Java on Guice

Dependency Injection

the Java Way!

Bob Lee and Kevin Bourrillion
Google, Inc.
April 26, 2007

Does this sound like you?

- **"My application is easy to unit-test!"**
- **"I don't worry about Dependency Bloat!"**
- **"My code is clean! It has high signal-to-noise!"**
- **"My tests never mysteriously fail when run out of order!"**
- **"And I don't think I will have any of these problems either!"**

If so....

If that described you...

- **What the heck are you doing here??**
- **Go outside**
- **Enjoy the sunshine**

What we'll talk about

- **What's Guice?**
- **How do you use it?**

OK, any questions?

A Problem

Using `static` references comes with big problems!

- **Tight coupling: depending on implementation, not interface**
- **Not polymorphic! Not OO!**
- **It's extremely global (no scoping)**
- **Leads to "all-or-nothing" testing**
- **And it's viral! It gives you "static cling!"**

A radical idea...

What if you didn't need `static`?

We'd see some benefits right away!

- **All our references could be abstract**
- **Now you can test the parts you want to and mock the parts you don't**
- **Much less cleanup after tests**
- **Without the spread of static cling, we become free to think in terms of objects all the time**
- **Aaaaaaahhhh.**

But *how*?

Does a Factory Pattern help?

- **No, not really at all**
- **How are you going to get the Factory?**
- **You'll still depend on implementation, just once removed**
- **And enjoy all that boilerplate code!**

Does a Service Locator help?

- **Somewhat**
- **But how are you going to get the Service Locator?**
- **Hard to validate that requested services will be available**

- **Hard (say, for a test, or a tool) to tell what services a given object will need**
- **Difficult to reuse classes in different contexts**

These fail partly because they're "pull."

To work, your objects need to *already have* references to the other objects they need... somehow.

DI: an anti-static mat for your code

A dependency injector gives each of your objects the references to the other objects it needs (its "dependencies") before that object has to do anything.

Beats Service Locator because

- **we can validate up front!**
- **objects are usable without the framework**
- **dependency graph is explicit (tools can use it)**

What's Guice then?

Google Guice (pronounced "Guice") is a Java framework that does dependency injection for you.

It is a dependency injector.

That's it.

(well, kind of.)

Let the toy examples begin...

Using Guice: *reductio ad absurdum*

```
public class MyApplication {
    public static void main(String[] args) {
    Injector injector = Guice.createInjector();
    Greeter greeter
           = injector.getInstance(Greeter.class);
    greeter.sayHello();
  }
}

 public class Greeter {
 void sayHello() {
     System.out.println("Hello, world!");
   }
 }
```

Let's inject something! (Field injection)

```
public class MyApplication {
  public static void main(String[] args) {
    Injector injector = Guice.createInjector();
    Greeter greeter
           = injector.getInstance(Greeter.class);
    greeter.sayHello();
  }
}
public class Greeter {
  @Inject Displayer displayer;
  void sayHello() {
    displayer.display("Hello, world!");
  }
}
public class Displayer {
  void display(String s) { out.println(s); }
}
```

What is this "application?"

```
public class MyApplication {
  public static void main(String[] args) {
    Guice.createInjector()
       .getInstance(Greeter.class)
       .sayHello();
  }
}
```

The Application...

- **has your `main()` method (or other entry point(s))**
- **creates the `Injector`**
- **"bootstraps"**

It should have no need to store and publish the Injector reference!

We'll omit this class on most of the following slides.

Method injection

```
public class Greeter {
   Displayer displayer;

  @Inject void setDisplayer(Displayer displayer) {
     this.displayer = displayer;
   }
  void sayHello() {
    displayer.display("Hello, world!");
  }
}
```

`setDisplayer()` can be named anything you want, and can take multiple parameters.

Constructor injection

```
public class Greeter {
```

```
    final Displayer displayer;

  @Inject Greeter(Displayer displayer) {
      this.displayer = displayer;
    }
  void sayHello() {
   displayer.display("Hello, world!");
  }
}
```

Constructor injection to the nth

```
public class Greeter {
    final Displayer displayer;
 final Person person;

 @Inject Greeter(
        Displayer displayer, Person person) {
      this.displayer = displayer;
      this.person = person;
    }
  void sayHello() {
   displayer.display("Hello, " + person);
  }
}
```

Note: only one constructor can be marked with `@Inject`. If none are, Guice expects a default constructor.

Breaking a dependency

Now let's extract an interface for our displayer.

```
public class Greeter {
  @Inject Displayer displayer;
  void sayHello() {
   displayer.display("Hello, world!");
  }
}
public interface Displayer {
  void display(String s);
```

```
}
 public class StdoutDisplayer implements Displayer {
  public void display(String s) {
      out.println(s);
    }
}
```

Tell me what you want!

But in the previous slide, how would Guice have any idea which implementation class you wanted to use? Well, it doesn't!

```
Error at example.Greeter.displayer(Greeter.java:10) Binding
to example.Displayer not found. No bindings to that type
were found.
```

Clearly, Guice wants a "binding." For us to supply this, we will introduce another new concept first...

Modules

```
public class DisplayModule implements Module {
  public void configure(Binder binder) {
   binder.bind(Displayer.class).to(StdoutDisplayer.class);
  }
 }
```

Modules supply extra information to Guice.

Usually, this is bindings, which instruct Guice how to fulfill requests for a particular (exact!) type.

Hooking up your new Module

Tell Guice about your module, and now our sample app is working again.

```
public class MyApplication {
  public static void main(String[] args) {
    Injector injector
        = Guice.createInjector(new DisplayModule());
    Greeter greeter
        = injector.getInstance(Greeter.class);
    greeter.sayHello();
  }
}
```

Working again? But there's no XML! *How can our Java code work with no XML??*

Modules and Applications

- **Modules and Applications are many-to-many.**
  - **One Application will usually contain many Modules. (`createInjector()` is a varargs method.)**
  - **One Module can be used in any number of different Applications.**
- **The granularity of a Module? Whatever you want it to be. A unit of reuse.**
- **Modules may install other Modules**
  - **But you may or may not want to do this**
  - **In Guice 1.0 this inclusion graph must be a strict tree; this has been relaxed to a DAG for 1.1.**

How do I use Guice to unit-test?

You don't! Your class is already unit-testable.

```
public class GreeterTest extends TestCase {
  static class MockDisplayer implements Displayer {
    String received;
```

```
    public void display(String s) {
      assertNotNull(s);
      assertNull(received);
      received = s;
    }
  }
  public void test() {
    MockDisplayer mock = new MockDisplayer();
    Greeter realGreeter = new Greeter(mock);
    realGreeter.sayHello();
    assertEquals("Hello, world!", mock.received);
  }
}
```

Multiple implementations per type

Of course, some would say that an interface with only one implementation is like a cactus with only one fishing pole. (???)

Now the owner of `DisplayModule` wishes to support not only `StdoutDisplayer` but also `TimesSquareDisplayer`. The interface these present to the client code is just `Displayer` in both cases. But if you try

```
public class DisplayModule implements Module {
  public void configure(Binder binder) {
    binder.bind(Displayer.class).to(StdoutDisplayer.class);

binder.bind(Displayer.class).to(TimesSquareDisplayer.class);
  }
}
```

You'll see

```
 1) Error at
example.DisplayModule.configure(DisplayModule.java:22):
    A binding to example.Displayer was already configured at
    example.DisplayModule.configure(DisplayModule.java:21).
```

You can't bind the same type twice. Not within one module, and not across separate modules that you include in the same

application. Unless...

Using binding annotations

In the simple case, each client of Displayer knows at compile-time which "flavor" of implementation it would like (although not which actual class). We can use a binding annotation to distinguish between the flavors.

```
public class DisplayModule implements Module {
  public void configure(Binder binder) {
    binder.bind(Displayer.class)
          .annotatedWith(ForStdout.class)
          .to(StdoutDisplayer.class);
    binder.bind(Displayer.class)
     .annotatedWith(ForTimesSquare.class)
          .to(TimesSquareDisplayer.class);
  }
  }
public class NewGreeter {
  @Inject @ForTimesSquare Displayer displayer;
  void sayHello() {
    displayer.display("Hello, tourists!");
  }
 }
```

Creating binding annotations

Where did these annotation types like `@ForTimesSquare` come from?

You just create these yourself, whenever you need them. Annotation types are cheap! The syntax looks like...

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.PARAMETER})
@BindingAnnotation
public @interface ForTimesSquare {}
```

Hmm... well... you get used to it.

Again: annotation types are cheap!

Named bindings

Or if you really don't want to create your own annotation, you can use Guice's bundled support for named bindings.

```
public class DisplayModule implements Module {
  public void configure(Binder binder) {
    binder.bind(Displayer.class)
     .annotatedWith(Names.named("T1M3Z 5QUAR3!!!!1"))
           .to(TimesSquareDisplayer.class);
    }
}
public class TimesSquareGreeter {
  @Inject @Named("T1M3Z 5QU4R3!!!!1")
    Displayer displayer;

  void sayHello() {
    displayer.display("Hello, Cleveland!");
  }
}
```

These names live in a totally unmanaged global namespace and are less safe and tool-friendly. Ew.

Constant bindings

In Guice, constants are typeless.
You might bind them with one type and inject them in another.
We support certain conversions.

```
bindConstant()
   .annotatedWith(TheAnswer.class)
   .to("42");

@Inject @TheAnswer int answer;
```

"It depends!"

In that last example, each client knows at compile-time which "flavor" it always wants. We described this as being "the simplest case." But sometimes the answer to "which dependency do you want?" is "it depends!"

```
public class DisplayerPicker {
  public Displayer get() {
    switch (new Instant().getDayOfWeek()) {
      case DateTimeConstants.TUESDAY:
        return new TimesSquareDisplayer();
      default:
        return new StdoutDisplayer();
    }
  }
}
```

Introducing the custom provider

If Guice doesn't know how to create your instances, tell it who does know.

```
public class DisplayModule implements Module {
  public void configure(Binder binder) {
    binder.bind(Displayer.class)
     .toProvider(DisplayerPicker.class);
     }
}
```

Introducing the customer provider (cont)

```
public class DisplayerPicker
      implements Provider<Displayer> {
  public Displayer get() {
    switch (new Instant().getDayOfWeek()) {
      case DateTimeConstants.TUESDAY:
        return new TimesSquareDisplayer();
      default:
        return new StdoutDisplayer();
```

```
    }
  }
 }
```

`DisplayPicker` is a first-class citizen like any other -- it can have injections, etc.

In Guice 1.1, a cooler way to do this is (hopefully) coming!

Uh oh

There is a slight problem with this implementation though!

```
switch (new Instant().getDayOfWeek()) {
  case TUESDAY: return new TimesSquareDisplayer();
  default:    return new StdoutDisplayer();
}
```

When you see `new`, Guice is not the one instantiating, so the object is not getting injected.

`StdoutDisplayer` may have `@Inject` annotations, but they'll do nothing, because this instance is not "in the club."

You want to let the Injector do most of your instantiating for you. Think of it as a complement to the garbage collector.

An imperfect solution

```
public class DisplayerPicker
      implements Provider<Displayer> {
    @Inject TimesSquareDisplayer tsd;
    @Inject StdoutDisplayer sd;

  public Displayer get() {
    switch (new Instant().getDayOfWeek()) {
     case TUESDAY: return tsd;
     default:    return sd;
    }
  }
```

```
}
```

No changes to our Module, as long as all our classes are injectable.

The slight problem is that it contains a false dependency. If we care about this, there is a way to fix it...

Time to twist your brains!

Not only can you *implement* `Provider` when you have something to provide; you can also *receive* a `Provider` and use it when you need something provided to you.

```
public class DisplayerPicker
      implements Provider<Displayer> {
    @Inject Provider<TimesSquareDisplayer> tsdp;
    @Inject Provider<StdoutDisplayer> sdp;

  public Displayer get() {
    switch (new Instant().getDayOfWeek()) {
     case TUESDAY: return tsdp.get();
     default:    return sdp.get();
    }
  }
  }
```

Provider Injection

What we've just seen is just one example of provider injection.

We didn't need any special bindings. If you can `@Inject Foo`, you can `@Inject Provider<Foo>`, always.

Other uses:

- **Get multiple instances**
- **Be very lazy**
- **And more...**

If you bind a provider, and inject a provider, they might not be the same instance!

Scopes

- **A scope is a policy for reusing injected instances**
- **Default: no scope**
- **Create, inject, forget**
- **Providers are no different - they can have scopes**
- **Providers don't need to scope what they return (orthogonal)**
- **`com.google.inject` defines one scope: Singleton**
- **You can write your own, too**

Singletons

- **Singletons are not evil as you've been told**
- **If we stop thinking of "singleton" as something a class is**
- **Think of it as just a way the class is used - a property of the module**
- **Applications are free to use it how they want to**
- **Tests aren't prevented from doing whatever they need to**

Two ways to make a singleton

```
public void configure(Binder binder) {
  binder.bind(TimesSquareDisplayer.class)
    .in(Singleton.class);
  }
```

Or

```
@Singleton
 public class TimesSquareDisplayer { ... }
```

The Module takes precedence!

Two ways to make an eager singleton

```
public void configure(Binder binder) {
 binder.bind(TimesSquareDisplayer.class)
   .asEagerSingleton();
 }
```

Or you can let your Module itself do the instantiating...

```
public void configure(Binder binder) {
 binder.bind(TimesSquareDisplayer.class)
   .toInstance(new TimesSquareDisplayer());
 }
```

We prefer the first way!

Guicing a web application

- **Plugs in to Struts2, WebWork, . . .**
- **Actions created on every request**
- **Now Actions, Interceptors are "in the club"**
- **No more -Aware and manual DI**

Servlet scopes

- `@RequestScoped`
- `@SessionScoped`
- **future:** `@ConversationScoped`
- **"Two-way" guarantee**
- **How to handle backward scope access**

How to choose a scope

- **If the object is stateless and inexpensive - who cares?**
- **If the object is stateful, it should be obvious (?)**
- **If using Singleton or SessionScoped, be careful about concurrency!**
- **For RequestScoped and "no scope", no worries about concurrency**

- **Singleton is for expensive-to-construct or resource-hogging objects**
- **Session scope is overrated**
- **You may need to write a custom scope (transaction scope?)**

Field vs. Method vs. Constructor Injection

Field injection

- **Most compact syntax (great for slides!)**
- **Can't take any special action upon injection**
- **Your class is not testable!**

Method injection

- **Isn't field injection**
- **Only thing that works for some strange edge cases**

Field vs. Method vs. Constructor Injection

Constructor injection

- **Fields can be `final`**
- **Cannot possibly have been skipped**
- **What the idea of construction is all about**
- **Easy to see dependencies at a glance**
- **Doesn't support optional injections**
- **Useless in certain cases, like servlets**
- **Subclasses need to "know about" the injections needed by their superclasses**
- **Less convenient for tests that only "care about" one of the parameters**

Any questions?

A guide to the colors used in this presentation:

- **Blue = I felt like putting something in Blue**
- **Green = I felt like putting something in Green**
- **Red = I felt like putting something in Red**
- **Orange = I felt like putting something in Orange**