SOFTWAREMILL

The no-framework Scala Dependency Injection framework

Adam Warski

BuildStuff 2013

You don't need anything special to do Dependency Injection

We often over-complicate

Who Am I?

- **Day**: coding @ SoftwareMill
- **Afternoon**: playgrounds, Duplos, etc.
- **Evenings**: blogging, open-source
  - Original author of Hibernate Envers
  - ElasticMQ, Veripacks, MacWire
  - http://www.warski.org

What is Dependency Injection?

What is DI?

```
class PresentPackager {
  def wrap() {
    new RibbonSelector().selectRandom()
    …
  }
}
```

What is DI?

```
class PresentPackager(rs: RibbonSelector) {
  def wrap() {
    rs.selectRandom()
    …
  }
}
```

Yes, DI is just using parameters

Why?

- Restrict the knowledge of the class

```
class PresentPackager {

   def wrap() {
      new RibbonSelector()
         .selectRandom()
      …
   }
}
```

```
class PresentPackager
   (rs: RibbonSelector) {

   def wrap() {
      rs.selectRandom()
      …
   }
}
```

But still …

- We need to have the **news** somewhere

Let’s create a DI container!
a.k.a. framework

DI in Java

- Many frameworks
- Configuration via:
  - XML
  - annotations
  - Java

What's wrong with that?

- Do I really need a DI framework?

Let's go back ...

- … and just use our host language
- in this case, Scala
- mapping DI framework concepts to native language constructs

Manual DI!

```
object PresentWrapper extends App {

    val ribbonSelector =
            new RibbonSelector()

    val wrappingPaperFeeder =
            new WrappingPaperFeeder()

    val presentPackager =
            new PresentPackager(
                ribbonSelector,
                wrappingPaperFeeder)

}
```

Manual DI!

```
object PresentWrapper extends App {

    lazy val ribbonSelector =
            new RibbonSelector()

    lazy val wrappingPaperFeeder =
            new WrappingPaperFeeder()

    lazy val presentPackager =
            new PresentPackager(
               ribbonSelector,
               wrappingPaperFeeder)

}
```

MacWire

```
import com.softwaremill.macwire.MacwireMacros._

object PresentWrapper extends App {

   lazy val ribbonSelector =
           wire[RibbonSelector]

   lazy val wrappingPaperFeeder =
           wire[WrappingPaperFeeder]

   lazy val presentPackager =
           wire[PresentPackager]


}
```

Side-note: Scala Macros

Side-note: Scala Macros

- Scala code executed at compile time
- Operate on trees
- Can inspect the environment, generate code
  - the code is type-checked

Side-note: Scala Macros

- E.g. debug macro

```
def debug(params: Any*) = macro debug_impl

def debug_impl
       (c: Context)
       (params: c.Expr[Any]*): c.Expr[Unit]


debug(presentCount) ⟹

  println("presentCount = " + presentCount)
```

Side-note: Scala Macros

- Debug macro implementation

```
import c.universe._

val paramRep = show(param.tree)
val paramRepTree = Literal(Constant(paramRep))
val paramRepExpr = c.Expr[String](paramRepTree)

reify { println(
   paramRepExpr.splice +
   " = " +
   param.splice) }
```

Side-note: Scala Macros

- MacWire

```
def wire[T] = macro wire_impl[T]

def wire_impl
        [T: c.WeakTypeTag]
        (c: Context): c.Expr[T]
```

MacWire

```
import com.softwaremill.macwire.MacwireMacros._

object PresentWrapper extends App {
   lazy val ribbonSelector =
            wire[RibbonSelector]

   lazy val wrappingPaperFeeder =
            wire[WrappingPaperFeeder]

   lazy val presentPackager =
            wire[PresentPackager]
}
```

Scopes

- How long will an object (instance) live?

Singleton & dependent

```
object NorthPole extends App {
   // Singleton
   lazy val santaClaus = wire[SantaClaus]

   // Dependent
   def gnome = wire[Gnome]
}
```

Arbitrary scopes

```
trait WebFrontEnd {
   lazy val loggedInUser =
      session(new LoggedInUser)

   def session: Scope
}


trait Scope {
   def apply(factory: => T): T
}
```

Arbitrary scopes

```
object MyApp extends WebFrontEnd {
  val session: Scope =
        new ThreadLocalScope()

  val filter = new ScopeFilter(session)

  // bootstrap the web server
  // using the filter
}
```

Arbitrary scopes

```
class ScopeFilter(sessionScope: ThreadLocalScope)
        extends Filter {

   def doFilter(request: ServletRequest) {
      sessionScope
         .withStorage(request.getSession()) {

         request.proceed()

      }
   }
}
```

Modules

- Pre-wired
- Composable
- Dependencies
- Module per package?
    - Veripacks :)

Modules

- Module: `trait`
- Pre-wired: `new`, MacWire
- Composable: `extends/with`
- Dependencies: `extends/with` / abstract members

Modules

```
trait PresentWrapper {
   lazy val ribbonSelector =
          wire[RibbonSelector]
   lazy val wrappingPaperFeeder =
          wire[WrappingPaperFeeder]
   lazy val presentPackager =
          wire[PresentPackager]
}
```

Modules

```
trait PresentFactory extends PresentWrapper {
   lazy val teddyBearProvider =
           wire[TeddyBearProvider]
   lazy val toyTrainProvider =
           wire[ToyTrainProvider]
   lazy val presentAssembly =
           wire[PresentAssembly]
}
```

Modules

```
trait HomeOfSanta {
   lazy val santaClaus = wire[SantaClaus]
   lazy val rudolf = wire[Rudolf]
   lazy val fireplace = wire[Fireplace]

   def presentAssembly: PresentAssembly
}
```

Modules

```
trait PresentWrapper { … }
trait PresentFactory extends PresentWrapper { }
trait HomeOfSanta { … }


object NorthPole
   extends PresentWrapper
      with PresentFactory
      with HomeOfSanta {

   santaClaus.deliver()

}
```

Testing Santa's Home

```
class HomeOfSantaTest extends FlatSpec {
   it should "deliver presents" in {
      val mockPresentAssembly = …
      new HomeOfSanta {
         lazy val presentAssembly =
            mockPresentAssembly }
      …
   }
}
```

Cake Pattern

```
trait PresentPackagerModule {

   class PresentPackager {
      def wrap() {
         ribbonSelector.selectRandom()
         …
      }
   }


   lazy val presentPackager = new PresentPackager()
   def ribbonSelector: RibbonSelector
}
```

Cake Pattern

```
val cake = new PresentPackagerModule
   with RibbonSelectorModule
   with WrappingPaperFeederModule
   with TeddyBearProviderModule
   with ToyTrainProviderModule
   with PresentAssemblyModule
   with … { }
```

Other features

- Interceptors

```
trait Chimney {
  lazy val presentTransferer =
    transactional(wire[PresentTransferer])
  def transactional: Interceptor
}
```

Other features

- Factories
  - a dedicated object or …

```
trait PresentBoxer {
  def box(size: Size) = wire[Box]
}
```

Other features

- Instance maps
  - for integrating e.g. with Play
- Factories
- In-method wiring
- More coming, someday :)

Summing up

- Reconsider using a framework
- Native Scala gives a lot of power
  - use it
  - wisely
- More flexibility (less constraints)

Links

- http://www.warski.org
- https://github.com/adamw/macwire
- http://springsource.com/

Thanks!

- Questions?
- Stickers ->
- adam@warski.org