Dependency injection with Guice

Testable code with less boilerplate

Nicholas Lesiecki (ndlesiecki@apache.org)
Software engineer
Google

09 December 2008

Guice is Google's open source dependency injection framework for Java™ development. It enables better testing and modularity by taking away the pain of writing your own factories. Nicholas Lesiecki offers a tour of the most important Guice concepts that will leave you ready to Guice up your applications.

Guice is a *dependency injection* (DI) framework. I've suggested for years that developers use DI, because it improves maintainability, testability, and flexibility. By watching engineers react to Guice, I've learned that the best way to convince a programmer to adopt a new technology is to make it really easy. Guice makes DI really easy, and as a result, the practice has taken off at Google. I hope to continue in the same vein in this article by making it really easy for you to learn Guice.

A tour, not a polemic

Guice followed on the heels of a family of groundbreaking DI frameworks. (The folks at PicoContainer, one of the frameworks in question, have a page explaining the history and interrelationships; see Resources.) Guice's late arrival sparked commentary on which framework was better, and whether yet another DI framework was necessary. As with any technology choice, each library has its pros and cons. I think Guice brings something new to the table, but I'll stick to a tour of Guice's features in this article rather than add to the debate. (You can search on the Web for "guice vs spring" for some lively discussion.)

Guice 2.0 beta

As I write this, the Guice team is working hard on Guice 2.0 and expects to release before the end of 2008. An early beta is posted on the Google Code download site (see Resources). This is good news, because the Guice team has added features that will make your Guice code easier to use and understand. The beta lacks some features that will make it into the final version, but it's stable and high quality. In fact, Google uses the beta version in production software. I advise you to do the same. I've written this article specifically for Guice 2.0, covering some new Guice features and glossing over features from 1.0 that have been deprecated. The Guice team has assured me that the features I cover won't change between the current beta and final release.

If you already understand DI and know why you'd want a framework to help you with it, you can skip to the Basic injection with Guice section. Otherwise, read on to learn about DI's benefits.

The case for DI

I'll start with an example. Let's say I'm writing a superhero application, and I'm implementing a hero named Frog Man. Listing 1 contains the code, as well as my first test. (I hope I don't need to convince you of the value of writing unit tests.)

Listing 1. A basic hero and his test

```
public class FrogMan {
  private FrogMobile vehicle = new FrogMobile();
  public FrogMan() {}
  // crime fighting logic goes here...
}

public class FrogManTest extends TestCase {
 public void testFrogManFightsCrime() {
    FrogMan hero = new FrogMan();
    hero.fightCrime();
    //make some assertions...
  }
}
```

All seems well until I try running the test, whereupon I get the exception in Listing 2:

Listing 2. Dependencies can be troublesome

```
java.lang.RuntimeException: Refinery startup failure.
  at HeavyWaterRefinery.<init>(HeavyWaterRefinery.java:6)
  at FrogMobile.<init>(FrogMobile.java:5)
  at FrogMan.<init>(FrogMan.java:8)
  at FrogManTest.testFrogManFightsCrime(FrogManTest.java:10)
```

It seems that the `FrogMobile` constructs a `HeavyWaterRefinery` and, well, let's just say there's no way I can construct one of those in my test. I can do so in production, sure, but no one will grant me a second refinery permit just for testing. In real life, you're not likely to refine deuterium oxide, but you are likely to depend on remote servers and beefy databases. The principle is the same: These dependencies are hard to start and slow to interact with, and they cause your tests to fail more often than they should.

Enter DI

To avoid this problem, you can create an interface (for example, `Vehicle`) and have your `FrogMan` class accept the `Vehicle` as a constructor argument, as in Listing 3:

Listing 3. Depend on interfaces, and have them injected

```
public class FrogMan {
  private Vehicle vehicle;

  public FrogMan(Vehicle vehicle) {
    this.vehicle = vehicle;
  }
  // crime fighting logic goes here...
}
```

This idiom is the essence of DI — have your classes accept their dependencies through references to interfaces instead of constructing them (or using static references). Listing 4 shows how DI makes your test easier:

Listing 4. Your test can use mocks instead of troublesome dependencies

```
static class MockVehicle implements Vehicle {
  boolean didZoom;

  public String zoom() {
    this.didZoom = true;
    return "Mock Vehicle Zoomed.";
  }
}

public void testFrogManFightsCrime() {
  MockVehicle mockVehicle = new MockVehicle();

  FrogMan hero = new FrogMan(mockVehicle);
  hero.fightCrime();

  assertTrue(mockVehicle.didZoom);
  // other assertions
}
```

This test uses a hand-written mock object to replace the `FrogMobile`. Not only does DI free the test from the painful refinery startup cost, but it also keeps the test from knowing about `FrogMobile`. All it needs is the `Vehicle` interface. In addition to making tests easier, DI also helps your code's overall modularity and maintainability. Now, if you want to switch the `FrogMobile` for the `FrogBarge`, you can do so without modifying `FrogMan`. All `FrogMan` depends on is the interface.

There's a catch, however. If you're like me the first time I read about DI, you're thinking: "Great, now all `FrogMan`'s *callers* have to know about the `FrogMobile` (and the refinery, and the refinery's dependencies, and so on...)." But if that were true, DI would never have caught on. Instead of forcing callers to assume the burden, you can write *factories* to manage the creation of an object and its dependencies.

Factories are where frameworks come in. Factories require a lot of tedious, repetitive code. In the best case, they annoy the program author (and readers), and in the worst, they never get written because of the inconvenience. Guice and other DI frameworks serve as flexible "super factories" that you configure to build your objects. Configuring the framework is a lot easier than writing your own factories. As a result, programmers write more code in a DI style. More tests, better code, and happy programmers follow.

Basic injection with Guice

I hope I've convinced you that DI adds value to your designs and that using a framework will make your life even easier. Let's dive into Guice, starting with the `@Inject` annotation and modules.

Tell Guice you want your class `@Inject`-ed

The only difference between `FrogMan` and `FrogMan` on Guice is `@Inject`. Listing 5 shows `FrogMan`'s constructor with the annotation:

Listing 5. `FrogMan` has been `@Injected`

```
@Inject
public FrogMan(Vehicle vehicle) {
  this.vehicle = vehicle;
}
```

Some engineers dislike the idea of adding an `@Inject` to their class. They prefer that a class be totally ignorant of the DI framework. This is a reasonable point, but I'm not convinced by it. As dependencies go, an annotation is pretty mild. The `@Inject` tag takes on meaning only when you ask Guice to construct your class. If you don't ask Guice to create `FrogMan`, the annotation has no effect on the code's behavior. The annotation provides a nice clue that Guice will participate in the construction of the class. However, using it does require source-level access. If the annotation bothers you or you're using Guice to create objects whose source you don't control, Guice has an alternate mechanism (see the Other uses for provider methods sidebar later in the article).

Tell Guice which dependency you want

Now that Guice knows that your hero needs a `Vehicle`, it needs to know which `Vehicle` to provide. Listing 6 contains a `Module` : a special class that you use to tell Guice which implementations go with which interfaces:

Listing 6. The `HeroModule` binds `Vehicle` to `FrogMobile`

```
public class HeroModule implements Module {
  public void configure(Binder binder) {
    binder.bind(Vehicle.class).to(FrogMobile.class);
  }
}
```

A module is an interface with a single method. The `Binder` that Guice passes to your module lets you tell Guice how you want your objects constructed. The binder API forms a *domain-specific language* (see Resources). This minilanguage lets you write expressive code such as `bind(X).to(Y).in(Z)`. You'll see more examples of what the binder can do as we proceed. Each call to `bind` creates a *binding*, and the set of bindings are what Guice uses to resolve injection requests.

Bootstrap with an `Injector`

Next, you bootstrap Guice using the `Injector` class. Typically you want to create your injector very early in your program. This lets Guice create most of your objects for you. Listing 7 contains an example main program that starts a heroic adventure using an `Injector`:

Listing 7 Bootstrapping your application with an `Injector`

```
public class Adventure {
  public static void main(String[] args){
    Injector injector = Guice.createInjector(new HeroModule());
    FrogMan hero = injector.getInstance(FrogMan.class);
    hero.fightCrime();
  }
}
```

To get an injector, you call `createInjector` on the Guice class. You pass `createInjector` a list of modules it should use to configure itself. (This example has only one, but you could add a `VillainModule` that configures evildoers.) Once you have the injector, you ask it for objects with `getInstance`, passing in the `.class` you'd like to get back. (Astute readers will notice that you don't need to tell Guice about `FrogMan`. It turns out that if you ask for a concrete class, and it has an `@Inject` constructor or public no-argument constructor, Guice creates it without needing a call to `bind`.)

This is the first way of getting Guice to construct your objects: asking explicitly. However, you don't want to do this outside of your bootstrapping routine. The better, easier way is to have Guice inject your dependencies, and the dependencies' dependencies, and so on. (As the saying goes, "it's turtles all the way down"; see Resources). This will seem disconcerting at first, but after a while you'll get used to it. As an example, Listing 8 shows the `FrogMobile` having its `FuelSource` injected:

Listing 8. The `FrogMobile` accepting a `FuelSource`

```
@Inject
public FrogMobile(FuelSource fuelSource){
  this.fuelSource = fuelSource;
}
```

This means that when you retrieve `FrogMan`, Guice constructs a `FuelSource`, a `FrogMobile`, and then finally a `FrogMan`, even though your application interacted with the injector only once.

Of course, you don't always get the chance to control your application's `main` routine. Many Web frameworks, for example, automagically construct "actions," "templates," or something else that serve as your starting point. You can usually find a place to shim Guice in, either with a plug-in for the framework or some handwritten code of your own. (For example, the Guice project has released a plug-in for Struts 2 that lets Guice configure your Struts actions; see Resources.)

Other forms of injection

So far, I've shown `@Inject` applied to constructors. When it finds this annotation, Guice picks through the constructor arguments and tries to find a configured binding for each of them. This is known as *constructor injection*. According to the Guice best practices guide, constructor injection is the preferred way to ask for your dependencies. But it's not the only way. Listing 9 shows another way to configure the `FrogMan` class:

Listing 9. Method injection

```
public class FrogMan{
  private Vehicle vehicle;

  @Inject
  public void setVehicle(Vehicle vehicle) {
    this.vehicle = vehicle;
  }
//etc. ...
```

Notice that I've gotten rid of the injected constructor and instead have a method tagged with `@Inject`. Guice calls this method immediately after it constructs my hero. Fans of the Spring

Framework may see this as "setter injection." However, Guice cares only about `@Inject`; your method can be named anything you like, and it can take multiple parameters. It can be package-protected or private too.

If you think Guice's decision to access private methods seems intrusive, wait until you see Listing 10, in which `FrogMan` uses *field injection*:

Listing 10. Field injection

```
public class FrogMan {
  @Inject private Vehicle vehicle;
  public FrogMan(){}
//etc. ...
```

Again, all Guice cares about is the `@Inject` annotation. It finds any fields you annotate and tries to inject the appropriate dependency.

Which one is best?

All three versions of `FrogMan` exhibit the same behavior: Guice injects the appropriate `Vehicle` when they're constructed. However, I prefer constructor injection, as do Guice's authors. Here's a quick analysis of the three styles:

- **Constructor injection** is straightforward. Because Java technology guarantees constructor invocation, you don't need to worry about objects arriving in an uninitialized state — whether or not Guice creates them. You can also mark your fields `final`.
- **Field injection** harms testability, especially if you mark the fields `private`. That defeats one of the main aims of DI. You should use field injection only in very limited circumstances.
- **Method injection** can be useful if you don't control a class's instantiation. You can also use it if you have a superclass that needs some dependencies. (Constructor injection makes this difficult.)

Selecting your implementation

So let's say you have more than one `Vehicle` in your application. The equally heroic Weasel Girl can't drive the `FrogMobile`! At the same time, you don't want to hardcode a dependency on the `WeaselCopter`. Guice solves this problem by letting you annotate your dependencies. Listing 11 shows Weasel Girl requesting a faster mode of transport:

Listing 11. Use annotations to request a specific implementation

```
@Inject
public WeaselGirl(@Fast Vehicle vehicle) {
  this.vehicle = vehicle;
}
```

In Listing 12, the `HeroModule` uses the binder to tell Guice that the `WeaselCopter` is "fast":

Listing 12. Tell Guice about your annotation in your `Module`

```
public class HeroModule implements Module {
 public void configure(Binder binder) {
    binder.bind(Vehicle.class).to(FrogMobile.class);
    binder.bind(Vehicle.class).annotatedWith(Fast.class).to(WeaselCopter.class);
  }
}
```

Notice that I chose an annotation that describes the sort of vehicle I want in abstract terms (`@Fast`) rather than ones that are too closely tied to the implementation (`@WeaselCopter`). If you use annotations that describe the intended implementation too precisely, you create an implicit dependency in the minds of your readers. If you use `@WeaselCopter` and Weasel Girl borrows the Wombat Rocket, it could confuse programmers reading or debugging the code.

To create the `@Fast` annotation, you need to copy the boilerplate in Listing 13:

Listing 13. Copy-paste this code to create a binding annotation

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

If you write a lot of `BindingAnnotations`, you'll wind up with a lot of these little files, each of which differs only in the name of the annotation. If you find this really annoying or want to do some quick prototyping, you could consider Guice's built-in `@Named` annotation, which accepts a string attribute. Listing 14 illustrates that alternative:

Listing 14. Using `@Named` instead of a custom annotation

```
// in WeaselGirl
@Inject
public WeaselGirl(@Named("Fast") Vehicle vehicle) {
  //...
}

// in HeroModule
binder.bind(Vehicle.class)
  .annotatedWith(Names.named("Fast")).to(WeaselCopter.class);
```

This works, but because the name lives inside a string, you give up the benefits of compile-time checks and autocompletion. On the whole, I'd rather write my own annotations.

What if you don't want to use an annotation at all? Even adding `@Fast` or `@Named("Fast")` makes your class partially responsible for configuring itself. If that troubles you, read on.

Provider methods

Other uses for provider methods

If you're nervous about the annotations that come with the Guice territory, or if you can't use them (you're creating a third-party class, for instance), provider methods solve your problem elegantly. Because your provider method lives in your module, you can annotate it without worrying that the rest of your source will see the annotations. For example, Listing 16 contains a provider method for `BadgerBoy` (a third-party hero):

```
@Provides @Inject
private BadgerBoy provideBadgerBoy(WeaselCopter copter) {
return new BadgerBoy(copter);
}
```

With this provider method, you can use Guice to configure `BadgerBoy`, and even to select which `Vehicle` he needs, all without changing `BadgerBoy` at all. This means that you don't need to use `@Inject` or binding annotations such as `@Fast` if you don't want to. You can use provider methods to select your dependencies instead.

Which technique you choose depends on your personal preference, whether you're using third-party classes, and how tightly you want to tie a class to its dependencies. Provider methods allow the totally hands-off approach. Annotated dependencies allow configuration information to live a little closer to the class, which makes the source more comprehensible while preserving flexibility and testability. Your application will likely use both ways to specify dependencies as the circumstance warrants.

You're sick of sending Frog Man on every adventure. You'd like a random hero for each new escapade. However, Guice's default binder API doesn't allow a call like "bind the `Hero` class to a different implementation each call." However, you *can* tell Guice to use a special method to create every new `Hero`. Listing 15 shows a new method added to the `HeroModule`, annotated with the special `@Provides` annotation:

Listing 15. Using a provider to write custom creation logic

```
@Provides
private Hero provideHero(FrogMan frogMan, WeaselGirl weaselGirl) {
  if (Math.random() > .5) {
    return frogMan;
  }
  return weaselGirl;
}
```

Guice automatically discovers all methods in your `Module`s that have the `@Provides` annotation. Based on the return type of `Hero`, it works out that when you ask for a hero, it should call this method to provide it. You can stuff provider methods with logic to construct your object, pick it randomly, look it up from a cache, or obtain it by other means. Provider methods are an excellent way to integrate other libraries into your Guice module. They're also new as of Guice 2.0. (The Guice 1.0 way was to write custom provider classes, which were clunkier and more verbose. If you've decided to use Guice 1.0, the user's guide has documentation on the old way, and the sample code supplied with this article has a custom provider you can look at.)

Guice automatically injects the provider method in Listing 15 with the correct arguments. This means Guice will find `WeaselGirl` and `FrogMan` from its list of bindings without you needing to construct them manually within the provider method. This illustrates the "it's turtles all the way down" principle. You rely on Guice to provide your dependencies, even when you're configuring your Guice module itself.

Asking for a `Provider` instead of a dependency

Suppose you want multiple heroes in one story — a `Saga`. If you ask Guice to inject a `Hero`, you'll get only one. But if you ask for a "provider of heroes," you can create as many as you like, as in Listing 17:

Listing 17. Inject a provider to take control of instantiation

```
public class Saga {
  private final Provider<Hero> heroProvider;

  @Inject
  public Saga(Provider<Hero> heroProvider) {
    this.heroProvider = heroProvider;
  }

  public void start() throws IOException {
    for (int i = 0; i < 3; i++) {
      Hero hero = heroProvider.get();
      hero.fightCrime();
    }
  }
}
```

Providers also let you delay the retrieval of the heroes until the saga actually starts. This is handy if the hero depends on data that's time- or context-sensitive.

The `Provider` interface has one method: `get<T>`. To access the object provided, you simply call the method. Whether or not you get a new object each time, and how that object is configured, depends on how Guice was configured. (See the next section on Scopes for details about singletons and other long-lived objects.) In this case, Guice uses your `@Provides` method because it's the registered way to construct a new `Hero`. This means the saga should consist of a mix of three random heroes.

Providers shouldn't be confused with provider methods. (In Guice 1.0, they were considerably harder to tell apart.) Although the `Saga` gets its heroes from your custom `@Provides` method, you can ask for a `Provider` of any Guice-instantiated dependency. If you wanted, you could rewrite `FrogMan`'s constructor according to Listing 18:

Listing 18. You can ask for a `Provider` instead of the dependency

```
@Inject
public FrogMan(Provider<Vehicle> vehicleProvider) {
  this.vehicle = vehicleProvider.get();
}
```

(Note that you didn't have to change the module code at all.) This rewrite doesn't serve any purpose; it just illustrates that you can always ask for a `Provider` instead of the dependency directly.

Scopes

By default, Guice creates a new instance of each dependency you ask for. If your objects are lightweight, this policy will serve you well. However, if you have an expensive-to-create dependency, you may want to share an instance among several clients. In Listing 19, `HeroModule` binds the `HeavyWaterRefinery` as a singleton:

Listing 19. `HeavyWaterRefinery` bound as a singleton

```
public class HeroModule implements Module {
  public void configure(Binder binder) {
    //...
    binder.bind(FuelSource.class)
      .to(HeavyWaterRefinery.class).in(Scopes.SINGLETON);
  }
}
```

Are singletons evil?

If you search the Web for "singletons are evil," you'll find plenty of blog posts, articles, and rants about singletons. It turns out that there's a difference between an "application singleton" and a "JVM singleton" (which is the kind commentators rail against). The JVM singleton enforces its "singletonness" by using linguistic tricks and by forcing clients to reference it using a static reference. An application singleton, on the other hand, doesn't enforce this policy. Another layer (for example, Guice) enforces that one instance of the class exists *per application*. Client code doesn't know about this rule, and this keeps the design flexible and tests easy to write.

This means Guice will keep the refinery around, and whenever another instance requires a fuel source, Guice will inject the *same* refinery. This prevents more than one refinery from starting up in the application.

Guice offers you an option when you select scopes. You can configure them using the binder, or you can annotate the dependency directly, as in Listing 20:

Listing 20. Choosing scope with an annotation instead

```
@Singleton
public class HeavyWaterRefinery implements FuelSource {...}
```

Guice provides `Singleton` scope out of the box, but it allows you to define your own scopes if you wish. As an example, the Guice servlet package supplies two additional scopes: `Request` and `Session`, which serve up a unique instance of your class per servlet request and servlet session.

Constant binding and module configuration

The `HeavyWaterRefinery` requires a license key to start. It turns out that Guice can bind constant values as well as new instances. Check out Listing 21:

Listing 21. Binding a constant value in a module

```
public class HeavyWaterRefinery implements FuelSource {
  @Inject
  public HeavyWaterRefinery(@Named("LicenseKey") String key) {...}
}

// in HeroModule:
binder.bind(String.class)
  .annotatedWith(Names.named("LicenseKey")).toInstance("QWERTY");
```

The binding annotation is necessary here because otherwise, Guice couldn't tell different `String`s apart.

Note that I chose to use the `@Named` annotation despite recommending against it earlier. That's because I want to show off the code in Listing 22:

Listing 22. Configuring a module using a properties file

```
//In HeroModule:
private void loadProperties(Binder binder) {
  InputStream stream =
    HeroModule.class.getResourceAsStream("/app.properties");
  Properties appProperties = new Properties();
  try {
    appProperties.load(stream);
    Names.bindProperties(binder, appProperties);
  } catch (IOException e) {
    // This is the preferred way to tell Guice something went wrong
    binder.addError(e);
  }
}

//In the file app.properties:
LicenseKey=QWERTY1234
```

This code uses the Guice `Names.bindProperties` utility function to bind every property in the app.properties file to a constant with the right `@Named` annotation. This is cool in itself, and it also shows how you can make module code arbitrarily complex. If you like, you can load binding information from a database or XML file. Modules are plain Java code, and that gives you *a lot* of flexibility.

What comes next?

To summarize the main concepts of Guice:

- You ask for dependencies with `@Inject`.
- You bind dependencies to implementations in `Module`s.
- You bootstrap using the `Injector`.
- You add flexibility with `@Provides` methods.

There's plenty more to know about Guice, but you should be able to dive in with the topics I've covered in this article. I recommend downloading it, along with the sample code for this article, and giving it a whirl. Or better yet, create your own sample application. Playing with the concepts without worrying about your production code is a lot of fun. If you want to learn more about Guice's advanced features (such as its aspect-oriented programming support), I suggest following some of the links in Resources.

Speaking of production code, one downside to DI is that it can feel viral. Once you inject one class, it leads to injecting the next and the next. This can be good, because DI makes your code better. On the other hand, it can lead to big refactorings of existing code. To keep the work manageable, you can store the Guice `Injector` somewhere and call it directly. You should treat this as a crutch — necessary, but something you want to do without in the long run.

Guice 2.0 should be out shortly. Some of the features that I didn't cover will make it easier to configure your modules and to support larger, more sophisticated configuration schemes. You can follow links in Resources to learn about upcoming features.

I hope you'll consider adding Guice to your toolkit. In my experience, DI is critical for a flexible, testable codebase. Guice makes DI easy and even fun. What could be better than flexible, testable code that's a blast to write?

Downloads

Description	Name	Size
Java files for this article	j-guice.zip	19KB

Resources

Learn

- Guice: The Guice home page is a great starting point, and the Guice User's Guide touches on a number of advanced Guice topics.
- Dependency injection and domain specific languages: Martin Fowler is always a terrific resource. Check out his site's coverage of these topics.
- Inversion of Control History: Read about the history of, and interrelationships among, DI frameworks such as Spring, PicoContainer, and HiveMind.
- Guice Best Practices: Be sure to check out this page in the Guice Wiki.
- "Expert Guice: 50 some odd ways to Guice up your Java": Watch Guice creator Bob Lee explain Guice and the best ways to use it in this videotaped presentation.
- Guice 2.0: Learn about the new features in the next version of Guice.
- Turtles all the way down: Wikipedia explains this expression.
- "Unit testing with mock objects" (Alexander Chaffee and William Pietri, developerWorks, November 2002) and "Mock Roles, not Objects" (Steve Freeman et al., jmock.org, 2004): Learn more about unit testing with mock objects.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming, including a number of other articles on Dependency Injection

Get products and technologies

- Guice: Download the Guice 2 beta release.
- Guice Plugin: Get the Guice plugin for Struts.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Nicholas Lesiecki

Nicholas Lesiecki has been writing, speaking, and obsessing about better software since 2000. When not serving as "cat throne," he slings code for Google's Seattle engineering office.