Scala Implicits

Tuấn-Anh Nguyễn

Aug 17, 2022

Implicit Conversions

- Technically both are conversions (lang feature).
  - `implicit def`
  - `implicit class`
- Colloquially (practical patterns).
  - When converted value is passed as argument,
  - to paper over type incompatibility (bad),
  - instead of introducing new methods (good);
  - and usually implemented by the former.
- `JavaConversions` (bad) vs. `JavaConverters` (good)

Implicit Parameters (and Arguments/Values)

- Definitions

```
def withLoggingELK(action: String)(implicit exec: ExecutionContext): Future[Response[T]]
def createKafkaProducer(topic: String)(implicit config: Config): KafkaProducer[Shipment]
class ShipmentUpdateToNotifications(implicit clock: PPClock)
```

- Constructions

```
// Create implicit values.
implicit object NoneEmptyStringReads extends Reads[String]
implicit val config: Config = getUserConfig(env)
implicit val clock: PPClock = PPClock.real

// Methods with implicit parameters can now use these
input
  .process(new RewriteNewEventsProcess(ttlConfig))
  .process(new ShipmentUpdateToNotifications)
```

- Imports

```
// Bring in implicit values for anyone needed.
import org.apache.flink.api.scala._
import com.parcelperform.flink.ConfigParameters.ConfigOps
import scala.concurrent.ExecutionContext.Implicits.global

// Get a hold of the object to do something with it.
implicitly[Notification.Targeted[Channel]].toSubscribed( ... )
```

- Resolution: local scope, companion objects, (and other)

Context Bounds

- Syntatic sugar for implicit params of second-order types.

```
def producer[T](topic: String)(
  implicit schema: FlinkSerializationSchema[T], config: Config)
def producer[T: FlinkSerializationSchema](topic: String)(
  implicit config: Config)

class CirceJsonMappingFunc[T](jobName: String)(
  implicit decoder: Decoder[T])
class CirceJsonMappingFunc[T: Decoder](jobName: String)
```

- Sound fancy but look mundane. How is this special?

Pattern: Contexts

- Scoped globals, in a way.
- More useful when intermediate callers just pass them on.
  - Can be used as a dependency injection mechanism.
  - Example: `Job` → `Functions` → `DB/HTTP Clients`

Pattern: Extension Methods

Adding new methods to existing data types.

```
implicit class ConfigOps(val config: Config) extends AnyVal {
  def getStringifiedList(path: String, separator: String): String =
    config.getStringList(path).asScala.mkString(separator)

  def getCommaSeparatedList(path: String): String =
    getStringifiedList(path, ",")
}

implicit class SugarOps[T](val x: T) extends AnyVal {
  def between(y: T, z: T)(implicit ord: Ordering[T]): Boolean =
    ord.lteq(y, x) && ord.lteq(x, z)
}
```

```
config.getStringifiedlist("kafka.brokers")
delivery.between(start, end)
```

Pattern: Type Classes

- Abstract parameterized types.

```
trait Decoder[A] { def apply(c: HCursor): Decoder.Result[A] }
trait TypeInformation[T]
trait SerializationSchema[T] { def serialize[T]: Array[Byte] }
```

- Enable composition
  - Adding new behaviors to closed data types.
  - Nothing to do with `class` (inheritance).
- Go hand-in-hand with context bounds.

```
def reinterpretAsKeyedStream[K: TypeInformation](keySelector: T ⇒ K): KeyedStream[T, K]
def map[K: TypeInformation, V: TypeInformation](name: String): MapStateDescriptor[K, V]
def producer[T: SerializationSchema](topic: String): KafkaProducer[T]
```

- Implementations are called instances.

```
implicit val decodeDateTime: Decoder[DateTime] = new Decoder[DateTime] {
  def apply(c: HCursor): Decoder.Result[DateTime] = ???
}

object Carrier {
  implicit val decoder: Decoder[Event] = deriveDecoder
}

createTypeInformation[Carrier]
```

Anti-pattern: Automatic Conversions

- Hard to read and navigate.
- Surprising behaviors.
- Bad compilation performance.

Compilation Performance

- Many implicit "scopes" to search through.
- As with type inference, aim for:
  - Implicit at call site
  - Explicit types at definition/creation site

```
// Bad
implicit val decoder = deriveDecoder[NotificationConfig]
implicit val config = getUserConfig()

// Good
implicit val decoder: Decoder[NotificationConfig] = deriveDecoder
implicit val config: Config = getUserConfig()
```

- Scala3 enforces this.

Combining Conversion and Parameter

- Conversion and parameter.
  - Conditional conversion
    - `implicit def(implicit x: X)`
    - `implicit class(implicit x: X)`
  - Conditional extension method
    - `def (implicit)` inside an `implicit class`
  - Parameter goodness redeems conversion badness, somewhat.
- Conversion as parameter.
  - Implicit views, view bounds.
  - Forget this historical misstep.

Type Proofs

- Sub-type constraints <:<, =:=

```scala
def dateTimeToHHMM(x: DateTime): Option[String] = x.map(_.toString(HHMM))
def dateTimeToHHMM[T <: DateTime](x: T): Option[String] = x.map(_.toString(HHMM))
def dateTimeToHHMM[T](x: T)(implicit ev: T <:< DateTime): Option[String] = x.map(_.
// Why would the last be useful???
```

- Boolean logic on type constraints

```scala
// Inlined through evidence parameter
def foo[T](x: T)(implicit ev: (T <:< Int) | (T <:< String)) = x

// Named context bound
type CheckIntOrString[T] = (T <:< Int) | (T <:< String)
def foo[T: CheckIntOrString](x: T) = x

foo(5) // compiles
foo("5") // compiles
foo(5.5) // does not compiles
```

- Scala 3 doesn't need this; has proper union/intersection types.

Type Proofs - Boolean Logic Impl

```scala
import scala.annotation.{implicitNotFound, implicitAmbiguous}

object TypeConstraints {
  def unused: Nothing = sys.error("don't call this, it's for type checking only")

  @implicitNotFound("Cannot prove that (${A}) And (${B})")
  trait And[A, B] extends Serializable
  private val singletonAnd = new And[Any, Any] { def apply(x: Any): Any = x }
  implicit def andConforms[A, B](implicit evA: A, evB: B): And[A, B] =
    singletonAnd.asInstanceOf[And[A, B]]

  trait Not[A] extends Serializable
  private val singletonNot = new Not[Any] { def apply(x: Any): Any = x }
  implicit def notConforms[A]: Not[A] = singletonNot.asInstanceOf[Not[A]]
  @implicitAmbiguous("Cannot disprove ${A}")
  implicit def notAmbiguous1[A](implicit ev: A): Not[A] = unused
  implicit def notAmbiguous2[A](implicit ev: A): Not[A] = unused

  @implicitNotFound("Cannot prove that (${A}) Or (${B})")
  trait Or[A, B] extends Serializable
  private val singletonOr = new Or[Any, Any] { def apply(x: Any): Any = x }
  implicit def orConforms[A, B](implicit ev: Not[Not[A] & Not[B]]): Or[A, B] =
    singletonOr.asInstanceOf[Or[A, B]]

  type ~[A] = Not[A]
  type &[A, B] = And[A, B]
  type |[A, B] = Or[A, B]
}
```

Exercise

```
override def toBCC(notification_config_id: Int,
                   recipient: String, bccRecipients: Seq[String],
                   shipment_update: CircleJson): BCC =
  throw new UnsupportedOperationException("SMS doesn't support BCC")
```

Turn the above runtime exception into a compile-time error

- Type constraints may make it easier.
- Typeclass-based solution would be cleaner.

Scala 3 - Less Overloaded Keywords

- Parameter, declaration, definition: `using` clauses
  - Can be anonymous
- Argument, construction, call site: `given` instances
  - Import: `module._` -> `module.given`
  - Conjure: `implicitly[T]` -> `summon[T]`
  - Can be anonymous

Scala 3 - Given Instances

```scala
given Config = getUserConfig(env)
given PPClock = PPClock.real

given intOrd: Ord[Int] with
  def compare(x: Int, y: Int) = ???

import org.apache.flink.api.scala.given
import com.parcelperform.flink.ConfigParameters.given
import scala.concurrent.ExecutionContext.Implicits.{given ExecutionContext}
```

Scala 3 - Extension Methods

First-class implementation syntax:

```
extension (config: Config)
  def getStringifiedList(path: String, separator: String): String =
    config.getStringList(path).asScala.mkString(separator)

  def getCommaSeparatedList(path: String): String =
    getStringifiedList(path, ",")

extension [T](x: T)(using ord: Ordering[T])
  def between(y: T, z: T): Boolean =
    ord.lteq(y, x) && ord.lteq(x, z)
```

Scala 3 - Type Classes

- Same: declaration, bounds, usage.
- First-class implementation syntax:

```
given Decoder[DateTime] with
  extension(d: DateTime) def apply(c: HCursor): Decoder.Result[DateTime] = ???
```

Scala 3 - A Lot More

https://docs.scala-lang.org/scala3/reference/contextual/index.html

- Type class derivation
- Multiversal equality
- By-name context parameters
- Negated givens

Scala 3 - Context Functions

- Interesting, but contentious
  - Making checked exceptions work
  - Effect typing vs. monads
- Discuss offline if you want to know more
  - Recommendation: don't