Effective Scala

Marius Eriksen, Twitter Inc.
marius@twitter.com (@marius)

Table of Contents

Other languages

日本語 Русский 简体中文

Introduction

Scala is one of the main application programming languages used at Twitter. Much of our infrastructure is written in Scala and we have several large libraries supporting our use. While highly effective, Scala is also a large language, and our experiences have taught us to practice great care in its application. What are its pitfalls? Which features do we embrace, which do we eschew? When do we employ "purely functional style", and when do we avoid it? In other words: what have we found to be an effective use of the language? This guide attempts to distill our experience into short essays, providing a set of *best practices*. Our use of Scala is mainly for creating high volume services that form distributed systems — and our advice is thus biased — but most of the advice herein should translate naturally to other domains. This is not the law, but deviation should be well justified.

Scala provides many tools that enable succinct expression. Less typing is less reading, and less reading is often faster reading, and thus brevity enhances clarity. However brevity is a blunt tool that can also deliver the opposite effect: After correctness, think always of the reader.

Above all, *program in Scala*. You are not writing Java, nor Haskell, nor Python; a Scala program is unlike one written in any of these. In order to use the language effectively, you must phrase your problems in its terms. There's no use coercing a Java program into Scala, for it will be inferior in most ways to its original.

This is not an introduction to Scala; we assume the reader is familiar with the language. Some resources for learning Scala are:

- Scala School
- Learning Scala
- Learning Scala in Small Bites

This is a living document that will change to reflect our current "best practices," but its core ideas are unlikely to change: Always favor readability; write generic code but not at the expensive of clarity; take advantage of simple language features that afford great power but avoid the esoteric ones (especially in the type system). Above all, be always aware of the trade offs you make. A sophisticated language requires a complex implementation, and complexity begets complexity: of reasoning, of semantics, of interaction between features, and of the understanding of your collaborators. Thus complexity is the tax of sophistication — you must always ensure that its utility exceeds its cost.

And have fun.

Formatting

The specifics of code *formatting* — so long as they are practical — are of little consequence. By definition style cannot be inherently good or bad and almost everybody differs in personal preference. However the *consistent* application of the same formatting rules will almost always enhance readability. A reader already familiar with a particular style does not have to grasp yet another set of local conventions, or decipher yet another corner of the language grammar.

This is of particular importance to Scala, as its grammar has a high degree of overlap. One telling example is method invocation: Methods can be invoked with ".", with whitespace, without parenthesis for nullary or unary methods, with parenthesis for these, and so on. Furthermore, the different styles of method invocations expose

different ambiguities in its grammar! Surely the consistent application of a carefully chosen set of formatting rules will resolve a great deal of ambiguity for both man and machine.

We adhere to the Scala style guide plus the following rules.

Whitespace

Indent by two spaces. Try to avoid lines greater than 100 columns in length. Use one blank line between method, class, and object definitions.

Naming

Use short names for small scopes
: `i`s, `j`s and `k`s are all but expected in loops.

Use longer names for larger scopes
: External APIs should have longer and explanatory names that confer meaning. `Future.collect` not `Future.all`.

Use common abbreviations but eschew esoteric ones
: Everyone knows `ok`, `err` or `defn` whereas `sfri` is not so common.

Don't rebind names for different uses
: Use `val`s

Avoid using `s to overload reserved names.
: `typ` instead of `` `type` ``

Use active names for operations with side effects
: `user.activate()` not `user.setActive()`

Use descriptive names for methods that return values
: `src.isDefined` not `src.defined`

Don't prefix getters with `get`
: As per the previous rule, it's redundant: `site.count` not `site.getCount`

Don't repeat names that are already encapsulated in package or object name
: Prefer:
```
object User {
  def get(id: Int): Option[User]
}
```
to
```
object User {
  def getUser(id: Int): Option[User]
}
```
They are redundant in use: `User.getUser` provides no more information than `User.get`.

Imports

Sort import lines alphabetically
: This makes it easy to examine visually, and is simple to automate.

Use braces when importing several names from a package
: `import com.twitter.concurrent.{Broker, Offer}`

Use wildcards when more than six names are imported
: e.g.: `import com.twitter.concurrent._`
Don't apply this blindly: some packages export too many names

When using collections, qualify names by importing `scala.collection.immutable` and/or `scala.collection.mutable`
: Mutable and immutable collections have dual names. Qualifiying the names makes is obvious to the reader which variant is being used (e.g. "`immutable.Map`")

Do not use relative imports from other packages
: Avoid
```
import com.twitter
import concurrent
```
in favor of the unambiguous
```
import com.twitter.concurrent
```

Put imports at the top of the file
: The reader can refer to all imports in one place.

Braces

Braces are used to create compound expressions (they serve other uses in the “module language”), where the value of the compound expression is the last expression in the list. Avoid using braces for simple expressions; write

```
def square(x: Int) = x*x
```

but not

```
def square(x: Int) = {
  x * x
}
```

even though it may be tempting to distinguish the method body syntactically. The first alternative has less clutter and is easier to read. *Avoid syntactical ceremony* unless it clarifies.

Pattern matching

Use pattern matching directly in function definitions whenever applicable; instead of

```
list map { item =>
  item match {
    case Some(x) => x
    case None => default
  }
}
```

collapse the match

```
list map {
  case Some(x) => x
  case None => default
}
```

it's clear that the list items are being mapped over — the extra indirection does not elucidate.

Comments

Use ScalaDoc to provide API documentation. Use the following style:

```
/**
 * ServiceBuilder builds services
 * ...
 */
```

but *not* the standard ScalaDoc style:

```
/** ServiceBuilder builds services
 * ...
 */
```

Do not resort to ASCII art or other visual embellishments. Document APIs but do not add unnecessary comments. If you find yourself adding comments to explain the behavior of your code, ask first if it can be restructured so that it becomes obvious what it does. Prefer “obviously it works” to “it works, obviously” (with apologies to Hoare).

Types and Generics

The primary objective of a type system is to detect programming errors. The type system effectively provides a limited form of static verification, allowing us to express certain kinds of invariants about our code that the compiler can verify. Type systems provide other benefits too of course, but error checking is its Raison d’Être.

Our use of the type system should reflect this goal, but we must remain mindful of the reader: judicious use of types can serve to enhance clarity, being unduly clever only obfuscates.

Scala’s powerful type system is a common source of academic exploration and exercise (eg. Type level programming in Scala). While a fascinating academic topic, these techniques rarely find useful application in production code. They are to be avoided.

Return type annotations

While Scala allows these to be omitted, such annotations provide good documentation: this is especially important for public methods. Where a method is not exposed and its return type obvious, omit them.

This is especially important when instantiating objects with mixins as the scala compiler creates singleton types for these. For example, `make` in:

```
trait Service
def make() = new Service {
  def getId = 123
}
```

does *not* have a return type of `Service`; the compiler creates the refinement type `Object with Service{def getId: Int}`. Instead use an explicit annotation:

```
def make(): Service = new Service{}
```

Now the author is free to mix in more traits without changing the public type of `make`, making it easier to manage backwards compatibility.

Variance

Variance arises when generics are combined with subtyping. Variance defines how subtyping of the *contained* type relates to subtyping of the *container* type. Because Scala has declaration site variance annotations, authors of common libraries — especially collections — must be prolific annotators. Such annotations are important for the usability of shared code, but misapplication can be dangerous.

Invariants are an advanced but necessary aspect of Scala’s typesystem, and should be used widely (and correctly) as it aids the application of subtyping.

Immutable collections should be covariant. Methods that receive the contained type should “downgrade” the collection appropriately:

```
trait Collection[+T] {
```

```
  def add[U >: T](other: U): Collection[U]
}
```

Mutable collections should be invariant. Covariance is typically invalid with mutable collections. Consider

```
trait HashSet[+T] {
  def add[U >: T](item: U)
}
```

and the following type hierarchy:

```
trait Mammal
trait Dog extends Mammal
trait Cat extends Mammal
```

If I now have a hash set of dogs

```
val dogs: HashSet[Dog]
```

treat it as a set of Mammals and add a cat.

```
val mammals: HashSet[Mammal] = dogs
mammals.add(new Cat{})
```

This is no longer a HashSet of dogs!

Type aliases

Use type aliases when they provide convenient naming or clarify purpose, but do not alias types that are self-explanatory.

```
() => Int
```

is clearer than

```
type IntMaker = () => Int
IntMaker
```

since it is both short and uses a common type. However

```
class ConcurrentPool[K, V] {
  type Queue = ConcurrentLinkedQueue[V]
  type Map   = ConcurrentHashMap[K, Queue]
  ...
}
```

is helpful since it communicates purpose and enhances brevity.

Don't use subclassing when an alias will do.

```
trait SocketFactory extends (SocketAddress => Socket)
```

a `SocketFactory` *is* a function that produces a `Socket`. Using a type alias

```
type SocketFactory = SocketAddress => Socket
```

is better. We may now provide function literals for values of type `SocketFactory` and also use function composition:

```
val addrToInet: SocketAddress => Long
val inetToSocket: Long => Socket

val factory: SocketFactory = addrToInet andThen inetToSocket
```

Type aliases are bound to toplevel names by using package objects:

```
package com.twitter
package object net {
  type SocketFactory = (SocketAddress) => Socket
}
```

Note that type aliases are not new types — they are equivalent to the syntactically substituting the aliased name for its type.

Implicits

Implicits are a powerful type system feature, but they should be used sparingly. They have complicated resolution rules and make it difficult — by simple lexical examination — to grasp what is actually happening. It's definitely OK to use implicits in the following situations:

- Extending or adding a Scala-style collection
- Adapting or extending an object ("pimp my library" pattern)
- Use to *enhance type safety* by providing constraint evidence
- To provide type evidence (typeclassing)
- For `Manifest`s

If you do find yourself using implicits, always ask yourself if there is a way to achieve the same thing without their help.

Do not use implicits to do automatic conversions between similar datatypes (for example, converting a list to a

stream); these are better done explicitly because the types have different semantics, and the reader should beware of these implications.

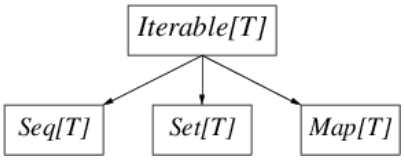
Collections

Scala has a very generic, rich, powerful, and composable collections library; collections are high level and expose a large set of operations. Many collection manipulations and transformations can be expressed succinctly and readably, but careless application of these features can often lead to the opposite result. Every Scala programmer should read the collections design document; it provides great insight and motivation for Scala collections library.

Always use the simplest collection that meets your needs.

Hierarchy

The collections library is large: in addition to an elaborate hierarchy — the root of which being `Traversable[T]` — there are `immutable` and `mutable` variants for most collections. Whatever the complexity, the following diagram contains the important distinctions for both `immutable` and `mutable` hierarchies



`Iterable[T]` is any collection that may be iterated over, they provide an `iterator` method (and thus `foreach`). `Seq[T]`s are collections that are *ordered*, `Set[T]`s are mathematical sets (unordered collections of unique items), and `Map[T]`s are associative arrays, also unordered.

Use

Prefer using immutable collections. They are applicable in most circumstances, and make programs easier to reason about since they are referentially transparent and are thus also threadsafe by default.

Use the `mutable` namespace explicitly. Don't import `scala.collection.mutable._` and refer to `Set`, instead

```
import scala.collection.mutable
val set = mutable.Set()
```

makes it clear that the mutable variant is being used.

Use the default constructor for the collection type. Whenever you need an ordered sequence (and not necessarily linked list semantics), use the `Seq()` constructor, and so on:

```
val seq = Seq(1, 2, 3)
val set = Set(1, 2, 3)
val map = Map(1 -> "one", 2 -> "two", 3 -> "three")
```

This style separates the semantics of the collection from its implementation, letting the collections library use the most appropriate type: you need a `Map`, not necessarily a Red-Black Tree. Furthermore, these default constructors will often use specialized representations: for example, `Map()` will use a 3-field object for maps with 3 keys.

The corollary to the above is: in your own methods and constructors, *receive the most generic collection type appropriate.* This typically boils down to one of the above: `Iterable`, `Seq`, `Set`, or `Map`. If your method needs a sequence, use `Seq[T]`, not `List[T]`. (A word of caution: the *default* `Traversable`, `Iterable` and `Seq` types in scope – defined in `scala.package` – are the `scala.collection` versions, as opposed to `Map` and `Set` – defined in `Predef.scala` – which are the `scala.collection.immutable` versions. This means that, for example, the default `Seq` type can be both the immutable *and* mutable implementations. Thus, if your method relies on a collection parameter being immutable, and you are using `Traversable`, `Iterable` or `Seq`, you *must* specifically require/import the immutable variant, otherwise someone *may* pass you the mutable version.)

Style

Functional programming encourages pipelining transformations of an immutable collection to shape it to its desired result. This often leads to very succinct solutions, but can also be confusing to the reader — it is often difficult to discern the author's intent, or keep track of all the intermediate results that are only implied. For example, let's say we wanted to aggregate votes for different programming languages from a sequence of (language, num votes), showing them in order of most votes to least, we could write:

```
val votes = Seq(("scala", 1), ("java", 4), ("scala", 10), ("scala", 1), ("python", 10))
val orderedVotes = votes
  .groupBy(_._1)
  .map { case (which, counts) =>
    (which, counts.foldLeft(0)(_ + _._2))
  }.toSeq
  .sortBy(_._2)
  .reverse
```

this is both succinct and correct, but nearly every reader will have a difficult time recovering the original intent of the author. A strategy that often serves to clarify is to *name intermediate results and parameters*:

```
val votesByLang = votes groupBy { case (lang, _) => lang }
val sumByLang = votesByLang map { case (lang, counts) =>
```

```
  val countsOnly = counts map { case (_, count) => count }
  (lang, countsOnly.sum)
}
val orderedVotes = sumByLang.toSeq
  .sortBy { case (_, count) => count }
  .reverse
```

the code is nearly as succinct, but much more clearly expresses both the transformations take place (by naming intermediate values), and the structure of the data being operated on (by naming parameters). If you worry about namespace pollution with this style, group expressions with `{}`:

```
val orderedVotes = {
  val votesByLang = ...
  ...
}
```

Performance

High level collections libraries (as with higher level constructs generally) make reasoning about performance more difficult: the further you stray from instructing the computer directly — in other words, imperative style — the harder it is to predict the exact performance implications of a piece of code. Reasoning about correctness however, is typically easier; readability is also enhanced. With Scala the picture is further complicated by the Java runtime; Scala hides boxing/unboxing operations from you, which can incur severe performance or space penalties.

Before focusing on low level details, make sure you are using a collection appropriate for your use. Make sure your datastructure doesn't have unexpected asymptotic complexity. The complexities of the various Scala collections are described here.

The first rule of optimizing for performance is to understand *why* your application is slow. Do not operate without data; profile[1] your application before proceeding. Focus first on hot loops and large data structures. Excessive focus on optimization is typically wasted effort. Remember Knuth's maxim: "Premature optimisation is the root of all evil."

It is often appropriate to use lower level collections in situations that require better performance or space efficiency. Use arrays instead of lists for large sequences (the immutable `Vector` collections provides a referentially transparent interface to arrays); and use buffers instead of direct sequence construction when performance matters.

Java Collections

Use `scala.collection.JavaConverters` to interoperate with Java collections. These are a set of implicits that add `asJava` and `asScala` conversion methods. The use of these ensures that such conversions are explicit, aiding the reader:

```
import scala.collection.JavaConverters._

val list: java.util.List[Int] = Seq(1,2,3,4).asJava
val buffer: scala.collection.mutable.Buffer[Int] = list.asScala
```

Concurrency

Modern services are highly concurrent — it is common for servers to coordinate 10s–100s of thousands of simultaneous operations — and handling the implied complexity is a central theme in authoring robust systems software.

Threads provide a means of expressing concurrency: they give you independent, heap-sharing execution contexts that are scheduled by the operating system. However, thread creation is expensive in Java and is a resource that must be managed, typically with the use of pools. This creates additional complexity for the programmer, and also a high degree of coupling: it's difficult to divorce application logic from their use of the underlying resources.

This complexity is especially apparent when creating services that have a high degree of fan-out: each incoming request results in a multitude of requests to yet another tier of systems. In these systems, thread pools must be managed so that they are balanced according to the ratios of requests in each tier: mismanagement of one thread pool bleeds into another.

Robust systems must also consider timeouts and cancellation, both of which require the introduction of yet more "control" threads, complicating the problem further. Note that if threads were cheap these problems would be diminished: no pooling would be required, timed out threads could be discarded, and no additional resource management would be required.

Thus resource management compromises modularity.

Futures

Use Futures to manage concurrency. They decouple concurrent operations from resource management: for example, Finagle multiplexes concurrent operations onto few threads in an efficient manner. Scala has lightweight closure literal syntax, so Futures introduce little syntactic overhead, and they become second nature to most programmers.

Futures allow the programmer to express concurrent computation in a declarative style, are composable, and

have principled handling of failure. These qualities have convinced us that they are especially well suited for use in functional programming languages, where this is the encouraged style.

Prefer transforming futures over creating your own. Future transformations ensure that failures are propagated, that cancellations are signalled, and free the programmer from thinking about the implications of the Java memory model. Even a careful programmer might write the following to issue an RPC 10 times in sequence and then print the results:

```
val p = new Promise[List[Result]]
var results: List[Result] = Nil
def collect() {
  doRpc() onSuccess { result =>
    results = result :: results
    if (results.length < 10)
      collect()
    else
      p.setValue(results)
  } onFailure { t =>
    p.setException(t)
  }
}

collect()
p onSuccess { results =>
  printf("Got results %s\n", results.mkString(", "))
}
```

The programmer had to ensure that RPC failures are propagated, interspersing the code with control flow; worse, the code is wrong! Without declaring `results` volatile, we cannot ensure that `results` holds the previous value in each iteration. The Java memory model is a subtle beast, but luckily we can avoid all of these pitfalls by using the declarative style:

```
def collect(results: List[Result] = Nil): Future[List[Result]] =
  doRpc() flatMap { result =>
    if (results.length < 9)
      collect(result :: results)
    else
      Future.value(result :: results)
  }

collect() onSuccess { results =>
  printf("Got results %s\n", results.mkString(", "))
}
```

We use `flatMap` to sequence operations and prepend the result onto the list as we proceed. This is a common functional programming idiom translated to Futures. This is correct, requires less boilerplate, is less error prone, and also reads better.

Use the Future combinators. `Future.select`, `Future.join`, and `Future.collect` codify common patterns when operating over multiple futures that should be combined.

Do not throw your own exceptions in methods that return Futures. Futures represent both successful and failed computations. Therefore, it's important that errors involved in that computation are properly encapsulated in the returned Future. Concretely, return `Future.exception` instead of throwing that exception:

```
def divide(x: Int, y: Int): Future[Result] = {
  if (y == 0)
    return Future.exception(new IllegalArgumentException("Divisor is 0"))

  Future.value(x/y)
}
```

Fatal exceptions should not be represented by Futures. These exceptions include ones that are thrown when resources are exhausted, like OutOfMemoryError, and also JVM-level errors like NoSuchMethodError. These conditions are ones under which the JVM must exit.

The predicates `scala.util.control.NonFatal` — or Twitter's version `com.twitter.util.NonFatal` — should be used to identify exceptions which should be returned as a Future.exception.

Collections

The subject of concurrent collections is fraught with opinions, subtleties, dogma and FUD. In most practical situations they are a nonissue: Always start with the simplest, most boring, and most standard collection that serves the purpose. Don't reach for a concurrent collection before you *know* that a synchronized one won't do: the JVM has sophisticated machinery to make synchronization cheap, so their efficacy may surprise you.

If an immutable collection will do, use it — they are referentially transparent, so reasoning about them in a concurrent context is simple. Mutations in immutable collections are typically handled by updating a reference to the current value (in a `var` cell or an `AtomicReference`). Care must be taken to apply these correctly: atomics must be retried, and `vars` must be declared volatile in order for them to be published to other threads.

Mutable concurrent collections have complicated semantics, and make use of subtler aspects of the Java memory model, so make sure you understand the implications — especially with respect to publishing updates — before you use them. Synchronized collections also compose better: operations like `getOrElseUpdate` cannot be implemented correctly by concurrent collections, and creating composite collections is especially error prone.

Control structures

Programs in the functional style tend to require fewer traditional control structures, and read better when written in the declarative style. This typically implies breaking your logic up into several small methods or

functions, and gluing them together with `match` expressions. Functional programs also tend to be more expression-oriented: branches of conditionals compute values of the same type, `for (..) yield` computes comprehensions, and recursion is commonplace.

Recursion

Phrasing your problem in recursive terms often simplifies it, and if the tail call optimization applies (which can be checked by the `@tailrec` annotation), the compiler will even translate your code into a regular loop.

Consider a fairly standard imperative version of heap FIX-DOWN:

```
def fixDown(heap: Array[T], m: Int, n: Int): Unit = {
  var k: Int = m
  while (n >= 2*k) {
    var j = 2*k
    if (j < n && heap(j) < heap(j + 1))
      j += 1
    if (heap(k) >= heap(j))
      return
    else {
      swap(heap, k, j)
      k = j
    }
  }
}
```

Every time the while loop is entered, we're working with state dirtied by the previous iteration. The value of each variable is a function of which branches were taken, and it returns in the middle of the loop when the correct position was found (The keen reader will find similar arguments in Dijkstra's "Go To Statement Considered Harmful").

Consider a (tail) recursive implementation[2]:

```
@tailrec
final def fixDown(heap: Array[T], i: Int, j: Int) {
  if (j < i*2) return

  val m = if (j == i*2 || heap(2*i) < heap(2*i+1)) 2*i else 2*i + 1
  if (heap(m) < heap(i)) {
    swap(heap, i, m)
    fixDown(heap, m, j)
  }
}
```

here every iteration starts with a well-defined *clean slate*, and there are no reference cells: invariants abound. It's much easier to reason about, and easier to read as well. There is also no performance penalty: since the method is tail-recursive, the compiler translates this into a standard imperative loop.

Returns

This is not to say that imperative structures are not also valuable. In many cases they are well suited to terminate computation early instead of having conditional branches for every possible point of termination: indeed in the above `fixDown`, a `return` is used to terminate early if we're at the end of the heap.

Returns can be used to cut down on branching and establish invariants. This helps the reader by reducing nesting (how did I get here?) and making it easier to reason about the correctness of subsequent code (the array cannot be accessed out of bounds after this point). This is especially useful in "guard" clauses:

```
def compare(a: AnyRef, b: AnyRef): Int = {
  if (a eq b)
    return 0

  val d = System.identityHashCode(a) compare System.identityHashCode(b)
  if (d != 0)
    return d

  // slow path..
}
```

Use `return`s to clarify and enhance readability, but not as you would in an imperative language; avoid using them to return the results of a computation. Instead of

```
def suffix(i: Int) = {
  if      (i == 1) return "st"
  else if (i == 2) return "nd"
  else if (i == 3) return "rd"
  else             return "th"
}
```

prefer:

```
def suffix(i: Int) =
  if      (i == 1) "st"
  else if (i == 2) "nd"
  else if (i == 3) "rd"
  else             "th"
```

but using a `match` expression is superior to either:

```
def suffix(i: Int) = i match {
  case 1 => "st"
  case 2 => "nd"
  case 3 => "rd"
  case _ => "th"
}
```

Note that returns can have hidden costs: when used inside of a closure,

```
seq foreach { elem =>
  if (elem.isLast)
    return

  // process...
}
```

this is implemented in bytecode as an exception catching/throwing pair which, used in hot code, has performance implications.

`for` loops and comprehensions

`for` provides both succinct and natural expression for looping and aggregation. It is especially useful when flattening many sequences. The syntax of `for` belies the underlying mechanism as it allocates and dispatches closures. This can lead to both unexpected costs and semantics; for example

```
for (item <- container) {
  if (item != 2) return
}
```

may cause a runtime error if the container delays computation, making the `return` nonlocal!

For these reasons, it is often preferrable to call `foreach`, `flatMap`, `map`, and `filter` directly — but do use `for`s when they clarify.

`require` and `assert`

`require` and `assert` both serve as executable documentation. Both are useful for situations in which the type system cannot express the required invariants. `assert` is used for *invariants* that the code assumes (either internal or external), for example

```
val stream = getClass.getResourceAsStream("someclassdata")
assert(stream != null)
```

Whereas `require` is used to express API contracts:

```
def fib(n: Int) = {
  require(n > 0)
  ...
}
```

Functional programming

Value oriented programming confers many advantages, especially when used in conjunction with functional programming constructs. This style emphasizes the transformation of values over stateful mutation, yielding code that is referentially transparent, providing stronger invariants and thus also easier to reason about. Case classes, pattern matching, destructuring bindings, type inference, and lightweight closure- and method-creation syntax are the tools of this trade.

Case classes as algebraic data types

Case classes encode ADTs: they are useful for modelling a large number of data structures and provide for succinct code with strong invariants, especially when used in conjunction with pattern matching. The pattern matcher implements exhaustivity analysis providing even stronger static guarantees.

Use the following pattern when encoding ADTs with case classes:

```
sealed trait Tree[T]
case class Node[T](left: Tree[T], right: Tree[T]) extends Tree[T]
case class Leaf[T](value: T) extends Tree[T]
```

The type `Tree[T]` has two constructors: `Node` and `Leaf`. Declaring the type `sealed` allows the compiler to do exhaustivity analysis since constructors cannot be added outside the source file.

Together with pattern matching, such modelling results in code that is both succinct and “obviously correct”:

```
def findMin[T <: Ordered[T]](tree: Tree[T]) = tree match {
  case Node(left, right) => Seq(findMin(left), findMin(right)).min
  case Leaf(value) => value
}
```

While recursive structures like trees constitute classic applications of ADTs, their domain of usefulness is much larger. Disjoint unions in particular are readily modelled with ADTs; these occur frequently in state machines.

Options

The `Option` type is a container that is either empty (`None`) or full (`Some(value)`). It provides a safe alternative to the use of `null`, and should be used instead of null whenever possible. Options are collections (of at most one item) and they are embellished with collection operations — use them!

Write

```
var username: Option[String] = None
...
```

```
username = Some("foobar")
```

instead of

```
var username: String = null
...
username = "foobar"
```

since the former is safer: the `Option` type statically enforces that `username` must be checked for emptyness.

Conditional execution on an `Option` value should be done with `foreach`; instead of

```
if (opt.isDefined)
  operate(opt.get)
```

write

```
opt foreach { value =>
  operate(value)
}
```

The style may seem odd, but provides greater safety (we don't call the exceptional `get`) and brevity. If both branches are taken, use pattern matching:

```
opt match {
  case Some(value) => operate(value)
  case None => defaultAction()
}
```

but if all that's missing is a default value, use `getOrElse`

```
operate(opt getOrElse defaultValue)
```

Do not overuse `Option`: if there is a sensible default — a *Null Object* — use that instead.

`Option` also comes with a handy constructor for wrapping nullable values:

```
Option(getClass.getResourceAsStream("foo"))
```

is an `Option[InputStream]` that assumes a value of `None` should `getResourceAsStream` return `null`.

Pattern matching

Pattern matches (`x match { ...`) are pervasive in well written Scala code: they conflate conditional execution, destructuring, and casting into one construct. Used well they enhance both clarity and safety.

Use pattern matching to implement type switches:

```
obj match {
  case str: String => ...
  case addr: SocketAddress => ...
```

Pattern matching works best when also combined with destructuring (for example if you are matching case classes); instead of

```
animal match {
  case dog: Dog => "dog (%s)".format(dog.breed)
  case _ => animal.species
  }
```

write

```
animal match {
  case Dog(breed) => "dog (%s)".format(breed)
  case other => other.species
}
```

Write custom extractors but only with a dual constructor (`apply`), otherwise their use may be out of place.

Don't use pattern matching for conditional execution when defaults make more sense. The collections libraries usually provide methods that return `Option`s; avoid

```
val x = list match {
  case head :: _ => head
  case Nil => default
}
```

because

```
val x = list.headOption getOrElse default
```

is both shorter and communicates purpose.

Partial functions

Scala provides syntactical shorthand for defining a `PartialFunction`:

```
val pf: PartialFunction[Int, String] = {
  case i if i%2 == 0 => "even"
}
```

and they may be composed with `orElse`

```
val tf: (Int => String) = pf orElse { case _ => "odd"}

tf(1) == "odd"
tf(2) == "even"
```

Partial functions arise in many situations and are effectively encoded with `PartialFunction`, for example as arguments to methods

```
trait Publisher[T] {
  def subscribe(f: PartialFunction[T, Unit])
}

val publisher: Publisher[Int] = ...
publisher.subscribe {
  case i if isPrime(i) => println("found prime", i)
  case i if i%2 == 0 => count += 2
  /* ignore the rest */
}
```

or in situations that might otherwise call for returning an `Option`:

```
// Attempt to classify the the throwable for logging.
type Classifier = Throwable => Option[java.util.logging.Level]
```

might be better expressed with a `PartialFunction`

```
type Classifier = PartialFunction[Throwable, java.util.Logging.Level]
```

as it affords greater composability:

```
val classifier1: Classifier
val classifier2: Classifier

val classifier: Classifier = classifier1 orElse classifier2 orElse { case _ => java.util.Logging.Level.FINEST }
```

Destructuring bindings

Destructuring value bindings are related to pattern matching; they use the same mechanism but are applicable when there is exactly one option (lest you accept the possibility of an exception). Destructuring binds are particularly useful for tuples and case classes.

```
val tuple = ('a', 1)
val (char, digit) = tuple

val tweet = Tweet("just tweeting", Time.now)
val Tweet(text, timestamp) = tweet
```

Laziness

Fields in scala are computed *by need* when `val` is prefixed with `lazy`. Because fields and methods are equivalent in Scala (lest the fields are `private[this]`)

```
lazy val field = computation()
```

is (roughly) short-hand for

```
var _theField = None
def field = if (_theField.isDefined) _theField.get else {
  _theField = Some(computation())
  _theField.get
}
```

i.e., it computes a results and memoizes it. Use lazy fields for this purpose, but avoid using laziness when laziness is required by semantics. In these cases it's better to be explicit since it makes the cost model explicit, and side effects can be controlled more precisely.

Lazy fields are thread safe.

Call by name

Method parameters may be specified by-name, meaning the parameter is bound not to a value but to a *computation* that may be repeated. This feature must be applied with care; a caller expecting by-value semantics will be surprised. The motivation for this feature is to construct syntactically natural DSLs — new control constructs in particular can be made to look much like native language features.

Only use call-by-name for such control constructs, where it is obvious to the caller that what is being passed in is a “block” rather than the result of an unsuspecting computation. Only use call-by-name arguments in the last position of the last argument list. When using call-by-name, ensure that the method is named so that it is obvious to the caller that its argument is call-by-name.

When you do want a value to be computed multiple times, and especially when this computation is side effecting, use explicit functions:

```
class SSLConnector(mkEngine: () => SSLEngine)
```

The intent remains obvious and the caller is left without surprises.

flatMap

`flatMap` — the combination of `map` with `flatten` — deserves special attention, for it has subtle power and great

utility. Like its brethren `map`, it is frequently available in nontraditional collections such as `Future` and `Option`. Its behavior is revealed by its signature; for some `Container[A]`

```
flatMap[B](f: A => Container[B]): Container[B]
```

`flatMap` invokes the function `f` for the element(s) of the collection producing a *new* collection, (all of) which are flattened into its result. For example, to get all permutations of two character strings that aren't the same character repeated twice:

```
val chars = 'a' to 'z'
val perms = chars flatMap { a =>
  chars flatMap { b =>
    if (a != b) Seq("%c%c".format(a, b))
    else Seq()
  }
}
```

which is equivalent to the more concise for-comprehension (which is — roughly — syntactical sugar for the above):

```
val perms = for {
  a <- chars
  b <- chars
  if a != b
} yield "%c%c".format(a, b)
```

`flatMap` is frequently useful when dealing with `Options` — it will collapse chains of options down to one,

```
val host: Option[String] = ...
val port: Option[Int] = ...

val addr: Option[InetSocketAddress] =
  host flatMap { h =>
    port map { p =>
      new InetSocketAddress(h, p)
    }
  }
```

which is also made more succinct with `for`

```
val addr: Option[InetSocketAddress] = for {
  h <- host
  p <- port
} yield new InetSocketAddress(h, p)
```

The use of `flatMap` in `Future`s is discussed in the futures section.

Object oriented programming

Much of Scala's vastness lies in its object system. Scala is a *pure* language in the sense that *all values* are objects; there is no distinction between primitive types and composite ones. Scala also features mixins allowing for more orthogonal and piecemeal construction of modules that can be flexibly put together at compile time with all the benefits of static type checking.

A motivation behind the mixin system was to obviate the need for traditional dependency injection. The culmination of this "component style" of programming is the cake pattern.

Dependency injection

In our use, however, we've found that Scala itself removes so much of the syntactical overhead of "classic" (constructor) dependency injection that we'd rather just use that: it is clearer, the dependencies are still encoded in the (constructor) type, and class construction is so syntactically trivial that it becomes a breeze. It's boring and simple and it works. *Use dependency injection for program modularization*, and in particular, *prefer composition over inheritance* — for this leads to more modular and testable programs. When encountering a situation requiring inheritance, ask yourself: how would you structure the program if the language lacked support for inheritance? The answer may be compelling.

Dependency injection typically makes use of traits,

```
trait TweetStream {
  def subscribe(f: Tweet => Unit)
}
class HosebirdStream extends TweetStream ...
class FileStream extends TweetStream ...

class TweetCounter(stream: TweetStream) {
  stream.subscribe { tweet => count += 1 }
}
```

It is common to inject *factories* — objects that produce other objects. In these cases, favor the use of simple functions over specialized factory types.

```
class FilteredTweetCounter(mkStream: Filter => TweetStream) {
  mkStream(PublicTweets).subscribe { tweet => publicCount += 1 }
  mkStream(DMs).subscribe { tweet => dmCount += 1 }
}
```

Traits

Dependency injection does not at all preclude the use of common *interfaces*, or the implementation of common code in traits. Quite the contrary — the use of traits are highly encouraged for exactly this reason: multiple interfaces (traits) may be implemented by a concrete class, and common code can be reused across all such

classes.

Keep traits short and orthogonal: don't lump separable functionality into a trait, think of the smallest related ideas that fit together. For example, imagine you have an something that can do IO:

```
trait IOer {
  def write(bytes: Array[Byte])
  def read(n: Int): Array[Byte]
}
```

separate the two behaviors:

```
trait Reader {
  def read(n: Int): Array[Byte]
}
trait Writer {
  def write(bytes: Array[Byte])
}
```

and mix them together to form what was an `IOer`: `new Reader with Writer`... Interface minimalism leads to greater orthogonality and cleaner modularization.

Visibility

Scala has very expressive visibility modifiers. It's important to use these as they define what constitutes the *public API*. Public APIs should be limited so users don't inadvertently rely on implementation details and limit the author's ability to change them: They are crucial to good modularity. As a rule, it's much easier to expand public APIs than to contract them. Poor annotations can also compromise backwards binary compatibility of your code.

`private[this]`

A class member marked `private`,

```
private val x: Int = ...
```

is visible to all *instances* of that class (but not their subclasses). In most cases, you want `private[this]`.

```
private[this] val x: Int = ...
```

which limits visibility to the particular instance. The Scala compiler is also able to translate `private[this]` into a simple field access (since access is limited to the statically defined class) which can sometimes aid performance optimizations.

Singleton class types

It's common in Scala to create singleton class types, for example

```
def foo() = new Foo with Bar with Baz {
  ...
}
```

In these situations, visibility can be constrained by declaring the returned type:

```
def foo(): Foo with Bar = new Foo with Bar with Baz {
  ...
}
```

where callers of `foo()` will see a restricted view (`Foo with Bar`) of the returned instance.

Structural typing

Do not use structural types in normal use. They are a convenient and powerful feature, but unfortunately do not have an efficient implementation on the JVM. However — due to an implementation quirk — they provide a very nice shorthand for doing reflection.

```
val obj: AnyRef
obj.asInstanceOf[{def close()}].close()
```

Error handling

Scala provides an exception facility, but do not use it for commonplace errors, when the programmer must handle errors properly for correctness. Instead, encode such errors explicitly: using `Option` or `com.twitter.util.Try` are good, idiomatic choices, as they harness the type system to ensure that the user is properly considering error handling.

For example, when designing a repository, the following API may be tempting:

```
trait Repository[Key, Value] {
  def get(key: Key): Value
}
```

but this would require the implementor to throw an exception when the key is absent. A better approach is to use an `Option`:

```
trait Repository[Key, Value] {
  def get(key: Key): Option[Value]
}
```

This interface makes it obvious that the repository may not contain every key, and that the programmer must handle missing keys. Furthermore, `Option` has a number of combinators to handle these cases. For example, `getOrElse` is used to supply a default value for missing keys:

```
val repo: Repository[Int, String]
repo.get(123) getOrElse "defaultString"
```

Handling exceptions

Because Scala's exception mechanism isn't *checked* — the compiler cannot statically tell whether the programmer has covered the set of possible exceptions — it is often tempting to cast a wide net when handling exceptions.

However, some exceptions are *fatal* and should never be caught; the code

```
try {
  operation()
} catch {
  case _ => ...
}
```

is almost always wrong, as it would catch fatal errors that need to be propagated. Instead, use the `com.twitter.util.NonFatal` extractor to handle only nonfatal exceptions.

```
try {
  operation()
} catch {
  case NonFatal(exc) => ...
}
```

Garbage collection

We spend a lot of time tuning garbage collection in production. The garbage collection concerns are largely similar to those of Java though idiomatic Scala code tends to generate more (short-lived) garbage than idiomatic Java code — a byproduct of the functional style. Hotspot's generational garbage collection typically makes this a nonissue as short-lived garbage is effectively free in most circumstances.

Before tackling GC performance issues, watch this presentation by Attila that illustrates some of our experiences with GC tuning.

In Scala proper, your only tool to mitigate GC problems is to generate less garbage; but do not act without data! Unless you are doing something obviously degenerate, use the various Java profiling tools — our own include heapster and gcprof.

Java compatibility

When we write code in Scala that is used from Java, we ensure that usage from Java remains idiomatic. Oftentimes this requires no extra effort — classes and pure traits are exactly equivalent to their Java counterpart — but sometimes separate Java APIs need to be provided. A good way to get a feel for your library's Java API is to write a unittest in Java (just for compilation); this also ensures that the Java-view of your library remains stable over time as the Scala compiler can be volatile in this regard.

Traits that contain implementation are not directly usable from Java: extend an abstract class with the trait instead.

```
// Not directly usable from Java
trait Animal {
  def eat(other: Animal)
  def eatMany(animals: Seq[Animal) = animals foreach(eat(_))
}

// But this is:
abstract class JavaAnimal extends Animal
```

Twitter's standard libraries

The most important standard libraries at Twitter are Util and Finagle. Util should be considered an extension to the Scala and Java standard libraries, providing missing functionality or more appropriate implementations. Finagle is our RPC system; the kernel distributed systems components.

Futures

Futures have been discussed briefly in the concurrency section. They are the central mechanism for coordination asynchronous processes and are pervasive in our codebase and core to Finagle. Futures allow for the composition of concurrent events, and simplify reasoning about highly concurrent operations. They also lend themselves to a highly efficient implementation on the JVM.

Twitter's futures are *asynchronous*, so blocking operations — basically any operation that can suspend the execution of its thread; network IO and disk IO are examples — must be handled by a system that itself provides futures for the results of said operations. Finagle provides such a system for network IO.

Futures are plain and simple: they hold the *promise* for the result of a computation that is not yet complete. They are a simple container — a placeholder. A computation could fail of course, and this must also be encoded: a

Future can be in exactly one of 3 states: *pending*, *failed* or *completed*.

> **Aside: *Composition***
>
> Let's revisit what we mean by composition: combining simpler components into more complicated ones. The canonical example of this is function composition: Given functions *f* and *g*, the composite function *(g ∘ f)(x) = g(f(x))* — the result of applying *f* to *x* first, and then applying *g* to the result of that — can be written in Scala:
>
> ```
> val f = (i: Int) => i.toString
> val g = (s: String) => s+s+s
> val h = g compose f  // : Int => String
>
> scala> h(123)
> res0: java.lang.String = 123123123
> ```
>
> the function *h* being the composite. It is a *new* function that combines both *f* and *g* in a predefined way.

Futures are a type of collection — they are a container of either 0 or 1 elements — and you'll find they have standard collection methods (eg. `map`, `filter`, and `foreach`). Since a Future's value is deferred, the result of applying any of these methods is necessarily also deferred; in

```
val result: Future[Int]
val resultStr: Future[String] = result map { i => i.toString }
```

the function `{ i => i.toString }` is not invoked until the integer value becomes available, and the transformed collection `resultStr` is also in pending state until that time.

Lists can be flattened;

```
val listOfList: List[List[Int]] = ...
val list: List[Int] = listOfList.flatten
```

and this makes sense for futures, too:

```
val futureOfFuture: Future[Future[Int]] = ...
val future: Future[Int] = futureOfFuture.flatten
```

since futures are deferred, the implementation of `flatten` — it returns immediately — has to return a future that is the result of waiting for the completion of the outer future (**`Future[`**`Future[Int]`**`]`**) and after that the inner one (`Future[`**`Future[Int]`**`]`). If the outer future fails, the flattened future must also fail.

Futures (like Lists) also define `flatMap`; `Future[A]` defines its signature as

```
flatMap[B](f: A => Future[B]): Future[B]
```

which is like the combination of both `map` and `flatten`, and we could implement it that way:

```
def flatMap[B](f: A => Future[B]): Future[B] = {
  val mapped: Future[Future[B]] = this map f
  val flattened: Future[B] = mapped.flatten
  flattened
}
```

This is a powerful combination! With `flatMap` we can define a Future that is the result of two futures sequenced, the second future computed based on the result of the first one. Imagine we needed to do two RPCs in order to authenticate a user (id), we could define the composite operation in the following way:

```
def getUser(id: Int): Future[User]
def authenticate(user: User): Future[Boolean]

def isIdAuthed(id: Int): Future[Boolean] =
  getUser(id) flatMap { user => authenticate(user) }
```

an additional benefit to this type of composition is that error handling is built-in: the future returned from `isAuthed(..)` will fail if either of `getUser(..)` or `authenticate(..)` does with no extra error handling code.

Style

Future callback methods (`respond`, `onSuccess`, `onFailure`, `ensure`) return a new future that is *chained* to its parent. This future is guaranteed to be completed only after its parent, enabling patterns like

```
acquireResource() onSuccess { value =>
  computeSomething(value)
} ensure {
  freeResource()
}
```

where `freeResource()` is guaranteed to be executed only after `computeSomething`, allowing for emulation of the native `try .. finally` pattern.

Use `onSuccess` instead of `foreach` — it is symmetrical to `onFailure` and is a better name for the purpose, and also allows for chaining.

Always try to avoid creating `Promise` instances directly: nearly every task can be accomplished via the use of predefined combinators. These combinators ensure errors and cancellations are propagated, and generally encourage *dataflow style* programming which usually obviates the need for synchronization and volatility declarations.

Code written in tail-recursive style is not subject to stack-space leaks, allowing for efficient implementation of loops in dataflow-style:

```
case class Node(parent: Option[Node], ...)
def getNode(id: Int): Future[Node] = ...

def getHierarchy(id: Int, nodes: List[Node] = Nil): Future[Node] =
  getNode(id) flatMap {
    case n@Node(Some(parent), ..) => getHierarchy(parent, n :: nodes)
    case n => Future.value((n :: nodes).reverse)
  }
```

`Future` defines many useful methods: Use `Future.value()` and `Future.exception()` to create pre-satisfied futures. `Future.collect()`, `Future.join()` and `Future.select()` provide combinators that turn many futures into one (ie. the gather part of a scatter-gather operation).

Cancellation

Futures implement a weak form of cancellation. Invoking `Future#cancel` does not directly terminate the computation but instead propagates a level triggered *signal* that may be queried by whichever process ultimately satisfies the future. Cancellation flows in the opposite direction from values: a cancellation signal set by a consumer is propagated to its producer. The producer uses `onCancellation` on `Promise` to listen to this signal and act accordingly.

This means that the cancellation semantics depend on the producer, and there is no default implementation. *Cancellation is but a hint*.

Locals

Util's `Local` provides a reference cell that is local to a particular future dispatch tree. Setting the value of a local makes this value available to any computation deferred by a Future in the same thread. They are analogous to thread locals, except their scope is not a Java thread but a tree of "future threads". In

```
trait User {
  def name: String
  def incrCost(points: Int)
}
val user = new Local[User]

...

user() = currentUser
rpc() ensure {
  user().incrCost(10)
}
```

`user()` in the `ensure` block will refer to the value of the `user` local at the time the callback was added.

As with thread locals, `Local`s can be very convenient, but should almost always be avoided: make sure the problem cannot be sufficiently solved by passing data around explicitly, even if it is somewhat burdensome.

Locals are used effectively by core libraries for *very* common concerns — threading through RPC traces, propagating monitors, creating "stack traces" for future callbacks — where any other solution would unduly burden the user. Locals are inappropriate in almost any other situation.

Offer/Broker

Concurrent systems are greatly complicated by the need to coordinate access to shared data and resources. Actors present one strategy of simplification: each actor is a sequential process that maintains its own state and resources, and data is shared by messaging with other actors. Sharing data requires communicating between actors.

Offer/Broker builds on this in three important ways. First, communication channels (Brokers) are first class — that is, you send messages via Brokers, not to an actor directly. Secondly, Offer/Broker is a synchronous mechanism: to communicate is to synchronize. This means we can use Brokers as a coordination mechanism: when process `a` has sent a message to process `b`; both `a` and `b` agree on the state of the system. Lastly, communication can be performed *selectively*: a process can propose several different communications, and exactly one of them will obtain.

In order to support selective communication (as well as other composition) in a general way, we need to decouple the description of a communication from the act of communicating. This is what an `Offer` does — it is a persistent value that describes a communication; in order to perform that communication (act on the offer), we synchronize via its `sync()` method

```
trait Offer[T] {
  def sync(): Future[T]
}
```

which returns a `Future[T]` that yields the exchanged value when the communication obtains.

A `Broker` coordinates the exchange of values through offers — it is the channel of communications:

```
trait Broker[T] {
  def send(msg: T): Offer[Unit]
  val recv: Offer[T]
}
```

so that, when creating two offers

```
val b: Broker[Int]
val sendOf = b.send(1)
val recvOf = b.recv
```

and `sendOf` and `recvOf` are both synchronized

```
// In process 1:
sendOf.sync()

// In process 2:
recvOf.sync()
```

both offers obtain and the value `1` is exchanged.

Selective communication is performed by combining several offers with `Offer.choose`

```
def choose[T](ofs: Offer[T]*): Offer[T]
```

which yields a new offer that, when synchronized, obtains exactly one of `ofs` — the first one to become available. When several are available immediately, one is chosen at random to obtain.

The `Offer` object has a number of one-off Offers that are used to compose with Offers from a Broker.

```
Offer.timeout(duration): Offer[Unit]
```

is an offer that activates after the given duration. `Offer.never` will never obtain, and `Offer.const(value)` obtains immediately with the given value. These are useful for composition via selective communication. For example to apply a timeout on a send operation:

```
Offer.choose(
  Offer.timeout(10.seconds),
  broker.send("my value")
).sync()
```

It may be tempting to compare the use of Offer/Broker to SynchronousQueue, but they are different in subtle but important ways. Offers can be composed in ways that such queues simply cannot. For example, consider a set of queues, represented as Brokers:

```
val q0 = new Broker[Int]
val q1 = new Broker[Int]
val q2 = new Broker[Int]
```

Now let's create a merged queue for reading:

```
val anyq: Offer[Int] = Offer.choose(q0.recv, q1.recv, q2.recv)
```

`anyq` is an offer that will read from first available queue. Note that `anyq` is *still synchronous* — we still have the semantics of the underlying queues. Such composition is simply not possible using queues.

Example: A Simple Connection Pool

Connection pools are common in network applications, and they're often tricky to implement — for example, it's often desirable to have timeouts on acquisition from the pool since various clients have different latency requirements. Pools are simple in principle: we maintain a queue of connections, and we satisfy waiters as they come in. With traditional synchronization primitives this typically involves keeping two queues: one of waiters (when there are no connections), and one of connections (when there are no waiters).

Using Offer/Brokers, we can express this quite naturally:

```
class Pool(conns: Seq[Conn]) {
  private[this] val waiters = new Broker[Conn]
  private[this] val returnConn = new Broker[Conn]

  val get: Offer[Conn] = waiters.recv
  def put(c: Conn) { returnConn ! c }

  private[this] def loop(connq: Queue[Conn]) {
    Offer.choose(
      if (connq.isEmpty) Offer.never else {
        val (head, rest) = connq.dequeue()
        waiters.send(head) map { _ => loop(rest) }
      },
      returnConn.recv map { c => loop(connq.enqueue(c)) }
    ).sync()
  }

  loop(Queue.empty ++ conns)
}
```

`loop` will always offer to have a connection returned, but only offer to send one when the queue is nonempty. Using a persistent queue simplifies reasoning further. The interface to the pool is also through an Offer, so if a caller wishes to apply a timeout, they can do so through the use of combinators:

```
val conn: Future[Option[Conn]] = Offer.choose(
  pool.get map { conn => Some(conn) },
  Offer.timeout(1.second) map { _ => None }
).sync()
```

No extra bookkeeping was required to implement timeouts; this is due to the semantics of Offers: if `Offer.timeout` is selected, there is no longer an offer to receive from the pool — the pool and its caller never simultaneously agreed to receive and send, respectively, on the `waiters` broker.

Example: Sieve of Eratosthenes

It is often useful — and sometimes vastly simplifying — to structure concurrent programs as a set of sequential processes that communicate synchronously. Offers and Brokers provide a set of tools to make this simple and uniform. Indeed, their application transcends what one might think of as "classic" concurrency problems — concurrent programming (with the aid of Offer/Broker) is a useful *structuring* tool, just as subroutines, classes, and modules are — another important idea from CSP.

One example of this is the Sieve of Eratosthenes, which can be structured as a successive application of filters to a stream of integers. First, we'll need a source of integers:

```
def integers(from: Int): Offer[Int] = {
  val b = new Broker[Int]
  def gen(n: Int): Unit = b.send(n).sync() ensure gen(n + 1)
  gen(from)
  b.recv
}
```

`integers(n)` is simply the offer of all consecutive integers starting at `n`. Then we need a filter:

```
def filter(in: Offer[Int], prime: Int): Offer[Int] = {
  val b = new Broker[Int]
  def loop() {
    in.sync() onSuccess { i =>
      if (i % prime != 0)
        b.send(i).sync() ensure loop()
      else
        loop()
    }
  }
  loop()

  b.recv
}
```

`filter(in, p)` returns the offer that removes multiples of the prime `p` from `in`. Finally, we define our sieve:

```
def sieve = {
  val b = new Broker[Int]
  def loop(of: Offer[Int]) {
    for (prime <- of.sync(); _ <- b.send(prime).sync())
      loop(filter(of, prime))
  }
  loop(integers(2))
  b.recv
}
```

`loop()` works simply: it reads the next (prime) number from `of`, and then applies a filter to `of` that excludes this prime. As `loop` recurses, successive primes are filtered, and we have a Sieve. We can now print out the first 10000 primes:

```
val primes = sieve
0 until 10000 foreach { _ =>
  println(primes.sync()())
}
```

Besides being structured into simple, orthogonal components, this approach gives you a streaming Sieve: you do not a priori need to compute the set of primes you are interested in, further enhancing modularity.

Acknowledgments

The lessons herein are those of Twitter's Scala community — I hope I've been a faithful chronicler.

Blake Matheny, Nick Kallen, Steve Gury, and Raghavendra Prabhu provided much helpful guidance and many excellent suggestions.

---

1. Yourkit is a good profiler [back]
2. From Finagle's heap balancer [back]