Scala Tutorial

Jan Macháček — jan.machacek@bamtechmedia.com — @jmachacek

June 15, 2018

1 Syntax crash-course

Scala's syntax follows the syntax of other C-like languages, though–like Pascal–the type specification follows an identifier. Scala's `class` behaves exactly like Java's `class`, and its syntax is not wildly different. (See Listing 1.)

```
Class declaration means the same thing as Java; constructor parameters are specified in the block immediately following the class name.
Note that the types follow the identifier; instead of String constructorParam1 Scala uses constructorParam1: String
class MyClass(constructorParam1: String, constuctorParam2: Int) {

  Methods begin with the keyword def, followed by name and parameters. The return type follows similar pattern; Unit means void.
  def execute(methodParam1: List[Int]): Unit = { The body of the method follows the equals sign. }

}

↓ Constructing instances uses the typical new keyword...
new MyClass("foo", 42).execute(List(1, 2, 3))
                       ↑ Method invocation is exactly like Java's
```

Listing 1: Classes and methods

Most of this syntax is familiar and unsurprising; the only thing that might feel odd is the square bracket for "generics" in `List[Int]`: in Java, this would be written as `List<Int>`. This is part of Scala's legacy. A long, long time ago, XML was very exciting; and Scala allowed (and still allows!) XML literals. These XML literals use the angle brackets. This meant that a different symbol had to be used for type parameters. Because the square bracket is used for type parameters, array indexing is also done using regular parentheses[1].

Interfaces use the `trait` keyword in Scala; their usage and features are similar to `interface` in Java (particularly Java 8 which adds default implementations). It is possible to make anonymous implementations of a `trait`, as well as to implement it in ordinary `class` es. (Viz Listing 2.)

```
Apart from the trait keyword, the syntax is unremarkable
trait ReportGenerator {
  Interface methods are public and abstract; they specify parameters and return type
  def generate(userId: Int): Array[Byte]
}

A trait can be implemented in a class using the keyword extends. Additional traits to be implemented use the with keyword.
class ReportGeneratorImpl extends ReportGenerator with Cloneable {
  Instead of the @Override annotation Scala uses the override keyword.
  override def generate(userId: Int): Array[Byte] = ...
  override def clone(): AnyRef = ...
}

It is also possible to make an anonymous implementation of a trait using the new keyword.
new ReportGenerator {
  override def generate(userId: Int): Array[Byte] = ...
}
```

Listing 2: Traits

It is worth noting that there is no special syntax for *array of X* in Scala. Instead, it uses `Array` with the specified type parameter. (So, Java's `byte[]` becomes `Array[Byte]`, `User[]` becomes `Array[User]`, and so on.) Also notice the `AnyRef` in the implementation of the `clone()` method–it is equivalent to `java.lang.Object`.

So far, there are no major surprises: classes and interfaces work just like most other languages, constructor, method, and parameter definition also looks fairly ordinary. The syntax for functions (finally!) uses parameters (each with its type following the `:` symbol, if needed) followed by fat arrow ⇒, and the body of the function. Usage is the same as Java 8; and Scala's collections library contains concepts that are fairly similar.

```
Double every Int in the list
List(1, 2, 3, ...).map { (x: Int) ⇒ 2 * x }

Scala can infer the type of x, so there's no need to specify it
List(1, 2, 3, ...).map { x ⇒ 2 * x }

An alternative syntax uses paretheses in place of brackets
List(1, 2, 3, ...).map(x ⇒ 2 * x)

It is possible to avoid having to declare the parameter x, and use _ instead. Scala compiler replaces the every occurrence of _
with a fresh parameter declaration.
List(1, 2, 3, ...).map(2 * _)

Finally, the "shape" of the function * in the Int instance has the right type, so it can be used directly.
List(1, 2, 3, ...).map(2.*)
```

Listing 3: Fields & variables

[1] I know, it looks like VisualBasic or Fortran (formerly FORTRAN)!

The last forms $2 * \_$ and $2.*$ are somewhat unusual and worthy of further explanation. The Scala compiler mechanically translates each underscore in the function body into a parameter (and "replaces" the underscore with that parameter); $2 * \_$ is translated into $p\_0 \Rightarrow 2 * p\_0$, and this satisfies the type that the map function expects. The type of the function $*$ in the Int type is $Int \Rightarrow Int$ because it is defined as $\textbf{\textit{def}}\ *(that\colon\ Int)\colon\ Int$: one can read this as "when applied to a value of type Int, a value of type Int remains". The type of $2.*$ is therefore $Int \Rightarrow Int$, just like the type of $2 * \_$, or $\_ * 2$, or $x \Rightarrow x * 2$ –assuming the parameter can be inferred to be Int. Consequently, it is possible to leave out the underscore in the map function and only write $List(1,\ 2,\ 3,\ \ldots).map(2.*)$ [2].

Finally, fields (and variables) use the keywords $\textbf{\textit{val}}$ and $\textbf{\textit{var}}$. The first declares an immutable variable (and a getter if field), the second declares a mutable variable (and a getter and setter if field); see Listing 4.

```
Field definitions have to specify initial value; use _ for default value.
class User {
  var id: Long = _
  var name: String = _
  var dob: Date = _
}

val user = new User
```

Listing 4: Fields & variables

This code is terrible! The default value for reference types ($AnyRef$ s) is $\textbf{\textit{null}}$, and variants of *zero* for primitive types. The $name$ and dob fields in the $user$ instance are $\textbf{\textit{null}}$, and id is $0L$. What's worse, the class has setters for these fields, and it's possible to invoke them. The syntax is somewhat nicer–it looks like plain assignment using the $=$ operator, though it is actually invoking a setter–but the semantics of the code in Listing 5 is terrible.

```
Field definitions have to specify initial value; use _ for default value.
class User {
  var id: Long = _
  var name: String = _
  var dob: Date = _
}

val user = new User
user.id = 5
user.name = "Foo Quux"
user.dob = ...
```

Listing 5: Fields & variables II

While syntactically valid code, it is very confusing. The $user$ variable is declared immutable, yet it is possible to invoke its setters. The equivalent Java code would declare the $user$ variable $\textbf{\textit{final}}$, but then still use the setters to mutate it.

[2] This might be somewhat familiar to Java 8 programmers with method references with the double colons

2 Killer features

The syntax (and its application) so far looks just like Java with less typing. There must be something else that makes it worth leaving the creature comforts of Java. The following points are also the code that is most likely to be encountered in typical Scala codebases. Typical here refers to code that *uses* Scala and its ecosystem of libraries; it does not refer to the code that might be found in the bowels of the libraries themselves.

- everything is an expression
- case classes
- pattern matching
- for comprehensions
- implicits
- rich type system

2.1 Everything is an expression.

In Java, C, and similar languages, there are statements and expressions. Statements do not have value, they are typical control-flow constructs. For example, in Java, `if (a == b) X else Y` cannot be assigned to a variable, because `if-then-else` is a statement. In Scala, everything except definitions of identifiers (i.e. variables, functions, classes, etc.) yields a value that can be assigned to a variable. There are languages where "the value of a function is the value of the last evaluated statement," and that's a good starting point for thinking about expressions and their values in Scala. Listing 6 shows a very Java-esque implementation of a method that checks whether a date is the user's birthday.

```
class User {
  var dob: Date = _

  This is a very Java-style implementation; though to make it a bit more interesting, the asOf parameter
  has a default value set to "now".
  def hasBirthday(asOf: Date = new Date()): Boolean = {
    if (dob ≠ null && asOf ≠ null) {
      val c1 = Calendar.getInstance()
      val c2 = Calendar.getInstance()
      c1.setTime(dob)
      c2.setTime(asOf)
      if (c2.get(Calendar.MONTH) == c1.get(Calendar.MONTH)) {
        c2.get(Calendar.DAY_OF_MONTH) == c1.get(Calendar.DAY_OF_MONTH)
      } else false
    } else false
  }

}
```

Listing 6: Expressions

The `java.util.Date` code[3] muddies the explanation; removing it yields code in Listing 7.

```
class User {
  var dob: Date = _

  def hasBirthday(asOf: Date = new Date()): Boolean = {
    if (cond₁) {
      val c1 = Calendar.getInstance()
      val _ = c1.setTime(dob)
          ↑ imagine that the compiler creates a variable for every expression that isn't the last one in a block.
      ...
      if (cond₂) { Expression that evaluates to Boolean } else { Expression that evaluates to Boolean }
      ↑ because both then and else branches yield Boolean, the value of the entire if expression is Boolean.
    } else false
    ↑ because both then and else branches yield Boolean, the value of the entire if expression is Boolean.
      consequently, the value of hasBirthday is whatever the outer if expression evaluates to.
  }

}
```

Listing 7: Expressions

[3]This project *refuses* to include `DateUtils`!

Notice the pattern where everything that follows the = sign can be and is evaluated. This applies to variable declarations `val x: Tpe = ...;` `def x(): Tpe = ...`. Simple expressions do not have to be surrounded by curly braces. It is not surprising that `val x = 3; val msg = if (x % 2 == 0) "Even" else "Odd"` results in the declaration of a variable *msg* of type *String*, whose value is *"Odd"*, because the value of *x* is *3*. It might be somewhat surprising that the same syntax applies to methods: `def m(x: Int): String = if (x % 2 == 0) "Even" else "Odd"` is just as legal Scala code.

Exercise: Fizz Buzz

Create the "Fizz Buzz" function that takes an *Int* and evaluates to a *String* such that

- if the input is divisible by *3*, the output is *"Fizz"*
- if the input is divisible by *5*, the output is *"Buzz"*
- if the input is divisible by *3* and *5*, the output is *"FizzBuzz"*
- for all other input values, the output is the *String* representation of the input

In *src/main/scala*, in the *com.acme* package, create a new class *FizzBuzzMain* and complete the body of the *fizzBuzz* function.

```
object FizzBuzzMain extends App {
  def fizzBuzz(i: Int): String = ...

  println(fizzBuzz(1))    | "1"
  println(fizzBuzz(3))    | "Fizz"
  println(fizzBuzz(30))   | "FizzBuzz"

}
```

Listing 8: Fizz Buzz

2.2 Case classes.

A case class is just like a class in that it is a container for data and methods, but the fields it contains only have getters. (Immutability only goes as far as immutability of the references. Even in Scala, immutability without any additional code is equivalent to using ***final*** in Java.) Nevertheless, case classes are fantastically convenient to define data structures. Consider the *Person* case class defined in Listing 9.

```
case class Person(firstName: String, lastName: String, age: Int)
```

Listing 9: Case class *Person*

This is all it takes to define an immutable structure (with the T&Cs from above) with the fields *firstName*, *lastName*, and *age*; but also with appropriate *toString*, *hashCode*, and *equals* implementations. These automatically generated implementations delegate to the *toString*, *hashCode*, and *equals* methods of all the fields, in the order in which they are specified.

To create an instance of a case class, do not use the ***new*** keyword; instead, write the parameters directly after the case class name, as shown in Listing 10.

```
Notice that there is no new keyword; the parameter values are applied directly after the case class name.
val fq = Person("Foo", "Quux", 42)

To access the fields, use the familiar . notation.
fq.firstName   | "Foo"

Invoking the toString method prints a reasonable representation of the case class.
fq.toString   | "Person(Foo,Quux,42)"

It is possible to vary the order of the parameters if the parameter names are also specified. This can help readability.
val fb = Person(lastName = "Baz", firstName = "Foo", age = 50)
fb.toString   | "Person(Foo,Baz,50)"

Equality is implemented by delegation to the parameters' hashCode and equals methods.
Person("Foo", "Quux", 42) == fq     | true, even though they are different instances.
fq == Person("Foo", "Quux", 42)     | true, even though they are different instances.
```

Listing 10: Using case class *Person*

The consciseness of Scala's syntax is beginning to show. It would have been much more cumbersome to implement all this (including correct *hashCode*, *equals*, and reasonable *toString*) in Java. Even with correct implementations, it would not have been possible to use == to test for instance equality.

Exercise: Using case classes and functions

Create a new case class *Person* file in *src/main/scala/com.acme* from Listing 9. Then use *List.fill(N)(Person(...))* and useful methods in *scala.util.Random* to generate N random *Person* instances. Assign those to a variable.

- find the oldest person in the list. (Hint: use *map*, *max*, and *find* functions on the *List[Person]*.)
- find the 10 youngest people in the list. (Hint: use *sortBy* or *sortWith*, followed by *take*.)

```
...
object PersonMain extends App {

  val people = List.fill(100) ...
  println(people) |🖨 [Person(...),Person(...),...]

  val oldest = people.map...

}
```

Listing 11: Pattern matching

Case classes are also called *product* types. The word product refers to the possible number of values that a case class can hold. Take ***case class*** *ABC(a: Boolean, b: Boolean, c: Boolean)*: there are eight possible values: *ABC(**false**, **false**, **false**)*, ..., *ABC(**true**, **true**, **true**)*. The value 8 is the result of *multiplying* the possible values of all parameters. *Boolean* s have two values; three *Boolean* values yield $2 \times 2 \times 2$ possible values of the *ABC* type. Similarly, ***case class*** *IB(i: Int, b: Boolean)* has 4294967295×2 possible values.

Imagine for a moment that *Boolean* is not a built-in primitive. It would be defined as the *sum* of two [degenerate] products of 1 value: ***sealed trait*** *Boolean*; ***case object*** *True* ***extends*** *Boolean*; ***case object*** *False* ***extends*** *Boolean*. The products *True* and *False* have exactly one value; consequently, the sum type *Boolean* has $1 + 1$ possible values. Languages like Haskell[3], F#[2], and others have convenient syntax for sum types; see Listing 12.

```
Defines the data type Boolean as a sum of two products, but each product only has one value.
data Boolean = True
             | False

Defines the subscription type as either a "one-off" that carries a particular end-date, or a recurring
subscription that carries the start date and the billing period.
data Subscription = OneOff Date
                  | Recurring Date Period


sealed trait Boolean
case object True extends Boolean
case object False extends Boolean

sealed trait Subscription
case class OneOff(endDate: Date) extends Subscription
case class Recurring(startDate: Date, period: Period) extends Subscription
```

Listing 12: Sum types

Finally, using the keyword ***sealed*** specifies that the trait cannot be extended outside the source file containing its definition. This allows the Scala compiler to verify that pattern matches (using the ***match*** ... ***case*** construct) cover every possible case, reporting a warning if a case is missed.

Sums of products are very useful way to express alternatives that might not have anything in common other than being of the given type, without clumsy *instanceof* checks with pattern matching.

2.3 Patten matching.

A pattern match is a way to check that a value has the right "shape", and to pull out some or all values from that shape. Think of the simplest pattern matching expression as Java's ***switch*** statement in Listing 13 that can compare arbitrary values against "templates"; and if there is a match, be able to pull out elements of the matched expression.

This is the equivalent of Java's `switch` statement, with multi-case and a default case.

```
Random.nextInt(10) match {
  // Matches values 0..3
  case 0|1|2|3 =>
  // Matches only value 6
  case 6 =>
  // Matches all other values
  case _ =>
}
```

Listing 13: Pattern matching

It is possible to match on much more complicated structures; such as tuples. A tuple is a collection of a specific length with elements of arbitrary types. For example `(1, "foo")` is a tuple of 2 elements of types `Int` and `String`; `("foo", "bar", 1.3)` is a tuple of 3 elements of types `String`, `String`, and `Double`. It is possible to pattern match on those, as shown in Listing 14

```
import scala.util.Random
(Random.nextString(Random.nextInt(16)), Random.nextInt(100)) match {
  // Matches tuples with any first element, and second element == 10.
  case (s, 10)                        => // The first element is accessible as s here.

  // Matches only the ("Lucky?", 42) tuple.
  case ("Lucky?", 42)                 => // None of the elements are accessible here.

  // Matches any tuple as long as the first element is longer than 5 characters and second element is less than 10.
  case (x, y) if x.length > 5 && y < 10 => // Both first and second elements are accessible as x and y.

  // Matches any value
  case _                              => // None of the elements are accessible here.
}
```

Listing 14: Pattern matching II

As useful as tuples are for quick ad-hoc structures, the real deal in pattern matching are case classes. Just like matching on a tuple, it is possible to match on case classes and extract their fields as needed. In addition to using the ***match*** keyword followed by ***case*** s, pattern matching also applies to declarations of variables in Listing 15.

```
// Pattern matches on the right-hand side, pulls out the first parameter as val first: String.
val Person(first, _, _) = Person("Foo", "Bar", 99)

// Pattern matches on the right-hand side, comparing the value of the first parameter to be equal to in-scope variable first. (Notice the backticks.)
val Person(`first`, _, _) = Person("Foo", "Bar", 99)

// Because there are no alternatives, if the val pattern match fails, it raises an exception.
val Person(`first`, _, _) = Person("Fooz", "Bar", 99)
```

Listing 15: Pattern matching III

The pattern match that raises an exception if it fails is often convenient to use in tests.

Exercise: Using case classes and pattern matching

Build an numerical expression evaluator; the evaluator should support binary operators $+, -, \times, /$ and support arbitrary nesting. It should be able to evaluate, for example $4 + 3(5/(12 - 7))$. To make things simpler, it is not necessary to include error reporting (division by zero, for example), and it is not necessary to provide a way to turn a `String` into an expression.

```
// The sum type Expr's subtypes define the "operations" we support.
sealed trait Expr
// The different cases need to support our operations and constants.
case class Plus(left: Expr, right: Expr) extends Expr
...
case class Const(const: Int) extends Expr

// It will be convenient to package our evaluator in its own module.
object Evaluator {
  // Hint: you will need to pattern-match on the different Exprs.
  def eval(expr: Expr): Int = expr match {
    case Const(x) => x
    ...
  }
}

object EvaluatorMain extends App {
  val e = Evaluator.eval(Plus(Plus(Const(4), Const(5)), Const(3)))
  println(e)  // 12
}
```

Listing 16: Expression evaluator

Exercise (extra): logical expression simplifier

Build a logical expression simplifier that reduces the number of logical operations to be performed, and can spot tautologies and contraditions. For example, $(a \wedge b) \vee (a \wedge b)$ should simplify to just $a \wedge b$; $(a \vee \neg a)$ should simplify to $\top$ (tautology; always true); $(a \vee \neg a) \wedge (a \oplus a)$ should simplify to $\perp$ (contradiction; always false).

```
The sum type LExpr defines the different logical expressions we support.
sealed trait LExpr

The different cases need to support our operations and constants.
case class LAnd(left: LExpr, right: LExpr) extends LExpr
case object Contradiction extends LExpr
...
case class Var(named: String) extends LExpr

Just like the evaluator, package the simplifier in its own module.
object Simplifier {
  Hint: you will need to pattern-match on the different LExprs.
  def simplify(expr: LExpr): LExpr = expr match {
    ...
  }
}

object SimplifierMain extends App {
  val se = Simplifier.simplify(
    LAnd(
      LOr(Var("a"), LNot(Var("a"))),
      LXor(Var("a"), Var("a"))
    )
  )
  println(se)  | "Contradition"
}
```

Listing 17: Logical expression simplifier

2.4 For comprehensions

The ***for*** keyword in Scala can be used in the familiar "for loop" style, but it is actually much more powerful construct. Java has two styles of the ***for*** loop: the old-school one with ***for*** *(**init**; condition; step) { body }* as well as the "for-each" style ***for*** *(element : iterable) body*. Apart from the convenience of the for-each version, both loops ultimately express only iteration. Scala also has two basic styles of the ***for*** expression.

- ***for*** *(element ← iterable) body*. This style of the for expression evaluates iterates over all elements in the given *iterable*, and applies *body* to each element; each element is accessible using the name *element* in the *body*. It is the same as writing *iterable.foreach(element ⇒ body(element))*.
- ***for*** *(element ← iterable)* ***yield*** *body*. This style's value has the same type as *iterable*, but the elements of the evaluated *iterable* are the results of applying *expression* to each one. It is the same as writing *iterable.map(element ⇒ body(element))*

Writing ***for*** *(x ← List(1, 2, 3, ...))* ***yield*** *x * 2* might be more natural way to write *List(1, 2, 3, ...).map(x ⇒ x * 2)*; and in simple expressions, the choice between using the *map*, *flatMap*, and *filter* versus using the ***for*** expression can be left to one's taste (or the project's code-style). The real power of Scala's for expressions comes from the fact that it is possible to map & flatten, map, and filter multiple values together.

The rule to be able to decipher and write for expressions in Scala is to remember that:

- the first expression inside ***for*** has to be *element ← value*, where *value* contains the method *flatMap*,
- subsequent ← operation also means *flatMap*,
- subsequent = operation means *map*,
- subsequent ***if*** statement means *filter*,
- if the expression ends with ***yield*** *body*, the value of the ***for*** expression is the value of the ***yield*** *body* represented in the same type as the first value on the right of ←,
- if the expression does not end with ***yield*** but just *body*, the value of the ***for*** expression is *Unit*.

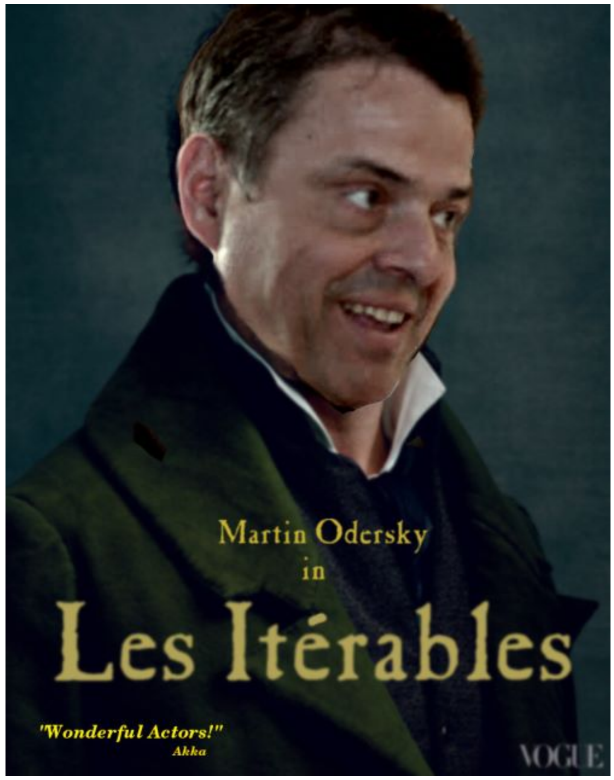

Figure 1: Les Itérables

This is indeed so much power that it inspired a meme in Figure 1 and code in Listing 18 with examples of different styles of ***for*** .

```
val listOfTen = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

↓ types to Unit, because this is the variant without yield body.
for {
 x ← listOfTen
 y ← listOfTen
} println(x, y)    | 🖶 the Cartesian product of the two lists: (1, 1), (1, 2), ..., (1, 10), ..., (10, 9), (10, 10)

↓ types to List[(Int, Int)], because the original iterable was List, and the yield body evaluates to type (Int, Int); contains 100 elements.
for {
  x ← listOfTen
  y ← listOfTen
} yield (x, y)

↓ types to List[(Int, Int)] ; contains 50 elements because of the if filter.
for {
  x ← listOfTen
  if x % 2 == 0
  y ← listOfTen
} yield (x, y)


↓ types to List[(Int, List[Int])]; contains only 10 elements.
for {
  x ← listOfTen
  y = listOfTen
} yield (x, y)
```

Listing 18: Various for expessions

In fact, the word *iterable* is somewhat misleading; the values used in the ***for*** expression only have to contain the functions *map* , *flatMap* , and *filter* . The ***for*** expression is actually a *syntactic sugar* that translates ← to *flatMap* , = to *map* , and ***if*** to *filter* [4]. This means that the ***for*** expression can be used on values such

[4] Actually–and for efficiency–the de-sugaring prefers to use *withFilter* instead of *filter* , which avoids having to create intermediate iterables.

as *Option*, *Either*, ..., even your own types, as long as the requirements for mapping, mapping & flattening, and filtering are satisfied. Even better, the compiler only checks that the requirements are satisfied as they are needed.

Option and Either. The types *Option*[5] and *Either* are a part of the Scala standard library; they are used to pack "one or none" value (*Option*) and "exclusive or of two values" (*Either*). The type *Option[A]* is a sum of two products: *Some[A](value: A)* and *None*; what these values represent is self-evident. The type *Either[L, R]* is a sum of two products: *Left[L](value: L)* and *Right[R](value: R)*. By convention, the value *Right[R]* is used to indicate "the right value–success", and the value *Left[L]* is used to indicate failure. Unlike *None*, which carries no further information other than "missing", the value *Left[L](value: L)* carries a value; this value is typically some kind of error message. Crucially, both *Option* and *Either* include the *flatMap* functions, which means that they can be used in a ***for*** expression. Table 1 outlines the behaviour of the *map* and *flatMap* in *Option* and *Either*.

	map	*flatMap*	*filter*
Option[A]	*f: A ⇒ B* *Some(a) ⇒ Some(f(a))* *None ⇒ None*	*f: A ⇒ Option[B]* *Some(a) ⇒ f(a)* *None ⇒ None*	*f: A ⇒ Boolean* *Some(a)* ***if*** *f(a) ⇒ Some(a)* *_ ⇒ None*
Either[L, R]	*f: R1 ⇒ R2* *Right(r) ⇒ Right(f(r))* *Left(l) ⇒ Left(l)*	*f: R1 ⇒ Either[L, R2]* *Right(r) ⇒ f(r)* *Left(l) ⇒ Left(l)*	– – –

Table 1. map and flatMap

Putting it simply once *Option* becomes *None* it stays *None*; once *Either* becomes *Left* it stays *Left*. Thanks to *None* not carrying any value, *Option* can implement the *filter* method; because *Left* contains a value it cannot contain the *filter* method. (In case the filtering function returned ***false***, what would the *value* on the *Left* be?) Nevertheless, the concept of filtering is useful, it only needs to be extended a little; so *Either[L, R]* contains the *filterOrElse(predicate: R ⇒ Boolean, zero: L)* function. If the predicate returns ***false***, the function evaluates to *Left* carrying the value of the *zero* parameter.

Exercise: Error reporting for the evaluator

Improve the evaluator so it doesn't throw exception on division by zero. Instead of relying on exceptions, use *Either* as the returned type. Use plain *String* for the error type, keep *Int* as the evaluated type. In other words, the *eval* function will return *Left(e)* (where *e*'s type is *Error*) in case of errors; and *Right(x)* (where *x*'s type is *Int*). Make the most of the ***for*** expression.

```
It will be useful to package the PML conversions in their own module.
object Expr {
  The type alias makes the return type of eval even more expressive.
  type Error = String

  def eval(expr: Expr): Either[Error, Int] = {
    case Div(l, r) ⇒
      for {
        l ← eval(l)
        r ← eval(r)
        ↑ this doesn't quite solve it; the value of r can still be Right(0).
      } yield l / r
    ...
  }

}
```

Listing 19: Expression evaluator

Hint: recall the *Either.filterOrElse* function; it applies the predicate on values on the right; if the predicate fails, it returns the value of the second parameter on the left. For example, *Right(5).filterOrElse(_ < 5, "Should␣be␣more␣than␣5")* evaluates to *Left("Should␣be␣more␣than␣5")*.

[5] Java includes *Optional*, C++ *std::optional*, Swift *Optional*; the only sore thumb is Haskell with *Maybe*. Nevertheless, all languages include the concept of *flatMap*, *map*, and *filter* on optionals.

2.5 Implicits

Implicits are one of the key features of Scala. As the word suggests, implicit parameter values are supplied "automatically" by the compiler looking for the appropriate values in the current *implicit scope*[6]. The look-up ignores the parameter names, it only cares about the types. Implicits also apply when the compiler encounters an identifier on a value of some type that the type does not implement. In that case, the compiler will follow the same implicit scoping rules to find an implicit conversion that turns the given type into another type that contains the identifier.

```
object SimpleImplicitsMain extends App {
  Defines an implicit conversion from a String into LoudString, which adds the method LOUD.
  implicit class LoudString(s: String) {
    def LOUD: String = s.toUpperCase() + "!!"
  }

  Define the implicit value of type String.; notice the implicit application of the LoudString.LOUD.
  implicit val completelyArbitraryName: String = "Hello, world".LOUD

  The sayHello1 method has single implicit parameter list, with one parameter of type String.
  def sayHello1(implicit greeting: String): Unit = println(greeting)

  The sayHello2 method has two parameter lists, one empty, and one implicit with one parameter of type String.
  def sayHello2()(implicit greeting: String): Unit = println(greeting)

  sayHello1       | to apply, omit the implicit parameter list, leaving just sayHello1.
  sayHello2()     | to apply, omit the implicit parameter list, leaving just sayHello2().
}
```

Listing 20: Simple implicits

Defining implicit *String* s demonstrates the principle, but it is rather useless because values of type *String* are so ubiquitous in typical programs. Different types are sometimes more useful; consider a *javax.sql.Connection* which might be passed implicitly to various data access code to avoid having to do too much typing. The ability to turn values into values that contain identifiers that are not available on the original types are also quite useful. The real power, though, comes from the fact that the implicit resolution does not stop at the first step. The compiler follows all possible paths, using as many conversions as necessary, to get the code to compile.

Pimp my library. Providing implicit conversions from existing types to other types that contain useful or convenient identifiers is called *pimp my library*. It is a fancy name for the adapter pattern[1], but implicits provide the convenience to avoid having to create instances of the adapters manually. It is a useful way to construct domain-specific languages, particularly when combined with Scala's ability to call single parameter method without dots and braces, as though it were an operator.

```
implicit class RichDouble(x: Double) {
  def ^(y: Double): Double = math.pow(x, y)
}

val x: Double = 42
x ^ 2  | Computes the second power of x by essentially doing new RichDouble(x).^(2).
```

Listing 21: Pimp my library

[6]The implicit scope is somewhat complex; for now, it will be sufficient to remember that implicit scoping is similar to regular visibility scoping

Exercise: Pimp my library

Build a DSL that allows the expression evaluator to be written using natural-looking Scala code–though sadly not using the standard `+`, `-`, `*`, `/` operators, but using our own `plus`, `minus`, `mult`, `div` instead. The expression `5 plus 10` should evaluate to `Plus(Const(5), Const(10))`. The type `Int` and `Expr` do not contain the methods `plus`, ..., `div`; it will be necessary to implement implicit conversions (use `implicit class` for convenience) that contains those methods.

```
It will be useful to package the PML conversions in their own module.
object Expr {
  Two conversions will be necessary: Int → Expr, Expr → RichExpr. The two P.M.L. implicit classes
  will contain the same functions; the only exception is that the one for Int will need to box the given
  Int into Const.
  implicit class RichInt(x: Int) {
    def plus(that: Expr): Expr = Plus(Const(x), that)
    ...
  }
  implicit class RichExpr(x: Expr) {
    def plus(that: Expr): Expr = Plus(x, that)
    ...
  }

}

object EvaluatorMain extends App {
  import Expr._
  val e = Evaluator.eval((5 plus 10) minus 8)
  println(e)
}
```

Listing 22: Expression evaluator

Hints:

- because the methods `plus(r: Expr): Expr`, ..., `div(r: Expr): Expr` would be the same, consider defining them in a trait (`trait ExprOps` would be a jolly good name!) that is then mixed into the implicit classes, where the trait defines abstract member `self: Expr`, which is then used as the left-hand side in the `Expr` data constructors; the right-hand side coming from the parameter of the method.

- implementing a class that adds the methods `plus`, ..., `div` to `Int` will allow for code such as `10 plus Const(5)`, but it will not compile `10 plus 5`, because there is no function `plus`, ..., `div` that takes an `Int`. The preferred option is to add those over implicit conversion from `Int` to `Expr`: `implicit def intToExpr(i: Int): Expr` is generally frowned-upon because it allows silent and potentially very powerful conversions.

If you are feeling inventive or perhaps mischievous, pick Unicode identifiers for the boring ASCII method names in the DSL. Resurrect APL by using ÷ instead of `div`!

Type classes. Type class is a concept from Haskell[3]; it is a definition of methods that can be implemented for some type. A type class is therefore an interface parametrized by a type; the implementations are instances of this interface for some types. Suppose the expression evaluator needs to be able to evaluate not just `Int`s, but also other number-like values. The only change to its structure is an interface that implements the addition, subtraction, multiplication, and division for *some* number-like type; see Listing 23.

```
Define the typeclass that defines the methods that instances for the type A must implement.
trait NumberLike[A] {
  def plus(x: A, y: A): A
  ...
  def div(x: A, y: A): A
}

sealed trait Expr
...
Modify the Const data constructor to take any type A instead of the concrete Int.
case class Const[A](a: A) extends Expr

object Evaluator {

  The eval function now needs to take the interface that implements the number-like behaviour for type A.
  def eval[A](expr: Expr, numberLike: NumberLike[A]): Either[Error, A] = expr match {
    case Plus(l, r) ⇒
      for { l ← eval(l, numberLike); r ← eval(r, numberLike) } yield numberLike.plus(l, r)
    ...
```

```
    case Const(a: A) ⇒ a
  }

}

object EvaluatorMain extends App {
  import Expr._
  import Evaluator._

  val expr = 5 plus 10
  eval(expr, new NumberLike[Int] {
    def plus(x: Int, y: Int): Int = x + y
    ...
    def div(x: Int, y: Int): Int = x / y
  })  | evaluates to Right(15).

}
```

Listing 23: Expression evaluator without typeclasses

This is rather tedious to write, especially all the points where the $numberLike$ instance has to be passed around to the recursive calls of $eval$. Moving it to the implicit parameter list makes the code cleaner in Listing 24.

```
trait NumberLike[A] { ... }

object Evaluator {

  def eval[A](expr: Expr)(implicit N: NumberLike[A]): Either[Error, A] = expr match {
    case Plus(l, r) ⇒ for { l ← eval(l); r ← eval(r) } yield N.plus(l, r)
    ...
    case Const(a: A) ⇒ a
  }

}

object EvaluatorMain extends App {
  import Expr._
  import Evaluator._
  val expr = 5 plus 10
  eval(expr)(new NumberLike[Int] {...})  | evaluates to Right(15).
}
```

Listing 24: Expression evaluator with typeclasses-ish

This is an improvement, though it is still annoying to have to explicitly specify the value of the $numberLike$ parameter in the implicit (sic!) parameter list in $EvaluatorMain$. The Scala compiler is able to find the implicit value, if one is available in the implicit scope; all that remains to be done is to provide an *instance of the* $NumberLike$ *typeclass for the type* Int, and any other required types. (Viz Listing 25.)

```
trait NumberLike[A] { ... }

object EvaluatorMain extends App {
  import Expr._
  import Evaluator._

  implicit object IntNumberLike extends NumberLike[Int] {
    def plus(x: Int, y: Int): Int = x + y
    ...
  }

  Complex numbers are numbers too!
  case object Complex {
    val e: Complex = ...
    val pi: Complex = ...
    val i: Complex = ...
  }
  case class Complex(re: Double, im: Double) {
    def +(rhs: Complex): Complex = ...
    def ^(rhs: Complex): Complex = ...
  }
  implicit object ComplexNumberLike extends NumberLike[Complex] {
    def plus(x: Complex, y: Complex): Complex = x + y
    ...
  }

  And now the evaluator works for any type A that lies in the NumberLike typeclass; i.e. where there is an in-scope
  way to access or follow steps to create instances of NumberLike[A].
  eval[Int](5 plus 10)              | evaluates to Right(15).
  eval[Complex](e ^ (pi * i))  | evaluates to Right(-1).

}
```

Listing 25: Expression evaluator with typeclasses

Exercise: Flexible evaluator

Allow the expression evaluator to operate on any number-like types, not just `Int` s. Instead of relying on inheritance, forcing all users of the evaluator to evaluate values conforming to some trait (with the number-crunching methods `+`, `-`, `*`, `/`), use parametric polymorphism and type classes. The `Const` data constructor will need to take any type instead of `Int` ; `RichInt` implicit class will need to pimp any type, not just `Int` ; the `eval` method will also need to be generic, but will need to (implicitly) require a typeclass: use the `Fractional` provided by the standard library, or define your own `NumberLike` typeclass. (The Scala standard library defines `Fractional` and `Integral` type classes, which contain the division operation whereas `Numeric` typeclass does not. Pick one, the limitation that not all types are fractional or integral. "Dotty will fix this!")

```
object Expr {

  trait ExprOps {
    def self: Expr
    def plus ...  This will need to take generic type A.
    ...
  }

  implicit class RichA ...  This will need to take generic type A.
  implicit class RichExpr(val self: Expr) extends ExprOps

}

sealed trait Expr
...
case class Const ...  This will need to take generic type A.

object Evaluator {

  It will be convenient to use the full implicit syntax here
  def eval[A](expr: Expr)...: Either[Error, A] = expr match {
    ...
  }

}

object EvaluatorMain extends App {
  import Expr._
  The point of application constrains the generic parameter, it is necessary to specify the type A. in eval.
  val e = Evaluator.eval[Double]((5.4 plus math.Pi) minus 8.8)
  ...
}
```

Listing 26: Flexible evaluator with DSL and typeclasses

Hints:

- it is not necessary to constrain the type of the `Const` data constructor; it can be "forall A." `Const[A]` .
- it will be convenient to use the full implicit syntax in the `eval` method: `def eval[A](expr: Expr)(implicit N: Fractional[A]): A` instead of `def eval[A : Fractional](expr: Expr): A`
- the type information for the generic type `A` will be erased, causing compiler warning in the pattern match. It is possible to eliminate the warning by using another type class. Look for *Scala type tags and manifests.*

Unlike subtype polymorphism, typeclasses bring *parametric polymorphism*. Parametric polymorphism is a different way of requiring values to contain specified behaviour. Subtype polymorphism for the evaluator would require the values in the `Const` data constructor to be some supertype of all numbers. That does not sound so bad until one realises that the methods `plus` , ..., `div` in this supertype would have to return that supertype. Imagine in Listing 27 that it is possible to extend the types in `scala.lang` , like `Int` .

```
trait Num {
  def plus(that: Num): Num
  def minus(that: Num): Num
  ...
}

This isn't actually allowed, but humour the author.
class MyInt extends Int with Num {
  Attempts to implement the Num trait won't be successful...
  def plus(that: Num): Num = this + that
                                   ↑ this doesn't quite work: can't add Int and Num.

  Or the implementation won't at all be satisfying.
  def plus(that: Num): Num = that match {
    case x: MyInt ⇒ this + x
```

```
    case _ ⇒
          ↑ now what? throw exception?; add toInt() method to Num?
  }
}
```

This is Scala's common approach to static `of`-like methods in Java. Recall `Optional.of` and similar.

```
object MyInt {
  def apply(i: Int): MyInt = new MyInt(i)
}
```

But never mind all above, the show must go on.

```
case class Const(a: Num) extends Expr

def eval(expr: Expr): Either[Error, Num] = expr match {
  case Plus(l, r) ⇒ for { l ← eval(l); r ← eval(r) } yield l.plus(r)
  ...
}
```

I allowed `Int` to be extended; requiring integer literals to be instances of `MyInt` is too much!
Using the **object** `MyInt.apply` allows us to avoid the **new** keyword in this case.

```
val e = Evaluator.eval(MyInt(5) plus MyInt(3) minus MyInt(1))
     ↑ the type of e is Num, even though MyInt was used.
```

Listing 27: Exploring the Num supertype

The code in Listing 27 is bad enough *now*, but it has the potential to become much *worse* if it becomes widely used. The authors of the code were able to implement sensible inheritance structure for *MyInt* s, *MyDouble* s, even *MyBigDecimal* s and shipped the JAR. Then someone else decided to use the expression evaluator for complex numbers; but even with the implementation of ***class*** *Complex* ***extends*** *Num* {...} , it is still possible to call *MyInt(5).plus(Complex(1, 4))* . The class *MyInt* has no knowledge about the *Complex* class! (Exactly what happens in that call depends on the implementation of the *MyInt.plus* method, but it won't be pretty.)

Parametric polymorphism, as demonstrated in your own work on the Flexible evaluator exercise, presents a neat solution. It is possible to use typeclasses to add behaviour for concrete types without losing the concreteness.

Parametric polymorphism in other languages. Scala's take on typeclasses should clarify Haskellers' usual statement that "typeclasses are not like OOP classes, they are more like interfaces." Quite: a typeclass is an interface with a type parameter that defines behaviour for the type. Its implementations specify a concrete type and implement the behaviour. Typeclass instances are then regular values (remember the singletons from the Scala code?) that implement the typeclass interface.

Require the type A to have an instance of the `NumberLike` trait available for it.
Compiler will supply the value in the parameter N of the implicit parameter list.

```
def eval[A](expr: Expr)(implicit N: NumberLike[A]): A = expr match {
  Call the typeclass's methods using the standard method call notation.
  case Plus(l, r) ⇒ N.plus(l, r)
  ...
}
```

Define the typeclass `NumberLike` for type a with methods like `plus`, ...

```
class NumberLike a where
  plus :: a → a → a
  ...
```

Provide an instance of `NumberLike` for the type **Int**.

```
instance NumberLike Int where
  plus x y = x + y ≡ addl %rdx, %rax | Hoping that we have fastcall convention and the parameters are in %rax and %rdx.
  ...
```

Provide an instance of `NumberLike` for the type **Complex**.

```
instance NumberLike Complex where
  ...
```

Just like the Scala version, `eval` requires the type a to lie in the `NumberLike` typeclass;
that is, that there is an instance of `NumberLike` for that a.

```
eval :: (NumberLike a) ⇒ Expr → a
```

However, the typeclass instance does not get a name in the function. Instead, all its functions
`plus`, ... are available in the function without any additional notation.

```
eval (Plus l r) = plus (eval l) (eval r)
                       ↑ In Scala, this would be N.plus(eval(l), eval(r))
```

There are other languages that include features that may considered to be parametric polymorphism. For example, Swift's protocol conformance and extension methods, particularly with type constraints, certainly feel like typeclasses. Consider adding equality to arrays: two arrays can be *Equatable* of their elements are *Equatable* and what that might look like using parametric polymorphism in Listing 28.

```
protocol Equatable {
    static func == (Self, Self) → Bool
}
```

An array is Equatable if its elements lie in the Equatable typeclass. No, wait, that's not what Swift programmers say. They say "an array is equatable if its elements conform to the Equatable protocol".

```
extension Array : Equatable where Element : Equatable {

    static func ==(lhs: Array<Element>, rhs: Array<Element>) → Bool {
        if lhs.count ≠ rhs.count { return false }
        for i in lhs.indices {
            if lhs[i] ≠ rhs[i] { return false }
        }
        return true
    }
}
```

```
trait Equatable[A] {
  def ==(lhs: A, rhs: A): Boolean
}
```

The Scala compiler can create an instance of Equatable for Array of elements of type A as long as the type A lies in the Equatable typeclass.

```
def arrayEquatable[A](implicit E: Equatable[A]): Equatable[Array[A]] =
  new Equatable[Array[A]] {
    def ==(lhs: Array[A], rhs: Array[A]) {
      if (lhs.length ≠ rhs.length) false
      else lhs.indices.forall(i ⇒ E.==(lhs(i), rhs(i)))
    }
  }
```

Listing 28: Extension methods and protocol conformance

2.6 Rich type system.

The type parameters encountered so far look like generics; a bit more powerful, but generics nonetheless. However, Scala's type system is much more powerful than just generics. Of course, the type parameters are translated into generics, and they look like generics. *List[A]* , *Option[A]* certainly look just like their Java counterparts *Optional<A>* , *List<A>* . Unfortunately, generics become rather complicated in the presence of subtype polymorphism.

Variance. To help with the following explanation, imagine there are two types *Reader[A]* and *Writer[A]* , and the infamous OOP structure shown in Listing 29. (The *dog-people* amongst the readers will forgive.)

```
class Animal
class Cat extends Animal
class AngryCat extends Cat

trait Reader[A] {
  def read: A
}

trait Writer[A] {
  def write(value: A): Unit
}
```

Listing 29: Reader and Writer

The substitution rule in subtype polymorphism says that a subtype can be used wherever its supertype is expected; ***val*** *x: Animal =* ***new*** *Cat* is valid, but ***val*** *x: AngryCat =* ***new*** *Cat* is not. Following the substitution principle and the behaviour of *Reader[A]* , ***val*** *x: Reader[Animal] =* ***new*** *Reader[Cat] ...* should be valid, but ***val*** *x: Reader[AngryCat] =* ***new*** *Reader[Cat] ...* should not be valid. The type of the *read* method in *Reader[Cat]* returns a value of type that *is* or can be *used in place of* *Cat* ; the code inside the *Reader[AngryCat].read* method contains all the machinery to read *AngryCat* values; an *AngryCat* value can certainly stand in place of a *Cat* value. A value of type *Reader[AngryCat]* can be used in place of *Reader[Cat]* as well as *Reader[Animal]* . Applying the same substitution rule reveals that ***val*** *w: Writer[Cat] =* ***new*** *Writer[AngryCat] ...* should not be valid: the code in *Writer[AngryCat].write(value : AngryCat)* operates on *AngryCat* s (the *write* function must be given *at least* *AngryCat* as its parameter); an instance of *Writer[AngryCat]* cannot be substituted for a value of type *Writer[Cat]* , never mind *Writer[Animal]* .

More generally, whenever *Thing[AngryCat]* can be used in place of *Thing[Cat]* or *Thing[Animal]* , the generic parameter of *Thing* is *covariant*, and written as *Thing[+A]* . When a *Thing[Animal]* can be used in place of *Thing[Cat]* or *Thing[AngryCat]*, the generic parameter of *Thing* is *contravariant*, and written as *Thing[−A]*. When no substitutions are acceptable, the generic parameter is invariant, and does not have + or −.

The names *Reader* and *Writer* aren't entirely accidental: a rule of thumb is that when the type contains functionality that "reads" values of some type *A* it should be covariant with respect to *A*, when a type contains functionality that "writes" values of type *A* it should be contravariant with respect to *A*. The code from Listing 29 should really be Listing 32.

```
trait Reader[+A] {
  def read: A
}

trait Writer[-A] {
  def write(value: A): Unit
}
```

Listing 30: Reader and Writer with proper variance

Exercise: Flexible evaluator with variance

Explore and explain the difference in just adding the generic parameter to the *Const* data constructor (leaving ***sealed trait*** *Expr* intact), or letting *Expr* become ***sealed trait*** *Expr[A]*. Explain the effects of covariant type parameter in ***sealed trait*** *Expr[+A]*. Try ***val*** *e: BigDecimal = Evaluator.eval(Const(4))* with invariant *Expr* and then with covariant *Expr* to get started.

Types, type constructors & higher-kinded types. Scala uses parentheses to apply parameters to functions; given the function ***def*** *f(a: Int): Int* the code *f(4)* applies the value *4* to the function *f*. The application contains all parameter values (imagine that the value *4* eats off the parameters from the left), so all that remains is the returned *Int* value. That is not surprising at all.

Scala uses square brackets to apply types to type constructors; given the type constructor *List*, the code *List[Int]* applies the type *Int* the type constructor *List*. The application contains all type values (imagine that the type *Int* eats off the types from left to right), so all that remains is the concrete type *List[Int]*. This is *now* not surprising at all.

Just like values have types, types have kinds. Take the type constructor *List[_]* (or just *List*): it is not a concrete type (list of what? *String* s? *Int* s, *StormTrooper* s?), it is a type constructor that needs one more type to produce a concrete type. Type constructors are sometimes described using a simple type arity syntax. The arity syntax only specifies the number (and structure) of types that need to be applied in order to arrive at a concrete type. In this syntax, the kind of the *List* type constructor is $* \rightarrow *$. It looks like a [type-level] function that when applied to one type produces a type. The *Either* type constructor's arity syntax is $(*, *) \rightarrow *$. (Recall that *Either* takes a pair of type parameters.) The arity syntax is accurate for type constructors that place no constraints on their type parameters; it isn't that accurate (in Scala) for type constructors that constrain their type parameters. Nevertheless, it is usually sufficient.

The expression writer implemented earlier hides the fact that it is not a pure function: it *writes* something somewhere, it returns *Unit*, but its intent (and implementation) reveals that it cannot be simply substituted for any other *()* value. This is not surprising: after years of exposure to most languages, programmers know that *some* operations aren't useful for their return value, but for the side-effects they perform on the world. The expectation for the *Writer.write* function is that it is not called for the *()* it returns, but for the *effect of writing the value*. This is a useful abstraction, and in the case of a *Writer* that sends a textual representation of the value to the standard output the level of abstraction is sufficient: there are only few error scenarios that need to be handled (and maybe errors can be ignored altogether). The abstraction might not be sufficient for *Writer* that performs a REST call to another service to submit the value: what if the service is unavailable?; what if it is too slow to respond?; what if the caller is capable of performing asynchronous, non-blocking calls easily? In case of failures, one could consider throwing exceptions (and not forgetting to document those!); in the non-blocking case, the value *()* is simply not sufficient. The source of a lot of subtle and not-so-subtle bugs in programs stem from leaky abstractions. The abstraction hides something for convenience (like returning just *()* from the *write* function, allowing the callers to pretend that "everything is fine"); the more complex the abstraction, the more complex its configuration and behaviour becomes; unfortunately, its unexpected behaviours become more complex, too.

Back to our *Writer*. Another level of abstraction is needed: the *Writer.write* function needs to return the *()* "boxed" in another value. To make this box useful in most programs, it will be necessary for it to lie in a typeclass that contains functions for transforming the value contained in it, and unpacking the value and combining with other boxes. In other words, it will need the *map* and *flatMap* functions. (This will make it useful in ***for*** expressions, how neat!) At the point of defining the *Writer* trait, the shape of this box is not known; all that is known is that it will be some type that takes another type. Scala uses identifiers in square brackets to define type parameters (recall ***def*** *eval[A](...): A*), the identifier *A* refers to a concrete type. An identifier that represents a type constructor that needs one more type to make it concrete is simply *A[_]*. Applying this to the *Writer* trait yields code in Listing 31.

```
trait Writer[-A] {
  def write[M[_]](value: A): M[Unit]
}
```

Listing 31: Defining and using higher-kinded types

Unfortunately, this definition would make it somewhat difficult to implement the trait: there would be no way to construct instances of *M[_]*. The implementation of the *write* function needs to receive a typeclass instance that allows it to construct a value of type *M[_]*; and the constructed *M[_]* should be sequenceable. It should lie in the $\mathfrak{MONAD}$ typeclass.

Exercise: Flexible writers

And so it has come to this: implement the expression writer to be usable in caller-defined contexts by requiring its higher-kinded type parameter *M* to lie in the *Monad* typeclass.

```
trait Writer[-A] {
  def write[M[_]: Monad](value: A): M[Unit]
}

object Writer {
  def apply[A](): Writer[A] = new Writer[A] {
    Implement the write method; again, you will find it convenient to use the full implicit syntax,
    because you will need to access the instance of the Monad typeclass for the [higher-kinded] type M.
  }
}

object EvaluatorMain extends App {
  ...

  import scala.concurrent.ExecutionContext.Implicits.global
  import scalaz.std.scalaFuture.futureInstance

  Writer().write[Future](1 plus 3).foreach(_ ⇒ println("Done"))
  Writer().write[IO](1 plus 3).unsafePerformIO()

  This is needed to give the Future time to be scheduled and executed.
  Thread.sleep(1000)
}
```

Listing 32: Reader and Writer with proper variance

References

[1] *Adapter Pattern*. https://en.wikipedia.org/wiki/Adapter_pattern.

[2] *F#*. https://fsharp.org.

[3] *Haskell*. https://www.haskell.org.