Guice

Java DI Framework

Agenda

- Intro to dependency injection
- Cross-cutting concerns and aspect-oriented programming
- Intro to Guice
- More Guice

What is DI?

Dependency injection is a design pattern that's like a "super factory".

Like factories...

- Dependency injection separates users of an interface from the interface's implementation
- It enables some measure of hot-swapping classes in production
- It increases **encapsulation** and **polymorphism**, since interface users don't have undue control over which implementation they get

What is DI?

But factories...

- Are often implemented separately for each interface to implementation binding
- Are more difficult to keep in sync if one implementation needs to exist with another in order to make sense -- more on this later
- Are scattered among classes that perform actual application logic, so finding and modifying them is more difficult
- Usually require setup and teardown code, introducing global state into the application code and making testing more complex

What is DI?

A simple way to think of dependency injection is, rather than using "new" or implementing a factory for each variable, the variables are passed in as constructor arguments.

This sounds like a good way of solving our problems with factories (now bindings can be centralized and separated from application code, for example), but doing it manually requires backtracking through large object trees -- each object in the chain needs its variables to be injected as well.

So why should we do this?

Cross-cutting concerns & AOP

Cross-cutting concerns occur when many orthogonal tasks exist in a system, often in a way that forces their code to be intermingled.

This is **extremely common** in production code.

Cross-cutting concerns & AOP

Cross-cutting concerns, scenario 1:

- Imagine a system that requires authentication before doing a wide variety of tasks
- Many classes must make use of a session manager object
- If we're not careful, switching our session manager's implementation to a new system will be tedious and mistake-prone, since it's sprinkled everywhere
- For example, we might need a completely different session manager when we're testing the application, complete with test user accounts, but we don't want side effects of this session manager appearing in production

Cross-cutting concerns & AOP

Cross-cutting concerns, scenario 2:

- We would like to do analytics on user interactions and engagement throughout many areas of the application
- Analytics data must be gathered, formatted, and sent to a remote server for analysis
- Every analytics provider has vastly different format requirements
- What happens if we need to change analytics providers? How about if we need to hook into a sandbox analytics system when we're testing the application?

Cross-cutting concerns & AOP

Aspect-oriented programming tries to separate these cross-cutting concerns into independent, easy-to-vary **aspects.**

This isn't a programming paradigm in the same way as, say, OOP or functional programming, but rather an aspect (snerf) that can be applied to any programming style.

Cross-cutting concerns & AOP

Aspects don't necessarily correspond to a single class -- in fact, they are usually broader.

Example:

For testing purposes, you might write output to a local file rather than to a database. Not only does the actual output routine have to change, but the output formatting needs to change as well, in order to be easily user-readable.

Cross-cutting concerns & AOP

This is where factories can fall short. Making aspects switchable requires a lot of boilerplate, often across multiple separate factory classes.

Intro to Guice

Guice represents these aspects in the form of modules, collections of related interface-implementation bindings that can be easily swapped out.

Let's look at a module.

Intro to Guice

```
public class TestOutputModule extends
AbstractModule {
  @Override
  protected void configure() {
                        bind(OutputWriter.
     class).to(FileOutputWriter.class);
     bind(OutputFormatter.class).to
     (TestOutputFormatter.class);

  }
}
```

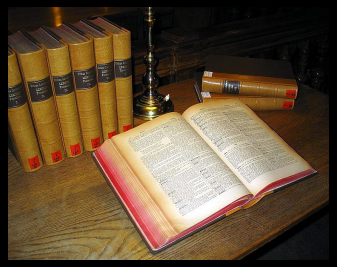
Intro to Guice

When we want to write a module, we usually start by overriding its configure function.

```
bind(Interface.class).to(Implementation.class);
```

is a pretty common use pattern in Guice. What this tells Guice is "any time you are asked to inject Interface, create a new instance of Implementation".

Intro to Guice



In order to use our module, Guice needs to know which module to use, and it needs to know where it should insert instances of Implementation.

First, in some fairly global area, usually main(), create a new injector:

Injector injector = Guice.createInjector(new TestOutputModule());

Intro to Guice

Now, for classes participating in dependency injection, we need to tell Guice which constructor to use. We do this with an @Inject annotation.

For example, assume our OutputFormatter interface has one method, format().

Intro to Guice

```
class TestOutputFormatter implements
OutputFormatter {

  public void format() {
     System.out.println("formatted!");
   }
}
```

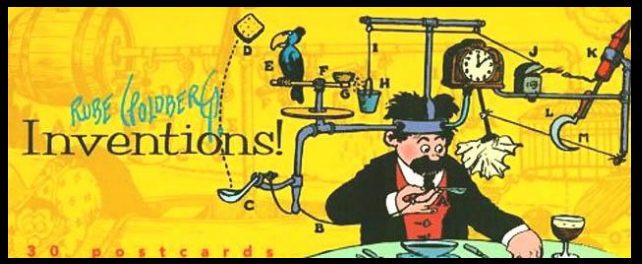
Intro to Guice

Now we can go back to our main loop and create an OutputFormatter using the injector:

```
Injector injector = Guice.createInjector(new
TestOutputModule());
```

OutputFormatter formatter = injector.
getInstance(OutputFormatter.class);

Intro to Guice



For this class, Guice used the default constructor, realized that it didn't need any parameters passed in, and simply returned a new TestOutputFormatter.

At first, this seems like crazy overkill. Right now, it is! We imported Guice, made a whole new class, and initialized an injector. Using "new TestOutputFormatter()" or even homerolling a factory would have been easier.

Intro to Guice

Here's where it starts to get better.

Let's say our TestOutputWriter requires an OutputFormatter to organize its output.

Of course, it also needs a write() method, like other OutputWriters.

Intro to Guice

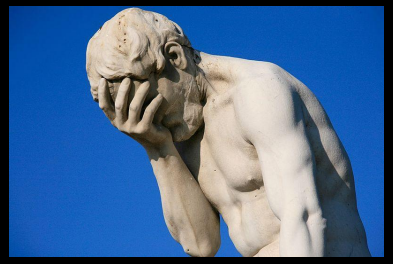
```
public class FileOutputWriter implements OutputWriter {
    private final OutputFormatter _formatter;

    public FileOutputWriter(OutputFormatter formatter) {
        _formatter = formatter;
    }

    @Override
    public void write() {
        _formatter.format();
        System.out.println("writing to file!");
    }

}
```

Intro to Guice

And here's our new main function:

```
public static void main(String[] args) {
	Injector injector = Guice.createInjector(new
TestOutputModule());
	OutputWriter writer = injector.getInstance
(OutputWriter.class);
	writer.write();
}
```

Intro to Guice



But this won't run! Instead, if we try to run it, we'll get a unhelpful Guice error about creating injectors.

What we've forgotten is the @Inject annotation, which tells Guice's injector to use the annotated constructor to create the object, and to inject any arguments provided.

Intro to Guice

```
public class FileOutputWriter implements OutputWriter {
    private final OutputFormatter _formatter;

    @Inject
    public FileOutputWriter(OutputFormatter formatter) {
        _formatter = formatter;
    }

    @Override
    public void write() {
        _formatter.format();
        System.out.println("writing to file!");
    }

}
```

Intro to Guice

Now our main function should run.

But what if we want to pass an integer flag to the TestOutputFormatter?

Intro to Guice

```
class TestOutputFormatter implements OutputFormatter {

  private final Integer_flag;

  @Inject
  public TestOutputFormatter(Integer flag) {
       _flag = flag;
  }

  public void format() {
       System.out.println("formatted with flag " + _flag);
    }
}
```

Intro to Guice

Now we're passing in a literal value, but how do we tell Guice what value to use?

We'll have to add another binding, but this time an *instance* binding.

Intro to Guice

```
public class TestOutputModule extends AbstractModule {
  @Override
  protected void configure() {
                        bind(OutputWriter.class).to
      (FileOutputWriter.class);
      bind(OutputFormatter.class).to(TestOutputFormatter.
      class);
      bind(Integer.class).annotatedWith(Names.named
      ("formatflag")).toInstance(1);
  }
}
```

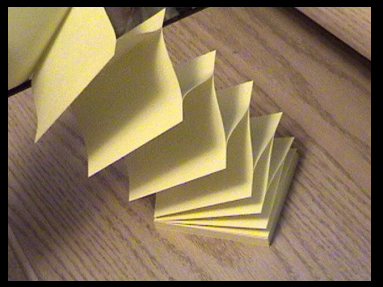
Intro to Guice

Annotate the constructor argument accordingly:

public TestOutputFormatter(@Named("format flag") Integer flag)

and the code should now run properly without Guice warnings.

Intro to Guice



More broadly, this allows for consistent, module-wide constants.

Note that we can also used the annotatedWith specifier on interface-implementation bindings.

Further capabilities of Guice...

More Guice

Providers

Sometimes, an object needs more than its constructor to be ready for action. Maybe it needs to be constructed, then have a delegate set sometime afterward. Likely it's a class we don't have full control over.

So, we implement **Provider** to create a factory for the troublesome class.

More Guice

```
public class InterfaceProvider implements Provider<Interface> {
  private final DelegateInterface _delegate;

  @Inject
  public InterfaceProvider(DelegateInterface delegate) {
    _delegate = delegate;
  }

  public get() {
    Interface interface = new Implementation();
    interface.setDelegate(_delegate);
    return interface;
  }
}
```

More Guice

Now we create a different kind of binding. Instead of interface-implementation or class-instance,

```
bind(Interface.class).toProvider(InterfaceProvider.class);
```

When Guice needs an instance of Interface, it will construct the provider, then call get() to retrieve the provided object.

More Guice

Singleton objects

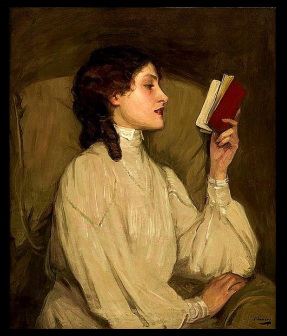
Annotate an **implementation** with @Singleton to tell Guice it should reuse one object instead of creating a new instance every time it injects the class.

More Guice

Things to try out:

- Guice resolves bindings at runtime, so an alternative to "production" and "test" modules is one module reading bindings from production.properties or test.properties.
  - Usual warnings about introspection apply, and errors will be especially hairy due to Guice's confusion
- JUnit tests pair well with Guice, try expanding on the snippets presented in the slides and writing some unit tests.
- **Guice errors have a steep learning curve. If you want to be comfortable with Guice, try to make the example code fail in various ways.**

Further reading



- http://code.google.com/p/google-guice/wiki/GettingStarted
- http://www.youtube.com/watch?feature=player_embedded&v=hBVJbzAagfs