Dead-Simple Dependency Injection

Northeast Scala Symposium, 2012

Rúnar Óli Bjarnason

About Rúnar

Senior Software Engineer, Capital IQ, Boston

Contributor to Scalaz

Author, Functional Programming in Scala (Manning, 2012)
http://manning.com/bjarnason

```
def setUserPwd(id: String, pwd: String) = {
  Class.forName("org.sqlite.JDBC")
  val c = DriverManager.getConnection("jdbc:sqlite::memory:")
  val stmt = c.prepareStatement(
    "update users set pwd = ? where id = ?")
  stmt.setString(1, pwd)
  stmt.setString(2, id)
  stmt.executeUpdate
  c.commit
  c.close
}
```

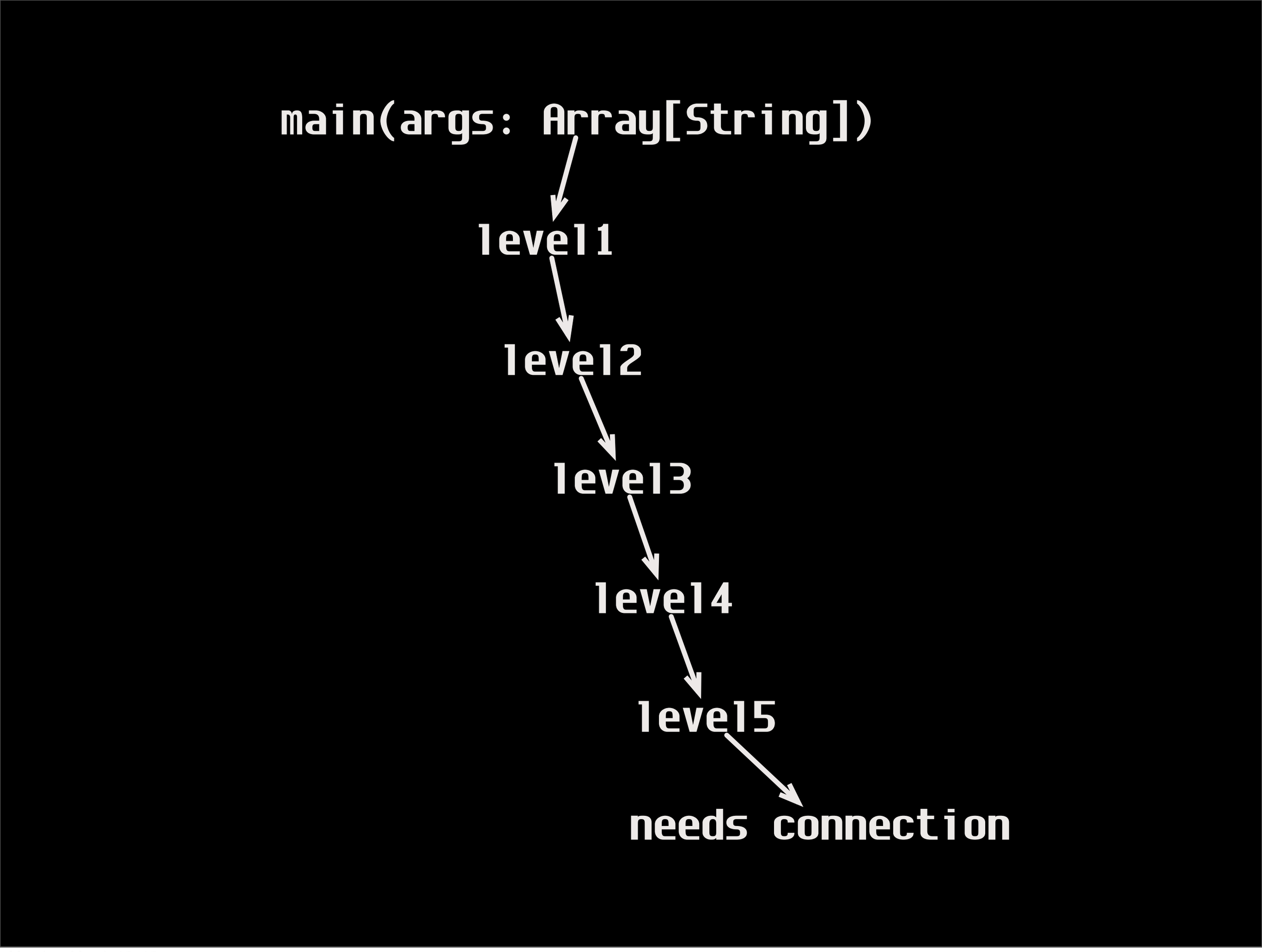
```
def setUserPwd(id: String, pwd: String) = {
  val c = ConnectionFactory.getConnection
  val stmt = c.prepareStatement(
    "update users set pwd = ? where id = ?")
  stmt.setString(1, pwd)
  stmt.setString(2, id)
  stmt.executeUpdate
  stmt.close
}
```

A global Connection factory?

Bad idea.
Hidden dependency.
Requires magic initialization step.
Not much better than a pointy stick in the eye.
Same goes for thread-local connections.

Inversion of Control

```
def setUserPwd(id: String,
               pwd: String,
               c: Connection) = {
  val stmt = c.prepareStatement(
    "update users set pwd = ? where id = ?")
  stmt.setString(1, pwd)
  stmt.setString(2, id)
  stmt.executeUpdate
  stmt.close
}
```


main(args: Array[String])
level1
level2
level3
level4
level5
needs connection

Before Currying

```
def setUserPwd(id: String,
               pwd: String,
               c: Connection) = {
  val stmt = c.prepareStatement(
    "update users set pwd = ? where id = ?")
  stmt.setString(1, pwd)
  stmt.setString(2, id)
  stmt.executeUpdate
  stmt.close
}
```

After Currying

```
def setUserPwd(id: String,
               pwd: String): Connection => Unit =
  c => {
    val stmt = c.prepareStatement(
      "update users set pwd = ? where id = ?")
    stmt.setString(1, pwd)
    stmt.setString(2, id)
    stmt.executeUpdate
    stmt.close
  }
```

Connection Reader

```
case class DB[A](g: Connection => A) {
  def apply(c: Connection) = g(c)
}
```

Lift existing functions

```
case class DB[A](g: Connection => A) {
  def apply(c: Connection) = g(c)
  def map[B](f: A => B): DB[B] =
    DB(c => f(g(c)))
}
```

```
// map : (A => B) => (DB[A] => DB[B])
```

Combine two actions

```
case class DB[A](g: Connection => A) {
  def apply(c: Connection) = g(c)
  def map[B](f: A => B): DB[B] =
    DB(c => f(g(c)))
  def flatMap[B](f: A => DB[B]): DB[B] =
    DB(c => f(g(c))(c))
}

// flatMap : (A => DB[B]) => (DB[A] => DB[B])
```

Connection Reader Monad

```
case class DB[A](g: Connection => A) {
  def apply(c: Connection) = g(c)
  def map[B](f: A => B): DB[B] =
    DB(c => f(g(c)))
  def flatMap[B](f: A => DB[B]): DB[B] =
    DB(c => f(g(c))(c))
}


def pure[A](a: A): DB[A] = DB(c => a)
```

Monad comprehension

```
def changePwd(userid: String,
              oldPwd: String,
              newPwd: String): DB[Boolean] =
  for {
    pwd <- getUserPwd(userid)
    eq <- if (pwd == oldPwd) for {
            _ <- setUserPwd(userid, newPwd)
          } yield true
          else pure(false)
  } yield eq
```

DB interpreter

```
abstract class ConnProvider {
  def apply[A](f: DB[A]): A
}

def mkProvider(driver: String, url: String) =
  new ConnProvider {
    def apply[A](f: DB[A]): A = {
      Class.forName(driver)
      val conn =
        DriverManager.getConnection(url)
      try { f(conn) }
      finally { conn.close }
    }
  }
```

Concrete instances

```
lazy val sqliteTestDB =
  mkProvider("org.sqlite.JDBC", "jdbc:sqlite::memory:")
```

```
lazy val mysqlProdDB =
  mkProvider(
    "org.gjt.mm.mysql.Driver",
    "jdbc:mysql://prod:3306/?user=one&password=two")
```

Needs a ConnProvider

```
def myProgram(userid: String): ConnProvider => Unit =
  r => {
    println("Enter old password")
    val oldPwd = readLine
    println("Enter new password")
    val newPwd = readLine
    r(changePwd(userid, oldPwd, newPwd))
  }
```

“Injection”

```
def runInTest[A](f: ConnProvider => A): A =
  f(sqliteTestDB)
```

```
def runInProduction[A](f: ConnProvider => A): A =
  f(mysqlProdDB)

def main(args: Array[String]) =
  runInTest(myProgram(args(0)))
```

Dependency injection framework:

```scala
case class Reader[C, A](g: C => A) {
  def apply(c: C) = g(c)
  def map[B](f: A => B): Reader[C, B] =
    Reader(c => f(g(c)))
  def flatMap[B](f: A => Reader[C, B]): Reader[C, B] =
    Reader(c => f(g(c))(c))
}

def pure[A](a: A): C => A = Reader(c => a)

implicit def reader[A,B](f: A => B) = Reader(f)
```

Reader monad FTW

- Dead-simple. Just function composition.
- Explicit, type-safe dependencies.
- Lift any function.
- No frameworks, annotations, or XML.
- No initialization step.
- Doesn't rely on esoteric language features.

Reader monad FTL

- Combining with other monads can get verbose (see scalaz.Kleisli).
- Juggling multiple configurations at once can be awkward.
- Monadic style
- No “auto-wiring” (implicits compensate).

Taking it further

```
trait KeyValueStore {
  def put(key: String, value: String): Unit
  def get(key: String): String
  def delete(key: String): Unit
}

def modify(k: String, f: String => String):
  Reader[KeyValueStore, Unit] =
    for {
      v <- _.get(k)
      _ <- _.put(k, f(v))
    } yield ()
```

A little language

```
sealed trait KVS[A]

case class Put[A](key: String,
                  value: String,
                  a: A) extends KVS[A]

case class Get[A](key: String,
                  h: String => A) extends KVS[A]

case class Delete[A](key: String,
                     a: A) extends KVS[A]
```

A little language

```
def modify(k: String, f: String => String) =
  Get(k, v => Put(f(v), ()))
```

A little language

```
def modify(k: String, f: String => String):
  KVS[KVS[Unit]] =
    Get(k, v => Put(f(v), ()))
```

A little language

```
KVS[KVS[A]] => KVS[A]  ??
```

Free monads!

(see? told you)

```
case class Done[F[_]:Functor, A](a: A)
  extends Free[F, A]
case class More[F[_]:Functor, A](k: F[Free[F, A]])
  extends Free[F, A]

class Free[F[_], A](implicit F: Functor[F]) {
  def flatMap[B](f: A => Free[F,B]): Free[F, B] =
    this match {
      case Done(a) => f(a)
      case More(k) => More(F.map(k)(_ flatMap f))
    }
  def map[B](f: A => B): Free[F, B] =
    flatMap(x => Done(f(x)))
}
```

Functor

```
trait Functor[F[_]] {
  def map[A,B](a: F[A])(f: A => B): F[B]
}

implicit val kvsFunctor: Functor[KVS] =
  new Functor[KVS] {
    def map[A,B](a: KVS[A])(f: A => B) = a match {
      case Put(k, v, a) => Put(k, v, f(a))
      case Get(k, h) => Get(k, x => f(h(x)))
      case Delete(k, a) => Delete(k, f(a))
    }
  }
```

KVS monad

```
def put(k: String, v: String): Free[KVS, Unit] =
  More(Put(k, v, Done(())))
def get(k: String): Free[KVS, String] =
  More(Get(k, v => Done(v)))
def delete(k: String): Free[KVS, Unit] =
  More(Delete(k, Done(())))

def modify(k: String,
           f: String => String): Free[KVS, Unit] =
  for {
    v <- get(k)
    _ <- put(k, f(v))
  } yield ()
```

KVS interpreter

```
def runKVS[A](kvs: Free[KVS, A],
              table: Map[String, String]):
  Map[String, String] =
    kvs match {
      case More(Put(k, v, a)) =>
        runKVS(a, table + (k -> v))
      case More(Get(k, f)) =>
        runKVS(f(table(k)), table)
      case More(Delete(k, a)) =>
        runKVS(a, table - k)
      case Done(a) => table
    }
```

Conclusions

- Scala is not Java
- Don’t let habits from old languages dictate designs in new languages.

Conclusions

Old and busted	=>	New hotness
Frameworks, factories, magic initialization	=>	functions from inputs to outputs
Dependency injection	=>	Little languages
Many implementations of an interface	=>	Many interpreters of a language

Questions?

Takk.