealed trait` , and you can log-the-error-and-carry-on in the same way, but all that really does is convolute your code considerably for (in this case) no benefit.

Error Flags

Error flags are a convention which goes something like

- If some succeeds, it does something
- If it fails, it does nothing and sets a special variable e.g.

```
var error = -1 // no error

def doThing() = {
  ...
  if (didntWork) error = 5 //
  ...
}

doThing()
if (error == 5) println("It failed =/")
```

This has a few problems:

- It is much less safe than the above cases of using Option, ADTs or Exceptions: if you forget to check the error flag, the program keeps running, possibly doing the wrong thing!
- It is *incredibly* easy to forget to check error flags. They aren't shown in type signatures, and you have to remember that for each method there's a special mutable variable you have to check each time.

Overall, you should avoid them as much as is possible.

Sometimes it is not possible: error flags are probably the fastest way of transmitting the "something failed" information, and it could matter in hot code paths. Like using Mutability For Perf, it is reasonable to *sometimes* drop down to using an error flag to eek out the last 5% of performance after you've profiled the code and identified the bottleneck as error-handling.

Even when you do this, make sure to encapsulate the error flag as tightly as possible, keeping it local to the internals of the class or method or package, and documenting it like crazy as the dangerous performance-hack that it is.

Asynchronous Return Types

1. *The simplest case, return `T`*
2. *For asynchronous results, return `Future[T]`*
3. *Only pass in callbacks e.g. `onSuccess: T => Unit` as a last resort*

If you can get away without any asynchrony, you should. Otherwise, `Future[T]` is the first preferred asynchronous return type. Only use callback functions to "return" asynchronous results when `Future`s don't work (e.g. you want to call it more than once).

Return T

```
def foo(): T
```

This is the simplest case, and is almost not worth discussing. If you can get away without asynchrny, do so: your code will run faster, stack-traces will be more useful, debuggers will work better. For all the hype about non-blocking asynchronous stuff, plain-old-synchronous-code works better in cases where you can get away with it.

Return Future

```
def foo(): Future[T]
```

The next alternative is returning a `Future[T]` that represents an asynchronous result. While other languages e.g. Java or Javascript, and low-level APIs may often use callbacks `Future`s come first in the principle-of-least-power scale because they are strictly weaker than callbacks: unlike callbacks, they can only provide a single result.

If you follow this scale, someone who receives a `Future` from your code knows it can only fire once. What's more, someone who passes an asynchronous callback into your code *knows* it will fire multiple times because otherwise you would have given a `Future` instead!

`Future`s provide lots of benefits over callbacks. For example, consecutive `Future`s can be "flattened out" in a way that callbacks cannot. `Future`s also propagate errors by default, where-as in callback-driven APIs it is very easy to accidentally ignore them.

Take a callback: T => Unit

```
def foo(onSuccess: T => Unit): Unit
```

Callbacks are the oldest and most "raw" method for providing asynchronous results. You call a function, and when it's done it calls your passed-in-function with the result you can then do things with.

In general, they should be avoided in favor of `Future`s whenever possible. There are cases where `Future`s don't work and callbacks are necessary, e.g. when they fire multiple times, but when `Future`s work use a `Future`.

Dependency Injection

1. *First, hard-code the dependency*
2. *Second, pass it as a parameter to the method which needs it*
3. *Third, inject it into multiple methods by passing it into a containing class*
4. *Fourth, if your containing class is split into traits in multiple files, use abstract members*
5. *Fifth, make the method parameter implicit*
6. *If all else fails, use "Dynamic Variables" aka global-mutable-state*
7. *Don't use setter injection*

Hardcode It

```
def sayHello(msg: String) = println("Hello World: " + msg)

sayHello("Mooo")
```

This is the simplest case, and is very common. If you can get away with hard-coding the dependency (e.g. to `println`), just do it. If you need additional flexibility to swap it out, refactor later.

Method Parameter

```
def sayHello(msg: String, log: String => Unit) = log("Hello World: " + msg)

sayHello("Mooo", System.out.println)
sayHello("ErrorMooo", System.err.println)
```

As a next step, if you need to swap out `println`, pass it in as a method parameter. Here we see we can call it with either `System.out.println` or `System.err.println`, or with our remote-logger, or test-logger to inspect results or other things.

Constructor Injection

If you find yourself passing the same logger to lots of different functions:

```
def func(log: String => Unit) = ... func2(log) ...
def func2(log: String => Unit) = ... func3(log) ...
def func3(log: String => Unit) = ... func4(log) ...
def func4(log: String => Unit) = ... func5(log) ...
def func5(log: String => Unit) = ... func6(log) ...
def func6(log: String => Unit) = ... func7(log) ...
def func7(log: String => Unit) = ... func8(log) ...
def func8(log: String => Unit) = ... func9(log) ...
def func9(log: String => Unit) = ... sayHello("Moo", log) ...

def sayHello(msg: String, log: String => Unit) = log("Hello World: " + msg)

func(System.out.println)
func(System.err.println)
```

You can wrap those functions into a class and inject it into the class instead:

```
class Container(log: String => Unit){
  def func() = ... func2() ...
  def func2() = ... func3() ...
  def func3() = ... func4() ...
  def func4() = ... func5() ...
  def func5() = ... func6() ...
  def func6() = ... func7() ...
  def func7() = ... func8() ...
  def func8() = ... func9() ...
  def func9() = ... sayHello("Moo") ...

  def sayHello(msg: String) = log("Hello World: " + msg)
}

new Container(System.out.println).func()
new Container(System.err.println).func()
```

The same technique applies if you need the same logger injected into different `class` es or `object` s: simply nest those `class` es or `object` s inside (in this case) the `Container` class and they'll all get access to it without needing to add `log` into each-and-every method signature.

Abstract Members

If you're trying to use Constructor Injection but your `Container` class is getting too large, split it into separate files as individual `trait` s.

```
// Foo.scala
trait Foo{
  def log: String => Unit
  def func() = ... func2(log) ...
  def func2() = ... func3(log) ...
  def func3() = ... func4(log) ...
}

// Bar.scala
trait Bar extends Foo{
  def log: String => Unit
  def func4() = ... func5(log) ...
  def func5() = ... func6(log) ...
  def func6() = ... func7(log) ...
}

// Baz.scala
trait Baz extends Bar{
  def log: String => Unit
  def func7() = ... func8(log) ...
  def func9() = ... sayHello("Moo", log) ...
  def sayHello(msg: String) = log("Hello World: " + msg)
}

// Main.scala
class Container(val log: String => Unit) extends Foo with Bar with Baz
new Container(System.out.println).func()
new Container(System.err.println).func()
```

In doing so, you will need to pass `log` into each individual `trait` . `trait` s can't take constructor parameters, but you can do it by giving them an abstract method `log` and having the main `Container` class extend all these `trait` s and implement `log` concretely.

Implicit Method Parameter

If you find yourself passing the same logger everywhere:

```
def sayHello(msg: String, log: String => Unit) = log("Hello World: " + msg)

sayHello("Mooo", System.out.println)
sayHello("Mooo2", System.out.println)
sayHello("Mooo3", System.out.println)
sayHello("Mooo4", System.out.println)
sayHello("Mooo5", System.out.println)
sayHello("Mooo6", System.out.println)
sayHello("Mooo7", System.out.println)
sayHello("Mooo8", System.out.println)
sayHello("Mooo9", System.out.println)
```

And the use-sites are "all over the place", not limited to one (or a small number of) classes, then make it an implicit parameter

```
def sayHello(msg: String)(implicit log: String => Unit) = log("Hello World: " + msg)

implicit val logger: String => Unit = System.out.println
sayHello("Mooo")
sayHello("Mooo2")
sayHello("Mooo3")
sayHello("Mooo4")
sayHello("Mooo5")
sayHello("Mooo6")
sayHello("Mooo7")
sayHello("Mooo8")
sayHello("Mooo9")
```

In general, you should only do this if you are passing it around a *lot*: into dozens of callsites at least. Nevertheless, in a large program that's not unreasonable, especially for common things like logging.

However, if the use sites are all relatively localized, you should prefer to use Constructor Injection rather than creating a new implicit parameter just for one small section of your code. Reserve implicit parameters for the cases where the callsites are scattered and constructor injection into all the disparate classes becomes tedious.

"Dynamic Variables" aka Global Mutable State

This is a common pattern, where some global/thread-local variable is set before running your code, and your code reads from it as a global. It is used in almost every web framework out there: from Scala frameworks like Lift, to Python frameworks like Flask or Django. It looks like this:

```
var log: String => Unit = null
def func() = ... func2(log) ...
def func2() = ... func3(log) ...
def func3() = ... func4(log) ...
def func4() = ... func5(log) ...
def func5() = ... func6(log) ...
def func6() = ... func7(log) ...
def func7() = ... func8(log) ...
def func8() = ... func9(log) ...
def func9() = ... sayHello("Moo", log) ...

def sayHello(msg: String) = log("Hello World: " + msg)

log = System.out.println
func() // Logs to stdout
log = System.err.println
func() // Logs to stderr
```

It has a lot of convenience: you don't need to wrap everything into a `class`, you don't need to add an `implicit` into all your method signatures or pass the `log` function around manually.

On the other hand, it is the most dangerous: it is no longer obvious which parts of the code have a `log` function "available" and which parts don't. It is easy to make a mistake like calling a method using `log` "too early", before it has been initialized, and having it compile perfectly well but blow up at runtime:

```
var log: String => Unit = null
def func() = ... func2(log) ...
def func2() = ... func3(log) ...
def func3() = ... func4(log) ...
def func4() = ... func5(log) ...
def func5() = ... func6(log) ...
def func6() = ... func7(log) ...
def func7() = ... func8(log) ...
def func8() = ... func9(log) ...
def func9() = ... sayHello("Moo", log) ...

def sayHello(msg: String) = log("Hello World: " + msg)

func() // KA-BOOM
log = System.out.println
func() // Logs to stdout
log = System.err.println
func() // Logs to stderr
```

This is a problem that does not exist in the dependency-injection techniques suggested earlier/above: hard-coding it, using method-parameters (explicit or implicit), constructor injection, abstract-member injection, in all these cases the compiler knows statically which methods have a `log` function available and which don't, and will helpfully provide a compile-error if you attempt to use `log` where it's not available. With Dynamic Variables, you get no such assurance.

Thus, while this gives you the most power, it also gives you the least safety and should be avoided if possible.

Setter Injection

"Setter Injection" refers to instantiating an object, and then setting some variable onto that object which it will use when you call methods on it.

As described in the section on Immutability & Mutability, don't do that: mutable state should only be used to model mutable things, and not as an ad-hoc way of passing parameters to a function call.

About the Author: *Haoyi is a software engineer, an early contributor to Scala.js (http://www.scala-js.org/), and the author of many open-source Scala tools such as the Ammonite REPL (lihaoyi.com/Ammonite) and FastParse (https://github.com/lihaoyi/fastparse).*

If you've enjoyed this blog, or enjoyed using Haoyi's other open source libraries, please chip in (or get your Company to chip in!) via Patreon (https://www.patreon.com/lihaoyi) so he can continue his open-source work

← Code Reviewing My Earliest Surviving Program (CodeReviewingMyEarliestSurvivingProgram.html)Talks I've Given → (TalksIveGiven.html)

Updated 2016-02-19 (https://github.com/lihaoyi/blog/commit/0ee2646783a067226df962661b391d0fa3818174)2016-02-14 (https://github.com/lihaoyi/blog/commit/6e92e9bb4cd6f68a4f214f0e1cf830165ae7ab5b)2016-02-14 (https://github.com/lihaoyi/blog/commit/325f74286854a461a2ac432eae12bb0f5e5431f6)2016-02-14 (https://github.com/lihaoyi/blog/commit/9eef8204f025f23a48da9563f236257045f82bc6)2016-02-14 (https://github.com/lihaoyi/blog/commit/2c8bbc0d1ce4f10f746a7d3c46925b050ea2fc45)2016-02-14 (https://github.com/lihaoyi/blog/commit/00f45a23193e4545471a1ce4d3737ee67f657e79)

57 Comments **lihaoyi.com** 1 **Login**

♡ **Recommend** 28 Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

martin odersky • 3 years ago

I very much enjoyed reading the blog post. The premise you are starting from, i.e. "Principle of Least Power" makes a lot of sense and you draw well reasoned conclusions from it. It would be fun to apply the same principle to more areas!

42 ^ | v • Reply • Share ›

Ken Scambler • 3 years ago

This is the best Scala style guide I've read yet! Great job Haoyi.

However, since this is the internet, and it's boring if we agree:

I think the biggest nit for me is primitives and strings; while I generally agree that using in-built types is preferable to custom ones (barring special requirements), there's often a good case to avoid them on the "principle of least power".

Do we really have an int? Well, can we do arithmetic on it? Perhaps yes, for "numProperties", but no for "customerId". Do we really have a string? Well, would the program make sense if we put the complete works of Shakespeare in there? Perhaps yes for "userComment", but no for "emailAddress".

Where the primitive type:

where the primitive type.
- Hides more structure that must be rediscovered
- Cannot be generally treated as that type without special information
- Is not interchangeable with other values of the type (ie "userId" vs "productId")

then using a primitive makes the program less safe, allows more incorrect programs to exist, and gives the coder less information to reason with. These directly relate to the goals of your article!

Of course, if a number is just a number (ie coordinates, counts, etc), then it makes sense to just use the obvious type. So I don't think I disagree with your point as such; but I do think it needs heavy qualification.

10 ^ | ⌄ • Reply • Share ›

Kyle Willett ➜ Ken Scambler • 3 years ago

I agree, I've been using case classes with one primitive type property. Maybe its overkill but I like the extra information. Its sort of a marker interface for a primitive. I'd be interested to hear the pros/cons regarding this.

1 ^ | ⌄ • Reply • Share ›

codingismy11to7 ➜ Kyle Willett • 3 years ago

yeah, and I've read other people espousing the same ideal. Sometimes when you just have a String or Int as (say) an argument, it's way more useful for refactoring and clarity if the argument is a LastName(s: String) or VmMemory(memInMb: Int)

2 ^ | ⌄ • Reply • Share ›

David Gregory ➜ Kyle Willett • 3 years ago

I hope you are aware that you can mitigate the cost of this to some degree by extending AnyVal; see http://docs.scala-lang.org/...

1 ^ | ⌄ • Reply • Share ›

Kyle Willett ➜ David Gregory • 3 years ago

Cool, thanks for sharing. Did not know about value classes.

^ | ⌄ • Reply • Share ›

Siddhartha Gadgil ➜ Kyle Willett • 3 years ago

I had an experience of value classes reducing computation time from 20 minutes to 30 seconds.

1 ^ | ⌄ • Reply • Share ›

Enpassant ➜ Ken Scambler • 9 months ago

I agree with you.
1. "*This is the best Scala style guide I've read yet! Great job Haoyi!*"
2. Built-in primitives are more powerful than custom types.

Scott Wlaschin gave a very interesting talk about "Designing with capabilities" (https://vimeo.com/162209391), which is based on the same principle, principle of least power (privilege or authority).
He also has many interesting posts in the "Designing with types" series (https://fsharpforfunandprof....

^ | ⌄ • Reply • Share ›

Heiko Seeberger • 3 years ago

Great stuff! Thanks for putting it together.

One thing: `collection.Seq` is **not** immutable. See https://hseeberger.wordpres... and https://issues.scala-lang.o....

4 ^ | ⌄ • Reply • Share ›

codingismy11to7 ➜ Heiko Seeberger • 3 years ago

haha, that was the biggest thing that bugged me - mostly because I just learned that recently and now have a tic about importing immutable.Seq and using toVector instead of toSeq

1 ^ | ⌄ • Reply • Share ›

Sébastien Doeraene • 3 years ago

This is a very well written post, based on meaningful premises, and good reasoning. I definitely agree with the premises, and consequently I agree with most of what is written here.

However, there is an area where I believe the reasoning was flawed, possibly due to confirmation bias: case classes vs opaque classes. Of course my counter-argument could very well be confirmation-biased as well ... we'll see.

You argue throughout for the principle of least power. But when it comes to case classes, you justify their benefits with arguments like:

> everyone who sees your case class knows immediately what to expect: constructor, accessors, pattern-matching, hashCode, toString, equality, **all nice things to have** when dealing with dumb-struct-style objects.

(emphasis mine). This seems in direct contradiction with the principle of least power: a case class gives more power than usually necessary. And you argue for that additional power as "nice to have". If I do not intend to use a class in pattern matching--in a `case`--, chances are I don't *need* the power of a `case` class, and therefore I should not use that.

An opaque class with an `apply` method, a private constructor, and public `val` fields is about the least powerful custom

see more

4 ^ | v • Reply • Share ›

Li Haoyi Mod ➜ Sébastien Doeraene • 3 years ago

I guess the difference of opinion here is you mean *power* to mean *convenience*, while I use *power* to mean *flexibility*. IMO case classes are a case of "more convenient, less flexible", which is probably the ideal situation to be in! You want to write as little code as possible, while simultaneously restricting your code to as narrow a behavior as possible.

Sure the lack-of-flexibility is based more on Convention rather than Compiler Enforcement, as is their final-ness. But in reality, nobody except Martin Odersky ever extends their `case class`es... or at least I haven't found anyone doing it in all the code I've come across. Similarly, you can have case classes that have all the same `var`s and other rubbish that lives in a normal `class`, but the convention is not to do that. I guess that's what I meant by *simple* `case class`es.

This is more a convention-based reasoning than compiler-enforced-reasoning, and obviously anything in the standard library/collections doesn't apply it (that code is crazy) but third-party code I've found follows these conventions pretty closely. When I see a `case class` i have a pretty good sense of what they're not using it for: no inheritance-based API, no hidden private state, no real encapsulation, probably not much *code* in method-bodies and initializers.

----

see more

7 ^ | v • Reply • Share ›

ches ➜ Sébastien Doeraene • 2 years ago

Yeah, particularly with the early emphasis on "don't fear refactoring" my expectation was set for Haoyi's entire post to take a perspective of user/application code, not library maintainers—and I think there's an undertone of advocating particularly to that audience that Scala doesn't have to seem so complex. Don't scare them off with a whole new realm of considerations to take into account! ;-)

Not that you're wrong of course, and I suppose there are large enterprise code bases out there, with subsystems maintained by multiple teams, where they do begin to feel some costs of binary incompatibility and they grow to have a more careful attitude about it by necessity. Your insights are certainly helpful to any in that position or getting into widely depended-upon open source projects. Hopefully an open source library maintainer has accepted and embraced this new facet of complexity that they have to preside over when enlisting for duty :-)

Case classes are probably one of the clearest points of departure on these grounds: library maintainers think, "run away!", and application developers think, "this is one of my favorite features of Scala"!

^ | v • Reply • Share ›

Christophe Calvès • 3 years ago

Very interesting! I could not agree more on "Complexity is your Enemy". One question is: which notion of complexity are you considering? Ruby, Python and others seem easy to read because the background they require is well known by many people. It's not fair to say that Scala is more complex to read. From a functional-programming background, Scala code is easy but Python/Ruby/Java/C code may not be.

Another and more objective notion of complexity is, as developed in the paper "Out of the tar pit" by Mosely and Marks, the ability to reason (at least informally) about the code. Python/Ruby/Java/C/... reach simplicity by sweeping lots of issues under the carpet: uncontrolled mutation, type unsafety, ... Indeed, parachute jumping is much easier without a parachute but you end up much more dead too. Scala makes reasoning about the code a lot easier than usual languages do. Functional programming, traits, algebraic data types, pattern matching, type-classes, implicits, Types and kind polymorphism, ... are wonderful features that once used to, produce safer, cleaner and more concise code. The only "problem" is that this knowledge is still not very widespread.

3 ^ | v • Reply • Share ›

Josep Prat • 3 years ago

Awesome post! I think such posts are really needed in the Scala community so more and more companies adopt Scala as main language.

I have one point to add under the "Error Handling" section. What do you think about adding the use of the scala.util.Try[T] object as an alternative? As you say, the Option[T] type only gives you the possibility to express 2 concepts: value or absence of it, whereas with Try[T] you can express the concept of having a value and at the same time handle the failure depending on the exception thrown.

Looking forward to hear your opinion.

3 ^ | v • Reply • Share ›

Kyle Willett ➜ Josep Prat • 3 years ago

The Error Handling section seems to be about how to model errors. Since a Try encapsulates a computation that may succeed or cause an exception, it would be used with option #3 (but maybe its worth mentioning under that option).

1 ^ | v • Reply • Share ›

Li Haoyi Mod ➜ Kyle Willett • 3 years ago

Yeah I missed out on `Try`. It's basically isomorphic to an Exception, and you can convert back and forth pretty straightforwardly.

6 ⌃ | ⌄ • Reply • Share ›

Siddhartha Gadgil ➜ Li Haoyi • 3 years ago

One nice thing about Try is one can just wrap code as Try(....) which is to me far easier to follow than try-catch

⌃ | ⌄ • Reply • Share ›

OlegYch • 3 years ago

agree with most of this
however " modeling mutable things with mutable state is fine." does not sound like a good advice
most things that seem "mutable" can be better modeled without vars

3 ⌃ | ⌄ • Reply • Share ›

Joe Clueless • 3 years ago

Nice article, thanks a lot :)

One question though: why isn't Either presented as a way to know what went wrong ?

Thanks in advance

4 ⌃ | ⌄ • Reply • Share ›

János Háber ➜ Joe Clueless • 2 years ago

@Li Haoyi +1 for Either.

⌃ | ⌄ • Reply • Share ›

Mateusz Maciaszek • 2 years ago

```
def getFibs(n: Int): Seq[Int] = {

val fibs = mutable.Buffer(1, 1)

while(fibs.length < n){

fibs.append(fibs(fibs.length-1) + fibs(fibs.length-2))

}

fibs

}
```

Wouldn't be better to explicitly go from mutable version to immutable in last instruction by doing sth like fibs.toImmutable (dont know exact name). Such way we use mutable efficient version in body but client of method is not aware of that and as a result use immutable one. What do you think?

2 ⌃ | ⌄ • Reply • Share ›

lmm • 3 years ago

Excellent guide.

I think it's worth pointing out that generics (parametricity) reduce power as this can be counterintuitive. E.g. if a method takes a `List[Int]` but only operates on the structure of the list then it's better to write it as `[A] List [A]`.

When passinga function I find it aids debugging to define that function as an `object` rather than anonymously.

Opaque handles can be safer than primitives. E.g. if your library returns some kind of "handle" or id to be passed back later it should probably be an opaque type rather than an int.

It would be good to say when an interface should use a typeclasses constraint.

For a simple sealed trait for error handling it's worth pointing out the easy ways to make this support for/yield sugar (e.g. using scalaz disjunction rather than your own, or some kind of derivation mechanism)

For dependency injection it's worth mentioning the option of a reader monad or free monad style.

2 ⌃ | ⌄ • Reply • Share ›

Dermot Haughey • 10 months ago

I think this guide is so well written and clear it actually transcends being about Scala and becomes good advice for all programming.

Maybe you should write a "Clean Code" style book someday, but with all the things you've in particular learned about software construction. I personally feel there aren't enough books that have a 'back to basics' kind of focus but with modern principles.

1 ^ | v • Reply • Share ›

Mikaël Mayer • a year ago

I love to read these design principles, good job.
Small typo: The link for "If you know there is only one thing that can go wrong, use an Option" targets #Option, but it should be #option (with small 'o')

^ | v • Reply • Share ›

Mikaël Mayer ➜ Mikaël Mayer • a year ago

asynchrny => asynchrony.
Then delete these two posts once you're done.

^ | v • Reply • Share ›

Daniel • a year ago

> In Java, this is bad, but very common

> ArrayList<int> myList = new java.util.ArrayList<int>()

That code should probably be:

List<int> myList = new java.util.ArrayList<int>()

and/or you should says what's bad about it before proceeding to giving equivalent Scala code. (The first thing bad I noticed about that code was the (apparently) overly specific type on the left.)

^ | v • Reply • Share ›

Alex Zvolinskiy • 2 years ago

The Data Types section was the most interesting for me.

What can you say about a function return type?

BTW: Error Handling section is brilliant =)

^ | v • Reply • Share ›

在原佐为 • 2 years ago

wonderful! Can i translate this into chinese and republich in gitbook?

^ | v • Reply • Share ›

Li Haoyi Mod ➜ 在原佐为 • 2 years ago

Sure! Just make sure you link to the original page at the top of the translation

^ | v • Reply • Share ›

rviswanadha • 2 years ago

> Scala is statically typed, so you don't need to fear refactoring
I disagree with this statement. The only way to not fear refactoring is when the code is accompanied by an exhaustive suite of tests. Static typing might help you make sure that code can be compiled, but it will not guarantee that the code will produce expected results.

^ | v • Reply • Share ›

Mark van Buskirk • 2 years ago

In Asynchronous Return Types:
Do you think returning Future[T] because it indicates to the user something about the nature of the call? How about tradeoffs between returning a Future[T] and a Sealed Trait with Errors like Timeout?

^ | v • Reply • Share ›

Mark van Buskirk • 2 years ago

Recommendation question:
Do you think think the Built-Ins section would be clearer if the int "x" in question had a clear meaning. Even make it a case class UserId like this:

case class UserId(id: Int)
class Foo(val id: UserId, val s: String, val d: Double)
val foo = new Foo(UserId(123), "hellol", 1.23)

----- better ---
def handle(userId: UserId) = {
... userId.id ... // Only one usage of userId.id
}

handle(foo.id)

---- worse ---

def handle(foo: Foo) = {
... foo.id.id ... // Only one usage of foo.id.id
}

handle(foo)

^ | v • Reply • Share ›

Philippe Derome • 2 years ago

Thanks a lot Haoyi, you contribute a lot to the community. While learning Scala and Functional Programming on a pet project over the past year, numerous times I had the temptation to try out a novel, sexy, advanced concept (a particular design pattern or a more exotic/recent Scala feature) only to find out after some analysis that in a simple enough project (< 3000 LOCs), it didn't bring anything useful to the table, instead it made things more complex and hard to understand, confirming the premise of the blog. Reading this blog a few days ago also encouraged me to simplify some code further and reduced much boiler-plate and learning some things along the way. In particular, the Hard-code dependency discussed here does work well in many cases as it makes the code easier to understand and maintain and not particularly big. As everybody here knows, accomplishing small, simple, robust code is quite hard and labour intensive, the first solution that comes to mind can often be full of boiler-plate or be unnecessarily complex.

^ | v • Reply • Share ›

dividebyzero • 2 years ago

Great post, clearly written by someone with a lot of experience!

Apart from also missing a mention to Either, I would like to challenge you a bit on the mutability part... Since I started learning Scala, I've been convinced it is really good to stick to immutable data types, and that performance losses are actually pretty rare. Articles about the topic even point out that some tasks are actually faster with immutable, contradicting common sense of most, and we're not even talking about exploiting parallelism here...

You gave the impression mutable data types can routinely offer performance gains of 10x to 100x, and that might justify using them most of the time. But the whole literature about immutable data types, specially inside Scala, seem to point out the opposite, that you should triple-check any uses of mutable, and that you won't be missing too much in performance for that.

^ | v • Reply • Share ›

Zax • 2 years ago

Awesome post and a great series!

Can someone explain to me why in the section "Simple Sealed Trait" the example shows a companion object that contains the case classes extending the sealed trait as opposed to just the train and the case classes?

^ | v • Reply • Share ›

Henry Story • 2 years ago

nice to see this principle which is fundamental to the web being applied to Scala. https://www.w3.org/2001/tag...

^ | v • Reply • Share ›

Cristian Arcaroli • 2 years ago

Great article!! It was a pleasure to read it, I think everything is sacred word! Well done! I would have added a mention to Try[T] but that's only to be picky ;-)

^ | v • Reply • Share ›

LG Optimusv • 2 years ago

How can I be notified when this post is modified? If there's no way to do it, could you consider moving this to github like https://github.com/johnpapa...

^ | v • Reply • Share ›

dwalend • 3 years ago

Why no mention of Tuples (for better or worse) in Data Types?

^ | v • Reply • Share ›

dwalend • 3 years ago

Another typo : ADT isn't defined, but seems to have something to do with simple sealed traits.

^ | v • Reply • Share ›

John Sullivan ➦ dwalend • 2 years ago

"abstract data type" https://en.wikipedia.org/wi...

1 ^ | v • Reply • Share ›

Daniel ➦ John Sullivan • a year ago

"abstract data type" or "algebraic data type"?

^ | v • Reply • Share ›

John Sullivan ➦ Daniel • a year ago

You're right. Thanks for the correction Daniel.

^ | v • Reply • Share ›

dwalend • 3 years ago

Great read. I'll be recommending it to others. Typo in "If I see a primitive or built-in collection, I know exactly [what] it contains."

^ | v • Reply • Share ›

Reply • Share ›

gertjanassies • 3 years ago

Good article!, one comment though: shouldn't names always be descriptive? your `tpose` should in my opinion also be the longer descriptive one as it saves me from dissecting that jumble of letters and brackets that follow it

Reply • Share ›

Li Haoyi Mod → gertjanassies • 3 years ago

This post explicitly avoids the question of the *content* of the names, focusing only on the length. I intentionally chose names that were roughly equally cryptic in these examples just to make the point that length matters separately from all other concerns like "conventions" or "english" or "domain-specific relevance"

I could talk about name content that but it would be another post

Reply • Share ›

ScalaCourses.com • 3 years ago

This post provides excellent guidelines. Thank you for once again donating you time and energy to the world for a good cause!

Reply • Share ›

Tom Flaherty • 3 years ago

Brilliant post. Glad Martin tweeted it. Opens up new possibilities and language paradigms.

Reply • Share ›

Juan José Vázquez Delgado • 3 years ago

Very enjoyable reading. It could be summed up as "less is more", right?.

Reply • Share ›

Load more comments

ALSO ON **LIHAOYI.COM**

So, what's wrong with SBT?

32 comments • 10 months ago

martin odersky — I agree with most of your points. For context, here is a critique of SBT I wrote 6 years ago where I suggested some radical changes. Did not …

uTest: the Essential Test Framework for Scala

2 comments • a year ago

Shani Elharrar — You always seem to write such a good libraries and to maintain them too. How do you manage to split your time between work, life, and …

Build your own Command Line with ANSI escape codes

12 comments • 2 years ago

Alex Gerdom — Great article. You actually should be able to avoid having to use `sys.stdout`, by using a trailing comma in python 2 or with the end keyword …

What's Functional Programming All About?

26 comments • 2 years ago

Carlos Silva — This article is an absolute Masterpiece.It removes all the clutter when comparing these programming paradigms, using a great analogy.I am …

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Disqus' Privacy PolicyPrivacy PolicyPrivacy**