Modularity and Dependencies

Eliminate Static Cling and Test Your Code

Agenda

1. Static Cling
2. Classes vs Objects
3. Constructor Params (a.k.a. The Simplest Thing)
4. Run Time Dependency Injection
5. Compile Time Dependency Injection
6. The Cake Pattern
7. Parfait (or Cup Cake)

Static Cling

- Phrase coined in Java for inflexible resource dependencies caused by static methods
- In Scala, the same is true for objects hard wired in for resource, e.g.

```
object PostgresDBDetails {
  val dbURL: String = "jdbc:postgresql://localhost:5432/data"
  val dbUser: String = "user"
  val dbPass: String = "pass"
}

object PostgresDBConnection {
  import PostgresDBDetails._  // hard wired above
  def db = DB(dbURL, dbUser, dbPass)
}

class UserManagement {
  val db = PostgresDBConnection.db  // more hard wiring
  def findUser(id: Int): User =
    withTransaction(db) { implicit session =>
      session.query(s"select * from users where id = $id").map(resultsToUser)
    }
}
```

Classes vs Objects

- Objects are initialized when they are first referenced
- Unlike classes, there are no constructor parameters
- Singletons are the enemy of flexible configuration

Constructor Parameters

- Also known as, the simplest thing that can possibly work:

```
trait DBConnection { val db: DB }

class PostgresDBConnection(dbURL: String, dbUser: String, dbPass: String)
    extends DBConnection {
  val db: DB = DB(dbURL, dbUser, dbPass)
}

class UserManagement(dbConn: DBConnection) {
  val db = dbConn.db
  def findUser(id: Int): User =
    withTransaction(db) { implicit session =>
      session.query(s"select * from users where id = $id").map(resultsToUser)
    }
}

val dbConn = new PostgresDBConnection(
  "jdbc:postgresql://localhost:5432/data", "user", "pass")
val um = new UserManagement(dbConn)
val user = um.findUser(101)
```

Constructor Parameters in a Large System

The problem with constructor parameter passing is that it quickly becomes unmanageable

All parameters must be chained down from the top level to the components that need them, pretty soon you are going to end up with methods that look like this:

```
def wsLookupUserWeather(
  s3Conn: S3Connection,
  dbConn: DBConnection,
  wsClient: WebServiceClient,
  weatherService: WeatherService,
  retryPreferences: Retrying,
  emailer: EmailService,
  security: SecurityService
)
```

And so on. When a method needs a new dependency passed in, it must then be added all the way up to the top.

Run Time Dependency Injection

E.g. Guice:

```
class UserManagement @Inject()(dbConn: DBConnection) {
  val db = dbConn.db

  def findUser(id: Int): User =
    withTransaction(db) { implicit session =>
      session.query(s"select * from users where id = $id").map(resultsToUser)
    }
}

class SystemModule extends AbstractModule {
  val dbConn = new PostgresDBConnection(
      "jdbc:postgresql://localhost:5432/data", "user", "pass")

  override protected def configure(): Unit = {
    bind(classOf[DBConnection]).toInstance(dbConn)
    bind(classOf[UserManagement]).to(classOf[UserManagement])
  }
}

  // and in the main method:
  val injector = Guice.createInjector(new SystemModule)
  val userManagement = injector.getInstance(classOf[UserManagement])
```

So Why Not Guice (or Equivalent)

- Maybe it is a fit for you - certainly it's a popular solution, but...
- It's run time (cannot be verified at compile time)
- The syntax is kind of awkward (although libraries exist to improve that)
- Annotations are "so Java 2004"
- Put simply, we can do better in Scala without needing any libraries

Compile Time Dependency Injection

Patterns for compile-time verifiable dependency injection, without needing third party libraries

- Cake:
  - Used for the Scala compiler
  - Gained popularity in many projects
  - Uses self traits and abstract members
  - Some boilerplate
  - Fully compile time verifiable
  - Slow compile times (heavy trait usage)

- Parfait (aka Cup Cake)
  - A variant being used in Dotty
  - Derived through a series of simple, logical steps
  - Uses abstract traits, and implicit parameters
  - Little (but some) boilerplate
  - Fully compile time verifiable
  - Faster compile times

Cake

```
abstract class DBAccess {
    def database: Database     // abstract definition

    def findAgeOfPerson(name: String): Int =
        withTransaction(database.currentSession) { tx =>
            tx.find(name = "Fred").select("age")
        }
}

trait MySQLDB {
    this: DBAccess =>     // self type
    val database = MySQLDatabase     // provide concrete database
}

val access = new DBAccess with MySQLDB    // Inject
println(access.findAgeOfPerson("Fred"))
```

While still widely used, Cake has a number of issues, discussions of which can be found easily on the web.

Suggest researching to this pattern, compile times are a problem, more background: https://www.youtube.com/watch?v=yLbdw06tKPQ

Parfait (with Multiple Dependency Modules)

```
trait DBConfig {
   val db: Database
}

trait WSConfig {
   def ws: WeatherService
}

trait MySQLDB extends DBConfig {
    lazy val db = MySQLDatabase     // provide concrete database
}

trait WeatherUnder extends WSConfig {
   def ws: WeatherService = new WeatherUndergroundRest
}
```

Start with pure abstract traits and various concrete implementations

Parfait...

```
class DBAccess(implicit config: DBConfig) {
    def database: Database = config.db

    def findZip(city: String): Int =
        withTransaction(database.currentSession) { tx =>
            tx.find(name = city).select("zip")
        }
}

class WeatherAccess(implicit config: WSConfig) {
   def weatherService: WeatherService = config.ws
   def findTempForZip(zip: String): Float = 55.0F // .. implementation
}
```

A DBAccess that uses DBConfig, and a WeatherAccess that uses WSConfig, so far so simple...

Parfait...

```
class DoItAll(implicit dbWsConfig: DBConfig with WSConfig) {
   val dbAccessor = new DBAccess    // gets config implicitly

   def lookupTemp(city: String): Float = {
       val myWS = new WeatherAccess     // also gets config implicitly
       val zip = dbAccessor.findZip(city)
       myWS.findTempForZip(zip)
   }
}

implicit object StandardConfig extends MySQLDB with WeatherUnder
(new DoItAll).lookUpTemp("Denver")
```

Outer method provides both implicits. Both DBConfig and WSConfig are implicit throughout the scope of the DoItAll class body.

Alternatively, Group Configs:

```
trait AllConfig extends DBConfig with WSConfig

class DoItAll(implicit allConfig: AllConfig) {
   val dbAccessor = new DBAccess    // gets config implicitly

   def lookupTemp(city: String): Float = {
       val myWS = new WeatherAccess     // also gets config implicitly
       val zip = dbAccessor.findZip(city)
       myWS.findTempForZip(zip)
   }
}

implicit object StandardConfig extends AllConfig with MySQLDB with WeatherUnder
(new DoItAll).lookUpTemp(“Denver”)
```

Testing

```
test("DoItAll with mocks") {
  val mockDb = new MockDB
  val mockWs = new MockWS

  implicit object TestConfig extends AllConfig with MySQLDB with WeatherUnder {
    override db = mockDB
    override ws = mockWS
  }

  (new DoItAll).lookUpTemp("Denver") should be (55.0)
}
```

We don't need to mix in MySQLDB and WeatherUnder in this case since all abstracts are provided, but this demonstrates that you can bring in the standard definitions and override just the things you want to.

Traits Don't Have Constructor Parameters

They will be added in a future version of Scala

In the meantime your traits can either just specify abstract members for the values they need (which the classes can provide automatically by, e.g. `import DBConfig._`)

Or require the actual config required, use standard names:

```
trait DBFinder {
  implicit val dbConfig: DBConfig  // to be supplied by the class mixed in to
  // can now work just like the class does
}
```

Everything is still compile time verified.

Objects Don't Have Constructor Parameters

True, but the implicits can be passed into methods instead in those cases, e.g. the classic case of Actor companion object props:

```
object WeatherActor {
  def props(implicit allConfig: AllConfig): Props = {
    Props(new WeatherActor)  // class which needs injecting, we are back on track
  }
}
```

Other Alternatives:

Other possibilities exist, particularly if you are open to using third party libraries:

- Macwire - https://github.com/adamw/macwire - uses macros
- Reader monad from either ScalaZ or Cats:
  - http://blog.originate.com/blog/2013/10/21/reader-monad-for-dependency-injection/
  - http://eed3si9n.com/herding-cats/Reader.html