Patterns

Commonly Used Patterns in Scala

Agenda

1. What is a Pattern
2. What about GoF?
3. ADTs (Algebraic Data Types)
4. Loan (Resource Management)
5. Trait patterns
6. Streamlined Reflection
7. Type classes
8. DSLs and Fluent APIs
9. Compile-time Verified Dependency Injection
10. Mutable Construction/Immutable Result

What is a Design Pattern?

- A general reusable solution to a commonly occurring problem within a given context -- wikipedia
- A shortcoming in your language, requiring the same solution repeatedly?
- A way of selling books…
- But actually patterns, and anti-patterns, can be useful and save time

What About the Gang of Four Patterns?

- These are OO centric patterns
- Furthermore, they are often provided by features already in Scala:

Pattern	--	Scala Feature
Singleton		object
(Abstract) Factory		apply method (+ traits for abstract)
Builder		Several options, including: infix methods for fluent API case classes and named/default params methods using copy on case classes, etc.
Prototype		case classes with copy method
Adapter		Trait mixins or possibly type-classes
Bridge		Traits
Composite		case classes
Decorator		Type-classes
Facade		Traits (and just-in-time traits)

Scala GoF Patterns Continued

Pattern	Scala Feature
Flyweight	Map (possibly extend Function) Or use Guava Cache
Proxy	Various features, e.g. `lazy val`
Chain of Responsibility	Partial functions (`.isDefinedAt`)
Command	Function literals
Interpreter	case classes (ADTs), pattern matching All the way through to Free Monads…
Iterator	Iterator class or `iterator` method on collections
Mediator	Just-in-time traits is one solution
Memento	Type-classes
Observer	Substitute with Pattern Matching
State	Discouraged in FP unless you use State Monad
Strategy	Traits, ADTs
Template	Traits with some abstract members
Visitor	Sealed traits, pattern matching, ADTs

Pattern -- Scala Feature

Algebraic Data Types (ADTs)

- The building blocks of many higher-level patterns and abstractions

```
sealed trait StatementLine

case class Print(printList: Seq[String]) extends StatementLine

case class For(variable: String, start: Int, end: Int, by: Option[Int])
  extends StatementLine

case class Goto(lineNumber: Int) extends StatementLine

case object Next extends StatementLine
```

```
val basicProgram: Seq[StatementLine] = Seq(
  For("x", 1, 10, Some(1)),
  Print(Seq("Hello, world")),
  Print(Seq("Another line")),
  Next
)

basicProgram.collect {
  case Print(items) => items
}
```

Recursive ADTs

- E.g. a linked list (based on a simplified version of Scala's List)
- https://github.com/scala/scala/blob/v2.12.4/src/library/scala/collection/immutable/L

```scala
sealed trait LinkedList[T] {
  def isEmpty: Boolean
  def head: T
  def tail: LinkedList[T]
}

case class Empty[T]() extends LinkedList[T] {
  def isEmpty: Boolean = true
  def head: T = throw new IllegalStateException("head of empty list")
  def tail: LinkedList[T] = throw new IllegalStateException("tail of empty list")
}

case class Cons[T](head: T, tail: LinkedList[T]) extends LinkedList[T] {
  def isEmpty: Boolean = false
}

val myLinkedList = Cons(1, Cons(2, Empty[Int]()))
myLinkedList.head          // 1
myLinkedList.tail.head     // 2
```

Loan Pattern (Auto Resource Management)

```
import java.io._
import scala.io.Source

// might need to change this choice on windows:
val bashFile = new File(".bashrc")

def withFileLines[A](file: File)(fn: Seq[String] => A): A = {
  val source = Source.fromFile(file)
  try {
    fn(source.getLines().toList)
  } finally source.close()
}

val matches = withFileLines(bashFile) { iter =>
  for (line <- iter if line.contains("alias "))
    yield line.split("alias ").last
}

matches foreach println
```

- Be careful with lazy collections like iterator and stream, they may still close before evaluated

Alternative ARM to Loan/AutoCloseable

```
// build.sbt:
libraryDependencies += "com.jsuereth" %% "scala-arm" % "2.0"

import resource._

for {
  input <- managed(new java.io.FileInputStream("test.txt"))
  output <- managed(new java.io.FileOutputStream("test2.txt"))
} {
  val buffer = new Array[Byte](512)
  def read(): Unit = input.read(buffer) match {
    case -1 => ()
    case  n =>
    output.write(buffer,0,n)
    read()
  }
  read()
}
```

Trait Patterns

- Simple mixin

```
trait Logger {
  def info(msg: String): Unit =
    println(s"INFO: $msg")

  def warn(msg: String): Unit =
    println(s"WARN: $msg")

  def error(msg: String): Unit =
    println(s"ERROR: $msg")
}

class Demo extends Logger {
  def testLogging(): Unit = {
    info("This is an info")
    warn("This is a warning")
    error("This is an error")
  }
}

(new Demo).testLogging()
```

Because it's selfless

```
object Logger extends Logger

class Demo2 {
  import Logger._
  def testLogging(): Unit = {
    info("This is an info")
    warn("This is a warning")
    error("This is an error")
  }
}

(new Demo2).testLogging()
```

- Selfless means the trait is fully contained with no dependencies or undefined behavior
- It can therefore be extended by an object, which means the behavior can be imported as an alternative to trait inheritance.

Stacking Traits

```
trait Logger {
  def message(msg: String): String
  def info(msg: String): Unit = println("INFO: " + message(msg))
  def warn(msg: String): Unit = println("WARN: " + message(msg))
  def error(msg: String): Unit = println("ERROR: " + message(msg))
}

trait StandardLogger extends Logger {
  override def message(msg: String): String = msg
}
```

```
trait DateLogger extends Logger {
  abstract override def message(msg: String): String =
    s"${LocalDateTime.now()}: ${super.message(msg)}"
}

class Demo3 extends StandardLogger with DateLogger {
  def testLogging(): Unit = {
    info("This is an info")   // INFO: 2017-11-21T15:01:21.039: This is an info
    warn("This is a warning") // WARN: 2017-11-21T15:01:21.039: This is a warning
    error("This is an error") // ERROR: 2017-11-21T15:01:21.040: This is an error
  }
}
```

Poor Man's Interface Injection

```scala
class Door {
  def close(): Unit = println("SLAM!")
}

def closeAll(items: Seq[Closeable]): Unit = {
  for (item <- items) yield Try(item.close())
}

// note that Door is not Closeable, but I can do this:

val door1 = new Door with Closeable
val door2 = new Door with Closeable
val os = new PrintWriter("temp.txt")

closeAll(Seq(door1, os, door2))
```

- This is a nice, orderly alternative to having to resort to structured typing
- Also works with AutoCloseable (and ARM blocks)

Streamlined Reflection

```
val s = "hello"

s.charAt(1)

val a: Any = s

// The traditional way
val charAt = a.getClass.getMethod("charAt", classOf[Int])
charAt.invoke(a, new Integer(1)).asInstanceOf[Char]


// The scala way:
a.asInstanceOf[{def charAt(i: Int): Char}].charAt(1)
```

- Of course, you probably shouldn't be resorting to reflection at all

Type Classes

One of the most common patterns. Provides "ad-hoc" polymorphism, or a way of providing behavior for a class without needing to affect the inheritance hierarchy of the class (which also makes it possible to add behavior for other classes that do not belong to you).

e.g.:

```
trait JSONWrite[T] {
    def toJsonString(item: T): String
}

def jsonify[T: JSONWrite](item: T): String =
    implicitly[JSONWrite[T]].toJsonString(item)
```

```
implicit object StringJSONWrite extends JSONWrite[String] {
    def toJsonString(item: String) = s""""$item""""
}

jsonify("hello")
> res3: String = """
> "hello"
> """
```

Composing Type Classes

```scala
implicit object IntJsonWrite extends JSONWrite[Int] {
  def toJsonString(item: Int) = item.toString
}

implicit def listJsonWriter[T: JSONWrite]: JSONWrite[List[T]] =
  { (xs: List[T]) =>
    val tJson = implicitly[JSONWrite[T]]

    xs.map(tJson.toJsonString).mkString("[", ",", "]")
  }

jsonify(List(1,2,3))  // [1,2,3]
jsonify(List("hello", "world")) // ["hello","world"]
```

- Note that the above also uses the automatic SAM expansion introduced in Scala 2.12

More Composition - Auto Case Classes

What about if we want to do a case class?

```
case class Person(name: String, age: Int)
```

Start with a generic abstract base class:

```
import scala.reflect.runtime.universe._

abstract class CaseClassAbstractJsonWriter[T: TypeTag]
    extends JSONWrite[T] {
  val tt = typeTag[T]
  implicit val writer: JSONWrite[T] = this
}
```

By making the implicit writer a reference back to `this`, we make this class its own implicit when mixed into a companion object.

Reflecto-magic for Case Classes

```scala
abstract class CaseClassJsonWriter2[A: JSONWrite, B: JSONWrite, T: TypeTag]
    extends CaseClassAbstractJsonWriter[T] {

  def unapply(x: T): Option[(A, B)]
  private val aJson = implicitly[JSONWrite[A]]
  private val bJson = implicitly[JSONWrite[B]]

  private val name =
    this.getClass.getSimpleName.filterNot(_ == '$')

  private val fieldNames = tt.tpe.member(TermName("copy")).
    info.paramLists.flatMap(_.map(_.name.toString))
```

This class is intended to be mixed into a companion object, hence unapply will be defined.

We get the name of the class and filter out `$` (on the end of the companion name)

We use the copy method to retrieve the names for the fields

Completing the Case Class Implementation

```
  private def fields(x: T): List[String] = unapply(x) match {
    case Some((a, b)) =>
      List(
        aJson.toJsonString(a),
        bJson.toJsonString(b)
      )
    case None => throw new IllegalStateException("Cannot serialize")
  }

  override def toJsonString(item: T): String = {
    val fieldPairs = fieldNames.zip(fields(item))

    val fieldStrings = for ((name, value) <- fieldPairs) yield {
      s"$name: $value"
    }

    val allFields = fieldStrings.mkString(", ")

    s"""{
       |  "$name": {$allFields}
       |}""".stripMargin
  }
}
```

In Use

```
implicit object Person extends CaseClassJsonWriter2[String, Int, Person]

val person = Person("Harry", 20)

jsonify(person)

// res3: String = "{
//    "Person": {"name": "Harry", "age": 20}
// }
```

Because of the implicit writer self reference in the abstract class, the companion definition is also its own type class, searched automatically when we need a Person writer.

We inherit from the CaseClassJsonWriter (of correct arity) in the companion object, supplying the types required.

Alternatively, you could use a more specific implementation, or macros if you really must.

E.g. https://github.com/spray/spray-json/blob/master/src/main/scala/spray/json/ProductFormats.scala

DSLs and Fluent APIs

```
val or: String = "or"
val to: String = "to"

object To {
  def be(o: or.type) = this
  def not(t: to.type) = this
  def be: String = "That is the question"
}

To be or not to be     // That is the question
```

Combines infix and singleton types to build a little grammar

```
val toBeOrNotToBe = "That is the question"
```

Remember, the effort is not always worth the reward

Named/Default Parameters

Remember that `case classes` can make a great API when used effectively with default/named parameters:

```
object TransactionType extends Enumeration {
  val Long, Atomic = Value
}

case class DBConnection(
  url: String,
  user: String = "postgres",
  pass: String = "password",
  txnType: TransactionType.Value = TransactionType.Atomic
)

DBConnection(url = "postgres:127.0.0.1/mydb")

DBConnection(
  url = "postgres:127.0.0.1/mydb",
  user = "dbUser",
  pass = "secret",
  txnType = TransactionType.Long
)
```

You don't have to get all fancy to make something readable.

Compile Time Verified Dependency Injection

- Covered in detail in Module 7
- Cake is how the Scala compiler used to work
  - But this pattern has demonstrated unacceptably long compile times for large projects
- Parfait is how the Dotty compiler works
  - Also simpler forms are evident throughout the Scala core libraries, e.g. Futures
- There are also options like macwire, although you will need to include a library for that
  - https://github.com/adamw/macwire
- Certainly there is no need to settle for runtime dependency injection in Scala

Mutable Constructor Pattern

```scala
import scala.util.control.NonFatal

class NamedTest(val name: String, val test: () => Unit)

abstract class DemoSuite {
  private[this] var _tests = List.empty[NamedTest]

  protected def test(testName: String)(test: => Unit): Unit =
    _tests = new NamedTest(testName, () => test) :: _tests

  protected lazy val tests: List[NamedTest] = _tests.reverse
  def run(): Unit = {
    for (namedTest <- tests) {
      print(s"Running ${namedTest.name}: ")
      try {
        namedTest.test()
        println("Passed")
      }
      catch {
        case NonFatal(ex) =>
          println(s"Failed ${ex.getMessage}")
      }
    }
  }
}
```

Mutable Constructor In Use

```
class MyTests extends DemoSuite {
  test("1 + 1 should be 2") {
    assert(1 + 1 == 2)
  }

  test("1 + 1 should be 3") {
    assert(1 + 1 == 3)
  }

  test("1 / 0 should be 0") {
    assert(1 / 0 == 0)
  }
}

(new MyTests).run()

// Running 1 + 1 should be 2: Passed
// Running 1 + 1 should be 3: Failed assertion failed
// Running 1 / 0 should be 0: Failed / by zero
```