

Dependency Injection

DESIGN PATTERNS USING SPRING AND GUICE

DHANJI R. PRASANNA



MANNING

Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact


Special Sales Department
Manning Publications Co.
Sound View Court 3B fax: (609) 877-8256
Greenwich, CT 06830 email: orders@manning.com

©2009 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

© Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15% recycled and processed without the use of elemental chlorine.

	Development Editor: Tom Cirtin
Manning Publications Co.	Copyeditor: Linda Recktenwald
 Sound View Court 3B	Typesetter: Gordan Salinovic
Greenwich, CT 06830	Cover designer: Leslie Haines

ISBN 978-1-933988-55-9

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – MAL – 14 13 12 11 10 09

1	<i>Dependency injection: what's all the hype?</i>	1
1.1	Every solution needs a problem	2
	<i>Seeing objects as services</i>	2
1.2	Pre-DI solutions	4
	<i>Construction by hand</i>	5
	<i>The Factory pattern</i>	7
	<i>The Service Locator pattern</i>	12
1.3	Embracing dependency injection	13
	<i>The Hollywood Principle</i>	13
	<i>Inversion of Control vs. dependency injection</i>	15
1.4	Dependency injection in the real world	17
	<i>Java</i>	17
	<i>DI in other languages and libraries</i>	19
1.5	Summary	19
2	<i>Time for injection</i>	21
2.1	Bootstrapping the injector	22
2.2	Constructing objects with dependency injection	23

- 2.3 Metadata and injector configuration 26
 - XML injection in Spring 27* ▪ *From XML to in-code configuration 30*
 - Injection in PicoContainer 31* ▪ *Revisiting Spring and autowiring 34*
- 2.4 Identifying dependencies for injection 36
 - Identifying by string keys 37* ▪ *Limitations of string keys 42*
 - Identifying by type 44* ▪ *Limitations of identifying by type 46*
 - Combinatorial keys: a comprehensive solution 47*
- 2.5 Separating infrastructure and application logic 51
- 2.6 Summary 52

3 *Investigating DI* 54

- 3.1 Injection idioms 55
 - Constructor injection 55* ▪ *Setter injection 56* ▪ *Interface injection 60*
 - Method decoration (or AOP injection) 62*
- 3.2 Choosing an injection idiom 65
 - Constructor vs. setter injection 66* ▪ *The constructor pyramid problem 69*
 - The circular reference problem 71* ▪ *The in-construction problem 75*
 - Constructor injection and object validity 78*
- 3.3 Not all at once: partial injection 81
 - The reinjection problem 81* ▪ *Reinjection with the Provider pattern 82*
 - The contextual injection problem 84* ▪ *Contextual injection with the Assisted Injection pattern 86* ▪ *Flexible partial injection with the Builder pattern 88*
- 3.4 Injecting objects in sealed code 92
 - Injecting with externalized metadata 93* ▪ *Using the Adapter pattern 95*
- 3.5 Summary 96

4 *Building modular applications* 99

- 4.1 Understanding the role of an object 100
- 4.2 Separation of concerns (my pants are too tight!) 101
 - Perils of tight coupling 102* ▪ *Refactoring impacts of tight coupling 105* ▪ *Programming to contract 108* ▪ *Loose coupling with dependency injection 111*
- 4.3 Testing components 112
 - Out-of-container (unit) testing 113* ▪ *I really need my dependencies! 114*
 - More on mocking dependencies 115* ▪ *Integration testing 116*
- 4.4 Different deployment profiles 118
 - Rebinding dependencies 118* ▪ *Mutability with the Adapter pattern 119*
- 4.5 Summary 121

5 *Scope: a fresh breath of state* 123

- 5.1 What is scope? 124
- 5.2 The no scope (or default scope) 125
- 5.3 The singleton scope 128
 - Singletons in practice* 131 ▪ *The singleton anti-pattern* 135
- 5.4 Domain-specific scopes: the web 139
 - HTTP request scope* 141 ▪ *HTTP session scope* 149
- 5.5 Summary 154

6 *More use cases in scoping* 156

- 6.1 Defining a custom scope 157
 - A quick primer on transactions* 157 ▪ *Creating a custom transaction scope* 158 ▪ *A custom scope in Guice* 160 ▪ *A custom scope in Spring* 164
- 6.2 Pitfalls and corner cases in scoping 166
 - Singletons must be thread-safe* 167 ▪ *Perils of scope-widening injection* 169
- 6.3 Leveraging the power of scopes 180
 - Cache scope* 181 ▪ *Grid scope* 181 ▪ *Transparent grid computing with DI* 183
- 6.4 Summary 184

7 *From birth to death: object lifecycle* 186

- 7.1 Significant events in the life of objects 187
 - Object creation* 187 ▪ *Object destruction (or finalization)* 189
- 7.2 One size doesn't fit all (domain-specific lifecycle) 191
 - Contrasting lifecycle scenarios: servlets vs. database connections* 191 ▪ *The Destructor anti-pattern* 196 ▪ *Using Java's Closeable interface* 197
- 7.3 A real-world lifecycle scenario: stateful EJBs 198
- 7.4 Lifecycle and lazy instantiation 201
- 7.5 Customizing lifecycle with postprocessing 202
- 7.6 Customizing lifecycle with multicasting 205
- 7.7 Summary 207

8 *Managing an object's behavior* 210

- 8.1 Intercepting methods and AOP 211
 - A tracing interceptor with Guice* 212 ▪ *A tracing interceptor with Spring* 214
 - How proxying works* 216 ▪ *Too much advice can be dangerous!* 219

- 8.2 Enterprise use cases for interception 221
 - Transactional methods with warp-persist* 222 ▪ *Securing methods with Spring Security* 224
- 8.3 Pitfalls and assumptions about interception and proxying 228
 - Sameness tests are unreliable* 228 ▪ *Static methods cannot be intercepted* 230 ▪ *Neither can private methods* 231 ▪ *And certainly not final methods!* 233 ▪ *Fields are off limits* 234 ▪ *Unit tests and interception* 236
- 8.4 Summary 238

9 *Best practices in code design* 240

- 9.1 Objects and visibility 241
 - Safe publication* 244 ▪ *Safe wiring* 245
- 9.2 Objects and design 247
 - On data and services* 247 ▪ *On better encapsulation* 252
- 9.3 Objects and concurrency 257
 - More on mutability* 258 ▪ *Synchronization vs. concurrency* 261
- 9.4 Summary 264

10 *Integrating with third-party frameworks* 266

- 10.1 Fragmentation of DI solutions 267
- 10.2 Lessons for framework designers 270
 - Rigid configuration anti-patterns* 271 ▪ *Black box anti-patterns* 276
- 10.3 Programmatic configuration to the rescue 280
 - Case study: JSR-303* 280
- 10.4 Summary 286

11 *Dependency injection in action!* 289

- 11.1 Crosstalk: a Twitter clone! 290
 - Crosstalk's requirements* 290
- 11.2 Setting up the application 290
- 11.3 Configuring Google Sitebricks 294
- 11.4 Crosstalk's modularity and service coupling 295
- 11.5 The presentation layer 296
 - The HomePage template* 298 ▪ *The Tweet domain object* 301
 - Users and sessions* 302 ▪ *Logging in and out* 304

11.6	The persistence layer	308
	<i>Configuring the persistence layer</i>	<i>310</i>
11.7	The security layer	311
11.8	Tying up to the web lifecycle	312
11.9	Finally: up and running!	313
11.10	Summary	314
<i>appendix A</i>	<i>The Butterfly Container</i>	<i>315</i>
<i>appendix B</i>	<i>SmartyPants for Adobe Flex</i>	<i>320</i>
	<i>index</i>	<i>323</i>

1

Dependency injection: what's all the hype?

This chapter covers:

- Seeing an object as a service
- Learning about building and assembling services
- Taking a tour of pre-existing solutions
- Investigating the Hollywood Principle
- Surveying available frameworks

“We all agree that your theory is crazy, but is it crazy enough?”

—Niels Bohr

So you're an expert on dependency injection (DI); you know it and use it every day. It's like your morning commute—you sleepwalk through it, making all the right left turns (and the occasional wrong right turns before quickly correcting) until you're comfortably sitting behind your desk at work. Or you've heard of DI and Inversion of Control (IoC) and read the occasional article, in which case this is your first commute to work on a new job and you're waiting at the station, with a strong suspicion you are about to get on the wrong train and an even stronger suspicion you're on the wrong platform.

Or you're somewhere in between; you're feeling your way through the idea, not yet fully convinced about DI, planning out that morning commute and looking for the best route to work, *MapQuesting* it. Or you have your own home-brew setup that works just fine, thank you very much. You've no need of a DI technology: You bike to work, get a lot of exercise on the way, and are carbon efficient.

Stop! Take a good, long breath. Dependency injection is the art of making work come home to you.

1.1 **Every solution needs a problem**

Most software today is written to automate some real-world process, whether it be writing a letter, purchasing the new album from your favorite band, or placing an order to sell some stock. In object-oriented programming (OOP), these are *objects* and their interactions are *methods*.

Objects represent their real-world counterparts. An Airplane represents a 747 and a Car represents a Toyota; a PurchaseOrder represents you buying this book; and so on.

Of particular interest is the interaction between objects: An airplane flies, while a car can be driven and a book can be opened and read. This is where the value of the automation is realized and where it is valuable in simplifying our lives.

Take the familiar activity of writing an email; you compose the message using an email application (like Mozilla Thunderbird or Gmail) and you send it across the internet to one or more recipients, as in figure 1.1. This entire activity can be modeled as the interaction of various objects.



Figure 1.1 Email is composed locally, delivered across an internet relay, and received by an inbox.

This highlights an important precept in this book: the idea of an *object acting as a service*. In the example, email acts as a message composition service, internet relays are delivery agents, and my correspondent's inbox is a receiving service.

1.1.1 **Seeing objects as services**

The process of emailing a correspondent can be reduced to the composing, delivering, and receiving of email by each responsible object, namely the Emitter, Internet-Relay, and RecipientInbox. Each object is a *client* of the next.

Emitter uses the InternetRelay as a service to *send* email, and in turn, the InternetRelay uses the RecipientInbox as a service to *deliver* sent mail.

The act of composing an email can be reduced to more granular tasks:

- Writing the message
- Checking spelling
- Looking up a recipient's address

And so on. Each is a fairly specialized task and is modeled as a specific service. For example, “writing the message” falls into the domain of editing text, so choosing a `TextEditor` is appropriate. Modeling the `TextEditor` in this fashion has many advantages over extending `Emailer` to write text messages itself: We know exactly where to look if we want to find out what the logic looks like for editing text.

- Our `Emailer` is not cluttered with distracting code meant for text manipulation.
- We can reuse the `TextEditor` component in other scenarios (say, a calendar or note-taking application) without much additional coding.
- If someone else has written a general-purpose text-editing component, we can make use of it rather than writing one from scratch.

Similarly, “checking spelling” is done by a `SpellChecker`. If we wanted to check spelling in a different language, it would not be difficult to swap out the English `SpellChecker` in favor of a French one. `Emailer` itself would not need to worry about checking spelling—French, English, or otherwise.

So now we’ve seen the value of decomposing our services into objects. This principle is important because it highlights the relationship between one object and others it uses to perform a service: An object depends on its services to perform a function.

In our example, the `Emailer` depends on a `SpellChecker`, a `TextEditor`, and an `AddressBook`. This relationship is called a *dependency*. In other words, `Emailer` is a client of its dependencies.

Composition also applies transitively; an object may depend on other objects that themselves have dependencies, and so on. In our case, `SpellChecker` may depend on a `Parser` to recognize words and a `Dictionary` of valid words. `Parser` and `Dictionary` may themselves depend on other objects.

This composite system of dependencies is commonly called an *object graph*. This object graph, though composed of many dependencies, is functionally a single *unit*.

Let’s sum up what we have so far:

- *Service*—An object that performs a *well-defined* function when called upon
- *Client*—Any consumer of a service; an object that calls upon a service to perform a *well-understood* function

The service–client relationship implies a clear *contract* between the objects in the role of performing a specific function that is formally understood by both entities. You will also hear them referred to as:

- *Dependency*—A specific service that is *required* by another object to fulfill its function.
- *Dependent*—A client object that needs a dependency (or dependencies) in order to perform its function.

Not only can you describe an object graph as a system of discrete services and clients, but you also begin to see that a client cannot function *without* its services. In other words, an object cannot function properly without its dependencies.

NOTE DI as a subject is primarily concerned with reliably and efficiently building such object graphs and the strategies, patterns, and best practices therein.

Let's look at ways of building object graphs before we take on DI.

1.2 *Pre-DI solutions*

Figure 1.2 shows a simple relationship between an object and its dependency.

Were you asked to code such a class without any other restrictions, you might attempt something like this:

```
public class Emailer {
    private SpellChecker spellChecker;
    public Emailer() {
        this.spellChecker = new SpellChecker();
    }

    public void send(String text) { .. }
}
```

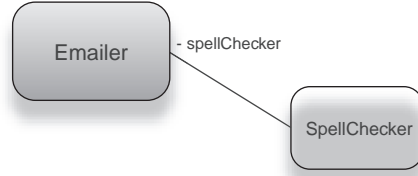


Figure 1.2 Object `Emailer` is composed of another object, `SpellChecker`.

Then constructing a working `Emailer` (one with a `SpellChecker`) is as simple as constructing an `Emailer` itself:

```
Emailer emailer = new Emailer();
```

No doubt you have written code like this at some point. I certainly have. Now, let's say you want to write a unit test for the `send()` method to ensure that it is checking spelling before sending any message. How would you do it? You might create a *mock* `SpellChecker` and give that to the `Emailer`. Something like:

```
public class MockSpellChecker extends SpellChecker {
    private boolean didCheckSpelling = false;
    public boolean checkSpelling(String text) {
        didCheckSpelling = true;
        return true;
    }

    public boolean verifyDidCheckSpelling()
        { return didCheckSpelling; }
}
```

Called by test to
verify behavior

Of course, we can't use this mock because we are unable to substitute the internal `spellchecker` that an `Emailer` has. This effectively makes your class *untestable*, which is a showstopper for this approach.

This approach also prevents us from creating objects of the same class with different behaviors. See listing 1.1 and figure 1.3.

Listing 1.1 An email service that checks spelling in English

```
public class Emailer {
    private SpellChecker spellChecker;
```

```

public Emailer() {
    this.spellChecker = new EnglishSpellChecker();
}
...
}

```

In this example, the `Emailer` has an `EnglishSpellChecker`. Can we create an `Emailer` with a `FrenchSpellChecker`? We cannot! `Emailer` encapsulates the *creation* of its dependencies.

What we need is a more flexible solution: *construction by hand* (sometimes called manual dependency injection), where instead of a dependent creating its own dependencies, it has them provided externally.

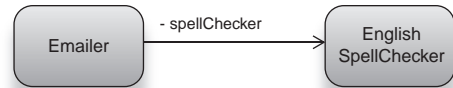


Figure 1.3 An email service that checks spelling in English

1.2.1 Construction by hand

Naturally, the solution is not to encapsulate the creation of dependencies, but what does this mean and to whom do we offload this burden? Several techniques can solve this problem. In the earlier section, we used the object's constructor to create its dependencies. With a slight modification, we can keep the structure of the object graph but offload the burden of creating dependencies. Here is such a modification (see figure 1.4):

```

public class Emailer {
    private SpellChecker spellChecker;
    public void setSpellChecker(SpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }
    ...
}

```

Notice that I've replaced the constructor that created its own `SpellChecker` with a method that *accepts* a `SpellChecker`. Now we can construct an `Emailer` and substitute a mock `SpellChecker`:

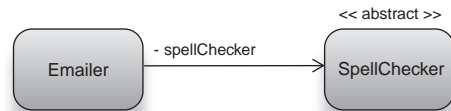


Figure 1.4 An email service that checks spelling using an abstract spelling service

```

@Test
public void ensureEmailerChecksSpelling() {
    MockSpellChecker mock = new MockSpellChecker();
    Emailer emailer = new Emailer();
    emailer.setSpellChecker(mock);
    emailer.send("Hello there!");
    assert mock.verifyDidCheckSpelling();
}

```

Pass in mock dependency for testing

Verify the dependency was used properly

Similarly, it is easy to construct `Emailers` with various behaviors. Here's one for French spelling:

```

Emailer service = new Emailer();
service.setSpellChecker(new FrenchSpellChecker());

```

And one for Japanese:

```
Emailer service = new Emailer();
service.setSpellChecker(new JapaneseSpellChecker());
```

Cool! At the time of creating the `Emailer`, it's up to you to provide its dependencies. This allows you to choose particular flavors of its services that suit your needs (French, Japanese, and so on) as shown in figure 1.5.

Since you end up connecting the pipes yourself at the time of construction, this technique is referred to as construction by hand. In the previous example we used a setter method (a method that accepts a value and sets it as a dependency). You can also pass in the dependency via a constructor, as per the following example:

```
public class Emailer {
    private SpellChecker spellChecker;
    public Emailer(SpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }
    ...
}
```

Then creating the `Emailer` is even more concise:

```
Emailer service = new Emailer(new JapaneseSpellChecker());
```

This technique is called *constructor injection* and has the advantage of being explicit about its contract—you can never create an `Emailer` and forget to set its dependencies, as is possible in the earlier example if you forget to call the `setSpellChecker()` method. This concept is obviously very useful in OOP. We'll study it in greater detail in the coming chapters.

NOTE The idea of connecting the pipes together, or giving a client its dependency, is sometimes also referred to as “injecting” objects into one another and other times as “wiring” the objects together.

While construction by hand definitely helps with testing, it has some problems, the most obvious one being the burden of knowing how to construct object graphs being placed on the client of a service. If I use the same object in many places, I must repeat code for wiring objects in all of those places. Construction by hand, as the name suggests, is really tedious! If you alter the dependency graph or any of its parts, you may be forced to go through and alter all of its clients as well. Fixing even a small bug can mean changing vast amounts of code.

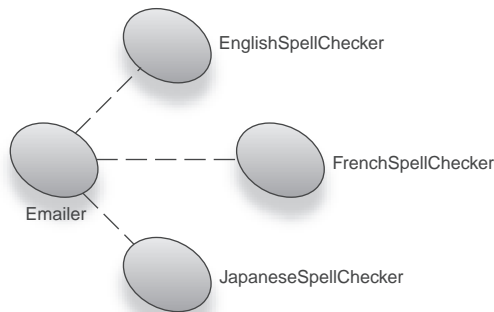


Figure 1.5 The same `Emailer` class can now check spelling in a variety of languages.

Another grave problem is the fact that users need to know how object graphs are wired internally. This violates the principle of encapsulation and becomes problematic when dealing with code that is used by many clients, who really shouldn't have to care about the internals of their dependencies in order to use them. There are also potent arguments against construction by hand that we will encounter in other forms in the coming chapters. So how can we offload the burden of dependency creation and not shoot ourselves in the foot doing so? One answer is the *Factory pattern*.

1.2.2 The Factory pattern

Another time-honored method of constructing object graphs is the Factory design pattern (also known as the Abstract Factory¹ pattern). The idea behind the Factory pattern is to offload the burden of creating dependencies to a third-party object called a factory (shown in figure 1.6). Just as an automotive factory creates and assembles cars, so too does a Factory pattern create and assemble object graphs.

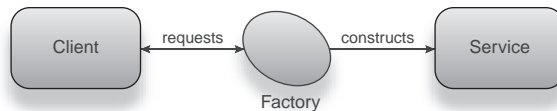


Figure 1.6 A client requests its dependencies from a Factory.

Let's apply the Factory pattern to our Emailer. The Emailer's code remains the same (as shown in listing 1.2).

Listing 1.2 An email service whose spellchecker is set via constructor

```

public class Emailer {
    private SpellChecker spellChecker;
    public Emailer(SpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }
    ...
}
  
```

Instead of constructing the object graph by hand, we do it inside another class called a factory (listing 1.3).

Listing 1.3 A "French" email service Factory pattern

```

public class EmailerFactory {
    public Emailer newFrenchEmailer() {
        return new Emailer(new FrenchSpellChecker());
    }
}
  
```

¹ Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional Computing Series, 1994). Sometimes called the "Gang of Four" book.

Notice that the Factory pattern is very explicit about what kind of `Emailer` it is going to produce; `newFrenchEmailer()` creates one with a French spellchecker. Any code that uses French email services is now fairly straightforward:

```
Emailer service = new EmailerFactory().newFrenchEmailer();
```

The most important thing to notice here is that the client code has no reference to spellchecking, address books, or any of the other *internals* of `Emailer`. By adding a level of abstraction (the Factory pattern), we have separated the code *using* the `Emailer` from the code that *creates* the `Emailer`. This leaves client code clean and concise.

The value of this becomes more apparent as we deal with richer and more complex object graphs. Listing 1.4 shows a Factory that constructs our `Emailer` with many more dependencies (also see figure 1.4).

Listing 1.4 A Factory that constructs a Japanese emailer

```
public class EmailerFactory {
    public Emailer newJapaneseEmailer() {
        Emailer service = new Emailer();
        service.setSpellChecker(new JapaneseSpellChecker());
        service.setAddressBook(new EmailBook());
        service.setTextEditor(new SimpleJapaneseTextEditor());
        return service;
    }
}
```

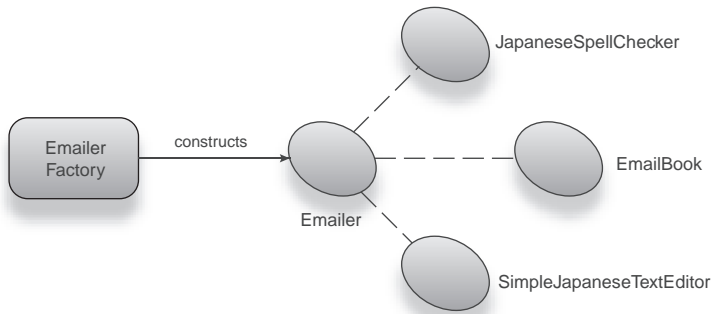


Figure 1.7 `EmailerFactory` constructs and assembles a Japanese `Emailer` with various dependencies.

Code that uses such an `Emailer` is as simple and readable as we could wish:

```
Emailer emailer = new EmailerFactory().newJapaneseEmailer();
```

The beauty of this approach is that client code only needs to know which Factory to use to obtain a dependency (and none of its internals).

Now how about testing this code? Are we able to mock `Emailer`'s dependencies? Sure; our test can simply ignore the Factory and pass in mocks:

```
@Test
public void testEmailer() {
    MockSpellChecker spellChecker =
```

```

        new MockSpellChecker();
    ...

    Emailer emailer = new Emailer();
    emailer.setSpellChecker(spellChecker);
    ...

    emailer.send("Hello there!");

    //verify emailer's behavior
    assert ...;
}

```

Create mocks for each dependency
Set mocked dependencies on emailer
Ensure everything worked

So far, so good. Now let's look at a slightly different angle: How can we test *clients* of *Emailer*? Listing 1.5 shows one way.

Listing 1.5 A client uses *Emailer* to send messages typed by a user

```

public class EmailClient {
    private Emailer emailer =
        new EmailerFactory().newEnglishEmailer();

    public void run() {
        emailer.send(someMessage());
        confirm("Sent!");
    }
}

```

Get ourselves an Emailer from its Factory
Send message and confirm to user

This client does not know anything about *Emailer*'s internals; instead, it depends on a *Factory*. If we want to test that it correctly calls *Emailer.send()*, we need to use a mock. Rather than set the dependency directly, we must pass in the mock via the *Factory*, as in listing 1.6.

Listing 1.6 Test sets up a mock instance for *EmailClient* using *Emailer*'s *Factory*

```

@Test
public void testEmailClient() {
    MockEmailer mock = new MockEmailer();
    EmailerFactory.set(mock);

    new EmailClient().run();

    assert mock.correctlySent();
}

```

Set mock instance on factory using a static method
Verify that the mock was used correctly by EmailClient

In this test, we pass in *MockEmailer* using the static method *EmailerFactory.set()*, which stores and uses the provided object rather than creating new ones (listing 1.7).

Listing 1.7 A client uses *Emailer* to send messages typed by a user

```

public class EmailerFactory {
    private static Emailer instance;

    public Emailer newEmailer() {
        if (null == instance)
            return new Emailer(..);
    }
}

```

Static holder stores mock instance for later use
If no mock is present, create a new Emailer with dependencies


```

        return instance;
    }

    static void set(Emailer mock) {
        instance = mock;
    }

```

Save mock instance
to static holder

In listing 1.7, `EmailerFactory` has been heavily modified to support testing. A test can first set up a mock instance via the static `set()` method, then verify the behavior of any clients (as shown in listing 1.6).

Unfortunately, this is not the complete picture, since forgetting to clean up the mock can interfere with other `Emailer`-related tests that run later. So, we must *reset* the Factory at the end of every test:

```

@Test
public void testEmailClient() {
    MockEmailer mock = new MockEmailer();
    EmailerFactory.set(mock);

    new EmailClient().run();

    assert mock.correctlySent();
    EmailerFactory.set(null);
}

```

Reset factory's
state to null

We're not quite out of the woods yet. If an exception is thrown before the Factory is reset, it can still leave things in an erroneous state. So, a safer cleanup is required:

```

@Test
public void testEmailClient() {
    MockEmailer mock = new MockEmailer();
    EmailerFactory.set(mock);
    try {
        new EmailClient().run();

        assert mock.correctlySent();
    } finally {
        EmailerFactory.set(null);
    }
}

```

Reset factory's state
inside a finally block

Much better. A lot of work to write a simple assertion, but worth it! Or is it? Even this careful approach is insufficient in broader cases where you may want to run tests in parallel. The static mock instance inside `EmailerFactory` can cause these tests to clobber each other from concurrent threads, rendering them useless.

NOTE This is an essential problem with shared state, often portended by the *Singleton pattern*. Its effects and solutions are examined more closely in chapter 5.

While the Factory pattern solves many of the problems with construction by hand, it obviously still leaves us with significant hurdles to overcome. Apart from the testability problem, the fact that a Factory must accompany every service is troubling. Not only this, a Factory must accompany *every* variation of *every* service. This is a sizable amount

of distracting clutter and adds a lot of peripheral code to be tested and maintained. Look at the example in listing 1.8, and you'll see what I mean.

Listing 1.8 Factories that create either French or Japanese email services

```
public class EmailerFactory {
    public Emailer newJapaneseEmailer() {
        Emailer service = new Emailer();
        service.setSpellChecker(new JapaneseSpellChecker());
        service.setAddressBook(new EmailBook());
        service.setTextEditor(new SimpleJapaneseTextEditor());
        return service;
    }
    public Emailer newFrenchEmailer() {
        Emailer service = new Emailer();
        service.setSpellChecker(new FrenchSpellChecker());
        service.setAddressBook(new EmailBook());
        service.setTextEditor(new SimpleFrenchTextEditor());
        return service;
    }
}
```

Specific dependencies of a Japanese email service

Specific dependencies of a French email service

If you wanted an English version of the `Emailer`, you would have to add yet another method to the Factory. And consider what happens when we replace `EmailBook` with a `PhoneAndEmailBook`. You are forced to make the following changes:

```
public class EmailerFactory {
    public Emailer newJapaneseEmailer() { ...
        service.setAddressBook(new PhoneAndEmailBook());
        ...
    }
    public Emailer newFrenchEmailer() { ...
        service.setAddressBook(new PhoneAndEmailBook());
        ...
    }
    public Emailer newEnglishEmailer() { ...
        service.setAddressBook(new PhoneAndEmailBook());
        ...
    }
}
```

All three of the changes are identical! It is clearly not a desirable scenario. Furthermore, any client code is at the mercy of available factories: You must create new factories for each additional object graph variation. All this spells reams of additional code to write, test, and maintain.

In the case of `Emailer`, following the idea to its inevitable extreme yields:

```
Emailer service = new EmailerFactoryFactory()
    .newAdvancedEmailerFactory()
    .newJapaneseEmailerWithPhoneAndEmail();
```

"General" factory produces simple or advanced factories

Factory for advanced Emailers

Factory producing specific Emailer configurations

It is very difficult to test code like this. I have seen it ruin several projects. Clearly, the Factory pattern technique's drawbacks are serious, especially in larger and more complex code. What we need is a broader mitigation of the core problem.

1.2.3 The Service Locator pattern

A *Service Locator* pattern is a kind of Factory. It is a third-party object responsible for producing a fully constructed object graph.

In the typical case, Service Locators are used to find services published by external sources; the service may be an API offered by a bank to transfer funds or a search interface from Google.

These external sources may reside in the same application, machine, or local area network, or they may not. Consider an interface to the NASA Mars Rover several millions of miles away, a simple library for text processing, bundled within an application, or a Remote Enterprise Java Bean (EJB).² Look at figure 1.8 for a visual.

Let's look at what a service locator does:

```
Mailer emailer = (Mailer) new ServiceLocator().get("Mailer");
```

Notice that we pass the Service Locator a *key*, in this case the word “Mailer.” This tells our locator that we want an `Mailer`. This is significantly different from a Factory that produces only one kind of service. A Service Locator is, therefore, a Factory that can produce *any* kind of service.

Right away this helps reduce a huge amount of repetitive Factory code, in favor of the single Service Locator.

JNDI: a Service Locator

The Java Naming and Directory Interface (JNDI) is a good example of a Service Locator pattern. It is often used by application servers to register resources at start time and later by deployed applications to look them up. Web applications typically use JNDI to look up data sources in this manner.

Let's apply the Service Locator pattern to the earlier example:

```
Mailer emailer = (Mailer) new ServiceLocator()
    .get("JapaneseEmailerWithPhoneAndEmail");
```

Key precisely describes
the desired service

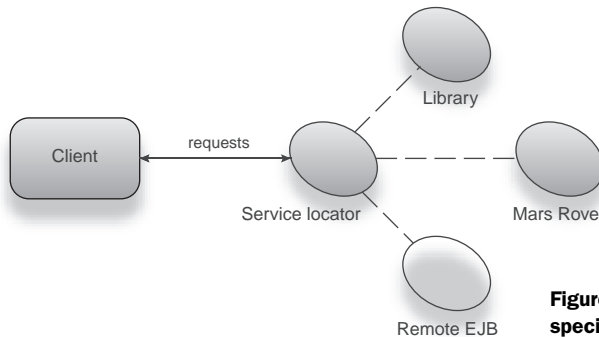


Figure 1.8 Service locators find specific services requested by clients.

² A Remote EJB is a type of service exposed by Java EE application servers across a network. For example, a brokerage may expose an order-placing service as a Remote EJB to various client banks.

This code is simple and readable. The identity of a service (its key) is sufficient to obtain exactly the right service and configuration. Now altering the behavior of a service identified by a particular key or fixing bugs within it by changing its object graph will have no effect on dependent code and can be done transparently.

Unfortunately, being a kind of Factory, Service Locators suffer from the same problems of testability and shared state. The keys used to identify a service are opaque and can be confusing to work with, as anyone who has used JNDI can attest. If a key is bound improperly, the wrong type of object may be created, and this error is found out only at runtime. The practice of embedding information about the service within its identifier (namely, "JapaneseEmailerWithPhoneAndEmail") is also verbose and places too much emphasis on arbitrary conventions.

With DI, we take a completely different approach—one that emphasizes testability and concise code that is easy to read and maintain. That's only the beginning; as you will see soon, with DI we can do a great deal more.

1.3 Embracing dependency injection

With dependency injection, we take the best parts of the aforesaid pre-DI solutions and leave behind their drawbacks.

DI enables testability in the same way as construction by hand, via a setter method or constructor injection. DI removes the need for clients to know about their dependencies and how to create them, just as factories do. It leaves behind the problems of shared state and repetitive clutter, by moving construction responsibility to a library.

With DI, clients need know nothing about French or English Emailers, let alone French or English SpellCheckers and TextEditors, in order to use them. This idea of not *explicitly* knowing is central to DI. More accurately, not *asking* for dependencies and instead having them provided to you is an idea called the *Hollywood Principle*.

1.3.1 The Hollywood Principle

The Hollywood Principle is "Don't call us; we'll call you!" Just as Hollywood talent agents use this principle to arrange auditions for actors, so do DI libraries use this principle to provide objects with what they depend on.

This is similar to what we saw in construction by hand (sometimes referred to as *manual* dependency injection). There is one important difference: The task of creating, assembling, and wiring the dependencies into an object graph is performed by an external framework (or library) known as a *dependency injection framework*, or simply a *dependency injector*. Figure 1.9 illustrates this arrangement.

Control over the construction, wiring, and assembly of an object graph no longer resides with the clients or services

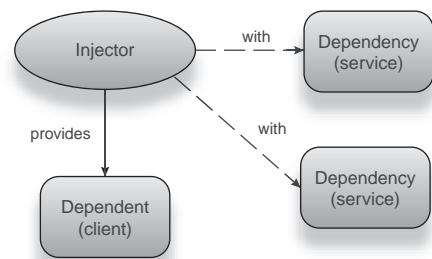


Figure 1.9 The injector provides an object with its dependencies.

themselves. The Hollywood Principle's reversal of responsibilities is sometimes also known as IoC.

NOTE DI frameworks are sometimes referred to as IoC containers. Examples of such frameworks are PicoContainer (for Java), StructureMap (for C#), and Spring (for Java and C#).

Listing 1.9 shows the Hollywood Principle in action.

Listing 1.9 An Emailer and a client

```
public class Emailer {
    ...
    public void send(String text) { .. }
}

public class SimpleEmailClient {
    private Emailer emailer;
    public SimpleEmailClient(Emailer emailer) {
        this.emailer = emailer;
    }
    public void sendEmail() {
        emailer.send(readMessage());
    }
}
```

① SimpleEmailClient depends on Emailer

② Sends an email read from input

In this example, our dependent is SimpleEmailClient ① and the service it uses ② is Emailer. Notice that neither class is aware of how to construct its graph, nor does either explicitly ask for a service.

In order to send mail, SimpleEmailClient does not need to expose anything about Emailer or how it works. Put another way, SimpleEmailClient encapsulates Emailer, and sending email is completely opaque to the user. Constructing and connecting dependencies is now performed by a dependency injector (see figure 1.10).

The dependency is shown as a class diagram in figure 1.11.

Notice that SimpleEmailClient knows nothing about what kind of Emailer it needs or is using to send a message. All it knows is that it accepts *some kind* of Emailer, and this dependency is used when needed. Also notice that the client code is now starting to resemble service code; both are free of logic to create or

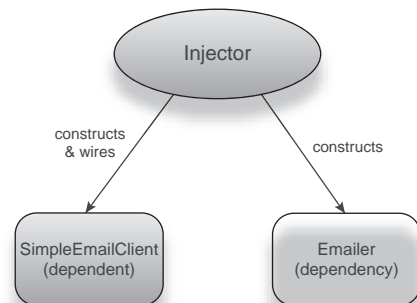
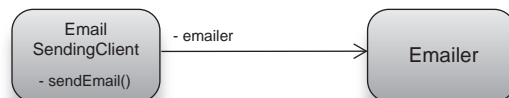


Figure 1.10 The injector constructs and wires SimpleEmailClient with a dependency (Emailer).

Figure 1.11 Client code that uses an email service provided by dependency injection



locate dependencies. DI facilitates this streamlining, stripping code of distracting clutter and infrastructure logic, leaving purposed, *elementary logic* behind.

We have not yet seen how the wiring is done, as this differs from injector to injector, but what we've seen is still very instructive because it highlights the separation of *infrastructure* code (meant for wiring and construction) from *application* code (the core purpose of a service). The next couple of sections explore this idea, before we jump into the specifics of working with dependency injectors.

1.3.2 Inversion of Control vs. dependency injection

Worthy as they are of a heavyweight bout, these two terms are not really opposed to one another as the heading suggests. You will come across the term IoC quite often, both in the context of dependency injection and outside it. The phrase Inversion of Control is rather vague and connotes a general reversal of responsibilities, which is nonspecific and could equally mean any of the following:

- A module inside a Java EE application server
- An object wired by a dependency injector
- A test method automatically invoked by a framework
- An event handler called on clicking a mouse button

Pedantic users of the term suggest that all of these cases are consistent with its definition and that DI itself is simply one instance of IoC.

In common use, dependency injectors are frequently referred to as *IoC containers*.

In the interest of clarity, for the rest of this book I will abandon the term IoC (and its evil cousin IoC container) in favor of the following, more precise terms:

- *Hollywood Principle*—The idea that a dependent is *contacted* with its dependencies
- *Dependency injector*—A framework or library that embodies the Hollywood Principle
- *Dependency injection*—The range of concerns with designing applications built on these principles

Dependency injection

As near as I can determine, it wasn't until Martin Fowler wrote "Inversion of Control Containers and the Dependency Injection Pattern"³ that the term dependency injection came into popular use. The term came out of many discussions on the subject among Fowler, the authors of PicoContainer (Paul Hammant and Aslak Hellesøy), Jon Tirsén, and Rod Johnson, among others.

³ The article has been updated many times over the years. You can read the latest version at <http://martinfowler.com/articles/injection.html>.

Early frameworks differed by the forms of wiring that they proffered and promoted. Over the years, the efficacy of setter and constructor wiring overtook that of other forms of wiring, and more flexible solutions that emphasized the safety of contracts and the reduction of *repetitive* code emerged. With the rise of related paradigms such as *aspect-oriented programming* (AOP), these features continued to improve. Applications built with DI became streamlined and tolerant to rapid structural and behavioral change.

The modes of configuring a dependency injector also evolved from verbose sets of contracts and configuration files to more concise forms, using new language constructs such as *annotations* and *generics*. Dependency injectors also took better advantage of class manipulation tools like *reflection* and *proxying* and began to exploit design patterns such as *Decorator*⁴ and *Builder*,⁵ as well as *Domain Specific Languages (DSL)*.

Domain Specific Language (DSL)

Domain Specific Language, or DSL is a language used to solve a specific problem as opposed to a language like C++ that is intended to solve general (or any) problems. DSLs are sometimes used in dependency injection as a precise way of expressing the structural relationship between objects. They are also used to capture other concerns (such as scope) in the *language* of DI. (For more information, see the forthcoming book from Manning Publications, *Building Domain Specific Languages in Boo* by Ayende Rahien.)

A growing emphasis on unit testing continues to be a natural catalyst to the growth of DI popularity. This made for agile programs, which are easily broken down into discrete, modular *units* that are simple to test and swap out with alternative behaviors. Consequently, *loose coupling* is a core driver in the evolution of DI.

Loose coupling

Loose coupling describes an adaptable relationship between a client and a service, where significant changes to the internals of the service have minimal impact on the client. It is generally achieved by placing a rigid contract between client and service so that either may evolve independently, so long as they fulfill the obligations laid down by the contract. Loose coupling is explored in greater detail in chapter 4.

We will explore the best approaches to solving problems with DI and look in detail at bad practices, pitfalls, and corner cases to watch out for, as well as the safety, rigor, and power of DI, properly applied. And most of all we'll see how DI can make your code lean, clean, and something mean.

⁴ Erich Gamma et al, *Design Patterns*: "The Decorator Pattern."

⁵ Ibid., "The Builder Pattern."

1.4 Dependency injection in the real world

We have now looked at several possible solutions and hinted at a cool alternative called DI. We've also taken an evening stroll down History Lane, turning at the corner of Terminology Boulevard (and somehow visited Hollywood on the way!). Before we proceed to the nuts and bolts of dependency injection, let's survey the landscape to learn what libraries are available, how they work and how they originated.

This is by no means a comparison or evaluation of frameworks; it's a brief introduction. Neither is it meant to be a comprehensive list of options. I will only touch on relatively well-known and widely used DI libraries in Java—and only those that are open source and freely available. Not all of them are purely DI-focused, and very few support the full gamut of DI design patterns described in this book. Nevertheless, each is worth a look.

1.4.1 Java

Java is the birthplace of dependency injection and sports the broadest and most mature set of DI libraries among all platforms. Since many of the problems DI evolved to address are fundamental to Java, DI's effectiveness and prevalence are especially clear in this platform. We'll look at five such libraries ranging from the earliest incarnations of DI in Apache Avalon, through mature, widely adopted libraries like Spring and PicoContainer, to the cutting-edge, modern incarnation in Google, Guice.

APACHE AVALON

Apache Avalon is possibly the earliest DI library in the Java world, and it's perhaps the first library that really focused on dependency injection as a core competency. Avalon styled itself as a complete application container, in the days before Java EE and application servers were predominant. Its primary mode of wiring was via custom interfaces, not setter methods or constructors. Avalon also supported myriad lifecycle and event-management interfaces and was popular with many Apache projects. The Apache James mail server (a pure Java SMTP, POP3, and IMAP email server) is built on Avalon.

Avalon is defunct, though Apache James is still going strong. Some of the ideas from Avalon have passed on to another project, Apache Excalibur, whose DI container is named Fortress. Neither Avalon nor Excalibur/Fortress is in common use today.

SPRING FRAMEWORK

Spring Framework is a groundbreaking and cornerstone dependency injection library of the Java world. It is largely responsible for the popularity and evolution of the DI idiom and for a long time was almost synonymous with dependency injection. Spring was created by Rod Johnson and others and was initially meant to solve specific pains in enterprise projects. It was established as an open source project in 2003 and grew rapidly in scope and adoption. Spring provides a vast set of abstractions, modules, and points of integration for enterprise, open source, and commercial libraries. It consists of much more than dependency injection, though DI is its core competency. It primarily supports setter and constructor injection and has a variety of options for managing

objects created by its dependency injector. For example, it provides support for both the AspectJ and AopAlliance AOP paradigms.

Spring adds functionality, features, and abstractions for popular third-party libraries at alarming rates. It is extremely well-documented in terms of published reference books as well as online documentation and continues to enjoy widespread growth and adoption.

PICOCONTAINER AND NANOCONTAINER

PicoContainer was possibly the first DI library to offer constructor wiring. It was envisioned and built around certain philosophical principles, and many of the discussions found on its website are of a theoretical nature. It was built as a lightweight engine for wiring components together. In this sense it is well-suited to extensions and customization. This makes PicoContainer useful under the covers. PicoContainer supports both setter and constructor injection but clearly demonstrates a bias toward the latter. Due to its nuts-and-bolts nature, many people find PicoContainer difficult to use in applications directly. NanoContainer is a sister project of PicoContainer that attempts to bridge this gap by providing a simple configuration layer, with PicoContainer doing the DI work underneath. NanoContainer also provides other useful extensions.

APACHE HIVEMIND

Apache HiveMind was started by Howard Lewis Ship, the creator of Apache Tapestry, a popular *component-oriented* Java web framework. For a long time Tapestry used HiveMind as its internal dependency injector. Much of the development and popularity of HiveMind has stemmed from this link, though HiveMind and Tapestry have parted company in later releases. HiveMind originally began at a consulting project of Howard's that involved the development and interaction of several hundreds of services. It was primarily used in managing and organizing these services into common, reusable patterns.

HiveMind offers both setter and constructor injection and provides unusual and innovative DI features missing in other popular libraries. For example, HiveMind is able to create and manage *pools* of a service for multiple concurrent clients. Apache HiveMind is not as widely used as some other DI libraries discussed here; however, it has a staunch and loyal following.

GOOGLE GUICE

Guice (pronounced "Juice") is a hot, new dependency injector created by Bob Lee and others at Google. Guice unabashedly embraces Java 5 language features and strongly emphasizes type and contract safety. It is lightweight and decries verbosity in favor of concise, type-safe, and rigorous configuration. Guice is used heavily at Google, particularly in the vast AdWords application, and a version of it is at the heart of the Struts2 web framework. It supports both setter and constructor injection along with interesting alternatives. Guice's approach to AOP and construction of object graphs is intuitive and simple. Guice is also available inside the Google Web Toolkit (GWT), via its cousin Gin.⁶

⁶ Google Gin is a port of Guice for the GWT, a JavaScript web framework. Find out more at <http://code.google.com/p/google-gin>.

It is a new kid on the block, but its popularity is rising and its innovations have already prompted several followers and even some mainstays to emulate its ideas. Guice has a vibrant and thoughtful user community.

1.4.2 DI in other languages and libraries

Several other DI libraries are also available. You could write a whole book on the subject! Some provide extra features, orthogonal to dependency injection, or are simply focused on a different problem space. JBoss Seam is one such framework; it offers complex state management for web applications and integration points for standardized services such as EJBs. It also offers a reasonably sophisticated subset of dependency injection.

As languages, C# and .NET are structurally similar to Java. They are both statically typed, object-oriented languages that need to be compiled. It is no surprise, then, that C# is found in the same problem space as Java and that dependency injectors are applied to equal effect in applications written in C#. However, the prevalence of DI is much lower in the C# world in general. While C# has ports of some Java libraries like Spring and PicoContainer, it also has some innovative DI of its own, particularly in Castle MicroKernel, which does a lot more than just DI. StructureMap, on the other hand, is a mainstay and a more traditional DI library.

Some platforms (or languages) make it harder to design dependency injectors because they lack features such as garbage collection and reflection. C++ is a prime example of such a language. However, it is still possible to write dependency injectors in these languages with other methods that substitute for these tools. For instance, some C++ libraries use *precompilation* and *source code generation* to enable DI.

Still other languages place different (and sometimes fewer) restrictions on types and expressions, allowing objects to take on a more dynamic role in constructing and using services. Python and Ruby are good examples of such *duck-typed* languages. Neither Python nor Ruby programs need to be compiled; their source code is directly *interpreted*, and type checking is done on the fly. Copland is a DI library for Ruby that was inspired by (and is analogous to) Apache HiveMind for Java. Copland uses many of the same terms and principles as HiveMind but is more flexible in certain regards because of the dynamic nature of Ruby.

1.5 Summary

This chapter was an exposition to the subject of building programs in units and has laid the groundwork for DI. Most software is about automating some process in the real world, such as sending a friend a message by email or processing a purchase order. Components in these processes are modeled as a system of objects and methods. When we have vast swaths of components and interactions to manage, constructing and connecting them becomes a tedious and complex task. Development time spent on maintenance, refactoring, and testing can explode without carefully designed architectures. We can reduce all of this to the rather simple-sounding problem of finding and

constructing the dependencies, the requisite services of a component (client). However, as you've seen, this is nontrivial when dealing with objects that require different behaviors and does not often scale with typical solutions.

Prior to the use of dependency injectors, there were several solutions of varying effectiveness:

- *Construction by hand*—This involves the client creating and wiring together all of its dependencies. This is a workable solution and certainly conducive to testing, but it scales very poorly as we've merely offloaded the burden of creating and wiring dependencies to clients. It also means clients must know about their dependencies and about *their* implementation details.
- *The Factory pattern*—Clients request a service from an intermediary component known as a Factory. This offloads the burden to Factories and leaves clients relatively lean. Factories have been very successful over the years; however, they pose very real problems for testing and can introduce more with shared state. In very large projects they can cause an explosion of infrastructure code. Code using factories is difficult to test and can be a problem to maintain.
- *The Service Locator pattern*—Essentially a type of Factory, it uses keys to identify services and their specific configurations. The Service Locator solves the explosion-of-factories problem by hiding the details of the service and its configuration from client code. Since it is a Factory, it is also prone to testability and shared state problems. Finally, it requires that a client explicitly request each of its dependencies by an arbitrary key, which can be unclear and abstruse.

Dependency injection offers an innovative alternative via the Hollywood Principle: "Don't call us; we'll call you!" meaning that a component need know nothing about its dependencies nor explicitly ask for them. A third-party library or framework known as the dependency injector is responsible for constructing, assembling, and wiring together clients with services.

DI solves the problem of object graph construction and assembly by applying the Hollywood Principle across components. However, this is just a small window into what is possible when an application is designed and built around DI. We will explore managing state and modifying the behavior and lifecycle of objects in broader contexts. We'll look at the nuances, pitfalls, and corner cases of various configurations and the subtle consequences of particular design choices. This book will equip you with the knowledge and careful understanding you need to design robust, testable, and easily maintainable applications for any purpose or platform. In the next chapter, you'll get a chance to get your hands dirty. We'll dive right in to using some of the DI libraries surveyed in this chapter and contrast their various approaches. Keep a bar of soap handy.

Time for injection

This chapter covers:

- Using dependency injection
- Identifying dependencies by keys
- Comparing string, type, and combinatorial keys
- Separating infrastructure from application logic

“We shape our buildings, thereafter they shape us.”

—Winston Churchill

In chapter 1, we saw that dependency injection offers a unique and concise solution to the problem of constructing object graphs, with a strong emphasis on unit testing. The crux of this solution is the fact that all your code is freed from constructing dependencies. Code written with DI in mind is behaviorally focused and without distracting clutter. Why is this important?

While the answer may seem self-evident (hard to argue with the less and more precise), it is well worth spelling out:

- *Behaviorally focused*—Presumably, you set out to write code toward a specific purpose, and that purpose is not to construct more objects! Whether writing an email system, a game, or an enterprise messaging system, your primary focus is the behavior of the system. When architectural concerns (such as constructing and locating services) impinge on this logic, it unnecessarily distracts from the original purpose.
- *Modular*—Code designed in discrete, modular units is not only easy to maintain, but it is reusable and easy to package. Individual units are structurally simple to work with. Understanding whether code meets specified requirements is often a difficult proposition for developers. Modular code is focused on small functional subsets (for instance, a spellchecker) and thus is easier to develop.
- *Testable*—Since modules are easy to test for general, purposed behavior without specific awareness of the overall system, it follows that they provide for more robust code. While nominal testability is a benefit in itself, there is much more that concise, loosely coupled code purports. The ability to swap out dependencies with mock counterparts is crucial when testing objects that make use of expensive resources like database connections.

In this chapter, we're going to work with dependency injectors, using various libraries. We'll look at how to identify dependencies, distinguish them from one another, and tell the injector what to do with them. This chapter also examines best practices in configuring dependency injectors and in choosing identifiers (keys) for objects. Let's begin with bootstrapping the injector.

2.1 *Bootstrapping the injector*

The dependency injector is itself a service like any other. And like any service, it must be constructed and prepared before it can be used. This bootstrap of the injector occurs before any object in the application can benefit from DI. When and how the injector is bootstrapped is often specific to the kind of application in question. It may be done in the following situations:

- In the *init* lifecycle stage of a web application, upon being deployed (figure 2.1)

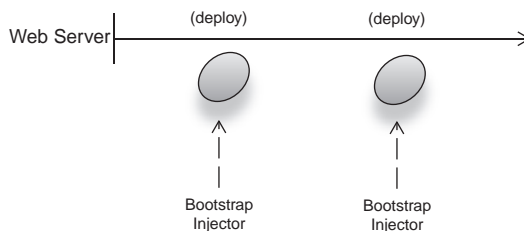


Figure 2.1 Injector is bootstrapped in the *init* lifecycle stage of a web application, on deployment.

- On *startup* of a desktop program (figure 2.2)

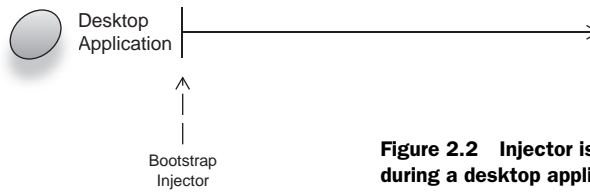


Figure 2.2 Injector is bootstrapped during a desktop application's startup.

- *On demand*, every time it is needed (for instance, on remote invocations [figure 2.3])

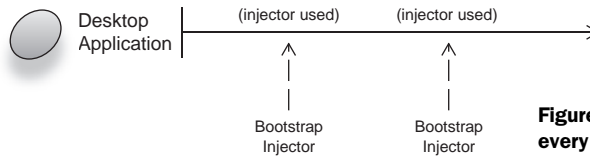


Figure 2.3 Injector is bootstrapped every time it is needed (on demand).

- *Lazily*, when it is *first* needed (figure 2.4)

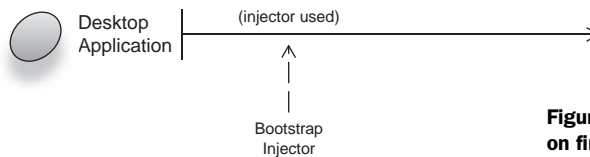


Figure 2.4 Injector is bootstrapped on first use (lazily).

Each option conforms to a different architectural scenario. None is *the* correct choice for all environments. It is typical to find that injector startup coincides with application startup. Once the injector has been bootstrapped, we can obtain an object from it for use.

2.2 Constructing objects with dependency injection

Let's look at one such scenario with a dependency injector named Guice. In this example, we will let Guice create and assemble an `Emailer` for us and provide it with dependencies (an `English spell-checker`). See figure 2.5.

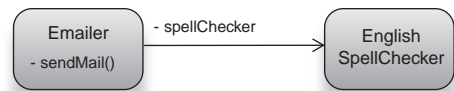


Figure 2.5 A class model of the emailer and English spellchecker

```
Guice.createInjector().getInstance(Emailer.class).send("Hello");
```

Notice that we created the injector and obtained the emailer from it, all in one line—this is an example of the *on-demand* bootstrapping of injectors. In the example, we have done three things worth talking about:

- 1 Created the injector.
- 2 Obtained an instance of `Emailer` from the injector.
- 3 Signaled the `Emailer` to send an email message.

Figure 2.6 illustrates this sequence.

Let's break down this example in code. The first invocation is a simple static method that asks Guice for a new injector:

```
Guice.createInjector()
```

We can think of Guice as a factory class for dependency injectors (recall the Factory pattern from chapter 1). This is fairly straightforward. The second invocation is of greater interest; it asks the injector for an instance of `Emailer`:

```
Guice.createInjector().getInstance(Emailer.class)
```

The `Emailer` obtained from our injector is properly wired with its dependencies, so when it needs to send a message it can call on its `SpellChecker` to do the spelling verification—all very tidy with a single-use injector:

```
Guice.createInjector().getInstance(Emailer.class).send("Hello!");
```

This, of course, is functionally identical to the more verbose form:

```
Injector injector = Guice.createInjector();
Emailer emailer = injector.getInstance(Emailer.class);
emailer.send("Hello!");
```

This is reasonable at first glance and seems to have brought us some kind of compelling result. However, upon closer inspection it evokes a sense of *déjà vu*. Remember this?

```
Emailer emailer = (Emailer) new ServiceLocator().get("Emailer");
```

Comparing this to obtaining an emailer from the injector yields more than a suspicious similarity:

```
Emailer emailer = injector.getInstance(Emailer.class);
```

Ask injector for an
Emailer by key

It looks as though `injector` is performing the role of a `Service Locator` pattern and that I've used `Emailer.class` as a key that identifies emailers. As we've already noted, keys need not be strings, so it seems I have indeed used a `Service Locator`. That can't be right, can it? Well, yes, because an injector can itself be used as a `Service Locator`.

So what is the hype really about? In a sense the injector is an automated, all-purpose service locator prepackaged for convenience. This is a fair characterization, but with *three important differences*, and the hype is really about these three things:

- *Client code (the Emailer) is not invoking the injector, unlike with service locators.* Rather, it is the application bootstrapper that invokes the injector. The key difference here is that the latter does not participate in the behavior of the program.

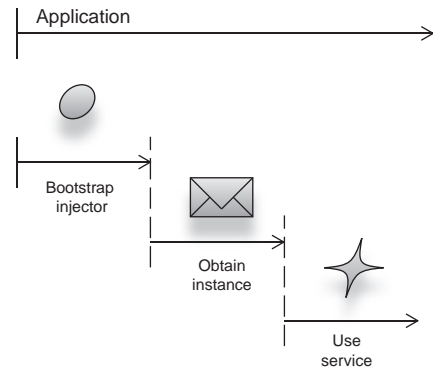


Figure 2.6 Injector obtains emailer for a client, which is then used to send mail.

- *Only a root object is obtained from the injector.* This object is generally the *entry point* of an application and is typically requested only once. This means that no matter how many client-service systems exist within the application, their creation, injection, and usage cascades from the root object. For example, an HTTP servlet is a root object, with data-source dependencies injected into it.
- *The injector obtains instances within the current execution context.* In other words, the instance returned may or may not be different depending on the state of the application and the current context. This is known as *scoping* and is a very powerful feature of dependency injection—one that essentially separates it from Service Location. See chapter 5 for more on scopes.

In most real-world applications the act of obtaining and running objects is typically done for you by integration layers, such as the *guice-servlet*¹ extension does with HTTP servlets and Guice. Many DI libraries provide such integration layers out of the box.

This introduces us to the notion that a single unit is an entry point into a program, like a starting line from which other objects (dependencies) stem. The “offspring” of the root object are built and wired as required. In turn, each object may construct and call several of its own dependencies, and so on down the tree of dependencies. This cascading creation of objects is called *viral injection*.

Unlike traditional Service Location patterns this means your code is not *explicitly* dependent on a service (the dependency injector). Of course, apart from this is the not-insignificant fact that you don’t have to write, test, or maintain the injector itself.

So what do our *Emailer* (client) and its *SpellChecker* (dependency) look like? Listing 2.1 has one version.

Listing 2.1 An emailer with a spellchecker

```
import com.google.inject.Inject;

public class Emailer {
    private SpellChecker spellChecker;

    @Inject
    public Emailer(SpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }

    public void send(String text) {
        spellChecker.check(text);
        //send if ok...
    }
}
```

Dependencies, wired via constructor

An annotation tells Guice to use this constructor

Dependency used directly when needed

Notice the `@Inject`² annotation ❶ placed on *Emailer*’s constructor. This piece of metadata tells Guice to use the annotated constructor. The injector constructs and

¹ Guice-servlet allows your HTTP servlets to be created and injected by Guice. A detailed example is provided in chapter 5. For more on guice-servlet, visit <http://code.google.com/p/google-guice>.

² The `@Inject` annotation ships with Guice’s library distribution.

wires the `Emailer` using it. Guice inspects classes at runtime and determines how to build them. It detects that `SpellChecker` must also be constructed and provided to `Emailer` before it is ready to use.

NOTE Annotations are a feature of Java that allow classes and programs to be enhanced with extra information. Tools, libraries, or even humans can use them to glean additional meaning or intent about the program. Annotations do not manipulate program semantics directly but instead provide a standard way of enhancing them along with appropriate tools.

When `send()` is called, `Emailer` has been wired with a `SpellChecker` and can be called on to check() spelling.

In the popular Spring Framework, the same thing is achieved with slightly different annotations:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class Emailer {
    private SpellChecker spellChecker;

    @Autowired
    public Emailer (SpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }

    public void send(String text) {
        spellChecker.check(text);
        //send if ok..;
    }
}
```

← Instead of
@Inject

What we see here is one way of passing an injector some information via program metadata. In the next section, we'll see how injectors are *configured* in different ways.

2.3 Metadata and injector configuration

Annotations and custom attributes are an elegant and unintrusive form of metadata that helps indicate some information about your code to an injector, namely, which constructor to use (and consequently what its dependencies are).

In essence, what I did was to use the `@Inject` annotation to configure the injector (`@Autowired` and `@Component` for Spring), providing it with the appropriate information it needed to construct the `Emailer`. DI libraries provide a number of mechanisms for configuring the injector (as shown in figure 2.7):

- Using an XML configuration file
- Invoking into a programmatic API
- Using an injection-oriented DSL
- By inference; that is, implicitly from the structure of classes
- By inference from annotated source code

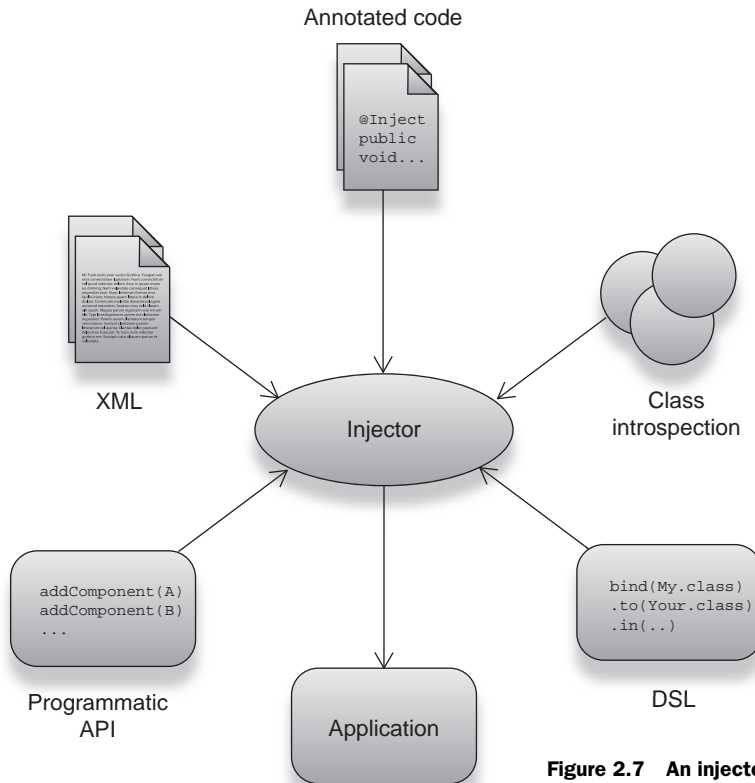


Figure 2.7 An injector can be configured in several different ways.

Or some combination of mechanisms.

That's certainly a lot of options. And choosing among them doesn't always come down to taste. Before judging the effectiveness of each option, let's take our emailer out for a spin with a couple of available options.

2.3.1 XML injection in Spring

Spring's primary mode of configuration is an XML file. Though Spring offers many alternatives (we saw one with `@Autowired`), the XML configuration is its most easily identified and common configuration mechanism. Recall our three wonted steps:

- Creating the injector
- Obtaining a component instance from it
- Using the instance to send mail (spamming away!)

In Guice this was written as:

```
Guice.createInjector().getInstance(Emailer.class).send("Hello!");
```

Translating to Spring yields the following:

```

BeanFactory injector = new FileSystemApplicationContext("email.xml");
Emailer emailer = (Emailer) injector.getBean("emailer");
emailer.send("Hello!");

```

Here, `BeanFactory` refers to the Spring dependency injector. The call to `getBean()` is analogous to the original call to `getInstance()`. Both return an instance of `Emailer`. One interesting difference is that we use a string key to identify emailers (and are therefore required to *downcast*³ it into the variable `emailer`):

```

Emailer emailer = (Emailer) injector.getBean("emailer");

```

Notice that sending an email remains exactly the same in both DI libraries:

```

emailer.send("Hello!");

```

This is an important point, because it means that using code for its original, intended purpose remains exactly the same regardless of what library you choose. And it underlines the fact that dependency injection is not an invasive technique.

Note also how an injector is created. Rather than using a Factory, I've used construction by hand to create a `FileSystemApplicationContext` and assigned it to a variable of type `BeanFactory`. Both `BeanFactory` and `FileSystemApplicationContext` refer to Spring's injector. The latter is a specific kind of `BeanFactory` (just like an `EnglishSpellChecker` is a specific kind of `SpellChecker`).

The file `email.xml` contains the configuration required for Spring's injector to provide us with an `emailer` correctly wired with its dependencies. Listing 2.2 shows what `email.xml` looks like, and figure 2.8 shows how this works.

Listing 2.2 Spring's injector configuration, `email.xml`

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd" >

    <bean id="emailer" class="Emailer">
        <constructor-arg ref="spellChecker"/>
    </bean>

    <bean id="spellChecker" class="SpellChecker"/>
</beans>

```

Boilerplate to declare this as a Spring configuration file ①

Inject object bound to "spellChecker" ③

Declare instances of Emailer ②

Declare instances of SpellChecker ④

Let's look at this in some detail. For clarity, we will omit the boilerplate XML schema declaration⁴ at the top of the file ①. First, a `<bean>` tag ② declares a component of

³ *Downcasting* (or *casting*) is the checked process of converting an instance of a general type into that of a more specific type; this is typically done by assigning the instance to a variable of the target type. In the given case, the general type `Object` is cast into a more specific type `Emailer`.

⁴ An XML schema is a formal description of the structure of an XML document toward a particular purpose. In this case, it describes the legal form and structure of a Spring configuration file.

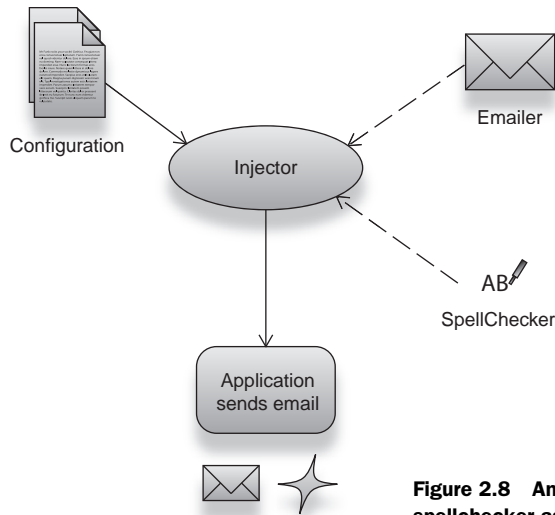


Figure 2.8 An emailer is wired with a spellchecker as per configuration by the injector.

class `Emailer` bound to the string identifier "emailer." This identifier is the key used to identify instances of `Emailer` in the vocabulary of the injector.

```
<bean id="emailer" class="Emailer">
```

Recall that in Guice the key served as both identifier as well as the binding to the `Emailer` class and was *implicit*. In Spring's XML, this binding takes the form of an explicit declaration in a `<bean>` element. Dependencies of `Emailer` are similarly declared in their own `<bean>` tags, in this case a `SpellChecker`:

```
<bean id="spellChecker" class="SpellChecker">
```

Now turn your attention back to the first `<bean>` tag ③, where the interesting part lies:

```
<bean id="emailer" class="Emailer">
  <constructor-arg ref="spellChecker"/>
</bean>
```

The `<constructor-arg>` tag tells Spring to use constructor wiring. It also indicates that `Emailer` instances should be wired with objects identified by a string key, "spellChecker." `SpellChecker` itself is declared ④ in another `<bean>` tag. This method of configuration has the somewhat useful advantage of being explicit about everything, but that is not necessarily a benefit in itself. As you will see later, it can be more flexible under certain circumstances. Also worth mentioning is a more compact rendering that does the same job for us:

```
<beans ...>
  <bean id="emailer" class="Emailer">
    <constructor-arg><bean class="SpellChecker"/></constructor-arg>
  </bean>
</beans>
```

In this case, I've nested the declaration of `SpellChecker` inside `Emailer`'s constructor-argument tag and omitted its identifier altogether. This is nicer for a couple of reasons:

- The description of `SpellChecker` is available right inside that of `Emailer`.
- `SpellChecker` is not exposed via its own identifier. Therefore it can be obtained only within the context of a `Emailer`.

Both these ideas provide us with a neat encapsulation of `Emailer` and its dependencies. Dependency encapsulation means you can't accidentally wire a `SpellChecker` meant for an `Emailer` to some other object (a Klingon editor, for example).

This concept of encapsulation is roughly analogous to *class member encapsulation*, which you should be familiar with in Java code. Listing 2.3 revisits our original `Emailer`'s source code for Spring.

Listing 2.3 The emailer, stripped of annotations

```
public class Emailer {
    private SpellChecker spellChecker;
    public Emailer(SpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }
    public void send(String text) {
        spellChecker.check(text);
        //send if ok...;
    }
}
```

← private keyword hides (encapsulates) members

There is no change—except that now I no longer need Guice's annotations and can safely remove them, leaving me with a pristine, shiny `Emailer`.

2.3.2 From XML to in-code configuration

Both Spring and Guice provide the notion of explicit configuration (in Spring, as you saw, via XML). The advantage to this approach is that you have a central directory of all the services and dependencies used in a particular application. In a large project with many developers working on the same source code, such a directory is especially useful. In Guice this file is called a `Module`.

Guice modules are regular Java classes that implement the `com.google.inject.Module` interface (shown in listing 2.4).

Listing 2.4 A typical Guice module

```
import com.google.inject.Binder;
import com.google.inject.Module;

public class MyModule implements Module {
    public void configure(Binder binder) {
        ...
    }
}
```

Services are registered by using the given binder and making a sequence of calls that configure Guice accordingly. This is similar to the XML `<bean>` tags you saw in the previous section. Listing 2.5 shows how to declare an emailer’s bindings explicitly.

Listing 2.5 Guice module registering emailers

```
import com.google.inject.Binder;
import com.google.inject.Module;

public class EmailModule implements Module {
    public void configure(Binder binder) {
        binder.bind(Emailer.class);
        binder.bind(SpellChecker.class);
    }
}
```

Register keys/services explicitly

This is pretty simple.

TIP To save yourself some typing, extend the `AbstractModule` class, which is a convenience provided by Guice. This marginally neater version with exactly the same semantics is shown in listing 2.6.

Listing 2.6 Slightly neater module registering emailers

```
import com.google.inject.AbstractModule;

public class EmailerModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(Emailer.class);
        bind(SpellChecker.class);
    }
}
```

Alternative to implementing Module

Service binding is more concise

Listings 2.5 and 2.6 are pretty close to Spring’s XML configuration. You should be able to see the similarity quite easily:

```
<beans ...>
  <bean id="emailer" class="Emailer">
    <constructor-arg ref="spellChecker" />
  </bean>
  <bean id="spellChecker" class="SpellChecker" />
</beans>
```

Before we look at keys and bindings (tying keys to their services) in greater detail, let’s look at a few more cool DI libraries.

2.3.3 Injection in PicoContainer

Guice configures the injector with annotations and optionally via inference; similarly Spring provides an XML facility for explicit injector configuration. Yet another alternative is provided by PicoContainer—this is similar to inference but takes the form of a programmatic API.

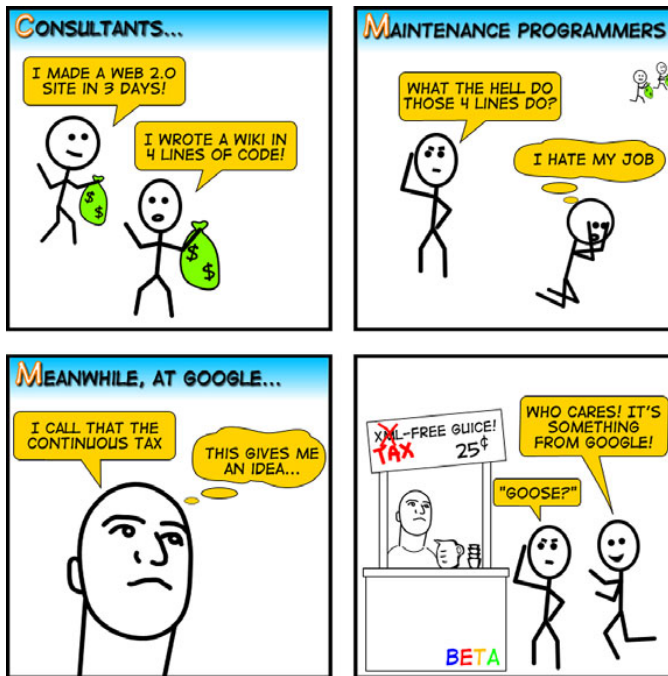


Figure 2.9 An ironic comic strip about Guice and creator Bob Lee from Eric Burke⁵

First, create the injector. Then grab an emailer from it, and dispatch those endearing greeting emails.

```
MutablePicoContainer injector = new DefaultPicoContainer();
injector.addComponent(Emailer.class);
injector.addComponent(SpellChecker.class);
injector.getComponent(Emailer.class).send("Hello!");
```

This is much closer to the original Guice version with the only real difference being that we use construction by hand to create the injector itself, namely, a `DefaultPicoContainer`. The method `getComponent()` is exactly like `getInstance()` in Guice (and `getBean()` in Spring), taking `Emailer.class` as an identifier.

NOTE Unless you are using `PicoContainer 2.0`, you will need to *downcast* the returned instance to type `Emailer`. In this example (and for the rest of this book), I've preferred `PicoContainer 2.0`, as it takes advantage of Java 5 and generics to add a measure of type safety.

What's this line doing here?

```
injector.addComponent(Emailer.class);
```

That appears to be a bit more than the three sweet steps of our wonted familiarity. In fact, what we have done is combine bootstrap and configuration in the same sequence

⁵ Read more of Eric's comic strips at <http://stuffthathappens.com>.

of method calls. The source code for `Emailer` itself does not change. So how does the injector know about `Emailer`'s dependencies? There has been no explicit description of `SpellCheckers` and whether or not to use constructor wiring. Nor have we annotated the class itself with `@Inject` or anything like it.

By default, `PicoContainer` prefers constructor wiring and greedily looks for available constructors. Greedy, in this sense, refers to the widest set of arguments that it can successfully provide to a class's constructor.

This can seem somewhat counterintuitive, but let's explore an example. For consistency (and because it is cool), I will stick with the `emailer`. If we imagine that the `emailer` now requires an editor to write messages in (listing 2.7 and figure 2.10), this can be modeled as a second dependency.

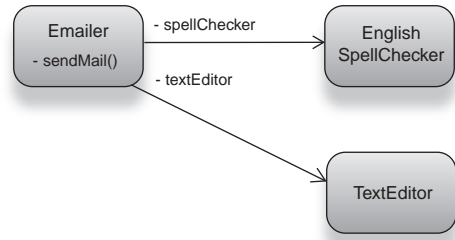


Figure 2.10 Emailer with two dependencies: `SpellChecker` and `TextEditor`

Listing 2.7 The `emailer`, now with two dependencies

```

public class Emailer {
    private SpellChecker spellChecker;
    private TextEditor editor;

    public Emailer(SpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }
    public Emailer(SpellChecker spellChecker, TextEditor editor) {
        this.spellChecker = spellChecker;
        this.editor = editor;
    }
    public void send() {
        spellChecker.check(editor.getText());
        // send if ok..
    }
}
  
```

**Second constructor
accepts both
dependencies**

①

Method `send()` now calls `editor.getText()` ① to get the contents of the composed message and send it. Listing 2.8 shows the changes to injector configuration required for this new functionality.

Listing 2.8 Injector bootstrap and using the `emailer` with its engine

```

MutablePicoContainer injector = new DefaultPicoContainer();
injector.addComponent(Emailer.class);
injector.addComponent(SpellChecker.class);
injector.addComponent(TextEditor.class);
Emailer emailer = injector.getComponent(Emailer.class);
  
```

**Method
`send()` directly
retrieves text
from the editor**

This code works because `PicoContainer` greedily picks the *second* constructor (the one with a greater number of arguments). This default behavior is confusing but fortunately

can be changed to suit your preference. We've now seen a couple of forms of configuration that tell a dependency injector how to construct objects. Next we'll use a different idiom: showing the injector how to construct objects.

2.3.4 *Revisiting Spring and autowiring*

I mentioned in several places that Spring also supports other configuration styles, including the inference style of Guice. One compelling style it offers that you will come across very often is *autowiring*. Autowiring, as the name suggests, is a mode where Spring automatically resolves dependencies by inference on class structure. Back to our email scenario, if this is the structure of the object I want to build with autowiring:

```
public class Emlaler {
    private SpellChecker spellChecker;

    public Emlaler(SpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }

    public void send(String text) {
        spellChecker.check(text);
        // send if ok...
    }
}
```

then I can have Spring wire Emlaler's SpellChecker dependency without explicitly specifying it:

```
<beans ...>
    <bean id="spellChecker" class="SpellChecker"/>

    <bean id="emlaler" class="Emlaler" autowire="constructor"/>
</beans>
```

Attribute `autowire="constructor"` informs the injector to guess dependencies by introspecting on Emlaler's constructor. While we still had to register all the available dependency classes in the XML file, there was no need to specify how the graph is wired. Now when an instance of Emlaler is constructed, it is automatically wired with a SpellChecker (see figure 2.11):

```
BeanFactory injector = new FileSystemXmlApplicationContext("email.xml");
Emlaler emlaler = (Emlaler) injector.getBean("emlaler");
```

And the sending of emails can continue unfettered!

```
emlaler.send("Hello!");
```

Constructor autowiring can save you a lot of typing in XML, though it does force you to look in two places to understand your object graph. It is especially useful if most of your dependencies are of unique types, that is, without many variants. And it is particularly useful if you can enforce the convention of only one constructor per class.

From Spring 2.5, you can even resolve the ambiguity between multiple constructors in a class by placing an `@Autowired` annotation on the particular constructor you

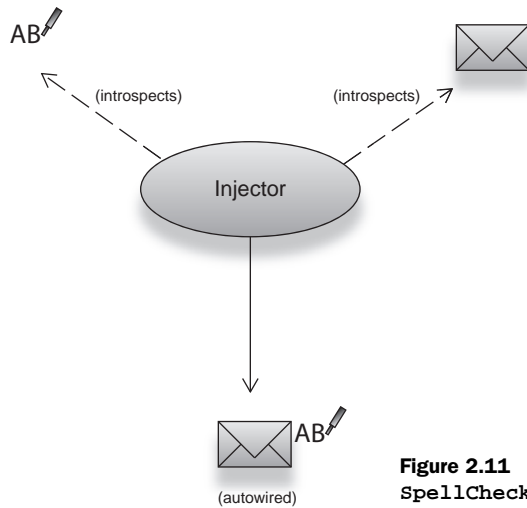


Figure 2.11 Emailer autowired with its `SpellChecker` with no explicit configuration

are interested in. Recall this from earlier in the chapter where we used it in place of `@Inject`. For example:

```

@Component
public class Emailer {
    private SpellChecker spellChecker;

    public Emailer() {
        this.spellChecker = null;
    }

    @Autowired
    public Emailer(SpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }

    public void send(String text) {
        spellChecker.check(text);
    }
}
  
```

`@Autowired` performs the same role as Guice's `@Inject` (in addition to `@Component` annotation placed on the class itself). It instructs the injector to choose the annotated constructor when introspecting the class for injection. Remember that you still need to place a `<bean>` tag in XML to make Spring aware of autowired classes:

```

<beans ...>
    <bean id="spellChecker" class="SpellChecker"/>
    <bean id="emailer" class="Emailer"/>
</beans>
  
```

Note the absence of any explicit wiring directive (`<constructor-arg>` or `<property>`) that we used in the pure XML configuration system. Note also the more conspicuous absence of the `autowire="..."` attribute, which is unnecessary now that we have `@Autowired`.

Like Guice, this method is both concise and *exemplar*. In other words, what you see in constructor code is what you get in the object graph. The next few sections examine this concept in detail.

2.4 Identifying dependencies for injection

We’ve already seen that a service generally has some kind of identity, whether an arbitrary string identifier, the class it belongs to, or some other combinatorial key. In this section we will briefly examine a few approaches and the benefits or drawbacks of each.

First, let’s formally address what it means to identify components and dependencies. Recall our discussion of the construction and assembly of object graphs from chapter 1. Essentially, a dependency is some implementation of a service. It may only be a flat object with no dependencies of its own. Or it may be an object with a vast graph of interconnected dependencies, which themselves have dependencies, and so on.

To sum up:

- A dependency is any particular permutation of objects in a graph that represents the original service; that is, all permutations obey the same *contract*.
- Permutations of the same service may be thought of as *variants* (or implementations) of that service.

Recall the example of the French and English Emailers. They are two variants of the same service (formed by two *different* object graphs). As shown in listing 2.9, dependent Emitter is the same for either variant.

Listing 2.9 An email service with French or English spellcheckers

```
public class Emitter {
    private SpellChecker spellChecker;

    public Emitter(SpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }

    public void send(String text) {
        spellChecker.check(text);
        // send if ok...
    }
}

interface SpellChecker {
    public boolean check(String text);
}

class FrenchSpellChecker implements SpellChecker {
    public boolean check(String text) {
        //perform some checking...
    }
}
```

Injected spellchecker
wired via constructor

Provided dependency
is used transparently

Interface represents
Emitter's dependency

French implementation
of SpellChecker

```
class EnglishSpellChecker implements SpellChecker {  
    public boolean check(String text) {  
        //perform some checking...  
    }  
}
```

English implementation
of SpellChecker

Were we to construct them by hand, there would be two possible paths to take. First, for the English email service (the one with the English spellchecker):

```
Emailer emailer = new Emailer(new EnglishSpellChecker());  
emailer.send("Hello!");
```

Second, the version with French spelling:

```
Emailer emailer = new Emailer(new FrenchSpellChecker());  
emailer.send("Bonjour!");
```

In both these cases, we want an `Emailer` but with different dependencies set. Using a simple string identifier such as `emailer` or a class identifier like `Emailer.class` is clearly insufficient. As an alternative we might try these approaches:

- Use a more specific string identifier, for example `"englishEmailer"` or `"frenchEmailer"`
- Use the class identifier `Emailer.class` together with a discriminator, for example, the ordered pair `[Emailer.class, "english"]`

Existing DI libraries use various methods to solve this problem, usually by falling into one of the two aforementioned paths. Both these options give us an *abstract identity* over a particular service implementation. Once its object graph has been described, a service implementation is easily identified by this abstract identity (which we refer to as its key).

If you wanted to use one particular implementation instead of another, you'd only need to point the dependent to its key. Keys are also essential when we want to leverage other features of dependency injectors, such as:

- *Interception*—Modifying an object's behavior
- *Scope*—Managing an object's state
- *Lifecycle*—Notifying an object of significant events

Keys provide a *binding* between a label and the object graph that implements the service it identifies. The rest of this chapter will focus primarily on the merits and demerits of approaches to keys and bindings.

2.4.1 Identifying by string keys

A key that explicitly spells out the identity of the service implementation should generally have three properties:

- It is *unique* among the set of keys known to an injector; a key must identify only one object graph.

- It is *arbitrary*; it must be able to identify particular, arbitrary service implementations that a user has cooked up. In other words, it has to be more flexible than the service *name* alone.⁶
- It is *explicit*; it must clearly identify the object graph, preferably to the letter of its function.

While these are not hard-and-fast rules, they are good principles to follow. Too often, people dismiss the worth of clear and descriptive keys. Let's look at real-world examples:

A Set in Java is a data structure that has many implementations. Among other things, the contract of Set disallows duplicate entries. The core Java library ships with the following implementations of Set:

- `java.util.TreeSet`—A *binary-tree*⁷ implementation of the Set service
- `java.util.HashSet`—A *hash-table*⁸ implementation of the Set service

How should we choose keys for these variants? The names of these implementations give us a good starting point—nominally, "binaryTreeSet" and "hashTableSet." This certainly isn't rocket science! These keys are explicit, sufficiently different from one another to be unique, and they clearly identify the behavior of the implementation (either binary-tree or hash-table behavior). While this may seem fairly obvious, it is not often the case. You'd be surprised at how many real-world projects are obfuscated with unclear, overlapping keys that say little if anything about an object graph. It is as much for your benefit as the developer and maintainer of an application as it is for the dependency injector itself that you follow these guidelines. Lucid, articulate keys are easily identified and self-documenting, and they go a long way toward preventing accidental misuse, which can be a real nuisance in big projects.

I also strongly encourage use of *namespaces*, for example, "set.BinaryTree" and "set.HashTable," which are nicer to read and comprehend than "binaryTreeSet" and "hashTableSet." Namespaces are a more elegant and natural nomenclature for your *key space* and are eminently more readable than strings of grouped capitalized words. An email service with a French bent might be "emailer.French," and its counterparts might be "emailer.English," "emailer.Japanese," and so forth.

I am especially fond of namespaces in dependency injection. Their value was immediately apparent to me the first time I was on a project with a very large number of XML configuration files. They allowed my team and me to clearly separate services by areas of function and avert the risk of abuse or misapprehension. For instance, data services were prefixed with `dao` (for Data Access Object) and business services with `biz`. Helper objects that sat at the periphery of the core business purpose were confined to a `util`.

⁶ Recall the example of `SpellChecker` (the service) being insufficient to distinguish between its English and French implementations.

⁷ A binary tree is a data structure in which each stored item may have two successors (or children), starting at a single root item.

⁸ A hash table is a data structure designed to store and look up entries in a single step using a calculated address known as a hash code.

namespace. I can't emphasize enough how useful this was for us. Consider some of the changes we made in a vast system of objects and dependencies identified solely by string keys (also see figure 2.12):

- UserDetailsService became dao.User.
- UserDataUtils became util.Users.
- DateDataUtils became util.Dates.
- UserService became biz.Users.

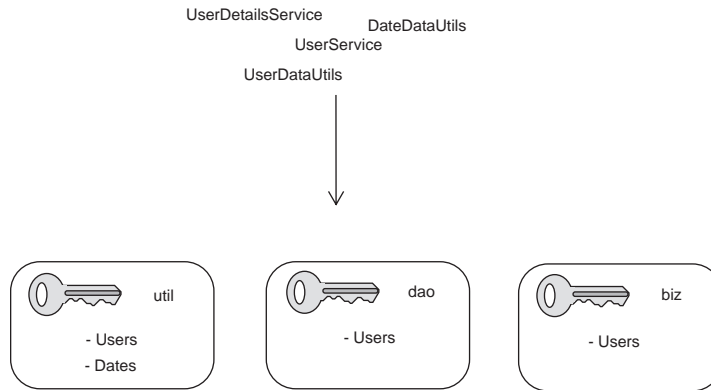


Figure 2.12
Organizing string keys
into namespaces

The astute reader will appreciate how much clearer—and more succinct—the latter form is. Of course, there is nothing new or innovative about the concept of namespaces, though one rarely sees it preached in documentation.

Spring and its XML configuration mechanism benefit heavily from this approach. If we were to declare the variant implementations of a Set, they may look something like this:

```
<beans ...>
  <bean id="set.HashTable" class="java.util.HashSet"/>
  <bean id="set.BinaryTree" class="java.util.TreeSet"/>
</beans>
```

And obtaining these objects from the injector accords to their keys:

```
BeanFactory injector =
    new FileSystemXmlApplicationContext("treesAndHashes.xml");
Set<?> items = (HashSet<?>) injector.getBean("set.HashTable");
```

For a more complete scenario, consider this pattern for the email service and its two variant spellcheckers (listing 2.10).

Listing 2.10 Variants of an email service using namespaces

```
<beans>
  <bean id="spelling.English" class="EnglishSpellChecker"/>
  <bean id="emailer.English" class="Emailer">
```

```

        <constructor-arg ref="spelling.English"/>
    </bean>

    <bean id="spelling.French" class="FrenchSpellChecker">
        <bean id="emailer.French" class="Emailer">
            <constructor-arg ref="spelling.French"/>
        </bean>
    </bean>
</beans>

```

Better yet, listing 2.11 is a more compact, encapsulated version of the same.

Listing 2.11 A compact form of listing 2.10

```

<beans ...>
    <bean id="emailer.English" class="Emailer">
        <constructor-arg><bean class="EnglishSpellChecker"/></constructor-arg>
    </bean>

    <bean id="emailer.French" class="Emailer">
        <constructor-arg><bean class="FrenchSpellChecker"/></constructor-arg>
    </bean>

    <bean id="emailer.Japanese" class="Emailer">
        <constructor-arg><bean class="JapaneseSpellChecker"/></constructor-arg>
    </bean>
</beans>

```

By now, you should be starting to appreciate the value of namespaces, encapsulation, and well-chosen keys. Remember, a well-chosen key is *unique*, *arbitrary*, and *explicit*. When you must use string keys, choose them wisely. And use namespaces.

To drive the point home, look at the following two setups in pseudocode: one that uses poorly chosen keys and a flat keyspace and a second that conforms to our principles of good key behavior. Listing 2.12 shows an injector configuration with bad keys and no namespaces. Ugly! Wouldn't you agree?

Listing 2.12 Poorly chosen keys for services (in pseudocode configuration)

```

configure() {
    personService      { ... }
    personDataService  { ... }
    personnelSoapService { ... }
    userAuth           { ... }
    userAuthz          { ... }
    userService        { ... }
    database            { ... }

    app {
        app = new Application()
        ...
    }
}

```

← Start configuration
 Services for personnel management
 Security services
 ← Wiring logic goes here

Now compare this with listing 2.13 (and figure 2.13), which presents an improved version of the same configuration. Much better!

Listing 2.13 Good-practice version of listing 2.12

```
configure() {
    define namespace :biz {
        biz.personnel { ... }
        biz.user
    }
    define namespace :data {
        data.database { ... }
        data.personnel { ... }
    }
    define namespace :soap {
        soap.personnel { ... }
    }
    define namespace {
        security.authentication { ... }
        security.authorization { ... }
    }
}

main {
    personnelService = injector.biz.personnel
    personnelDao = injector.data.personnel
}
```

← **Services organized into biz namespace**

← **Data objects are in data namespace**

Security object keys named lucidly

Get and use instances from injector

String keys are flexible, and if chosen well they work well. But well chosen or not, string keys have limitations that can be confounding and rather, well, limiting! Let's talk about some of them.

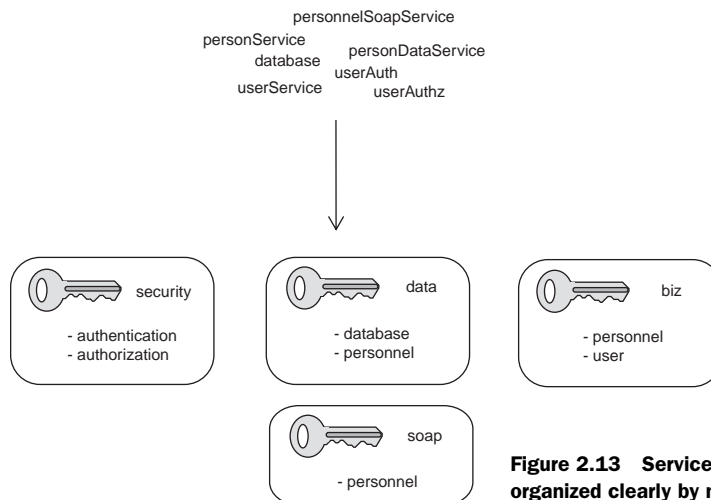


Figure 2.13 Services with similar names organized clearly by namespace

2.4.2 Limitations of string keys

String keys are inherently problematic when one factors human error into the equation. We have all been there. Take the possible ways to spell a common name, for example:

- Alan
- Allan
- Alain
- Allen
- Allun (an Irish variant)

This is a short name (two syllables) and it does not incur the distractions of case and multiple-word capitalization.⁹ Yet we can barely agree on a spelling even in the English-speaking world. Mistyping and misreading exacerbates the problem. It is not a stretch to imagine how things can get out of hand very quickly with a large number of services identified by many similar-looking (and sounding) keys. A misspelled key that maps to no valid service can be difficult to detect until well into the runtime of the program.

Furthermore, if you are working with a statically typed language like Java, this is a poor sacrifice to have to make. These languages are supposed to offer guarantees around type validity at compile time. Ideally, one should not have to wait until runtime to detect problems in key bindings.

Static and dynamic typing¹⁰

Types are specific classes of data in a programming language. For example, the number 32 is of an `Integer` type. Types may also be user defined, such as `Car` or `Song`. Before operations on data can occur, the data's particular type must be determined. This is known as *type resolution*. A *dynamically typed* language resolves types when an operation is *performed*, whereas a *statically typed* language resolves types when an operation is *defined* (typically, but not necessarily, at compile time).

Sometimes tools like IDEs or build agents can help do this extra level of checking for you, during or before compilation. *IntelliJ IDEA*¹¹ provides plug-ins for Spring for just this purpose. Naturally, it is difficult for any tool to guarantee the correct resolution of dependencies into dependents without bootstrapping the injector and walking across every bound object (or just about every one). As a result, this problem can arise often in injectors that use string keys.

⁹ A phrase or set of words in a key is often delimited by capitalizing each first letter. For example, “Technicolor dream coat” would be written as `TechnicolorDreamCoat` or `technicolorDreamCoat`, which are two forms of the CamelCase convention.

¹⁰ Do not confuse static and dynamic typing with *strong* and *weak* typing, or indeed with *duck* typing.

¹¹ IntelliJ IDEA is an advanced Java IDE developed by JetBrains. It is usually at the forefront of innovation in developer productivity. Find out more at <http://www.jetbrains.com>.

In your injector configuration there is no way to determine the type of a binding if all you have is a string key. Without starting up the injector and inspecting the specific instance for a key, it is hard to determine what type the key is bound to. Take the following example of a game console (the Nintendo Wii) and a game to be played on it:

```
<beans>
  <bean id="nintendo.wii" class="com.nintendo.Wii">
    <constructor-arg ref="game.HalfLife"/>
  </bean>

  <bean id="game.HalfLife" class="com.valve.HalfLifeGame"/>
</beans>
```

Here our keys appear reasonably well chosen: they are unique, arbitrary, and explicit (*and* they use namespaces). I've followed every tenet of good key behavior but something is horribly wrong with this injector configuration. The program crashes when I try to run it. To understand why, delve into the code behind these bindings:

```
package com.nintendo;

public class Wii {
  private WiiGame game;
  public Wii(WiiGame game) {
    this.game = game;
  }

  public void play() { .. }
}
```

Game is wired
via constructor

The Nintendo game system takes an argument of type `WiiGame` via its constructor. This dependency represents a game I want to play (I enjoy *Half-Life* so let's go with that) and is loaded by the game system when `play()` is called. This is the game *Half-Life*:

```
package com.valve;

public class HalfLife implements PCGame {
  public void load() { .. }
}
```

Oh, no! *Half-Life* is a game for the PC, *not* the Wii! It is not an instance of `WiiGame` and therefore cannot be wired into `Wii`'s constructor (see figure 2.14).

In Java, executing this program and requesting an instance by key `nintendo.wii` will result in an `UnsatisfiedDependencyException` being thrown. This indicates that the configured binding is of the wrong type for `Wii`'s dependency, and it cannot be converted to the right type.

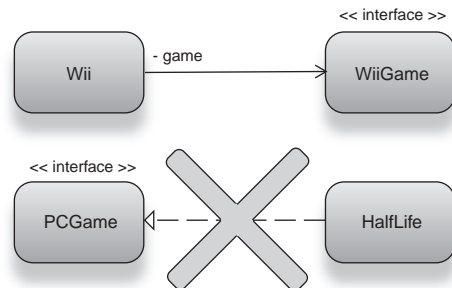


Figure 2.14 `Wii` and game `HalfLife`, which implements the wrong interface, `PCGame`

More significant for us, however, is the fact that what appeared to be a perfectly well-written injector configuration turned out to be fatally flawed. Worse, there was no way to detect this until the offending object graph was constructed and used. This highlights a serious limitation of string keys.

Once the problem is identified, we can fix it by selecting the correct implementation of `HalfLife`. Listing 2.14 shows this fix, illustrated in figure 2.15.

Listing 2.14 The fixed version of `games.xml`

```
<beans>
  <bean id="nintendo.wii" class="com.nintendo.Wii">
    <constructor-arg ref="game.HalfLife"/>
  </bean>

  <bean id="game.HalfLife" class="com.valve.wii.HalfLife"/>
</beans>
```

Changed to
a variant of
HalfLife

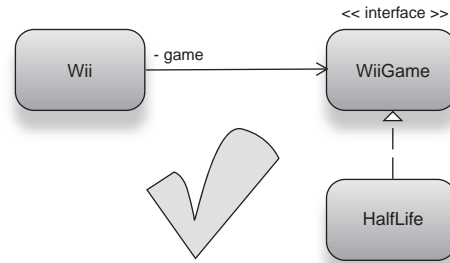
This time, the game is of the appropriate type and everything works as expected.

```
package com.valve.wii;

public class HalfLife implements WiiGame {
  public void load() { .. }
}
```

Dependency variant `HalfLife`
is now of the correct type

In large applications with many hundreds of object graphs, string identifiers incur many of these problems. No matter what language or injector you decide on, poorly chosen keys are a recipe for disaster. Oh, and use namespaces. Seriously!



2.4.3 Identifying by type

We have seen how services are identified by a type. Recall some of the examples I've used so far in this book:

- `SpellChecker.class` identifies `EnglishSpellChecker` or `FrenchSpellChecker`.
- `Emailer.class` identifies itself.
- `WiiGame.class` identifies `HalfLife`.
- `Wii.class` identifies itself.

Unlike strings, referring to a type is not quite the same in all languages, but the essential semantic is the same. Java uses *type literals* that are analogous to *string literals* and essentially identify the type directly in code:

- `Set.class`
- `SpellChecker.class`
- `Airplane.class`

Figure 2.15 `HalfLife` now correctly implements `WiiGame`.

Identifying by type is a natural and idiomatic way of referring to a dependency. It is self-descriptive and can be inferred by introspection tools automatically. This has additional advantages in a statically typed language where types are checked at compile time and any errors in types are revealed early.

Look at how identifying by type helps prevent the very error we just saw with string keys and the incorrect wiring of dependencies. Listing 2.15 revisits the broken dependency where the wrong version of the game `HalfLife` is used in the game console.

Listing 2.15 The class `com.nintendo.Wii` bound to key `nintendo.wii`

```
package com.nintendo;

public class Wii {
    private WiiGame game;
    public Wii(WiiGame game) {
        this.game = game;
    }

    public void play() { .. }
}

public class HalfLife implements PCGame {
    public void load() { .. }
}
```

Injected game wired via constructor

HalfLife implements the wrong type!

This time, instead of XML configuration and Spring, I use Guice and its Module binding:

```
public class MyGamesModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(WiiGame.class).to(HalfLife.class);
    }
}
```

Configured via subclasses of Module or AbstractModule

This line results in a compiler error

This injector configuration ❶ will not compile, failing with an error saying there is no such method `to()` ❷ taking an argument of type `HalfLife`. The error is caught nice and early. What the compiler really means is that type key `HalfLife.class` refers to the wrong type. Or, it is *not* an implementation of `WiiGame`. Another nice thing is that bad spelling is also caught early, with a compiler error. IDEs help you catch this kind of error as you are writing code, and let you smart complete the correct values when typing, which is especially nice. We can also achieve a similar end using Spring's `JavaConfig` configuration mechanism:

```
@Configuration
public class MyGamesConfig {
    @Bean
    public WiiGame game() {
        return new HalfLife();
    }

    @Bean
    public Wii gameConsole() {
```

Binding of dependency HalfLife

```

        return new Wii(game());
    }
}

```

← Type checked like
normal Java code

Clearly, identifying by type is a winner when compared to plain string keys. PicoContainer also offers identifying by type in a similar fashion to what we've just seen with Guice. However, identifying by type does have serious limitations. In the next sections we'll explore what these are and how they may be overcome.

2.4.4 Limitations of identifying by type

I'll wager you're already familiar with some of the limitations of type (or *class literal* keys). We saw the primary one at the beginning of this section—the inability to distinguish between different implementations of a service. The type key `Car.class` can refer either to `BmwZ4s` or to `ToyotaCamrys`, as shown in figure 2.16.

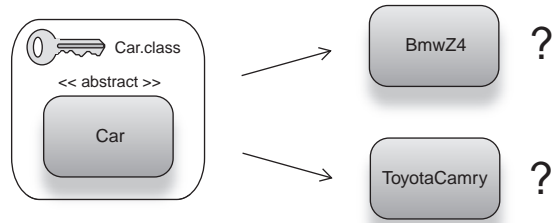


Figure 2.16 Which implementation does type key `Car.class` refer to?

And the injector is none the wiser as to which one to select and wire to clients of `Car`. This is irksome if we wanted to plug in a variety of alternative implementations by just changing keys.

You can refer to a specific implementation *directly* (like `BmwZ4.class` instead of `Car.class`), but this is a poor solution and incurs the same problem of the one default implementation (can't replace it with a `ToyotaCamry`). Worse, it is difficult to test and tightly couples client code to dependencies, which is undesirable.

Identifying by type can be restrictive in other ways. Recall the three holy tenets of well-chosen keys that helped make them lucid and maintainable. Keys must be unique, arbitrary, and explicit. How does identifying by type fare under these tenets?

- *Not unique*—Variant implementations of a service are indistinguishable by type key alone (`BMWZ4s` and `ToyotaCamrys` both have type key `Car.class`).
- *Not arbitrary*—A key bears only the label of the service (that is, the dependency). There is no room to describe arbitrary variants. The game console accepts a `WiiGame`, not `HalfLife` or `SuperMarioBrothers`. Try getting your kid to spend his money on a nondescript, gray disc over a shiny new copy of `Half-Life`.
- *May or may not be explicit*—One could argue that there is no better explanation of the service's function than its contract. However, were a service contract to leave some details up to its implementations, or likely to other dependencies, we would lose all explicit enunciation, as in an `Emailer` with English spellchecking rather than French (both identified by `Emailer.class`).

It does not fare very well at all: one or possibly none out of three important requisites of good key behavior! Type keys are safer in the context of detecting errors early on, particularly in statically typed languages like Java and C#, but they are rigid and don't

offer the breadth of abstraction that string keys do. String keys, on the other hand, are dangerous through misspelling and can be abstruse and opaque if improperly chosen. To summarize: type keys are *safe but inflexible*, and string keys are *flexible but unsafe* and potentially confusing.

It would be ideal if there were a key strategy that was both type and contract safe—and was flexible enough to wire dependents with many implementations that we could identify easily. Fortunately for us, *combinatorial keys* provide the best of both worlds.

2.4.5 Combinatorial keys: a comprehensive solution

Earlier, I described a sample combinatorial key for an English variant of our favorite email service; it looked like this:

```
[Emailer.class, "english"]
```

The ordered pair `[Emailer.class, "english"]` is one case of a combinatorial key consisting of a type key *and* a string key. The type key identifies the service that dependents rely on, in a safe and consistent manner. Its counterpart, the string key, identifies the specific variant that the key is bound to but does so in an abstract manner. Comparing this combinatorial key with a plain string key, the latter clearly has the advantage of type safety. Its second part (the string key) provides a clear advantage over plain type keys because it is able to distinguish between implementations. Let's examine this with a couple of examples:

```
Key 1: [Emailer.class, "english"]
Key 2: [Emailer.class, "french"]
Key X: [Book.class, "dependencyInjection"]
Key Y: [Book.class, "harryPotter"]
```

Keys 1 and 2 identify object graphs of an email service with English and French spellchecking, respectively. Keys X and Y identify two different books.

Another example we've used before is an emailer that depends on a `SpellChecker`:

```
[Emailer.class, "withSpellChecker"]
```

Consider a more complex take on this example, where our emailer depends on a `SpellChecker`, which itself depends on a dictionary to resolve word corrections (as figure 2.17 models).

The combinatorial key identifying this object graph might look something like:

```
[Emailer.class, "withSpellCheckerAndDictionary"]
```

Even complex object graphs with many permutations are identified easily with combinatorial keys. They also comply with our rules for good key behavior:

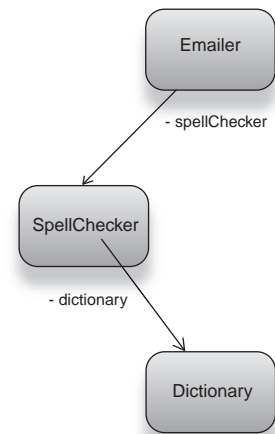


Figure 2.17 A class model of two levels of dependencies below emailer

- *Keys are unique.* The second part to the ordered pair ensures that two implementations are identified uniquely.
- *Keys are arbitrary.* While the dependency is identified by the first part, any specific behavior exhibited by a custom variant is easily described by the second part of the key.
- *Keys are explicit.* More so than with pure string keys, combinatorial keys are able to identify both the service contract they exhibit and any specific variation to the letter of its function.

So, combinatorial keys are safe, explicit, and able to identify any service variants clearly and precisely. They solve the problems of type-safety that pure string keys can't, and they ease the rigidity of pure type keys. And they do so in an elegant manner. The only drawback is that we have a string part that incurs the problems of misspelling and misuse identified earlier. For example, it is quite easy to see how the following combinatorial keys, relatively safe though they are, can still end up problematic:

```
[Emailer.class, "englihs"]
[Emailer.class, "English"]
[Emailer.class, "français"]
[Emailer.class, "withFaser"]
[Emailer.class, "with.SpellChecker"]
[Emailer.class, "With.SpellChecker"]
```

Each of these keys will result in an erroneous injector configuration that won't be detected until runtime. While there are certainly far fewer possible *total* errors with this approach, there are nonetheless the same perils and pitfalls so long as we rely on a string key in any consistent and foundational manner. How then can we fix this problem?

It turns out we can still keep all the benefits of combinatorial keys and eliminate the problems that the string part of the key presents. By using an *annotation* in place of the string part of the combinatorial key, we get the benefit of a safe key and the flexibility of an arbitrary and explicit string key.

Consider the following combinatorial keys that are composed of a type and an *annotation* type:

```
[Emailer.class, English.class]
[Emailer.class, WithSpellChecker.class]
[Database.class, FileBased.class]
```

Misspelling the annotation name results in a compiler error, which is an early and clear indication of the problem. This is exactly what we were after. What we have done is effectively replace the string key with a *second* type key. In combination with the service's type key, this retains the qualities of good behavior in well-chosen keys but also eliminates the problems posed by arbitrary, abstract strings. This is quite a comprehensive and elegant solution. The fact that annotations can be reused also makes them good for more than the one service:

```
[Emailer.class, English.class]
[Dictionary.class, English.class]
[SpellChecker.class, English.class]
```

We use the `@English` annotation to distinguish variant implementations of not only the `Emailer` but also the `Dictionary` and `TextEditor` services. These keys are self-documenting and still give us the flexibility of a string key. Since annotations are cheap to create (hardly a couple of lines of code) and reuse, this approach scales nicely too.

Guice embodies the use of type/annotation combinatorial keys and was probably the first library to use this strategy. Listing 2.16 and figure 2.18 show what the injector configuration might look like for the spellchecking service I presented in the previous example.

Listing 2.16 Guice module binding services to combinatorial keys

```
public class SpellingModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(SpellChecker.class)
            .annotatedWith(English.class)
            .to(EnglishSpellChecker.class);

        bind(SpellChecker.class)
            .annotatedWith(French.class)
            .to(FrenchSpellChecker.class);
    }
}
```

Then using either of these services in a client is as simple as using the appropriate annotation:

```
Guice.createInjector(new SpellingModule())
    .getInstance(Key.get(SpellChecker.class, English.class))
    .check("Hello!");
```

When this code is executed, Guice obtains the `SpellChecker` instance bound to the combinatorial key represented by `[SpellChecker.class, English.class]`. Invoking

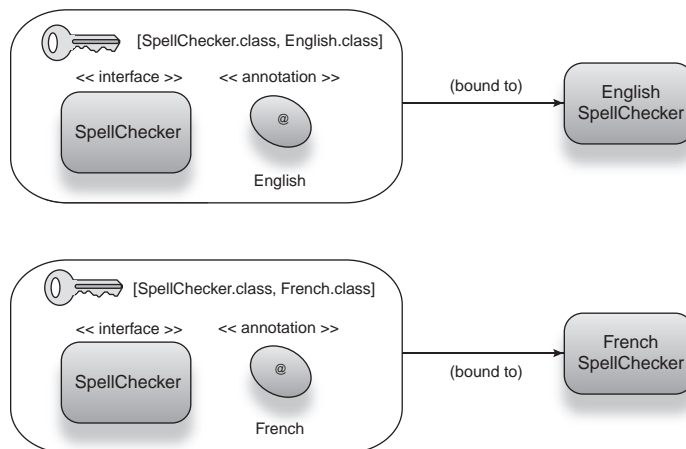


Figure 2.18
Combinatorial type keys use annotations to identify variant implementations of an interface.

the `check()` method on it runs the service method on the correct (that is, English) implementation.

This is also easily done in any *client* of `SpellChecker` via the use of `@English` annotation near the injection point of the dependency. Here's one such example:

```
public class SpellCheckerClient {

    @Inject
    public SpellCheckerClient(@English SpellChecker spellChecker) {
        //use provided spellChecker
    }
}
```

This might strike you as odd—while the client does not depend directly on a specific implementation, it *seems* to couple via the use of `@English` annotation. Doesn't this mean `SpellCheckerClient` is tightly coupled to English spellcheckers? The answer is a surprising no. To understand why, consider the following altered injector configuration:

```
public class SpellingModule extends AbstractModule {

    @Override
    protected void configure() {
        bind(SpellChecker.class)
            .annotatedWith(English.class)
            .to(FrenchSpellChecker.class);
    }
}
```

In this example, I've changed the service implementation bound to the combinatorial key `[SpellChecker.class, English.class]` so that any dependent referring to an `@English` annotated `SpellChecker` will actually receive an implementation of type `FrenchSpellChecker`. There are no errors, and `SpellCheckerClient` is unaware of any change and more importantly *unaffected* by any change. So they aren't really tightly coupled.

No client code needs to change, and we were able to alter configuration transparently.

Similarly, you can build such combinatorial bindings in Spring JavaConfig using types and method names:

```
@Configuration
public class EmailConfig {

    @Bean
    public SpellChecker english() {
        return new EnglishSpellChecker();
    }

    @Bean
    public Mailer mailer() {
        return new Mailer(english());
    }
}
```

**Binding of English
spellchecker**

←

**Create an
English mailer**

←

Here, the combinatorial key is `[SpellChecker, english()]`, the latter being the name of the method. It performs a very similar function to the `@English` annotation we saw just now, though the two approaches differ slightly.

Now let's look at how to take this practice a step further and separate code by area of concern.

2.5 Separating infrastructure and application logic

You have seen how object graphs that represent various services may be created, assembled, and referenced. You have also seen the steps to creating and using dependency injectors and ultimately the objects that they manage for an application. DI libraries differ slightly in the manner in which these steps are achieved and the rigor with which they are performed, but ultimately they all follow the same principles.

Together, these properties make for lean and focused components with code only to deal with their primary business purpose, be it:

- Rendering a web page
- Sending email
- Purchasing stock
- Checking bad spelling

Logic geared toward such purposes is termed *application logic*. Everything else is meant only to support and enhance it. Logic for constructing and assembling object graphs, obtaining connections to databases, setting up network sockets, or crunching text through spellcheckers is all peripheral to the core purpose of the application.

While this *infrastructure logic* is essential to all applications, it is important to distinguish it from the application logic itself. This not only helps keep application logic clear and focused, but it also prevents tests from being polluted with distracting bugs and errors that may have nothing to do with the code's purpose. Dependency injection helps in this regard. Figure 2.19 describes the fundamental injector and object composition that forms the basis of modern architectures.

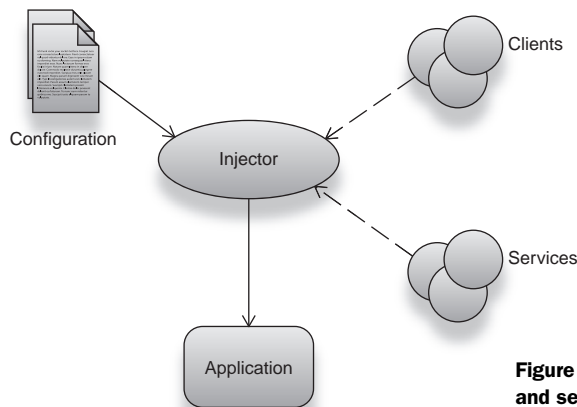


Figure 2.19 Injectors assemble clients and services as per configuration, into comprehensive applications.

Good DI libraries enforce and exemplify this core ideology. They are as much about preventing abuse as they are about proffering best practices or flexibility. As such, a DI library that takes extra steps to prevent you from accidentally shooting yourself in the foot (by checking bindings at compile time, for instance) is preferable. On the other hand, DI libraries that offer a great deal of flexibility (and weak type safety) can draw even experienced developers into traps. This is where careful design is important. Don't be afraid to refactor and redesign your code when you discover violations of good design. A little bit of effort up front to ensure that infrastructure logic remains separate from application logic will go a long way toward keeping your code readable, maintainable, and robust throughout its life.

2.6 **Summary**

This chapter was a headfirst dive into dependency injectors, their configuration, and their use. You saw how injectors must be bootstrapped with specified configuration and then used to obtain and perform operations on components they build and wire. At first, the injector looks no different to a service locator, and this is roughly correct in the context of obtaining the first, or the root, component from which the rest of an object graph extends.

All services that are used as dependencies are labeled by a key, which the injector uses as a way of referring to them during configuration or service location. The coupling of a key to a particular object graph is called a binding. These bindings determine how components are provided with their dependencies and thus how object graphs are ultimately constructed and wired. There are several kinds of keys, the most common being simply string identifiers, which are common in XML-configured DI libraries such as Spring, StructureMap, and HiveMind. String keys are flexible and provide us with the freedom to describe arbitrary and various assemblies of object graphs that portend a service. Different object graphs that conform to the same set of rules, that are the same service, may be thought of as different *implementations* of that service. String keys allow easy identification of such dependency *variants*.

However, string keys have many limitations, and the fact that they are unrestricted character strings means that they are prone to human error and to misuse and poor design choices. This leads to the necessity of well-chosen string identifiers that portend good key behavior. We defined the characteristics of well-chosen keys as being unique, arbitrary, and explicit. Unique keys ensure that no two object graphs are identified by the same key accidentally and so ensure that there is no ambiguity when a dependency is sought. Arbitrary keys are useful in supporting a variety of custom variants of services, which a creative user may have cooked up. Finally, these keys must also be explicit if they are to exactly describe what a service implementation does as clearly and concisely as is possible. Well-chosen keys combined with the use of namespaces greatly improve readability and also reduce the probability of accidental misuse.

String keys satisfy all of these qualities, but they have serious drawbacks since they lack the knowledge of the type (or class) they represent. This can result in syntactically

correct injector configuration that fails at runtime, sometimes even as late as the first use of a faultily configured object graph. In rare cases, it can even result in incorrect dependency wiring without any explicit errors, which is a very scary situation! In a statically typed language like Java, better solutions are imperative to these problems, especially in large projects that make use of *agile* software methods that require rapid, iterative execution of partial units.

An alternative is the use of type keys. These are keys that simply refer to the data type of the service. Type keys solve the misspelling and misuse problem of string keys but sacrifice a lot of flexibility and abstraction in doing so. PicoContainer is one DI library that supports the use of type keys. Their primary limitation is the inability to distinguish between various implementations without directly referring to them in client code. As a result, type keys violate almost all of the requirements of well-chosen keys, though they are safe and help catch errors early, at compile time.

One solution is to combine the two approaches and use type keys to narrow down the service's type and pair it with a string key that distinguishes the specific variant. These are called combinatorial keys, and they provide all of the flexibility of string keys and retain the rigor of type keys. PicoContainer also provides for these hybrid type/string combinatorial keys. However, the use of string identifiers at all still means that there is a risk of misspelling and accidental misuse. Guice provides a comprehensive solution: the use of custom annotation types in place of the string part of a combinatorial key. The combination of the type key referring to a service and an annotation-type key referring to an abstract identifier to distinguish between implementations provides for a compelling mitigation of the problems of even partial string keys.

Finally, whatever DI library you choose, dependency injectors are geared toward the same goals, that is, separate logic meant for constructing and assembling object graphs, managing external resources and connections, and so on from logic that is intended solely for the core purpose of the application. This is called the separation of infrastructure from application logic and is an important part of good design and architecture. The majority of the concepts presented in this book revolve around this core ideology and in many ways serve to emphasize or enhance the importance of this separation. A successful rendition of dependency injection gives you more time to focus effort on application logic and helps make your code testable, maintainable, and concise.

Investigating DI

This chapter covers:

- Learning about injection by setter and constructor
- Investigating pros and cons of injection idioms
- Identifying pitfalls in object graph construction
- Learning about reinjecting objects
- Discovering viral injection and cascaded object graphs
- Learning techniques to inject sealed code

“The best way to predict the future is to invent it.”

—Alan Kay

Previously we discussed two methods of connecting objects with their dependencies. In the main we have written classes that accept their dependencies via *constructor*. We have also occasionally used a single-argument method called a *setter method* to pass in a dependency. As a recap, here are two such examples from chapter 1:

```
public class Emailer {  
    private SpellChecker spellChecker;  
  
    public Emailer(SpellChecker spellChecker) {  
        this.spellChecker = spellChecker;  
    }  
}
```

The same class accepting dependencies by setter:

```
public class Emailer {  
    private SpellChecker spellChecker;  
  
    public void setSpellChecker(SpellChecker spellChecker) {  
        this.spellChecker = spellChecker;  
    }  
}
```

These are two common forms of wiring. Many dependency injectors also support other varieties and bias toward or away from these idioms.

The choice between them is not always one of taste alone. There are several consequences to be considered, ranging from scalability to performance, development rigor, type safety, and even software environments. The use of third-party libraries and certain design patterns can also influence the choice of injection idiom.

DI can make your life easier, but like anything else, it requires careful thought and an understanding of pitfalls and traps and the available strategies for dealing with them. The appropriate choice of injection idiom and accompanying design patterns is significant in any architecture. Even if you were to disavow the use of a DI library altogether, you would do well to study the traps, pitfalls, and practices presented in this chapter. They will stand you in good stead for designing and writing healthy code.

In this chapter, we will look at an incarnation of each major idiom and explain when you should choose one over another and why. I provide several strong arguments in favor of constructor injection. However, I will show when setter injection is preferable and how to decide between the two. Understanding injection idioms and the nuances behind them is central to a good grasp of dependency injection and architecture in general. Let's start with the two most common forms.

3.1 Injection idioms

The key to understanding the differences between setter and constructor injection is to understand the differences between methods and constructors. The advantages and limitations arise from these essential language constructs. Constructors have several limitations imposed on them by the language (for example, they can be called only once). On the other hand, they have advantages, such as being able to set final fields. We'll start by examining these limitations and advantages in detail.

3.1.1 Constructor injection

Essentially, a constructor's purpose is to perform initial *setup* work on the instance being constructed, using provided arguments as necessary. This setup work may be wiring of dependencies (what we are typically interested in) or some computation that is necessary prior to the object's use. A constructor has several odd and sometimes confounding restrictions that differentiate it from an ordinary method. A constructor:

- Cannot (does not) return anything.
- Must *complete* before an instance is considered fully constructed.

- Can initialize *final* fields, while a method cannot.
- Can be called only once per instance.
- Cannot be given an arbitrary name—in general, it is named after its class.
- Cannot be *virtual*; you are forced to implement a constructor if you declare it.

A constructor is something of an initialization *hook* and somewhat less than a method, even with a generous definition of methods. Constructors are provided in most object-oriented languages as a means of adequately preparing an object prior to use. The flow chart in figure 3.1 shows the steps taken internally to perform constructor injection.

A more thorough look at the pros and cons of construction injection is available later in this chapter. As we said earlier, setters offer benefits in areas where constructors are limited. They are just like any other method and therefore are more flexible. In the following section, we examine the advantages to this flexibility and where it can sometimes cause problems.

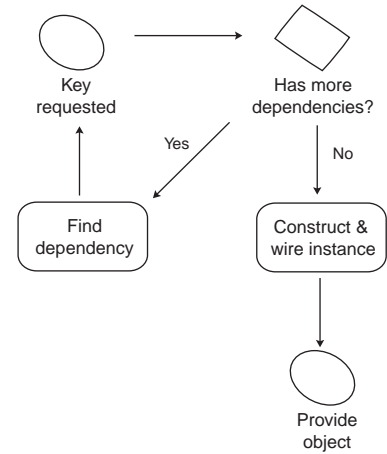


Figure 3.1 Sequence of operations in constructor injection

3.1.2 Setter Injection

The other approach we have seen (and the one that is very common) is setter injection. It involves passing an object its dependencies via arguments to a so-called setter method. Let's revisit some of our past uses of setter injection:

```

public class Emailer {
    private SpellChecker spellChecker;

    public void setSpellChecker(SpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }
}
  
```

Figure 3.2 depicts the flow of steps in how objects are wired using setter injection. Contrast this with the flow chart presented in figure 3.1.

This sequence is almost identical to that of constructor injection, except that a method named `setSpellChecker()` is used in place of a constructor and the wiring takes place *after* the instance is fully constructed. In common usage, if you want to pass in additional dependencies, you provide additional setters, as shown in listing 3.1 (modeled in figure 3.3).

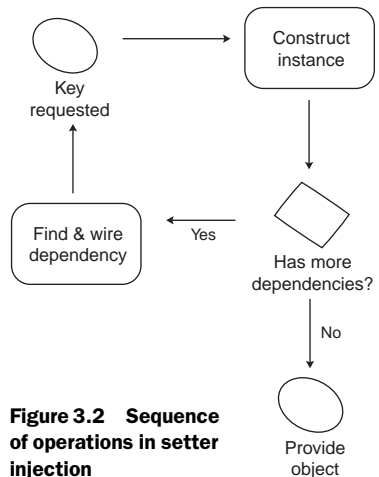


Figure 3.2 Sequence of operations in setter injection

Listing 3.1 A stereo amplifier wired by setter

```

public class Amplifier {
    private Guitar guitar;
    private Speaker speaker;
    private Footpedal footpedal;
    private Synthesizer synthesizer;

    public void setGuitar(Guitar guitar) {
        this.guitar = guitar;
    }

    public void setSpeaker(Speaker speaker) {
        this.speaker = speaker;
    }

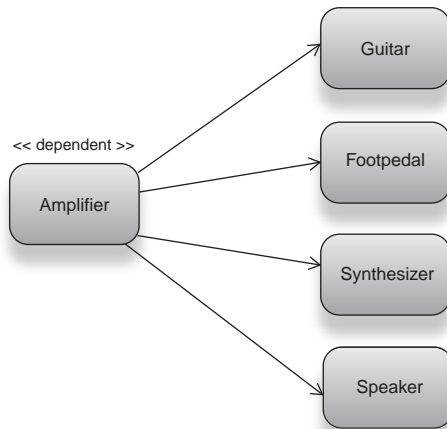
    public void setFootpedal(Footpedal footpedal) {
        this.footpedal = footpedal;
    }

    public void setSynthesizer(Synthesizer synthesizer) {
        this.synthesizer = synthesizer;
    }
}

```

Dependencies of a stereo amplifier

Setters for each dependency

**Figure 3.3 Class model of the amplifier and its four dependencies**

As with constructor injection, DI libraries diverge slightly in how they are configured to handle setter injection. In Spring parlance, a setter directly refers to a property on the object and is set via the `<property>` XML element, as in listing 3.2.

Listing 3.2 Spring XML configuration for setter injection of Amplifier

```

<beans ...>

    <bean id="amplifier" class="stereo.Amplifier">
        <property name="guitar" ref="guitar"/>
        <property name="speaker" ref="speaker"/>
        <property name="footpedal" ref="footpedal"/>
    
```



```

        <property name="synthesizer" ref="synthesizer"/>
    </bean>

    <bean id="guitar" class="equipment.Guitar"/>
    <bean id="speaker" class="equipment.Speaker"/>
    <bean id="footpedal" class="equipment.Footpedal"/>
    <bean id="synthesizer" class="equipment.Synthesizer"/>
</beans>

```

Here we have declared four `<property>` tags under bean `amplifier` that refer (via the `ref=".."` attribute) to dependencies of `Amplifier`. Spring knows which setter method we are talking about by matching the `name=".."` attribute of the `<property>` tag to the setter method's name (minus the `set` prefix). So, `name="guitar"` refers to `setGuitar()`, `name="speaker"` to `setSpeaker()`, and so on. In listing 3.3, we can use encapsulation for a more compact configuration.

Listing 3.3 The same Spring configuration, now with encapsulation

```

<beans ...>
    <bean id="amplifier" class="stereo.Amplifier">
        <property name="guitar">
            <b><bean class="equipment.Guitar"/></b>
        </property>
        <property name="speaker">
            <b><bean class="equipment.Speaker"/></b>
        </property>
        <property name="footpedal">
            <b><bean class="equipment.Footpedal"/></b>
        </property>
        <property name="synthesizer">
            <b><bean class="equipment.Synthesizer"/></b>
        </property>
    </bean>
</beans>

```

The elements in bold are encapsulated within `<property>` declarations. Nothing changes with the code itself. Dependencies are created and wired to the graph when the key `amplifier` is requested from Spring's injector.

NOTE The order in which these setters are called usually matches the order of the `<property>` tags. This ordering is unique to the XML configuration mechanism and is not available with autowiring.

As an alternative, listing 3.4 shows what the Guice version of `Amplifier` would look like.

Listing 3.4 A stereo amplifier wired by setter (using Guice)

```

import com.google.inject.Inject;

public class Amplifier {
    private Guitar guitar;

```

```

private Speaker speaker;
private Footpedal footpedal;
private Synthesizer synthesizer;

@Inject
public void setGuitar(Guitar guitar) {
    this.guitar = guitar;
}

@Inject
public void setSpeaker(Speaker speaker) {
    this.speaker = speaker;
}

@Inject
public void setFootpedal(Footpedal footpedal) {
    this.footpedal = footpedal;
}

@Inject
public void setSynthesizer(Synthesizer synthesizer) {
    this.synthesizer = synthesizer;
}
}

```

The main difference is annotating each setter with `@Inject`.

NOTE The order in which these setters are called is undefined, unlike with Spring's XML configuration.

But one nice thing is that you do not need the setter naming convention in Guice, so you can rewrite this class in a more compact way. Listing 3.5 shows that I have replaced the four separate setters with just one taking four arguments.

Listing 3.5 Compact setter injection (in Java, using Guice)

```

public class Amplifier {
    private Guitar guitar;
    private Speaker speaker;
    private Footpedal footpedal;
    private Synthesizer synthesizer;

    @Inject
    public void set(Guitar guitar, Speaker speaker, Footpedal footpedal,
        Synthesizer synthesizer) {
        this.guitar = guitar;
        this.speaker = speaker;
        this.footpedal = footpedal;
        this.synthesizer = synthesizer;
    }
}

```

Notice that this new setter looks very much like a constructor:

- It returns nothing.
- It accepts a bunch of dependencies.
- It is called only once when the object is created by the injector.

Guice does not place any restrictions on the name or visibility of setters either. So you can hide them from accidental misuse and even give them meaningful names:

```
@Inject
void prepareAmp(Guitar guitar, Speaker speaker, Footpedal footpedal,
                Synthesizer synthesizer) {
    this.guitar = guitar;
    this.speaker = speaker;
    this.footpedal = footpedal;
    this.synthesizer = synthesizer;
}
```

Much nicer, indeed.

Setter injection is easily the most common idiom in use. Many examples and tutorials use setter injection without explaining it or saying why it ought to be used. In fact, setter injection is quite a flexible option and can be very convenient in particular cases. It certainly has a place in the DI spectrum. Yet, it is not an entirely satisfactory situation. A method whose sole purpose is accepting a dependency seems awkward. Worse still, if one is required for each dependency (as in Spring), the number of setters can quickly get out of hand.

This problem is mitigated by the fact that setters are rarely exposed to client code (via interfaces or public classes, for example). Hiding setters makes them less of a danger to abuse. If they are only used by the injector (or by unit tests), there is less risk of objects becoming intertwined.

There are quite a few subtleties (some of the sledgehammer variety!) to deciding when setter injection is appropriate. We will look at them in some detail very shortly. But before that we'll look at some other forms of injection that are less common. Interface injection, in the following section, predates the more-familiar forms. Though it has fallen out of use of late, it's well worth our time to investigate this design idiom.

3.1.3 Interface Injection

Interface injection is the idea that an object takes on an *injectable* role itself. In other words, objects receive their dependencies via methods declared in interfaces. In a sense, interface injection is the same as setter injection except that each setter is housed in its own interface. Here's a quick example (listing 3.6).

Listing 3.6 Phaser mounted on a starship via interface wiring

```
public class Starship implements PhaserMountable {
    private Phaser phaser;

    public void mount(Phaser phaser) {
        this.phaser = phaser;
    }
}

public interface PhaserMountable {
    void mount(Phaser phaser);
}
```

This method is called by the injector

Interface solely for wiring Phasers to dependents

Starship is a kind of PhaserMountable (it *implements* PhaserMountable), which is an interface we've made up solely for the purpose of wiring phasers and starships together. If you wanted to add more dependencies to a starship, you would similarly have to create a *role interface* for each such dependency. Listing 3.7 shows how adding just two more dependencies, a *reactor* and *shuttlecraft*, alters our code.

Listing 3.7 Various dependencies of a starship via interface wiring

```
public class Starship implements PhaserMountable, ReactorMountable,
ShuttleDockable {
    private Phaser phaser;
    private Reactor reactor;
    private Shuttle shuttle;

    public void mount(Phaser phaser) {
        this.phaser = phaser;
    }

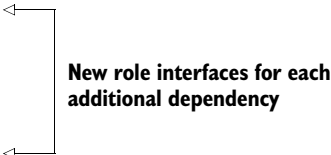
    public void mount(Reactor reactor) {
        this.reactor = reactor;
    }

    public void dock(Shuttle shuttle) {
        this.shuttle = shuttle;
    }
}

public interface PhaserMountable {
    void mount(Phaser phaser);
}

public interface ReactorMountable {
    void mount(Reactor reactor);
}

public interface ShuttleDockable {
    void dock(Shuttle shuttle);
}
```




The way in which these role interfaces are used by the injector is not a whole lot different from what we saw in some of the other idioms. The now-defunct Apache Avalon framework is the only real exponent of interface injection worth mentioning.

Role interfaces can be used not only to wire dependencies to objects but also to notify them of particular events, such as *initialize*, *pause*, *destroy*, and so forth. It is not uncommon for objects managed by interface injection to expose a battery of such interfaces. Listing 3.8 shows a rocket ship that exposes many such roles.

Listing 3.8 A rocket ship with many role interfaces

```
public class Rocket implements EngineMountable, Loadable, Startable,
Stoppable {
    private Engine engine;
    private Cargo cargo;

    public void mount(Engine engine) {
```



```

        this.engine = engine;
    }
    public void load(Cargo cargo) {
        this.cargo = cargo;
    }

    public void start() {
        //start this rocket's engine...
    }

    public void stop() {
        //stop this rocket...
    }
}

```

Interface injection methods

Methods for other infrastructure roles

One advantage of interface injection is that you can name the methods-accepting dependencies whatever you want. This gives you the opportunity to use natural, purpose-specific names that are concise and self-explanatory. The method `load(Cargo)` is more intuitive than, say, `setCargo(Cargo)`.

The disadvantages of interface injection are quite obvious. It is extremely verbose. Creating a new interface for each dependency is rather wasteful, and the long string of implemented interfaces following the class definition isn't exactly pretty. These interfaces clutter the class signature and distract attention from other interfaces you may be more interested in. Practically speaking, the fact that the defunct Apache Avalon is the only DI library that offers interface injection also makes it a less-than-attractive option. It is largely an idiom that has gone out of style for these reasons.

As we will see later, some of the ideas behind interface injection form the basis of important design patterns and solutions to several knotty design problems. Another lesser-known injection idiom is *method decoration*, sometimes called *AOP injection*. This form is useful in certain very specific cases and is naturally not very common. However, it may be indispensable for solving particular types of problems, as you will see in the following section.

3.1.4 Method decoration (or AOP injection)

Method decoration is an interesting variation on dependency injection. Method decoration takes the approach that methods rather than objects are the target of injection. This generally means that when they are called, these methods return an injector-provided value instead of their normal value. This requires a fundamental redefinition of a method's behavior and is generally done via some form of *interception*. More often than not, an AOP framework supplies the means of interception. We will look in much greater detail at interception and AOP in chapter 8.

This process is called *decorating* the method and gets its name from the Decorator¹ design pattern.

¹ Decorator design pattern, from *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma et al. (Addison-Wesley Professional Computing Series, 1994). Sometimes called the “Gang of Four” book.

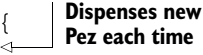
It is useful if you want to make a Factory out of an *arbitrary method* on your object. The use cases for this are somewhat rare. But it can be a very powerful design pattern if applied correctly. Here's an example:

```
package candy;

public class Dispenser {

    public Pez dispense() {
        return ...;
    }
}

public class Pez { .. }
```

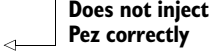


Dispenses new Pez each time

What we're after is for the `dispense()` method to be a Factory for Pez. Since we would like Pez to be created and wired by the injector, it isn't enough just to manually construct and return an instance:

```
public class Dispenser {

    public Pez dispense() {
        return new Pez(..);
    }
}
```




Does not inject Pez correctly

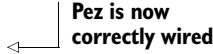
Clearly we need to bring the injector in, so Pez's dependencies can be correctly wired. One solution is to inject dependencies of Pez into Dispenser itself, then wire them up manually each time `dispense()` is called:

```
public class Dispenser {
    private Sugar sugar;

    public Pez dispense() {
        return new Pez(sugar);
    }
}
```



Provided by injection



Pez is now correctly wired

This solution looks like it works (at least it compiles and runs), but it doesn't quite give us what we're after:

- The same instance of Sugar is wired to every new Pez, so we have just moved the problem away one level, not solved it. What we want is *new* Sugar for *new* Pez.
- Dispenser must know how to construct Pez and consequently is *tightly coupled* to its internals.
- Dispenser is unnecessarily cluttered with Pez's dependencies.

All this looks like we're not putting the injector to full use. So how do we get there? Method decoration provides a compelling solution. To explain how this works, let's rewind to the original example:

```
package candy;

public class Dispenser {

    public Pez dispense() {
```

```

        return ...;
    }
}

```

What goes here?

```

public class Pez { .. }

```

Now, in order to make this class legal, we will write a *dummy* implementation of `dispense()`. This implementation does nothing and returns `null` with the understanding that it should never be called or used directly:

```

public Pez dispense() {
    return null;
}

```

Now we can proceed to configuring the injector. In listing 3.9 I use Spring to demonstrate.

Listing 3.9 Pez and Dispenser configuration (using Spring), `pez.xml`

```

<beans ...>
    <bean id="pez" class="candy.Pez">
        <constructor-arg><bean class="candy.Sugar" /></constructor-arg>
    </bean>

    <bean id="dispenser" class="candy.Dispenser"/>
</beans>

```

In listing 3.10, I have configured the injector to construct, assemble, and provide instances of `Pez` and `Dispenser`. However, we're not quite there yet.

Listing 3.10 Pez dispensed with method decoration (using Spring), `pez.xml`

```

<beans ...>
    <bean id="pez" class="candy.Pez">
        <constructor-arg><bean class="candy.Sugar" /></constructor-arg>
    </bean>

    <bean id="dispenser" class="candy.Dispenser">
        <lookup-method name="dispense" bean="pez"/>
    </bean>
</beans>

```

In listing 3.10, the `<lookup-method>` tag tells Spring to intercept `dispense()` and treat it as a `Factory` and that it should return object graphs bound to key `pez` when called. Now when we bootstrap and use the injector, we can use `dispense()` to get some candy!

```

BeanFactory injector = new FileSystemXmlApplicationContext("pez.xml");
Dispenser dispenser = (Dispenser) injector.getBean("dispenser");
Pez candy1 = dispenser.dispense();
...

```

For a more detailed look at where method decoration is useful, browse down to the “The ReInjection Problem.” Like any powerful DI feature, method decoration is fraught with pitfalls and corner cases. Some of them, particularly regarding method

interception, are examined in chapter 8. Carefully consider all of these issues before settling on method decoration as your choice of injection idiom.

Field injection

Guice provides the rather obvious but often-overlooked facility to wire dependencies *directly* into fields of objects (bypassing constructors or setters). This is often useful in writing small test cases or scratch code, and it is particularly so in tutorial and example code where space is limited and meaning must be conveyed in the least-possible number of lines. Consider this rather trivial example:

```
public class BrewingVat {
    @Inject Barley barley;
    @Inject Yeast yeast;

    public Beer brew() {
        //make some beer from ingredients
        ...
    }
}
```

Annotating fields `barley` and `yeast` with `@Inject` tells the injector to wire dependencies directly to them when `BrewingVat` is requested. This happens after the class's constructor has completed and before any annotated setters are called for setter injection. This syntax is compact and easy to read at a glance. However, it is fraught with problems when you consider anything more than the simplest of cases.

Without the ability to set dependencies (whether mocked or real) for testing, unit tests cannot be relied on to indicate the validity of code. It is also not possible to declare field-injected fields immutable, since they are set post construction. So, while field injection is nice and compact (and often good for examples), it has little worth in practice.

With so many options, how do you know what the right choice is?

3.2 Choosing an injection idiom

Well, interface injection is largely unsupported, and we've seen that it is rather verbose and unwieldy, so we can rule it out. Method decoration seems to be something of a corner-case solution, though I maintain that it has its place. That leaves us with constructor and setter injection, which are by far the most dominant forms of the idiom in use today.

The answer to the question of which idiom is the right choice is not simply one of taste, as I have said before. This is a point I can't emphasize too much—there are such important consequences to either choice that potentially lead to difficult, underperformant, and even broken applications. Some of this has to do with objects used in multithreaded environments. Some of it is about testing and maintainability. So you must take special care before choosing an injection idiom. It is well worth the time

and effort up front, to avoid intractable problems later. The rest of this chapter is about such problems and how to go about solving them.

3.2.1 Constructor vs. setter injection

The merits and demerits of constructor and setter injection are the same as those for setting *fields* by constructor or setter.

An important practice regards the *state* of fields once set. In most cases we want fields to be set once (at the time the object is created) and never modified again. This means not only that the dependent can rely on its dependencies throughout its life but also that they are ready for use right away. Such fields are said to be *immutable*. If you think of private member variables as a *spatial* form of encapsulation, then immutability is a form of *temporal encapsulation*, that is, unchanging with respect to time. With respect to dependency injection, this can be thought of as *freezing* a completely formed object graph.

Constructor injection affords us the ability to create immutable dependencies by declaring fields as final (modeled in figure 3.4):

```
public class Atlas {
    private final Earth earthOnBack;

    public Atlas(Earth earth) {
        this.earthOnBack = earth;
    }
}
```

Atlas stands today as he did at the time of his (and the earth's) creation, with a fixed, *immutable* earthOnBack. Any attempt, whether accidental or deliberate, to alter field earthOnBack will simply not compile (see listing 3.11). This is a very useful feature because it gives us a strong assurance about the state of an object and means that an object is a *good citizen*.

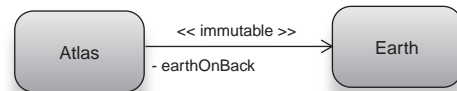


Figure 3.4 Atlas depends on Earth and cannot be changed once wired.

Listing 3.11 An object with immutable fields cannot be modified, once set

```
public class Atlas {
    private final Earth earth;

    public Atlas(Earth earth) {
        this.earth = earth;
        this.earth = null;
    }

    public void reset() {
        earth = new Earth();
    }
}
```

↖
 Raises a
 compile error
 ↗

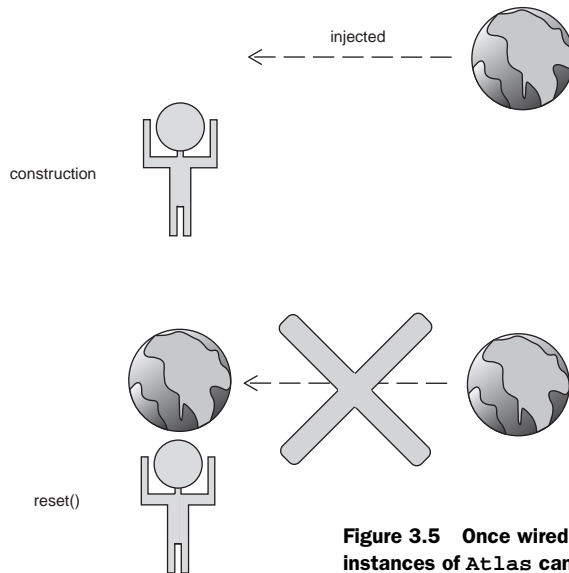


Figure 3.5 Once wired with an Earth, instances of `Atlas` cannot be wired again.

Visualize listing 3.11’s mutability problem in figure 3.5.

Immutability is essential in application programming. Unfortunately, it is not available using setter injection. Because setter methods are no different from ordinary methods, a compiler can make no guarantees about the *one-call-per-instance* restriction that is needed to ensure field immutability. So score one for constructor injection; it leverages immutability while setter injection can’t.

What else can we say about their differences? Plenty: dependencies wired by constructor mean they are wired at the time of the object’s creation. The fact that all dependences are wired and that the dependent is ready for use immediately upon construction is a very compelling one. It leads us to the notion of creating *valid* objects that are good citizens. Were a dependency not available (either in a unit test or due to faulty configuration), an early compile error would result. No field can refer to *half-constructed* dependencies. Moreover, a constructor-injected object cannot be created unless *all* dependencies are available. With setter injection such mistakes are easy to make, as shown in listing 3.12 and illustrated in figure 3.6.

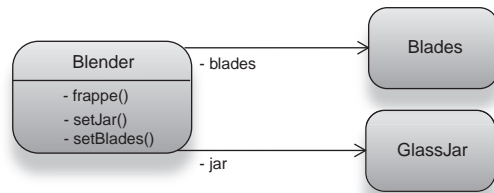


Figure 3.6 Blender is setter injected with `Blades` and `GlassJar` (see listing 3.12).

Listing 3.12 Improperly setter-injected object graph (using Guice)

```
public class Blender {
    private GlassJar jar;    ← Wired correctly
    private Blades blades;
```

```

@Inject
public void setJar(GlassJar jar) {
    this.jar = jar;
}

public void setBlades(Blades blades) {    ← Oops, missing an @Inject
    this.blades = blades;
}

public void frappe() {
    jar.fillWithOranges();
    blades.run();    ← Exception raised
}
}

```

A faulty configuration causes much heartache, since it compiles properly and everything *appears* normal. The program in listing 3.12 runs fine until it hits `blades.run()`, which fails by raising a `NullPointerException` since there is no object referenced by field `blades` because it was never set by the injector. Figure 3.7 illustrates the problem.

So let's say you found and fixed the injector configuration (after a bit of pain). Does everything work properly now? No, because you could still encounter this problem in a unit test where dependencies are set manually (without an injector present), as this foolish test of the Blender from listing 3.12 does:

```

public class BlenderTest {

    @Test
    public void blendOranges() {
        new Blender().frappe();

        //assert something
    }
}

```

The writer of this test has forgotten to call the appropriate setter methods and prepare the Blender correctly before using it. It results in a false negative; that is, an unexpected `NullPointerException` is raised, indicating a test failure when nothing is wrong with Blender's code as such. The reported error is caught only at runtime and isn't particularly intuitive in defining the problem. Worse, you may need to read through a fair thicket of execution traces before discovering the cause of the problem. This exists only with setter injection, since any test with unset dependencies in a constructor won't compile. Score two for constructor injection.

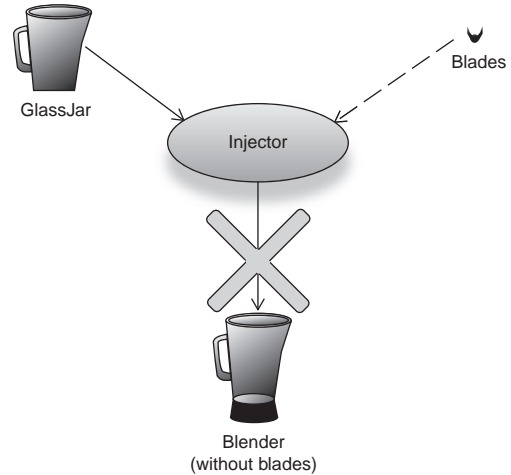


Figure 3.7 Blender is incompletely wired, an inadvertent side effect of setter injection.

Another problem with setters, as we have already seen, is that their number can very quickly get out of hand. If you have several dependencies, a setter for each dependency can result in a cluttered class with an enormous amount of repetitive code.

On the other hand, the explicitness of setter injection can itself be an advantage. Constructors that take several arguments are difficult to read. Multiple arguments of the same type can also be confusing since the only thing distinguishing them is the order in which they appear, particularly so if you use an IDE like IntelliJ IDEA to generate constructors automatically. For instance, consider the following class wired by constructor:

```
public class Spy {
    private String realName;
    private String britishAlias;
    private String americanAlias;

    public Spy(String name1, String name2, String name3, ...) { .. }
}
```

It is rather difficult to follow how to wire this object. The smallest spelling or ordering mistake could result in an erroneous object graph that goes *completely* undetected (since everything is a `String`). This is a particularly good example because there is no easy way to validate that dependencies have not gotten crossed:

```
new Spy("John Doe", "James Bond", "Don Joe");
```

Everything looks all right, but my `Spy`'s British alias was really Don Joe and not the other way around. With setter injection, however, this is clear and explicit (the following example is in Java using setters):

```
Spy spy = new Spy();
spy.setRealName("John Doe");
spy.setBritishAlias("James Bond");
spy.setAmericanAlias("Don Joe");
```

So when you have large numbers of dependencies of the same type, setter injection seems more palatable. This is particularly true for unit tests, which are an important companion to any service and which inject dependencies manually. You'll find more on unit testing and dependency injection in chapter 4. Chalk up a victory for setter injection. One additional problem with constructor injection is that when you have different object graph permutations, the number of constructors can get out of hand very quickly. This is called the *constructor pyramid* problem, and the following section takes an in-depth look at it.

3.2.2 The constructor pyramid problem

Constructor injection also runs into trouble when dealing with objects that are injected differently in different scenarios. Each of these scenarios is like a different *profile* of the object. Listing 3.13 shows the example of an amphibian that has different dependencies when on land than on water.

Listing 3.13 An amphibian can live on land or in water

```

public class Amphibian {
    private Gills gills;
    private Lungs lungs;
    private Heart heart;

    public Amphibian(Heart heart, Gills gills) {
        this.heart = heart;
        this.gills = gills;
    }

    public Amphibian(Heart heart, Lungs lungs) {
        this.heart = heart;
        this.lungs = lungs;
    }

    ...
}

```

Constructor for
life in water
 Constructor for
life on land

When we want a water Amphibian, we construct one with a Heart and Gills. On land, this same Amphibian has a different set of dependencies, mutually exclusive with the water variety (a pair of Lungs). Heart is a common dependency of both Amphibian profiles. The two profiles are modeled in figures 3.8 and 3.9.

Ignore for a moment that real amphibians can transition from land to water. Here we require two *mutually exclusive* constructors that match either profile but not both (as listing 3.13 shows). Were you to require more such profiles, there would be more constructors, one for each case. All these constructors differ by is the type of argument they take; unlike setters, they can't have different names. This makes them hard to read and to distinguish from one another. Where an object requires only a partial set of dependencies, additional, smaller constructors need to be written, further adding to the confusion. This issue is called the constructor pyramid problem, because the collection of constructors resembles a rising pyramid. On the other hand, with setters, you can wire any permutation of an object's dependencies without writing any additional code. Score another for setter injection.

Having said all of this, however, the use cases where you require multiple profiles of an object are rare and probably found only in legacy or sealed third-party code. If you

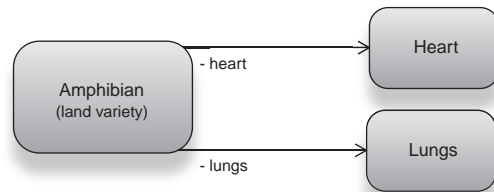


Figure 3.8 The land-dwelling variety of Amphibian depends on a Heart and Lungs.

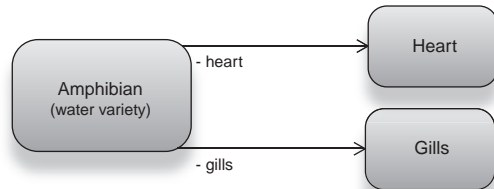


Figure 3.9 The aquatic variety of Amphibian depends on a Heart and Gills.

find yourself encountering the pyramid problem often, you should ask serious questions about your design before pronouncing setter injection as a mitigant. Another problem when using only constructors is how you connect objects that are dependent on the same instance of each other. This chicken-and-egg problem, though rare, is a serious flaw in constructor injection and is called the *circular reference* problem.

3.2.3 The circular reference problem

Sometimes you run into a case where two objects are dependent on each other. Figure 3.10 shows one such relationship.

Any *symbiotic* relationship between two components embodies this scenario. Parent/child relationships are typical manifestations. One example is shown in listing 3.14.

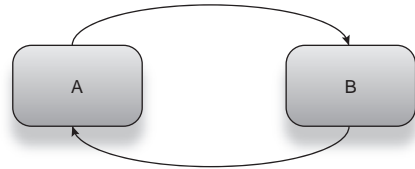


Figure 3.10 A and B are dependent on each other (circular dependency).

Listing 3.14 A host and symbiote interdependency

```

public class Host {
    private final Symbiote symbiote;

    public Host(Symbiote symbiote) {
        this.symbiote = symbiote;
    }
}

public class Symbiote {
    private final Host host;

    public Symbiote(Host host) {
        this.host = host;
    }
}
  
```

In listing 3.14 both `Host` and `Symbiote` refer to the same instances of each other. `Host` refers to `Symbiote`, which refers back to `Host` in a circle (as described in figure 3.11).

Syntactically, there is no conceivable way to construct these objects so that circularity is satisfied with constructor wiring alone. If you decide to construct `Host` first, it requires a `Symbiote` as dependency to be valid. So you are forced to construct `Symbiote` first; however, the same issue resides with `Symbiote`—#@\$! Its only constructor requires a `Host`.

This classic chicken-and-egg scenario is called the circular reference problem.

There is also no easy way to configure an injector to solve the circular reference problem, at least not with the patterns we've examined thus far. You can also imagine indirect circular references where object A refers to B, which refers to C, which itself refers back to A (as illustrated in figure 3.12).

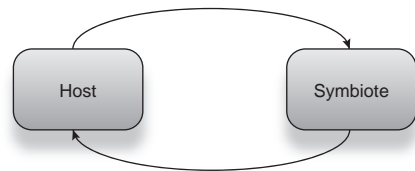


Figure 3.11 `Host` and `Symbiote` are circularly interdependent.

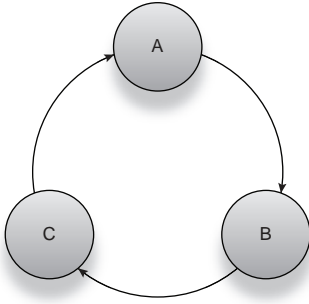


Figure 3.12 Triangular circularity:
A depends on B, which depends on C, which depends on A.

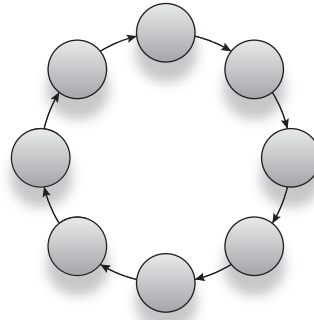


Figure 3.13 Any number of objects
may be part of a circular dependency.

In fact, this triangular flavor of circular references is the one more commonly seen. There is no reason why we should stop at three, either. Any number of objects can be added to the circle, and so long as two end points connect, there is no clear strategy for which object to instantiate first. It is the old problem of finding the *beginning* of a cycle (figure 3.13).

The circular reference problem has two compelling solutions that fall, like everything else, between the camps of setter and constructor injection. The first and most obvious solution is simply to switch to setter injection. For our purposes, I'll examine the two-object variety of the circular reference problem (though the same solution is equally applicable to other cases) in listing 3.15.

Listing 3.15 A host and symbiote interdependency with setter injection

```

package example;

public class Host {
    private Symbiote symbiote;

    public void setSymbiote(Symbiote symbiote) {
        this.symbiote = symbiote;
    }
}

public class Symbiote {
    private Host host;

    public void setHost(Host host) {
        this.host = host;
    }
}
  
```

Then, configuring the injector is straightforward (via setter injection, in Spring):

```

<beans ...>
  <bean id="host" class="example.Host">
    <property name="symbiote" ref="symbiote"/>
  </bean>
  
```

```

<bean id="symbiote" class="example.Symbiote">
  <property name="host" ref="host"/>
</bean>
</beans>

```

Now the circularity is satisfied. When the injector starts up, both `host` and `symbiote` object graphs are constructed via their *nullary* (zero-argument) constructors and then wired by setter to reference each other. Easy enough. Unfortunately there are a couple of serious drawbacks with this choice. The most obvious one is that we can no longer declare either dependency as `final`. So, this solution is less than ideal.

Now let's look at the constructor injection alternative. We already know that we can't directly use constructor wiring to make circular referents point to one another. What alternative is available under these restrictions? One solution that comes to mind is to break the circularity without affecting the *overall* semantic of interdependence. You achieve this by introducing a *proxy*. First let's decouple `Host` and `Symbiote` with interfaces (see listing 3.16).

Listing 3.16 A host and symbiote interdependency, decoupled to use interfaces

```

public interface Host { .. }
public interface Symbiote { .. }
public class HostImpl implements Host {
  private final Symbiote symbiote;

  public HostImpl(Symbiote symbiote) {
    this.symbiote = symbiote;
  }
}
public class SymbioteImpl implements Symbiote {
  private final Host host;

  public SymbioteImpl(Host host) {
    this.host = host;
  }
}

```

HostImpl refers to interface Symbiote

SymbioteImpl refers to interface Host

The class diagram now looks like figure 3.14.

The dependencies of `HostImpl` and `SymbioteImpl` are now on a contract (interface) of each other, rather than a *concrete class*. This is where the proxy comes in:

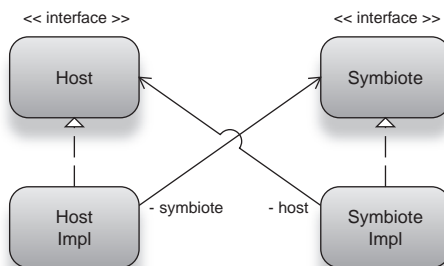


Figure 3.14 A class model for decoupled `Host` and `Symbiote` services (see listing 3.16)


```
public class HostProxy implements Host {
    private Host delegate;

    public void setDelegate(Host delegate) {
        this.delegate = delegate;
    }
}
```

HostProxy wired
with a Host

HostProxy, the intermediary, is wired with setter injection allowing HostImpl and SymbioteImpl (the “real” implementations) to declare their fields immutable and use constructor injection. To complete the wiring now, we use the following construction order:

- 1 Construct HostProxy.
- 2 Construct SymbioteImpl with the instance of HostProxy (from step 1).
- 3 Construct HostImpl and wire it with the SymbioteImpl that refers to HostProxy (from step 2).
- 4 Wire HostProxy to *delegate* HostImpl (from step 3) via setter injection.

Did that do it for us? Indeed—the Host instance now refers to a Symbiote instance that itself holds a reference to the HostProxy. When the SymbioteImpl instance calls any methods on its dependency host, these calls go to HostProxy, which transparently passes them to its delegate, HostImpl. Since HostImpl contains a direct reference to its counterpart SymbioteImpl, the circularity is satisfied, and all is well with the world. Injector configuration for this is shown in listing 3.17.

Listing 3.17 Circular references wired via a proxy (using Spring)

```
<beans>
  <bean id="hostProxy" class="example.HostProxy">
    <property name="delegate" ref="host" />
  </bean>

  <bean id="host" class="example.HostImpl">
    <constructor-arg ref="symbiote" />
  </bean>

  <bean id="symbiote" class="example.SymbioteImpl">
    <constructor-arg ref="hostProxy" />
  </bean>
</beans>
```

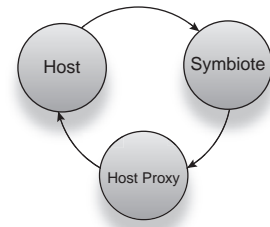


Figure 3.15 illustrates the solution.

Guice offers a transparent mitigation of this problem (illustrated in figure 3.16). All you need do is configure the injector and bind the relevant keys to one another:

```
public class CircularModule extends AbstractModule {
    @Override
    public void configure() {
        bind(Host.class).to(HostImpl.class).in(Singleton.class);
        bind(Symbiote.class).to(SymbioteImpl.class).in(Singleton.class);
    }
}
```

Figure 3.15 Injecting circular referents via a proxy

The Guice injector automatically provides the proxy in the middle. We can also trust the injector to work out the correct order of construction. We don't deal directly with proxies or setter injection or any other infrastructure concern. If you are like me, you find this feature of Guice particularly useful where circular references are warranted. A particular variant of the circular reference problem is the *in-construction* problem. In this scenario, it's impossible to break the cycle even with a proxy. We examine the in-construction problem in the following section.

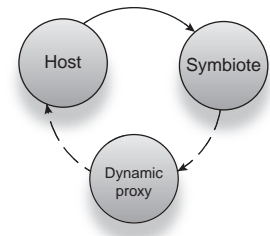


Figure 3.16 Injecting circular referents in Guice with a dynamic proxy

3.2.4 The in-construction problem

Earlier, when describing constructor injection, we said that one of the purposes of a constructor was to “perform some initialization logic” on an object. This may be as simple as setting dependencies on fields or may involve some specific computation needed in order to prepare the object (for example, setting a window's title). The constructor is a good place to put this because you can guarantee that any such logic is run prior to the object being used anywhere. If this computation is expensive, it is worthwhile performing it up front so you don't encumber the object later on.

One close relative of the circular reference problem (section 3.2.3) is where initialization logic requires a dependency that is not yet ready for use. What this typically means is that a circular reference was solved with a proxy, which has not yet been wired with its delegate (recall the four steps to resolving circular references with a proxy, from the previous section). Listing 3.18 illustrates this scenario.

Listing 3.18 Proxy solution inadequate for in-constructor use (using Guice)

```

public interface Host { .. }

public interface Symbiote { .. }

public class HostImpl implements Host {
    private final Symbiote symbiote;

    @Inject
    public HostImpl(Symbiote symbiote) {
        this.symbiote = symbiote;
    }

    public int calculateAge() {
        // figure out age
    }
}

public class SymbioteImpl implements Symbiote {
    private final Host host;
    private final int hostAge;

    @Inject
    public SymbioteImpl(Host host) {

```

Derived property
computed on initialization



```

        this.host = host;
        this.hostAge = host.calculateAge();
    }
}

```

Computation
will fail

In listing 3.18, both dependents in the circle attempt to use each other within their constructors. One calls a method on the other and is doomed to fail. Since `Host` has been proxied but does not yet hold a valid reference to its delegate, it has no way of passing through calls to the dependency. This is the *in-construction* problem in a nutshell.

Circular references in nature (symbiosis)

Symbiotes in nature are organisms that live in biological dependence on one another. This is a more closely linked relationship than a parent and child. Symbiotes typically cannot survive without their partners. Here are some interesting examples of circular interdependence in nature:

Coral reefs are a symbiosis between coral and various algae that live communally. The algae photosynthesize and excrete food for the coral, which in turn protect and provide a place to live for the algae.

Siboglinid tube worms have no digestive tract. Instead, bacteria that live inside them break down their food into nutrients. In turn, the worms provide the bacteria with a home and a constant supply of food. These fascinating worms were discovered living in hydrothermal vents in the ground.

Several herbivorous animals (plant eaters) have gut fauna living in their stomachs, which help break down plant matter. Plant matter is inherently more difficult to digest because of the thick cell walls plants possess. Like the tube worms, these animals house and provide food for their symbiotes.

The in-construction problem has no obvious solution with constructor injection. The only recourse in such cases is setter injection, as shown in listing 3.19.

Listing 3.19 In-construction problem solved via setter injection (using Guice)

```

public interface Host { .. }

public interface Symbiote { .. }

public class HostImpl implements Host {
    private Symbiote symbiote;

    @Inject
    public void setSymbiote(Symbiote symbiote) {
        this.symbiote = symbiote;
    }

    public int calculateAge() {
        // figure out age
    }
}

```

```

}

public class SymbioteImpl implements Symbiote {
    private Host host;
    private int hostAge;

    @Inject
    public void setHost(Host host) {
        this.host = host;
        this.hostAge = host.calculateAge();
    }
}

```

Initialization logic now works

Now initialization is guaranteed to succeed because both objects hold references to each other prior to any call to `host.calculateAge()`. This is not the whole story—the in-construction problem can run even deeper. Consider listing 3.20’s addendum, which exacerbates the problem.

Listing 3.20 In-construction problem not quite solved

```

public interface Host { .. }

public interface Symbiote { .. }

public class HostImpl implements Host {
    private Symbiote symbiote;
    private int symbioteAge;

    @Inject
    public void setSymbiote(Symbiote symbiote) {
        this.symbiote = symbiote;
        this.symbioteAge = symbiote.calculateAge();
    }
    ...
}

public class SymbioteImpl implements Symbiote {
    private Host host;
    private int hostAge;

    @Inject
    public void setHost(Host host) {
        this.host = host;
        this.hostAge = host.calculateAge();
    }
    ...
}

```

New initialization logic

Now both objects not only *refer* to one another but also *use* each other when dependencies are being wired. Essentially, this is an in-construction *variation* of the circular reference problem. And there is no known solution to this circular initialization mess with constructor injection. In fact, there is no solution to the problem with setter injection either!

More important, this is not really a wiring problem. The issue is with putting objects into a usable *state*. While constructors are the traditional pack mule for this sort of work, they don't quite work in this case. If you find yourself encountering this problem, you are probably better off using lifecycle as a solution. See chapter 7 for an exploration of object lifecycle. Thus far, we've outlined many of the problems with using constructor injection. While many of them are rare, they are nonetheless troubling. Now let's take a look at some of the benefits that really make constructor injection worthwhile.

3.2.5 Constructor injection and object validity

So far, we've talked about several benefits of constructor injection. None are perhaps as significant as *object validity*. Knowing if an object is properly constructed is one of the most overlooked design issues in OOP. Simply having a reference to it is insufficient. Dependencies may not be set, initialization logic may not have run yet, and so on. Even if all this has been achieved in one thread, other participating threads may not agree.

Consider the class in listing 3.21.

Listing 3.21 A class allowing too many unsafe modes of injection

```
public class UnsafeObject {
    private Slippery slippery;
    private Shady shady;

    public UnsafeObject() { }

    public void setSlippery(Slippery slippery) {
        this.slippery = slippery;
    }

    public void setShady(Shady shady) {
        this.shady = shady;
    }

    public void init() { .. }
}
```

Dependencies wired by setter
Nullary constructor does nothing
Initializer method

Looking over `UnsafeObject`, it's obvious that dependencies are wired by setter injection. But we're not quite sure why. It has forced both dependencies `slippery` and `shady` to be non-final, that is, *mutable*. And it looks like `init()` is expected to be called at some stage before the `UnsafeObject` is ready to perform duties. This object is unsafe, because there are too many ways to construct it incorrectly. Look at the following abusive injector configurations that are permitted by it:

```
<beans ...>
    <bean id="slippery" class="Slippery"/>
    <bean id="shady" class="Shady"/>

    <bean id="unsafel" class="UnsafeObject"/>
</beans>
```

We have forgotten to set any of its dependencies. No complaints even after the object is constructed and injected. It falls over in a big heap the first time it is used.

```
<beans ...>
<bean id="slippery" class="Slippery"/>
<bean id="shady" class="Shady"/>

<bean id="unsafe2" class="UnsafeObject">
  <property name="slippery" ref="slippery"/>
</bean>
</beans>
```

There are still no complaints, and everything will work until dependency shady is used. This is potentially riskier (since code using slippery will work normally).

```
<beans ...>
<bean id="slippery" class="Slippery"/>
<bean id="shady" class="Shady"/>

<bean id="unsafe4" class="UnsafeObject">
  <property name="slippery" ref="slippery"/>
  <property name="shady" ref="shady"/>
</bean>
</beans>
```

At first glance, this looks okay. Why is it labeled unsafe? Recall that the `init()` method must be called as part of the object's graduation to readiness. This is among the worst of all cases because no obvious error may come up. Several programmers' days have been wasted over this problem, trivial though it seems at first glance.

```
<beans ...>
<bean id="slippery" class="Slippery"/>
<bean id="shady" class="Shady"/>

<bean id="unsafe5" class="UnsafeObject" init-method="init">
  <property name="slippery" ref="slippery"/>
  <property name="shady" ref="shady"/>
</bean>
</beans>
```

Finally, a configuration you can rely on! All dependencies are accounted for and the initialization hook is configured correctly. Or is it? Has everything been accounted for? Recall that objects in Spring are singletons by default. This means that other multiple threads are concurrently accessing `unsafe5`. While the injector is itself safe to concurrent accesses of this sort, `UnsafeObject` may not be. The Java Memory Model makes no guarantees about the visibility of non-final, non-volatile fields across threads. To fix this, you can remove the concept of shared visibility, so an instance is exposed to only a single thread:

```
<beans ...>
<bean id="slippery" class="Slippery" scope="prototype"/>
<bean id="shady" class="Shady" scope="prototype"/>

<bean id="safe" class="UnsafeObject" init-method="init" scope="prototype">
  <property name="slippery" ref="slippery"/>
</bean>
```

```

        <property name="shady" ref="shady"/>
    </bean>
</beans>

```

The attribute `scope="prototype"` tells the injector to create a new instance of `UnsafeObject` each time it is needed. Now there are no concurrent accesses to `UnsafeObject`. Notice that you are forced to mark both dependencies with `scope="prototype"` as well. Otherwise they may be shared *underneath*.

Well! That was exhaustive. And it involved a lot of configuration just to account for an arbitrary design choice. With constructor injection these problems virtually disappear (as shown in listing 3.22).

Listing 3.22 A safe rewrite of `UnsafeObject` (see listing 3.21)

```

public class SafeObject {
    private final Slippery slippery;
    private final Shady shady;

    public SafeObject(Slippery slippery, Shady shady) {
        this.slippery = slippery;
        this.shady = shady;
        init();
    }

    private void init() { .. }
}

```

Dependencies wired
by constructor

Initializer runs
inside constructor

`SafeObject` is immune from all of the problems we encountered with its evil cousin. There is only one way to construct it, since it exposes only the one constructor. Both dependencies are declared `final`, making them safely published and thus guaranteed to be visible to all threads. Initialization is done at the end of the constructor. And `init()` is now private, so it can't accidentally be called after construction. Here's the corresponding configuration for `SafeObject` in Guice (it involves placing a single annotation on the class's constructor, as you can see in listing 3.23).

Listing 3.23 Safe object managed by Guice

```

public class SafeObject {
    private final Slippery slippery;
    private final Shady shady;

    @Inject
    public SafeObject(Slippery slippery, Shady shady) {
        this.slippery = slippery;
        this.shady = shady;
        init();
    }

    private void init() { .. }
}

```

Use this
constructor!

Constructor injection forces you to create valid, well-behaved objects. Moreover, it prevents you from creating invalid objects by raising errors early. I urge you to consider

constructor injection as your first preference, whenever possible. Now, let's look at a problem where some of the more esoteric idioms that we examined earlier in the chapter may prove useful. An example is the partial injection problem, where not all the dependencies are available up front.

3.3 Not all at once: partial injection

Sometimes you don't know about all the dependencies you need right away. Or you are given them later and need to build an object around them. In other cases, you may know about all the dependencies but may need to obtain new ones after each use (as we saw with the Pez dispenser in "Method decoration"). These object graphs can't be described precisely at startup. They need to be built dynamically, at the point of sale.

This introduces to us the idea of a *partial injection*. We'll shortly examine several scenarios where partial (or delayed) injection is useful and see how to go about achieving it. First, we'll look at a scenario where all dependencies are known at construction time but need to be reinjected on each use (because the instance is *used up* and needs to be replenished each time).

3.3.1 The reinjection problem

As the header suggests, this problem is about injecting an object that has already been injected once earlier. Reinjection is typical in cases where you have a *long-lived* dependent with *short-lived* dependencies. More specifically, reinjection is common where a dependency is put into an unusable state after it is used (that is, it has been depleted or used up in some way). You might have a data-holding object, and once it is saved with user input, you may need a new instance for fresh input. Or the dependency may be on a file on disk, which is closed (and disposed) after a single use. There are plenty of incarnations of this problem. Let's look at one purely illustrative case:

```
public class Granny {
    private Apple apple;

    public Granny(Apple apple) {
        this.apple = apple;
    }

    public void eat() {
        apple.consume();
        apple.consume();
    }
}
```

Causes an error as apple is already consumed

Granny is given one apple when she comes into existence. Upon being told to `eat()`, she consumes that apple but is still hungry. The apple cannot be consumed again since it is already gone. In other words, the *state* of the dependency has changed and it is no longer usable. Apple is short lived, while Granny is long lived, and on each use (every time she eats), she needs a new Apple.

There are a couple of solutions to this problem. You should recall one solution we applied to a similar problem earlier—the Pez dispenser. We could use method

decoration to achieve reinjection in this case. But we can't replace `eat()` because it performs a real business task. You could put another method on `Granny`. That would certainly work:

```
public class Granny {
    public Apple getApple() {
        return null;
    }

    public void eat() {
        getApple().consume();
        getApple().consume();
    }
}
```

Replaced with
method decoration

This is certainly a workable solution. However, it belies our original object graph (of `Granny` depending on an `Apple`) and more importantly is difficult to test. It also relies on a DI library that supports method decoration, and not all of them do. So what are other possible solutions? Common sense says we should be using some kind of Factory, one that can take advantage of DI. The Provider pattern is one such solution.

3.3.2 Reinjection with the Provider pattern

In a nutshell, a Provider pattern is a Factory that the injector creates, wires, and manages. It contains a single method, which *provides* new instances. It is particularly useful for solving the reinjection problem, since a dependent can be wired with a Provider rather than the dependency itself and can obtain instances from it as necessary:

```
public class Granny {
    public Provider<Apple> appleProvider;

    public void eat() {
        appleProvider.get().consume();
        appleProvider.get().consume();
    }
}
```

Provides
apples

Notice this line:

```
appleProvider.get().consume();
```

This effectively says that before we use an apple, we get it from the Provider, meaning that a new `Apple` instance is created each time. This architecture is shown in figure 3.17.

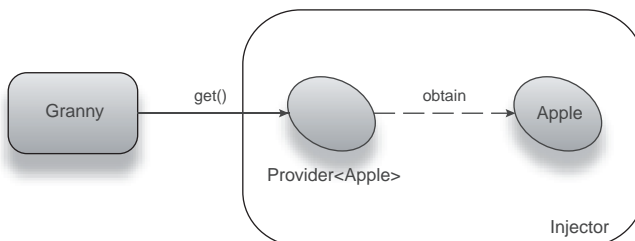


Figure 3.17 Granny obtains Apple instances from the injector via a provider

There are a couple of things we can say about Providers:

- A Provider is unique (and *specific*) to the type of object it provides. This means you don't do any downcasting on a `get()` as you might do with a Service Locator.
- A Provider's *only* purpose is to provide *scoped* instances of a dependency. Therefore it has just the one method, `get()`.

In Java, which has support for generics, a single provider is all that you need to write. Under this you are free to create as many implementations as needed for each purpose. Guice provides Providers out of the box for all bound types:

```
package com.google.inject;

public interface Provider<T> {
    public T get();
}
```

Method `get()` is fairly straightforward: it says get me an instance of `T`. This may or may not be a new instance depending on the scope of `T`. (For example, if the key is bound as a singleton, the same instance is returned every time.²) So, in Granny's case, all that's required is an `@Inject` annotation:

```
public class Granny {
    private Provider<Apple> appleProvider;

    @Inject
    public Granny(Provider<Apple> ap) {
        this.appleProvider = ap;
    }

    public void eat() {
        appleProvider.get().consume();
        appleProvider.get().consume();
    }
}
```

Provider is wired
via constructor

You don't need to do anything with the injector configuration because Guice is clever enough to work out how to give you an `Apple` provider. For libraries that don't support this kind of behavior out of the box, you can create a your own `Provider` that does the same trick. Here I've encapsulated a lookup from the injector within a `Provider` for `Spring`:

```
public class AppleProvider implements Provider<Apple>, BeanFactoryAware {
    private BeanFactory injector;

    public Apple get() {
        return (Apple) injector.getBean("apple");
    }

    public void setBeanFactory(BeanFactory injector) {
        this.injector = injector;
    }
}
```

Apple is looked up
from the injector

² See chapter 5 for more on scope.

There are two interesting things about `AppleProvider`:

- It exposes interface `Provider<Apple>`, meaning that it's in the business of providing apples.
- It exposes `BeanFactoryAware`.³ When it sees this interface, Spring will wire the injector itself to this provider.

The corresponding XML configuration is shown in listing 3.24.

Listing 3.24 Injector configuration for Granny and her Apple provider (in Spring)

```
<beans ...>
  <bean id="appleProvider" class="AppleProvider"/>

  <bean id="granny" class="Granny">
    <constructor-arg ref="appleProvider"/>
  </bean>
</beans>
```

Notice that I didn't have to do anything with `AppleProvider`. Spring automatically detects that it is an instance of `BeanFactoryAware` and wires it appropriately. Listing 3.25 shows a slightly more compact, encapsulated version.

Listing 3.25 Encapsulated version of listing 3.24 (in Spring)

```
<beans ...>
  <bean id="granny" class="Granny">
    <constructor-arg><bean class="AppleProvider"/></constructor-arg>
  </bean>
</beans>
```

What if you need to pass in an argument to the dependency? Something that's available only at the time of use, like data from a user, or some resource that acts as a *context* for the dependency. This is a variation on partial injection called the *contextual injection* problem.

3.3.3 The contextual injection problem

You could say that contextual injection and reinjection are related problems. You could even say that contextual injection is a special case of reinjection. These use cases come up quite often, particularly with applications that have some form of external user.

Consider an automated mailing list for a newsletter. Several users sign up with their email addresses, and you periodically send them a copy of your newsletter. The long-lived object in this case is triggered when it's time for an edition of the newsletter to go out, and its context is the newsletter. This is a case of partial injection—where the long-lived object (let's call it `NewsletterManager`) needs a new instance of a shorter-lived one (let's call this `Deliverer`) for every newsletter sent out. See figure 3.18 for a visual representation.

³ This is an example of interface injection (which we saw in chapter 2).

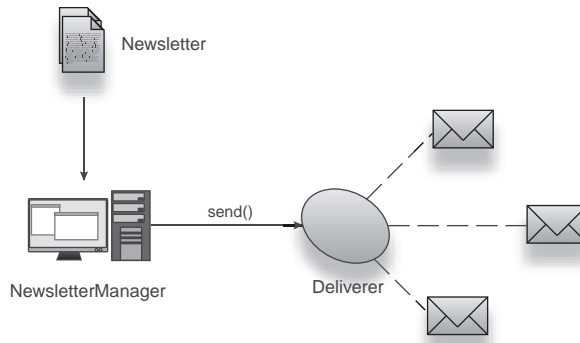


Figure 3.18 A Deliverer sends Newsletters out to recipients, on prompting by NewsletterManager

You could treat this purely as a reinjection problem and pass the newsletter directly to Deliverer on every use, say via a setter. Listing 3.26 shows how that might look.

Listing 3.26 A newsletter-sending component and contextual delivery agent

```

public class NewsletterManager {
    private final List<Recipient> recipients;
    private final Provider<Deliverer> deliverer;

    public NewsletterManager(List<Recipient> rs,
                             Provider<Deliverer> dp) {
        this.recipients = rs;
        this.deliverer = dp;
    }

    public void send(Newsletter letter) {
        for (Recipient recipient : recipients) {
            Deliverer d = deliverer.get();
            d.setLetter(letter);
            d.deliverTo(recipient);
        }
    }
}

public class Deliverer {
    private Newsletter letter;

    public void setLetter(Newsletter letter) {
        this.letter = letter;
    }

    ...
}
  
```

**Dependencies wired
by constructor**

**Set contextual
dependency**

In listing 3.26, I've exposed a setter and used a form of manual dependency injection to set the current Newsletter (context) on the Deliverer. This solution works, but it has the obvious drawbacks that go with using a setter method. What would be really nice is *contextual* constructor injection. And that's brings us to the Assisted Injection pattern.

3.3.4 Contextual injection with the Assisted Injection pattern

Since the contextual injection problem is a relative of reinjection, it follows that their solutions are also related. I used the Provider pattern to solve reinjection. Listing 3.27 reenvisions the newsletter system with a similar pattern: Assisted Injection. Its architecture is described in figure 3.19.

Listing 3.27 A newsletter manager and delivery agent, with Assisted Injection

```
public class NewsletterManager {
    private final List<Recipient> recipients;
    private final AssistedProvider<Deliverer, Newsletter>
        deliverer;

    public NewsletterManager(List<Recipient> rs,
        AssistedProvider<Deliverer, Newsletter > dp) {
        this.recipients = rs;
        this.deliverer = dp;
    }

    public void send(Newsletter letter) {
        for (Recipient recipient : recipients) {
            Deliverer d = deliverer.get(letter);
            d.deliverTo(recipient);
        }
    }
}

public interface AssistedProvider<T, C> {
    T get(C context);
}

public class DelivererProvider implements AssistedProvider<Deliverer,
    Newsletter> {
    public Deliverer get(Newsletter letter) {
        return new Deliverer(letter);
    }
}
```

Get dependency for context

Alternative to implementing Module

Construct by hand!

In listing 3.27, I have two classes of importance:

- NewsletterManager—The long-lived, dependent class
- DelivererProvider—The Assisted Injection provider, which provides Deliverer instances wired with a context (i.e., Newsletter)

The other interesting artifact in listing 3.27 is the interface AssistedProvider, which takes two type parameters:

- T refers to the generic type provided by a call to get().
- C refers to the generic type of the context object.

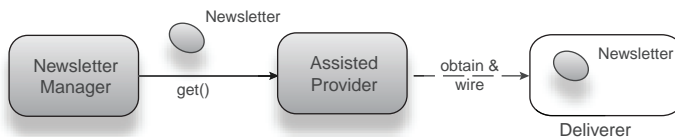


Figure 3.19
AssistedProvider creates a contextualized Deliverer with the current Newsletter.

They let you use `AssistedProvider` in other, similar areas. Like `Provider`, `AssistedProvider` is a piece of framework code that you can customize to suit your needs.

There's one obvious flaw with this plan. We've used construction by hand in the `AssistedProvider`. That's not ideal because we lose viral dependency injection, interception, scope, and so on. One way to fix it is by using another `Provider` inside the `AssistedProvider`. The trade-off is that we must use setter injection. This is actually a reasonable solution if used sparingly. Listing 3.28 demonstrates it with Guice.

Listing 3.28 A newsletter component, using Assisted Injection

```
public class DelivererProvider implements
    AssistedProvider<Deliverer, Newsletter> {
    private final Provider<Deliverer> deliverer;

    @Inject
    public DelivererProvider(Provider<Deliverer> deliverer) {
        this.deliverer = deliverer;
    }

    public Deliverer get(Newsletter letter) {
        Deliverer d = deliverer.get();
        d.setLetter(letter);
        return d;
    }
}
```

Provider is itself injected

Context set manually

In listing 3.28, we not only get the benefits of dependency injection but also can add the newsletter context to it.

Guice also provides an extension called `AssistedInject` that can help make this problem easier. `AssistedInject` lets us declare a Factory-style interface and creates the implementation for us, so we don't have to create the intermediary wiring code by hand. It is also compelling because this means we do not have to use setter injection and can take advantage of all the goodness of constructor injection:

```
import com.google.inject.assistedinject.Assisted;

public class Deliverer {
    private final Newsletter letter;

    @Inject
    public Deliverer(@Assisted Newsletter letter) {
        this.letter = letter;
    }

    ...
}
```

Tells Guice to assist this injection

The `@Assisted` annotation is an indicator to the generated factory that this constructor parameter should be obtained from a factory, rather than use the normal course of injection. The code from listing 3.28 now looks like this:

```
public class NewsletterManager {
    private final List<Recipient> recipients;
    private final DelivererFactory factory;

    public NewsletterManager(List<Recipient> rs,
```

Use an injected factory

```

        DelivererFactory factory) {
            this.recipients = rs;
            this.factory = factory;
        }

        public void send(Newsletter letter) {
            for (Recipient recipient : recipients) {
                Deliverer d = factory.forLetter(letter);
                d.deliverTo(recipient);
            }
        }
    }

    public interface DelivererFactory {
        Deliverer forLetter(Newsletter letter);
    }

```

Get dependency for context

Factory backed by Guice

Then in our module, we tell Guice about the DelivererFactory:

```

public class NewsletterModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(DelivererFactory.class).toProvider(
            FactoryProvider.newFactory(DelivererFactory.class,
                Newsletter.class));
        ...
    }
}

```

Context set manually

Here, the `FactoryProvider.newFactory()` method is used to create a provider that will create the relevant factory implementation for us. This saves the step of creating a `DelivererFactory` that adds the contextual dependency by hand.

It doesn't take much imagination to see that this too can be insufficient, for instance, if you had two context objects. One mitigant would be to create another kind of `AssistedProvider` with a type signature that supported two contexts:

```

public interface TwoAssistsProvider<T, C1, C2> {
    T get(C1 context1, C2 context2);
}

```

This works, but what about three contexts? Or four? It very quickly gets out of hand. For such cases you will find the Builder pattern much more powerful.

3.3.5 *Flexible partial injection with the Builder pattern*

If you are familiar with the Gang of Four book on design patterns, you have doubtless heard of the Builder pattern. Builders are often used to assemble object graphs when there are many possible permutations. A Builder is one way to mitigate the constructor pyramid, adding each dependency incrementally. They come in two varieties:

- Using constructor wiring and thus able to produce safe, valid, and immutable objects, but at the sacrifice of injector management
- Using flexible setter wiring (and injector management) but sacrificing the safety of constructor injection

The Builder is a natural solution to all partial-injection problems. Builders can produce objects given only a partial view of their dependencies. In short, that's their job. Listing 3.29 reimagines the reinjection problem with a Builder.

Listing 3.29 Granny gets apples using a Builder

```
public class Granny {
    private AppleBuilder builder;

    @Inject
    public Granny(AppleBuilder b) {
        this.builder = b;
    }

    public void eat() {
        builder.build().consume();
        builder.build().consume();
    }
}

public class AppleBuilder {
    public Apple build() {
        return new Apple();
    }
}
```

Builder is wired via constructor

Construct apples by hand

In this trivial example, there is no difference between a Builder and provider. Notice that I did not use a generic interface for the Builder. Instead I relied directly on an `AppleBuilder`. Although we had the same level of type-safety with the parameterized `Provider<Apple>`, there's an extra level of flexibility we get with Builders. Let's say that some apples are green and some are red. How would you build them with a provider? You can't directly. One solution might be to create a `Provider<GreenApple>` and `Provider<RedApple>`, but this exposes the underlying implementation (of red or green apples) to a client that shouldn't know these details. Builders give us a much simpler solution:

```
public class Granny {
    private AppleBuilder builder;

    @Inject
    public Granny(AppleBuilder b) {
        this.builder = b;
    }

    public void eat() {
        builder.buildRedApple().consume();
        builder.buildGreenApple().consume();
    }
}
```

While Granny's dependencies don't change, she can still get red or green apples without ever knowing anything about `RedApple` or `GreenApple`. To Granny, they are just instances of `Apple`. All this needs is a small addition to `AppleBuilder`:


```

public class AppleBuilder {
    public Apple buildRedApple() {
        return new RedApple();
    }

    public Apple buildGreenApple() {
        return new GreenApple();
    }
}

```

AppleBuilder now encapsulates the construction and any implementation details of Apple (red, green, or otherwise). The dependent, Granny, is kept completely free of implementation details.

Similarly, Builders can be a powerful solution to the contextual injection problem. Listing 3.30 shows this application of the Builder pattern to the newsletter example.

Listing 3.30 A newsletter component, using Builder injection

```

public class NewsletterManager {
    private final List<Recipient> recipients;
    private final DelivererBuilder builder;  ← Inject a Builder
                                           for dependency

    public NewsletterManager(List<Recipient> rs,
                             DelivererBuilder db) {
        this.recipients = rs;
        this.builder = db;
    }

    public void send(Newsletter letter) {
        for (Recipient recipient : recipients) {
            builder.letter(letter);
            Deliverer d = builder.buildDeliverer();  ← Set context and
                                                    build dependency
            d.deliverTo(recipient);
        }
    }
}

public class DelivererBuilder {
    private Newsletter letter;

    public void letter(Newsletter letter) {  ← Setter method
                                              accepts context
        this.letter = letter;
    }

    public Deliverer buildDeliverer() {  ← Construct
                                         with context
        return new Deliverer(letter);
    }
}

```

The interesting thing about listing 3.30 is that we have a setter method named `letter()`, which takes the newsletter (context) object. It temporarily holds the context until it is time to build a `Deliverer` around it: in the `buildDeliverer()` method. Using construction by hand, `DelivererBuilder` ensures that `Deliverer` instances are wired by constructor (and are thus immutable, valid, and safe). Builders are especially

powerful when you have more than one context object to deal with—all you need to do is add setter methods on the Builder. Contrast the following `AssistedProviders` for multiple contexts:

```
public interface TwoAssistsProvider<T, C1, C2> {
    T get(C1 c1, C2 c2);
}

public interface ThreeAssistsProvider<T, C1, C2, C3> {
    T get(C1 c1, C2 c2, C3 c3);
}

...
```

with the builder approach:

```
public class DelivererBuilder {
    private Newsletter letter;
    private String mailServerUrl;
    private int port;

    public void letter(Newsletter letter) {
        this.letter = letter;
    }

    public void mailServerUrl(String url) {
        this.mailServerUrl = url;
    }

    public void port(int port) {
        this.port = port;
    }

    public Deliverer buildDeliverer() {
        return new Deliverer(letter, mailServerUrl, port);
    }
}
```

Now any number of contexts can be set by the `NewsletterManager` (dependent) directly in its `send()` method:

```
public void send(Newsletter letter) {
    for (Recipient recipient : recipients) {
        builder.letter(letter);
        builder.mailServerUrl("mail.wideplay.com");
        builder.port(21);

        Deliverer d = builder.buildDeliverer();
        d.deliverTo(recipient);
    }
}
```

These context objects are all set at the time they are needed (on the Builder, rather than the Deliverer). This not only allows us to hide actual construction and assembly code but also gives us an abstraction layer between dependent and dependency. If you wanted to ignore the port (defaulting to an SSL port instead), you would change only the appropriate Builder calls. This is useful if you need to make small tweaks in service

code without upsetting clients and also for performing some lightweight validation of data (for instance, checking that a port number is within range). Builders also benefit from dependency injection themselves and can do extra setup work if need be. For instance, this Builder transforms email server details into a service for Deliverer:

```
public class DelivererBuilder {
    private final MailServerFinder finder;

    private Newsletter newsletter;
    private String mailServerUrl;
    private int port;

    @Inject
    public DelivererBuilder(MailServerFinder finder) {
        this.finder = finder;
    }

    ...

    public Deliverer buildDeliverer() {
        MailServer server = finder.findMailServer(mailServerUrl, port);

        return new Deliverer(letter, server);
    }
}
```

One important bit of housekeeping: if you are going to reuse a builder, remember to reset it first:

```
public Deliverer buildDeliverer() {
    try {
        return new Deliverer(letter, mailServerUrl, port);
    } finally {
        letter = null;
        mailServerUrl = null;
        port = -1;
    }
}
```

Resetting it ensures that a failed Build sequence doesn't contaminate future uses. You could also obtain a new Builder every time (via a provider). This is especially advisable. And it ensures that a Builder is never reused or used concurrently on accident. Thus far, all the code that we've been dealing with has been under our control. Often it may be necessary to work with code that is not under your control and still apply dependency injection to it. This is the problem of injecting objects in sealed code.

3.4 *Injecting objects in sealed code*

Not all the code you work with is under your control. Many third-party libraries come in binary form and cannot be altered to work with dependency injectors. Adding annotations, refactoring with providers or builders, is out of the question. We'll call this *sealed* code. (Don't confuse this with the C# keyword.)

So if we have no control over sealed code, what can be done to make it work with dependency injection? One answer might be to find a way *not* to use annotations.

3.4.1 Injecting with externalized metadata

Recall some of the early Spring XML configuration. It eliminates the need for annotations, right off the bat, in essence moving configuration metadata from source code to an external location (the XML file). Listing 3.31 shows a sealed class injected purely with externalized metadata.

Listing 3.31 A sealed class injected via external configuration

```
public class Sealed {
    private final Dependency dep;

    public Sealed(Dependency dep) {
        this.dep = dep;
    }
}

<!-- XML injector configuration -->
<beans ...>
    <bean id="sealed" class="Sealed">
        <constructor-arg><bean class="Dependency"/></constructor-arg>
    </bean>
</beans>
```

Here `Sealed` did not have to change, and the injector configuration is straightforward. This is possible even with Guice, using the module to select the appropriate constructor to inject:

```
public class SealedModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(Sealed.class).toConstructor(sealedConstructor());
    }

    private Constructor<Sealed> sealedConstructor() {
        try {
            return Sealed.class.getConstructor(Dependency.class);
        } catch (NoSuchMethodException e) {
            addError(e);
            return null;
        }
    }
}
```

Bind directly to a
constructor of `Sealed`

Resolve the constructor
we want to inject

This allows us to skip the `@Inject` annotation and get Guice to directly inject the sealed code.

In Spring, this technique even works with setter injection. See figure 3.20.

But sealed code often throws you more curveballs than this. It may have completely private constructors and

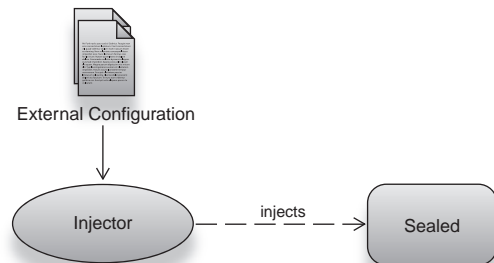


Figure 3.20 External metadata allows you to inject classes in sealed code easily.

expose only a static factory method. This is not uncommon in library code (see listing 3.32).

Listing 3.32 A sealed class injected via external metadata

```
public class Sealed {
    private final Dependency dep;

    private Sealed(Dependency dep) {
        this.dep = dep;
    }

    public static Sealed newInstance(Dependency dep) {
        return new Sealed(dep);
    }
}

<!-- XML injector configuration -->
<beans ...>
    <bean id="sealed" class="Sealed" factory-method="newInstance">
        <constructor-arg><bean class="Dependency"/></constructor-arg>
    </bean>
</beans>
```

Listing 3.32 shows how Spring is able to call on Factory methods just as though they were constructors (the `<constructor-arg>` element now passes arguments to the Factory). If it is an unfriendly factory that completely encapsulates construction, the situation is a bit trickier:

```
public class Sealed {
    private final Dependency dep;

    Sealed(Dependency dep) {
        this.dep = dep;
    }

    public static Sealed newInstance() {
        return new Sealed(new Dependency());
    }
}
```

There is no obvious way to provide the constructor with an instance of `Dependency`. Custom or mock implementations have been removed from the equation, also making testing very difficult. Here's another scenario where the XML falls over:

```
public class Sealed {
    private Dependency dep;

    public Sealed() {
    }

    public void dependOn(Dependency dep) {
        this.dep = dep;
    }
}
```

This class accepts its dependency via setter injection. However, the setter method does not conform to Spring's naming convention. Rather than being named

setDependency(), it is called `dependOn()`. Remember, we can't change any of this code—it is *sealed*.

Even if there were some way around it, misspelling method names can easily cause you much chagrin. The Adapter pattern provides a better solution.

3.4.2 Using the Adapter pattern

The Adapter is yet another design pattern from the Gang of Four book. It allows you to alter the behavior of existing objects by extending them. The following Adapter allows you to inject classes whose constructors are not public (so long as they are not private):

```
public class SealedAdapter extends Sealed {
    @Inject
    public SealedAdapter(Dependency dep) {
        super(dep);
    }
}
```

The call to `super(dep)` chains to `Sealed`'s hidden constructor. Since `SealedAdapter` extends `Sealed`, it can be used by any dependent of `Sealed` transparently. And it benefits from dependency injection (see figure 3.21).

The same solution applies when you have a *package-local* constructor, except you build `SealedAdapter` into the same package as `Sealed`.

Similarly, this works with the unconventional setter methods:

```
public class SealedAdapter extends Sealed {
    @Inject
    public SealedAdapter(Dependency dep) {
        dependOn(dep);
    }
}
```

Here's a more convoluted example with both combined:

```
public class SealedAdapter extends Sealed {
    @Inject
    public SealedAdapter(Dependency dep1, Dependency dep2, Dependency dep3) {
        super(dep1);
        dependOn(dep2);
        relyOn(dep3);
    }
}
```

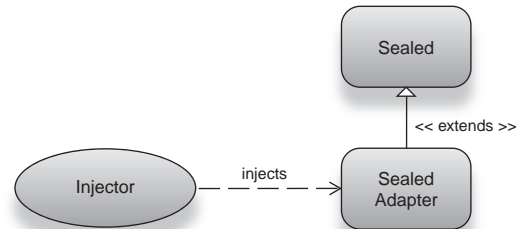


Figure 3.21 `SealedAdapter` helps `Sealed` to be injected transparently.

Using an Adapter:

- Any infrastructure logic is cleanly encapsulated within `SealedAdapter`.
- Typing mistakes and misspellings are caught at compile time.
- Dependents of `Sealed` require no code changes (unlike a provider, for example).
- Testing follows the same method as the original sealed class.

Adapters can sometimes be more verbose, but they have no impact on client code. And they can even be passed back to *other* sealed classes transparently, something that's hard to do with providers.

TIP If you're particularly clever, you can bridge the gap between adapter and provider to solve the reinjection problem. Do this by overriding each method of the original component and looking up a new instance as needed with a provider. Then delegate all methods to that instance. It leaves minimal impact on dependent code and is a compelling alternative.

One other interesting option for injecting sealed code is the Builder pattern we saw earlier. You can incrementally add dependencies to a builder and let it decide how to construct the sealed component. Builders are particularly useful with unconventional setters or if there are several constructors or factories to choose from.

3.5 **Summary**

This chapter was a real workout! First, we explored the injection idioms of setter, constructor, interface, and method decoration. Setter injection has advantages in being flexible, particularly if you don't need every dependency set all of the time. However, it has significant drawbacks since fields can't be made immutable (constant), which is important to preserving object graph integrity. Constructor injection is a compelling alternative, since it does allow you to set immutable fields. It is also useful since you can't accidentally forget to set a dependency (such configuration fails early and fast). Constructor injection is also less verbose and prevents accidental overwriting of previously set dependencies.

However, it too has some drawbacks: If you have several dependencies (or worse, several similar ones), it is difficult to tell them apart. Furthermore, a circular reference (where two components depend on each other) is impossible to resolve with simple constructor wiring. Setter injection helps here, since both objects can be constructed in an incomplete state and later wired to one another. Setter injection also helps avoid the constructor pyramid, which is a design issue when you have multiple profiles of a component, in other words, where a component uses different subsets of its dependencies in different scenarios. Setters are also explicit about the dependencies they wire and thus make for more browsable code.

Constructor injection can be used to solve the circular reference problem, but it requires an intermediary placeholder known as a proxy. Some DI libraries (such as Guice) provide this out of the box, without any additional coding on your part. A related issue, the in-construction problem, where circular interdependents are used

before the proxied wiring can complete, can be solved by combining constructor and setter injection or purely with setter injection. Better yet, you can solve this by using a special initializing hook.

However, the drawbacks of field mutability and the vulnerability that setter injection has to accidental invalid construction are too great to ignore. Non-final fields are also potentially unsafe to multiple interleaving threads (more on that in chapter 9). So, always try constructor injection first and fall back to setter injection only where you must.

There are also other injection idioms:

- *Interface injection*—This involves exposing role interfaces that are effectively setter methods in their own interface. It has the advantage of being explicit just like setters, and it can be given meaningful names. However, interface injection is extremely verbose and requires several single-use interfaces to be created for the sole purpose of wiring. It is also not supported by many DI libraries.
- *Method decoration*—This involves intercepting a method call and returning an injector-provided value instead. In other words, method decoration turns an ordinary method into a Factory method, but importantly, an injector-backed Factory method. This is useful in some rare cases, but it is difficult to test without an injector and so should be carefully weighed first.
- *Other idioms*—Field injection is a useful utility, where fields are set directly using reflection. This makes for compact classes and is good in tutorial code where space is at a premium. However, it is nearly impossible to test (or replace with mocks) and so should be avoided in production code. Object enhancement is a neat idea that removes the injector from the picture altogether, instead enhancing objects via source code or otherwise leaving them to inject themselves. This is different from Factories or Service Location because it does not pollute source code at development time. There are very few such solutions in practice.

Some components “use up” their dependencies and need them reinjected (within their lifetimes). This introduces a problem, namely reinjection. Reinjection can be solved by the only, which involves the use of a single-method interface that encapsulates the work of constructing a dependency by hand or looks it up from the injector by service location. Providers let you abstract away infrastructure logic from dependent components and are thus a compelling solution to reinjection.

Contextual injection is a related problem. Here, a component may need new dependencies on each use, but more important it needs to provide them with context. This is a form of dependency that is known only at the time of use. Assisted injection can help. A slight variation of the Provider pattern, an Assisted Provider does the same job but passes in the context object to a partially constructed dependency. Since this can quickly get out of hand if you have more than one context object, try looking at the Builder pattern instead. The Builder incrementally absorbs dependencies (context objects or otherwise) and then decides how to construct the original service. It may use constructor wiring or a series of setters or indeed a combination as appropriate. The

builder is also an easy solution to the reinjection problem and in such cases is very similar to a provider.

Some code is beyond your control, either third-party libraries, frozen code, or code that can't be changed for some other reasons (such as time constraints). You still need these components, so you need a way to bring them into a modern architecture with dependency injection. DI is really good at this work and at neutralizing such undulations in a large code base. One solution is to use external metadata (if your injector supports it) such as an XML file. This leaves the sealed classes unaffected but still provides them with requisite dependencies. However, if sealed code uses factories or unconventional setter methods, this may be difficult. In this case, the Adapter pattern is a powerful solution. You inject the Adapter as per usual, and then the Adapter decides how to wire the original component. Adapters are a strong solution because they can leave client code unaffected. They are also a transparent alternative to the reinjection problem, though they are sometimes verbose compared to providers.

Dependency injection is about components that are designed, tested, and maintained in discrete modular units that can be assembled, constructed, and deployed in various configurations. It helps separate your code into purposed, uncluttered pieces, which is essential for testing and maintaining code, especially in large projects with many contributors. Chapter 4 examines these ideas in detail, exploring the value of such modular separation and how to achieve it.

4

Building modular applications

This chapter covers:

- Organizing code in modules
- Watching out for tight coupling
- Designing with loose coupling
- Testing code in discrete units
- Working with key rebinding

“To define it rudely but not ineptly, engineering is the art of doing that well with one dollar, which any bungler can do with two after a fashion.”

—Arthur Wellesley

So far, we’ve looked at what it means for objects to be well behaved and classes to be well designed in the small. But in any real-world project, there’s also a big picture to be considered. Just as there are principles that help our design of objects in the micro, there are principles for good design in the macro sense. In the coming sections, we’ll talk about these design principles and see how to relate them to dependency injection.

Chief among these is testing. DI facilitates *testing* and *testable* code, the latter being an extremely important and often-overlooked condition to successful development. Closely tied to this is the concept of *coupling*, where good classes are linked to the dependencies in ways that facilitate easy replacement and therefore testing—bad ones are not. We'll see how to avoid such cases, and finally we'll look at the advanced case of modifying injector configuration in a running application.

First, let's start at in the micro level and explore the role objects play as the *construction units* of applications.

4.1 *Understanding the role of an object*

We're all very familiar with objects; you work with them every day and use them to model problems, effortlessly. But let's say for a moment you were asked to define what an object is—what might you say?

You might say an object is:

- A logical grouping of data and related operations
- An instance of a class of things
- A component with specific responsibilities

An object is all these things, but it is also a building block for programs. And as such, the design of objects is paramount to the design of programs themselves. Classes that have a specific, well-defined purpose and stay within clearly defined boundaries are well behaved and reliable. Classes that grow organically, with functionality bolted on when and where required, lead to a world of hurt.

A class can itself be a member of a larger unit of collective responsibilities called a *module*. A module is an independent, contractually sound unit that is focused on a broad part of business responsibility. For example, a *persistence* module may be responsible for storing and retrieving data from a database.

A module may not necessarily be meant for business functionality alone. For example, a *security* module is responsible for guarding unwarranted access to parts of an application. Modules may also be focused on infrastructure or on application logic but typically not both. In other words, a module is:

- *Whole*—A module is a *complete unit* of responsibility. With respect to an application, this means that modules can be picked up and dropped in as needed.
- *Independent*—Unlike an object, a module does not have dependencies on other modules to perform its core function. Apart from some common libraries, a module can be developed and tested independently (that is, in an isolated environment).
- *Contractually sound*—A module conforms to well-defined behavior and can be relied on to behave as expected under all circumstances.
- *Separate*—A module is not invasive of collaborators, and thus it is a discrete unit of functionality.

These qualities of a module are largely important in relation to its collaborators. Many modules interacting with one another through established, patent boundaries form a healthy application. Since modules may be contributed by several different parties (perhaps different teams, sister projects, or even external vendors), it's crucial that they follow these principles. Swapping in replacement modules, for example, replacing persistence in a database with a module that provides persistence in a data cluster or replacing a web presentation module with a desktop GUI, ought to be possible with a minimum of fuss. So long as the overall purpose of the application is maintained, a system of well-designed modules is tolerant to change. Much as different incarnations of an object graph provide variant implementations of a service, different assemblies of modules provide different application semantics. A module is thus an independent, atomic unit of reuse.

Objects and modules that collaborate typically have strong relationships with each other. Often the design of a dependency is influenced by its collaborators. However, each object has its area of responsibility, and well-designed objects stick to their areas without intruding on their collaborators. In other words, each object has its area of concern. Good design keeps those concerns separated.

4.2 Separation of concerns (*my pants are too tight!*)

Tight pants are very cumbersome! If, like me, you live in a sweltering subtropical environment, they can be especially discomfiting. Similarly, tightly coupled code can cause no end of development pain. It's worth taking the time at the head of a project to prevent such problems from creeping in later.

A big part of modular design is the idea that modules are deployable in different scenarios with little extra effort. The *separation* and *independence* of modules are core to this philosophy, which brings us to the tight-pants problem.

If we take a different perspective on the whole matter, it's not difficult to see that a module is itself a kind of object. The same principles that apply to module design and behavior also apply right down to objects themselves. In the next few sections we'll examine what it means to treat objects with the principle of separation. Earlier we laid down a similar principle: separating infrastructure from application logic so that logic that dealt with construction, organization, and bootstrapping was housed independently from core business logic. If you think of infrastructure as being orthogonal to application logic, then separating the two can be seen as dividing *horizontally* (see figure 4.1).

Similarly, logic dealing with different business areas can be separated *vertically* (see figure 4.2).

Checking spelling and editing text are two core parts of any email application. Both deal with application logic and are thus focused on business purpose. However, neither is especially related to the other. It

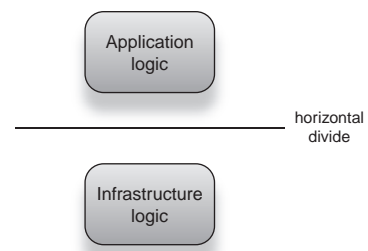


Figure 4.1 Conceptually, application logic sits on top of infrastructure logic.

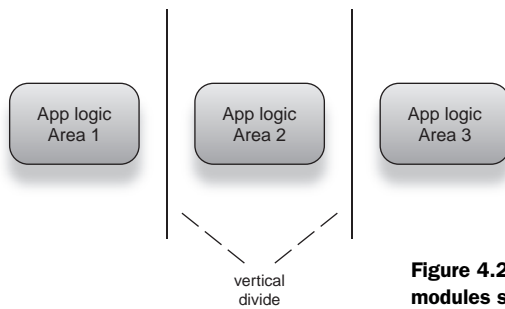


Figure 4.2 Application logic modules sit next to each other.

behoooves us to separate them from each other just as we would separate any infrastructure code from the two of them. In other words, separating logic by area of concern is good practice. Figure 4.3 shows a group of modules, separated by area of application as well as infrastructure concern. This kind of modularity is indicative of healthy design.

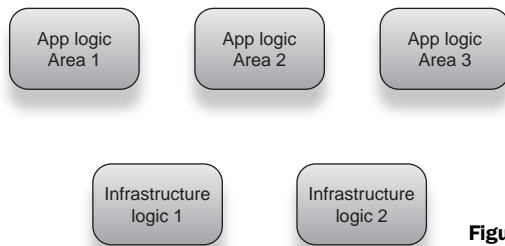


Figure 4.3 An assembly of discrete, separated modules along both infrastructure and core logic lines

When we lose this healthy separation between modules (and indeed objects), things start to get messy. Broadly speaking, this is tight coupling, and it has been a major headache for a very long time in OOP. Many language constructs were developed just to deal with this problem (interfaces and virtual methods, for instance). To know how to write healthy code, one must first be able to recognize tightly coupled code and understand *why* it is a problem.

4.2.1 *Perils of tight coupling*

In chapter 1, we looked at a classic example of tightly coupled code, declared that it was bad, tossed it away, and laid the foundation for DI. Let's resurrect that example and examine it closely (listing 4.1).

Listing 4.1 An email service that checks spelling in English

```
public class Emailer {
    private EnglishSpellChecker spellChecker;

    public Emailer() {
        this.spellChecker = new EnglishSpellChecker();
    }

    ...
}
```

This email service was poorly written because it encapsulated not only its dependencies but also the *creation* of its dependencies. In other words:

- It prevents any external agent (like the injector) from reaching its dependencies.
- It allows for only one *particular* structure of its object graph (created in its constructor).
- It is forced to know how to construct and assemble its dependencies.
- It is forced to know how to construct and assemble dependencies of its dependencies, and so on ad infinitum.

By preventing any external agent from reaching its dependencies, not only does it prevent an injector from creating and wiring them, it prevents a unit test from substituting *mock objects* in their place. This is an important fact because it means the class is *not testable*. Take the test case in listing 4.2 for instance:

Listing 4.2 A test case for `Emailer` using a mocked dependency

```
public class EmailerTest {
    @Test
    public final void ensureSpellingWasChecked() {
        MockSpellChecker mock = new MockSpellChecker();

        new Emailer(mock).send("Hello!");
        assert mock.verifyDidCheckSpelling()
            : "failed to check spelling";
    }
}
```

Mock tracks if `checkSpelling()` was called

And the mock spellchecker is as follows:

```
public class MockSpellChecker implements SpellChecker {
    private boolean didCheckSpelling = false;

    public boolean check(String text) {
        didCheckSpelling = true;
        return true;
    }

    public boolean verifyDidCheckSpelling() {
        return didCheckSpelling;
    }
}
```

Mocked implementation of interface method

Mock-only method reports if spelling was checked

This test case is impossible to write on `Emailer` as it exists in listing 4.1. That's because there is no constructor or setter available to pass in a mock `SpellChecker`. Mock objects are extremely useful:

- They allow you to test one class and none other. This means that any resultant errors are from that class and none other. In other words, they help you focus on discrete units.
- They allow you to replace computationally expensive dependencies (for instance, those that require hardware resources) with *fake* versions.

- They assert that the class follows the appropriate *contract* when speaking to its dependencies.
- They assert that everything happened as expected and in the expected *order*.

Even if we rewrote the original `Emailer` to take its dependency as an argument, we would still be stuck in this untenable position (see listing 4.3).

Listing 4.3 An email service that checks spelling in English, modified

```
public class Emailer {
    private EnglishSpellChecker spellChecker;

    public Emailer(EnglishSpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }

    ...
}
```

Now an external agent can set dependencies. However, the problem with testing remains, because `Emailer` is bound inextricably to an `EnglishSpellChecker`. Passing in a mock

```
MockSpellChecker mock = new MockSpellChecker();
new Emailer(mock);
```

results in a compilation error, because `MockSpellChecker` is not an `EnglishSpellChecker`. When a dependent is inextricably bound to its dependencies, code is no longer testable. This, in a nutshell, is tight coupling.

Being tightly coupled to a specific implementation, `Emailer` also prevents you from producing variant implementations. Take the following injector configuration, which tries to assemble an object graph consisting of an `Emailer` with an `EnglishSpellChecker` and a Latin character set in Spring XML (also see figure 4.4):

```
<beans ...>
    <bean id="emailer" class="Emailer">
        <constructor-arg>
            <bean class="EnglishSpellChecker">
                <constructor-arg>
                    <bean class="LatinCharset"/>
                </constructor-arg>
            </bean>
        </constructor-arg>
    </bean>
</beans>
```

Such a configuration is impossible given the tightly coupled object graph. By deciding its own dependencies, `Emailer` prevents any variant of them from being created. This makes for poor reuse and a potential explosion of similar code, since a new kind of `Emailer` would have to be written for each permutation.

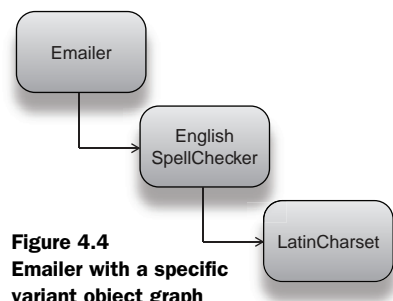


Figure 4.4
Emailer with a specific
variant object graph

The example I used in chapter 1 is illustrative of the same problem—we were restricted to English spellcheckers, unable to provide variants in French or Japanese without a parallel development effort. Tight coupling also means that we have opened the door in the iron wall separating infrastructure from application logic—something that should raise alarm bells on its own.

4.2.2 Refactoring impacts of tight coupling

We’ve built up quite a potent argument *against* coupling dependents to their dependencies. There’s one major reason we can add that would outweigh all the others: the impact to coupled code when dependencies change. Consider the example in listing 4.4.

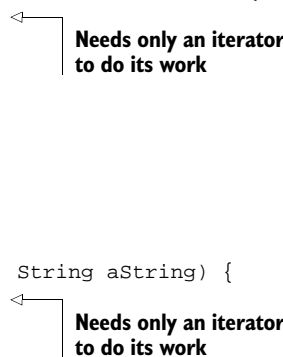
Listing 4.4 A utility for searching lists of strings

```
public class StringSearch {
    public String startsWith(ArrayList<String> list, String aString) {
        Iterator<String> iter = list.iterator();
        while(iter.hasNext()) {
            String current = iter.next();

            if (current.startsWith(aString))
                return current;
        }
        return null;
    }

    public boolean contains(ArrayList<String> list, String aString) {
        Iterator<String> iter = list.iterator();
        while(iter.hasNext()) {
            String current = iter.next();

            if (aString.equals(current))
                return true;
        }
        return false;
    }
    ...
}
```



Needs only an iterator to do its work

Needs only an iterator to do its work

```
//elsewhere
String startsWith = new StringSearch().startsWith(myList, "copern");
boolean contains = new StringSearch().contains(myList, "copernicus");
```

StringSearch provides a couple of utilities for searching a list of strings. The two methods I have in listing 4.5 test to see if the list contains a particular string and if it contains a partial match. You can imagine many more such utilities that comprehend a list in various ways. Now, what if I wanted to change StringSearch so it searched a HashSet instead of an ArrayList?

Listing 4.5 Impact of refactoring tightly coupled code

```
public class StringSearch {
    public String startsWith(HashSet<String> set, String aString) {
```



```

        Iterator<String> iter = set.iterator();
        while(iter.hasNext()) {
            String current = iter.next();

            if (current.startsWith(aString))
                return current;
        }

        return null;
    }

    public boolean contains(HashSet<String> set, String aString) {
        Iterator<String> iter = set.iterator();
        while(iter.hasNext()) {
            String current = iter.next();

            if (aString.equals(current))
                return true;
        }

        return false;
    }

    ...
}

```

Client code can now search over a `HashSet` instead but only after a significant number of changes to `StringSearch` (one per method):

```

String startsWith = new StringSearch().startsWith(mySet, "copern");
boolean contains = new StringSearch().contains(mySet, "copernicus");

```

Furthermore, you would have to make a change in every method that the implementation appeared in, and the class is no longer compatible with `ArrayLists`. Any clients already coded to use `ArrayLists` with `StringSearch` must be rewritten to use `HashSets` at a potentially an enormous refactoring cost, which may not even be appropriate in all cases. One way to solve this problem is by making the list a dependency, as shown in listing 4.6.

Listing 4.6 Refactor the list to a dependency

```

public class StringSearch {
    private final HashSet<String> set;

    public StringSearch(HashSet<String> set) {
        this.set = set;
    }

    public String startsWith(String aString) {
        Iterator<String> iter = set.iterator();
        while(iter.hasNext()) {
            String current = iter.next();

            if (current.startsWith(aString))
                return current;
        }
    }
}

```

```

        return null;
    }

    public boolean contains(String aString) {
        Iterator<String> iter = set.iterator();
        while(iter.hasNext()) {
            String current = iter.next();

            if (aString.equals(current))
                return true;
        }

        return false;
    }

    ...
}

```

Consequently in client code, we can use one instance of the searching class to perform many actions on a set of strings:

```

StringSearch stringSearch = new StringSearch(mySet);
String startsWith = stringSearch.startsWith("copern");
boolean contains = stringSearch.contains("copernicus");

```

One problem is out of the way—we can now search HashSets. And if you had a new requirement in the future to search, say, TreeSets, then the refactoring impact would be much smaller (we’d need to change only the dependency and any clients). But this is still far from ideal. A change every time you choose a different collection data structure is excessive and untenable. We’re still feeling the pangs of tight coupling. Listing 4.7 shows how *loose* coupling code cures this ailment.

Listing 4.7 Refactor to a loosely coupled dependency

```

public class StringSearch {
    private final Collection<String> collection;

    public StringSearch(Collection<String> collection) {
        this.collection = collection;
    }

    public String startsWith(String aString) {
        Iterator<String> iter = collection.iterator();
        while(iter.hasNext()) {
            String current = iter.next();

            if (current.startsWith(aString))
                return current;
        }

        return null;
    }

    public boolean contains(String aString) {
        Iterator<String> iter = collection.iterator();
        while(iter.hasNext()) {

```

```

        String current = iter.next();
        if (aString.equals(current))
            return true;
    }
    return false;
}
...
}

```

By placing an abstraction between dependent and dependency (the interface `Collection`), the code loses any awareness of the underlying data structure and interacts with it only through an interface. All of these use cases now work without any coding changes:

```

boolean inList = new StringSearch(myList).contains("copern");
boolean inSet = new StringSearch(mySet).contains("copernicus");
boolean hasKey = new StringSearch(myMap.keySet()).contains("copernicus");
...

```

Now `StringSearch` is completely oblivious to the decisions of clients to use any kind of data structure (so long as they implement the `Collection` interface). You can even add new methods to `StringSearch` without impacting existing clients. This ability of the `Collection` interface to act as a contract for various services is extremely valuable, and you can mimic it in your own services to achieve a similar goal. Services that act according to a contract are well behaved. Going about this can be tricky, and we'll outline some best practices in the next section.

4.2.3 *Programming to contract*

Loose coupling via interfaces leads nicely to the idea of *programming to contract*. Recall that in chapter 1 we described a service as in part defined by a well-understood set of responsibilities called a contract. Programming to contract means that neither client nor service is aware of the other and communicates only via the contract. The only common ground is the understanding provided by this contract (interface). This means that either can evolve independently so long as both abide by the terms of the contract. A contract is also much more than a means of communication:

- It is a precise specification of behavior.
- It is a metaphor for *real-world* business purpose.
- It is a revelation of intent (of appropriate use).
- It is an inherent mode of documentation.
- It is the keystone for behavior *verification*.

The last point is extremely important because it means that a contract provides the means of testing classes. Verifying that an implementation works as it should is done by testing it against a contract. In a sense, a contract is a *functional requirements specification*

for any implementation and one that is native to the programming language itself. Furthermore, a contract represents conceptual purpose and is thus akin to a *business contract*.

When ordering coffee at Starbucks, you follow a well-defined business process:

- 1 Stand in line.
- 2 Place your order.
- 3 Move to the edge of the counter to pick it up.

If Starbucks were to automate this process, the conceptual obligation between you as a client and the attendant would be a *programmable* contract—probably not too different from listing 4.8.

Listing 4.8 The coffee shop contract!

```
public interface Customer {
    void waitInLine();

    boolean placeOrder(String choice);

    void pickup(Coffee prepared);
}

public interface Attendant {
    boolean takeOrder(String choice);

    Coffee prepareOrder();
}
```

Of course, there are plenty more steps in the real world (exchange of money, for one) but let's set them aside for now. What's important to note is that neither interface says anything about *how* to perform their function, merely *what* that function is. Well-behaved objects adhere strictly to their contracts, and accompanying unit tests help verify that they do. If any class were to violate the terms of its contract, the entire application would be at risk and most probably broken. If Starbucks were out of cappuccino and an order for one were placed, `Attendant.takeOrder()` would be expected to return `false`. If instead it returned `true` but failed to serve the order, this would break `Customers`.

Any class developed to the contract `Customer` or `Attendant` can be a drop-in replacement for either, while leaving the overall Starbucks system behaviorally consistent. Imagine a `BrazilianCustomer` served by a `RoboticAttendant` or perhaps a properly trained `ChimpanzeeCustomer` and `GorillaAttendant`.

Modeling conceptual business purpose goes much further. A carefully developed contract is the cornerstone for communication between business representatives and developers. Developers are clever, mathematical folk who are able to move quickly between abstractions and empiricism. But this is often difficult for representatives of the business who may be unfamiliar with software yet in turn contain all the knowledge about the problem that developers need. A contract can help clarify confusion and provide a bridge for common understanding between abstraction-favoring developers and

empirical business folk. While you probably won't be pushing sheets of printed code under their noses, describing the system's interfaces and interoperation forms a reliable basis for communicating ideas around design.

A contract is also a *revelation of intent*. As such, the appropriate manner and context of a service's usage are revealed by its contract definition. This even extends to error handling and recovery. If instead of returning false, `Attendant.takeOrder()` threw an `OutOfCoffeeException`, the entire process would fail. Similarly, an `InputStream`'s methods throw `IOException` if there is an error because of hardware failure. `IOException` is explicitly declared on each relevant method and is thus part of the contract. Handling and recovering from such a failure are the responsibility of a client, perhaps by showing a friendly message or by trying the operation again.

Finally, a contract should be simple and readable, clear as to its purpose. Reading through a contract should tell you a lot about the classes that implement it. In this sense, it is an essential form of documentation. Look at this semantic interface from the `StringSearch` example shown earlier:

```
public interface StringSearch {
    boolean contains(String aString);

    String startsWith(String fragment);
}
```

This is a start, but there are several missing pieces. In the simple case of `contains()` we can infer that since it returns a boolean, it will return true if the string is found. However, what exactly does `startsWith()` return? At a guess, it returns a single successful match. But is this the first match? The last? And what does it return when there are no matches? Clearly, this is an inadequate specification to develop against. One further iteration is shown in listing 4.9.

Listing 4.9 `StringSearch` expressed as a contract

```
public interface StringSearch {

    /**
     * Tests if a string is contained within its list.
     *
     * @param aString Any string to look for.
     * @returns Returns true only if a matching string is found.
     */
    boolean contains(String aString);

    /**
     * Tests if a string in the list begins with a given sequence.
     *
     * @param fragment A partial string to search on.
     * @returns Returns the first match found or null if
     *         none were found.
     */
    String startsWith(String fragment);
}
```

Now `StringSearch` is clear about its behavior; `startsWith()` returns the first match or null if no matches are found. The implied benefit of this contractual programming is that code can be discretized into modules and units without their severely affecting one another when they change. DI is a natural fit in providing a framework for programming such loosely coupled code.

4.2.4 Loose coupling with dependency injection

So far, we've seen that loose coupling makes testing, reuse, and evolution of components easy. This makes for modules that are easy to build and maintain. Dependency injection helps by keeping classes relatively free of infrastructure code and by making it easy to assemble objects in various combinations. Because loosely coupled objects rely on contracts to collaborate with dependencies, it is also easy to plug in different implementations, via injection or mocks in a unit test. All it takes is a small change to the binding step. Keeping object graph structure described in one place also makes it easy to find and modify parts of it, with little impact to core application code.

Let's see this in practice. Let's say we have a book catalog, consisting of a library to search and locate books with. By following the principles of loose coupling, we've arrived at the interfaces shown in listing 4.10.

Listing 4.10 A book catalog and its library data service

```
public interface BookCatalog {
    Book search(String criteria);
}

public interface Library {
    Book findByTitle(String title);
    Book findByAuthor(String title);
    Book findByIsbn(String title);
}
```

`BookCatalog` refers only to the interface `Library`. Its `search()` method translates some free-form text search criteria into a search by title, author, or ISBN, subsequently calling the appropriate method on `Library`. In a simple case, the `Library` may be implemented as a file-based service, storing and retrieving `Book` records from disk. In bigger applications it will likely be an external database (like PostgreSQL¹ or Oracle). Altering the catalog system to use a database-backed variant of `Library` is as simple as changing `Library`'s binding (see listing 4.11) to use the database-driven implementation.

Listing 4.11 A book catalog and a database-backed library service

```
public class BooksModule extends AbstractModule {
    @Override
    protected void configure() {
```

¹ PostgreSQL is an excellent, lightweight, and feature packed open source database. Find out more at <http://www.postgresql.org>.

```

        bind(Library.class).to(DatabaseLibrary.class);
        bind(BookCatalog.class).to(SimpleBookCatalog.class);
    }
}

```

So long as `DatabaseLibrary` correctly implements the `Library` interface, the program continues to work unhindered, and `BookCatalog` is none the wiser about its underlying storage mechanism. You don't have to compile it again. Listing 4.12 shows another change—this time of the catalog.

Listing 4.12 A desktop-GUI book catalog and a database-backed library service

```

public BooksModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(Library.class).to(DatabaseLibrary.class);
        bind(BookCatalog.class).to(DesktopGUIBookCatalog.class);
    }
}

```

This time it's the `Library` that's oblivious to the goings on in our application. Loose coupling enables any of our services to evolve down their own paths and yet remain verifiable and behaviorally consistent (figure 4.5 visualizes this evolution), performing the intended service for end users.

Loose assembly of modules is extremely important to testing. It provides a tangible benefit since such code can be constructed easily in tests and given mock dependencies without any additional work. This last part is crucial to writing testable code. In the next section we will see why writing testable code is important and how it helps in developing better application designs in general.

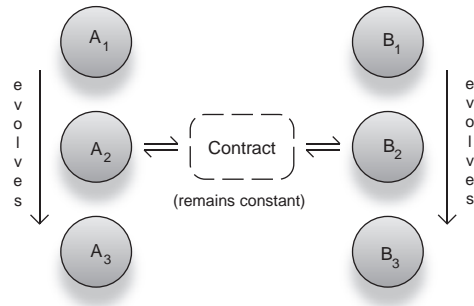


Figure 4.5 Independent evolution of A and B, with no impact to their collaboration

4.3 Testing components

One of the most important parts of software development is testing. Technology pundits argue that there is little value to code that has not been verified by some formal process. Without being able to assert expectations against an object's behavior, there's no way to tell if it fulfills its function. The fact that it compiles correctly, or runs in a deployment environment, is fairly useless in the broad sense. A functional test of an application (say by clicking through menu items in a desktop application) is a mite better than nothing at all, but not significantly so. It ignores several unseen failure points, such as bad data or concurrency bottlenecks.

While functional testing has its place in the quality assurance spectrum, the veracity of code simply can't be met without automated unit and integration tests. These tests provide the following:

- Unit-level assurance of functionality
- Simulation of disaster situations like failing hardware or resource depletion
- Detection of exceptional situations such as bugs in collaborating modules
- Detection of concurrency bottlenecks or thread-unsafe code
- Assurance of regression integrity²
- Benchmarking performance under extreme duress

Not only can you test a much broader spectrum of possible scenarios in code, but you can simulate disaster situations and measure performance under various loads. Automated tests are thus a vital part of any application, particularly one that is *programmed to contract*. Testing comes in different flavors, some or all of which can be automated by writing test code. Primary among these is the test accompanying an individual class or unit. This kind of test is independent of the rest of the application and intended to verify just the one unit. These tests are generally called *unit tests* but sometimes are also called *out-of-container* tests. Another form of testing asserts integration behavior between modules in a simulated environment.

4.3.1 Out-of-container (unit) testing

The idea of testing an individual unit of functionality (a class) raises questions about the surrounding environment. What about its dependencies? Its collaborators? Its clients? Testing a unit is *independent* of these surrounds. And this means independent of an injector too. As listing 4.13 shows, unit tests are not concerned with anything but the unit (single class) that they are testing.

Listing 4.13 A unit test of the book catalog, from listing 4.10

```
public class BookCatalogTest {
    @Test
    public final void freeFormSearch() {
        MockLibrary mock = new MockLibrary();

        new SimpleBookCatalog(mock)
            .search("dependency injection");

        assert mock.foundByKeyword();
    }
}
```

Test verifies search

Verify mock after search

We are quite happy to use construction by hand, in particular because there are very few dependencies (all of them mocked) and no dependents besides the test itself. It would be unnecessarily tedious to configure an injector just for a few mocks. If the test passes, there's some assurance that a correctly configured injector will work too, since the test uses normal language constructs. Furthermore, a *mocking framework* can make it a lot easier to create an object with mock dependencies.

² Regression integrity is the idea that previously verified behavior is maintained on new development iterations, so you don't undo the work that's already been done (and successfully tested).

NOTE EasyMock, Mockito, and JMock are such powerful mock objects frameworks for Java. I would highly recommend Mockito to start with and EasyMock for more sophisticated uses.

This kind of test tests services outside an injector and outside an application—hence the name out-of-container testing. A significant point about this kind of testing is that it forces you to write loosely coupled classes. Classes that rely specifically on injector behavior or on other parts of the application are difficult to test in this fashion because they rely on some dependency that only the final application environment can provide. A database connection is a good example. If it's outside an application, these dependencies are not readily available and must be simulated using mocks.

When you encounter classes whose dependencies cannot be mocked, you should probably rethink design choices. Older versions of EJB encouraged such container-dependent code. Infrastructure concerns like security, transactions, or logging often make things harder since their effects are not directly apparent. Developers tend to want to test everything as a whole and end up hacking together parts of a real application or environment. These don't do justice to testing, because they remove the focus on testing individual units of functionality. Errors raised by such tests may be misleading because of subtle differences in application environments. So when testing, try focusing your attention on a single unit or class, mocking out the rest. If you can do this individually for all units, you will have much more confidence in them and can easily assemble them into working modules.

4.3.2 *I really need my dependencies!*

If the class you're testing does little more than call into its dependencies, you might be tempted to use an injector—at least for the relevant module and mock the rest. It's not unusual to see tests like this:

```
public class BookCatalogTest {
    private Injector injector;

    @BeforeMethod
    public final void setup() {
        injector = Guice.createInjector(new TestBooksModule());
    }

    @Test
    public final void freeFormBookSearch() {
        new SimpleBookCatalog(injector.getInstance(Library.class))
            .search("..");
        ...
    }
}
```

This is a bad idea. Not only will you introduce unnecessary complexity in your tests (they now depend on an injector), but you'll also have to maintain ancillary code in the form of `TestBooksModule`. This is injector configuration that exists purely for the unit test and adds no real value. Furthermore, if there are errors in the injector

configuration, the test will fail with a false negative. You may spend hours looking for the bug in the wrong place, wasting time and effort on maintaining code that adds very little value to the test case.

4.3.3 More on mocking dependencies

If your classes do nothing more than call into their dependencies, is it still worth writing unit tests for them? *Absolutely*. In fact, it is imperative if you want to assert your module's integrity in any meaningful way. Verifying the behavior of a service is only half the picture. A class's use of its dependencies is a critical part of its behavior. Mock objects can track and verify that these calls are according to expectation. There are many ways to do this. Let's look at one using EasyMock and writing a behavior script. Listing 4.14 shows such a script for Library and verifies its proper use by BookCatalog. Remember we're trying to verify that BookCatalog depends correctly on Library, in other words, that it *uses* Library properly.

Listing 4.14 A mock script and assertion of behavior

```
import static org.easymock.EasyMock.*;

public class BookCatalogTest {

    @Test
    public final void freeFormBookSearch() {
        Library mock = createStrictMock(Library.class);

        String criteria = "dependency injection";

        expect(mock.findByAuthor(criteria))    ← First try author
            .andReturn(null);

        expect(mock.findByKeyword(criteria))   ← Then try keyword
            .andReturn(null);

        Book di = new Book("dependency injection");
        expect(mock.findByTitle(criteria))     ← Finally try title
            .andReturn(di);

        replay(mock);                          ← Signal the mock
                                              to be ready

        new SimpleBookCatalog(mock)
            .search(criteria);

        verify(mock);    ← Verify its usage
    }
}
```

In listing 4.14, we script the Library mock to expect searches by author and keyword first on each and to return null, signaling that no such book exists. Finally, on title search, we return a hand-created instance of book that we're after. This not only asserts the correct use of Library by SimpleBookCatalog but also asserts correct *order* of use. If SimpleBookCatalog were written to prefer searching by title over keyword and author, this test would fail, alerting us to the mistake. To complete the test, we can

add yet another assertion that tests the behavior of `SimpleBookCatalog`. Here's the relevant portion from listing 4.14, modified appropriately:

```
Book di = new Book("dependency injection");  
  
...  
Book result = new SimpleBookCatalog(mock).search(criteria);  
  
assert di.equals(result) : "Unexpected result was returned";
```

Notice that we perform this assertion in addition to the assertions that the mocks provide. Now we have a complete test of `SimpleBookCatalog`. A more elaborate form of testing takes more than one unit into account. This helps detect problems in the coherency between units and determines whether the application behaves well in a broader context. This form of testing is known as integration testing.

4.3.4 *Integration testing*

So far we've shown how to test individual units and assert their behavior. This ought to be the first step in any development process. We've also laid down reasons for keeping tests independent of an injector or other environment-specific concerns. As your code matures, and the time arrives for a completed module to be integrated into the larger system, new testing concerns arise. These concerns have to do with the interoperation of modules, their dependence on external factors like hardware or database resources, and the overall coherence of the system. Automated tests can help in this regard too, and this flavor of tests is called integration tests.

Integration testing is about testing how various pieces fit together as a whole. Naturally, the dependency injector plays a vital part in this assembly by helping connect modules together in a transparent and cohesive fashion. Figure 4.6 illustrates this architecture.

As figure 4.6 shows, infrastructure concerns like persistence and security are encapsulated in their own modules and interfaced with logic in other modules. The dependency injector is responsible for pulling all of them together to form a coherent whole.

The idea behind integration testing is to simulate user interaction with a system or, if the application is not directly exposed to a human user, to simulate the next most relevant actor. This might be:

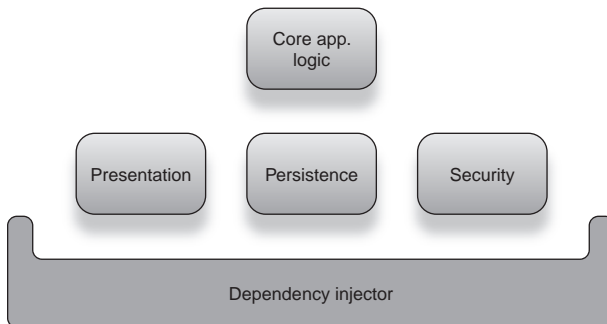


Figure 4.6 Architecture of modules brought together by dependency injection

- A batch of cleared bank checks
- Monitored cooling of a nuclear power plant
- Just about anything else you can imagine

Let's use the example of a web application that serves a page from a database. This will be an automated integration test (listing 4.15).

Listing 4.15 Integration test for a web application

```
public class HomePageIntegrationTest {
    private Injector injector;

    @BeforeTest
    public final void prepareContainer() {
        injector = Guice.createInjector(new WebModule(),
                                       new TestPersistenceModule());
    }

    @AfterTest
    public final void cleanup() { .. }

    @Test
    public final void renderHomePage() throws Exception {
        HttpServletRequest request = createMock(..);
        HttpServletResponse response = createMock(..);

        injector.getInstance(HomeServlet.class)
            .service(request, response);

        //assert something about response
    }
}
```

Use a test environment for data resources

Get page servlet from injector and test

There are interesting things about the integration test in listing 4.15. Foremost among them, it looks like we've violated nearly every rule of unit testing. We're using an injector, not mocking anything apart from HTTP request (and response), and using a specialized configuration for tests. Of course, the simple explanation is that this isn't a unit test, so we can ignore these restrictions. But we can do better than that:

- Integration testing is about testing how modules interoperate, so we want to use actual production modules where possible.
- An accurate analog of the production environment is important, so we try to include as many of the variables as we can that go into it: infrastructure modules, the dependency injector, and any external services.
- There will probably be differences between a production and integration environment (such as which database instance to use), and these are captured in a parallel configuration, `TestPersistenceModule`.
- Integration tests are automated, so external interaction must be simulated (for example, this is done by mocking or simulating HTTP requests).

- Integration tests can be expensive, hence the `prepareContainer()` method that runs once before testing begins.
- They also require clean shutdowns of external services (such as database connections and timers), hence the presence of the `cleanup()` method, which runs after testing is complete.

When used properly, integration testing can help reveal errors in configuration and modular assembly. Typically these are environment-specific and often subtle. A healthy codebase will reveal very few coding errors in integration tests, since classes are covered by unit tests. More often than not, one finds that integration tests reveal configuration quirks in libraries or interoperability issues. These tests are an additional safeguard against design flaws in an architecture. However, they are neither the first line of defense (unit tests) nor the final word on application behavior (acceptance tests and QA). But they are a must-have tool to go in the toolbox of any developer.

Next up, we'll take a gander at integration from a number of perspectives. Modular applications can be integrated in various permutations to suit different goals, whether it be testing or deployment on weaker hardware such as a mobile device, where not all of the modules are required.

4.4 *Different deployment profiles*

We've shown how modules are composite globs of functionality that are like prefab units for big applications. As such, modular applications can be composed in different *profiles*, to different fits. A security module guarding HTTP access can be combined with a presentation module and a file-based persistence module to quickly create a web application. Architectures faithful to modularity principles will be able to evolve better, for instance, by replacing file-based persistence with a module that persists data in an RDBMS database.

While this sounds very simple, it's not always a plug-and-play scenario. Much of it depends on the flexibility of a module and its component classes. The more general and abstract (in other words, loosely coupled) a module's interface, the easier it will be to integrate. This generally comes from experience and natural evolution. Designing a module to be too abstract and all-purposed at the beginning is not a good idea either (and very rarely works).

Once you do have such an architecture, however, there are powerful uses to which it can be put. Chief among those is altering the deployment profile *dynamically*, for instance, by changing injector configuration at runtime. And this is what we call *rebinding*.

4.4.1 *Rebinding dependencies*

First of all, I must say that altering an injector configuration and the model of objects at runtime is potentially dangerous. Without careful library support, this can mean a loss of some validation and safety features that you would normally get at startup time. It can lead to unpredictable, erratic, and even undetectable bugs. However, if used

with care, it can give your application very powerful dynamic features (such as *hot deployment* of changes). This is a fairly advanced use case, so convince yourself first that it is needed and that you can handle the requisite amount of testing and design.

Altering dependency bindings at runtime has several impacts. Injectors are there to give you a leg up, but they can't do everything, and putting them to unusual uses (like rebinding) is fraught with pitfalls:

- Once bound, a key provides instances of its related object until rebound.
- When you rebound a key, all objects already referencing the old binding retain the old instance(s).
- Rebinding is very closely tied to scope. A longer-lived object holding an instance of a key that has been rebound will need to be reinjected or discarded altogether.
- Rebinding is also tied to lifecycle. When significant dependencies are changed, relevant modules may need to be notified (more on this in chapter 7).
- Not all injectors support rebinding, but there are alternative design patterns in such cases.

Injectors that support rebinding are said to be mutable. This says nothing about field or object mutability (which is a different concept altogether). It refers purely to changing the association of a key to an object graph. PicoContainer is one such mutable injector. In the following section, we'll look at how to achieve mutability without a mutable injector, by using the well-known Adapter design pattern.

4.4.2 Mutability with the Adapter pattern

Here we will consider the case of an injector that does not support dynamic rebinding. The problem we're trying to solve is that there isn't enough knowledge while coding to bind all the dependencies appropriately. In other words, the structure of an object graph may change over the life of the application. One very simple solution is to maintain both object graphs and flip a switch when you need to move from one binding to another—something like this:

```
public class LongLived {
    private final DependencyA a;
    private final DependencyB b;
    private boolean useA = true;

    public LongLived(DependencyA a, DependencyB b) {
        this.a = a;
        this.b = b;
    }

    public void rebind() {
        useA = false;
    }

    public void go() {
        if (useA)
            ...
```

Start with a

Work with a

```

        else
            ...    ← Or work with b
    }
}

```

Here the method `rebind()` controls which dependency `LongLived` uses. At some stage in its life, when the rebinding is called for, you need to make a call to `rebind()`.

This works—and probably quite well—but it seems verbose. What’s more, it seems like there’s a lot of infrastructure logic mixed in with our application. If we’ve learned anything so far, it’s to avoid any such mixing. What’s really needed is an abstraction, an intermediary to which we can move the rebinding logic and still remain totally transparent to clients. Providers and builders don’t quite work because they either provide new instances of the *same* binding or provide an instance specific to some *context*. But adapters do. They are transparent to dependents (since an adapter extends its *adaptee*) and wrap any infrastructure, keeping it well hidden. Listing 4.16 demonstrates `LongLived` and its dependencies with the adapter pattern.

Listing 4.16 Dynamic rebinding with an adapter

```

public interface Dependency {
    int calculate();
}

public class DependencyAdapter implements Dependency, Rebindable {
    private final DependencyA a;
    private final DependencyB b;
    private boolean useA = true;

    @Inject
    public DependencyAdapter(DependencyA a, DependencyB b) {
        this.a = a;
        this.b = b;
    }

    public void rebind() {
        useA = false;
    }

    public int calculate() {
        if (useA)
            return a.calculate();

        return b.calculate();
    }
}

```

Now most of the infrastructure code has been moved to `DependencyAdapter`. When the rebinding occurs, flag `useA` is set to `false` and the adapter changes, now delegating calls to `DependencyB` instead. One interesting feature of this is the use of a *Rebindable role interface*. The control logic for dynamic rebinding is thus itself decoupled from the “rebinding” adapters. All it needs to do is maintain a list of *Rebindables*, go through them, and signal each one to `rebind()` when appropriate. This is neat because the logic of deciding which binding to use is completely known at coding time.

Some injectors even allow *multicasting* to make this an atomic, global process. Multicasting (and lifecycle in general) is explored in depth in chapter 7. Most of all, the use of an adapter ensures our client code is lean and behaviorally focused. Here's what LongLived looks like now, after the changes from listing 4.16.

```
public class LongLived {
    private final Dependency dep;

    @Inject
    public LongLived(Dependency dep) {
        this.dep = dep;
    }

    public void go() {
        int result = dep.calculate();
        ...
    }
}
```

That's certainly more concise. Rebinding of the key associated with Dependency is now completely transparent to LongLived—and any other client code for that matter. This is especially important because it means that unit tests don't have to be rewritten to account for a change in infrastructure. This is a satisfying saving.

4.5 Summary

Objects are discrete functional units of data mixed with operations on that data. In the same sense, larger collections of objects and their responsibilities are known as *modules*. A module is typically a *separate* and *independent* compilation unit. It can be maintained, tested, and developed in isolation. So long as it fulfills the terms of its contract to collaborators, a module can be dropped into any well-designed architecture almost transparently.

Components that are invasive of other components or rely on specific implementation details are said to be *tightly coupled*. Tight coupling is detrimental to maintenance and readability. Moreover, it prevents reuse of utility-style components because they are tightly bound to concrete classes rather than abstractions such as interfaces. This reduces the overall modular integrity of a program. To avoid tight coupling, choose abstractions between a client and service. So long as each fulfills the terms of the mediary contract, either can evolve or be replaced with no impact to the other or to the system as a whole. This flows down to component granularity from the concept of modules. Contracts reveal many other things about a component and about design. These are:

- A business metaphor
- An essential form of (self-) documentation
- A revelation of intent
- A specification for building implementations
- A means of verifying behavior

Code that is modular is also easy to test. Testing specific modules (or components) is a vital part of their development, as it is the primary way to verify their correctness. Tests that rely on components to have dependencies wired are poor tests because they can easily confuse the issue when an error occurs. You may spend hours tracking down whether the component under test is responsible or if one of its dependencies caused the error. The use of mock objects is a powerful remedy to this predicament, and indeed it's an important way to narrow down and verify the behavioral correctness of a piece of code. Never allow injectors or other environment-specific frameworks to creep into unit tests, even as a crutch. Try to use mocks and test units in isolation as much as possible. Code that conforms to this focus is said to be tested out of container.

Once a module has been developed and tested, it is useful to test whether it properly fits together with the rest of the system. This too can be done by automation via integration testing. Integration tests try to be a close analog of the eventual production environment but with some obvious differences (simulated direct user interaction, lower-grade hardware, or external resources). Integration testing can be very useful in detecting bugs caused by the software or hardware environment and by configuration. These are often subtle and not caught easily in unit tests. In a healthy program, integration tests should rarely fail.

In rare cases, it is useful to alter injector configuration dynamically. This is analogous to reassembling the modular structure of the application, but done so at runtime. Not all DI libraries support this kind of functionality directly. One mitigant is to provide dependents with all possible dependencies, then force them to decide which to use as appropriate. This works, but it isn't a great solution because it pollutes application logic with infrastructure concerns. As an alternative solution, the adapter pattern works nicely. It encapsulates any infrastructure logic and can be used in place of the original dependency with no impact to client code. A change in the binding is signaled to the adapter(s) via a rebinding notification, and the adapter shifts to the new binding. While this is a robust and workable solution, it is fraught with potential pitfalls and should be weighed carefully before being embarked upon.

Modular code is wonderful for many reasons. As I've already mentioned, these include testing, reuse, and independent development in isolation. It allows many streams of development (and indeed teams of developers) to work on pieces of a very large composite application and keeps complexity to a minimum. Modularity fits very nicely into dependency injection. DI is not only able to wire and assemble modular code quickly, but it is also able to cast it in different profiles (structures) with minimal impact on collaborators. Thus, dependency injection empowers modular programming.

5

Scope: a fresh breath of state

This chapter covers:

- Understanding what scope means
- Understanding singleton and no scope
- Applying practical scopes for the web

“Still this planet’s soil for noble deeds grants scope abounding.”

—Johann Goethe

In one sentence, *scope* is a fixed duration of time or method calls in which an object exists. In other words, a scope is a context under which a given key refers to the same instance. Another way to look at this is to think of scope as the amount of time an object’s state persists. When the scope context ends, any objects bound under that scope are said to be *out of scope* and cannot be injected again in other instances.

State is important in any application. It is used to incrementally build up data or responsibility. State is also often used to track the context of certain processes, for instance, to track objects in the same database *transaction*.

In this chapter we’ll talk about some of the general-purpose scopes: *singleton* and *no scope*. These are scopes that are universally applicable in managing state.

We'll also look at managing state in specific kinds of applications, particularly the web. Managing user-specific state is a major part of scoping for the web, and this is what the *request*, *session*, *flash*, and *conversation* scopes provide. We'll look at a couple of implementations of these with regard to Guice and Spring and how they may be applied in building practical web applications. First, we'll take a primer on scopes.

5.1 What is scope?

The real power of scope is that it lets you model the state of your objects declaratively. By telling the injector that a particular key is bound under a scope, you can move the construction and wiring of objects to the injector's purview. This has some important benefits:

- It lets the injector manage the *latent state* of your objects.
- It ensures that your services get new instances of dependencies as needed.
- It implicitly separates state by context (for example, two HTTP requests imply different contexts).
- It reduces the necessity for *state-aware* application logic (which makes code much easier to test and reason about).

Scope properly applied means that code working in a particular context is oblivious to that context. It is the injector's responsibility to manage these contexts. This means not only that you have an added separation between infrastructure and application logic, but also that the same services can be used for many purposes simply by altering their scopes. Take this example of a family bathroom and its toothpaste:

```
family.give("Joanie", injector.getInstance(Toothpaste.class));
family.give("Jackie", injector.getInstance(Toothpaste.class));
family.give("Sachin", injector.getInstance(Toothpaste.class));
```

Looking at this code we can say that the Toothpaste is used by Joanie first, then by Jackie, and finally by Sachin. We might also guess that each family member receives the same tube of toothpaste. If the tube were especially small, Sachin might be left with no toothpaste at all (as per figure 5.1).

This is an example of context: All three family members use the same bathroom and therefore have access to the same instance of Toothpaste. It is exactly the same as the following program, using construction by hand:

```
Toothpaste toothpaste = new FluorideToothpaste();
family.give("Joanie", toothpaste);
family.give("Jackie", toothpaste);
family.give("Sachin", toothpaste);
```

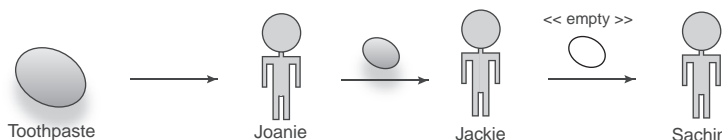


Figure 5.1
The injector distributes the same instance of Toothpaste to all family members.

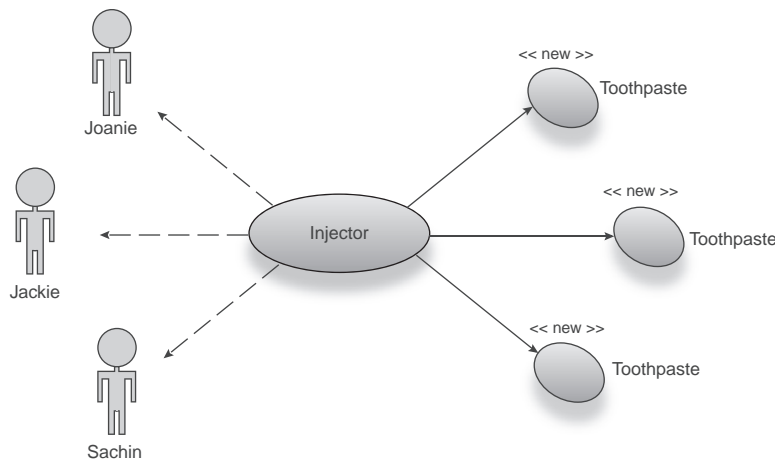


Figure 5.2
The injector creates
a new **Toothpaste**
instance for each
family member.

If this were the whole life of the injector, only one instance of **Toothpaste** would ever be created and used. In other words, **Toothpaste** is bound under *singleton* scope. If each family member had his *own* bathroom (each with its own tube of toothpaste), the semantics would change considerably (figure 5.2).

```
family.give("Joanie", injector.getInstance(Toothpaste.class));
family.give("Jackie", injector.getInstance(Toothpaste.class));
family.give("Sachin", injector.getInstance(Toothpaste.class));
```

Nothing has changed in the code, but now a *new* instance of **Toothpaste** is available to each family member. And now there is no danger of Sachin being deprived of toothpaste by Joanie or Jackie. In this case, the context under which each object operates is unique (that is, its own bathroom). You can think of this as the opposite of singleton scoping. Technically this is like having no scope at all.

5.2 The no scope (or default scope)

In a sense, no scope fulfills the functions of scope, as it

- Provides new instances transparently
- Is managed by the injector
- Associates a service (key) with some context

Or does it? The first two points are indisputable. However, there arises some difficulty in determining exactly what context it represents. To get a better idea of no scope's semantics, let's dissect the example of the toothpaste from earlier. We saw that it took no change in the use of objects to alter their scope. The `family.give()` sequence looks exactly the same for both singleton and no scope:

```
family.give("Joanie", injector.getInstance(Toothpaste.class));
family.give("Jackie", injector.getInstance(Toothpaste.class));
family.give("Sachin", injector.getInstance(Toothpaste.class));
```

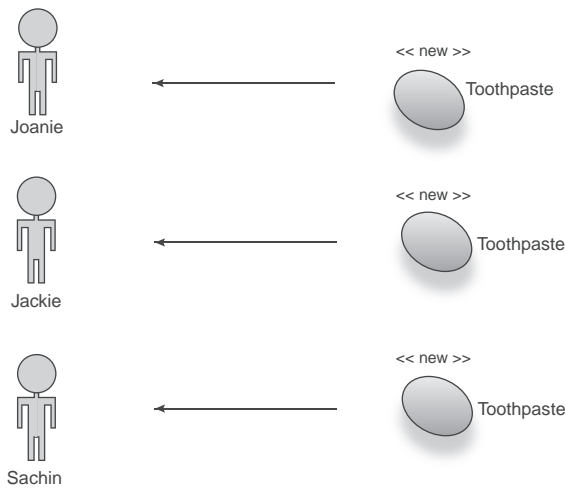


Figure 5.3 Each member of the family receives her own instance of Toothpaste.

Or, expanded using construction by hand (modeled in figure 5.3), the same code can be expressed as follows:

```

Toothpaste toothpaste = new FluorideToothpaste();
family.give("Joanie", toothpaste);

toothpaste = new FluorideToothpaste();
family.give("Jackie", toothpaste);

toothpaste = new FluorideToothpaste();
family.give("Sachin", toothpaste);

```

In no scope, every reference to `Toothpaste` implies a new `Toothpaste` instance. We likened this to the family having three bathrooms, one for each member. However, this is not exactly accurate. For instance, if Sachin brushed his teeth twice,

```

family.give("Joanie", injector.getInstance(Toothpaste.class));
family.give("Jackie", injector.getInstance(Toothpaste.class));
family.give("Sachin", injector.getInstance(Toothpaste.class));
family.give("Sachin", injector.getInstance(Toothpaste.class));

```

we would end up with a total of four `Toothpaste` instances (see figure 5.4).

In our conceptual model, there were only three bathrooms. But in practice there were four tubes of toothpaste. This means that no scope cannot be relied on to adhere to any conceptual context. No scope means that every time an injector looks

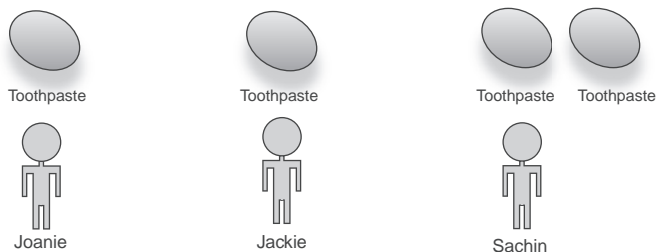


Figure 5.4 There are now four instances of `Toothpaste`, one for each use.

for a given key (one bound under no scope), *it will construct and wire a new instance*. Furthermore, let's say Sachin took Joanie on vacation and only Jackie was left at home. She would brush her teeth once, as follows:

```
family.give("Jackie", injector.getInstance(Toothpaste.class));
```

This would mean only one instance of Toothpaste was ever created for the life of the application. This was exactly what happened with singleton scoping, but this time it was purely accidental that it did. Given these two extremes, it is difficult to lay down any kind of strict rules for context with no scope. You could say, perhaps, that no scope is a split-second scope where the context is entirely tied to referents of a key. This would be a reasonable supposition. Contrast singleton and no scope in figure 5.5.

No scope is a very powerful tool for working with injector-managed components. This is partly because it allows a certain amount of flexibility in scaling upward. Dependents that exist for longer times (or in *wider* scopes) may safely obtain no-scoped objects as they are required. If you recall the Provider pattern from chapter 4, there is a close similarity. Granny obtained new instances of an Apple on each use (see listing 5.1, modeled in figure 5.6).

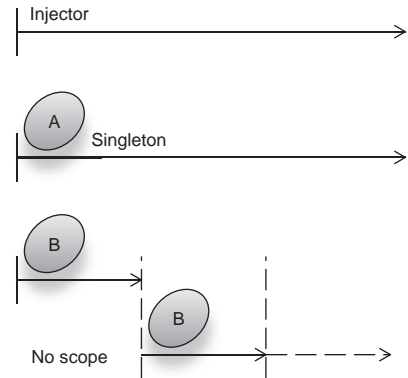


Figure 5.5 Timeline of contexts, contrasting singleton and no scope

Listing 5.1 An example of no scope using the Provider pattern

```
public class Granny {
    private Provider<Apple> appleProvider;

    public void eat() {
        appleProvider.get().consume();
        appleProvider.get().consume();
    }
}
```

Two new Apples created

In listing 5.1, the `eat()` method uses a provider to retrieve new instances, just as we did for Toothpaste, earlier. Here Apple is no scoped.

Guice and Spring differ in nomenclature with regard to the no scope. Spring calls it as the *prototype* scope, the idea being that a key (and binding) is a kind of template

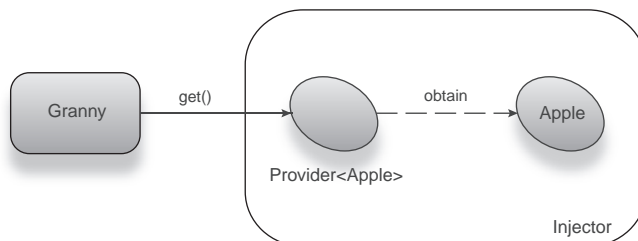


Figure 5.6 Granny obtains instances of Apple (bound to no scope) from a provider.

(or prototype) for creating new objects. Recall chapter 3, in the section on constructor injection and object validity, where no scoping enabled multiple threads to see independent instances of an object (modeled in figure 5.7):

```
<beans ...>
  <bean id="slippery" class="Slippery" scope="prototype"/>
  <bean id="shady" class="Shady" scope="prototype"/>

  <bean id="safe" class="UnsafeObject" init-method="init" scope="prototype">
    <property name="slippery" ref="slippery">
    <property name="shady" ref="shady">
  </bean>
</beans>
```

This object was actually safe, because any dependents of key *safe* were guaranteed to see new, independent instances of `UnsafeObject`. Like singleton, the name prototype comes from the Gang of Four book, *Design Patterns*. For the rest of this book I will continue to refer to it as, largely because it is a more descriptive name.

Like Guice, PicoContainer also assumes the no scope if a key is not explicitly bound to some scope:

```
MutablePicoContainer injector = new DefaultPicoContainer();
injector.addComponent(Toothpaste.class);

family.give("Joanie", injector.getComponent(Toothpaste.class));
family.give("Jackie", injector.getComponent (Toothpaste.class));
...
```

There's almost no difference.

NOTE You will sometimes also hear no scope referred to as the *default* scope. This is a less descriptive name and typically connotes either Guice or PicoContainer (since they *default* to no scope).

While no scope doesn't really lend itself to a context, singleton scope does so quite naturally. Although the design pattern is itself applied in many different ways, we can establish a context quite easily for a singleton.

5.3 The singleton scope

Very simply, a singleton's context is the injector itself. The life of a singleton is tied to the life of the injector (as in figure 5.8).

Therefore, only one instance of a singleton is ever created per injector. It is important to emphasize this last point, since it is possible for

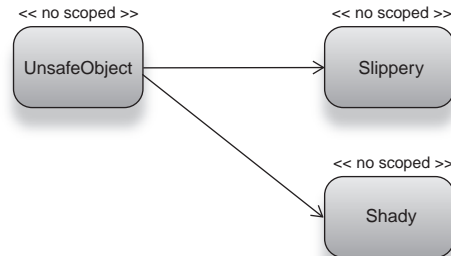


Figure 5.7 `UnsafeObject` and both its dependencies were no scoped.

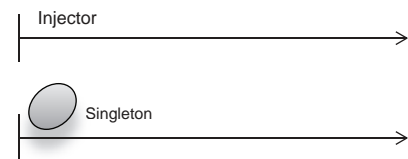


Figure 5.8 Timeline view of an injector and singleton-scoped object A

multiple injectors to exist in the same application. In such a scenario, each injector will hold a different instance of the singleton-scoped object. This is important to understand because many people mistakenly believe that a singleton means one instance for the entire life of an application. In dependency injection, this is not the case. The distinction is subtle and often confusing. In fact, PicoContainer has dropped the term *singleton* and instead refers to it as *cache scoping*.

I persist with *singleton scope*, however, for a few reasons:

- A *singleton-scoped* object is different from a *singleton-patterned* object (more on this shortly).
- The term *singleton* is well known and reasonably well understood, even if its particular semantics are not.
- *Cache scoping* is a different concept altogether (which you will see in the next chapter).

Identifying which service should be a singleton is quite a divisive issue. It is a design decision that should impinge on the nature of a service. If a service represents a conceptual nexus for clients, then it is a likely candidate. For instance, a database *connection pool* is a central port of call for any service that wants to connect to a database (see figure 5.9). Thus, connection pools are good candidates for singleton scoping.

Similarly, services that are stateless (in other words, objects that have no dependencies or whose dependencies are immutable) are good candidates. Since there is no state to manage, no scoping and singleton scoping are both equally viable options. In such cases, the singleton scope has advantages over no scope for a number of reasons:

- Objects can be created at startup (sometimes called *eager instantiation*), saving on construction and wiring overhead.
- There is a single point of reference when stepping through a debugger.
- Maintaining the lone instance saves on memory (memory complexity is *constant* as opposed to *linear* in the case of no scoping; see figure 5.10).

Business objects are perfect candidates for singleton scoping. They hold no state but typically require data-access dependencies. These services are sometimes referred to as

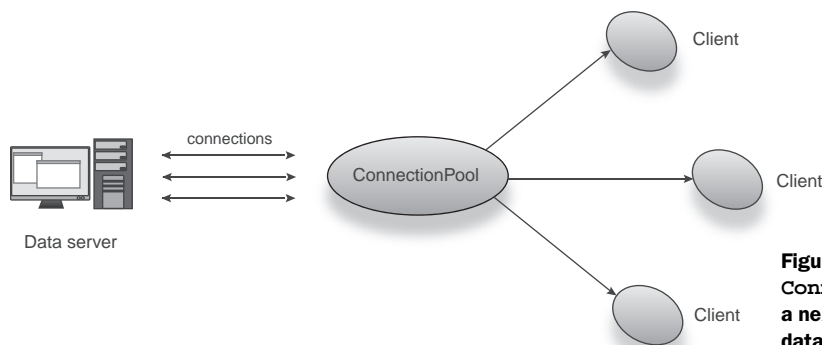


Figure 5.9
ConnectionPool is a nexus through which data connections are served to clients.

managers or simply business APIs, as in `PersonnelManager`, `SalaryManager`, and `StockOptionsService` in a hypothetical employee payroll system. Any services-oriented APIs are likewise good candidates to be stateless singletons. We'll talk a bit more about *services-oriented architecture* (SOA) in chapter 10.

Another oft-stated use case for singletons is when dealing with any object graphs that are expensive to construct and assemble. This may be due to any of the following reasons:

- The object graph itself being very large
- Reliance on slow external resources (like network storage)
- Some difficult computation performed after construction

These aren't good enough reasons in themselves to warrant singleton scoping. Instead you should be asking questions about context. If an external resource is designed to be held open over a long period, then yes, it may warrant singleton scoping, for example, the pool of database connections.

On the other hand, a TCP socket, while potentially expensive to acquire (and cheap to hold onto) may not warrant singleton scoping. If you were writing a game that logs in to a server over a TCP connection when playing against others, you certainly would not want it to be a singleton. If a user happened to log out and back in to a new server in a different network location, a new context for the network service would be established and consequently would need a new instance, not the old singleton instance.

Similarly, the size of the object graph should not play a major role in deciding an object's scope. Object graphs that have several dependencies, which themselves have dependencies, and so on, are not necessarily expensive to construct and assemble. They may have several cheaply buildable parts. So without seeing clear shortcomings in an object's performance, don't be in a rush to optimize it as a singleton. This is a mantra you can repeat to yourself every time you start to speculate and worry about performance in any context.

Computational expense may be a legitimate reason for scoping an object as a singleton, but here too, there are other prevailing concerns. Is the computation a one-time activity? Do its values never change throughout the life of the application? And can you separate the expense of computation from the object itself by, for instance, storing the resultant value in a *memo object*? If you can answer these questions, you may have a singleton on your hands. Remember the singleton litmus test: Should the object exist for the same duration as the injector? In the coming sections, we'll look at how the semantics of the singleton scope affect design in various situations.

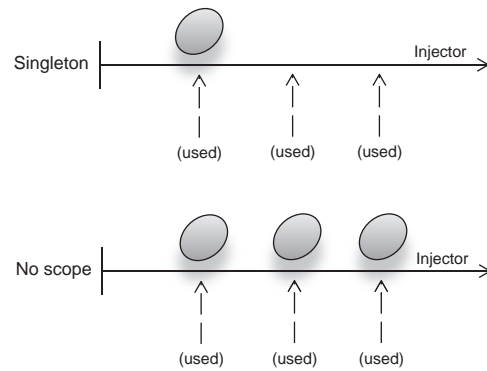


Figure 5.10 Memory usage for a singleton is constant compared to linear for no-scoped instances.

Algorithmic (or asymptotic) complexity

Complexity in computing is a measure of the scalability of an algorithm with regard to resources. These resources may be CPU cycles (time complexity), RAM (memory complexity), or any other kind of physical resource that the algorithm may demand (for instance, network bandwidth). Complexity is used to indicate how the algorithm performs with increasing size of input. The input is typically expressed as a number n , indicating the total number of input items. For instance, a text-search algorithm may count the number of characters in a string as its input.

Performance is typically expressed in big Oh notation: $O(n)$ or $O(n^2)$, and so on. Everything is relative to n ; the idea is to measure how an algorithm scales for very large values of n . Constant complexity, which is $O(1)$, indicates that the algorithm is independent of n , even for huge values of n . This is considered good because it means the algorithm will only ever consume a fixed amount of resources, regardless of its input.

Going back to the text-search algorithm, it is easy to see that its time complexity is dependent on the size of the input (since every character needs to be searched). This is called *linear complexity* and is usually written as $O(n)$.

In my example of no scoping, the input items are the number of *dependents* of the no-scoped dependency, and the amount of memory allocated scales in proportion to this number. Therefore, its memory complexity is also linear. If I changed to singleton scoping, the memory complexity would be constant, since only the one instance is created and shared by all dependents.

5.3.1 Singletons in practice

The important thing to keep in mind about scoping is that a key is bound under a scope, not an object or class. This is true of any scope, not just the singleton. Take the following example of a master terminal that can see several security cameras in a building:

```
<bean id="terminal" class="MasterTerminal" scope="singleton"/>
<bean id="camera.basement" class="SimpleCamera" scope="prototype">
  <constructor-arg ref="terminal"/>
</bean>
<bean id="camera.penthouse" class="SimpleCamera" scope="prototype">
  <constructor-arg ref="terminal"/>
</bean>
```

Notice that I explicitly declare `terminal` as a singleton (by default, Spring beans are all singletons) and both cameras as no scoped (`scope="prototype"`). In this configuration, both `camera.basement` and `camera.penthouse` share the same instance of `MasterTerminal`. Any further keys will also share this instance. Consider this equivalent in Guice, using a module:

```

public class BuildingModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(Camera.class).annotatedWith(Basement.class).
        ➡ to(SimpleCamera.class);
        bind(Camera.class).annotatedWith(Penthouse.class).
        ➡ to(SimpleCamera.class);

        bind(MasterTerminal.class).in(Singleton.class);
    }
}

```

Here we use combinatorial keys (see chapter 2) to identify the basement and penthouse security cameras. Since Guice binds keys under no scope by default, we need only scope `MasterTerminal`:

```
bind(MasterTerminal.class).in(Singleton.class);
```

This has the same semantics as the Spring configuration. All security camera instances share the same instance of `MasterTerminal`, as shown in figure 5.11.

Another option is to directly annotate `MasterTerminal` as a singleton:

```

@Singleton
public class MasterTerminal { .. }

public class BuildingModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(Camera.class).annotatedWith(Basement.class).
        ➡ to(SimpleCamera.class);
        bind(Camera.class).annotatedWith(Penthouse.class).
        ➡ to(SimpleCamera.class);
    }
}

```

This lets you skip an explicit binding (see `BuildingModule`). Figure 5.12 shows how a singleton-scoped instance is shared among no-scoped objects.

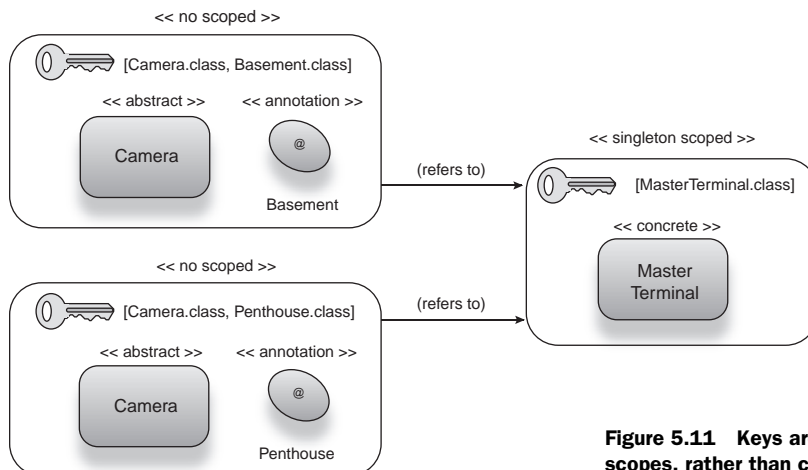


Figure 5.11 Keys are bound under scopes, rather than classes or objects

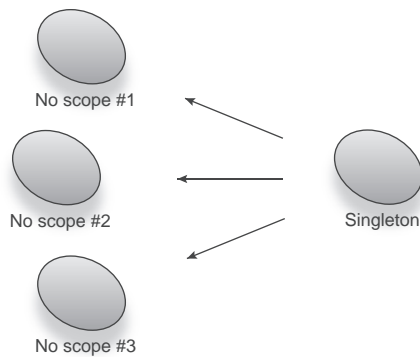


Figure 5.12 A singleton-scoped instance injected into many no-scoped instances

To further illustrate the difference between no scope and singleton scope, let's examine another interesting case. Here is a class `BasementFloor`, which represents a part of the building where security cameras may be installed:

```
public class BasementFloor {
    private final Camera camera1;
    private final Camera camera2;

    @Inject
    public BasementFloor(@Basement Camera camera1,
                        @Basement Camera camera2) {

        this.camera1 = camera1;
        this.camera2 = camera2;
    }
}
```

You might expect that both `camera1` and `camera2` refer to the same instance of security camera, that is, the one identified by key `[Camera, Basement]`. But this is not what happens—`camera1` and `camera2` end up with two different instances of `Camera`. This is because the key `[Camera, Basement]` is bound to no scope (see figure 5.13).

Similarly, any dependents of `[Camera, Penthouse]` will end up with new, unrelated instances of `Camera`. Consider another class, `Building`, which houses more than one kind of camera:

```
public class Building {
    private final Camera camera1;
    private final Camera camera2;
    private final Camera camera3;
    private final Camera camera4;

    @Inject
    public Building(@Basement Camera camera1,
```

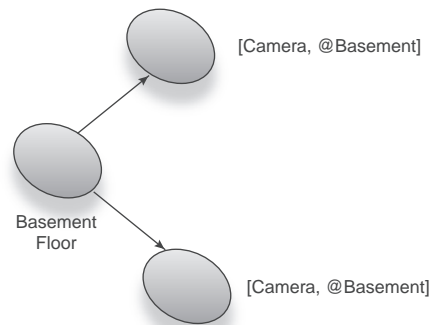


Figure 5.13 `BasementFloor` is wired with two separate instances of no-scoped key `[Camera, @Basement]`.

```

        @Basement Camera camera2,
        @Penthouse Camera camera3,
        @Penthouse Camera camera4) {

    this.camera1 = camera1;
    this.camera2 = camera2;
    this.camera3 = camera3;
    this.camera4 = camera4;
}
}

```

Here, all four fields of `Building` receive different instances of `Camera` even though only two keys are present. This is opposed to the following class, `ControlRoom`, which has four fields that refer to the same instance of `MasterTerminal` but via four references (modeled in figure 5.14):

```

public class ControlRoom {
    private final MasterTerminal terminal1;
    private final MasterTerminal terminal2;
    private final MasterTerminal terminal3;
    private final MasterTerminal terminal4;

    @Inject
    public ControlRoom(MasterTerminal terminal1,
                      MasterTerminal terminal2,
                      MasterTerminal terminal3,
                      MasterTerminal terminal4) {

        this.terminal1 = terminal1;
        this.terminal2 = terminal2;
        this.terminal3 = terminal3;
        this.terminal4 = terminal4;
    }
}

```

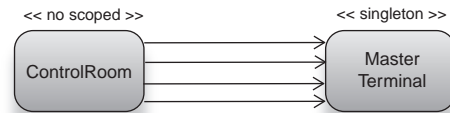


Figure 5.14 All four fields of `ControlRoom` are wired with the same instance of `MasterTerminal`.

If we added a new kind of `MasterTerminal`, bound to a different key, then this would be a different instance. Let's say I add a `MasterTerminal` for the basement only:

```

public class BuildingModule extends AbstractModule {

    @Override
    protected void configure() {
        bind(Camera.class).annotatedWith(Basement.class).
            ➡ to(SimpleCamera.class);
        bind(Camera.class).annotatedWith(Penthouse.class).
            ➡ to(SimpleCamera.class);

        bind(MasterTerminal.class).annotatedWith(Master.class).in(Singleton.class);

        bind(MasterTerminal.class).annotatedWith(Basement.class)
            .to(BasementTerminal.class)
            .in(Singleton.class);
    }
}

```

Now any dependents of [MasterTerminal, Master] see the same shared instance, but any dependents of key [MasterTerminal, Basement] see a different instance. The modified version of Building in listing 5.2 illustrates this situation.

Listing 5.2 Object with two different singleton-scoped dependencies

```
public class ControlRoom {
    private final MasterTerminal terminal1;
    private final MasterTerminal terminal2;
    private final MasterTerminal terminal3;
    private final MasterTerminal terminal4;

    @Inject
    public ControlRoom(@Master MasterTerminal terminal1,
                      @Master MasterTerminal terminal2,
                      @Basement MasterTerminal terminal3,
                      @Basement MasterTerminal terminal4) {

        this.terminal1 = terminal1;
        this.terminal2 = terminal2;
        this.terminal3 = terminal3;
        this.terminal4 = terminal4;
    }
}
```

Shared instance of
[MasterTerminal, Master]

Shared instance of
[MasterTerminal, Basement]

In listing 5.2, terminal1 and terminal2 share the same instance of MasterTerminal. But terminal3 and terminal4 share a different instance. I belabor this point because it is very important to distinguish that a key, rather than an object, is bound under a scope. Singleton scoping allows many instances of the same class to exist and be used by dependents; it allows only one shared instance of a key. This is quite different from the conventional understanding of singletons. This idiom implies a more absolute single instance per application and is definitely problematic. In the following section I'll explain why I go so far as to call it an anti-pattern when compared with the much more erudite singleton scope.

5.3.2 The singleton anti-pattern

You have probably heard a lot of discussion around the web and in technical seminars about the horrors of the singleton as an anti-pattern. Earlier we drew a distinction between *singleton scope*, a feature of DI, and *singleton objects* (or *singleton anti-patterns*), which are the focus of much of this debate. The singleton anti-pattern has several problems. Compounding these problems is the fact that singletons are very useful and therefore employed liberally by developers everywhere.

Its problems however, greatly outweigh its usefulness and should warn you off them for good (especially when dependency injection can save the day with singleton scoping). Let's break this down in code:

```
public class Console {
    private static Console instance = null;

    public static synchronized Console getInstance() {
```

```

        if (null == instance)
            instance = new Console();

        return instance;
    }

    ...
}

```

I'm sure you have seen or written code like this. I certainly have. Its purpose is quite simple; it allows only one instance of `Console` ever to be created for the life of the program (representing the one monitor in a computer, for example). If an instance does not yet exist, it creates one and stores it in a static variable `instance`:

```

    if (null == instance)
        instance = new Console();

    return instance;

```

The static `getInstance()` method is declared `synchronized` so that concurrent threads don't accidentally attempt to create instances concurrently. In essence, `getInstance()` is a Factory method—but a special type of Factory that produces only one instance, thereafter returning the stored instance every time. You might say this is the Factory equivalent of singleton scoping.

Apart from all the foibles that Factories bring, note that this immediately causes one major problem. This code is not conducive to testing. If an object relies on `Console.getInstance()` to retrieve a `Console` instance and print something to it, there is no way for us to write a unit test that verifies this behavior. We cannot pass in a substitute `Console` in the following code:

```

public class StockTicker {
    private Console console = Console.getInstance();

    public void tick() {
        //print to console
        ...
    }
}

```

`StockTicker` directly retrieves its dependency from the singleton Factory method. In order to test it with a mock `Console`, you'd have to rewrite `Console` to expose a setter method:

```

public class Console {
    private static Console instance = null;

    public static synchronized Console getInstance() {
        if (null == instance)
            instance = new Console();

        return instance;
    }

    public static synchronized void setInstance(Console console) {
        instance = console;
    }
}

```

```

    }
    ...
}

```

Patterns that force you to add extra code or infrastructure logic purely for the purposes of testing are poor servants of good design. Nonetheless, now you can test `StockTicker`:

```

public class StockTickerTest {

    @Test
    public final void printToConsole() {
        Console.setInstance(new MockConsole());
        ...
    }
}

```

But what if there are other tests that need to create their *own* mocked consoles for different purposes (say, a file listing service)? Leaving the mocked instance in place will clobber those tests. To fix that, we have to change the test again:

```

public class StockTickerTest {

    @Test
    public final void printToConsole() {
        Console previous = Console.getInstance();
        try {
            Console.setInstance(new MockConsole());
            ...
        } finally {
            Console.setInstance(previous);
        }
    }
}

```

I wrapped the entire test in a `try/finally` block to ensure that any exceptions thrown by the test do not subvert the `Console` reset.

TIP Depending on your choice of test framework, there may be other methods of doing this. I like to use *TestNG*,¹ which allows the declaration of setup and teardown methods that run before and after each test. See listing 5.3.

Listing 5.3 A test written in TestNG with setup and teardown hooks

```

public class StockTickerTest {
    private Console previous;

    @BeforeMethod
    void setup() {

```

¹ TestNG is a flexible Java testing framework created by Cedric Beust and others. It takes many of the ideas in JUnit and improves on them. Find out more about TestNG at <http://www.testng.org> and read Cedric's blog at <http://beust.com/weblog>.


```

        previous = Console.getInstance();
    }

    @Test
    public final void printToConsole() {
        Console.setInstance(new MockConsole());
        ...
    }

    @AfterMethod
    void teardown() {
        Console.setInstance(previous);
    }
}

```

This is a lot of boilerplate to write just to get tests working. It gets worse if you have more than one singleton dependency to mock. Furthermore, if you have many tests running in parallel (in multiple threads), this code doesn't work at all because threads may crisscross and interfere with the singleton. That would make all your tests completely unreliable. This ought to be a showstopper.

If you have more than one injector in an application, the situation grows worse. Singleton objects are shared even *between* injectors and can cause a real headache if you are trying to separate modules by walling them off in their own injectors (see figure 5.15).

Moreover, any object created and maintained outside an injector does not benefit from its other powerful features, particularly lifecycle and interception—and the Hollywood Principle. One of the great benefits of DI is that it allows you to quickly bind a key to a different scope simply by changing a line of configuration. That's not possible with the singleton object (figure 5.16).

Its class must be rewritten and retested to introduce scoping semantics. Refactoring between scopes is also a vital part of software development and emerging design. Singleton objects hinder this natural, iterative evolution.

So to sum up: singleton objects bad, singleton scoping good. Singleton objects make testing difficult if not impossible and are antithetical to good design with dependency injection. Singleton scoping, on the other hand, is a purely injector-driven feature and completely removed from the class in question (figure 5.17).

Singleton scope is thus flexible and grants the usefulness of the Singleton anti-pattern without all of its nasty problems. Scopes have a variety of other uses; they

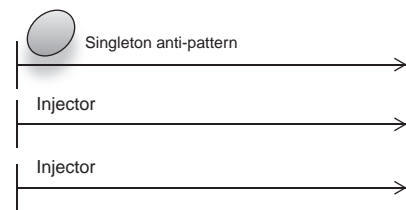


Figure 5.15 Singleton-patterned objects are shared even across injectors.

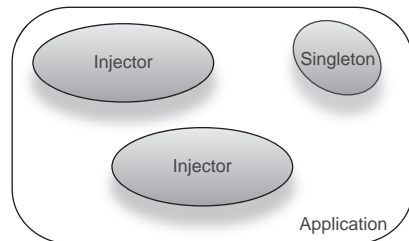


Figure 5.16 Singleton anti-pattern objects sit outside dependency injectors.

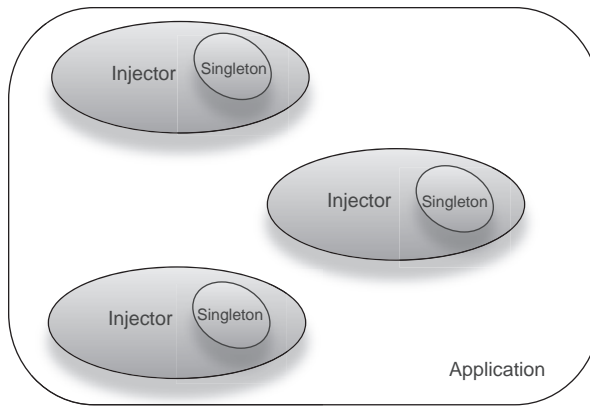


Figure 5.17 Singleton-scoped objects are well-behaved and live inside the injector.

needn't only be applied in the singleton and no scope idioms. In web applications, scopes are extremely useful as unintrusive techniques in maintaining user-specific state. These web-specific scopes are explored in the following section.

5.4 Domain-specific scopes: the web

So far we've seen that scopes are a context in which objects live. When a scope ends, objects bound to it go out of scope. And when it begins again, new objects (of the same keys) come "into scope." Scopes can also be thought of as a period in the objective life of a program where the state of objects is *persistent*; that is, it retains its values for that period. We also examined two of these scopes: the singleton and the no scope. They're rather unique: One sticks around forever, the other a split second. In a sense, these two scopes are universal. Any application has a use for no-scoped objects. And most applications will probably need access to some long-lived service that the singleton context provides.

But there is a whole class of uses that are very specific to a particular problem domain. These are *domain-specific* scopes. They are contexts defined according to the particular behavioral requirements of an application. For example, a movie theater has a specific context for each movie as it is being shown. In a multiplex, several movies may be showing simultaneously, and each of these is a unique context. We can model these contexts as scopes in an injector. A moviegoer watching a showing of the movie *The Princess Bride* is different from one who is watching *Godzilla* in another theater.

An important corollary to this model is that the moviegoer exists for the entire duration of the movie (that is, scope). So if you looked in on the *Godzilla* show, you would expect to see the same members of the audience each time.

The movie scopes are specific to the domain of movie theaters and intimately tied with their semantics. If a moviegoer exits one show just as it is ending and enters another show as it is starting (I used to do this in high school to save money), does it mean the moviegoer is carried across two contexts? Or should its state be lost and a new instance created? We can't answer these questions without getting deeper into

the movie theater analogy. More important, the answers to the questions can't be reused in any other problem domain.

One of the most important sets of domain-specific scopes is those around building *web* applications. Web applications have different contexts that emerge from interaction with users. Essentially, web applications are elaborate document retrieval and transmission systems. These are typically HTML documents, which I am sure you are infinitely familiar with. With the evolution of the web, highly interactive web applications have also arisen—to the point where they now closely resemble desktop applications.

However, the basic protocol for transmission of data between a browser and server has remained fairly unchanged. The contexts for a web application have their semantics in this protocol. Unlike a desktop application, a web application will generally have many users simultaneously accessing it. And these users may enter and leave a chain of communications with the web application at will. For example, when checking email via the popular Gmail service from Google, I sign in first (translated as a request for the inbox document), then open a few unread messages (translated as requests for HTML documents), and finally sign out. All this happens with Gmail running constantly on Google's servers. Contrast this with a desktop client like Mozilla Thunderbird, where I perform the same three steps, but they result in the program starting up from scratch and terminating when I've finished (entirely on my desktop). Furthermore, Google's servers host thousands (if not millions) of users doing very similar things simultaneously. Any service that would normally be singleton scoped, for example, my user credentials in Thunderbird, can no longer be a singleton in Gmail.² Otherwise, all users would share the same credentials, and we would be able to read each other's mail.

On the other hand, you can't make everything no scoped either. If you do, classes of the Gmail application would be forced to keep passing around user credentials in order to maintain state in a given request, and that would rewind all the benefits originally reaped from scoping. It would also perform rather poorly. Here's where web-specific scopes really help.

All interaction between a user (via a web browser) and a web application occurs inside an *HTTP request* (figure 5.18).

A *request* is—no surprise—a request for a document or, more generally, a resource. To provide this resource, an application might go to a database or a file or perform some *dynamic* computation. All this happens synchronously, that is, while the user is waiting. This entire transaction forms one HTTP request. Objects that live and operate within this context belong to the *request scope*.

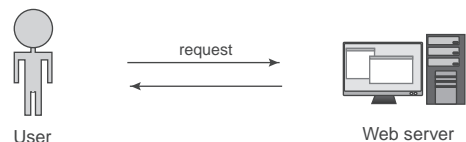


Figure 5.18 Interaction between a web server and a user happens in an HTTP request.

² Interesting fact: Gmail's frontend servers use Guice heavily for dependency injection concerns.

5.4.1 HTTP request scope

The request scope is interesting because it isn't strictly a segment in the life of an injector. Since requests can be concurrent, request scopes are also concurrent (but disparate from one another). Let's take the example of an online comic book store. *Sandman* is one of my favorite books, so I do a search for *Sandman* by typing in "sandman" at the appropriate page. To the comics store, this is a request for a document containing a list of *Sandman* titles. Let's call the service that generates this document `ComicSearch`. `ComicSearch` must

- Read my search criteria from the incoming request
- Query a database
- Render a list of results

Let's make another service that accesses the database and call this `ComicAccess`. `ComicAccess` will in turn depend on database-specific services like connections and statements in order to deliver results. Listing 5.4 describes the code for the comic store so far.

Listing 5.4 Comic store's search and data access components

```
public class ComicSearch {  
    private final ComicAccess comicAccess;  
  
    @Inject  
    public ComicSearch(ComicAccess comicAccess) {  
        this.comicAccess = comicAccess;  
    }  
  
    public HTML searchForComics(String criteria) {  
        ...  
    }  
}
```

Now it makes sense for the `ComicAccess` object to be a singleton—it has no state, connects to a database as needed, and will probably be shared by several clients (other web pages needing to access the comic store). The `searchForComics()` method takes search criteria (typed in by a user) and returns an HTML object.

NOTE Of course, this is a hypothetical class and the exact form of `searchForComics()` may be different depending on the web framework you choose. But its semantics remain the same—it takes in search criteria and converts them to an HTML page displaying a list of matches.

`ComicSearch` is itself stateless (since its only dependency, `ComicAccess`, is immutable), so we could bind it as a singleton. Given this case, it actually works quite well. Since there is no *request-specific* context, binding `ComicSearch` either as a singleton or no scope is viable.

Let's expand this example. Let's say we add a requirement that the store shows me items of interest based on my prior purchases. It's not important how it determines my interests, just that it does. Another service, `HttpContext`, will handle this work:

```

public class UserContext {
    private String username;

    private final ComicAccess comicAccess;

    @Inject
    public UserContext(ComicAccess comicAccess) {
        this.comicAccess = comicAccess;
    }

    public List<Comic> getItemsOfInterest() {
        ...
    }

    public void signIn(String username) {
        this.username = username;
    }
}

```

The method `getItemsOfInterest()` scans old purchases using `ComicAccess` and builds a list of suggestions. The interesting part about `UserContext` is its field `username` and method `signIn()`:

```

    public void signIn(String username) {
        this.username = username;
    }

```

When called with an argument, `signIn()` stores the *current* user. You'll notice that `signIn()` is more or less a setter method. But I've deliberately avoided calling it `setUsername()` to distinguish it from a dependency setter. `signIn()` will be called from `ComicSearch`, which itself is triggered by user interaction. Here's the modified code from listing 5.4, reflecting the change:

```

public class ComicSearch {
    private final ComicAccess comicAccess;
    private final UserContext user;

    @Inject
    public ComicSearch(ComicAccess comicAccess, UserContext user) {
        this.comicAccess = comicAccess;
        this.user = user;
    }

    public HTML searchForComics(String criteria, String username) {
        user.signIn(username);

        List<Comic> suggestions = user.getItemsOfInterest();
        ...
    }
}

```

`ComicSearch` is triggered on a request from a user, and method `searchForComics()` is provided with an additional argument, `username`, also extracted from the HTTP request. The `UserContext` object is configured with this `username`. Now any results it returns will be specific to the current user.

Let's put the `UserContext` to work and expand this another step. We'll add a requirement that the list of results should *not* show any comics that a user has already

purchased. One way to do this is with two queries, one with the entire set of results and the second with a history of purchases, displaying only the difference. That sequence would be:

- 1 Query all matches for criteria from database.
- 2 Query history of purchases for current user.
- 3 Iterate every item in step 1, comparing them to every item in step 2, and remove any matches.
- 4 Display the remainder.

This works, but it seems awfully complex. It is a lot of code to write and a bit superfluous. Furthermore, if these sets are reasonably large and contain a lot of overlap, it could mean doing a large amount of work to bring up results that are simply thrown away. Worse, two queries are two trips to the database, which is expensive and unnecessary in a high-traffic environment.

Another solution is to create a special *finder* method on `ComicAccess` that accepts the username and builds a database query sensitive to this problem. This is much better, because only the relevant results come back and the impact to `ComicSearch` is very small:

```
public HTML searchForComics(String criteria, String username) {
    user.signIn(username);

    List<Comic> suggestions = user.getItemsOfInterest();

    List<Comic> results = comicAccess.searchNoPriorPurchases(criteria,
    ➡ username);
    ...
}
```

But we can do one better. By moving this work off to `UserContext`, it avoids `ComicSearch` having to know the requisite details for querying comics:

```
public class UserContext {
    private String username;

    private final ComicAccess comicAccess;

    @Inject
    public UserContext(ComicAccess comicAccess) {
        this.comicAccess = comicAccess;
    }

    public List<Comic> getItemsOfInterest() {
        ...
    }

    public List<Comic> searchComics(String criteria) {
        return comicAccess.searchNoPriorPurchases(criteria, username);
    }

    public void signIn(String username) {
        this.username = username;
    }
}
```

Now `ComicSearch` is a lot simpler:

```
public HTML searchForComics(String criteria, String username) {
    user.signIn(username);

    List<Comic> suggestions = user.getItemsOfInterest();

    List<Comic> results = user.searchComics(criteria);
    ...
}
```

The real saving comes with request scoping. We already need to bind `ComicSearch` and `UserContext` in the request scope (because their state is tied to a single user's request). Now let's say that instead of signing in a user in the `searchForComics()` method, we're able to do it at the beginning of a request before the `ComicSearch` page (say, in a servlet *filter*). This is important because it means that authentication logic is separated from business logic. Furthermore, it means code to sign in the user need be written only once and won't have to litter every page:

```
public class ComicSearch {
    private final UserContext user;

    @Inject
    public ComicSearch(UserContext user) {
        this.user = user;
    }

    public HTML searchForComics(String criteria) {
        List<Comic> suggestions = user.getItemsOfInterest();
        List<Comic> results = user.searchComics(criteria);
        ...
    }
}
```

Notice that it is much leaner now and focused on its core purpose. But where has the code for signing in a user gone? Here's one possible way it may have disappeared.

REQUEST SCOPING IN GUICE WITH GUICE-SERVLET

Listing 5.5 shows one implementation using a Java servlet filter and the `guice-servlet` extension library for Guice.

Guice Servlet and Guice

Guice servlet is an extension to Guice that provides a lot of web-specific functionality, including web-domain scopes (request, session). Guice servlet is registered in `web.xml` as a filter itself and then later configured using a Guice module. It allows you to manage and intercept servlets or filters via Guice's injector (which is not otherwise possible).

Find out more about `guice-servlet` at <http://code.google.com/p/google-guice>.

Listing 5.5 An injector-managed servlet filter using guice-servlet (using Guice)

```
import javax.servlet.Filter;

@Singleton
public class UserFilter implements Filter {
    private final Provider<UserContext> currentUser;

    @Inject
    public UserFilter(Provider<UserContext> currentUser) {
        this.currentUser = currentUser;
    }

    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {

        currentUser.get().signIn(...);    ← Set up current user

        chain.doFilter(request, response); ← Continue processing
                                           request
    }
    ...
}
```

There are some interesting things to say about listing 5.5:

- UserFilter is a servlet filter applied at the head of every incoming request.
- It is declared under singleton scope (note the @Singleton annotation).
- It is injected with a Provider<UserContext> so that a *request-scoped* UserContext may be obtained each time.
- User credentials are extracted from the request and set on the current UserContext.

I use a Provider<UserContext> instead of directly injecting a UserContext because UserFilter is a singleton, and once a singleton is wired with any object, that object gets held onto despite its scope. This is known as *scope-widening injection* and is a problem that I discuss in some detail in the next chapter.

Interestingly enough, in listing 5.4 I was able to use constructor injection to get hold of the UserContext provider:

```
@Inject
public UserFilter(Provider<UserContext> currentUser) {
    this.currentUser = currentUser;
}
```

Ordinarily, this wouldn't be possible for a filter registered in web.xml according to the Java Servlet Specification. However, guice-servlet gets around this by sitting between Java servlets and the Guice injector. Listing 5.6 shows how this is done, with a web.xml that is configured to use guice-servlet.

Listing 5.6 web.xml configured with guice-servlet and Guice

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"

↓ web.xml namespace
  boilerplate
```



```

xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" >

<listener>
  <listener-class>example.MyGuiceCreator</listener-class>
</listener>

<filter>
  <filter-name>guiceFilter</filter-name>
  <filter-class>com.google.inject.servlet.GuiceFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>guiceFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
</web-app>

```

web.xml namespace boilerplate

Listener creates injector on web app deploy

Filter all URLs through guice-servlet

Guice-servlet's architecture is depicted in figure 5.19.

Notice that in listing 5.6 a servlet context listener named `MyGuiceCreator` is registered. This is a simple class that you create to handle the job of bootstrapping the injector. It is where you tell guice-servlet what filters and servlets you want the Guice injector to manage. Here's what a `MyGuiceCreator` would look like if it were configured to filter all incoming requests with `UserFilter` (to do our authentication work):

```

public class MyGuiceCreator extends GuiceServletContextListener {
    @Override
    protected Injector getInjector() {
        return Guice.createInjector(new ServletModule() {
    @Override
        protected void configureServlets() {
            filter("/*").through(UserFilter.class)
        }
    });
}

```

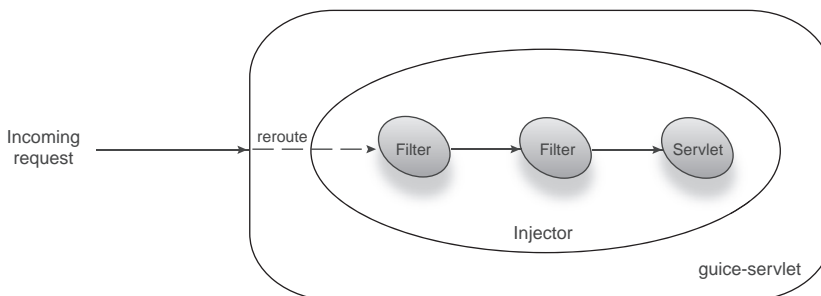


Figure 5.19 Incoming requests are rerouted by guice-servlet to injector-managed filters or servlets.

This code is self-explanatory, but let's go through it anyway. Remember, `MyGuiceCreator` is a class you provide to bootstrap and configure the injector (it must extend `GuiceServletContextListener`). The factory method `Guice.createInjector()` takes instances of Guice's `Module` as argument. You configure filters in `guice-servlet` via a programmatic API (rather than `web.xml`):

```
filter("/*").through(UserFilter.class)
```

You could continue adding filters and servlets. Each servlet and filter must be annotated `@Singleton`.

Let's get back to request scoping. By moving code out to the `UserContext` object, every request receives its own instance of `ComicSearch` and `UserContext`. We were able to achieve this transition without any impact to `ComicAccess` and minimal impact to `ComicSearch`. Declarative scoping of objects is thus a very powerful and unintrusive technique.

This is all very well. But how do we actually bind these scopes in Spring, Guice, and others? Let's take a look:

```
import com.google.inject.servlet.RequestScoped;

public class ComicStoreModule extends AbstractModule {

    @Override
    protected void configure() {

        bind(ComicAccess.class).to(ComicAccessImpl.class).in(Singleton.class);

        bind(UserContext.class).in(RequestScoped.class);
        bind(ComicSearch.class).in(RequestScoped.class);
        ...
    }
}
```

In the example code, I've bound both `UserContext` and `ComicSearch` in `@RequestScoped`. This is a scope made available by `guice-servlet` and represents the HTTP request scope in the world of Guice.³ `ComicAccess` is a simple singleton, and so it is a straightforward binding (to its implementation, `ComicAccessImpl`):

```
bind(ComicAccess.class).to(ComicAccessImpl.class).in(Singleton.class);
```

This same effect can be achieved in Spring with its own set of web-specific scoping utilities. The following section explores the techniques involved there.

REQUEST SCOPING IN SPRING

In Spring's XML configuration mode, these bindings are slightly different (see listing 5.7).

Listing 5.7 Spring XML configuration for the online comic store's components

```
<beans ...>
  <bean id="data.comics" class="ComicAccessImpl" scope="singleton">
```

³ Don't forget that you need to register `guice-servlet`'s `GuiceFilter` in `web.xml`, as shown previously.

```

...
</bean>

<bean id="web.user" class="UserContext" scope="request">
    <constructor-arg ref="data.comics"/>
</bean>

<bean id="web.comicSearch" class="ComicSearch" scope="request">
    <constructor-arg ref="data.comics"/>
    <constructor-arg ref="web.user"/>
</bean>
</beans>

```

The only new thing in listing 5.7 is the attribute `scope="request"` on bindings `web.user` and `web.comicSearch`. Of course, none of the three classes need change at all. Like Guice and `guice-servlet`, Spring requires additional configuration in `web.xml`. First off, you need to bootstrap a Spring injector when the web application is deployed. In Guice we used a `ServletContextListener` (recall `MyGuiceCreator`, the subclass of `GuiceServletContextListener`). You do this in Spring too:

```

<web-app ...>
    ...
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/comic-store.xml</param-value>
    </context-param>

    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    ...
</web-app>

```

This `web.xml` is similar to the one we saw earlier with `guice-servlet`. Notice that we needed to set a context parameter with the name of the XML configuration file. You can think of it as the web equivalent of the following:

```

BeanFactory injector = new FileSystemXmlApplicationContext("WEB-INF/comic-
➡ store.xml");

```

Now we've bootstrapped the injector and told it where to find its configuration. But that's not all; as `guice-servlet` did for Guice, there's still an integration layer that needs to be configured to get request scopes going:

```

<web-app ...>
    ...
    <listener>
        <listener-class>
            org.springframework.web.context.request.RequestContextListener
        </listener-class>
    </listener>
    ...
</web-app>

```

This listener must appear in addition to the `ContextLoaderListener` shown previously. Now you're set up to shop comics with Spring.

TIP We haven't quite looked at how to configure filters with Spring's injector. Guice-servlet handled this for us via its `GuiceFilter`. Spring does not have this support out of the box. But a sister project, *Spring Security* (formerly *Acegi Security*⁴) does. Spring Security's `FilterToBeanProxy` does a similar job.

Apart from these practical aspects, there are things to keep in mind about request scoping:

- A thread is typically dedicated to a request for the request's entire span. This means that request-scoped objects are not multithreaded.
- It also means that they are generally not thread-safe.
- Integration layers that provide request scoping often cache scoped objects in thread-locals.
- In rare cases, web servers may use multiple threads to service a request (for instance, while processing long-running asynchronous requests). If you are designing a request-scoping library in such a scenario, be aware of thread-local assumptions.

Most of these are pretty low-level and specific to the architecture in question. In the Java servlet world, threads and requests are almost always the same thing (though once completed, a thread may clear its context and proceed to service other requests). So you should be careful to clean up at the end of a request. If you are *designing* request scopes, you should carefully research these potential hazards. Perhaps even more useful than the request scope is the HTTP session scope. This technique allows you to keep objects around between requests in a semantic user session. Using dependency injection, a lot of the glue code to make this happen goes away. Session scoping is thus a powerful tool in the dependency injector's toolbox.

5.4.2 HTTP session scope

An HTTP session scope is the next step up from a request scope. HTTP sessions are an abstraction invented to make up for the fact that the HTTP protocol is stateless. This means that it does not easily allow for long-running interactions with a user to be maintained on the server side. To get around this, developers use clever techniques and string together a series of requests from the same user and call it a session (figure 5.20). An HTTP session has important characteristics:

- A session represents a single, unique user's interaction with a web application.
- A session is composed of one or more requests from the same user.
- Not all requests are necessarily part of a session.
- A session is a kind of store that preserves state between requests.
- The two logical end points of a session are user login and logout.

⁴ Find out about Acegi Security at <http://www.acegisecurity.org>.

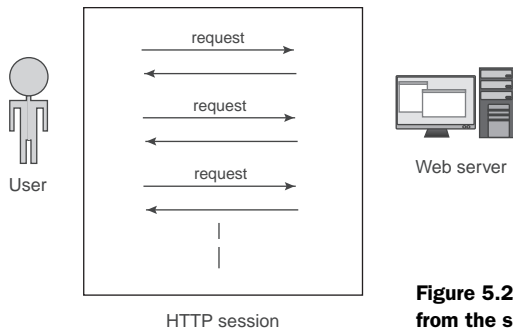


Figure 5.20 A series of related requests from the same user forms an HTTP session.

You can visualize a session as made up of multiple independent requests from the same user (as in figure 5.20). In figure 5.21, each instance of a request-scoped object is unique to that request (R1 to R3). But a *session-scoped* instance is *shared* across all those requests.

Sessions are extremely useful for tracking state relative to a specific user, since a session always exists around one user. These uses include:

- Tracking a user's credentials for security purposes
- Tracking a user's recent activity for quick navigation (for example, breadcrumbs)
- Tracking preferences, to personalize a user's experience
- Caching user-specific information for quick access
- Caching general, constant data for quick access

All these use cases involve storing state temporarily, generally to improve a user's experience through the site. Whether that is about presentation or under the covers, it's about performance. And that's essentially what sessions do. Objects scoped under a session retain their state across requests, essentially continuing from where the last request left off.

Typically, sessions start and end when a user logs in and logs out. This behavior may be customized as necessary. Some requests (for static content, for example) do not participate in a user session and are considered stateless. These requests are independent of sessions, and, generally speaking, services participating in them shouldn't have any user-specific functionality.

Like requests, sessions may also be concurrent. For instance, multiple users who log in at once are said to be in different, unique sessions. While session-scoped instances are shared across requests inside a session, they are independent between sessions. Figure 5.22 shows how this might look in an injector's timeline.

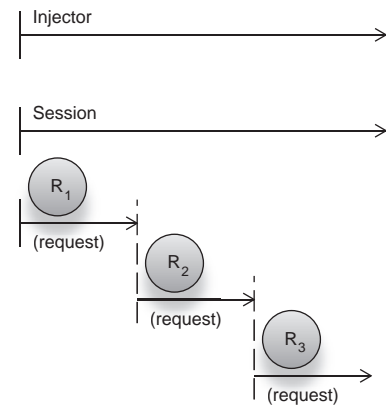


Figure 5.21 A session is composed of independent requests from a user.

In this figure, U_1 is an object that exists in the first user's session, and U_2 is a different instance of the same key that exists in the second user's session (it also is an aging Irish rock band). U_1 and U_2 are completely independent of one another. But within the first user's session, all requests share the same instance (U_1)—likewise with the second user and U_2 . Another interesting point is that the second user's session does not start for a while into the application's life. So there is a time when U_2 is out of scope while U_1 is in scope, even though both are instances of the same key (let's call it U) bound under session scope.

Another interesting thing about figure 5.22 is that the second user actually logs out and logs back in (at the point marked re-login). This means that there are two instances of U_2 for the second user because she started two sessions. The state of U_2 prior to the second login is totally independent from the state after the second login. Contrast this with singleton scoping, where instance state would have been shared for the entire life of the injector, regardless of the number of sessions (or users) involved.

Let's go back to the comic store example. We had three important components:

- **ComicSearch**—A request-scoped service that searched the comic catalog according to given criteria.
- **ComicAccess**—A singleton-scoped data-access service that acted as a bridge between **ComicSearch** and a database.
- **UserContext**—A request-scoped service that was specific to a user; it constructed personalized and filtered search results around a user's behavior.

There was also a filter that set up the **UserContext** each time, by extracting a username from incoming requests. Can you see any room for improvement? Let's revisit class **UserContext** to see if it provides any inspiration:

```
public class UserContext {
    private String username;

    private final ComicAccess comicAccess;

    @Inject
    public UserContext(ComicAccess comicAccess) {
        this.comicAccess = comicAccess;
    }

    public List<Comic> getItemsOfInterest() {
        ...
    }

    public List<Comic> searchComics(String criteria) {
        return comicAccess.searchNoPriorPurchases(criteria, username);
    }
}
```

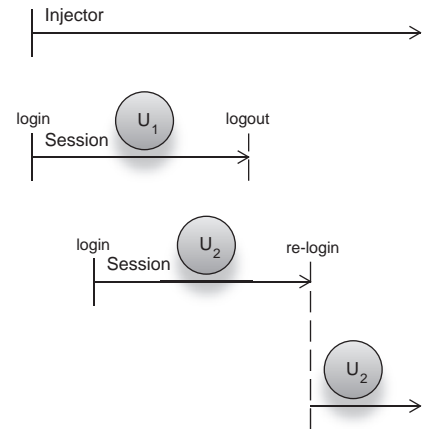


Figure 5.22 Multiple user sessions in the life of an injector

```

    }

    public void signIn(String username) {
        this.username = username;
    }
}

```

Straight away there is a clear benefit from making this object session scoped. We wouldn't have to sign in the user on every request! It would be enough to do it once, at the start of the session (logical, since this would be where a user logged in) and simply let it be shared across any further requests from the same user. This would be particularly useful if the application loaded user-specific information (such as a user's full name and date of birth) on sign in, since it would save several unnecessary trips to the database on subsequent requests.

Another saving comes from the use case around caching user-specific information for quick access. If we assume that items of interest for a user are unlikely to change within a single session, there is no need to search and collate them on every request. It can be done once and stored in a memo field for further requests. Here's how you might modify `getItemsOfInterest()` to do just that:

```

public class UserContext {
    private List<Comic> itemsOfInterest;    ← Memo field

    public List<Comic> getItemsOfInterest() {
        if (null == itemsOfInterest) {      ← Compute once
            ...                             and memorize
        }

        return itemsOfInterest;
    }

    ...
}

```

By storing computed suggestions in the `itemsOfInterest` memo field, I save a lot of unnecessary computation so long as subsequent requests come from the same user. Each user has her own memo, in their session-scoped `UserContext`.

Binding `UserContext` to session scope is also laughably simple. As far as the injector is concerned, all you need to do is change its binding (we'll do this in Guice and guice-servlet first):

```

import com.google.inject.servlet.RequestScoped;
import com.google.inject.servlet.SessionScoped;

public class ComicStoreModule extends AbstractModule {

    @Override
    protected void configure() {
        bind(ComicAccess.class).to(ComicAccessImpl.class).in(Singleton.class);

        bind(UserContext.class).in(SessionScoped.class);
        bind(ComicSearch.class).in(RequestScoped.class);
        ...
    }
}

```

Earlier we also saw a shortcut notation, where the class was itself annotated. This is possible with session scope too, but first it requires that you to set up a *scoping annotation*. This must be done manually because the web scopes aren't part of the core Guice distribution. It's fairly easy to do:

```
public class ComicStoreModule extends AbstractModule {
    @Override
    protected void configure() {
        install(new ServletModule());
        bind(ComicAccess.class).to(ComicAccessImpl.class).in(Singleton.class);
        bind(ComicSearch.class).in(ServletScopes.REQUEST_SCOPE);
        ...
    }
}
```

Notice that I've removed any explicit binding of `UserContext`. I can now simply annotate the class, and Guice will correctly bind it to the session scope:

```
@SessionScoped
public class UserContext {
    ...
}
```

You could do the same with request scopes and guice-servlet's `@RequestScoped` annotation.

The Spring equivalent for binding `UserContext` under session scope is also straightforward (once you have the context listeners set up):

```
<beans ...>
  <bean id="data.comics" class="ComicAccessImpl" scope="singleton">
    ...
  </bean>

  <bean id="web.user" class="UserContext" scope="session">
    <constructor-arg ref="data.comics"/>
  </bean>

  <bean id="web.comicSearch" class="ComicSearch" scope="request">
    <constructor-arg ref="data.comics"/>
    <constructor-arg ref="web.user"/>
  </bean>
</beans>
```

No additional configuration of the web server (or servlet container) is required. Cool! Finally, there are a couple of things to keep in mind when working with objects in the session scope:

- Multiple concurrent requests (from the same user) may belong to the same session.
- This means two threads may hit session-scoped objects at once.
- A session-scoped instance, if wired into a singleton, will stick around even when the user has logged out and the session has ended (*scope-widening injection*). Take care that this does not happen. We'll look at remedies in chapter 6.

So how does a session get scoped?

This is a rather tricky question. I said earlier that HTTP sessions are a hack for the fact that the HTTP protocol is *stateless*. This is more or less accurate—the common way of maintaining a user session is to use a browser *cookie*. A cookie is a small file with unique information that a web application can send to a browser as its identity. So, the next time that web browser hits the same URL, it will send back its cookies. When the web application sees this cookie, it “remembers” who is calling and correctly identifies the user—much like an ATM card reminds the bank of who you are. Cookies are specific to a website and URL, so there is no danger of cookies getting crossed between applications.

Not all browsers support cookies, and since some users consider them a risk, even browsers that do may not have them enabled. In these cases we need an alternative way of tracking sessions. One popular alternative is *URL rewriting*. URL rewriting is quite simple. When a browser requests a document, it does so via a URL. Usually this happens when you click a hyperlink on a previous page. Now, instead of the normal URL, the web application rewrites all the hyperlinks sent to a particular user by adding an identifier to it. Then when you (or any user) click the link, your browser requests the rewritten URL. Incoming requests of this nature are filtered and stripped of the extra bit identifying you as a unique user. This information is used to restore your session, and everything continues as normal.

Most web technologies can be configured to use either URL rewriting or cookies as their session continuation strategy. Some, like the *Java Servlet Framework*, automatically detect browser capability and choose the appropriate strategy. The Java Servlet Framework also provides abstractions for HTTP requests and sessions to make them easy to work with. Guice-servlet and the SpringFramework both provide integration layers that sit over the servlet APIs and enable transparent web scoping with dependency injection.

5.5 Summary

Scope is about managing the state of objects. Service implementations are bound to keys, which are realized as object instances in an application. Without scoping, keys requested from the injector return new instances each time (and each time they are wired as dependencies). This is known as the no scope. Since there is no way of specifying a duration for the time, state may be preserved in these instances. Singleton-scoped keys, on the other hand, are keys that the injector only wires or returns one instance for. In other words, singletons have one instance per key, per injector. This is different from singleton-patterned objects, which enforce one instance per entire application. I decry this flavor of singleton as an anti-pattern because it is nigh on impossible to test with mock substitution and is tricky in concurrently run tests and environments. Such anti-patterned singletons are also shared between multiple injectors in the same application, which may even violate the configuration of the injector. Always choose singleton scoping over singleton patterning.

Thus, scope may be thought of as the choice of instance each time a key is sought from it—whether new, or old, or shared. The singleton and the no scope are universal and have uses in most applications using dependency injection.

However, there is a whole class of scopes that are specific to particular kinds of applications. These scopes are closely tied to contexts specific to a particular problem domain. One such example is the web. Web-specific scopes are popular with many web frameworks and ship out of the box with many DI libraries. The HTTP request scope is the first and simplest of the web scopes. Keys requested from an injector within the same request always return the same instance. When the request completes, these request-scoped objects are discarded, and subsequent requests force the creation of new instances. Request scoping is interesting since the context that a request purports may be concurrent with several other requests. These requests are all walled off in their own unique scopes and keys, and when sought from the injector they are different across these requests.

HTTP session scope is a step up from request scope and is a natural extension of it. An HTTP session is an artificial construct above a string of requests from the same user. Session scope is thus persistent across all these requests (so long as they are from the same user). Sessions are useful for storing user-specific information temporarily, but they are often abused to store data that is relevant only to specific workflows within the user's session. Try to restrict your use of session scope to keeping around user-relevant information only. Good examples are credentials, preferences, and (relatively) constant biographical data.

More use cases in scoping



This chapter covers:

- Customizing scope to suit your needs
- Avoiding the perils of improper scoping
- Using advanced scoping: caches and data grids

“A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.”

—Douglas Adams

As you’ve just seen in chapter 5, scope is a very powerful tool for managing object state. In particular, it is a tool for making your objects and services *agnostic* of state management. In other words, you allow the injector to worry about the details of state and context, and you keep your objects lean. This has advantages in both productivity and design elegance.

In this chapter we’ll look at how to apply these same principles to any *stateful* context and to broaden the applicability of scoping as a tool in dependency injection. Most libraries provide some manner of extending the available scoping functionality. We’ll look at how to leverage this in Guice and Spring, in particular.

Closely related are the perils of scoping. While it is extremely powerful as a tool for reversing responsibility, state management can get very tricky to design and

build around. We'll look at common pitfalls and some very hairy corner cases, which one would do well to respect. We'll also see how good design can ameliorate or completely avoid these cases and how it can also facilitate good testing.

Finally, I present some ideas for using scope in unusual and potentially interesting ways for very large applications that require enormous scalability. But first, let's lay the groundwork and explore how to customize scope.

6.1 Defining a custom scope

We've had a long look at web scopes. HTTP request and HTTP session scopes provide a useful set of tools for working with context in web applications, particularly for dealing with user interactivity. Most applications (including web apps) are built around some form of data manipulation. In general this data is stored in a relational database like Oracle or PostgreSQL. In object-oriented languages, the same data is modeled as interrelating classes of objects and then mapped to database tables. Objects are marshaled to their counterpart tables, then retrieved as required in future interactions. This is often called *object* (or data) *persistence*.

The process of moving data between application memory (in the form of objects) and database storage (in the form of relational tables) occurs in series of tasks called *transactions*. Let's look at how to treat transactions as a context and define a custom scope around them.

6.1.1 A quick primer on transactions

A transaction is a form of interaction with a data store that is *atomic*, *consistent*, *isolated*, and *durable*. Such transactions are sometimes referred to as *ACID transactions*. This is what they imply:

- *Atomicity*—All of the tasks inside a transaction are successful or none are.
- *Consistency*—Data modified by a transaction does not violate prior constraints.
- *Isolation*—Other transactions can never see data in a partial state as it is being transacted.
- *Durability*—This refers to the ability to recover from failures while inside a transaction and leave data consistent.

These transactions extend to your application logic, meaning many different, unrelated components can all participate within a single transaction in order to perform a logical operation on data. Rather than pushing each task through in isolated fashion, it makes sense to combine them as a logical unit under an overall purpose. They are said to be *coarse grained*. When creating an employee record in the database, a transaction may include logic to create annual leave blocks, stock option grants, and other related records. If any one of these tasks fails (for instance, stock could not be assigned because of a shortage), the entire transaction is *rolled back* and no employee is created. This prevents the creation of a partial, *invalid* employee record and keeps the database in a consistent state. A new attempt must be made, once the stock shortage is corrected, in a separate transaction.

Like singletons or HTTP sessions, a transaction is also a specific context within an application. An injector can take advantage of this to reduce the passing around of ancillary data and placeholder objects and generally improve the maintainability of code by moving transactional concerns into *transaction-scoped* dependencies.

6.1.2 Creating a custom transaction scope

Before defining your own scope, it helps to ask a few questions:

- Is the context that this scope represents clearly defined?
- Objects must live for the entire life of this scope; does this work for all scoped instances? You should never have two life spans for a key within a scope.
- Is there going to be more than one thread accessing objects in this scope?

Transaction scope has well-understood start and end points (see figure 6.1). A transaction is also generally confined to one thread of execution. While it's by no means impossible to consider scopes that may be accessed by multiple threads, it is certainly preferable to design scopes that are confined to a single thread. These scopes are inherently simpler and easier to test and reason about. Classes that are bound under thread-confined scopes also don't need to worry about thread-safety, and they are therefore easy to design and test.

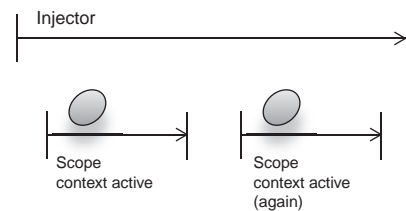


Figure 6.1 A scope generally exists over a repeating context.

So it looks like we're on good ground. Before scoping, let's define a use case. A nuclear power station needs software written to start the reactors and begin the flow of power to an electrical grid. The restriction is that there are three chambers to activate, and they must either all activate or none should. If the chambers are too hot, none must activate. We'll model this as a transaction with a common logical goal, that being the startup of the power station. Listing 6.1 imagines this transactional power station.

Listing 6.1 Three energy chambers activated in concert under a transaction

```
public class PowerStation {
    ...

    public void start() {
        transaction.begin();

        chamber1.fire();
        chamber2.fire();
        chamber3.fire();

        transaction.commit();
    }
}
```

We won't worry too much about the transaction architecture. The important thing is the context that it provides. In `PowerStation`'s case, that context exists for the

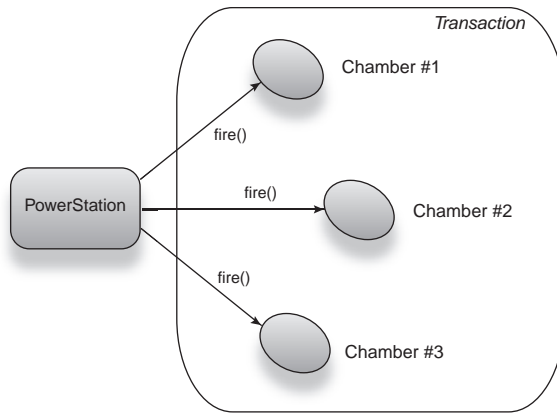


Figure 6.2 PowerStation fires all three Chambers in a single transaction.

duration of method `start()`, finishing when it returns (figure 6.2 shows a model of this transaction). The call to `transaction.commit()` implies a successful completion of the transaction.

Now let's introduce the failure scenario, where one of the chambers may fail to start by throwing an exception:

```

public void start() {
    transaction.begin();

    boolean failed = false;
    try {
        chamber1.fire();
        chamber2.fire();
        chamber3.fire();

    } catch (StartFailedException e) {
        failed = true;
    } finally {
        if (failed)
            transaction.rollback();
        else
            transaction.commit();
    }
}

```

In this modified version, `start()` expects that one of the chambers may fail, catches the relevant exception (`StartFailedException`), and proceeds to discard the changes to the transaction by rolling it back. The following lines ensure that a failure to start aborts the entire transaction:

```

    if (failed)
        transaction.rollback();
    else
        transaction.commit();

```

Now let's look at the code in each chamber that may cause the abortive fault:

```

public class Chamber {
    private final Gauge gauge;

```

```

...
public void fire() {
    gauge.measure();

    if (gauge.temperature() > 100)
        throw new StartFailedException();
    ...
}
}

```

If the chamber's gauge reads above 100, `fire()` will abort by throwing a `StartFailedException`, causing the transaction to abort too. This works, but it does not ensure that the cumulative temperature is under control. All three chambers together must have a temperature less than 100, which the code fails to ensure. We need a change to how `fire()` performs its temperature check. As things stand, `Gauge` measures the heat level from each chamber independently. In other words, each `Chamber` has its own `Gauge` (that is, `Gauge` is bound under the `no` scope). To fix the problem, we need to share the gauges among all three chambers. Singleton scoping is not quite right since this would mean *all* dependents of `Gauge` would share the same instance (perhaps a backup `Chamber` on the other side of the station or perhaps some component that's not a `Chamber` at all).

The context for this heat measurement is the starting up of the `PowerStation` (its `start()` method). Therefore, the *transaction* scope is an apt fit:

```

@TransactionScoped
public class Gauge { .. }

```

It's shown in figure 6.3 as a class model.

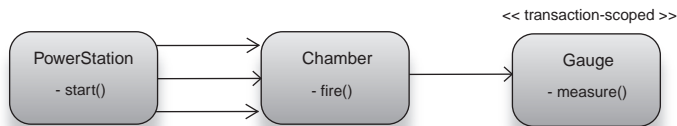


Figure 6.3
A `PowerStation`'s
`Chamber` depends on
a transaction-scoped
measuring `Gauge`.

Now when each chamber is fired and consequently measures its heat level, the `Gauge` correctly reports a cumulative measure of heat from all three chambers. There need be no further changes to `PowerStation`, `Chamber`, or any of their methods.

`@TransactionScoped` is something we just made up (there is certainly no library for it). So how would we go about creating this scope in an injector? The following section answers this question using Guice.

6.1.3 A custom scope in Guice

A scope in Guice is represented by the `Scope` interface. Scope is as follows:

```

public interface Scope {
    <T> Provider<T> scope(Key<T> key, Provider<T> unscoped);
}

```

A custom scope is required to implement only one method, `scope()`, which returns a scoped instance `Provider`. Recall the `Provider` pattern from chapter 4. This is the same, except that instead of being used against the reinjection problem, here it's used for providing scoped instances. The two arguments to `scope()` manage the particular object in its context:

- `Key<T> key`—The combinatorial key to be scoped
- `Provider<T> unscoped`—A provider that creates *new* instances of the bound key (in other words, it is a *no-scope* provider)

Use provider `unscoped` when a new context comes to life and new instances need to be provided. When the scope context completes, you simply discard all scoped instances, obtaining new ones from provider `unscoped` instead (see figure 6.4).

Listing 6.2 shows how this is done with the transaction scope.

Listing 6.2 TransactionScope implemented in Guice

```
public class TransactionScope implements Scope {
    private final ThreadLocal<Map<Key<?>, Object>> instances
        = new ThreadLocal<Map<Key<?>, Object>>();

    public <T> Provider<T> scope(
        final Key<T> key, final Provider<T> unscoped) {

        return new Provider<T>() {      ← Returns a scoped provider

            public T get() {
                Map<Key<?>, Object> map = instances.get();

                if (!map.containsKey(key)) {
                    map.put(key, unscoped.get())
                }
                return (T) map.get(key);
            }
        };
    }

    public void beginScope() {          ← Sets up a new context
        instances.set(new HashMap<Key<?>, Object>());
    }

    public void endScope() {           ← Closes an existing context
        instances.remove();
    }
}
```

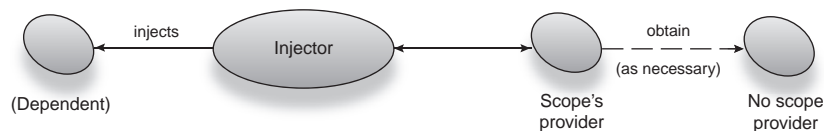


Figure 6.4 The injector obtains instances from a `Scope` provider, which itself may use the `no-scope` provider.

In listing 6.2, class `TransactionScope` exposes the Guice interface `Scope`. The implemented method `scope()` returns a *scoped* provider that acts as a bridge between the injector and its default no-scope provider. The no-scope provider `unscoped` is used to create new instances as required.

A scoped provider is returned for each key requested in method `scope()`. The provider checks a *thread-local* hash table, returning cached instances for the current thread:

```
public T get() {
    Map<Key<?>, Object> map = instances.get();

    if (!map.containsKey(key)) {
        map.put(key, unscoped.get());
    }

    return (T) map.get(key);
}
```

If an instance is not already present in the hash table, it is created and cached:

```
if (!map.containsKey(key)) {
    map.put(key, unscoped.get());
}
```

Otherwise, the instance mapped to the given key is returned normally:

```
return (T) map.get(key);
```

This ensures objects that don't yet exist in the current context are created when needed. Method `beginScope()` sets up a context for the current transaction by creating a new hash table to cache objects for the life of the scope (transaction):

```
public void beginScope() {
    instances.set(new HashMap<Key<?>, Object>());
}
```

The hash table maps Guice Keys to their scoped instances. Its complement, method `endScope()`, is called by the transaction framework when a transaction completes (either by a successful commit or an abortive rollback), disposing of the cached instances:

```
public void endScope() {
    instances.remove();
}
```

The entire context of the transaction, its scope, and scoped instances are confined to a single thread by using the `ThreadLocal` construct:

```
public class TransactionScope implements Scope {
    private final ThreadLocal<Map<Key<?>, Object>> instances
        = new ThreadLocal<Map<Key<?>, Object>>();

    ...
}
```

`ThreadLocals` are utilities that allow stored objects to be available only to the storing thread and none other. Because a transaction runs entirely in a single thread, we can be

sure that its context will only ever need to be accessed by that thread, and simultaneous transactions are separated from one another by confinement to their respective threads.

Now when a transaction-scoped object (such as Gauge from the previous section) is wired to any other object, it is done so inside an active transaction in the *current* thread. When a transaction completes, this instance is lost and any new transactions will see a new instance created. This also applies to concurrent transactions; they are kept walled apart by thread confinement.

Cool! What about when *no* transaction is active and a key is sought? Well, this is a serious problem, and it likely represents an error on the programmer's part. Either she did not start the transaction where it was meant to start or she configured the injector incorrectly. You should add a safety check that reports the error quickly and clearly in such cases:

```
public <T> Provider<T> scope(
    final Key<T> key, final Provider<T> unscoped) {

    return new Provider<T>() {

        public T get() {
            Map<Key<?>, Object> map = instances.get();

            if (null == map) {
                throw new
                OutOfScopeException("no transaction was active");
            }

            if (!map.containsKey(key)) {
                map.put(key, unscoped.get());
            }

            return (T) map.get(key);
        }
    };
}
```

This is our sanity check that ensures a proper exception is raised (`OutOfScopeException`) if a transaction-scoped object is sought outside a transaction.

Registering a custom scope in Guice is simple:

```
public class TransactionModule extends AbstractModule {

    @Override
    protected void configure() {
        bindScope(TransactionScoped.class, new TransactionScope());
        ...
    }
}
```

The scoping annotation `@TransactionScoped` is declared as follows:

```
@Retention(RetentionPolicy.RUNTIME)
@ScopeAnnotation
public @interface TransactionScoped { }
```

This is a standard bit of boilerplate that needs to be used when declaring any Guice scope annotation. Notice that `@ScopeAnnotation` is a meta-annotation (an annotation

on an annotation), which tells the injector that this is a scoping annotation. Now say that three times fast!

6.1.4 **A custom scope in Spring**

The same principles of context, instance longevity, and thread-safety apply to all injectors, and Spring is no different. The particular interfaces to expose for scoping, however, are slightly different. Spring also has a `Scope` interface that your scope implementation must expose. But it looks a bit different from Guice's `Scope`. It is shown in listing 6.3.

Listing 6.3 Spring's custom scope interface

```
package org.springframework.beans.factory.config;

public interface Scope {

    Object get(String key, ObjectFactory unscopedProvider);

    Object remove(String key);

    String getConversationId();

    void registerDestructionCallback(String key,
        Runnable destructionCallback);
}
```

Let's take a look at this interface and see how it relates to what we saw in the previous section. First, the scoping provider method:

```
Object get(String key, ObjectFactory unscoped);
```

This is almost identical to Guice's `scope()` method, except that instead of returning a scoped provider, `get()` returns the scoped instance itself. Instead of taking a combinatorial key, it takes a string key. You can think of `get()` as being the provider itself. The second argument, `ObjectFactory`, is an implementation of the `Provider` pattern (similar to Guice's `unscoped Provider`). It provides no-scoped instances of the given key.

Next, the `remove()` method:

```
Object remove(String key);
```

`remove` evicts any instance stored for the given key in the scope's internal cache (the hash table in `TransactionScope`), and then another method retrieves the scope's identity:

```
String getConversationId();
```

This is an unusually named method. What it essentially points to is the unique identity of the particular scope context. If a transaction were active for the current thread, this method would return a string that uniquely identified this transaction. If we were implementing a session scope, we would return a session ID, unique to a user.

Finally, there's a lifecycle support method:

```
void registerDestructionCallback(String key, Runnable
➡ destructionCallback);
```

`registerDestructionCallback()` is intended to support lifecycle for specific keys when their instances go out of scope. We won't delve much into this now since lifecycle is coming up in the next chapter.

So what might `TransactionScope` look like with Spring? Listing 6.4 takes a stab at it.

Listing 6.4 Transaction scope implemented as a custom scope in Spring

```
package my.custom;

public class TransactionScope implements Scope {
    private final ThreadLocal<Map<String, Object>> instances
        = new ThreadLocal<Map<String, Object>>();

    public Object get(String key, ObjectFactory unscoped) {
        Map<String, Object> map = instances.get();

        if (null == map)
            throw new IllegalStateException("no transaction is active");

        if (!map.containsKey(key)) {
            map.put(key, unscoped.getObject());
        }

        return map.get(key);
    }

    public void beginScope() {
        instances.set(new HashMap<String, Object>());
    }

    public void endScope() {
        instances.remove();
    }

    public Object remove(String key) {
        if (null == instances.get())
            throw new IllegalStateException("no transaction is active");

        return instances.get().remove(key);
    }

    public String getConversationId() {
        if (null == instances.get())
            throw new IllegalStateException("no transaction is active");

        return instances.get().toString();
    }

    public void registerDestructionCallback(String key,
        Runnable destructionCallback) {
        ...
    }
}
```

Scoped instances provider method

Starts scope context

Disposes current scope context

Disposes a specific key from current context

A unique string identifying the hash table (context)

Listing 6.4's `TransactionScope` is very similar to Guice's `TransactionScope`. The major difference is the three additional methods that Spring requires. For the conversation ID, I return the hash table itself (in `String` form). This is a simple hack for

identifying a context. Depending on the kind of Map, this may or may not have good results. You will want to generate a more appropriate identifier in a production version. I also perform a sanity check every time scoping methods are called:

```
if (null == instances.get())
    throw new IllegalStateException("no transaction is active");
```

This is important because it fails fast with a clear and easily identifiable `IllegalStateException`. Countless hours have been wasted because failing code does not indicate exactly what was wrong and where.

Registering this scope with the injector is straightforward, and it involves a bit of simple cut-and-paste code in your injector configuration:

```
<beans ...>
  <bean
    ➤ class="org.springframework.beans.factory.config.CustomScopeConfigurer">
    <property name="scopes">
      <map>
        <entry key="transaction">
          <bean class="my.custom.TransactionScope"/>
        </entry>
      </map>
    </property>
  </bean>

  ...
</beans>
```

Then you are free to use it just like any other scope when declaring keys:

```
<bean id="gauge" class="nuclear.Gauge" scope="transaction">
  ...
</bean>
```

Now your new scoped objects can be injected and used as needed. The next section switches around and looks at problems you can encounter when creating your own scopes. It is essential to understand these before you go off and create your own scopes.

6.2 *Pitfalls and corner cases in scoping*

Letting the injector manage the state of your objects is a wonderful thing when applied properly, but it also attracts danger. Scopes are extremely powerful because they invert the responsibility of managing state, and the injector takes care of passing it around to dependents. It keeps objects free of such concerns and makes them more focused and testable. However, this same power can lead to very grave pitfalls.

Many of these have to do with thread-safety. Some are about construction of object graphs that are striped with objects of different scopes. These pitfalls can lead to a range of unrelated problems that are not immediately apparent, for example, memory leaks, poor performance, or even erratic behavior. The counter to this latent peril is a thorough understanding of problem cases. With that said, let's start with scoping and thread-safety.

6.2.1 Singletons must be thread-safe

Singletons absolutely must be thread-safe—no exceptions. Any serious application has several threads of execution that are continually operating. They may be servicing different users, timers, remote clients, or any of a number of other purposes. A singleton-scoped key directly implies that only one *shared* instance of an object exists in the entire injector (and consequently the application). Multiple threads often end up accessing singleton instances and may even do so concurrently. Chapter 9 discusses the intricacies of threading and objects in detail (with a particular emphasis on Java’s thread and memory model).

There are two ways in which you can create thread-safe classes. The simpler and more straightforward is to make them immutable, as shown in figure 6.5.

The fields of the following class are immutable (see how this might look for threads accessing it in figure 6.6):

```
import net.jcip.annotations.Immutable;
...

@Immutable @Singleton
public class MySafeObject{
    private final Dependency dep1;
    private final Dependency dep2;

    @Inject
    public MySafeObject(Dependency dep1, Dependency dep2) {
        this.dep1 = dep1;
        this.dep2 = dep2;
    }

    public Dependency getDep1() { return dep1; }
    public Dependency getDep2() { return dep2; }
    ...
}
```

Notice that `MySafeObject` has no setters and no methods that can alter or affect its dependencies’ state. This is good immutable design. The marker annotation `@Immutable` is merely a tag that documents the behavior of `MySafeObject` (it has no effect in practice). Place it on classes that you have designed to be immutable. Don’t be confused by its proximity to `@Singleton`; it is there only as documentation. It helps unfamiliar pairs of eyes read, understand, and reason about your classes. It also reduces confusion about intent. This is especially useful when you have bugs, since a colleague can spot the bug and know immediately that it violates design intent.

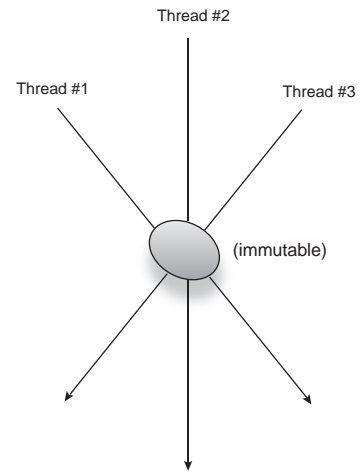


Figure 6.5 Multiple threads may safely use an immutable object.

TIP @Immutable and other thread-safety-related annotations are available from the *Java Concurrency in Practice* website.¹ They are downloadable at the site and provided under a flexible open source license. Neither Guice nor any other injector or production library reacts to the @Immutable annotation.

If their values never change, there is no risk of one thread modifying a value that's out of sync with its rivals. As figure 6.6 shows, mutable singletons, if not properly guarded, can get out of sync between threads.

With immutable objects, the order of thread execution is irrelevant, so you have fewer problems to design for. Try very hard to make singleton objects immutable. This is by far the simplest and best approach.

Of course, not every singleton can be made immutable. Sometimes you may need to alter the object's state (for example, in a counter or accumulator). If you decide to use setter injection to resolve circular dependencies or the in-construction problem (see chapter 3), you cannot make the class immutable. In these cases, you must carefully design the object and all of its dependencies to be thread-safe. Here I have the same class; it is no longer immutable but thread-safe insofar as it guarantees that all of its dependencies will be visible to all threads:

```
import net.jcip.annotations.ThreadSafe;

@ThreadSafe @Singleton
public class MySafeObject2{
    private volatile Dependency dep1;
    private volatile Dependency dep2;

    @Inject
    public void set(Dependency dep1, Dependency dep2) {
        this.dep1 = dep1;
        this.dep2 = dep2;
    }

    public Dependency getDep1() { return dep1; }
    public Dependency getDep2() { return dep2; }

    ...
}
```

@ThreadSafe, like @Immutable, is intended to convey information about the class's design, nothing more. Java's volatile keyword ensures that the value of a field is kept

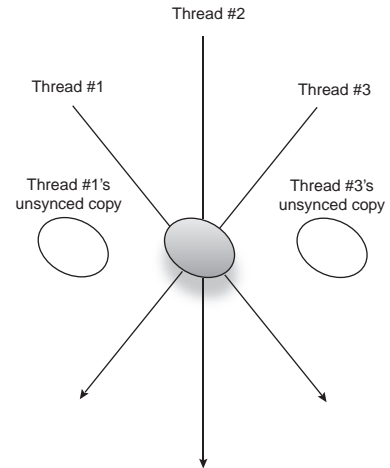


Figure 6.6 All visibility bets are off with multiple threads accessing a mutable object.

¹ Find out more at <http://jcip.net>. Chapter 9 has a lot of detailed information on concurrency and good class and object design. Jump ahead if you are curious.

coherent with all threads. In other words, when `MySafeObject2`'s dependencies are changed using the `set()` method, they are immediately visible to all threads. This is a very important point, and you will appreciate why when `@ThreadSafe` is contrasted with the following class:

```
import net.jcip.annotations.NotThreadSafe;

@NotThreadSafe @Singleton
public class MyUnsafeObject {
    private Dependency dep1;
    private Dependency dep2;

    @Inject
    public void set(Dependency dep1, Dependency dep2) {
        this.dep1 = dep1;
        this.dep2 = dep2;
    }

    public Dependency getDep1() { return dep1; }
    public Dependency getDep2() { return dep2; }
    ...
}
```

`MyUnsafeObject` is critically flawed, because the Java language makes no guarantees that any thread other than the one that set its dependencies will see the dependencies (without additional synchronization). In other words, the memory between threads in non-final, non-volatile fields may be incoherent. The subtle reason has to do with the way Java Virtual Machines (JVM) are designed to manage threads. Threads may keep local, cached copies of non-volatile fields that can quickly get out of sync with one another unless they are synchronized correctly. We'll study this problem and its solutions in greater detail in chapter 9.

Making a class thread-safe is often a difficult and involved process. No simple litmus test exists to say a class is completely thread-safe. Even the best of us can get tripped up when dealing with apparently simple threading and concurrency issues. Your best bet is to reason carefully about a class and its threading environment. Explore plenty of scenarios both common and atypical where a singleton may be used. Try to convince at least one other person of the safety of your code. Actively hunt for flaws; never assume they don't exist. Always validate your assumptions. Test! Test! And test again until you're satisfied.

Thread-safety problems can also present in other ways, particularly when dealing with dependencies that cross scoping boundaries. This is an especially tricky situation called *scope widening*, which we'll explore in detail in the next section.

6.2.2 *Perils of scope-widening injection*

I've mentioned a problem called *scope-widening injection* a few times so far in this book. It came up particularly when wider scoped objects depended on narrower scoped objects, such as a singleton that depends on a no-scoped or request-scoped object. When such a situation is encountered, you end up with a no-scoped (or

request-scoped) object that is held by the singleton, and it no longer comes under the purview of its original scope. That means when the context ends, the instance continues to exist as a dependency of the singleton. As you can imagine, this causes a whole swathe of knotty problems that can go undetected. Worse, they won't necessarily cause errors or exceptions to be raised and may appear to be functioning correctly, when really they're opening up a whole bunch of unintended semantics and consequences.

Scope widening also applies to any scopes that are logically of different contexts—even those that may belong to the same scope but refer to separate contexts, such as two separate user sessions. While this kind of scope widening is less likely (due to the walling off between contexts by the injector), it can still occur if a singleton is used to bridge them. I should really say, if a singleton is *abused* as a bridge. Not only does it break the semantics of the scope as understood at the time of its definition, it also can lead to serious performance, maintenance, and correctness problems. Scope widening is tricky, since it cannot easily be addressed by testing either. The simplest kind of peril in scope widening is an object whose scope has ended. This gremlin is called an *out-of-scope object*.

BEWARE THE OUT-OF-SCOPE OBJECT!

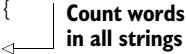
Instances that are out of scope present the most serious difficulty of the scope-widening cases. This naïve singleton class depends on a no-scoped component:

```
@Singleton
public class TextReader {
    private final WordCounter counter;

    @Inject
    public TextReader(WordCounter counter) {
        this.counter = counter;
    }

    public void scan(List<String> strings) {
        for(String string : strings)
            counter.count(string);

        System.out.println("Total words: " + counter.words());
    }
}
```




Count words
in all strings

TextReader is simple; its method `scan()` accepts a list of strings and prints the total number of words. Its dependency `WordCounter` counts the number of words it has received so far:

```
public class WordCounter {
    private int count;

    public void count(String text) {
        count += text.split(" ").length;
    }

    public int words() {
        return count;
    }
}
```



Split string
around spaces

So far, so good. Let's try using the `TextReader` to count a few example string sets:

```
Injector injector = Guice.createInjector();
TextReader reader = injector.getInstance(TextReader.class);
reader.scan(Arrays.asList("Dependency injection is good!",
    "Really, it is!"));
```

This code correctly prints a total word count of 7.

```
Total words: 7
```

Now let's extend the example and give it more to do:

```
Injector injector = Guice.createInjector();
    TextReader reader = injector.getInstance(TextReader.class);
reader.scan(Arrays.asList("Dependency injection is good!",
    "Really, it is!"));
reader.scan(Arrays.asList("Dependency injection is terrific!",
    "Use it more!"));
```

What does the program print now?

```
Total words: 7
Total words: 14
```

What went wrong? The second count should have been the same as the first. Double-check the words: There are only seven in both sets. We've been had by scope-widening injection. While you would normally expect the no-scoped `WordCounter` to be disposed of after the first call to `scan()` and a new instance to take its place subsequently, what really happened is that `TextReader` has held onto the original instance. This situation would not improve even if we were to obtain `TextReader` itself twice from the injector:

```
Injector injector = Guice.createInjector();
TextReader reader = injector.getInstance(TextReader.class);
reader.scan(Arrays.asList("Dependency injection is good!",
    "Really, it is!"));

TextReader reader2 = injector.getInstance(TextReader.class);
reader2.scan(Arrays.asList("Dependency injection is terrific!",
    "Use it more!"));
```

This program still prints an incorrect result:

```
Total words: 7
Total words: 14
```

The first run is okay, since at that time all instances (`TextReader` and `WordCounter`) are new ones. But the second run uses the old instance of `WordCounter` as well and so ends up accumulating the word count. If we added a third line this accumulation would continue (visualized in figure 6.7):

```
Injector injector = Guice.createInjector();
    TextReader reader = injector.getInstance(TextReader.class);
reader.scan(Arrays.asList("Dependency injection is good!",
```

```

    "Really, it is!");

    TextReader reader2 = injector.getInstance(TextReader.class);
    reader2.scan(Arrays.asList("Dependency injection is terrific!",
        "Use it more!"));

    TextReader reader3 = injector.getInstance(TextReader.class);
    reader3.scan(Arrays.asList("The quick brown fox", "is really annoying!"));

```

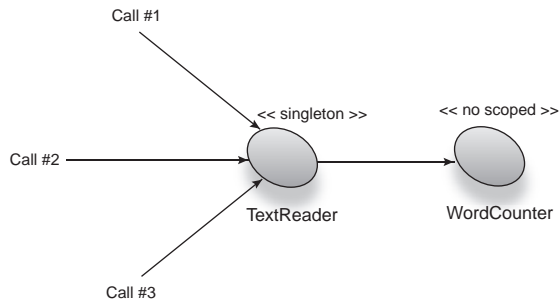


Figure 6.7 `WordCounter`'s scope is widened since it is held by singleton-scoped `TextReader`.

This should print three sets of 7, but instead it prints

```

Total words: 7
Total words: 14
Total words: 21

```

There are a couple of really easy solutions. The simpler is to remove the discrepancy in scope between `TextReader` and `WordCounter` by choosing the narrowest scope for both, in other words, make `TextReader` no scoped:

```
public class TextReader { .. }
```

Note the missing `@Singleton` annotation, implying no scope. This is shown in figure 6.8.

Now, the program can remain as is:

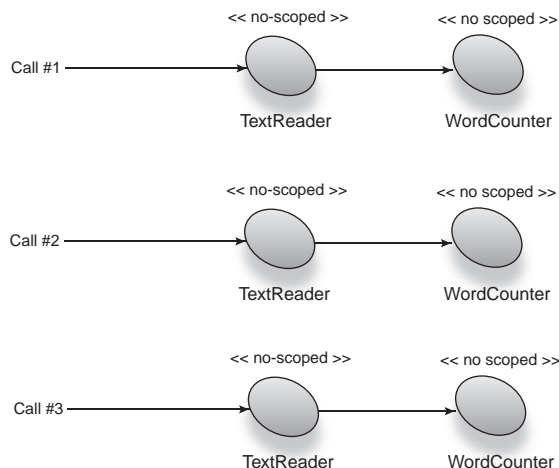


Figure 6.8 Both `TextReader` and its dependent have the same scope and are safe from scope widening.

```

Injector injector = Guice.createInjector();
    TextReader reader = injector.getInstance(TextReader.class);
reader.scan(Arrays.asList("Dependency injection is good!",
    "Really, it is!"));

TextReader reader2 = injector.getInstance(TextReader.class);
reader2.scan(Arrays.asList("Dependency injection is terrific!",
    "Use it more!"));

TextReader reader3 = injector.getInstance(TextReader.class);
reader3.scan(Arrays.asList("The quick brown fox", "is really annoying!"));

```

And it correctly prints the three independent word counts:

```

Total words: 7
Total words: 7
Total words: 7

```

This is an example of scope-widening injection that could not be caught with a unit test. Any test of `WordCounter` would pass, since it does correctly count the number of words in a string. Any test of `TextReader` would also pass, since a mocked `WordCounter` would only assert that correct calls were made to it around its *behavior* rather than around its *state*.

Well, this solution works, but by making `TextReader` no scoped rather than fixing the problem of scope widening, you might say I've sidestepped it. While this is a legitimate solution, there are times when this doesn't work. You need a singleton, and it depends on a no-scoped instance, or, more generally, a wider-scoped object depends on a narrower-scoped object.

Scope widening viewed in this light is a form of the *reinjection problem* (see chapter 3). We can use the same solution to mitigate scope widening. By replacing the narrower-scoped dependency with a provider (and fetching it each time), we allow the injector to intervene when new instances are required:

```

@Singleton
public class TextReader {
    private final Provider<WordCounter> counterProvider;

    @Inject
    public TextReader(Provider<WordCounter> counterProvider) {
        this.counterProvider = counterProvider;
    }

    public void scan(List<String> strings) {
        WordCounter counter = counterProvider.get();

        for(String string : strings)
            counter.count(string);

        System.out.println("Total words: " + counter.words());
    }
}

```

`TextReader` can now go back to being a singleton but continue depending on a no-scoped object without any scope-widening effects. Figure 6.9 imagines how this might work.

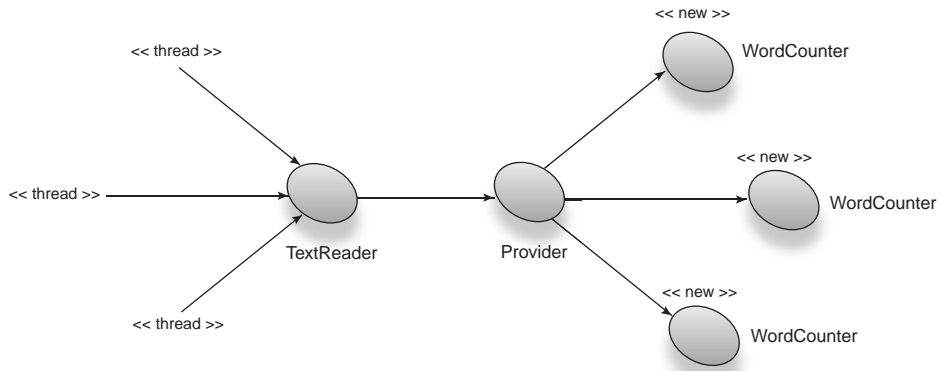


Figure 6.9 A provider acts as an intermediary and prevents widening WordCounter's scope.

Running the program yields the correct word counts:

```
Total words: 7
Total words: 7
Total words: 7
```

One subtle thing to note is that in the new TextReader's scan() method we retrieve WordCounter once per *method* rather than once per *use*. If we had done this instead:

```
public void scan(List<String> strings) {
    for(String string : strings)
        counterProvider.get().count(string);

    System.out.println("Total words: " +
        counterProvider.get().words());
}
```

we would have ended up with a very different (and rather obnoxious) result:

```
Total words: 0
Total words: 0
Total words: 0
```

There's nothing wrong with the coding of WordCounter or its logic, and that's fine. It's just that the statement that prints the count uses a new instance of WordCounter, discarding the one that has just run a count. This is one of the perils of stateful behavior. It would probably have had you looking everywhere except where the fault was (that is, in the scoping).

NO, REALLY: BEWARE THE OUT-OF-SCOPE OBJECT!

Out-of-scope objects can also cause bugs in other areas. One common version is the use of an out-of-scope object between (or before) scope contexts (see figure 6.10).

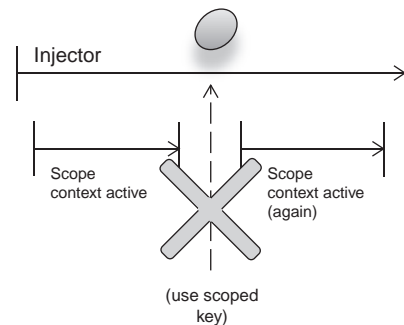


Figure 6.10 Beware of out-of-scope objects used between contexts of a scope.

This is a common problem that users of guice-servlet and Guice post about on the community mailing lists. Consider a servlet managed by guice-servlet:

```
import com.google.inject.servlet.RequestScoped;

@RequestScoped
public class HelloServlet extends HttpServlet {
    public void init(ServletConfig config) { .. }

    ...
}
```

and its injector configuration:

```
Guice.createInjector(new ServletModule() {
    protected void configureServlets() {

        serve("/hello").with(HelloServlet.class);

    }
});
```

Now there is one method of note in `HelloServlet`: `init()`, which is called by the guice-servlet on startup of the web application. What's interesting is that `HelloServlet` is itself request scoped. The servlet lifecycle methods are fired when the web application starts and there are no active requests at the time. The injector (via request scope) immediately detects that there is no active request, so it aborts with an `OutOfScopeException` (as shown in figure 6.11). This happens at the earliest point of the application's life, so it fails to load the entire web app.

Session scoping a servlet will similarly fail because no user session can be found under which to bring the key into scope. These are all cases of objects being used out of scope and represent serious flaws in the design of a program. In other words, `HelloServlet` is unavailable for the application to start up. We can use one of the mitigants of the reinjection problem, the Adapter pattern, to solve this problem. Consider this rather plain but functional version of `HelloServlet` bound with an Adapter:

```
@Singleton
public class HelloServletAdapter extends HttpServlet {
    private final Provider<HelloServlet> delegate;

    public void init(ServletConfig config) {
        //do init work
    }

    public void service(HttpServletRequest request,
```

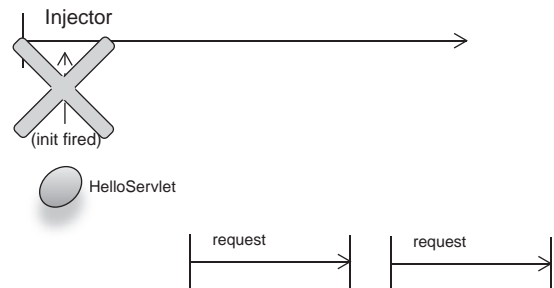


Figure 6.11 Request-scoped `HelloServlet` is used out of scope, that is, before any requests arrive.

```

        HttpServletResponse response) throws IOException,
        ServletException {
            delegate.get().service(request, response);
        }
        ...
    }
}

```

Also the modified injector configuration, using the Adapter instead:

```

Guice.createInjector(new ServletModule() {
    protected void configureServlets() {
        serve("/hello").with(HelloServletAdapter.class);
    }
});

```

`HelloServletAdapter` is a singleton wrapper around `HelloServlet` that acts as a go-between. It solves the out-of-scope problem (figure 6.12) by moving the logic for the portions that are outside the scope of `HelloServlet` into itself. Any calls to its `service()` method are delegated down to the request-scoped `HelloServlet`.

Since `service()` is called only during a request, we can rest assured that `HelloServlet` will always be inside a valid request. Now `HelloServlet` can keep its scope and still effectively function as expected. Now that we've seen all the domestic problems of scope widening, let's shift gears and look at the truly nasty pitfalls—ones that involve multiple threads and objects in widened scopes.

THREAD-SAFETY REVISITED

Scope widening can also lead to concurrency problems. Wiring any singleton with a narrower-scoped dependency means that it automatically becomes accessible to all the dependents of that singleton. If multiple threads access the singleton object, they are likely to modify its nonsingleton dependencies. If they are not immutable or thread-safe, you may see erratic and unpredictable behavior. Even if they are, they may be underperforming. Thread-safety is a particularly tough subject, since there are so many nuances to the concurrent modification (and access) of state. One of these is the issue of *safe publication*, which is dealt with in chapter 9.

You must take particular care to ensure that singletons never depend on objects that are not thread-safe, or if they do, that these dependencies are guarded properly. Here's an example of a class, scoped as a singleton, that depends on a counting service:

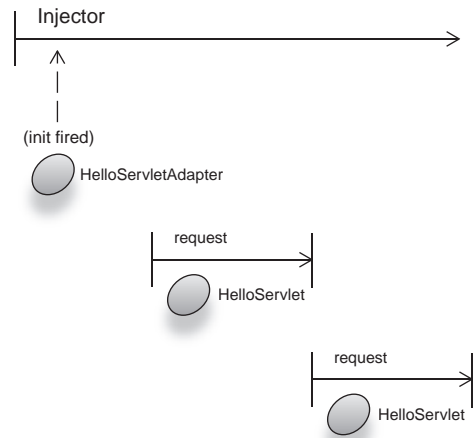


Figure 6.12 A wrapping singleton `HelloServletAdapter` mitigates the out-of-scope problem.

```
@Singleton
public class TextPrinter {
    private final Counter counter;

    public void print(String text) {
        System.out.println(text);

        counter.increment();

        if (counter.getCount() == 100)
            System.out.println("done!");
    }
}
```

TextPrinter has a thread *unsafe* dependency, Counter, which looks like this:

```
@NotThreadSafe
public class Counter {
    private long count;

    public void increment() {
        count++;
    }

    public long getCount() {
        return count;
    }
}
```

Because many threads hit TextPrinter's methods, print() and printCount(), they also increment and read Counter concurrently. When the number of lines printed hits 100, it prints "done!" Unfortunately, the way things stand this may never happen. The counter keeps incrementing, no matter how many threads hit it, so it is logical to assume that counter must reach 100 at some point (when a hundred lines are printed):

```
if (counter.getCount() == 100)
    System.out.println("done!");
```

So what's the problem? Right away there is the problem of memory *coherency*. Class Counter is not a singleton and so has been designed without thread-safety in mind. That would be fine were it used at any scope other than singleton (or more strictly, any scope that is confined to single threads). In practice, however, Counter has had its scope widened by being wired to TextPrinter, which is a singleton. One fix we can try is to make count volatile (as we did in a similar example not long ago):

```
//@ThreadSafe?
public class Counter {
    private volatile long count;

    public void increment() {
        count++;
    }

    public long getCount() {
        return count;
    }
}
```


Now, at last, all threads see the same value of count. However, this isn't quite enough. It is conceivable that two threads could run through the `print()` method at once—the first thread incrementing count to 100 and the second to 101. Then if either the first or second continues down the `print()` method, it will hit the statement testing whether count is 100. But the count has already hit 101, so the statement evaluates to false and keeps on going. Our base case is never reached, and the program never prints "done!"

Clearly, concurrent access to the counter is not desirable—at least not willy-nilly. To fix this problem, we need to ensure that each thread sees not only the latest value of count but also the value that it has set. In other words, each thread must have access to the counter exclusively. This is known as *mutual exclusion* and is achieved as follows (also illustrated in figure 6.13):

```
@ThreadSafe @Singleton
public class TextPrinter {
    private final Counter counter;

    public synchronized void print(String text) {
        System.out.println(text);

        counter.increment();

        if (counter.getCount() == 100)
            System.out.println("done!");
    }
}
```

The `synchronized` keyword ensures that no two threads execute method `print()` concurrently.

Threads queue up behind one another and execute `print()` one by one, ensuring that calls to `getCount()` always return the value set by the same thread in `counter.increment()` just above. Can we now call `Counter` thread-safe? It has proper visibility across threads, and it works in singletons that guarded it appropriately. But we can't quite go that far. As a dependency, it can still be used in a *thread-unsafe* manner. So we must leave it thus:

```
@NotThreadSafe
public class Counter {
    private long count;

    public void increment() {
        count++;
    }

    public long getCount() {
        return count;
    }
}
```

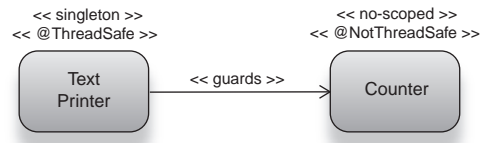


Figure 6.13 Singleton `TextPrinter` must guard access to its *unsafe* dependencies.

Thread-safety is an eternally difficult problem, and it's especially subtle in the case of scope widening since it is hard to detect. Good documentation and modular design will help you go a long way to avert its issues. Scope widening can also create nasty problems that don't involve threads, particularly with memory usage. In the next section we'll look at how this edge case can rear its ugly head and how to solve it.

SCOPE WIDENING AND MEMORY LEAKS

Scope-widening injection certainly has a lot of pitfalls! In Java, a garbage collector runs behind your programs, periodically checking for objects that are no longer in use, *reclaiming* the memory they occupy. Garbage collection is a wonderful feature of modern languages that frees you from having to worry about a large number of memory issues. Memory is automatically allocated to objects when they come into instance. When they're no longer in use, that memory is claimed for reuse.

Memory reclamation is a vital process that helps ensure programs run smoothly and that they can stay up for indefinite periods. Next to hardware failure, memory leaks are the greatest threat to this ability. Memory is said to leak when a programmer expects an object to be reclaimed but the garbage collector can't do so. Usually this is because of programmer oversights (like scope-widening injection). These unreclaimable zombie objects no longer serve any purpose, yet they cannot be claimed by the garbage collector because it cannot verify that they are not in use. Over time these objects can accumulate and consume all of a machine's memory, causing loss of performance and even program crashes.

So how does scope widening feed this problem? Take the case of a singleton. A singleton is held by an injector forever, that is, for the injector's (and usually the application's) entire life span. Therefore, a singleton is never reclaimed during the life of an application. Singletons are generally reclaimed only on shutdown. If any nonsingletons are accidentally wired to a singleton through scope-widening injection, it follows that these objects are also (indirectly) held forever. Here's a naïve example of a servlet that holds onto no-scoped instances on every request:

```
@Singleton
public class BadServlet extends HttpServlet {
    public final Provider<WebPage> page;
    public final List<WebPage> pages = new ArrayList<WebPage>();

    public void service(HttpServletRequest request,
        HttpServletResponse response) {
        pages.add(page.get());

        pages.get(0).handle(request, response);
    }
}
```

BadServlet is a frivolous example, but it serves to illustrate how a no-scoped dependency (WebPage) can be scope widened and held by a singleton, as shown in figure 6.14.

If you continually refresh the URL served by this servlet, your web server will eventually run out of memory and die with an `OutOfMemoryError`.

A simple solution is to discard instances as soon as they are used:

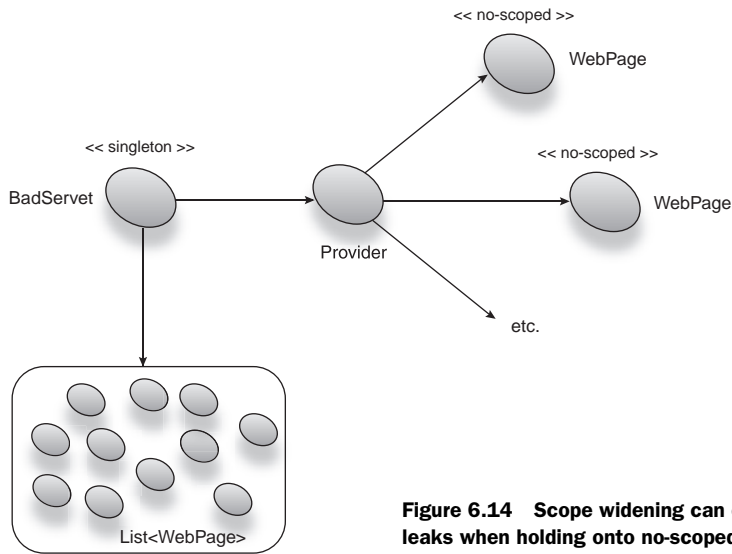


Figure 6.14 Scope widening can cause memory leaks when holding onto no-scoped instances.

```

@Singleton
public class GoodServlet extends HttpServlet {
    public final Provider<WebPage> page;

    public void service(HttpServletRequest request,
        HttpServletResponse response) {
        page.get().handle(request, response);
    }
}
  
```

The following method uses a no-scoped dependency but discards it immediately:

```

public void service(HttpServletRequest request, HttpServletResponse
    response) .. {
    page.get().handle(request, response);
}
  
```

GoodServlet servlet is properly behaved and allows instances of WebPage to be reclaimed by the garbage collector, and all is well with the universe. Scope-widening injection contains many such dangers. Avoid it where possible and try to use simpler designs, either by using the narrowest scope available or by choosing the Provider and Adapter design patterns as we've seen.

6.3 *Leveraging the power of scopes*

If there's one feature of dependency injection that you can't do without (besides wiring), it would be scope. Scopes apply the Hollywood Principle to the state of objects. They are a powerful way of removing boilerplate and infrastructure logic from your code. The importance of this cannot be understated, since it means your code is easier to read and test. Scopes also reduce objects' dependence on one another and therefore simplify your object graphs.

While many DI libraries provide scopes out of the box, and some web frameworks round out the complement with advanced scopes (like *flash* and *conversation*), there's still plenty of room for innovation. We saw just how a custom *transaction* scope can help make components easier to code and test. By pulling the menial labor into behind-the-scenes code, you save yourself a tremendous amount of repetition, coupling, and extra testing. That being said, here are some ideas for leveraging the power of scope.

6.3.1 Cache scope

Objects that are expensive to create (or data that is expensive to retrieve from a store) are especially conducive to in-memory *caching*. Caching is the idea that an object is kept around after its initial fetch, so that additional requests for it can use the cached copy rather than go through the (potentially expensive) process of obtaining it again.

A cache, specifically a disk cache, is powerful in that it can maintain the state of certain important objects across multiple lives of an injector (and, indeed, an application). A particular state can be restored this way for continuing work. In a computer game you play through several difficult levels, then are called away to more mundane pursuits, and so have to close down the game. But you don't want to play through all those levels again. A disk cache can let the application save game state for later resumption. Figure 6.15 imagines how this might look.

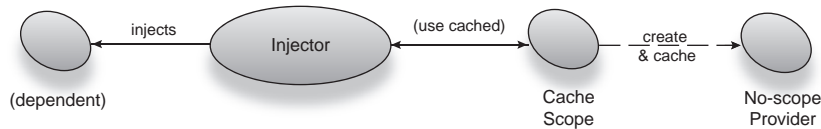


Figure 6.15 The injector obtains instances transparently from the cache scope.

This works in any scenario where long-lived data needs to be saved across injectors. A cache is also powerful for keeping a handle on data constants (such as a list of countries) that rarely change. As the application requests cached items by wiring them in as dependencies, the injector goes to an external store (or creates an instance) for the object. Any subsequent wiring is much faster using the cached copy.

6.3.2 Grid scope

A data grid is very much like a cache in that it is an in-memory store of frequently used data. However, it is a cache that is common across a large number of physical (or logical) machines. These individual machines share objects and act in concert as a *grid*. Grids are sometimes referred to as cluster *caches*. Some sophisticated grids (like Oracle Coherence) do much more than spread objects across a cluster, however. Coherence features querying and updating operations, very much like a database. Grids can also hold objects for very long periods of time, since they are able to distribute the storage load across many more machines than single-disk caches. As a result, they are also often faster and more flexible in clustered applications.

A grid scope that takes advantage of persisting instances in a data grid has no end of useful applications. Grid scoping allows you to transparently cluster a massive number of stateful services and data. This is done transparently so that the load is efficiently balanced across machines without tedious clustering logic and extra service programming. Figures 6.16 and 6.17 show how an instance can be injected in one node and transparently distributed across the grid, to be used by dependents anywhere.

The simplest use case is replication of user credentials across a cluster. A user logged into a single machine may have some relevant data stored in an HTTP session, but if the application is backed by several machines in a cluster, you need to ensure the credentials are also clustered correctly. Grid scoping is a compelling way of creating *single-sign-on* semantics across even different kinds of applications. Single sign-on is the idea that you log in once, on any application in a domain, and you are logged in to all applications on that domain. Here's how you might share a user's relevant information across a data grid:

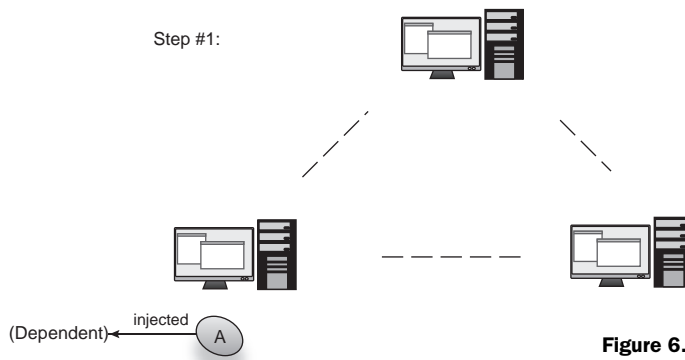


Figure 6.16 Key A (new instance) is provided to a dependent in one node.

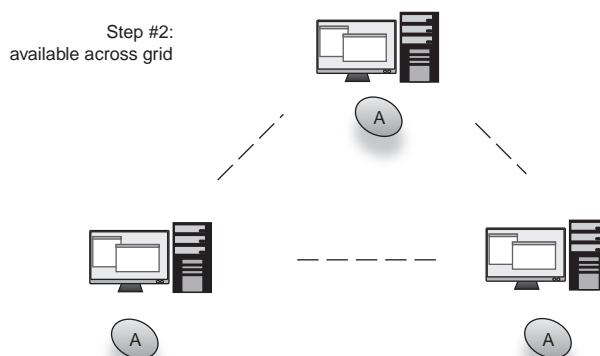


Figure 6.17 Now a grid-scoped instance of A is transparently available to the entire grid.

```
@GridScoped @ThreadSafe
public class Users {
    private final Map<String, UserContext> users;

    ...
}

@NotThreadSafe
public class UserContext {
    private String username;
    private boolean isActive;
    private Preferences prefs;

    ...
}
```

Thereafter, servlets, EJBs, service objects, mail components, and many others can all share a user's information by simply injecting the `Users` object. Of all the powerful applications of scope, grid scope is perhaps the most interesting since it offers so much potential for horizontally scaling applications.

6.3.3 *Transparent grid computing with DI*

The possibilities for client applications to scale and grow are endless, with SOA and enterprise systems cleverly using dependency injection's wonderful design patterns. We've seen the modularity and separation of responsibilities that this can bring. We've also seen how this can help us scale by moving applications off to dedicated hardware. And we've seen the flexibility it can bring in terms of accommodating new business processes and even interbusiness collaboration.

There are a few more exciting possibilities. The kind of scaling we've seen thus far, adding stronger hardware to individual layers of an application, is known as vertical scaling or tiering. You will often hear architectures referred to as an n-tiered or vertically tiered system. This kind of scaling has its advantages, but it can get expensive very quickly. And for very large amounts of traffic, it can even become untenable. An increasingly popular alternative system is known as horizontal scaling.

Horizontal scaling refers to an architecture that increases its processing capacity by the addition of any number of homogenous nodes, each capable of performing the same function as any other. Incoming traffic is evenly distributed across several of these worker machines from a few strong-load distributors. This kind of hardware is both cheap and easily replaceable. And as it turns out, it is extremely powerful when scaling to very large loads.

Apache's Hadoop project is an example of one such homogenous cluster-computing system. It consists of a master machine that distributes a single, computationally expensive task across a number of worker slaves. Then it gathers results from each one and recombines (or reduces) them into an expected total order. This is incredibly useful for searching across large datasets and performing expensive computations that can be broken down into simpler ones.

One possible use of DI would be to apply a similar abstraction over the grid computing network, where one interface and one method call are orchestrated across several machines using a grid service proxy, in much the same way as we used a remote proxy in the SOA case. There are early attempts at performing this sort of orchestration, but most of the effort has been focused on transparent slicing and distribution of tasks around the cluster rather than architectural design patterns for clients of such services. A library like that combined with a thin integration layer for clients could really make dependency injection shine.

6.4 **Summary**

Injectors are extensible via the definition of custom scopes. These are scopes that match some particular context of a problem domain that would benefit from the application of the Hollywood Principle. Database transaction scope is one such example. Keys bound within this scope return the same instance while a transaction is active in the current thread but return new instances in other transactions (either subsequent or concurrent). Be careful when crafting your own scopes, and adhere to the following principles:

- Define the context of a scope clearly and succinctly.
- Try to keep contexts confined to a single thread.
- Assess whether your scope improves testability of components. If not, stay away! There is no end to the pain that can be caused by a presumed performance or development improvement that ends up not being easy to test.

Custom scopes really require a lot of thought and planning. Objects wired across scoping boundaries can cause serious problems that can go undetected even with comprehensive unit testing.

The problem of “crossing” a scoping boundary is very serious indeed—wiring a narrow-scoped object (that is, a no-scoped one) to a wider-scoped object (a singleton) entirely changes the assumptions about that narrower scope. The instance becomes held by the singleton and effectively lives for the same amount of time. You should try to design an architecture so this scope-widening injection is rare. Binding all keys in the object graph under the narrowest scope is one simple mitigation. However, this doesn’t work for all cases; sometimes you need a singleton that depends on a no-scoped (or narrower-scoped) object. You can solve problems in this case by applying the Provider design pattern from chapter 3.

Another issue with scope widening is objects that are out of scope. An out-of-scope object is one for which a key is requested when its bound scope is not active, for example, a request-scoped key wired outside an HTTP request (say, during application startup). Such problems represent design flaws. Rethink the architecture of your long-lived components and their collaborators. If you must, solve the problem with the Adapter pattern, as shown in chapter 3.

A third issue that arises from scope-widening injection is thread-safety. Keys of narrower scope, when wired to a singleton, are susceptible to the action of multiple

concurrent threads. It is not enough to secure the singleton component alone; you must guard access to its narrower-scoped dependencies or ensure that they are immutable. This problem often goes undetected and can lead to erratic, unpredictable behavior (without any specific errors being raised), which can be a nightmare to debug. There is no easy way to detect thread-safety issues in scope-widened cases, since unit tests (and even integration tests) are not reliable in this regard. Think about your object graphs and convince yourself (and preferably an educated colleague) that they are thread-safe. Better yet, avoid scope-widening injection altogether.

Finally, scope-widening injection can cause memory leaks if the wider-scoped object holds onto its narrower-scoped dependencies. Memory leaking is impossibly hard to detect and results in arbitrary application crashes, which can be disastrous in production environments.

Although scopes have many pitfalls that may scare you off writing your own, you shouldn't be afraid. These are some positive features of scope:

- Reduction of boilerplate
- Reduction of interdependent code
- Reduction (or removal) of dependence on infrastructure
- Logical use of the Hollywood Principle to create focused, concise code
- Improved testability

The benefits vastly outweigh any perils. Consider the careful use of scopes, and design them according to your problem domain. Scopes can make the use of caches, clusters, and data grids transparent to vastly increase an application's scalability and give it powerful features such as stateful replication.

We've spent two chapters studying the problems and benefits of scope. In the next chapter let's change the theme slightly and look at a broader topic, applicable to all kinds of applications, object lifecycle.



From birth to death: object lifecycle

This chapter covers:

- Notifying objects of significant events
- Understanding domain-specific lifecycle
- Initializing lazy and eager singletons
- Customizing lifecycle with multicasting

“I agree with everything you say, but I would attack to the death your right to say it.”

—Tom Stoppard

Whether or not lifecycle is a part of dependency injection is a divisive issue. However, like scoping, it can be a powerful tool when used correctly, and it fits closely with dependency injection. In this chapter we’ll look at the basic form of lifecycle offered by the language runtime—constructors. We’ll also look at managed lifecycle as offered by more elaborate frameworks like servlets and EJBs, to illustrate how lifecycle is domain specific. On the way we’ll examine the pitfalls of relying on one-size-fits-all lifecycle models.

Finally, we’ll look at how to design and implement a custom lifecycle strategy and design classes that are simple and easy to test. First, let’s start by looking at what lifecycle is: events in the life of an application that an object is notified about.

7.1 Significant events in the life of objects

Lifecycle and scope are closely related concepts, so much so that they are occasionally confused with each other. An object's longevity can be seen in these two dimensions. Object longevity with regard to some context (duration, users, threads, and so on) is its scope. Periods in this context indicate various things about an object, for instance, whether it is ready to begin serving clients, or if the object represents a network resource, whether that resource has been closed or abnormally interrupted.

Clearly many of these *states* are specific to the nature of the object. A movie may have a *paused* state in its life; a web application has a *deployed* state, a game has a *game-over* state, and so on. The various states an object goes through in its life are collectively known as its lifecycle. When an object transitions between such states, it is said to undergo lifecycle changes. These are often modeled as events that an object responds to (see figure 7.1).

The most basic lifecycle events are *create* and *destroy*.

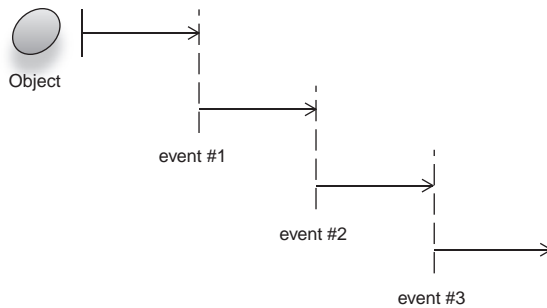


Figure 7.1 An object is notified of events as parts of its lifecycle.

7.1.1 Object creation

Objects come into instance when they are created and are notified about this event by a constructor. Language runtimes support this construct naturally, so this isn't really rocket science. After memory is allocated to an object, its constructor is called, which can perform some initialization logic. Ideally, one would do everything that's necessary on construction (any computation, dependency wiring, and the like) to put an object into a usable state. However, this isn't always possible for reasons that we'll discuss shortly.

In chapter 3 we saw how the constructor is used to wire an object with its dependencies. We saw that it is a powerful technique for creating objects that are good citizens. Good citizenry naturally extends to the lifecycle of an object. Constructing an object (with its dependencies) puts it in a usable state, and thus begins its lifecycle.

Listing 7.1's `Transporter` is notified via its constructor, so that it can prepare itself for use.

Listing 7.1 `Transporter's` lifecycle is begun with a call to its constructor

```
public class Transporter {
    private final ControlPanel control;
    private final PowerCell cell;
```

```

public Transporter(ControlPanel control, PowerCell cell) {
    this.control = control;
    this.cell = cell;

    cell.charge();
    control.activate();
}

public void energize() { .. }

```

Constructor is notified, begins lifecycle

Ready to be used

Figure 7.2 models these classes. In `Transporter`'s case, it calls into its dependencies first (as per figure 7.3).

Following this, `Transporter` is in the ready stage of its life. Method `energize()` can be called safely, to send crew members off on wild planetary adventures (see figure 7.4).

Furthermore, any preparatory computations are also a natural fit to the constructor:

```

public Transporter(ControlPanel control, PowerCell cell) {
    this.control = control;
    this.cell = cell;

    cell.charge();
    control.activate();
}

```

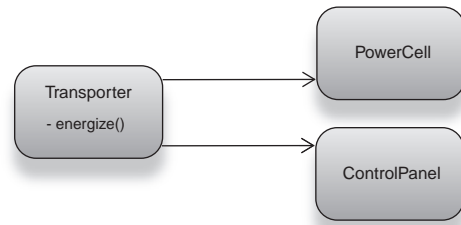


Figure 7.2 A `Transporter` depends on a `PowerCell` and `ControlPanel`.

There's nothing spectacular about this example or about using constructors to prepare objects, but the example serves to illustrate the idea behind lifecycle with states

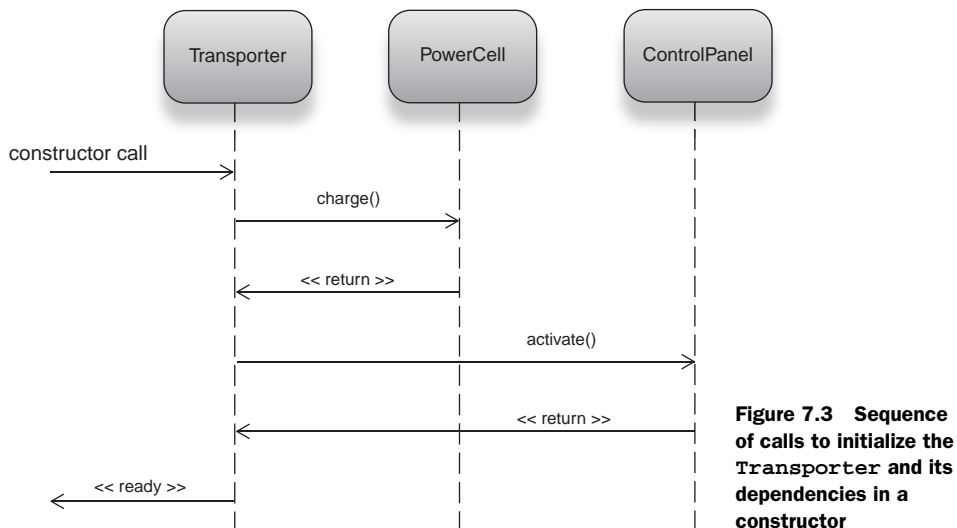


Figure 7.3 Sequence of calls to initialize the `Transporter` and its dependencies in a constructor

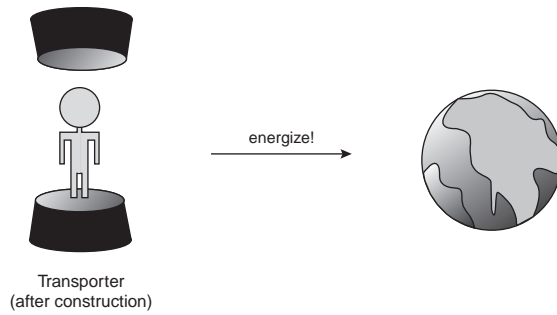


Figure 7.4 Once the constructor lifecycle method runs, we are ready for adventuring!

and about certain tasks that need to be performed to put an object into such states. Similarly, when a service is shut down, resources often need to be relinquished (memory reclaimed, sockets closed, and so on). This occurs when a service is destroyed.

7.1.2 Object destruction (or finalization)

In Java, the explicit task of allocating and reclaiming memory is taken care of by the runtime environment. This frees you from having to worry about when to reclaim allocated objects. Premanaged languages like C++ support an explicit *destructor*, which acts as a counterpart to constructors. The destructor is a lifecycle event method, which runs once, just prior to memory being freed. Java has something analogous: *finalizers*. A finalizer is a method that runs just prior to an object being reclaimed by the runtime garbage collector.

Unlike in C++, in Java we cannot guarantee when an object will be reclaimed and therefore when its finalizer runs. The only certainty is that a finalizer *will* be run before the object is reclaimed. This may be often and early (in the case of aggressive garbage collection), or rare and late, or even right at the shutdown of the application itself, or never if the application exits abnormally. For these reasons, disposal of finite resources (such as network sockets or database connections) inside a finalizer is not a good idea.

To illustrate why, let's take the example of a file service that displays the contents of various images in a directory (like a filmstrip preview). This service is described in listing 7.2.

Listing 7.2 This application displays impressions of images on a disk

```
public class Thumbnail {
    private final FileSystem fileSystem;
    private InputStream data;

    public Thumbnail(FileSystem fileSystem) {
        this.fileSystem = fileSystem;
    }

    public Image display(String path) throws IOException {
        data = fileSystem.getPath(path).newInputStream();
        ...
    }
}
```

InputStream
opened on
display

```

    }

    @Override
    protected void finalize() throws Throwable {
        data.close();
        data = null;
        super.finalize();
    }
}

```

← **InputStream closed
in finalizer (bad!)**

Each time the user flips to a new image, the Thumbnail object opens a new file and displays it. However, the input stream is closed only inside its `finalize()` method. Since there is no guarantee of when an object may be reclaimed, it's a real possibility that the program will open too many files from the operating system and thus run out of resources (see figure 7.5). This is obviously a poor use of the finalizer lifecycle.

Most times, you will find a better lifecycle method to release finite resources than a finalizer. So, should you never use finalizers? Not quite—there are some rare cases when they come in handy. While you can't rely on a finalizer to release finite resources, you can use a finalizer to *ensure* that a resource has been released previously:

```

@Override
protected void finalize() throws Throwable {
    if (null != data) {
        logger.warning("unclosed thumbnail: " + this);
        data.close();
    }

    data = null;
    super.finalize();
}

```

This is a sanity check that can help in cases where you mistrust clients of your code. Strictly speaking, this isn't common, but it's occasionally useful. Alternatively, you might use logging or profiling APIs inside a finalizer to test predictions about your program's performance.

One other use case for finalizers is releasing memory in collaborating libraries that are not managed. For example, memory resources of a native C library running inside a Java program can be released inside a finalizer by calling a native method (see figure 7.6).

These are corner cases that you are unlikely to encounter in everyday programming. However, it's important to understand the role of finalizers and more important to avoid *abusing* them. More on the pitfalls of finalization is presented in chapter 9.

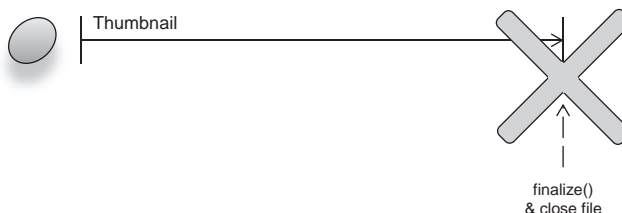


Figure 7.5
Method `finalize()` may run very late, so it's unsuitable for closing finite resources.

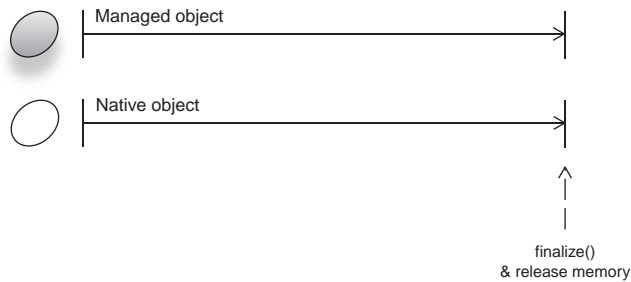


Figure 7.6 It's sometimes useful to release memory in collaborating native libraries with `finalize()`.

Construction and finalization are universal events that apply to any kind of object. But most types of lifecycle apply only to specific classes of objects—and then only in certain application domains. Remember that not all lifecycle is universal and that events are typically specific to particular problem domains.

7.2 One size doesn't fit all (domain-specific lifecycle)

We've seen that constructors are a lifecycle hook on creating objects and that finalizers are their complement prior to destruction. We've also seen that finalizers don't fit all clean-up use cases. Constructors are a bit better, but they too don't fit all initialization use cases. For example, a web application is deployed, and all its objects are constructed quite early. But it may only begin servicing requests at a much later time.

The application doesn't require many of its (finite) resources until that point. In the Java Servlet Framework, this directly translates to a servlet's `init()` method, where resources such as database connections can be acquired and held. Similarly, a network service may be created and configured but is not fully ready until a socket is opened.

This leads to the conclusion that lifecycle events are *not* universal and that the stages in an object's life are *specific* to a problem domain. For instance, the lifecycle of a servlet is very different from the lifecycle of a database connection or that of a movie player. Consequently, the times and frequency of initialization and destruction are dependent on the nature of the service. Web pages are created and destroyed frequently (on each request), while database connection pools are created once and held open almost indefinitely.

In this chapter, we'll look at some of these domains, how they differ, and how to design with them in mind. Let's start by contrasting two very common problem domains that have very different lifecycles: web servlets and database connections.

7.2.1 Contrasting lifecycle scenarios: servlets vs. database connections

A *servlet* is a web component used to render web pages that has three major stages in its lifecycle: *constructed*, *ready to service*, and *destroyed*. These are demarcated by two lifecycle events: *init* and *destroy*. A servlet's `init()` method is the lifecycle *hook* that's called to notify it of initialization. In other words, when the `init()` method returns, the servlet is expected to be ready for service. Similarly, method `destroy()` is called to notify a servlet that its life has ended. Any cleanup of the servlet's dependencies should be

performed here. Once destroyed, a servlet object is never called on to service requests again. Listing 7.3 shows a servlet object that initializes itself in its constructor and opens a connection to the database when notified of the init event. It releases this connection on servlet destruction. This sequence is illustrated in figure 7.7.

Listing 7.3 A Java servlet that starts and cleans up its resources inside lifecycle hooks

```
public class NewsServlet extends HttpServlet {
    private Connection con;
    private final NewsService newsService;

    public NewsServlet() {
        newsService = new NewsService();
    }

    @Override
    public void init() throws ServletException {
        try {
            con = DriverManager.getConnection(..);
        } catch (SQLException e) {
            ...
        }
    }

    @Override
    public void service(ServletRequest req, ServletResponse res) { .. }

    @Override
    public void destroy() {
        try {
            con.close();
            con = null;
        } catch (SQLException e) {
            ...
        }
    }
}
```

Create nonfinite
dependencies

Open database
connection

Close and release connection

There are some oddities about `NewsServlet`: Because it is managed directly by the servlet framework, it cannot benefit from dependency injection. Its lifecycle (that is, `init()` and `destroy()`) is managed entirely by the servlet container. It introduces the need to create dependencies by hand (or use a factory) and tightly couples the servlet to the underlying data layer. By inserting an integration layer like `guice-servlet`, we can change all that and have `NewsServlet` benefit from dependency injection as well as receive the servlet lifecycle events. Listing 7.4 reimagines `NewsServlet` in this light.

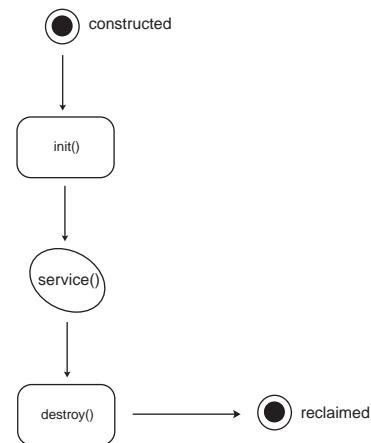


Figure 7.7 Stages in a servlet's lifecycle

Listing 7.4 A version of NewsServlet managed by Guice and guice-servlet

```

@Singleton
public class NewsServlet extends HttpServlet {
    private final PersistenceService persistence;
    private final NewsService newsService;

    @Inject
    public NewsServlet(PersistenceService persistence,
        NewsService newsService) {
        this.persistence = persistence;
        this.newsService = newsService;
    }

    @Override
    public void init() {
        persistence.start();
    }

    @Override
    public void service(ServletRequest req, ServletResponse res) { .. }

    @Override
    public void destroy() {
        persistence.shutdown();
    }
}

```

Abstract, nonfinite dependencies are injected

Start persistence service

Shut down persistence service

This version of NewsServlet is much neater, not simply because it is injected, but also because its lifecycle can naturally cascade to abstractions underneath. For example, PersistenceService may represent a database connection, flat disk file, or even in-memory network storage. Dependency injection in conjunction with the servlet lifecycle makes for simpler, more testable code.

Spring's DispatcherServlet provides a similar facility for routing requests from the servlet container to its own custom MVC framework. First you set up the DispatcherServlet in web.xml. Then you hook it up to a custom Spring MVC controller that does what you want (and is dependency injected by Spring):

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean name="/news.html" init-method="init" destroy-method="destroy"
        class="c7.NewsController">
        <constructor-arg ref="persistence"/>
        <constructor-arg ref="newsService"/>
    </bean>
    ...
</beans>

```

Here instead of implementing HttpServlet directly, we implement Spring's Controller interface:


```

import org.springframework.web.servlet.mvc.Controller;
...

public class NewsController implements Controller {
    private final PersistenceService persistence;
    private final NewsService newsService;

    public NewsController(PersistenceService persistence,
        NewsService newsService) {
        this.newsService = newsService; this.persistence = persistence;
    }

    public void init() {
        persistence.start();
    }

    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) { .. }

    public void destroy() {
        persistence.shutdown();
    }
}

```

Start persistence service
Shut down persistence service

Except for the variant `handleRequest()` method, this is very similar to the previous example.

NOTE Notice that we had to specify `init`-method and `destroy`-method in the XML configuration; this is necessary since we're moving from the servlet container to Spring's injector. While these do not exactly coincide (as in the case of Guice Servlet), they perform the same function.

Contrast this with the lifecycle of a different service, such as a database connection. Both can be managed by DI, and both have a role to play in web applications, but they lead very different lives. Listing 7.5 describes a fictional, pooled database connection class.

Listing 7.5 Connection wraps a raw database-driver connection for pooling purposes

```

public class PooledConnection {
    private Connection conn;
    private ConnectionState state;

    public synchronized void open() throws SQLException {
        conn = DriverManager.getConnection(...);
    }

    public synchronized void onCheckout() {
        this.state = IN_USE;
    }

    public synchronized void onReturn() {
        this.state = IDLE;
    }

    public synchronized void close() throws SQLException {
        if (IN_USE == state)

```

Connection is established
Connection is checked out for use
Connection is checked back in
Connection is disposed, if not in use

```

        throw new IllegalStateException();
    }
    conn.close();
}

```

In listing 7.5, there are four lifecycle hooks: `open()`, `close()`, `onCheckout()`, and `onReturn()`. These refer to events in the life of the database connection. `open` and `close` are self-explanatory; `onCheckout()` refers to the connection being taken out of its pool for use. Think of this as checking out a book from your local public library. When notified, the pool-aware connection puts itself in an `IN_USE` state, which is used to ensure that it isn't accidentally closed while serving data. Similarly, `onReturn()` is called when the connection is checked back into its pool, setting it into an `IDLE` state. Figure 7.8 is a flow chart describing the sequence of events in `Connection`'s lifecycle.

Idle connections are safe to close or to perform other background activity on (for example, connection *validation*).

The connection's lifecycle hooks are called from an owning connection pool. As previously said, its usage profile is significantly different from a servlet's—and indeed any other object managed by dependency injectors. The pool itself may be managed by a servlet and cohere with its lifecycle; however, this is not particularly relevant to the connection's lifecycle.

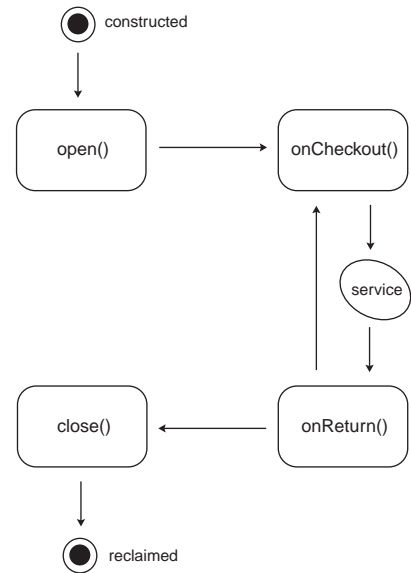


Figure 7.8 Flow of stages in an example database connection's lifecycle

Connection pooling and validation

Connections to RDBMS databases often have timeout values. After a certain period of inactivity, a driver automatically releases a held connection in order to prevent zombie connections from dragging down database performance. Since establishing new connections is potentially expensive, applications create and hold them in pools (at startup), taking the cost up front. Connections are then distributed from the pools to clients and returned after use, avoiding the need to dispose and reestablish connections during the life of busy applications. This is known as *connection pooling*.

If a connection remains idle in a pool for any length of time, it's in danger of being timed out. To prevent this, connection pools execute dummy SQL statements at periodic delays to reset the timer and keep the connection alive. Often these are as simple as `"SELECT 1 FROM DUAL"`, which returns the number 1 from a numerical sequence. This process is known as *connection validation*.

Just as it is difficult to fit connections and servlet into the same lifecycle, it is also difficult for any one lifecycle model to fit all use cases. Many programmers lead themselves into trouble by trying to do just that. One such anti-pattern is trying to apply a converse of the initialization lifecycle event to all services. I call this the destructor anti-pattern.

7.2.2 *The Destructor anti-pattern*

As we said earlier, finalizers are not well suited to freeing up finite resources in languages with managed memory models. To solve this, many developers use an all-purpose *destroy* lifecycle hook, which is called when the injector (or application itself) is shut down. Spring and PicoContainer provide these all-purpose destroy methods, which are called at some point in the application's lifecycle but not necessarily just when an object is being reclaimed. Here is the file service from earlier in the chapter, modeled using Spring's built-in destroy event:

```
<bean id="thumbnail" class="files.Thumbnail" destroy-method="close">
  <constructor-arg ref="fileSystem"/>
</bean>
```

The attribute `destroy-method=".."` refers to the name of a no-argument method to be called when the injector is shut down. While these are often useful in particular problem domains (such as the servlet lifecycle we saw previously), generalizing this event is not always appropriate. Take the `Thumbnail` example at the head of the chapter; the same problem occurs here as we saw with finalizers. Method `close()` is called when the application exits (Spring's injector calls destroy hooks when the JVM exits). This means that file handles will be held open while images are being viewed. Once again, you can very easily run out of finite OS resources without really using them.

A more sensible solution would be to close each file handle after the image preview has been extracted from it. There isn't a clear way to do that using only the `destroy-method` mechanism or, more generally, using the Destructor pattern.

Thus, indiscriminate use of this all-purpose lifecycle hook can lead to unintuitive designs. It's not uncommon to see Java Swing windows being provided a destructor. Often these are used to dispose of graphical widgets no longer needed within the window. The irony is that Swing widgets don't need to be disposed explicitly. Like any other Java object, Swing widgets are managed and reclaimed by the language runtime and require no explicit memory management.

Without a well-understood destruction hierarchy, it becomes difficult to predict the order in which objects are destroyed. For example:

```
<bean id="thumbnail" class="files.Thumbnail" destroy-method="close">
  <constructor-arg ref="fileSystem"/>
</bean>

<bean id="fileSystem" class="files.ZipFileSystem" destroy-method="close"/>
```

It's difficult to predict the order in which `ZipFileSystem.close()` and `Thumbnail.close()` will be called. If one depends on the other while shutting down, this may lead to accidental illegal states where a dependency has already been closed but a

dependent still needs it to shut itself down. It's not unusual to find conditions like the following, to account for just such a scenario:

```
public void close() {
    if (null != reader) {
        if (reader.isOpen())
            reader.someLastMinuteLogic();

        reader = null;
    }

    shutdownMyself();
}
```

Such contortions are abstruse and unnecessary, especially when we have managed memory models and patterns like DI to fall back on. In this light, I strongly discourage using the jack-of-all-trades destroy method or the Destructor anti-pattern. A good alternative is to pick domain-specific finalization patterns. For I/O, Java's `Closeable` interface functions nicely.

7.2.3 Using Java's `Closeable` interface

Not all objects can be left to garbage collection to be disposed of. Often, an external resource needs to be closed explicitly, at a time other than the application's exit, as we saw in the case of the thumbnail image viewer. Global, one-time destructors are ill-suited for the reasons we've just seen.

An object that acts as a data endpoint can be designed with *closeability* in mind if it exposes the `java.io.Closeable` interface. Here's a definition from the API documentation:

"A `Closeable` is a source or destination of data that can be closed. The `close` method is invoked to release resources that the object is holding (such as open files)."

Modeling finite resources (files, network sockets, and so on) as `Closeables` allows you to redact them into lifecycle stages. For example, a look-ahead caching service could open and read image data in the background. When they are finished being read, these files would be placed into a queue to be closed periodically by a lower-priority thread:

```
@ThreadSafe
public class ResourceCloser implements Runnable {
    @GuardedBy ("lock")
    public final List<Closeable> toClose = new ArrayList<Closeable>();

    private final Object lock = new Object();

    public void run() {
        synchronized(lock) {
            for (Closeable resource : toClose) {
                try {
                    resource.close();
                } catch (IOException e) {
                    ...
                }
            }
        }
    }
}
```

```

        }
    }
}

public void schedule(Closeable resource) {
    synchronized(lock) {
        toClose.add(resource);
    }
}
}

```

NOTE The annotation `@GuardedBy` indicates that access to mutable list `toClose` is guarded by a field named `lock`. Like `@ThreadSafe`, `@GuardedBy` is simply a documenting annotation and has no effect on program semantics.

Resources to be closed are enqueued via the `schedule()` method and closed periodically by the `run()` method. We won't worry too much about how this method is scheduled to run or how often. Let it suffice to say that it happens periodically and at a lower priority than other threads in the application. This satisfies the close lifecycle of a file handle and doesn't suffer the problems of destructors. So far, we've looked at how lifecycle is domain-specific and that while objects share common events like construction and finalization, they cannot be shoehorned into initialization and destruction patterns universally. To illustrate, let's examine a scenario that highlights how complex and specific lifecycle can get. Let's look at stateful EJBs, which are common in many real-world business applications.

7.3 A real-world lifecycle scenario: stateful EJBs

So far we've seen fairly simple lifecycle hooks in the form of constructors, some more involved lifecycles with servlets and pooled database connections, and some pitfalls in generalizing these models. Now let's take a look at a more complex case: *stateful EJBs*. EJBs are a programming model for business services. They are managed directly by an application server (commonly, a Java EE application server), and as such their lifecycle is also controlled by it. There are several types of EJBs, suited to various purposes such as persistence, message-passing, and business services. Stateful EJBs at a particular kind of business service are sometimes called *stateful session beans*.

These are basically objects that interact with a client around some specific context. They are called stateful because they can maintain this context across multiple interactions from the same client, in much the same manner as an HTTP session does with a browser client. As of this writing, the EJB specification is in version 3.0, which has some dependency injection features, where EJBs can be provided to one another via annotated fields. Similarly, they also have a controlled lifecycle, where the bean is notified of major events by the application server.

Listing 7.6 is an example of a stateful EJB representing a shopping cart. Its clients may be a website front end (like Amazon.com), a desktop application at a checkout counter, or even another EJB.

Listing 7.6 A stateful EJB representing a returning customer's shopping cart

```

@Remote
public interface ShoppingCart {
    void add(Item item);
    List<Item> list();
}

@Stateful
public class ShoppingCartEjb implements ShoppingCart {
    private List<Item> items = new ArrayList<Item>();
    private double discount;

    @EJB
    private InventoryEjb inventory;    ← Injected by EJB container

    @PostConstruct
    public void prepareCart() {
        discount = inventory.todayDiscount() * items.size();    ← Compute initial discount
    }

    @Remove
    public Status purchase() {
        return inventory.process(items, discount);    ← Process purchase and dispose EJB
    }

    public void add(Item item) { items.add(item); }
    public List<Item> list() { return items; }    ← Shopping cart public methods
}

```

In listing 7.6's `ShoppingCartEjb`, all methods except `add()` and `list()` are EJB lifecycle hooks. Method `prepareCart()` is marked with `@PostConstruct`, indicating to the EJB container that it should be called on initialization. `purchase()` performs a dual role: It processes the order in the shopping cart, returning a status code, and also tells the EJB container to end the stateful bean's lifecycle. Another lifecycle hook, `@PreDestroy`, is available for any explicit cleanup actions that need to occur on a *remove* event.

NOTE The annotation `@EJB` is used in much the same manner as Guice's `@Inject` or Spring's `@Autowired` but only on a field. This tells the EJB container to look up the appropriate EJB and inject an instance. EJB provides basic dependency injection in this fashion. This is undesirable because of the inability to replace with mock objects and for several other reasons outlined in chapters 3 and 4 (EJB does not support constructor injection as of version 3.0).

This is certainly quite a complex set of semantics for a simple shopping cart service. But it gets better—because a stateful EJB is meant to keep a running conversation with its client (until `@Remove` is reached), two more lifecycle hooks are available. These are used to transition a stateful EJB from its active servicing state to a passive state (while the client is off doing other things) and then back again to an active state when the client has returned.

During this passive time, it may be useful to have the shopping cart data stored in a more permanent storage medium like a database or disk cache. EJB provides a `@PrePassivate` lifecycle hook for this purpose:

```
@PrePassivate
public void passivate() {
    cache.store(items);
}
```

This is useful if there is some kind of failure in the application server. On reactivation, you may want to refresh the items in the shopping cart or recompute discounts to ensure that associated data have not become stale. An administrator may have modified the item's price while the stateful EJB was passive. The `@PostActivate` lifecycle hook is called by an EJB container every time a stateful EJB is reactivated:

```
@PostActivate
public void activate() {
    items = inventory.refresh(items);
}
```

While the EJB programming model is not a direct analog of dependency injectors, it nonetheless serves to illustrate how complex, managed lifecycle can work. This entire flow is illustrated in figure 7.9.

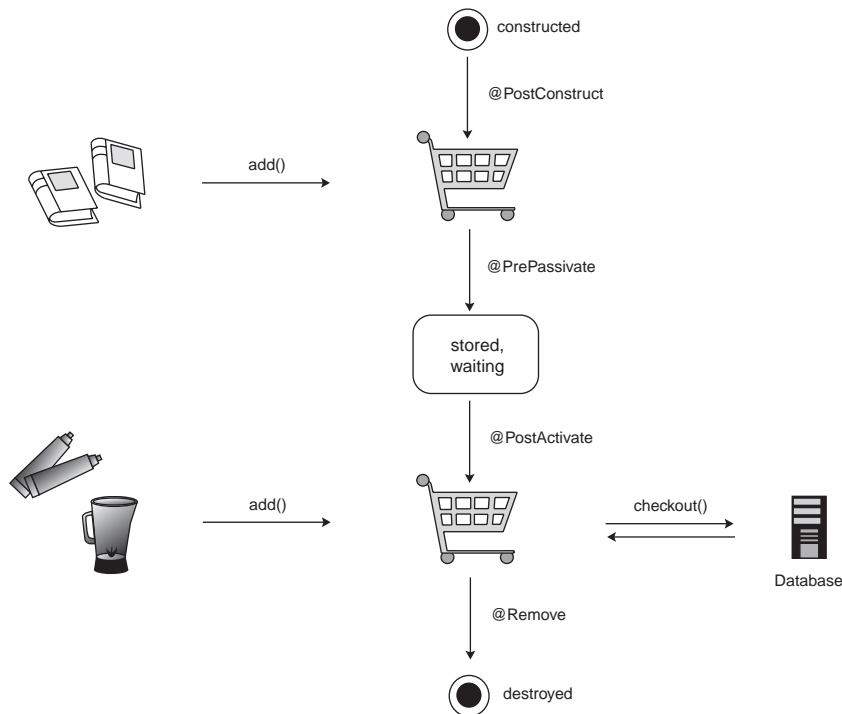


Figure 7.9 The lifecycle of a stateful EJB that models a shopping cart for *returning* clients

You will frequently find the need to apply similar patterns to your own environment, where objects are notified of significant events in their lifespan by an external actor or framework. Keep these examples in mind when designing lifecycle models for your own applications. Another important consideration when designing lifecycle is the timing of events. Not all objects are constructed at the same time. It is often desirable to delay construction until an object is needed. Among other things, this allows for better startup performance and lower memory usage. This type of delayed construction is called *lazy instantiation*.

7.4 Lifecycle and lazy instantiation

Another concept closely related to lifecycle is *on-demand* object construction. We saw this earlier with bootstrapping injectors on demand, per use. It's more common to find dependencies being constructed this way, particularly singleton-scoped objects. Rather than construct a dependency on injector startup, it's created when first needed for injection. In other words, it's created lazily (see figure 7.10).

Lazy creation of instances is useful when one is unsure whether all injector-managed services will be used early or used at all. If only a few objects are likely to be used early on and the remainder may be used infrequently or at a much later time, it makes sense to delay the work of creating them. This not only saves the injector from a lot of up-front cost but potentially also saves on the amount of memory used. It is especially useful during testing or debugging, where several configured services will probably never be used, or in a desktop or embedded application where fast startup is important.

The converse of lazy instantiation is *eager* instantiation. This is where singleton-scoped objects (in particular) are created along with the injector's bootstrap and cached for later use. Eager instantiation is very useful in large production systems where the performance hit can be taken up front, to make dependencies quickly available subsequently, as illustrated in figure 7.11.

The Guice injector can be built with a Stage directive, which tells it whether or not to eagerly instantiate singletons:

```
Guice.createInjector(Stage.PRODUCTION, new MyModule());
```

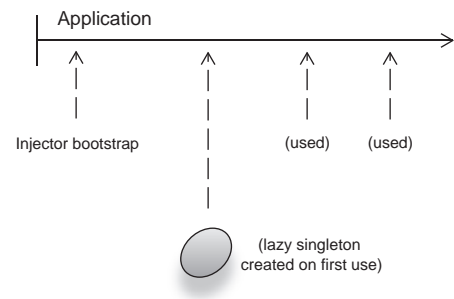


Figure 7.10 Lazily bound objects are created when first used.

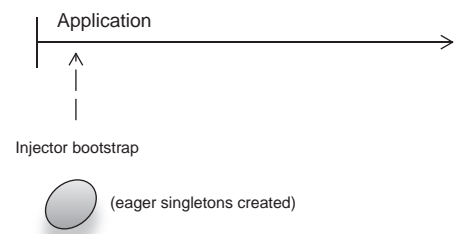


Figure 7.11 Eagerly bound singletons are created along with the injector itself.

← Singletons eagerly created

Or the alternative:

```
Guice.createInjector(Stage.DEVELOPMENT, new MyModule());
```

← Singletons lazily created

You can also force certain keys to bind eagerly:

```
bind(MyServiceImpl.class).asEagerSingleton();
```

Or they can bind implicitly, by binding to a preexisting instance:

```
bind(MyService.class).toInstance(new MyServiceImpl());
```

Note that in the latter case, `MyServiceImpl` does not benefit from dependency injection itself, so this method should be avoided as much as possible. In Spring, this is achieved using the `lazy-init=".."` attribute:

```
<bean id="slothful" class="sins.seven.Sloth" lazy-init="true">
...
</bean>
```

Singleton instance `slothful` is created only when it is first needed by the application. By default, all singleton-scoped objects are created eagerly in Spring. Lazy instantiation as a design idiom is quite common. Before you decide to use it, you should consider issues of timing and performance, particularly during startup.

This chapter has thus far dealt with lifecycle events provided by frameworks like Spring and EJB. Now let's explore how we can create our own kinds of lifecycle. The simplest way to do this using a dependency injector is with post-processing.

7.5 Customizing lifecycle with postprocessing

As you've seen, lifecycle is specific to particular use cases and applications. In this case, the injector cannot help you directly, since libraries cannot ship with all possible lifecycle models (obviously). They can help, however, by providing a means for extending the lifecycle model to suit your own requirements.

Libraries like Spring and Guice allow an instance, once created, to be further processed by some general logic. This may include additional wiring, computation, or registering the instance for later retrieval. And it happens before the instance is injected on dependents. This is called *postprocessing* and is perfect for creating a custom lifecycle model (see figure 7.12).

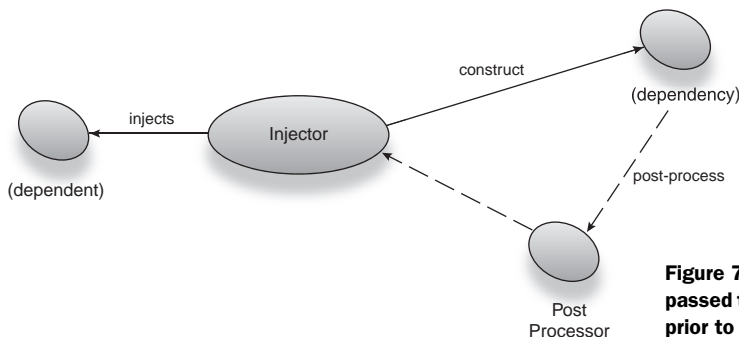


Figure 7.12 Dependencies are passed through the postprocessor prior to injection.

Let's take the example of a shopping arcade where there are several screens showing advertisements and shopping-related information. At midnight, when the arcade closes, these screens need to stop showing ads and display a notice saying that the mall is closed. Each screen's display is controlled from a separate set of feeds, based on their location. Modeling this as lifecycle of a feed (or screen) gives us a flexible and simple solution to the problem.

At midnight a notification is sent to all screens, putting them into a suspended state. Another notification is sent out the next morning when the mall reopens, so that normal programming can resume. Listing 7.7 shows how this Screen class would look in Java.

Listing 7.7 A screen with a timed lifecycle, driven by video feeds

```
package arcade;

public class Screen {
    private final Feed daytimeFeed;
    private final Feed overnightFeed;

    public Screen(Feed daytime, Feed overnight) { .. }

    public void suspend() {
        show(overnightFeed);
    }

    public void resume() {
        show(daytimeFeed);
    }

    ...
}
```

Class Screen has two interesting methods that embody its lifecycle:

- `suspend()`—Switches the screen to its overnight “mall closed” display
- `resume()`—Restores normal commercial programming

Both methods are specific to the shopping arcade problem domain. We need them to be called at specific times (midnight and early morning) to manage the activity of screens around the mall. The lifecycle sequence for this class is illustrated in figure 7.13.

If all instances of Screen are created by a dependency injector, it's possible to postprocess each instance and *register* it for subsequent lifecycle notification. Here's how a postprocessor might look in Spring, along with three independent Screens around the arcade:

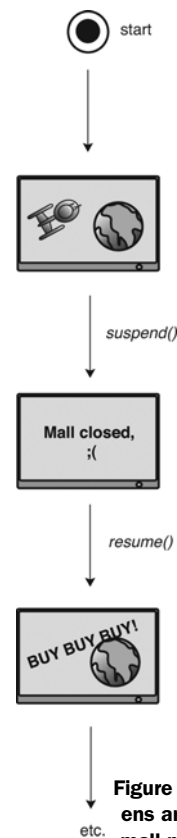


Figure 7.13 Screens around the mall move to a different feed when

```

<beans ...>
    ...

    <bean class="arcade.LifecyclePostProcessor"/>

    <bean id="screen.corridor" class="arcade.Screen">
        <constructor-arg ref="daytimeFeed"/>
        <constructor-arg ref="overnightFeed"/>
    </bean>

    <bean id="screen.foodcourt" class="arcade.Screen">
        <constructor-arg ref="daytimeFeed"/>
        <constructor-arg ref="overnightFeed"/>
    </bean>

    <bean id="screen.entrance" class="arcade.Screen">
        <constructor-arg ref="daytimeFeed"/>
        <constructor-arg ref="overnightFeed"/>
    </bean>

</beans>

```

And the postprocessor implementation, which registers available screens for lifecycle management, is as follows:

```

package arcade;

import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.beans.BeansException;

public class LifecyclePostProcessor implements BeanPostProcessor {
    private final List<Screen> screens = new ArrayList<Screen>();

    public Object postProcessAfterInitialization(
        Object object, String key) throws BeansException {
        if (object instanceof Screen)
            screens.add((Screen)object);
        return object;
    }

    public Object postProcessBeforeInitialization(
        Object object, String key) throws BeansException { .. }
}

```

To understand this, let's dissect the `BeanPostProcessor` interface:

```

package org.springframework.beans.factory.config;

import org.springframework.beans.BeansException;

public interface BeanPostProcessor {
    Object postProcessAfterInitialization(
        Object object, String key) throws BeansException;

    Object postProcessBeforeInitialization(
        Object object, String key) throws BeansException;
}

```

Classes exposing this interface must implement the `postprocess` methods. Their signatures are identical: Both take an instance to postprocess and the string key it is bound to. They also return an `Object`, which is expected to be the *postprocessed* instance. To

return the instance unaffected (that is, as per normal), you simply return the object that was passed in. For our purpose, this is exactly what we want. But first we register the instance in a collection of Screens to be used later by the lifecycle system:

```
public Object postProcessAfterInitialization(
    Object object, String key) throws BeansException {

    if (object instanceof Screen)
        screens.add((Screen)object);

    return object;
}
```

Notice that we're only interested in instances of the Screen class. Subsequently, lifecycle events can be fired on a timer trigger to all screens as necessary:

```
public class LifecyclePostProcessor implements BeanPostProcessor {
    private final List<Screen> screens = new ArrayList<Screen>();

    public void suspendAllScreens() {           ◀— Called at midnight, by timer
        for (Screen screen : screens)
            screen.suspend();                 ◀— Suspend each screen, in turn
    }

    public Object postProcessAfterInitialization(
        Object object, String key) throws BeansException { .. }

    public Object postProcessBeforeInitialization(
        Object object, String key) throws BeansException { .. }
}
```

Consider its complement, which is called by another timer, when the arcade reopens next morning:

```
public void resumeAllScreens() {
    for (Screen screen : screens)
        screen.resume();
}
```

This process can go on continuously every day, so long as the timer thread is in good health, and the screens will correctly switch feeds at the appropriate times. We didn't have to write a timer for each screen, nor did Screens have to keep track of the time themselves. Our single lifecycle manager was sufficient to push events out at the correct time to all Screens. This means not only less code to write but also far less code to *test* and fewer possible points of failure.

Custom post-processing is thus a powerful and flexible technique for creating lifecycle models suited to your application and its particular requirements. Another variant to building custom lifecycle is by multicasting a single method call to many recipients. This system has the advantage of being able to pass arbitrary arguments and even process returned values.

7.6 Customizing lifecycle with multicasting

Multicasting is very similar to what you just saw in section 7.5. Specifically, *multicasting* is the process of dispatching a single method call to multiple recipients. As such, it is

perfectly suited to sending lifecycle event notifications to objects managed by an injector. The primary difference between multicasting and the method we used with post-processors is that the framework takes care of broadcasting the event across instances in the injector. This is illustrated in figure 7.14.

This means that you can design your classes to be slightly more decoupled, with less effort. And you can model them according to the traits they embody in a lifecycle model. Listing 7.8 shows how the `Screen` would look if it were designed with multicasting in mind.

Listing 7.8 A screen with a timed lifecycle, designed with lifecycle traits

```
package arcade;

public class Screen implements Suspendable, Resumable {
    private final Feed daytimeFeed;
    private final Feed overnightFeed;

    public Screen(Feed daytime, Feed overnight) { .. }

    public void suspend() {
        show(overnightFeed);
    }

    public void resume() {
        show(daytimeFeed);
    }

    ...
}

public interface Suspendable {
    void suspend();
}

public interface Resumable {
    void resume();
}
```

Here `Suspendable` and `Resumable` are two *traits* that a `Screen` possesses. In other words, they are roles that the `Screen` can embody. This is very similar to the role interfaces that you saw with interface injection in chapter 3. Multicasting is supported in PicoContainer's Gems extension.¹ Listing 7.9 describes a lifecycle manager that uses multicasting to notify `Screens`.

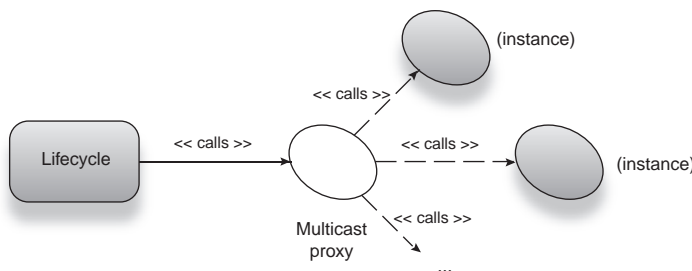


Figure 7.14
A multicast proxy
promulgates a single
lifecycle call across all
managed instances.

¹ Gems are simple, intuitive extensions to the PicoContainer core framework. Find out more at: <http://www.picocontainer.org>.

Listing 7.9 Lifecycle manager that suspends and resumes services via multicasting

```
package arcade;

import com.thoughtworks.proxy.factory.StandardProxyFactory;
import org.picocontainer.gems.util.Multicaster;
import org.picocontainer.PicoContainer;

public class LifecycleManager {
    private final PicoContainer injector = ..;

    public void suspendAll() {
        Suspendable target = (Suspendable)
            Multicaster.object(injector, true,
                new StandardProxyFactory());

        target.suspend();
    }
}
```

First, we create a *multicasting proxy*. This is a dynamically generated implementation of `Suspendable` that transparently delegates its method calls across every instance of `Suspendable` available to the injector:

```
Suspendable target = (Suspendable)
    Multicaster.object(injector, true, new
        StandardProxyFactory());
```

The first argument is the injector to inspect for `Suspendables`. The second tells the multicaster to proceed in the order in which `Suspendables` were originally created by the injector. The last argument is a factory utility, which is used to dynamically create the proxy implementation of `Suspendable`:

```
Suspendable target = (Suspendable)
    Multicaster.object(injector, true, new
        StandardProxyFactory());
```

`StandardProxyFactory` simply uses the JDK tool `java.lang.reflect.Proxy` to create dynamic classes of type `Suspendable`.

This is cool because now the same lifecycle manager can suspend and resume *any* services that expose `Suspendable` and `Resumable`, not just `Screens`. And if the implementation logic of a screen changes (say you move the feed control out into a `Hub`), you don't have to worry about breaking the lifecycle model. Multicasting is thus an evolved form of customizing lifecycle and probably the best choice in most cases.

7.7 Summary

While lifecycle isn't an immediate part of dependency injection, the two are nonetheless closely related. Every object goes through a series of states from construction to destruction that are demarcated by lifecycle *events*. In certain applications objects can be notified of these events by a framework or assisting library.

A class's constructor is the most straightforward and ready form of lifecycle. It is called after memory has been allocated for the object and used to perform some initialization logic, to put the object into a usable state. Dependency injectors often use

constructors to provide objects with their dependencies. This is detailed further in chapter 3.

In Java, when an object is about to be reclaimed by the garbage collector, its *finalizer* is called. A finalizer is called at an arbitrary, uncontrollable time, in a separate thread. Finalizers may not run until an application shuts down, or not even then! As a result, finalizers are unreliable for performing cleanup of finite system resources such as file handles or network sockets. Finalizers may be useful in rare cases for sanity checks or for reclaiming memory used in native libraries.

Lifecycle is not a universal concept. Different applications have different requirements for the transitions in state of their services. Lifecycle is thus specific to particular *problem domains*. For example, servlets undergo a separate initialization step well after deployment, to put them into a service-ready state. When a web application is stopped, an explicit destroy event is sent to a servlet. This is very different from a database connection's lifecycle, which may include notifications to set it "idle" or "in use," so validations or disconnects can be performed safely.

Some DI libraries provide a *destroy-method* hook, which is called by the injector on application shutdown. This is unsuitable for most cases for the same reasons that finalizers are. And more specific lifecycle models should be sought, rather than attempt to use this "destructor" anti-pattern. Java's `Closeable` interface is a suitable alternative for releasing finite resources that are data sources or producers. This also indicates that a service is designed with *closeability* in mind, as opposed to being an afterthought as with destructors.

Stateful EJBs are a programming model that supports a very complex lifecycle model, where EJBs are notified by an application server (or EJB container) periodically. Stateful EJBs are akin to HTTP sessions or "conversations" (see chapter 5), where clients may resume a previous interaction with the EJB. These points of resumption and suspension are modeled as lifecycle events to the EJB: `postactivate` and `prepassivate`.

An object's lifecycle is also related to timing—singleton-scoped objects can be created immediately, upon injector bootstrap, or lazily, when first needed. Lazy instantiation is useful when startup time is important, such as in a desktop application or when debugging. Eager instantiation is the opposite form, where all singletons are created when the injector starts up. This is useful in production servers where a performance hit can be taken up front, if it means that services can be obtained faster during an application's active life.

Customizing lifecycle is important, and it is useful in many applications. Post-processing is an idiom where an object is passed to a post-processor, prior to being made available for injection. These instances can be inspected and held in a registry for later reference. When a notification needs to be sent, you simply iterate the registry and notify each instance in turn. Spring offers postprocessing via an interface `BeanPostProcessor`.

Lifecycle can also be customized using multicasting. This is very similar to the post-processing technique, except that the framework is responsible for promulgating

events to instances managed by the injector. A single method call on an interface is transparently multicast to eligible instances. This is useful in designing more decoupled services that are easier to test and more amenable to refactoring.

Custom lifecycle is a powerful and flexible idiom for reducing boilerplate code and making your code tighter. It allows you to design classes that are more focused and easier to test and maintain. Almost all problem domains have some kind of use for framework lifecycle management. In the next chapter, we'll look at modifying object behavior by intercepting method calls. This can be thought of as the converse of lifecycle, since it involves changing how objects react to their callers rather than propagating events down to them as this chapter showed. As you will see, method interception can be a powerful technique for achieving focused goals across a broad spectrum of services.

Managing an object's behavior

This chapter covers:

- Intercepting methods with AOP
- Using transactions, security, and the like
- Watching for perils of proxying
- Avoiding corner cases

“Perfect behavior is born of complete indifference.”

—Cesare Pavese

Often one finds that certain types of logic are repeated throughout a program. Many separate parts of an application share similar concerns such as logging, security, or transactions. These are known as *crosscutting concerns*.

Rather than address these concerns in each individual class (as one would do normally), it's easier to address them all at once from a central location. Not only does this reduce repetitive boilerplate code, but it also makes maintenance of existing code less complicated.

Let's look at an example. Brokerage is a class that places an order for stock inside a database transaction. Traditionally, we would write code for starting and ending a transaction around each business method:

```
public class Brokerage {
    private final TransactionManager txn;

    public void placeOrder(Order order) {
        txn.beginTransaction();
        try {
            ...
        } catch(DataException e) {
            txn.rollback();
        } finally {
            if(!txn.rolledBack())
                txn.commit();
        }
    }
}
```

This transaction logic uses a `TransactionManager` to start and end a transaction, rolling back on an exception or committing as appropriate. This would be repeated in every method that needed to be transactional.

Now let's look at this if transactions were described declaratively:

```
public class Brokerage {

    @Transactional
    public void placeOrder(Order order) {
        ...
    }
}
```

This is much simpler. By declaring method `placeOrder()` as `@Transactional`, we're able to strip out most of the boilerplate code to wrap an order inside a transaction. Moreover, it removes `Brokerage`'s dependency on a `TransactionManager`, making it simpler to code and test. Also, now all transaction-specific logic is centralized in one place.

`@Transactional` is a piece of metadata that allowed us to declare a transaction around `placeOrder()`. What really happened underneath was that `placeOrder()` was intercepted and wrapped inside a transaction before proceeding normally. This is done via a technique known as aspect-oriented programming (AOP) and is the focus of this chapter. In this chapter we'll look at how to apply this technique to intercept methods on objects that are created by dependency injection and how to insert new behavior around each method. We'll start by examining how method interception is achieved.

8.1 Intercepting methods and AOP

Methods can be intercepted in a number of different ways, depending on the AOP library:

- At compile time, via a specialized compiler
- At load time, by altering class definitions directly
- At runtime, via dynamic proxying

This process is known as *weaving*, as in weaving in the new behavior. The introduced behavior is called *advice*.

Since we are concerned with dependency injection, we'll focus on the runtime flavor of weaving, which is done by the use of dynamic *proxies*.¹ Since DI libraries are responsible for creating objects, they are also able to intercede with proxies for intercepting behavior. This is transparent to any client, since the proxies are merely subclasses of the original types.

Dynamic proxies

A proxy is generated at runtime by a bytecode-generation library. The dynamic class-loading system in Java allows new classes to be defined even at runtime. The JDK core library also provides a utility for generating proxies. Several bytecode-manipulation libraries also exist that are able to do more sophisticated forms of code generation and proxying.

Proxying is a powerful technique. We can use it to replace almost any object created by an injector. Since a proxy possesses all the methods of the original type, it can alter the behavior of any of them.

Let's look at one such scenario where we intercept methods to add logging functionality using Guice.

8.1.1 A tracing interceptor with Guice

What we want to do here is trace the execution of every method on a certain class, by printing something to the console. We want to do this using interception rather than adding print statements to each method. Guice lets us do this by binding an interceptor (a utility class containing the advice):

```
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class TracingInterceptor implements MethodInterceptor {
    public Object invoke(MethodInvocation mi) throws Throwable {
        System.out.println("enter " + mi.getMethod().getName());
        try {
            return mi.proceed();
        } finally {
            System.out.println("exit " + mi.getMethod().getName());
        }
    }
}
```

This class is pretty simple; it contains one method—`invoke()`—which is called every time an interception occurs. This method then does the following:

¹ We encountered proxies in chapter 3 when dealing with the circular reference problem.

- Prints an alert before proceeding to the intercepted method
- Proceeds and returns the result from the intercepted method
- Prints another alert before returning control to the caller

The last bit is done inside a `finally` block so that any exception thrown by the intercepted method does not subvert the exit trace.

Applying this interceptor is done via Guice's binding API inside any `Module` class, as shown in listing 8.1:

Listing 8.1 A module that applies `TracingInterceptor` to all methods

```
import static com.google.inject.matcher.Matchers.*;

public class MyModule extends AbstractModule {

    @Override
    protected void configure() {
        ...
        bindInterceptor(any(), any(), new TracingInterceptor());
    }
}
```

TIP The `import static` statement at the top of this class makes it possible to omit the class name, `Matchers`, where the `any()` method is defined for convenience. It's a more readable shorthand for writing `Matchers.any()`.

The interesting part about this listing is how the interceptor is bound:

```
bindInterceptor(any(), any(), new TracingInterceptor());
```

The first two parameters are passed `any()`, which represents *any class* and *any method*, respectively. Guice uses these matchers to test for classes and methods to intercept. It comes with several such matchers out of the box, or you can write your own.

Now when we call any methods from objects wired by our injector, the methods will be traced on the console. Let's say `Chef` is one such class:

```
public class Chef {
    public void cook() {
        ...
    }

    public void clean() {
        ...
    }
}
```

Now calling `Chef`'s methods

```
Chef chef = Guice.createInjector(new MyModule())
    .getInstance(Chef.class);

chef.cook();
chef.clean();
```

produces the following output:

```
enter cook
exit cook
enter clean
exit clean
```

Chef still has no knowledge of how to print to console. Furthermore, inside a unit test, methods are *not* intercepted, and your test code can focus purely on asserting the relevant business logic.

AopAlliance and Guice

You'll notice that we used an AopAlliance API in this example. The AopAlliance is a public standard defined by many of the early AOP advocates and open source contributors. It's a simple standard that focuses on strong use cases rather than giving you the kitchen sink.

As such, it's not as full featured as some of the other AOP libraries, but on the other hand it's more lightweight and streamlined.

In the next section, we'll look at how to write the same tracing interceptor using a different framework. This will help you see the differences between the two major popular techniques.

8.1.2 A tracing interceptor with Spring

Spring uses a different AOP library (although it also supports the AopAlliance) but works under similar principles. Spring's library is *AspectJ*, which can itself be used independently to provide a whole host of AOP features beyond method interception. For our purposes we'll focus on its Spring incarnation. We'll use the same class *Chef*, but this time we'll intercept it with Spring and AspectJ, as shown in listing 8.2.

Listing 8.2 A tracing interceptor created as an aspect with Spring and AspectJ

```
import org.aspectj.lang.ProceedingJoinPoint;
public class TracingInterceptor {
    public Object trace(ProceedingJoinPoint call) throws Throwable {
        System.out.println("enter " + call.toShortString());
        try {
            return call.proceed();
        } finally {
            System.out.println("exit " + call.toShortString());
        }
    }
}
```

It looks almost the same as our *TracingInterceptor* from Guice. The primary difference is that we do not implement any interface; rather we will tell the injector directly about method *trace()*. This is achieved as shown in listing 8.3.

Listing 8.3 A tracing interceptor configuration with myAspect.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <bean id="chef" class="example.Chef"/>

  <bean id="tracer" class="example.TracingInterceptor"/>

  <aop:config>
    <aop:aspect ref="tracer">
      <aop:pointcut id="pointcuts.anyMethod"
        expression="execution(* example.*(..))" />
      <aop:around pointcut-ref="pointcuts.anyMethod" method="trace"/>
    </aop:aspect>
  </aop:config>
</beans>
```

This looks like a complex bit of mumbo-jumbo, but it's quite simple. First we declare our `TracingInterceptor` in a `<bean>` tag, naming it "tracer":

```
<bean id="tracer" class="example.TracingInterceptor"/>
```

Then we declare a *pointcut* using the `<aop:pointcut>` tag provided by Spring:

```
<aop:pointcut id="pointcuts.anyMethod"
  expression="execution(* example.*(..))" />
```

The expression `"execution(* example.*(..))"` is written in the AspectJ *pointcut language* and tells Spring to intercept the execution of any method with any visibility, name, or arguments from the `example` package. This pointcut expression is the equivalent in AspectJ of a matcher in Guice. Recall the binding expression for matching any method from listing 8.1:

```
bindInterceptor(any(), any(), new TracingInterceptor());
```

AspectJ's pointcut language also allows you to declare matchers of your choosing. Since it is a dedicated language, it doesn't come with any matchers out of the box.

The other tag worth mentioning in listing 8.3 is `<aop:aspect ref="tracer">`. This is the declaration of an *aspect*, which is essentially a binding between a matcher (or pointcut) and an interceptor (advice). It is the semantic equivalent of method `bindInterceptor()` shown earlier. Now running the example

```
BeanFactory injector = new FileSystemXmlApplicationContext("myAspect.xml");
Chef chef = (Chef) injector.getBean("chef");

chef.cook();
chef.clean();
```

produces the expected trace:

```
enter execution(cook)
exit execution(cook)
enter execution(clean)
exit execution(clean)
```

Now that you've seen how to apply method interception at a high level, let's examine how its internals work, by looking at how to work with dynamic proxies and their semantics.

8.1.3 *How proxying works*

We said earlier that a dynamic proxy is a subclass that is generated at runtime. The fact that it is of the same type means we can transparently replace the original implementation with a proxy and decorate its behavior as we please. Proxying is a tricky subject. The best way to think of it is to imagine a handwritten subclass where all the methods invoke an interceptor instead of doing any real work. The interceptor then decides whether to proceed with the real invocation or effect some alternate behavior instead.

Listing 8.4 shows how such a proxy might look for `Chef`.

Listing 8.4 A static (handwritten) proxy for `Chef`

```
public class ChefProxy extends Chef {
    private final MethodInterceptor interceptor;
    private final Chef chef;

    public ChefProxy(MethodInterceptor interceptor, Chef chef) {
        this.interceptor = interceptor;
        this.chef = chef;
    }

    public void cook() {
        interceptor.invoke(new MethodInvocation() { ... });
    }

    public void clean() {
        interceptor.invoke(new MethodInvocation() { ... });
    }
}
```

Rather than delegate calls directly to the intercepted instance `chef`, this proxy calls the interceptor with a control object, `MethodInvocation`. This control object can be used by `MethodInterceptor` to decide when and how to pass through calls to the original `chef`. These libraries are also able to generate the bytecode for `ChefProxy` on the fly. Let's look one such proxying mechanism.

PROXYING INTERFACES WITH JAVA PROXIES

The Java core library provides tools for dynamically generating proxies. It implements the same design pattern we just saw with a proxy and interceptor pair and is provided as part of the reflection toolset in `java.lang.reflect`. It's limited to proxying interfaces, but this turns out to be sufficient for the majority of use cases. If we imagined that `Chef` was an interface rather than a class,

```
public interface Chef {
    public void cook();

    public void clean();
}
```

we could create a dynamic subclass of Chef the following way:

```
import java.lang.reflect.Proxy;
...
Chef proxy = (Chef) Proxy.newProxyInstance(Chef.class.getClassLoader(),
                                           new Class[] { Chef.class },
                                           invocationHandler);
```

The first argument to `Proxy.newProxyInstance()` is the *classloader* in which to define the new proxy:

```
Proxy.newProxyInstance(Chef.class.getClassLoader(), ...)
```

Remember that this is not just a new object we're creating but an entirely new *class*. This can generally be the same as the default (context) classloader or, as in our example, the classloader to which the original interface belongs. It is sometimes useful to customize this, but for most cases you won't need to. You'll see more on this later in the chapter.

The second argument is an array of `Class` objects representing all the interfaces you want this proxy to intercept. In our example, this is just the one interface, `Chef`:

```
Proxy.newProxyInstance(..., new Class[] { Chef.class }, ...);
```

And finally, the last argument is the invocation handler we want the proxy to use:

```
Proxy.newProxyInstance(..., invocationHandler);
```

It must be an object that implements interface `InvocationHandler` from the `java.lang.reflect` package. As you can see, it is very similar to the AopAlliance's `MethodInterceptor`:

```
public interface InvocationHandler {
    Object invoke(Object proxy, Method method, Object[] args);
}
```

Here's a reimagining of the method-tracing interceptor from earlier in this chapter, using JDK Proxy and `InvocationHandler` with `Chef`:

```
public class TracingInterceptor implements InvocationHandler {
    private final Chef chef;

    public TracingInterceptor(Chef originalChef) {
        this.chef = originalChef;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        System.out.println("enter " + method.getName());
        try {
            return method.invoke(chef, args);
        }
    }
}
```



```

        } finally {
            System.out.println("exit " + method.getName());
        }
    }
}

```

To run it, we use the proxy obtained from `Proxy.newProxyInstance()`:

```

import java.lang.reflect.Proxy;
...
Chef proxy = (Chef) Proxy.newProxyInstance(Chef.class.getClassLoader(),
                                           new Class[] { Chef.class },
                                           new TracingInterceptor(originalChef));

proxy.cook();
proxy.clean();

```

This produces the expected trace:

```

enter cook
exit cook
enter clean
exit clean

```

By constructing `TracingInterceptor` with a constructor argument, `originalChef`, we make it possible for the interceptor to proceed onto *legitimate* calls against the intercepted instance should it need to. In our `invoke()` interception handler, this is illustrated by the reflective method invocation:

```

Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    System.out.println("enter " + method.getName());
    try {
        return method.invoke(chef, args);
    } finally {
        System.out.println("exit " + method.getName());
    }
}

```

This is akin to the `AopAlliance`'s `MethodInvocation.proceed()` and `AspectJ`'s `ProceedingJoinPoint.proceed()`.

Although Java provides a fairly powerful API for proxying *interfaces*, it provides no mechanism for proxying *classes*, abstract or otherwise. This is where third-party *bytecode manipulation tools* come in handy.

PROXYING CLASSES WITH CGLIB

A couple of useful libraries provide tools for proxying. Guice and Spring both use a popular library called `CGLib`² to generate proxies under the hood. `CGLib` fills in nicely where Java's proxy mechanism falls short. It's able to generate the bytecode that would dynamically extend an existing class and override all of its methods, dispatching them to an interceptor instead.

² `CGLib` stands for *Code Generation Library*. Find out more about `CGLib` at <http://cglib.sourceforge.net>.

If we had a concrete class `FrenchChef`, we would generate a proxy for it as follows:

```
import net.sf.cglib.proxy.Enhancer;
...
Chef chef = (Chef) Enhancer.create(FrenchChef.class,
    new TracingInterceptor());
```

The corresponding `TracingInterceptor` follows the design pattern we're intimately familiar with by now:

```
public class TracingInterceptor implements MethodInterceptor {
    public Object intercept(Object proxy, Method method, Object[] args,
        MethodProxy methodProxy) throws Throwable {
        System.out.println("enter " + method.getName());
        try {
            return methodProxy.invokeSuper(proxy, args);
        } finally {
            System.out.println("exit " + method.getName());
        }
    }
}
```

The interesting part is that we no longer have to hold onto the original chef; CGLib allows us to invoke the corresponding method directly on its superclass:

```
return methodProxy.invokeSuper(proxy, args);
```

Class proxying can be very powerful if used correctly. It's best used inside dependency injectors (or similar tools) that need to enhance a class's behavior. The consequences of intercepting class methods can be tricky. We'll look at some later in this chapter. It's always generally advisable to use interface proxying where possible, unless you're convinced of the need and its implications.

Of course, there are other problems with interception. One can be too eager to apply this technique. In the next section we'll look at how this can lead to problems.

8.1.4 *Too much advice can be dangerous!*

As in real life, you can run into trouble when you listen to too much advice. One of the important things to consider is that for each interception, you will incur the overhead of advising methods running before and after the original method. This is generally not an issue if you apply a few interceptors to a few methods. But when the chain gets longer, and it crosscuts critical paths in the application, it can degrade performance.

More important, the order of interception can play spoiler to program semantics. Here is a simple case:

```
public class Template {
    private final String template = "Hello, :name!";

    public String process(String name) {
        return template.replaceAll(":name", name);
    }
}
```

Class `Template` converts a dynamically provided name into a string containing a greeting. The following code

```
new Template().process("Josh");
```

returns a greeting to Josh:

```
Hello, Josh!
```

Now, using method interception, we can enhance this greeting by decorating it in bold (let's use HTML `` tags for familiarity):

```
import org.aopalliance.intercept.MethodInterceptor;

public class BoldDecoratingInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation mi) throws Throwable {
        String processed = (String)mi.proceed();
        return "<b>" + processed + "</b>";
    }
}
```

We bind this in using a simple matcher:

```
import static com.google.inject.matcher.Matchers.*;
...

bindInterceptor(subclassesOf(Template.class), any(), new
    BoldDecoratingInterceptor());
```

Now when a client processes anything from a template, it will be decorated in bold:

```
Guice.createInjector(...)
    .getInstance(Template.class)
    .process("Bob");
```

This prints the following:

```
<b>Hello, Bob!</b>
```

So far, so good. Now let's say we want to wrap this whole thing in HTML so it can be rendered in a website. If we had several such templates, another interceptor would save a lot of time and boilerplate code:

```
import org.aopalliance.intercept.MethodInterceptor;

public class HtmlDecoratingInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation mi) throws Throwable {
        String processed = (String) mi.proceed();
        return "<html><body>" + processed + "</body></html>";
    }
}
```

Now we have two bits of advice. If you were to naïvely bind in this interceptor at any arbitrary point, it could lead to very unexpected behavior:

```
import static com.google.inject.matcher.Matchers.*;
...
```

```
bindInterceptor(subclassesOf(Template.class), any(), new
    BoldDecoratingInterceptor());
bindInterceptor(subclassesOf(Template.class), any(),
    new HtmlDecoratingInterceptor());
```

This would print the following:

```
<b><html><body>Hello, Bob!</body></html></b>
```

Obviously, the order of interception matters. In this trivial example, it's easy for us to see where the problem is and correct it:

```
import static com.google.inject.matcher.Matchers.*;
...
    bindInterceptor(subclassesOf(Template.class), any(), new
        HtmlDecoratingInterceptor());
    bindInterceptor(subclassesOf(Template.class), any(), new
        BoldDecoratingInterceptor());
```

Now processing the interceptor chain renders our desired HTML correctly:

```
<html><body><b>Hello, Bob!</b></body></html>
```

This highlights a serious problem that would have gone undetected if we had assumed unit tests were sufficient verification of application behavior. In our example, all interceptors were in the same spot, and it was easy to identify and fix the problem. In more complex codebases, it may not be so straightforward. You should exercise caution when using AOP. In our example, the bold decorator could probably have been written without an interceptor. It probably would have worked better if we had folded it into the template itself.

Later in this chapter we'll show how to protect against the "too much advice" problem with integration tests. For now, let it suffice to say that you should apply interception only in valid use cases. In the next section we'll look at these use cases as applicable to enterprise applications.

8.2 Enterprise use cases for interception

The tracing example was fun, but it's quite a trivial use case. The real advantage of interception comes to light in enterprise use cases, particularly with transactions and security. These may be database transactions or logical groupings of any kind of task. And likewise with security—it may be about authorization to perform a particular action or simply about barring users who aren't logged in.

By far, the most prolific and useful case of interception is to wrap database transactions and reduce the overall level of boilerplate code in business logic methods. There are several ways to go about this; both Guice and Spring provide modules that allow you to use declarative transactions. It's even possible to roll your own, but there are a few edge cases that should lead you to use the library-provided ones, which are thoroughly tested. So let's look at how to achieve transactions with the warp-persist module for Guice.

8.2.1 *Transactional methods with warp-persist*

Like guice-servlet, which we encountered in chapter 5, warp-persist is a thin module library for Guice that provides support for persistence and transactions. Warp-persist provides integration with the following popular persistence engines:

- Hibernate (a popular object/relational mapping tool)
- Java Persistence API or JPA (a Java standard for object/relational mapping)
- Db4objects (an object database)

Hibernate and JPA both provide a mapping layer between Java objects and relational database tables (stored in an RDBMS like PostgreSQL). And Db4objects is a lightweight object database, which can store and retrieve native objects directly.

Warp-persist sits between the Guice injector and these frameworks, and it reduces the burden on you to wrap and integrate them. It also provides an abstraction over their particular transaction architectures and lets you use matchers to model transactional methods declaratively. Let's take the simple example of storing a new car in a database inventory:

```
import javax.persistence.EntityManager;
import com.wideplay.warp.persist.Transactional;

public class CarInventory {
    private final EntityManager em;
    public CarInventory(EntityManager em) {
        this.em = em;
    }

    @Transactional
    public void newCar(Car car) {
        em.persist(car);
    }
}
```

When a new car arrives, its details are entered into a `Car` object, and it's passed to method `newCar()`, which stores it using the `EntityManager`. The `EntityManager` is an interface provided by JPA that represents a session to the database. Entities (data) may be stored, retrieved, or removed via the `EntityManager` from within transactional methods.

The other important part of `CarInventory` is method `newCar()`, which is annotated `@Transactional`. This is an annotation provided by warp-persist that is used to demarcate a method as being transactional. If we didn't have the declarative approach with `@Transactional`, we would have to write the transaction behavior by hand:

```
public void newCar(Car car) {
    EntityTransaction txn = em.getTransaction();
    txn.begin();
    boolean succeed = true;
    try {
        em.persist(car);
    } catch (RuntimeException e) {
        txn.rollback();
    }
}
```

```

        succeed = false;
    } finally {
        if (succeed)
            txn.commit();
    }
}

```

All of this code is required to correctly determine whether to roll back a transaction and close it properly when finished. This is only a trivial case, since we aren't considering the beginning and closing of the `EntityManager` itself. A session to the database may also need to be opened and closed around transactions.

In listing 8.5, the caught exception is being swallowed after a transaction rollback without any significant action being taken. So, apart from saving yourself a world of repetitive boilerplate, declarative transactions also give you semantic control over the transactional state.

Let's look at how to tell the Guice injector that we're going to use warp-persist (see listing 8.5).

Listing 8.5 A module that configures warp-persist and `@Transactional`

```

import com.wideplay.warp.persist.PersistenceService;
import com.wideplay.warp.persist.jpa.JpaUnit;

public class CarModule extends AbstractModule {

    @Override
    protected void configure() {
        ...
        install(PersistenceService.usingJpa()
                .buildModule());

        bindConstant().annotatedWith(JpaUnit.class).to("carDB");
    }
}

```

The method `install()` is a way of telling Guice to add another module to the current one. It is exactly equivalent to passing in each module to the `createInjector()` method individually:

```

Guice.createInjector(new CarModule(), PersistenceService.usingJpa()
                .buildModule());

```

The other important piece of configuration is the constant bound to "carDB":

```

bindConstant().annotatedWith(JpaUnit.class).to("carDB");

```

This constant is used by warp-persist to determine which persistence unit to connect against. Persistence units are specified in the accompanying `persistence.xml` configuration file for JPA, which is typically placed in the `META-INF/` directory. Here's what it might look like in our imaginary car inventory:

```

<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```

```

xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">

<!-- A JPA Persistence Unit -->
<persistence-unit name="cardB" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <!-- JPA entities must be registered here -->
    <class>example.Car</class>

    <properties>
        <!-- vendor-specific properties go here -->
    </properties>
</persistence-unit>

</persistence>

```

Note that along with the name of the persistence unit, all the persistent classes are listed here. These are classes mapped with JPA annotations, which tell the persistence engine how to map an object to a row in the database. `example.Car` might look something like this:

```

@Entity
public class Car {
    @Id @GeneratedValue
    private Integer id;
    private String name;
    private String model;

    //get + set methods
}

```

When you run this application and call the `newCar()` method with a populated `Car` object,

```

@Inject CarInventory inventory;
...

Car car = new Car("BMW", "325Ci");
inventory.newCar(car);

```

a transaction is automatically started and committed around the call to `newCar()`. This ensures that the `EntityManager` enters the provided `Car` object into its corresponding database table.

Another interesting enterprise use case is that of security. Since interception can be applied from one spot, many parts of an application can be secured with a single, central step. In the following section, we'll explore how to secure groups of methods using Spring AOP and the Spring Security Framework.

8.2.2 **Securing methods with Spring Security**

Authorization is another common use case for intercepted methods. The pattern is sometimes even applied to web applications where HTTP requests are intercepted by servlet `Filters` and processed against security restrictions.

JPA and object/relational mapping (ORM)

The Java Persistence API is a specification for mapping simple Java objects to relational database tables. It provides a set of standard annotations (`@Entity`, `@Id`, and so on) that are used to map classes and fields to tables and columns. JPA has many vendor implementations, including Hibernate, Oracle TopLink, and Apache OpenJpa. All of these are available under open source licenses, free of charge.

JPA also includes tools for managing database transactions around local resources using a programmatic API. Both Spring and Guice (via warp-persist) provide integration modules for JPA programming models and for declarative transactions; among other niceties.

Find out more about the JPA standard at <http://java.sun.com/developer/technical-Articles/J2EE/jpa>.

And check out the warp-persist JPA tutorial at <http://www.wideplay.com/guiceweb-extensions2>.

Certain methods that provide privileged business functionality can come under the crosscutting banner of security. Like transactions, these methods must first verify that the user driving them has sufficient privileges to execute their functionality. This is a repetitive task that can be moved to the domain of interceptors, which have the rather elegant advantage of being able to suppress secured methods completely. They also allow for declarative management of security across a set of business objects, which is a useful model when applied in very large applications with similar, recurring security concerns.

Let's look at how this is done using Spring Security, an extension to Spring that provides many security features. First, Spring Security is meant primarily for web applications, so enabling it requires a HTTP filter to be present. This gives us a hook into Spring's security stack. Draw up a web.xml like the following:

```
<filter>
  <filter-name>springSecurity</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</
    filter-class>
</filter>

<filter-mapping>
  <filter-name>springSecurity</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Much like guice-servlet's GuiceFilter (from chapter 5), this filter mapping tells the servlet container to route all requests through Spring Security's DelegatingFilterProxy. This Spring-provided filter is able to call into Spring's injector and determine the correct set of security constraints to apply.

Here's a corresponding XML configuration for securing objects of an imaginary class PhoneService, managed by the Spring injector:


```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:security="http://www.springframework.org/schema/security"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/security
                           http://www.springframework.org/schema/security/spring-security-
➡ 2.0.xsd">

  <security:http auto-config="true">
    <security:intercept-url pattern="/**" access="ROLE_USER" />
  </security:http>

  <security:authentication-provider>
    <security:user-service>
      <security:user name="josh" password="supersecret"
                    authorities="ROLE_USER, ROLE_ADMIN" />
      <security:user name="bob" password="bobby"
                    authorities="ROLE_USER" />
    </security:user-service>
  </security:authentication-provider>

  <security:global-method-security>
    <security:protect-pointcut
      expression="execution(* example.PhoneService.*(..)"
      access="ROLE_ADMIN" />
    </security:global-method-security>
</beans>
```

Wow, that's quite a lot of configuration! Let's look at what it all means. First, we set up automatic configuration to use a whole bunch of defaults for customizable services that Spring Security provides:

```
<security:http auto-config="true">
  <security:intercept-url pattern="/**" access="ROLE_USER" />
</security:http>
```

These are good for our purpose, where we only want to demonstrate security around methods via interception as it applies to dependency injection.

Here we've also asked Spring to intercept all requests coming in and allow access only to users with the privilege `ROLE_USER`:

```
<security:http auto-config="true">
  <security:intercept-url pattern="/**" access="ROLE_USER" />
</security:http>
```

The path matcher `pattern="/**"` used by Spring Security³ is slightly different from the conventional servlet pattern and is equivalent to a URL mapping of `"/**"` in servlet parlance.

Next, we set up two users, josh and bob, and gave them both `ROLE_USER` access:

³ If you're familiar with Ant, note that Spring Security uses Ant-style paths to match against URIs. Ant is a popular build-scripting tool for Java from Apache: <http://ant.apache.org/>.

```

<security:authentication-provider>
  <security:user-service>
    <security:user name="josh" password="supersecret"
      authorities="ROLE_USER, ROLE_ADMIN" />
    <security:user name="bob" password="bobby"
      authorities="ROLE_USER" />
  </security:user-service>
</security:authentication-provider>

```

This means they will be able to access any page in the web application by default.

Finally, we applied a method security constraint via a *pointcut*. Recall pointcuts from the section early in this chapter on Spring AOP with AspectJ. This pointcut expression matches any method in an imaginary class PhoneService:

```

<security:global-method-security>
  <security:protect-pointcut
    expression="execution(* example.PhoneService.*(..))"
    access="ROLE_ADMIN" />
</security:global-method-security>

```

This security pointcut prevents access to methods in PhoneService from anyone but ROLE_ADMIN users. In our case only josh had this role:

```

<security:user name="josh" password="supersecret"
  authorities="ROLE_USER, ROLE_ADMIN" />
<security:user name="bob" password="bobby" authorities="ROLE_USER" />

```

Let's imagine PhoneService to look something like this:

```

public class PhoneService {
  public Response call(String number) {
    ...
  }

  public void hangUp() {
    ...
  }
}

```

Now if any client code tries to access methods from this class via the Spring injector

```

<bean class="example.PhoneCaller">
  <constructor-arg><bean class="example.PhoneService"/></constructor-arg>
</bean>

```

and the corresponding class,

```

public class PhoneCaller {
  private final PhoneService phone;

  public PhoneCaller(PhoneService phone) {
    this.phone = phone;
  }

  public void doAction() {
    Response response = phone.call("555-1212");
    if (response.notBusy())

```

```

        phone.hangUp();
    }
}

```

when a user triggers this action (by, for example, clicking a button to call a recipient), Spring Security intercepts calls to `PhoneService.call()` and `PhoneService.hangUp()` and ensures that the user has the appropriate role. If not, an exception is raised and the user is shown an authorization failure message. In this scenario, the target methods are never executed.

This example requires a lot of delving into ancillary topics, such as Spring's web integration layer and related concerns. We've taken only the most cursory glance at this security system, focusing instead on the method interception use case. For a more thorough examination, consult the source code accompanying this chapter and visit <http://www.springsource.org> for the Spring Security documentation.

In the rest of this chapter, we'll change gears and look at the flip side of all this interception and how basic assumptions in design may need to change depending on how you use AOP.

8.3 *Pitfalls and assumptions about interception and proxying*

As you saw earlier, proxying is an advanced topic and can involve some pretty low-level semantics dealing with the internals of Java. Thus there are several pitfalls and mistaken assumptions that can lead you astray when working with proxied classes or interfaces. Ideally, a proxy should behave identically to its replaced counterpart. However, this is not always the case.

You can understand and avoid many of the pitfalls if you keep in mind that a proxy is simply a subclass of the original type. Let's look at some of these cases.

8.3.1 *Sameness tests are unreliable*

If you have a reference to an object, never rely on the sameness (`==`) test to assert that it's the object you're expecting. An object created by the injector may have been proxied, and though it appears to be the same as the original object, it isn't. This is true even in the case of singletons. Here's such a scenario:

```

final Painting picasso = new Painting("Picasso");

Injector injector = Guice.createInjector(new AbstractModule() {
    protected void configure() {
        bind(Painting.class).toInstance(picasso);
        bindInterceptor(any(), any(), new TracingInterceptor());
    }
});

assert picasso == injector.getInstance(Painting.class); //will fail

```

In this case, we create an instance outside the injector and bind it via the `bind().toInstance()` directive:

```

final Painting picasso = new Painting("Picasso");

Injector injector = Guice.createInjector(

```

```
...
bind(Painting.class).toInstance(picasso);
...
```

This is an implicit singleton-scoped binding (since we've hand-created the only instance).

Next, we bound in a `TracingInterceptor` like the ones we've seen throughout this chapter. This runs on all classes and all methods:

```
bindInterceptor(any(), any(), new TracingInterceptor());
```

Now, we obtain the bound instance of `Painting` (`picasso`) from the injector and compare it using a sameness assertion to the instance we already know about:

```
assert picasso == injector.getInstance(Painting.class); //will fail
```

This assertion fails because even though we're semantically talking about the same instance, the physical instances are different. One has been proxied by the injector and the other is a direct reference to the original instance.

A related but much more common anti-pattern is to assume that the class of an available object is the same as the one it is bound to. For example, many logging frameworks publish log messages under the class of an object. Many people naively make the following mistake:

```
import java.util.logging.*;

public class ANoisyService {
    public final Logger log = Logger.getLogger(getClass().getName());
    ...
}
```

This is a dangerous assumption because while it appears as though method `getClass()` will return `Class<ANoisyService>`, in reality it may return the class of the dynamically generated proxy! So instead of logging under `ANoisyLogger`, your log statements may be printing under some unintelligible, generated class name.

Worse than this, if you use `getClass()` to make decisions on application logic, you can find some very erratic and unexpected behavior. Here's an example of a naïve `equals()` method implementation that mistakenly precludes a proxied object from an equality test:

```
public class EqualToNone {
    @Override
    public boolean equals(Object object) {
        if (object == null || object.getClass() != EqualsToNone.class)
            return false;
        ...
    }
}
```

This `equals()` method makes the assumption that subclass instances cannot be semantically equivalent to superclass instances. This is a wrong assumption when any

proxied objects are involved. It's not only dependency injectors that need to proxy objects for dynamic behavior modification. Many web frameworks and persistence libraries also employ this design pattern to provide dynamic behavior.

To fix this problem, you should always assume that an object can be proxied and program accordingly:

```
public class EqualToNone {
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof EqualToNone))
            return false;

        ...
    }
}
```

This is a much better implementation that's safe to proxying behavior. Similarly, you should always be explicit about your sameness semantic:

```
import java.util.logging.*;

public class ANoisyService {
    public final Logger log = Logger.getLogger(ANoisyService.class.getName());

    ...
}
```

Now, ANoisyService will always publish under the correct logging category regardless of whether or not it has been proxied. Another problem that arises from this situation is the inability to intercept static methods.

8.3.2 *Static methods cannot be intercepted*

Since proxies are merely subclasses and Java doesn't support overriding static methods, it follows that proxies cannot intercept static methods. However, something more subtle is at work. If you try the following example,

```
public class Super {
    public static void babble() {
        System.out.println("yakity yak");
    }
}

public class Sub extends Super {
    public static void babble() {
        System.out.println("eternal quiet");
    }

    public static void main(String...args) {
        babble();
    }
}
```

and run class Sub from the command line, what do you think it will print? Will the program even compile?

Yes, it will compile! And it prints

```
eternal quiet
```

which appears to all eyes like an overridden static method. We declared a static method `babble()` in `Super` and another static method `babble()` in `Sub`. When we ran `babble()` from `Sub`, it used the subclass's version. This looks very much like overriding instance methods in subclasses. Fortunately there's an explanation: What's really happening is that there are two static methods named `babble()` and the one in `Sub` is hiding the one in `Super` via its lexical context. In other words, if you were to run this from anywhere outside `Sub` or `Super`, you'd have to specify which method you were talking about:

```
import static Super.*;
...
babble();
```

This version correctly prints "yakity yak" as expected. So any code that *statically* depends on `Sub` or `Super` must always be explicit about which one it is talking about, meaning that there is no way to substitute a proxied subtype (even were it possible) without the client itself doing so. When you want dynamic behavior, don't place logic in static methods.

Private methods also face the same problem because they cannot be overridden. Although they have access to dynamic state, they cannot be proxied.

8.3.3 Neither can private methods

In Java, methods can have private visibility. A method that is private can't be called from anywhere outside its owning class. This applies to subclasses too, which means that dynamic proxies (which are just subclasses) cannot intercept private methods. If we go back to our `FrenchChef` and alter the visibility of method `clean()` as follows,

```
public class FrenchChef {
    public void cook() {
        ...
    }

    private void clean() {
        ...
    }
}
```

and then run it by adding a `main()` method to its body,

```
public static void main(String...args) {
    FrenchChef chef = Guice.createInjector(new AbstractModule() {

        @Override
        protected void configure() {
            bind(FrenchChef.class);
            bindInterceptor(any(), any(),
                new TracingInterceptor()); ← Trace all methods
        }

    }).getInstance(FrenchChef.class); ← Obtain a proxied FrenchChef
```

```

        chef.cook();
        chef.clean();
    }

```

it will produce the following output

```

enter cook
exit cook

```

with no mention of `clean()`. This was not because `clean` did not run; we called it from `main()`, after all. Rather it was because proxying doesn't permit private methods to be intercepted before they're dispatched. A good solution to this problem is to elevate the class's visibility slightly:

```

public class FrenchChef {
    public void cook() {
        ...
    }

    protected void clean() {
        ...
    }
}

```

Protected methods are good candidates for proxying, since they're hidden from all external users except subclasses of the original type. Running the program again produces a more satisfying result:

```

enter cook
exit cook
enter clean
exit clean

```

This is probably the best solution for the majority of cases although it may not always be ideal. While protected methods are hidden from classes outside the current hierarchy, they are still visible to nonproxy subclasses that may exist in different modules. If this is undesirable, Java has yet another visibility level: *package-local* visibility. This makes methods invisible to any code that's outside the owning class's package. Package-local⁴ visibility is perfect for hiding methods from subclasses that may extend your class but for which you don't want certain methods shown. These methods can still be intercepted by proxies, because proxies are typically generated within the same package as the original class. Denote package-local visibility by omitting an access keyword:

```

public class FrenchChef {
    public void cook() {
        ...
    }

    void clean() {
        ...
    }
}

```

⁴ Package-local visibility is sometimes also called package-private or default visibility.

NOTE I say that proxies are typically generated within the same package, but there are some cases where this may not be true. This behavior is dependent on specific decisions made by the library you're using. Keep an eye out for such decisions. Most dependency injectors can be relied on to place proxies within the same package as their parent classes.

Another variant of this problem (the inability to override certain methods) presents itself when a method has specifically declared itself to be final.

8.3.4 And certainly not final methods!

Methods can be declared as *final* to prevent subclasses from overriding them. The final keyword is not a visibility modifier, since it can be applied on public, protected, or package-local methods:

```
public class FrenchChef {
    public final void cook() {
        ...
    }
    void clean() {
        ...
    }
}
```

However, it does prevent a subclass from redefining the method by overriding it. The following code results in a compiler error:

```
public class FrenchSousChef extends FrenchChef {
    @Override
    public void cook() {    ← Cannot override final method
        ...
    }
}
```

Naturally, this means a proxy can't override it either, and therefore a final method can't be intercepted for behavior modification. The solution in such cases is obviously to remove the final modifier. But if this can't be done for some reason (perhaps you don't have access to the source code), you can wrap the method in a pass-through *delegator*:

```
public class FrenchChefDelegator implements Chef {
    private final FrenchChef delegate;

    public void cook() {
        delegate.cook();    ← Delegate to original class
    }

    ...
}
```

This class can now be proxied for interception. Method `cook()` is not final so it can be intercepted safely with a proxy. And the class `FrenchChefDelegator` implements `Chef`, so clients of `Chef` are not impacted by the change.

TIP An even better solution would be to proxy the Chef interface directly and pass calls through to the specific implementation. For example, Spring's AOP mechanism allows you to choose interface proxying as the default method. This is also easily accomplished by hand.

The same principle applies to final classes. These are classes that cannot be extended. Not only can their methods not be intercepted, but you can't generate subclasses of them at all:

```
public final class FrenchChef implements Chef {
    public void cook() {
        ...
    }
    public void clean() {
        ...
    }
}
```

The Delegator pattern also works well for this situation:

```
class FrenchChefDelegator implements Chef {
    private final FrenchChef delegate;
    ...
    public void cook() {
        delegate.cook();
    }
    public void clean() {
        delegate.clean();
    }
}
```



**Delegate to
original class**

When in doubt always choose this method over inheritance—even simpler would be proxying the interface directly. Just as final classes and methods cannot be intercepted at runtime, neither can class member fields.

8.3.5 *Fields are off limits*

Certain flavors of AOP allow you to make deferred modifications to any parts of a codebase. The build-time weaving from AspectJ allows you to alter fields, interfaces implemented, and even static methods. However, neither Spring nor Guice provides this kind of weaving. Almost all use cases for interception can be fulfilled with the runtime weaving flavor that they do provide. However, it does mean that fields, any kind of fields, are off limits with regard to interception.

Let's say FrenchChefs have a dependency on a RecipeBook:

```
public class FrenchChef implements Chef {
    private final RecipeBook recipes;
    public void cook() {
        ...
    }
}
```

```

    public void clean() {
        ...
    }
}

```

An interceptor can only advise behavior on methods `cook()` and `clean()` but cannot advise anything on `recipes`. Even if methods on `recipes` were called from `cook()` or `clean()`, an interceptor declared on `FrenchChef` would not be able to trap specific behavior:

```

    private final RecipeBook recipes;

    public void cook() {
        recipes.read("ratatouille");
        ...
    }

```

And a tracing interceptor declared with the specific matcher

```

bindInterceptor(subclassesOf(Chef.class), any(),
    new TracingInterceptor());

```

will ignore method calls on `RecipeBook` and trace only methods `cook()` and `clean()`. Of course, expanding the matching strategy will allow you to intercept `RecipeBook`'s methods. The following matcher is much better suited:

```

bindInterceptor(subclassesOf(Chef.class)
    ➡ .or(subclassesOf(RecipeBook.class)), any(), new TracingInterceptor());

```

The general one we have used earlier in the chapter

```

bindInterceptor(any(), any(), new TracingInterceptor());

```

will also work nicely:

```

enter cook
enter read
exit read
exit cook
enter clean
exit clean

```

The limitation here is that any intercepted dependency must also have been provided by dependency injection. No dependency injector can intercept methods on objects it doesn't create. For example, the following class

```

public class FrenchChef implements Chef {
    private final RecipeBook recipes = new RecipeBook();

    public void cook() {
        ...
    }

    public void clean() {
        ...
    }
}

```

creates its own dependency (see construction by hand from chapter 1) and therefore subverts the interception mechanism. No matcher will be able to intercept method calls on `RecipeBook` now:

```
enter cook
exit cook
enter clean
exit clean
```

Furthermore, any accesses to scalar fields like primitives or `Strings` are not open to interception whether or not they are injected by the dependency injector:

```
public class FrenchChef implements Chef {
    private int dishesCooked;
    private BigInteger potsWashed;
    private String currently;

    public void cook() {
        dishesCooked++;
        currently = "cooking";
    }

    public void clean() {
        potsWashed = potsWashed.add(BigInteger.ONE);
        currently = "cleaning";
    }
}
```

None of these accesses—whether via method calls or directly by assignment—are visible to interceptors. Typically this is not a problem, but it's something to keep in mind when you're designing services with scalar fields. Of course, not all dependencies will necessarily need interception.

8.3.6 *Unit tests and interception*

One of the nice things about DI is that it doesn't affect the way you write tests. If we were to test a class with many complex dependencies, all we'd need to do would be replace them with mock or stub equivalents. This principle can become somewhat muddled when interception comes into play.

While the imperative behavior of a class is the same in both test and application, its semantic behavior can change advice introduced by AOP. For example, a theme park's ticket-purchase system collects money and dispenses a ticket:

```
public class TicketBooth {
    public Ticket purchase(Money money) {
        if (money.equals(Money.inDollars("200")))
            return new Ticket();

        return Ticket.INSUFFICIENT_FUNDS;
    }
}
```

We can write a unit test for this class quite easily:

```

public class TicketBoothTest {
    @Test
    public void purchaseATicketSuccessfully() {
        Ticket ticket = new TicketBooth().purchase(Money.inDollars("200"));
        assert Ticket.INSUFFICIENT_FUNDS != ticket;
    }
}

```

← **Ensure this is a valid ticket**

If you decided to modify this behavior by printing a discount coupon for every 100th customer (via an interceptor), the original class's semantic has changed:

```

public class DiscountingInterceptor implements MethodInterceptor {
    private int count;

    public Object invoke(MethodInvocation mi) throws Throwable {
        Ticket ticket = (Ticket)mi.proceed();

        if (Ticket.INSUFFICIENT_FUNDS != ticket
            && (++count % 100) == 0) {
            return new DiscountedTicket(ticket);
        }

        return ticket;
    }
}

```

← **If the customer count is a multiple of 100**

← **Return a discounted ticket**

← **Return a normal ticket**

The immediate solution that comes to mind is simply to write another unit test, this time around the interceptor:

```

public class DiscountingInterceptorTest {
    @Test
    public void decorateEveryHundredthCall() throws Throwable {
        DiscountingInterceptor discounter = new DiscountingInterceptor();
        MockInvocation mi = new MockInvocation();

        for(int i = 1; i < 101; i++) {
            Object result = discounter.invoke(mi);

            if (i == 100)
                assert result instanceof DiscountedTicket;
        }
    }
}

```

← **Call interceptor with a mock invocation**

← **On the 100th run, we should see a discount**

But this is not where the difficulty lies. As you saw in “Too much advice can be dangerous!” the combination of two behavioral units can affect total semantic behavior and belie the assertions of individual unit tests. One effective solution is to write an integration test that brings the relevant units together and tests them as they might behave in the eventual application environment.

This is different from a full acceptance test or QA pass, since we’re interested in only a particular combination of behaviors and can restrict our attention to the classes we expect to be intercepted:

```

public class DiscountingInterceptorIntegrationTest {
    @Test

```

```

public void discountEveryHundredthTicket() {
    Injector injector = Guice.createInjector(new AbstractModule() {
        @Override
        protected void configure() {
            bind(TicketBooth.class);
            bindInterceptor(any(), any(),
                new DiscountingInterceptor());
        }
    });

    TicketBooth booth = injector.getInstance(TicketBooth.class);

    for(int i = 1; i < 101; i++) {
        Ticket ticket =
            booth.purchase(Money.inDollars("200"));

        if (i == 100)
            assert ticket instanceof DiscountedTicket;
    }
}

```

Apply discount
interceptor

Call original
object 100 times

This test looks very similar to the unit test on the interceptor, but it gives us more confidence about how these units come together. Notice that we call the `TicketBooth.purchase()` method just as clients would in the real application. And its behavior is decorated with interceptors that would also be present in the real app. Where this gets really useful is if we add a second or a third interceptor (as we did in “Too much advice can be dangerous!”); then the confidence that the integration test provides is far greater than the sum of individual unit tests.

When you have any sort of services in your program that rely on side effects like these, it’s important to write vertical integration tests to give yourself some assurance on their combined behavior. This will generally be in addition to any unit tests that exist on individual classes. Neither is enough alone, but together they give us sufficient breadth for verifying correct behavior.

8.4 Summary

In this chapter we encountered certain concerns that are global to a program. In traditional approaches, these are addressed individually with repetitive and boilerplate code. A technique known as AOP allows us to centralize this code in one location and apply it many times using a matcher (or pointcut). Matchers allow us to introduce additional behavior before and after business methods execute in a dynamic and declarative fashion. This introduced behavior is known as advice and is typically applied in the form of interceptors.

Interceptors trap the normal behavior of a method and decorate it with the provided advice. They are applied via the use of dynamic proxies, which are generated subclasses of the original class that reroute method calls through an interceptor. The Java core library provides a simple proxying mechanism for generating subclasses of interfaces on the fly. More sophisticated proxy libraries are available for proxying concrete classes too (CGLib is one example).

Because dependency injectors create the services used by an application, they are able to intercede with a proxy that intercepts behavior when applicable. These interceptors can be turned on or off with simple configuration options in both Spring and Guice.

Intercepting methods is a powerful technique for adding dynamic behavior that crosscuts a problem domain. Concerns like security, logging, and transactions are typical use cases for interception. However, it's fraught with pitfalls and requires a thorough understanding if it's to be applied properly, and it has unintended consequences. One pitfall is the order of interception: An interceptor that replaces the returned value of methods (or their arguments) can unintentionally disrupt the behavior of other interceptors yet to run. An integration test that puts together the expected set of interceptors can be a good confidence booster in such cases.

Warp-persist is an integration layer for Guice that allows you to apply transactions declaratively to Hibernate-, JPA-, and Db4objects-driven applications. It provides an `@Transactional` annotation that can be applied to introduce transactionality around any method in an application. Of course, it supports interception without annotations too.

Similarly Spring Security is an AOP-based security framework for Spring applications. Spring Security allows you to declare users and roles and attach them to methods via the use of a pointcut (a matching expression). When a user who is logged into the system executes these methods, an interceptor verifies that he has sufficient privileges, blowing up with an exception if not.

Other things to watch out for with proxies are sameness tests. The `==` operator tests whether references are equal; it can lead you astray if you test a reference against its intercepted equivalent, since the latter is a dynamically generated proxy. Semantically they refer to the same instance underneath. Equality testing should be applied in such cases. The same also applies to comparing classes of objects directly with the `getClass()` reflective method.

There are several parts of a class and its behaviors that simply cannot be intercepted with runtime proxies. For example, static methods are not attached to any instance, so they cannot be intercepted. Similarly, fields and methods on objects belonging to these fields are off-limits (unless they too have been injected). Methods having `private` visibility or a `final` modifier are also ineligible for proxying and thus behavior modification through interception.

With all these corner cases and pitfalls in mind, you can still apply this technique to great effect in your code. Reducing boilerplate and introducing behavior modification to services late in the game can be a powerful tool. Use interception wisely.

In the next chapter we'll look at some more general design issues specific to very large sets of applications that interact with one another, and we'll show how to use dependency injection in these environments.



Best practices in code design

This chapter covers:

- Understanding publication and wiring
- Designing objects to be thread-safe
- Optimizing for concurrency
- Dealing with type-safety
- Using objects in collections

“The effort to understand the universe is one of the very few things that lifts human life a little above the level of farce and gives it some of the grace of tragedy.”

—Steven Weinberg

It’s often a puzzle to programmers who are new to dependency injection about when to apply it. When it wasn’t widely understood, even experienced practitioners applied it too much or too sparingly. The immature state of libraries contributed to this fuzzy application too, giving one the tools but not the discretion of its applicability.

A more fundamental set of confusion arises from the best practices of the language itself. Are these distinct from the patterns relevant to DI? Are they superseded? Or can they coexist (figure 9.1)?

It turns out that such questions are based on the wrong premise entirely. The best practices portended by a language and its engineering fundamentals are the same as those proffered by dependency injection (figure 9.2).

In other words, design well for the language and problem domain, and you will have designed well for the principles of testing, succinctness, and flexibility.

One of the core problems in multi-threaded object-oriented languages is that of visibility. We showed in earlier chapters that altering the state of an object in one thread did not necessarily mean altering its state for all threads. This was particularly relevant with the problem of scope widening discussed in chapters 5 and 6. A more subtle incarnation of this problem occurs during (and immediately after) the construction of an object. Unless carefully designed, its fields may not be properly visible to all threads using it. This is a problem sometimes known as *unsafe publication*, and it's discussed in the following section on object visibility.

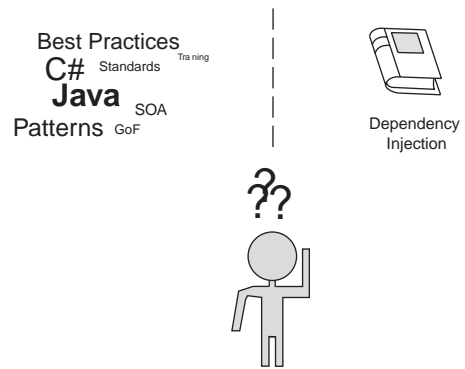


Figure 9.1 Best practices for the language, or architecture with DI?

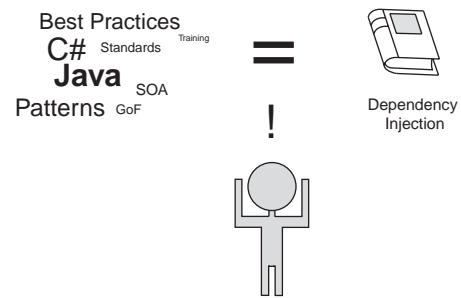


Figure 9.2 Best practices for the language and dependency injection are the same.

9.1 Objects and visibility

In chapter 3 we said that a constructor is a special method that runs once immediately after memory allocation of an object, in order to perform initialization work. As such, the object allocated is generally not visible to any other threads of execution than the one creating it. I say “generally,” as there are some special circumstances under which objects can be visible.

One of the best ways to understand visibility is to investigate a *hashtable*. A hashtable is a flexible data structure, sometimes called an associative array, that's used to store key-value pairs. Hashtables are ideal for us because they give us a scenario where one thread may come in and change values that are being read by other threads simultaneously, introducing problems of ordering and visibility. Consider a hashtable that stores email addresses by name (we'll use Java's `java.util.HashMap`):

```
Map<String, Email> emails = new HashMap<String, Email>();
emails.put("Dhanji", new Email("dhanji@gmail.com")); //yes, this is my
//real email =)
emails.put("Josh", new Email("josh@noemail.com"));
```



```
System.out.println("Dhanji's email address really is "
    + emails.get("Dhanji"));
```

Reading from this map is fairly straightforward, since there's only one thread of execution through this program (the main thread). We can confidently say that email address values are placed in the map before they're read. This is satisfied by simple lexical ordering. In other words, the `put()` method is called before the `get()`. The values are said to be safely published to the reader.

We can elaborate on this example by introducing a second thread:

```
public class UnsafePublication {
    private Map<String, Email> emails = new HashMap<String, Email>();

    public void putEmails() {
        emails.put("Dhanji", new Email("dhanji@gmail.com"));
        emails.put("Josh", new Email("josh@noemail.com"));
    }

    public void read() {
        System.out.println("Dhanji's email address really is "
            + emails.get("Dhanji"));
    }
}
```

If method `putEmails()` is called by Thread A and method `read()` is called by Thread B, there's a danger that the values `read()` is looking for may not yet be available. This problem cannot be solved by lexical ordering either (see listing 9.1).

Listing 9.1 Unsafe publication of hashtable values

```
public class Main {
    public static void main(String...args) {
        final UnsafePublication pub = new UnsafePublication();

        new Thread(new Runnable() {    ← Thread A
            public void run() {
                pub.putEmails();
            }
        }).start();

        new Thread(new Runnable() {    ← Thread B
            public void run() {
                pub.read();
            }
        }).start();
    }
}
```

In listing 9.1, two threads are started up in order. Thread A has the task of writing emails into the hashtable, while Thread B calls `read()`, which prints them. Now, even though it appears there is a lexical ordering, there's no guarantee that Thread A will run before Thread B. Several factors are at play. It may be that CPU scheduling pre-empts the start of Thread A. Or it may be that the compiler believes reordering

instructions will lead to better throughput. The point is, once you introduce multiple threads accessing a shared resource, lexical guarantees are no longer valid.

There's a further subtlety with regard to unsafe publication. Even if Thread A did execute on the CPU before Thread B, which we could presumably tell by adding the following print statement,

```
public class UnsafePublication {
    private Map<String, Email> emails = new HashMap<String, Email>();

    public void putEmails() {
        emails.put("Dhanji", new Email("dhanji@gmail.com"));
        emails.put("Josh", new Email("josh@noemail.com"));

        System.out.println("Map updated.");    ← Trace map update
    }

    public void read() {
        System.out.println("Dhanji's email address really is "
            + emails.get("Dhanji"));
    }
}
```

there's no guarantee that the program will behave properly. The following program's output is equally likely to happen:

```
Map updated.
Dhanji's email address really is null
```

What's going on? The "Map updated." signal was printed in Thread A *before* the read. We should quite certainly have my email address in the hashtable! We can further confirm the issue as shown in listing 9.2.

Listing 9.2 Unsafe publication of hashtable values even with apparent verification

```
public class UnsafePublication {
    private Map<String, Email> emails = new HashMap<String, Email>();

    public void putEmails() {
        emails.put("Dhanji", new Email("dhanji@gmail.com"));
        emails.put("Josh", new Email("josh@noemail.com"));

        System.out.println("Map updated.");
        read();
    }
    public void read() {
        System.out.println("Dhanji's email address really is "
            + emails.get("Dhanji"));
    }
}
```

← **Ensure map has expected value**

What will this program print? Can it possibly print something as absurd as this?

```
Map updated.
Dhanji's email address really is dhanji@gmail.com
Dhanji's email address really is null
```

Surely, not! In fact, it turns out this is quite possible because a guarantee of order alone is insufficient for safe publication. This is another manifestation of the visibility problems. There are several reasons why Thread B may see a null value, even though as far as Thread A is concerned, the map has been updated. We encountered one of these in an earlier chapter when looking at memory coherency. Without sufficient synchronization, the JVM makes no guarantees about visibility of fields between threads. This is a deliberate choice made by the language designers to allow for maximum flexibility in optimizations on various platforms. Thread A's updates to the hashtable may not yet be synchronized with main memory. Thus, they're not published to other threads. So while a thread can see its own updates, there's no assurance that others will. Now let's look at how to correctly publish to all threads.

9.1.1 **Safe publication**

It's easy to see how this problem can creep into DI as well. Unless properly published, objects may appear incompletely constructed to participating threads, with unavailable or partially constructed dependencies. Consider the equivalent of the hashtable we just saw:

```
public class MoreUnsafePublication {
    private EmailDatabase service;

    public MoreUnsafePublication(EmailDatabase service) {
        this.service = service;
    }

    public void read() {
        System.out.println("Dhanji's email address really is "
            + service.get("Dhanji"));
    }
}
```

← **Potentially unsafe publication**

MoreUnsafePublication is a simple variant of the hashtable that reads an email address using its dependency EmailDatabase. This time the issue of publication is not with the hashtable values but with the EmailDatabase dependency itself. Threads that call method read() cannot rely on the fact that the dependency is available. Without sufficient synchronization, the thread creating the object does not safely publish its fields to other threads. The following could easily result in a NullPointerException:

```
public void read() {
    System.out.println("Dhanji's email address really is "
        + service.get("Dhanji"));
}
```

Or worse, it could result in further corruption of the object graph. A very simple solution to this problem is to declare the field as `final`.¹ Final fields are given the guarantee of safe visibility to all threads concerned. Because these fields are generally always set in the constructor, they'll be visible to all threads once the constructor completes.

¹ See this article on developerWorks for more information: <http://www.ibm.com/developerworks/library/jjtp03304/>

```

public class SafePublication {
    private final EmailDatabase service;

    public SafePublication(EmailDatabase service) {
        this.service = service;
    }

    public void read() {
        System.out.println("Dhanji's email address really is "
            + service.get("Dhanji"));
    }
}

```

In `SafePublication`, threads may call `read()` and expect dependency `EmailDatabase` to be set correctly. This holds for any thread, even those that did not construct the instance. This is a very simple but powerful solution that's an ideal choice for the vast majority of multithreading problems. It's good practice to declare fields `final` even if you believe the instance will only ever participate in one thread, as this gives a clear indication of intent.

As we've stated in earlier chapters, thread-safety is a concept very relevant to dependency injection. In the following section, we'll look at exactly what these semantics imply for wiring objects with dependencies.

9.1.2 Safe wiring

We've just seen how we can run into visibility problems between threads. One typically encounters these problems only with singletons. This is because, in general, only singletons are shared by two (or more) threads. Final fields are a safe solution to this problem if you can guarantee that a reference to the object does not escape during construction (we'll look at what escaping means shortly). Many dependency injectors can also help by providing extra synchronization during the construction of singletons. For instance, `PicoContainer` can be put into a locking mode, which ensures object creation happens in a synchronized state:

```

MutablePicoContainer injector = new DefaultPicoContainer();
injector.as(LOCK, CACHE).addComponent(MyObject.class);
MyObject obj1 = injector.getComponent(MyObject.class);
MyObject obj2 = injector.getComponent(MyObject.class);

```

Instances `obj1` and `obj2` are created in a synchronized fashion, so the dangers of visibility are safely mitigated. The characteristics `LOCK` and `CACHE` instruct `PicoContainer` to use locking and to create singletons. This would be equivalent to the following:


```

public class SynchronizedPublication {
    private Lock lock = new ReentrantLock();
    private MyObject obj;

    public void constructObj() {
        lock.lock();
        try {
            obj = new MyObject();
        } finally {

```

Only one thread may
execute at a time



```

        lock.unlock();
    }

    //other threads can safely access obj's fields now
}

```

Now, fields of `MyObject` are correctly set and visible to all threads regardless of whether or not they are declared `final`:

```

public class MyObject {
    private Dependency dep;
    public MyObject(Dependency dep) {
        this.dep = dep;
    }
    ...
}

```

← **Safely published
to all threads**

Guice also provides a similar guarantee for all of its singletons. However, despite these niceties, one should *not* rely on the injector to do the work of safe wiring. If you design your classes to be well behaved in threaded environments, they will be safe regardless of whether or not you receive help from libraries. This is especially significant if you want to reuse code in different environments. It's also important when designing objects for concurrency. Furthermore, it's an implicit declaration of intent. When someone comes across your code and the final class members, it's very clear that they weren't meant to be changed. Even if you accidentally try to reassign field values after construction, you can't run afoul of mutability since the compiler alerts you to the problem. Recall that any attempt to modify a final field after construction,

```

public class EarlyWarning {
    private final ImmutableDependency dep;

    public EarlyWarning(ImmutableDependency dep) {
        this.dep = dep;
    }

    public void setImmutableDependency(ImmutableDependency newDep) {
        this.dep = newDep;
    }
}

```

← **Illegal
assignment**

results in a clear, fast failure of compilation:

```

Information:Compilation completed with 1 error and 0 warnings
Information:1 error
Information:0 warnings
EarlyWarning.java
Error:Error:line (9)cannot assign a value to final variable dep

```

The documenting thread-safety annotations, `@Immutable`, `@ThreadSafe`, `@NotThreadSafe`, and `@GuardedBy` that we've used in earlier chapters are an additional clarification of behavioral intent. While the injector can help in minor ways, it's ultimately up to you to create code that's both safe and efficient.

In fact, it's a good principle to design without the injector in mind for all threading cases. DI is meant to help your code become more flexible and testable in design, but it doesn't relieve you of the burden of engineering good code. Or should I say joy?

Another interesting but higher-level consideration is deciding when to use a dependency injector and when not to. This can often be quite tricky, as simplistic as it sounds. Ahead, we'll look at rules of thumb that help in making this decision.

9.2 Objects and design

The question of which classes of objects to manage using dependency injection is often a tricky one. Especially for programmers new to the technique, it can be quite a double-edged sword. Some are overeager to apply it, creating everything with the injector. Some are too careful and use DI sparingly, almost like a Factory or Service Locator. Neither of these extremes is particularly prudent. And both are especially counterproductive in large and complex codebases.

Most of the time, it's quite easy to determine what objects ought to be created and managed by dependency injection. A rule of thumb is to leave anything that's a service or action component to the purview of the injector and any class that models data to traditional, by-hand usage. Now let's see that principle in practice, in dealing with a specific example that uses data and service objects.

9.2.1 On data and services

Let's take the case of an imaginary online auction house. Here you have several objects for managing users, bids, list items, reconciliation, and so forth. These are couched as three primary services, as shown in figure 9.3:

- AuctionManager—Manages bids and auction status
- ItemManager—Manages list items and descriptions
- UserManager—Manages user accounts and history

These three services are classes with several dependencies of their own. For example, UserManager needs a DAO to read and write user details to a data store:

```
public class UserManager {
    private final UserDao userDao;

    public UserManager(UserDao userDao) {
        this.userDao = userDao;
    }

    //operations on user...
}
```



Figure 9.3 Three foundational services of the auction house application

UserDao, in the example, is our interface to the data store. UserManager uses it to create new users, update details such as name and password, and in rare cases mark the account as inactive. Listing 9.3 shows the same class in some detail (also see figure 9.4).

Listing 9.3 UserManager performs many operations on user data

```
public class UserManager {
    private final UserDao userDao;

    public UserManager(UserDao userDao) {
        this.userDao = userDao;
    }

    public void createNewUser(User user) {    ← Registers a new user
        validate(user);
        userDao.save(user);
    }

    public void updatePassword(long userId,    ← Updates an existing
        String password) {                    user's password
        User user = userDao.read(userId);
        user.setPassword(password);

        validate(user);
        userDao.save(user);
    }

    public void deactivate(long userId) {    ← Sets an account
        User user = userDao.read(userId);    as inactive
        user.deactivate();

        userDao.save(user);
    }

    ...
}
```

In listing 9.3, we have three methods that perform some kind of manipulation of the User data object. `createNewUser()` registers a new user (represented by an instance of the User object) by saving it to a data store:

```
public void createNewUser(User user) {
    validate(user);
    userDao.save(user);
}
```

This method also validates the instance of user, ensuring that all the data going in is correct. We don't deal with reporting validation errors at this juncture, since it is already assumed this has been done at the presentation layer (that is, the website). This validation step is purely a safeguard against programmer error. Once we're sure the data is valid, it's passed to UserDao to be saved.

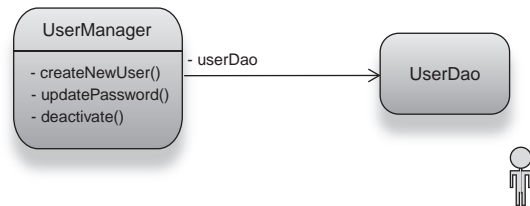


Figure 9.4 A class model of UserManager and its dependency, UserDao

Similarly, method `updatePassword()` performs data manipulation on `User`, but this time it's of an instance that already exists. The specific user is identified by a numeric `userId`, and it's the job of `UserDao` to locate and retrieve the relevant instance from the data store:

```
public void updatePassword(long userId, String password) {
    User user = userDao.read(userId);
    user.setPassword(password);

    validate(user);
    userDao.save(user);
}
```

We then set the new password on this instance, signaling a change of password, and resave the details in the same fashion.

The third method in our set of operations deactivates a user account (presumably due to inactivity or violation of terms of use). Aptly named, method `deactivate()` takes in the numeric `userId`, deactivates it, and saves the relevant instance:

```
public void deactivate(long userId) {
    User user = userDao.read(userId);
    user.deactivate();

    userDao.save(user);
}
```

We don't need to validate the `User` this time since we aren't modifying any data in it with untrusted data received from actual user input.

One thing that's clear to us from these three operations is that `UserDao` and `UserManager` are services. The common thread that runs through them is that they perform operations on instances of the `User` class, which is data. Clearly, `UserDao` and `UserManager` are appropriate to be constructed and tested with DI. Class `User`, on the other hand, is constructed often at the presentation layer and has no dependencies. Listing 9.4 describes this data model class (also shown in figure 9.5).

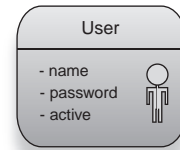


Figure 9.5 Class model of `User` and its properties

Listing 9.4 The `User` data model class

```
public class User {
    private long userId;
    private String name;
    private String password;
    private boolean active = true;

    public User(long userId, String name) {
        this.userId = userId;
        this.name = name;
    }

    public long getUserId() {
```



```

        return userId;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public boolean isActive() {
        return active;
    }

    public void deactivate() {
        this.active = false;
    }
}

```

In listing 9.4, all four fields are *scalar* data types. In other words, they are fields of type `String`, `boolean`, or `long`. And each datum has no real domain semantics. There's nothing special about `Strings`, `booleans`, or `longs` insofar as an auction house is concerned. Consider this in contrast to `User`, which is itself a data type, but one that's specific to the problem domain we're concerned with. It also captures specific operations around that domain (for example, *deactivating* user accounts).

Apart from having no real dependency structure, another interesting point about the `User` class is that almost all of its fields are mutable. Setter methods exist for `name`, `password`, and the active flag. This is because `User` models the real-world state of a user's account. Several instances of this class exist, and the specific values of each instance may change gradually over time, that is, when someone changes his password or has his account deactivated.

Clearly, `User` objects are not ideal to be instantiated by DI. There's no gain to be had from interception or lifecycle for `Users`. `Scope` also doesn't quite fit in this scenario, since all `Users` are not scoped, but at the same time it can persist forever in a data store. Furthermore, there isn't anything significant to test in `User` code. Most methods are dumb setters or accessors, so it's appropriate to create and manage them by hand, like any other scalar object. Listing 9.5 is an example of doing just that from a new-user registration page.

```

@RequestScoped
public class NewUserPage {
    private final UserManager userManager;

    @Inject

```

```

public NewUserPage(UserManager userManager) {
    this.userManager = userManager;
}

public HTML registerNewUser(String name) {
    userManager.createNewUser(new User(nextUserId(), name));
    ...
}
}

```

`NewUserPage` represents a web page on the auction website where new users sign up to list and bid on items (see figure 9.6).

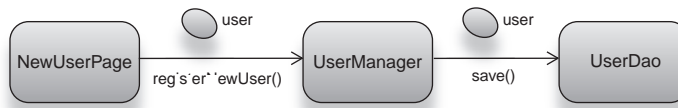


Figure 9.6 Creating a new User with relevant services

It is *HTTP request scoped* as is marked by the `@RequestScoped` annotation (recall guice-servlet and web scopes from chapter 5):

```

@RequestScoped
public class NewUserPage {
    private final UserManager userManager;
    ...
}

```

`NewUserPage` also takes data coming in from browser data entry and converts it into a created user by passing it on to the `UserManager`:

```

public HTML registerNewUser(String name) {
    userManager.createNewUser(new User(nextUserId(), name));
    ...
}

```

`UserManager` is a singleton service that's shared by all instances of the `NewUserPage` and is called at various data points for purposes like creating users, deactivating their accounts, or changing their details.

The other services, `AuctionManager` and `ItemManager`, also model actions on data model objects relating to auctions, list items, and so forth. Here too, the data model objects can be separated from their operations along simple lines (see figure 9.7).

Auctions, Items, ShoppingCarts, Money, and Users are all data classes that are operated on by the aforementioned singleton services and web pages. We can thus make an easy distinction between these classes and those that can benefit from dependency injection.

Another important design principle is *encapsulation*, sometimes called information hiding. In the following section, we'll look at how this applies to DI and techniques for taking advantage of it.

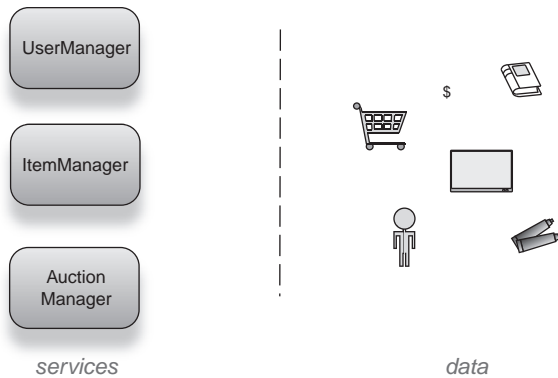


Figure 9.7 Separate data from operations on data.

9.2.2 On better encapsulation

One of the core principles of OOP is encapsulation, that is, the hiding of any information within a component that is relevant *only* to that component. In classes, this takes the form of marking members as `private`. Not only does this make sense in terms of hiding information that's irrelevant to anyone else, it also ensures against accidental *leaking* of semantics.

Leaking of private member fields can be very dangerous since it can lead to tight coupling between components. A class making use of a specific kind of messaging service (say, email messaging) should not expose its implementation details to the outside world. If other classes mistakenly begin using its dependencies, this can lead to tight coupling. Let's take the case of a `Messenger` service represented by the following interface

```
public interface Messenger {
    void send(Message msg);
}
```

and its email-backed implementation:

```
public class Emler implements Messenger {
    public void send(Message msg) {
        //send message via email...
    }
}
```

`Emler` converts the incoming generic message into an email and sends it away to recipients. So far so good. Now let's see what happens if we accidentally leak this abstraction:

```
public interface Messenger {
    void send(EmailMessage msg);
}
public class Emler implements Messenger {
    public void send(EmailMessage msg) {
```

```

        //send message via email...
    }
}

```

Now, both `Messaging` and `Emler` send `EmailMessages` in method `sendMessage()`. While this may seem okay at first glance, a closer examination reveals otherwise. Say you wrote a new kind of messaging service that used the popular *Jabber* instant messaging protocol to send messages:

```

public class JabberMessenger implements Messenger {
    public void send(JabberMessage msg) {
        //send message via jabber...
    }
    ...
}

```

Does not implement interface method

Because of the leaked abstraction, method `send()` in `JabberMessenger` does not actually implement `send()` from the `Messenger` interface. Fortunately, this code will fail on compilation, complaining that the `Messenger` interface is not fully implemented. But we're still stuck with the problem of not being able to change the messaging service's transport—stuck with email, that is. This highlights what is an extremely poor abstraction that has pretty much destroyed our encapsulation. We have effectively exposed the internals of the `Emler` class directly and rendered the `Messenger` interface irrelevant.

Of course, this is easily fixed in our case by reverting to the original implementation:

```

public interface Messenger {
    void send(Message msg);
}

public class Emler implements Messenger {
    public void send(Message msg) {
        EmailMessage email = convert(msg);
        //send message via email...
    }
}

```

Converts to email-specific message

This version of `Emler` correctly converts the incoming implementation-neutral `Message` object into an `EmailMessage`, which can be sent over email. Now it's easy for us to create and swap in a `JabberMessenger` system with little difficulty and no impact to client code:

```

public class JabberMessenger implements Messenger {
    public void send(Message msg) {
        JabberMessage jab = convert(msg);
        //send message via jabber...
    }
    ...
}

```

Converts to Jabber-specific message

JabberMessenger now successfully compiles, and the system behaves as expected.

We can take the idea of encapsulation a step further. It isn't always possible to make implementation details private—sometimes implementation classes need to share functionality among themselves. For example, a JabberMessenger interface may depend on a JabberMessageConverter and a JabberTransport in order to convert and send the message:

```
public class JabberMessenger implements Messenger {
    private final JabberTransport transport;
    private final JabberMessageConverter converter;
    public void send(Message msg) {
        ...
    }
}
```

Jabber-specific dependencies

These are dependencies that exist as public classes. And therefore they can be accessed, subclassed, and used by anyone, leading to a potential horde of tight coupling. In Java, there's an extra layer of visibility that we can take advantage of to prevent anyone outside a package (namespace) from knowing or using classes. This is sometimes called *package-privacy* or *package-local* access. Package-local access has no special keyword and is denoted by the lack of an access specifier. Dependency injectors allow us to take advantage of this special visibility by exposing only service interfaces and configuration from each package, hiding any implementation details within. Let's apply this to the Jabber example with Guice; see listing 9.5.

Listing 9.5 Package encapsulation of the Jabber services

```
package example.messaging;

public interface Messenger {
    void send(Message msg);
}

class JabberMessenger implements Messenger {
    private final JabberTransport transport;
    private final JabberMessageConverter converter;

    public void send(Message msg) {
        ...
    }
}

class JabberTransport {
    ...
}

class JabberMessageConverter {
    ...
}
```

In listing 9.5, all of the implementation classes are declared package-local. They can't be seen from outside the `example.messaging` package by any classes. Moreover, all communication and use of Jabber services must occur through the `Messenger` interface. This

allows us a degree of control over the contract and behavior of our messaging module. It's also easy to specify API behavior to clients, and the documentation for this module is ridiculously simple (nothing more than the single-method `Messenger` interface).

All we need to do to expose Jabber messaging to clients is place a single Guice module in the package, which contains all the nitty-gritty binding details:

```
public class MessagingModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(Messenger.class).to(JabberMessenger.class);
    }
}
```

The messaging module does nothing more than instruct the injector to use `JabberMessenger` wherever `Messaging` services are needed. Here's such a user of `Jabber` messaging:

```
public class MessageClient {
    private final Messenger messenger;

    public MessageClient(Messenger messenger) {
        this.messenger = messenger;
    }

    public void go() {
        messenger.send(new Message("Dhanji", "Hi there!"));
    }
}
```

This ensures loose coupling between client code and the messaging module, since clients need only attach to the `Messaging` interface. We can even use multiple implementations this way and distinguish between them using binding annotations:

```
public class MessagingModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(Messenger.class).annotatedWith(Im.class).to(JabberMessenger.class);
        bind(Messenger.class).annotatedWith(Mail.class).to(Emailer.class);
    }
}
```

Now clients can choose which implementation to use without coupling to any internals:

```
public class MessageClient {
    private final Messenger mailMessenger;
    private final Messenger imMessenger;

    public MessageClient(@Im Messenger imMessenger, @Mail Messenger
        mailMessenger) {
        this.imMessenger = imMessenger;
        this.mailMessenger = mailMessenger;
    }

    public void go() {
```

```

        Message msg = new Message("Dhanji", "Hi there!");
        imMessenger.send(msg);
        mailMessenger.send(msg);
    }
}

```

And all is well with the world.

A good example of this kind of encapsulation can be found in the warp-persist integration library we explored in chapter 8. Figure 9.8 is a screenshot of warp-persist's package tree. Notice the classes with a hollow bullet point to the left of them—these are package-local. Public components are restricted to hardly any public classes at all, thus vastly reducing the potential for tight coupling and abstraction leaking.

This is directly reflected in the API documentation for warp-persist. Look at the resulting streamlined and simple Javadoc frame in figure 9.9.

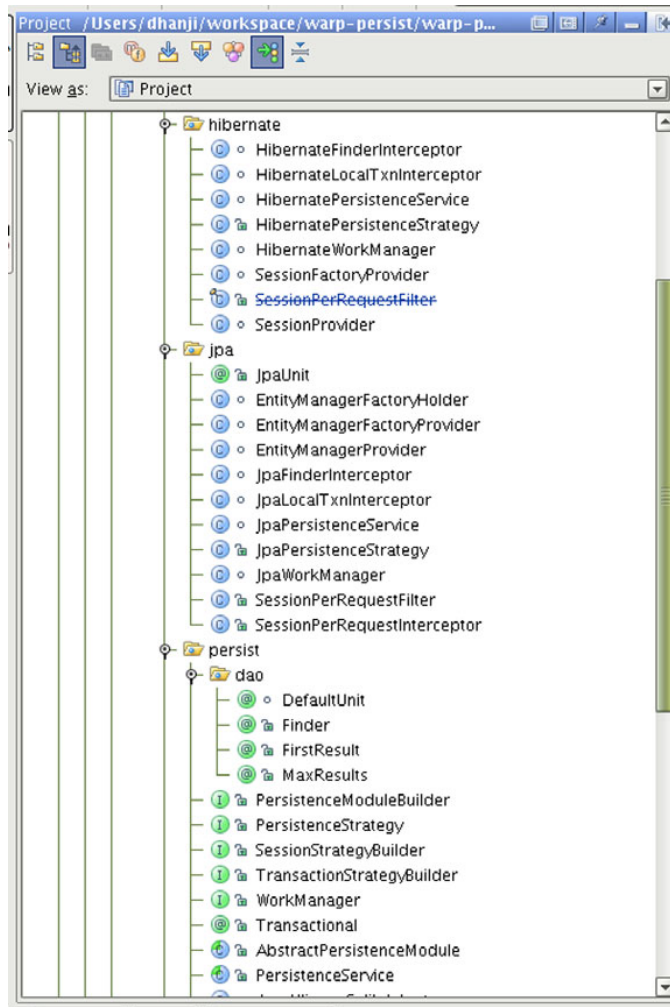


Figure 9.8 Screenshot of warp-persist's package tree (hollow bullets are package-private classes)

All Classes

- [Db4oBindingSupport](#)
- [Db4oObjects](#)
- [Finder](#)
- [FirstResult](#)
- [HibernateBindingSupport](#)
- [JpaBindingSupport](#)
- [JpaUnit](#)
- [MaxResults](#)
- [PersistenceModuleBuilder](#)
- [PersistenceService](#)
- [SessionPerRequestFilter](#)
- [SessionPerRequestFilter](#)
- [SessionStrategyBuilder](#)
- [Text](#)
- [Transactional](#)
- [TransactionStrategy](#)
- [TransactionType](#)
- [UnitOfWork](#)
- [WorkManager](#)

Overview Package Class **Tree** Deprecated Index Help

PREV NEXT FRAMES NO FRAMES

Packages

com.wideplay.warp.db4o	
com.wideplay.warp.hibernate	
com.wideplay.warp.jpa	
com.wideplay.warp.persist	
com.wideplay.warp.persist.dao	
com.wideplay.warp.util	

Overview Package Class **Tree** Deprecated Index Help

PREV NEXT FRAMES NO FRAMES

Figure 9.9 Javadoc of warp-persist's main package consisting of only enums, annotations, interfaces, and abstract classes

Neat! A good rule of thumb is to apply package privacy to anything that's not an interface, enum, annotation, or abstract class because these types have very low potential for leaking abstraction logic. You can apply these principles and take advantage of dependency injection to get better encapsulation at the package and module levels.

In previous chapters, we looked at safe design with respect to multiple threads. This often meant correct synchronization. Earlier in this chapter we also looked at visibility. In the following section, we'll take a more performance-focused approach and look at the action of multiple threads concurrently.

9.3 Objects and concurrency

Concurrency is increasingly a very important aspect of modern applications. As we scale to higher levels of traffic and demand, there's a greater need for multiple concurrent threads of execution. Thus, the role of objects managed by the dependency injector is extremely important. Singletons are a particularly significant example of this need.

In a large web application handling several hundreds of requests per minute, a poorly designed singleton can be a serious bottleneck. It introduces ceilings on concurrent performance and can even render the application unscalable under certain conditions.

Poor concurrent behavior is also more common than you might think. And since its effects are highlighted only during performance testing, it can be difficult to identify and mitigate, so it's quite relevant for us to study the effects of concurrency on singletons.

Mutability is an essential variable in this problem, so let's start with that.

9.3.1 *More on mutability*

Earlier in this chapter we explored the dangers of mutable objects as they manifested in visibility and publication problems. We also established that declaring fields `final` and making objects immutable was an efficient and robust solution to the thread-safety question. Now we'll explore exactly what it means to be immutable. Some pitfalls are contained within the idea of immutability too. To change things up, let's explore this as a series of puzzles.

IMMUTABILITY PUZZLE 1

Is the following class, `Book`, immutable?

```
public class Book {
    private String title;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

ANSWER 1

This one's easy: no. The value of its `title` field can be changed arbitrarily by calling `setTitle()`, so it is clearly *not* immutable. We can make `Book` immutable by declaring `title` `final`:

```
public class ImmutableBook {
    private final String title;

    public ImmutableBook(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }
}
```

Once set in the constructor, the value of `title` cannot change.

IMMUTABILITY PUZZLE 2

Is the following class, `AddressBook`, immutable?

```
public class AddressBook {
    private final String[] names;

    public AddressBook(String[] names) {
        this.names = names;
    }

    public String[] getNames() {
        return names;
    }
}
```

ANSWER 2

The value of the `names` field is set once in the constructor and is declared `final`. So `AddressBook` should be immutable, right? No! In fact, the subtle point is that since `names` is an array, only the *reference* to it is immutable by declaring it `final`. The following code is perfectly legal and can potentially lead to a world of hurt where multiple threads are concerned:

```
public class AddressBookMutator {
    private final AddressBook book;

    @Inject
    public AddressBookMutator(AddressBook book) {
        this.book = book;
    }

    public void mutate() {
        String[] names = book.getNames();

        for (int i = 0; i < names.length; i++)
            names[i] = "Censored!";

        for (int i = 0; i < names.length; i++)
            System.out.println(book.getNames()[i]);
    }
}
```

Method `mutate()` destructively updates the array, even though field `names` is unchangeable. If you run the program, it prints "Censored!" for every name in the book. The only real solution to this problem is not to use arrays—or to use them very sparingly behind well-understood safeguards and documentation. Choose library collections (such as those found in `java.util`) classes where possible as these can be guarded by unmodifiable wrappers. See puzzle 3 for an illustration of using `java.util.List` instead of an array.

IMMUTABILITY PUZZLE 3

Is the following class, `BetterAddressBook`, immutable?

```
public class BetterAddressBook {
    private final List<String> names;

    public BetterAddressBook(List<String> names) {
        this.names = Collections.unmodifiableList(names);
    }

    public List<String> getNames() {
        return names;
    }
}
```

ANSWER 3

Thankfully, yes; `BetterAddressBook` is immutable. The wrapper provided by the `Collections` library ensures that no updates can be made to the list once it has been set. The following code, though it compiles, results in an exception at runtime:

```
BetterAddressBook book = new BetterAddressBook(Arrays.asList(
    "Landau", "Weinberg", "Hawking"));
book.getNames().add(0, "Montana");
```

IMMUTABILITY PUZZLE 4

This is a variant on puzzle 3. Take the same `BetterAddressBook` class we saw earlier. Is it at all possible to construct it in such a way that I can mutate it? You're not allowed to change the code of `BetterAddressBook`.

ANSWER 4

The answer is simple, if a bit confounding:

```
List<String> physicists = new ArrayList<String>();
physicists.addAll(Arrays.asList("Landau", "Weinberg", "Hawking"));

BetterAddressBook book = new BetterAddressBook(physicists);

physicists.add("Einstein");
```

Now an iteration through `BetterAddressBook`'s list of names

```
for (String name : book.getNames())
    System.out.println(name);
```

actually produces the mutated list:

```
Landau
Weinberg
Hawking
Einstein
```

So, really, we must revise what we said in the answer to puzzle 3. `BetterAddressBook` is immutable only if its dependency list is not leaked anywhere else. Better yet, you can rewrite a completely safe version of it by copying the list at the time of its construction:

```
@Immutable
public class BestAddressBook {
    private final List<String> names;

    public BestAddressBook(List<String> names) {
        this.names = Collections.unmodifiableList(
            new ArrayList<String>(names));
    }

    public List<String> getNames() {
        return names;
    }
}
```

Now you're free to leak and mutate the original list,

```
List<String> physicists = new ArrayList<String>();
physicists.addAll(Arrays.asList("Landau", "Weinberg", "Hawking"));

BetterAddressBook book = new BetterAddressBook(physicists);

physicists.clear();
physicists.add("Darwin");
physicists.add("Wallace");
physicists.add("Dawkins");

for (String name : book.getNames())
    System.out.println(name);
```

and `BestAddressBook` remains unaffected:

```
Landau
Weinberg
Hawking
```

While it may not always be necessary to take such a cautious approach, it's advisable to copy argument lists if you're at all unsure about them escaping into other uses subsequent to construction of the immutable object.

IMMUTABILITY PUZZLE 5

Is the following class, `Library`, immutable? (Recall `Book` from puzzle 1.)

```
public class Library {
    private final List<Book> books;

    public Library(List<Book> books) {
        this.books = Collections.unmodifiableList(
            new ArrayList<Book>(books));
    }

    public List<Book> getBooks() {
        return books;
    }
}
```

ANSWER 5

`Library` depends on a list of `Books`, but it takes care to wrap the incoming list in an unmodifiable wrapper and copies it prior to doing so. Of course, its only field is final too. Everything looks right—or does it? It turns out that `Library` is mutable! While the collection of books is unchangeable, there's still the `Book` object itself, which, as you may recall from the first puzzle, allows its title to be set:

```
Book book = new Book();
book.setTitle("Dependency Injection");

Library library = new Library(Arrays.asList(book));

library.getBooks().get(0).setTitle("The Tempest");    //mutates Library
```

The golden rule with immutability and object graphs is that every dependency of an object must also be immutable. In the case of `BestAddressBook`, we got lucky, since `Strings` in Java are already immutable. Take care to ensure that every dependency you have is safely immutable before declaring an object as such. The `@Immutable` annotations mentioned in chapter 6 help a great deal in conveying and documenting this intent.

While immutability is a desirable goal, often you need to change state in order to do useful work. In the next section we'll look at how to do this and also keep our code performant.

9.3.2 Synchronization vs. concurrency

Sometimes it just happens that you really need mutable objects. It isn't always possible to make everything immutable (though it would be nice!). This need usually arises

when you have to maintain some kind of central state in a system that's shared between many or all threads. This is fairly common in large applications. You may need a counter to keep track of the number of requests to a particular resource, or you may be caching heavyweight data in application memory to avoid expensive trips to a database.

Generally, you want to isolate any such use cases and design them very carefully, giving yourself ample slack to test and reason about these multithreaded services. Essentially, such services need to synchronize the data between threads in such a way that the data remains coherent and that threads are not paused for long periods awaiting data. These are two very different problems, with quite different solutions. And they are classed under the headings of *synchronization* and *concurrency*, respectively.

THREAD-SAFE COUNTING

Consider a simple counter. Every time a message is received, it increments the count by one. There are several message handler threads, and they must all update the same counter. If they updated separate counters, there would be no thread-safety problem, but we'd be unable to tell what the total count was. So here's what such a counter might look like:

```
public class MessageCounter {
    private int count;

    public void messageReceived() {
        count++;
    }
}
```

Method `messageReceived()` is called by each message-handling thread upon receiving a message to increment the count. Now, variable `count` is not final and is updated concurrently by more than one thread. So it is very possible that it will get out of sync and read an invalid or corrupt count. A simple solution is to synchronize the counter so that only one thread may increment the count at any given time:

```
public class MessageCounter {
    private int count = 0;

    public synchronized void messageReceived() {
        count++;
    }
}
```

In this version, the singleton-scoped instance of `MessageCounter` itself acts as the lock. Each message-receiving thread must wait its turn in order to acquire the lock, increment the count, and release it. This solution works, and we are assured that the count never goes out of sync and that each increment is published safely to all threads.

For this very simple case, the synchronization solution is probably good enough. But think about what happens when there is an enormous number of threads going through the counter. Every one of them must wait to acquire and release the lock before it can proceed with handling the message. This can lead to a serious bottleneck. The problem gets even worse if additional work needs to be done before the

counter can be incremented (say, looking up who sent the message from a hashtable). All threads must wait inexorably, until the one holding the lock can finish. This can lead to very poor throughput.

CONCURRENT COUNTING

Concurrent algorithms and data structures are designed with speed and scalability in mind. Multiple threads may liberally hit the critical code without suffering the single-file throughput problems of synchronization. This is accomplished in many ways, primarily by taking advantage of some special *atomic* operations provided by modern CPUs. Atomic operations execute in one go on the CPU and cannot be interrupted by another thread being prioritized while they're executing. Atomic operations are thus somewhat like a very small block of synchronized code that performs just one instruction.

In Java, these are modeled by the `java.util.concurrent.atomic` library and associated data structures found in the `java.util.concurrent` package. Using these we can rewrite the counter to be concurrent rather than sequential:

```
public class MessageCounter {
    private final AtomicInteger count = new AtomicInteger(0);

    public void messageReceived() {
        count.incrementAndGet();
    }
}
```

The method `incrementAndGet()` executes atomically. This is opposed to `count++`, which is actually three operations masquerading as one:

- 1 Read value from `count`.
- 2 Increment value by one.
- 3 Write new value back to `count`.

In the *non-atomic* situation, intervening threads can easily alter `count` concurrently and corrupt its value. With the atomic `incrementAndGet()` this cannot happen.

Furthermore, if a thread is slow in incrementing its count (due to additional work that it's doing, perhaps), this doesn't block other threads from making progress in the meantime as synchronization would. This leads to real concurrency of threads and a substantial increase in throughput.

The value of concurrent data structures becomes more readily apparent when dealing with more complex forms of data, such as those stored in hashtables. Consider the second use case we mentioned, where data is placed into a cache and then looked up by multiple threads to prevent expensive trips to a database. Using a traditional synchronized hashtable this would look as follows:

```
public class SimpleCache {
    private final Map<String, Data> map =
        Collections.synchronizedMap(new HashMap<String, Data>());

    public void set(String key, Data val) {
        map.put(key, val);
    }
}
```

```

    public Data get(String key) {
        return map.get(key);
    }
}

```

SimpleCache uses a synchronized wrapper around a simple hashtable. This wrapper has a single lock that is acquired on every `get()` and every `put()` operation. This is done to safely publish values (as we discussed early in this chapter) to all threads. However, SimpleCache is extremely slow since every thread must wait for the hashmap to perform a lookup operation and release the lock held by the current thread. `put()` operations can be even worse since they can involve resizing the underlying array when it becomes full. In any high-traffic environment, this is unacceptably underperformant. What we need is for threads to be able to look up values concurrently and perhaps wait only when there are many insertions going on. What we need is a concurrent hashtable:

```

import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentCache {
    private final Map<String, Data> map
        = new ConcurrentHashMap<String, Data>();

    public void set(String key, Data val) {
        map.put(key, val);
    }

    public Data get(String key) {
        return map.get(key);
    }
}

```

This is a special hashtable implementation because it doesn't use locking to read from the map at all! It allows multiple threads to make progress to the same values without requiring a coarse-grained lock as synchronized hashtables do. Instead it locks only on insertion operations but using more fine-grained locks distributed across the hashtable. The keys are partitioned along a configurable number of *stripes*. Each stripe is assigned a lock, and any concurrent insertions to keys in the same stripe must wait for sequential access (just like synchronization). Dividing the table into even a small number of stripes is several times more efficient than locking the entire table every time.

Opt for concurrent data structures wherever you can. And reason carefully about the semantics of making any object a singleton, because you will have to worry about thread safety and concurrency. Finally, conquer your problem by assessing it through the three cordons of immutability, safe publication, and concurrency.

9.4 **Summary**

This chapter was a thorough workout in best practices and concurrency. Architecturally, the best practices portended by a language and its design patterns are the same as those required by dependency injection. Where possible you should design with testability in mind but *not* rely on the behavior of an injector.

One of the primary problems to face with singletons and objects shared between threads is that of visibility. Dependencies set on an object need to be visible to all threads that use the object. This is known as safe publication. One way to guarantee proper visibility is to make objects immutable. This involves plenty more than merely declaring fields final, as you saw with a series of five puzzles on the topic. Immutability requires that every object in a graph be immutable and that no dependencies “escape” after construction.

Designing objects for better encapsulation is also very important. Dependency injectors like Guice allow you to hide implementation classes in package privacy and expose only interfaces to clients. This helps prevent tight coupling and the accidental leakage of internals. Another important design question is which objects to create and manage via the injector. A rule of thumb is to ask whether an object is a service component or whether it models data. If the latter, there’s no benefit to be gained from dependency injection, lifecycle, scope, and so on, and you’re better served working with them by hand.

Finally, we explored what it means to be thread-safe in mutable cases. This was especially tricky, since synchronization through locks is insufficient in all cases. In high-traffic environments, you cannot afford to let threads queue up, waiting to acquire a single global lock to a shared resource. The answer is to use concurrent data structures such as those provided by the `java.util.concurrent` library. These data structures purport atomic operations and more fine-grained locking, which allows for better throughput in systems that require a high level of concurrency.

This chapter gave you a solid, low-level grounding in designing code for safe, highly performant Java applications using dependency injection. Next we’ll take a broader view and look at how to integrate an application with other libraries and utility frameworks.

10

Integrating with third-party frameworks

This chapter covers:

- Exploring lessons for framework designers
- Exploring framework anti-patterns
- Dealing with type-safety
- Dealing with framework interoperability

“No, no, you’re not thinking! You’re just being logical.”

—Niels Bohr

Our investigation of application design has led us to dependency injection in various forms and working with environments such as web applications, data persistence, filesystems, and GUIs. The techniques we’ve examined help us tie together the otherwise disparate parts into cohesive modules that are easy to test and maintain.

Such scenarios typically call for the use of utility libraries that facilitate better application design. When you work with persistence, you use an ORM tool like Hibernate. When you work with service clusters, it may be Oracle Coherence.

One of the major impediments to smooth integration between a dependency injector and a given library is that many of these libraries already provide a minimal form of IoC. In other words, they provide some means, usually half baked, to construct and wire objects with dependencies. Generally, these are specific to their areas of need, such as a pluggable connection manager in the case of a persistence engine or a strategy for listening to certain events in the case of a graphical user interface GUI.

Most of the time these solutions focus on only a very specific area and ignore broader requirements for testing and component encapsulation. These typically come with restrictions that make testing and integration difficult—and downright impossible in certain cases. In this chapter, we'll show how this lack of architectural foresight can lead to very poor extensibility solutions and how, with a little bit of thought, a library can be very flexible and easy to integrate with. We'll study popular frameworks that have made the wrong choice by either hiding too much of the component model or by creating their own.

We'll now do a short analysis of how frameworks create fragmentation by each creating its own partial dependency injection idiom.

10.1 *Fragmentation of DI solutions*

Nearly every application these days does some form of DI. That is, it moves part of its object construction and wiring responsibilities off to configured library code. This can take many forms. In the Java Servlet framework, a very primitive form of this is evident with the registration of servlets and filters in web.xml, as shown in listing 10.1.

Listing 10.1 web.xml is a precursor to solutions using DI

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" >

  <filter>
    <filter-name>webFilter</filter-name>
    <filter-class>
      com.wideplay.example.RequestPrintingFilter
    </filter-class>
  </filter>

  <filter-mapping>
    <filter-name>webFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <servlet>
    <servlet-name>helloServlet</servlet-name>
    <servlet-class>
      com.wideplay.example.servlets.HelloServlet
    </servlet-class>
  </servlet>
```

```

<servlet-mapping>
  <servlet-name>helloServlet</servlet-name>
  <url-pattern>/hi/*</url-pattern>
</servlet-mapping>
</web-app>

```

In listing 10.1, we have a web.xml configuration file for the servlet container that configures a single web application with

- A filter—`com.wideplay.example.RequestPrintingFilter`
- An HTTP servlet—`com.wideplay.example.servlets.HelloServlet`

The filter is a subclass of `javax.servlet.Filter` and performs the simple task of tracing each request by printing its contents. It's configured with the `<filter>` tag as follows:

```

<filter>
  <filter-name>webFilter</filter-name>
  <filter-class>com.wideplay.example.RequestPrintingFilter</filter-class>
</filter>

```

This is reminiscent of Spring's `<bean>` tag, where a similar kind of configuration occurs. By analogy that might be:

```
<bean id="webFilter" class="com.wideplay.example.RequestPrintingFilter"/>
```

Similarly, there's a `<servlet>` tag that allows you to map a `javax.servlet.http.HttpServlet` to a string identifier and URL pattern. The essential difference between this solution and that offered by Spring, Guice, or any other dependency injector is that the servlet framework does not allow you to configure any dependencies for servlets or filters. This is a rather poor state of affairs, since we can virtually guarantee that any serious web application will have many dependencies.

This configuration is fairly restrictive on the structure of servlets (and filters):

- A servlet must have a public, nullary (zero-argument) constructor.
- A servlet class must itself be public and be a concrete type (rather than an interface or abstract class).
- A servlet's methods cannot be intercepted for behavior modification (see chapter 8).
- A servlet must effectively have singleton scope.

We can already see that this is getting to be an inexorable set of restrictions on the design of our application. If all servlets are singletons with nullary constructors, then testing them is very painful.¹ As a result of this and other deficiencies in the programming model, many alternative web frameworks have arisen on top of the servlet framework. These aim to simplify and alleviate the problems portended by the restrictiveness of the servlet programming model, not the least of which has to do with DI (see table 10.1 for a comparison of these solutions).

¹ See the problems with testing singletons described in chapter 1 and more comprehensively in chapter 5.

Table 10.1 Java web frameworks (based on servlet) and their solutions for DI

Library	Out-of-box solution	Integrates Spring/Guice/others?	Website
Apache Wicket	Custom factory, no DI	Partially through plug-in. ^a	http://wicket.apache.org
Apache Struts2 (formerly WebWork)	Custom factory, no DI	Fully, depending on plug-in.	http://struts.apache.org
Apache Tapestry 4	Apache HiveMind	Extension integrates Spring through HiveMind lookups.	http://tapestry.apache.org/tapestry4.1
Apache Tapestry 5	TapestryIoC injector	Partially via extension. ^b	http://tapestry.apache.org
JavaServer Faces 1.2	Built-in, very limited DI ^c	Extension integrates variable lookups in Spring.	http://java.sun.com/javaee/jaserverfaces
Spring MVC	Spring IoC	Pretty much tied to Spring.	http://www.springframework.org
Google Sitebricks	Google Guice	Integrates Spring fully via Guice's Spring module.	http://code.google.com/p/google-sitebricks

a. I say partially because Wicket's page objects cannot be constructed by DI libraries (and are therefore constructor injected), even with plug-ins.

b. Tapestry's Spring integration requires special annotations when looking up Spring beans. There is no official support for Guice.

c. JSF's built-in injector lacks support for constructor injection and interception, and it has limitations object identifiers that prevent the use of namespaces (see chapter 2 for an examination of namespaces in dependency identifiers).

As you can see from table 10.1, there isn't a standard programming model for integrating dependency injectors with libraries based around servlets. Only Apache Struts2 and Google Sitebricks fully integrate Guice and Spring.

The situation doesn't get much better with other types of libraries either. The standard EJBs programming model provides its own form of dependency injection that supports direct field injection and basic interception via the `javax.interceptor` library. Direct field injection, as we discussed in chapter 3, is very difficult to test and mandates the presence of nullary constructors, which also means that controlling scope is not an option—all stateless session EJBs (service objects) exist for the duration of a method call. Stateful session EJBs are somewhat longer lived, as you saw in chapter 7. But these too are scoped outside your direct control and only marginally longer lived.

Why are we at such a sorry circumstance? Frameworks ought to simplify our lives, not complicate them. The reasons are numerous, but essentially it boils down to the fact that we are only recently learning the value of testability, loose-coupling, scope, and the other benefits ascribed to DI. Moreover, the awareness of how to go about designing with these principles in mind is still maturing. Guice is only a little over two years old itself. PicoContainer's and Spring's modern incarnations are even more recent than that.

That having been said, let's look at what future frameworks and libraries can do to avoid these traps and prevent this sort of unnecessary fragmentation.

10.2 *Lessons for framework designers*

Most of the critical problems with integration can be avoided if framework designers keep one fundamental principle in mind: *testability*. This means that every bit of client and service code that a framework interacts with ought to be easy to test. This naturally leads to broader concepts like loose coupling, scoping, and modular design. Library components should be easily replaced with mock counterparts. And pluggable functionality should not mandate unnecessary restrictions on user code (servlets requiring a public, nullary constructor, for example).

Replacement with mocks is at once the most essential and most overlooked feature in framework designs. While most frameworks are themselves rigorously tested, they often fail to consider that it is equally important that client code be conducive to testing. This means considering the design of client code as much as the design of APIs and of implementation logic. Frameworks designed with such a leaning often turn out to be easier to use and understand, since there are fewer points of integration, and these are naturally more succinct. Another significant problem is encapsulation—you saw this in the previous chapter when considering the design of your classes. The more public classes you expose, the more danger there is that clients will begin to extend and use them. If you want to make major changes to your framework's architecture or internal design, this process is difficult if your users are heavily tied to several parts of your framework. The Spring Framework's SpringMVC suffers from this. Every single class is public and nonfinal. This means users can bind to any of this code, and it can never change without breaking a lot of client applications.

Even many of Spring's modules extend and use other parts of the framework. This is why people find it very difficult to use Spring modules outside the Spring injector. Spring Security is a classic example of this—many of its components rely strongly on Spring's lifecycle system and on JavaBeans property editors, which make it difficult to port for use with any other dependency injection system or even to use by hand.

Forcing users to extend framework base classes for functionality is also problematic because it blurs the lines between a dependency and a parent class. Composition is a preferable option, because a delegate is easily replaced with a mock or stub for unit and integration testing. Parent class methods are harder to mock and therefore less conducive to testing.

These problems can be classified broadly into three categories:

- Those that prevent testing or make it very difficult
- Those that restrict functionality by subverting interception and scope
- Those that make integration difficult or impossible (the most egregious)

None of these is particularly unavoidable or even necessary, especially if you carefully consider them early in the life of your framework.

Let's start by looking at an egregious instance, the rigid configuration anti-pattern, which makes both testing and integration difficult.

10.2.1 Rigid configuration anti-patterns

Most frameworks provide some form of customization of behavior. This is really a major part of their appeal—for example, Hibernate allows you to use a variety of databases, connection pools, transaction strategies, and so on, taking the basic behavior of the library to the breadth of various use cases. Often this is done via the use of external *resource bundles* such as `.properties` or XML files. Most of this configuration is generally about specifying a certain amount or type of something (`max_connections=...`; or, `timeout=...`; or, `enable_logging=true`; and so on). So this kind of configuration is appropriate.

But sometimes libraries also provide *pluggable* services. They allow you to customize behavior by writing small components, usually adhering to a library interface, and plugging them in. Sometimes they are called plug-ins, other times extensions. But essentially the idea is the same—they are a user-provided dependency of the framework. It is when these plug-ins are configured that things often go wrong in framework integration. Many frameworks will use the same configuration mechanism (XML file or resource bundle) to specify the plug-in. Generally this, too, takes the form of a string name/value pair such as

```
extensibility.plugin=com.example.MyPlugin
```

The property `extensibility.plugin` identifies which plug-in component is being set. On the right side is the name of the user-provided class. On the surface this looks like a clean approach. It is concise, easy to understand, and specifies the needed extension nicely. On closer inspection it reveals several problems that make it both difficult and unwieldy to test and integrate.

TYPE UNSAFE CONFIGURATION ANTI-PATTERN

The gravest of these is probably the disregard for type-safety. Since resource bundles are stored as raw strings, there's no way to verify that the information is present in its appropriate form. You could easily misspell or mistype the configuration parameter or even leave it out completely.

OSCache is a framework that provides real-time caching services for applications. OSCache stores its configuration in a resource bundle named `oscache.properties`. This is loaded by the cache controller when it is started up and then configured. A sample configuration file for OSCache is shown in listing 10.2.

Listing 10.2 A sample `oscache.properties` configuration file

```
cache.memory=false
cache.use.host.domain.in.key=true
cache.path=/tmp

cache.persistence.class=com.opensymphony.oscache.plugins.diskpersistence.
➡ DiskPersistenceListener
```

```
cache.algorithm=com.opensymphony.oscache.base.algorithm.LRUCache
cache.blocking=false
cache.capacity=100
cache.unlimited.disk=false
```

In listing 10.2, we are shown a cache configuration that sets several routine options, such as whether or not to use a disk cache

```
cache.path=/tmp
```

and where to store the files for this disk cache, its capacity, interaction model, and so on:

```
cache.path=/tmp
...
cache.blocking=false
cache.capacity=100
cache.unlimited.disk=false
```

And then, there's one interesting line right in the middle that specifies the kind of caching service to use. In this case it's OSCache's own `DiskPersistenceListener`:

```
cache.persistence.class=com.opensymphony.oscache.plugins.diskpersistence.
➡ DiskPersistenceListener
```

The problem with this is immediately obvious—we have no knowledge when coding this property of whether OSCache supports a `DiskPersistenceListener`. It isn't checked by the compiler. A simple but all-too-common misspelling goes undetected and leaves you with a runtime failure:

```
cache.persistence.class=com.opensymphony.oscache.plugins.diskpersistence.
➡ DiscPersistenceListener
```

This is very similar to the issues with string identifiers (keys) we encountered in chapter 2. Furthermore, even if it is spelled correctly, things may not work as expected unless it implements the correct interface:

```
package com.example.oscache.plugins;

public class MyPersistenceListener {
    ...
}
```

`MyPersistenceListener` is a custom plug-in I've written to persist cache entries in a storage medium of my choice. We can configure OSCache by specifying the property thusly:

```
cache.persistence.class=com.example.oscache.plugins.MyPersistenceListener
```

This is incorrect because `MyPersistenceListener` doesn't implement `PersistenceListener` from OSCache. This error goes undetected until runtime, when the OSCache engine attempts to instantiate `MyPersistenceListener` and use it. The correct code would be

```
package com.example.oscache.plugins;

import com.opensymphony.oscache.base.persistence.PersistenceListener;
```

```
public class MyPersistenceListener implements PersistenceListener {
    ...
}
```

Now, you may be saying, this is only one location where things can go wrong and a bit of extra vigilance is all right. You could write an integration test that will detect the problem—a somewhat verbose but passable solution if used sparingly. But now consider what happens if you misspell the *left-hand* side of the same configuration property:

```
cache.persistance.class=com.example.oscache.plugins.MyPersistenceListener
```

(*Persistence* is spelled with an *e*.)

In this case, not only is there no compile-time check, but any sanity check will completely miss the error! This is because property `cache.persistance.class`, if not explicitly set, will automatically use a default value. Everything appears to be fine, and the configuration throws no errors even in an integration test because you’re freely allowed to set as many unknown properties as you like, with no regard to their relevance. Adding values for `cache.persistance.class`, `jokers.friend.is.batman`, and `my.neck.hurts` are all valid properties that a resource bundle won’t complain about.

This is a bad state of affairs, since even extra vigilance can let you down. It also leads programmers to resort to things like copying and pasting known working configurations from previous projects or from tutorials.

UNWARRANTED CONSTRAINTS ANTI-PATTERN

This kind of name/value property mapping leads to strange restrictions on user code, particularly, the plug-in code we have been studying—our plug-in is specified using a fully qualified class name:

```
cache.persistance.class=com.example.oscache.plugins.MyPersistenceListener
```

The implication here is that `MyPersistenceListener` must implement interface `PersistenceListener` (as we saw just earlier) but also that `MyPersistenceListener` must have a public nullary constructor that throws no *checked* exceptions:

```
package com.example.oscache.plugins;

import com.opensymphony.oscache.base.persistence;

public class MyPersistenceListener implements PersistenceListener {
    public MyPersistenceListener() {
    }
    ...
}
```

This necessity arises from the fact that `OSCache` uses *reflection* to instantiate the `MyPersistenceListener` class. Reflective code is used to read and manipulate objects whose types are not immediately known. In our case, there’s no source code in `OSCache` that’s aware of `MyPersistenceListener`, yet it must be usable by the original `OSCache` code in order for plug-ins to work. Using reflection, `OSCache` is able to construct instances of `MyPersistenceListener` (or any class named in the property) and use

them for persistence. Listing 10.3 shows an example of reflective code to create an instance of an unknown class.

Listing 10.3 Creating an object of an unknown class via reflection

```
String className = config.getProperty("cache.persistence.class");
Class<?> listener;
try {
    listener = Class.forName(className);
} catch (ClassNotFoundException e) {
    throw new RuntimeException("failed to find specified class",
        e);
}

Object instance;
try {
    instance = listener.getConstructor().newInstance();
} catch (InstantiationException e) {
    throw new RuntimeException("failed to instantiate listener",
        e);
} catch (IllegalAccessException e) {
    throw new RuntimeException("failed to instantiate listener",
        e);
} catch (InvocationTargetException e) {
    throw new RuntimeException("failed to instantiate listener",
        e);
} catch (NoSuchMethodException e) {
    throw new RuntimeException("failed to instantiate listener",
        e);
}
```

Let's break down this example. First, we need to obtain the name of the class from our configuration property:

```
String className = config.getProperty("cache.persistence.class");
```

This is done using a hypothetical config object, which returns values by name from a resource bundle. The string value of our plug-in class is then converted to a `java.lang.Class`, which gives us reflective access to the underlying type:

```
Class<?> listener;
try {
    listener = Class.forName(className);
} catch (ClassNotFoundException e) {
    throw new RuntimeException("failed to find specified class",
        e);
}
```

`Class.forName()` is a method that tries to locate and load the class by its fully qualified name. If it is not found in the classpath of the application, an exception is thrown and we terminate abnormally:

```

try {
    listener = Class.forName(className);
} catch (ClassNotFoundException e) {
    throw new RuntimeException("failed to find specified class",
        e);
}

```

Once the class is successfully loaded, we can try to create an instance of it by obtaining its nullary constructor and calling the method `newInstance()`:

```

Object instance;
try {
    instance = listener.getConstructor().newInstance();
} catch (InstantiationException e) {
    throw new RuntimeException("failed to instantiate listener ",
        e);
} catch (IllegalAccessException e) {
    throw new RuntimeException("failed to instantiate listener",
        e);
} catch (InvocationTargetException e) {
    throw new RuntimeException("failed to instantiate listener",
        e);
} catch (NoSuchMethodException e) {
    throw new RuntimeException("failed to instantiate listener",
        e);
}

```

There are four reasons why creating the instance may fail. And this is captured by the four catch clauses in the previous example:

- `InstantiationException`—The specified class was really an interface or abstract class.
- `IllegalAccessException`—Access visibility from the current method is insufficient to call the relevant constructor.
- `InvocationTargetException`—The constructor threw an exception before it completed.
- `NoSuchMethodException`—There is no nullary constructor on the given class.

If none of these four cases occurs, the plug-in can be created and used properly by the framework. Looked at from another point of view, these four exceptions are four restrictions on the design of plug-ins for extending the framework. In other words, these are four restrictions placed on your code if you want to integrate or extend `OSCache` (or any other library that uses this extensibility idiom):

- You must create and expose a concrete class with public visibility.
- This class must also have a public constructor.
- This class should not throw checked exceptions.
- It must have a nullary (zero-argument) constructor, which is public.

Apart from the restriction of not throwing checked exceptions, these seem to be fairly restrictive. As you saw in the chapter 9, in the section “Objects and design,” it’s quite desirable to hide implementation details in package-local or private access and expose only interfaces and abstract classes. As you’ve seen throughout this book, both immutability and testing suffer² when you’re unable to set dependencies via constructor, since you can’t declare them final and they’re hard to swap out with mocks.

Without serious contortions like the use of statics and/or the singleton anti-pattern, we are left with a plug-in that can’t benefit from DI. Not only does this spell bad weather for testing, but it also means we lose many other benefits such as scoping and interception.

CONSTRAINED LIFECYCLE ANTI-PATTERN

Without scoping or interception, your plug-in code loses much of the handy extra functionality provided by integration libraries like warp-persist and Spring. A persistence listener for cache entries can no longer take advantage of warp-persist’s `@Transactional` interceptor, which intercedes in storage code and wraps tasks inside a database transaction. You’re forced to write extra code to create and wrap data actions inside a managed database transaction. This similarly applies to other AOP concerns that we encountered in chapter 8, such as security and execution tracing.

Security, transaction, and logging code can no longer be controlled from one location (the interceptor) with configurable matchers and pointcuts. This leads to a fragmentation of crosscutting code, which adds to the maintenance overhead of your architecture.

Scoping is similarly plagued since the one created instance of your plug-in is automatically a singleton. Services that provide contextual database interactivity can no longer be used, since they are dependencies created and managed by an injector that’s unavailable to plug-in code. Data actions we perform routinely, like opening a session to the database and storing or retrieving data around that session, cannot be performed directly. This also means that sessions cannot be scoped around individual HTTP requests.

These are all unwarranted constraints placed on user code by a rigid configuration system.

Similarly, a class of anti-patterns that I classify as black box design makes integration and testing extremely difficult. I examine some of these in the following section.

10.2.2 Black box anti-patterns

Another commonly seen symptom of poor design in integration or framework interoperability is the tendency toward *black box* systems. A black box system is something that *completely* hides how it works to the detriment of collaborators. This is not to be confused with the beneficial practice of encapsulation, which involves hiding implementation details to prevent accidental coupling.

² For a thorough examination of this, see chapter 4’s investigation of testability and chapter 5’s section “The singleton anti-pattern.”

Black box systems don't so much hide the specific logic of their behaviors as they hide the mechanism by which they operate. Think of this as a runtime analog rigid configuration anti-pattern. Black box anti-patterns allow you to test in very limited ways, and they prevent the use of certain design patterns and practices we've examined by the excessive use of static fields, or abstract base-class functionality.

FRAGILE BASE CLASS ANTI-PATTERN

Many programmers are taught to use inheritance as a means of reusing code. Why rewrite all this great code you've done before? This is a noble enough idea in principle. When applied using class inheritance, it can lead to odd and often confounding problems. The first of these problems is the fragile base class. When you create a subclass to share functionality, you're creating a tight coupling between the new functionality and the base class. The more times you do this, the more tight couplings there are to the base class.

If you're the world's first perfect programmer, this isn't an issue. You would write the perfect base class that never needed to change, even if requirements did, and it would contain all the possible code necessary for its subclasses' evolution. Never mind that it would be enormous and look horrible.

But if you're like everyone else, changes are coming. And they are likely to hurt a lot. Changing functionality in the base class necessitates changing the behavior of all classes that extend it. Furthermore, it's very difficult to always design a correctly extensible base class. Consider the simple case of a queue based on a array-backed list:

```
public class Queue<I> extends ArrayList<I> {
    private int end = -1;
    public void enqueue(I item) {
        super.add(0, item);
        end++;
    }
    public I dequeue() {
        if (end == -1) return null;
        end--;
        return super.get(end + 1);
    }
}
```

Start with an empty queue

Return null if queue is empty

This is a pretty simple class. Queue uses ArrayList to represent items it stores. When an item is enqueued (added to the back of the list), it is inserted at index 0 in the ArrayList:

```
public void enqueue(I item) {
    super.add(0, item);
    end++;
}
```

We increment the end index, so we know what the length of the queue is currently. Then when an item is removed from the queue, we decrement the end index and return it:

```

public I dequeue() {
    if (end == -1) return null;

    end--;
    return super.get(end + 1);
}

```

We need to return the item at `(end + 1)` because this is the last item in the list (pointed at by `end`, before we decremented it). So far, so good; we saved ourselves a lot of array insertion and retrieval code, but now look at what happens when we start using this class in unintended ways:

```

Queue<String> q = new Queue<String>();
q.enqueue("Muse");
q.enqueue("Pearl Jam");
q.clear();
q.enqueue("Nirvana");

System.out.println(q.dequeue());

```

We expect this program to print out "Nirvana" and leave the queue in an empty state. But really it throws an `IndexOutOfBoundsException`. Why? It's the fragile base class problem—we've used a method directly from the base class, `ArrayList`, which clears every element in the queue. This seemed natural enough at the time. However, since `Queue` doesn't know about method `clear()`, except via its base class, it doesn't update index `end` when the list is emptied. When `dequeue()` is called, `end` points at index 2 since `enqueue()` has been called three times. In reality, `end` should be pointing at index 0 since there's only one item in the queue.

This has left the queue in a corrupt state, even though the underlying `ArrayList` is functioning as expected. How do we fix this? We could override method `clear()` and have it reset the `end` index:

```

public class Queue<I> extends ArrayList<I> {
    private int end = -1;

    public void enqueue(I item) {
        super.add(0, item);
        end++;
    }

    public I dequeue() {
        if (end == -1) return null;

        end--;
        return super.get(end + 1);
    }

    @Override
    public void clear() {
        end = -1;

        super.clear();
    }
}

```

That's certainly an option, but what about other methods that `ArrayList` provides? Should we override everything? If we do, we're certainly assured that the behavior will be correct, but then we've gained nothing from the inheritance. In fact, we've probably overridden a few methods that have no place in a `Queue` data structure anyway (such as `get(int index)`, which fetches entries by index). We'd have been much better off making the `ArrayList` a dependency of `Queue` and using only those parts of it that we require, as shown in listing 10.4.

Listing 10.4 Replacing inheritance with delegation in class `Queue`

```
public class Queue<I> {
    private int end = -1;
    private final ArrayList<I> list = new ArrayList<I>();

    public void enqueue(I item) {
        list.add(0, item);
        end++;
    }

    public I dequeue() {
        if (end == -1) return null;
        end--;
        return list.get(end + 1);
    }
}
```

Listing 10.4 shows how replacing inheritance with delegation helps solve the fragile base class problem with a much more resilient design pattern. Now we can easily add functionality to the `Queue` with no danger of unintentionally leaking `ArrayList` functionality:

```
public class Queue<I> {
    private int end = -1;
    private final ArrayList<I> list;

    public Queue(ArrayList<I> list) {
        this.list = list;
    }

    public void enqueue(I item) {
        list.add(0, item);
        end++;
    }

    public I dequeue() {
        if (end == -1) return null;
        end--;
        return list.get(end + 1);
    }

    public void clear() {
        end = -1;
        list.clear();
    }
}
```

And now we can use it safely:

```
Queue<String> q = new Queue<String>(new ArrayList<String>());
q.enqueue("Muse");
q.enqueue("Pearl Jam");
q.clear();
q.enqueue("Nirvana");

System.out.println(q.dequeue());
assert null == q.dequeue();
```

This code correctly prints out "Nirvana" and leaves the queue empty, and it's a more satisfying solution.

Several open source frameworks make liberal use of abstract base classes to share functionality with little or no regard for the abstract base class problem. Another problem with base classes for sharing code is that they cannot be replaced with a mock object. This makes testing them tricky, especially when you have more than one level of inheritance. The Apache CXF framework has some classes that sport four and five levels of inheritance. Isolating problems can be very difficult in such cases.

While these problems are pretty rough, they're not all insurmountable. Good design can overcome them and lead to the same amount of flexibility with no loss of testing or integration capability. In the following section we'll look at how programmatic configuration can make life easier.

10.3 *Programmatic configuration to the rescue*

The solution to rigid configuration and black box anti-patterns is surprisingly simple: programmatic configuration. Rather than shoehorn plug-in configuration via the rather lean path of resource bundles (.properties, XML, or flat text files), programmatic configuration takes the attitude that configuring plug-ins is the same as DI.

If you start to look at the plug-in as a dependency of the framework, then it becomes simple and natural to write integration between frameworks and injectors, between frameworks and tests, and indeed between various frameworks.

10.3.1 *Case study: JSR-303*

Bean Validation is the common name for an implementation of the JSR-303³ specification. Bean Validation is an initiative of the Java Community Process to create a flexible standard for validating data model objects via declarative constraints.

Listing 10.5 shows a typical data model object with annotations representing the declarative constraints acted on by the JSR-303 runtime.

Listing 10.5 A data model object representing a person

```
public class Person {
    @Length(max=150)
    private String name;
```

³ Find out more about JSR-303 at <http://jcp.org/en/jsr/detail?id=303>.

```
@After("1900-01-01")
private Date bornOn;

@email
private String email;

@Valid
private Address home;

public void setName(String name) {
    this.name = name;
}

public Date getBornOn() {
    return bornOn;
}

public void setBornOn(Date bornOn) {
    this.bornOn = bornOn;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public Address getHome() {
    return home;
}

public void setHome(Address home) {
    this.home = home;
}
}

public class Address {
    @NotNull
    @Length(max=200)
    private String line1;

    @Length(max=200)
    private String line2;

    @Zip(message="Zipcode must be five digits exactly")
    private String zipCode;

    @NotNull
    private String city;

    @NotNull
    private String country;

    public String getLine1() {
        return line1;
    }

    public void setLine1(String line1) {
        this.line1 = line1;
    }
}
```



```

    }

    public String getLine2() {
        return line2;
    }

    public void setLine2(String line2) {
        this.line2 = line2;
    }

    public String getZipCode() {
        return zipCode;
    }

    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }
}

```

Listing 10.5 is of class `Person`, which represents a person in a hypothetical roster. Each of `Person`'s fields is tagged with an annotation representing a constraint to be applied on instances of the object. Some have an optional message attribute, which can be used to customize the error message reported on validation failure:

```

@Zip(message="Zipcode must be five digits exactly")
private String zipCode;

```

If all the constraints pass, then the instance is considered valid:

```

Person lincoln = new Person();
lincoln.setName("Abraham Lincoln");
lincoln.setBornOn(new Date());
lincoln.setEmail("abe@lincoln.nowhere");

Address address = new Address();
address.setLine1("1600 Pennsylvania Ave");
address.setZipCode("51245");
address.setCity("Washington, D.C.");
address.setCountry("USA");

lincoln.setHome(address);

List<InvalidValue> errors = validator.validate(lincoln);
System.out.println(String.format("There were %d error(s)", errors.size()));

```

On running this code, you'll see an output of how many errors there were (there were none). Each of the constraints on the code is a custom annotation that we've made up for the purposes of this demonstration. The constraints to which these annotations are attached are determined by the annotation declaration itself. Listing 10.6 illustrates some of the annotations we've used in listing 10.5's example of `Person` and `Address` data classes.

Listing 10.6 Custom annotations that act as constraint-plugin in "configurators"

```
import java.lang.reflect.ElementType
import java.lang.reflect.RetentionPolicy

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@ConstraintValidator(ZipConstraint.class)
public @interface Zip {
    String message() default "Zip invalid";
}

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@ConstraintValidator(LengthConstraint.class)
public @interface Length {
    int min();
    int max();
}

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@ConstraintValidator(NotNullConstraint.class)
public @interface NotNull {
    String message() default "Cannot be null";
}
```

The immediate thing that is apparent from listing 10.6 is that each of these annotations refers to a real validator. In the case of `@Zip`, it refers to class `ZipConstraint`:

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@ConstraintValidator(ZipConstraint.class)
public @interface Zip {
    String message() default "Zip invalid";
}
```

Similarly, `@Length`, which validates that fields have a minimum and maximum length, is bound to the `LengthConstraint` class:

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@ConstraintValidator(LengthConstraint.class)
public @interface Length {
    int min();
    int max();
}
```

Essentially, this is JSR-303's plug-in configuration mechanism. You use the class name in a *meta-annotation*, to specify which validator plug-in it should use. This is a type-safe meta-annotation that prevents you from registering anything but a subtype of `java.beans.validation.Constraint`, which the validation framework expects.

This is simple and yet quite powerful. We mitigate the problems of misspelling the class name and property name and of specifying the wrong plug-in type all in one stroke. Furthermore, JSR-303 doesn't mandate the use of reflection to create instances of user plug-ins. Unlike the recalcitrants we encountered earlier in this chapter, JSR-303 doesn't have a rigid configuration mechanism and so places no restrictions on your plug-in code.

This is achieved by providing the runtime with a user-written `ConstraintFactory`. This is a simple interface that JSR-303 runtimes use to obtain plug-in instances from your code:

```
/**
 * This class manages the creation of constraint validators.
 */
public interface ConstraintFactory {
    /**
     * Instantiate a Constraint.
     *
     * @return Returns a new Constraint instance
     * The ConstraintFactory is <b>not</b> responsible for calling
     * Constraint#initialize
     */
    <T extends Constraint> T getInstance(Class<T> constraintClass);
}
```

To take advantage of a dependency injector, you simply register an instance of `ConstraintFactory` that obtains `Constraints` from the injector. Here's a very simple Guice-based implementation that creates an injector and uses it to produce constraints when called on by the JSR-303 runtime:

```
public class GuiceConstraintFactory implements ConstraintFactory {
    private final Injector injector = Guice.createInjector(new MyModule());

    public <T extends Constraint> T getInstance(Class<T> constraintKey) {
        return injector.getInstance(constraintKey);
    }
}
```

This allows you to create constraint plug-ins with their own dependencies and lifecycle, and they can be capable of interception if needed. Let's take a look at an example of the `LengthConstraint`, which was tied to the `@Length` annotation, ensuring a maximum length for strings in listing 10.7.

Listing 10.7 A sample `LengthConstraint` plug-in for checking a string's length

```
/**
 * Check that a string length is between min and max
 *
```

```

*/
public class LengthConstraint implements Constraint<Length> {
    private int min;
    private int max;

    private final StringTools tools;

    @Inject
    public LengthConstraint(StringTools tools) {
        this.tools = tools;
    }

    /**
     * Configure the constraint validator based on the elements
     * specified at the time it was defined.
     *
     * @param constraint the constraint definition
     */
    public void initialize(Length constraint) {
        min = constraint.min();
        max = constraint.max();
    }

    /**
     * Validate a specified value.
     * returns false if the specified value does not conform to the
     * definition
     * @exception IllegalArgumentException if the object is not of type
     * String
     */
    public boolean isValid(Object value) {
        if (value == null) return true;

        if ( !(value instanceof String) )
            throw new IllegalArgumentException("Expected String type");

        String string = (String) value;
        return tools.isLengthBetween(min, max, string);
    }
}

```

Inject a string checking utility

Use dependency to check length

In listing 10.7, LengthConstraint is dependency injected with StringTools, a reusable utility that does the dirty work of checking string length for us. We're able to take advantage of Guice's constructor injection because of the flexibility that JSR-303's ConstraintFactory affords us:

```

public class LengthConstraint implements Constraint<Length> {
    private int min;
    private int max;

    private final StringTools tools;

    @Inject
    public LengthConstraint(StringTools tools) {
        this.tools = tools;
    }

    ...
}

```

LengthConstraint dutifully ensures that it's working with a String before passing this on to its dependency to do the hard yards. LengthConstraint is itself configured using an interface method `initialize()` that provides it with the relevant annotation instance:

```
/**
 * Configure the constraint validator based on the elements
 * specified at the time it was defined.
 *
 * @param constraint the constraint definition
 */
public void initialize(Length constraint) {
    min = constraint.min();
    max = constraint.max();
}
```

Recall that we set this value in the Person class for field name to be between 0 and 150 (see listing 10.5 for the complete Person class):

```
public class Person {
    @Length(max=150)
    private String name;

    ...
}
```

Now, every time the validator runs over an instance of Person,

```
Person lincoln = new Person();
lincoln.setName("Abraham Lincoln");
...

Address address = new Address();
...
lincoln.setHome(address);

List<InvalidValue> errors = validator.validate(lincoln);
```

the LengthConstraint plug-in we wrote will be run. Being flexible in providing us an extension point with the ConstraintFactory and the @ConstraintValidator meta-annotation, JSR-303 rids itself of all the nasty perils of rigid configuration internals. It encourages user code that's easy to test, read, and maintain. And it allows for maximum reuse of service dependencies without any of the encumbering ill effects of static singletons or black box systems.

JSR-303 is an excellent lesson for framework designers who are looking to make their libraries easy to integrate and elegant to work with.

10.4 Summary

In this chapter we looked at the burgeoning problem of integration with third-party frameworks and libraries. Most of the work in software engineering today is done via the use of powerful third-party frameworks, many of which are designed and provided by open source communities such as the Apache Software Foundation or OpenSymphony.

Other random indispensable pieces of advice

“When wrestling for possession of a sword, the man with the handle always wins.”

—Neal Stephenson (in *Snow Crash*)

“Technical people are better off not looking at patents. If somebody sues you, you change the algorithm or you just hire a hit-man to whack the stupid git.”

—Linus Torvalds

“One should always play fairly when one has the winning cards.”

—Oscar Wilde

Many companies (like Google and ThoughtWorks) are releasing commercially developed libraries as open source for the good of the community at large.

Leveraging these frameworks within an application is an often tedious and difficult task. They’re generally designed with a specific type of usage in mind, and integrating them is often a matter of some complexity. The better-designed frameworks provide simple, flexible, and type-safe configuration options that allow you to interact with the framework as you would with any one of your services. Extending such frameworks is typically done via the use of pluggable user code that conforms to a standard interface shipped along with the library. Frameworks that sport rigid configuration such as via the use of resource bundles (`.properties`, XML, or flat text files) cause problems because they place undue restrictions on the classes written for plug-in extensions. Because they are simple strings in a text file, these class names are also prone to misspelling and improper typing and are forced to be publicly accessible.

Moreover, since they’re instantiated using reflection, these plug-in classes must have at least one public constructor that accepts no arguments (a *nullary* constructor). This immediately places unwarranted constraints on your code—you cannot declare fields `final` in your plug-in class. Not only does this have grave consequences for visibility, but it also makes your classes hard to test and swap out with mocked dependencies. The restriction on who creates your plug-in class also means that your code cannot take advantage of any of the other benefits of dependency injection. Lifecycle, scoping, and AOP interception must all be given up. This is particularly egregious in a scenario where your plug-in uses the same kind of services that your application does, for instance, a database-backed persistence system. Transactions, database connectivity, and security must all be managed independently from the application by your plug-in code.

The alternative is to share services via the use of static state and the singleton anti-pattern. This has negative consequences for testing, as you saw in chapter 5.

A well-designed framework, on the other hand, is cognizant of the testability mantra and allows you to configure plug-ins programmatically; that is, within the application itself. JSR-303 is a validation framework that allows you to create constraint plug-ins that are tied to declarative constraints placed on data model objects. These plug-ins are

applied universally based on annotation metadata and help specify the validity of a data object. JSR-303's constraint plug-ins are created and configured via the use of a custom `ConstraintFactory` interface. `ConstraintFactory` is a plug-in factory that you provide, which creates instances of the actual constraint plug-ins for JSR-303's runtime.

Programmatic configuration means that your plug-in configuration code is type-safe, contract-safe, and well-defined within the bounds of the framework's usage parameters. It also means that you can control the creation and wiring of your own code, giving it dependencies, scoping, and rigorous testability. This makes it open to all the great benefits of dependency injection that you've so come to love!

In the next chapter you'll apply many of the patterns you've learned up to this point and create a fully functional, running demo application.

11

Dependency injection in action!

This chapter covers:

- Building a complete web application from scratch
- Seeing DI principles in a working application
- Applying useful design patterns
- Solving many common problems

“A computer lets you make more mistakes faster than any invention in human history—with the possible exceptions of handguns and tequila.”

—Mitch Radcliffe

In this chapter we’ll put all of the things you’ve learned so far in this book into practice in a real, working application. While there probably aren’t enough pages to demonstrate a large enterprise application, we’ll look at a succinct application that gives us all the salient features of a broader program.

This program will start from bootstrapping the injector in a web application and walk through designing services for persistence, transactions, and user interactivity. We’ll also show how to apply some useful design patterns (such as the

Provider pattern from chapter 5) and ways to properly leverage scope, lifecycle, and interception (AOP).

Let's start by defining what this program will be.

11.1 Crosstalk: a Twitter clone!

I've chosen to do a simple, trivial clone of the popular microblogging service Twitter. (This isn't in the least bit because I spend large chunks of my day on Twitter!) While I was writing this chapter, I thought *crosstalk* was cool code name.

Crosstalk serves as a good illustration of a straightforward yet vastly scalable website that has plenty of user interactivity and data processing needs. These make it an ideal problem domain to showcase the wonderful patterns of dependency injection.

Its requirements are diverse—ranging from persistence and presentation to security and user management. This gives us a good basis for drawing an illustrative architecture from start to finish, in a few pages. First, let's look at these requirements.

11.1.1 Crosstalk's requirements

Our thesis is fairly straightforward: crosstalk allows users to write short text messages (called tweets) on their home page. This engenders some broad requirements. We need to

- Authenticate users when they log on
- Secure their home pages
- Persist tweets in a database

We also need to ensure that the site is scalable and concurrent. In other words, it must be easily tolerant to many users accessing their home pages at once. Given all these requirements, it follows that crosstalk has several layers:

- The presentation layer (website)
- The persistence layer (database mechanism)
- The security layer (authentication and securing pages)

We'll build this application using Google Guice as the injector and Google Sitebricks as the web application framework. Google Sitebricks is a simple, statically typed development system for rendering web pages. It follows the REST idiom, which makes it ideal for building HTML websites.

For persistence, we'll use Hibernate and warp-persist (seen in chapter 8) as the integration bridge with Guice. Let's get started.

11.2 Setting up the application

First, we'll need a layout and structure for the crosstalk application. This needs to follow the Java Servlet standard so that it can be deployed in a servlet container like Apache Tomcat or Mort Bay Jetty. We'll use Jetty for this illustration, since it's very simple to get up and running with. And as the database, we'll go with Hypersonic (sometimes called HSQL), which is an in-memory SQL database that's easy to set up and configure.

Here's the structure we'll start out with for the project (see figure 11.1):

- `src`—The main source directory, with all our classes
- `test`—A test sources directory, for our Jetty launcher
- `web`—The directory for all the web resources (HTML templates, CSS stylesheets)
- `web/WEB-INF`—A directory required by the servlet container to look for deployment descriptors

The file `web.xml` is a deployment descriptor that instructs the servlet container about how to configure this web application. We'll use a `web.xml` file very similar to the one used in chapter 5, when we were dealing with web scopes.

Once you have that layout set up (see figure 11.1), open a project in your favorite IDE and add the libraries shown in table 11.1 to the classpath:

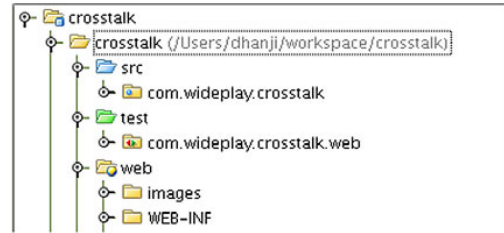


Figure 11.1 The layout of directories for the crosstalk application

Table 11.1 Libraries

Library	Description
guice-2.0.jar	Google Guice core http://code.google.com/p/google-guice/
aopalliance.jar	Guice AOP interface (included with Guice)
google-sitebricks.jar	Google's Sitebricks web framework http://code.google.com/p/google-sitebricks/
guice-servlet-2.0.jar	Servlet integration for Guice (included with Guice)
hibernate3.jar	Hibernate core persistence framework https://www.hibernate.org/
hibernate-annotations.jar	Annotations for Hibernate https://www.hibernate.org/
ejb3-persistence.jar	Annotations for Hibernate (included with Hibernate Annotations)
dom4j-1.6.1.jar	Dom4j XML parser (included with Hibernate)
jta.jar	Java Transaction API (included with Hibernate)
cglib-nodep-2.1_3.jar	CGLib used by Hibernate to proxy objects
antlr-2.7.5h3.jar	Antlr compiler library (included with Hibernate)
commons-collections.jar	Apache Collections library (included with Hibernate)
commons-logging.jar	Apache Commons-Logging (required by Hibernate for log output)
warp-persist-2.0-20090214.jar	Warp-persist integrates Hibernate with Guice http://www.wideplay.com/guicewebextensions2

Table 11.1 Libraries (*continued*)

Library	Description
hsqldb.jar	Hypersonic In-Memory SQL database http://hsqldb.org/
servlet-api-2.5-6.1.9.jar	Java Servlet API (included with Jetty)
jetty-6.1.9.jar	Mort Bay Jetty http://www.mortbay.org/jetty
jetty-util-6.1.9.jar	Utilities for working with Jetty
jcip-annotations.jar	The set of annotations used in chapter 10 for thread-safety documentation http://www.javaconcurrencyinpractice.com/

TIP To add these libraries to the classpath in IntelliJ IDEA, open Settings > Project Settings and select the main (crosstalk) module. Then open the Dependencies tab and click Add. Now select Project Library and Add Jar Directory, choosing the directory where all these jars reside (or add them individually using the single-entry module library option).

Once you have all of those lined up, you should be good to go. If you aren't using an IDE, you can place all the jars in a lib directory and specify them individually on the command line.

Now the central configuration for our application will reside in `CrosstalkBootstrap`. This is a simple Java class, a subclass of `GuiceServletContextListener`, from which we'll tell Guice how to wire our services together, as shown in listing 11.1.

Listing 11.1 The Guice configuration for crosstalk (also creates the injector)

```
public final class CrosstalkBootstrap extends GuiceServletContextListener {
    @Override
    protected Injector getInjector() {
        //bind in all of our service dependencies
        final Module services = new ServicesModule();

        //tell Sitebricks to scan this package
        final Module sitebricks = new SitebricksModule() {
            protected void configureSitebricks() {
                scan(CrosstalkBootstrap.class.getPackage());
            }
        };

        //map all incoming requests through the PersistenceFilter
        final Module servlets = new ServletModule() {
            protected void configureServlets() {
                filter("/*").through(PersistenceFilter.class);

                install(sitebricks);
            }
        };
    }
}
```

```

        //finally, create the injector with all our configuration
        return Guice.createInjector(services, servlets);
    }
}

```

It looks pretty simple, but let's examine what this does. By subclassing `GuiceServletContextListener`, we're able to create and register our own injector with `guice-servlet` and the servlet container itself. This listener must also be registered in `web.xml` so that it can be called when the application is deployed and made ready to run, as shown in listing 11.2.

Listing 11.2 Web.xml config for guice-servlet and a crosstalk-specific listener

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  version="2.5">

  <filter>
    <filter-name>guiceFilter</filter-name>
    <filter-class>com.google.inject.servlet.GuiceFilter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>guiceFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <listener>
    <listener-class>com.wideplay.crosstalk.CrosstalkBootstrap</listener-
    class>
  </listener>

</web-app>

```

`Web.xml` also creates a filter mapping so that all incoming requests are passed through the `guice-servlet` `GuiceFilter`. This allows `guice-servlet` to reroute requests through the filters and servlets configured in Guice's servlet pipeline, allowing us to take advantage of dependency injection idioms, lifecycle, scope and interception. We saw a setup very similar to this in chapter 5. `GuiceFilter` is provided by `guice-servlet` and requires no special treatment. We map it to `/*` to indicate that all URLs are to be filtered through it:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  version="2.5">

  <filter>
    <filter-name>guiceFilter</filter-name>
    <filter-class>com.google.inject.servlet.GuiceFilter</filter-class>

```

```

</filter>
<filter-mapping>
  <filter-name>guiceFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
...
</web-app>

```

This way, all incoming requests are rerouted through guice-servlet's `GuiceFilter`, allowing it to process requests that are specific to crosstalk and pass through requests for static resources like CSS stylesheets and images. Application-specific requests are handled by Google Sitebricks pages, which we'll now go about setting up.

11.3 Configuring Google Sitebricks

Since we'll be using the Google Sitebricks web framework, we'll need to do extra configuration to tell it how to behave. Google Sitebricks has a simple philosophy where individual HTML pages are modeled as Java classes with a backing HTML template. Each such page class is registered against URL patterns in much the same manner as a servlet (or filter). We'll see how this works in a second. But first, let's tell guice-servlet to route all requests through Google Sitebricks, as shown in listing 11.3.

Listing 11.3 **CrosstalkBootstrap** from listing 11.1

```

public final class CrosstalkBootstrap extends
➤ GuiceServletContextListener {
    @Override
    protected Injector getInjector() {
        ...

        //map all incoming requests through PersistenceFilter
        final Module servlets = new ServletModule() {
            protected void configureServlets() {
                filter("/*").through(PersistenceFilter.class);

                install(sitebricks);
            }
        };

        //finally, create the injector with all our configuration
        return Guice.createInjector(services, servlets);
    }
}

```

The line in bold directs Guice to install our web framework, meaning that Google Sitebricks gets a crack at all user requests and can decide which ones to process and which ones it can safely pass on to the servlet container (such as static resources).

Now, we can proceed to look at the Google Sitebricks configuration:

```

public final class CrosstalkBootstrap extends
➤ GuiceServletContextListener {

```

```

@Override
protected Injector getInjector() {
...
    //tell sitebricks to scan this package
    final Module sitebricks = new SitebricksModule() {
        @Override
        protected void configureSitebricks() {
            scan(CrosstalkBootstrap.class.getPackage());
        }
    };
    ...
}

```

The method `scan(...)` tells Google Sitebricks which packages to work with. You may specify as many packages as needed this way.

By providing it with `CrosstalkBootstrap`'s package, we tell Google Sitebricks to scan the entire package tree beginning with `com.wideplay.crosstalk` and to look for page classes and templates to serve.

Finally, we install one more module in our injector, which will handle all application services. This is, appropriately, the `ServicesModule`:

```

public final class CrosstalkBootstrap extends
    ➡ GuiceServletContextListener {

    @Override
    protected Injector getInjector() {

        //bind in all of our service dependencies
        final Module services = new ServicesModule();

        ...

        //finally, create the injector with all our configuration
        return Guice.createInjector(services, servlets);
    }
}

```

The `Guice.createInjector()` method is a *varargs* method, meaning that it takes any number of arguments. This is convenient for us as we can pass in our two modules, `services` and `servlets`, that configure our persistence and security, respectively. Note that Google Sitebricks web pages are processed by the `SitebricksModule`. We'll look at the `services` module in some more detail shortly, but first, let's examine the application's package structure.

11.4 Crosstalk's modularity and service coupling

In this section, we'll look at how `crosstalk` is broken down into modules by package, service type, and contract. We'll see some of the concepts presented in chapter 4 on modularizing code and in the chapter 10 on packaging clean designs that separate areas of concern.

Crosstalk’s packages also expose no implementation details, so that accidental coupling of services does not occur. All collaboration between modules happens through publicly exposed interfaces, with well-defined contracts. Let’s see how this structure is achieved.

Crosstalk’s packages are organized very simply (shown in figure 11.2).

They are divided among the core application services, Google Sitebricks web pages, and any data model classes:

- `com.wideplay.crosstalk.web`—The presentation layer (all page classes go here)
- `com.wideplay.crosstalk.services`—Persistence and security layer
- `com.wideplay.crosstalk.tweets`—The domain model package

Notice that the `services` package exposes only interfaces (apart from the Guice `ServicesModule` configuration class). In chapter 4 we discussed the concept of loose coupling. This allows us to modify specific implementation details without affecting the presentation layer in any way. For example, we may choose to persist the data in a cluster data store rather than Hibernate and HSQL. By changing the implementation of classes in the `services` package, we can do this with no impact to the rest of the application. All implementation classes are hidden away as package-local. Figure 11.3 provides a complete picture of this approach.

You may also notice that the web pages and the data model classes are public. This is because they are used directly by the framework (Google Sitebricks in the web pages’ case and Hibernate in the data model’s) and configured using flexible annotations, which already alleviate any coupling perils.

Now that we’ve taken a bird’s-eye tour of the structure, let’s get down to it. First, the business end of crosstalk, its user-facing presentation layer.

11.5 The presentation layer

Let’s look at some of crosstalk’s functionality. Crosstalk’s core requirement is the ability to post tweets on a user’s home page. For this, we create a class `HomePage`, with a corresponding template in the `web/ resources` directory. `HomePage` looks like figure 11.4.

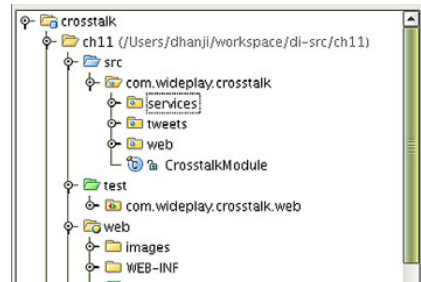


Figure 11.2 The layout of packages in the crosstalk application’s `src` directory

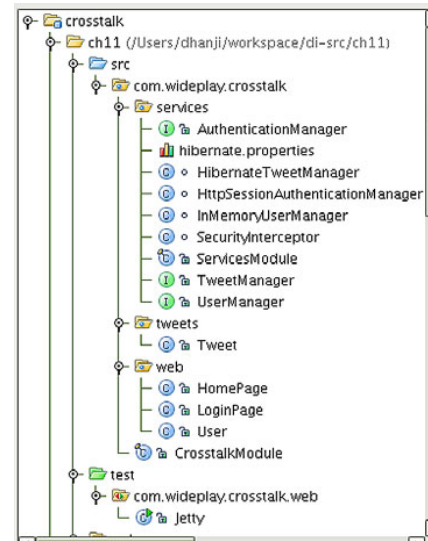


Figure 11.3 All the classes and packages in the eventual crosstalk application

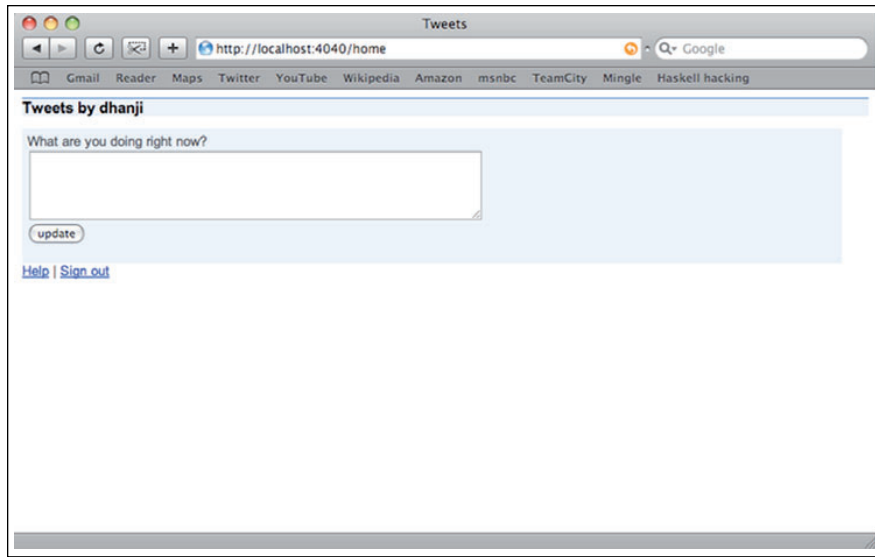


Figure 11.4 A user's home page with no content—it allows users to blog new tweets in a text box.

The Java source code behind `HomePage` is shown in listing 11.4.

Listing 11.4 `HomePage` Sitebricks page class models a crosstalk user's home page

```
package com.wideplay.crosstalk.web;

@At("/home") @Select("action") @RequestScoped
public class HomePage {
    //user context, tracks current user
    private User user;

    //page state variables
    private List<Tweet> tweets;
    private Tweet newTweet = new Tweet();

    //service dependencies
    private final TweetManager tweetManager;

    @Inject
    public HomePage(TweetManager tweetManager, User user) {
        this.tweetManager = tweetManager;
        this.user = user;
    }

    @Get("logout")
    public String logout() {
        user.logout();

        return "/login?message=Bye.";
    }

    @Get
    public String get() {
        //load tweets for current user
    }
}
```



```

        this.tweets = tweetManager.tweetsFor(user.getUsername());

        //stay on current page
        return null;
    }

    @Post
    public String post() {
        newTweet.setAuthor(user.getUsername());

        //contents are in newTweet, add it to the data store
        tweetManager.addTweet(newTweet);

        //redirect back to this page using a GET
        return "/home";
    }

    //getters/setters...

    public String getUser() {
        return user.getUsername();
    }

    public List<Tweet> getTweets() {
        return tweets;
    }

    public Tweet getNewTweet() {
        return newTweet;
    }
}

```

This looks pretty involved. Let's see what each bit actually means. First, we annotate the class with the `@At` annotation, which tells Google Sitebricks to serve this page at the given URI:

```

@At("/home") @On("action") @RequestScoped
public class HomePage { .. }

```

Now, whenever a user types in URL `http://localhost:8080/home` (assuming the application is deployed at `localhost:8080`), the `HomePage` will be served up by Google Sitebricks. We also see the familiar `@RequestScoped` annotation, which tells Guice to create a new instance of `HomePage` for every incoming request. This is important because it allows us to work with state that's specific to a user without stepping on other users' data, accidentally.

Finally, the `@On` annotation is used to resolve events to fire against. For now, let it suffice to say that `@On` controls which method annotated with `@Get` is called based on an incoming request parameter named `action`.

The visual companion to this page class is its HTML template. This template contains what a user will see. Let's see how it's built.

11.5.1 *The HomePage template*

`HomePage`'s template is a very simple HTML file. It contains dynamic text, which is provided by binding its data to the request-scoped instance of class `HomePage`, as shown in listing 11.5. It also contains a link to a stylesheet.

Listing 11.5 An HTML template that displays tweets, backed by HomePage

```

<html>
<head>
  <title>Tweets</title>

  <link rel="stylesheet" href="/crosstalk.css" />
</head>
<body>
  <h2>Tweets by ${user}</h2>
  <div class="box">
    <div class="box-content">
      <div>What are you doing right now?</div>

      <form action="/home" method="post">
        <textarea name="newTweet.text" rows="5" cols="60" />
        <input type="submit" value="update"/>
      </form>
    </div>
  </div>
  @Repeat(items=tweets, var="tweet")
  <div class="box">
    <div class="box-content">
      ${tweet.text} (${tweet.createdOn})
    </div>
  </div>

  <a href="http://manning.com/prasanna">Help</a> |
  <a href="?action=logout">Sign out</a>
</body>
</html>

```

The input box
for new tweets

This div is repeated
for every tweet

This line in particular is interesting because it resolves to a property of the `HomePage` class (that is, one that is accessed via a getter method):

```
<h2>Tweets by ${user}</h2>
```

This heading is dynamically evaluated at runtime by calling `HomePage.getUser()`. `HomePage.getUser()` returns the username of the current user logged in to crosstalk:

```

public String getUser() {
    return user.getUsername();
}

```

One other interesting part of the template is the `@Repeat` annotation:

```

@Repeat(items=tweets, var="tweet")
<div class="box">
  <div class="box-content">
    ${tweet.text} (${tweet.createdOn})
  </div>
</div>

```

This tells Google Sitebricks to repeat the annotated tag (in this case, `<div class="box">`) over all the items in `HomePage`'s property `tweets`. For each element in `tweets`, Google Sitebricks will render out the `div` tags expanding `tweet.text` and

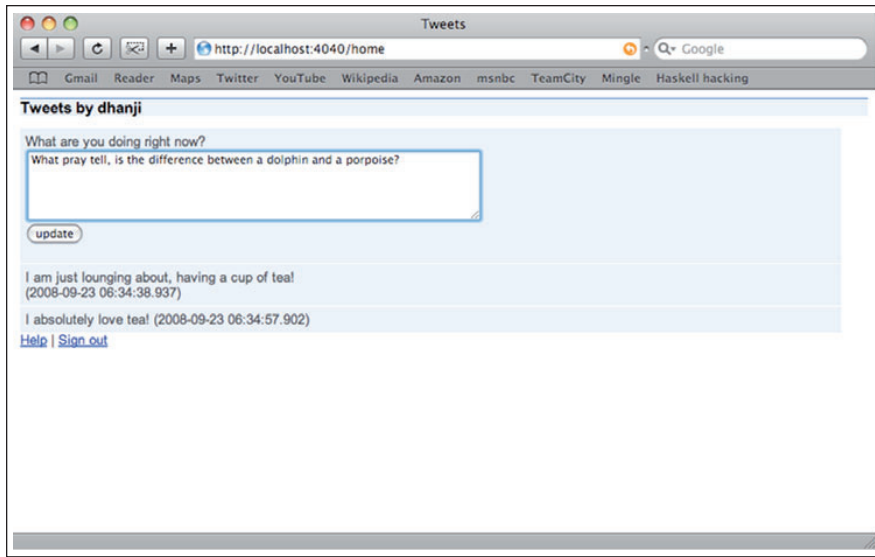


Figure 11.5 A user's home page with blogged content

`tweet.createdOn` properties. In other words, this renders out the list of tweets posted by a user on their home page (see figure 11.5).

The other significant part of this template is the form. Users type their new tweets into a text field and post them by submitting the form:

```
<div>What are you doing right now?</div>

<form action="/home" method="post">
  <textarea name="newTweet.text" rows="5" cols="60" />
  <input type="submit" value="update" />
</form>
```

Since the form's textarea is bound to `newTweet.text`, Google Sitebricks will attempt to write the incoming form post to `HomePage`'s property `newTweet`:

```
@At("/home") @On("action") @RequestScoped
public class HomePage {
    ...
    //page state variables
    private List<Tweet> tweets;
    private Tweet newTweet = new Tweet();
    ...
}
```

This is accomplished by navigating the `getNewTweet()` method to the appropriate setter methods on class `Tweet`, in this case, `setText()`. We take care to create a new instance of `Tweet` every time `HomePage` is created, thus every time it is requested, ensuring that tweets are never overwritten by accident.

Each new tweet entry by a user is recorded and modeled as an instance of class `Tweet`.

11.5.2 The Tweet domain object

`Tweet` is the sole class in our domain model, and it consists of simple data fields describing a message, its creation date, and its author, as shown in listing 11.6.

Listing 11.6 `Tweet` models a tweet message for capturing data and storing it

```
package com.wideplay.crosstalk.tweets;

@Entity
public class Tweet {
    @Id @GeneratedValue
    private Long id;

    private String author;
    private String text;
    private Date createdOn;

    public Tweet () {
        this.createdOn = new Date();
    }

    public String getAuthor() {
        return author;
    }

    public String getText() {
        return text;
    }

    public Date getCreatedOn() {
        return createdOn;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public void setText(String text) {
        this.text = text;
    }

    ...
}
```

Timestamp new tweets with “now”

Getters/setters for each property

This class also contains a private field `id`, which is used by Hibernate to track tweets in the database. This field is annotated with some boilerplate, identifying it as a primary key that’s generated automatically:

```
@Id @GeneratedValue
private Long id;
```

The class itself is annotated with `@Entity` to indicate to Hibernate that this class is to be treated as a data entity.

NOTE Internally, `@Entity` helps Hibernate separate classes that are mapped to tables from those that may be embedded as additional columns in a larger database table.

All of `Tweet`'s properties have JavaBeans getter and setter methods as per the convention with Hibernate. So far we've seen how this models all of our data and keeps it organized. We need to go a step further and provide `equals()` and `hashCode()` methods to ensure that instances of `Tweet` behave properly in collections. This is also important for any class that Hibernate uses, since it will attempt to compare copies when optimizing for cached access.

`equals()` and `hashCode()` are quite straightforward, as shown in listing 11.7; they take all three data fields into account since together they form the identity of the `Tweet`.

Listing 11.7 `equals()` and `hashCode()` for class `Tweet`, to compare instances

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Tweet)) return false;

    Tweet that = (Tweet) o;

    return this.author.equals(that.author)
        && this.createdOn.equals(that.createdOn)
        && this.text.equals(that.text);
}

@Override
public int hashCode() {
    int result;

    result = (author != null ?
        ➤ author.hashCode() : 0);
    result = 31 * result + (text != null ? text.hashCode() : 0);
    result = 31 * result + (createdOn != null ?
        ➤ createdOn.hashCode() : 0);

    return result;
}
```

If all properties are equal, return true

Hash code is computed from properties and a large prime

The idea is that objects are considered equal if they have the same values for `author`, `text`, and `createdOn`. Put another way, it's impossible to have all three match for two independent tweets. If two are posted at the same time, they will be from different authors. And if they are from the same author, they will necessarily have different creation times.

11.5.3 Users and sessions

In order to track users across independent requests, we must use an HTTP session. However, rather than interact with it directly, `crossstalk` takes advantage of scoping. The class `User` serves as a context for the current user of the system. Wherever it's referenced

from, User points to the user whose HTTP session is currently active. In listing 11.8 we mark User as scoped to the HTTP session by using the @SessionScoped annotation.

Listing 11.8 Session-scoped User tracks information about the current user

```
@SessionScoped @NotThreadSafe
public class User {
    private String username;

    public String getUsername() {
        return username;
    }

    //logs in a user, by setting username to the current session
    public void login(String username) {
        this.username = username;
    }

    //logs out a user, by clearing username from session
    public void logout() {
        this.username = null;
    }
}
```

User does nothing more than store a username for the user who logs in and clears that name when the user logs out. Web page classes and services alike can check to find out whether someone is logged in and, if so, who is logged in by querying this service.

In HomePage, User is directly injected directly as a dependency:

```
@At("/home") @Select("action") @RequestScoped
public class HomePage {
    //user context, tracks current user
    private final User user;

    //page state variables
    private List<Tweet> tweets;
    private Tweet newTweet = new Tweet();

    //service dependencies
    private final TweetManager tweetManager;

    @Inject
    public HomePage(TweetManager tweetManager, User user) {
        this.tweetManager = tweetManager;
        this.user = user;
    }

    ...
}
```

This is safe because the session scope is wider than the request scope. This means that HomePage instances are shorter lived than User instances and therefore run no risk of scope-widening injection. HomePage retrieves a user's tweets by querying the TweetManager for tweets by username:

```
@Get
public String get() {
```

```

        //load tweets for current user
        this.tweets = tweetManager.tweetsFor(user.getUsername());

        //stay on current page
        return null;
    }

```

This method is called in normal operation when an HTTP GET request is received by the web server. The annotation `@Get` indicates to Google Sitebricks to use this method as an event handler prior to rendering the page. Once tweets are loaded (and stored in the `tweets` field), the template is rendered displaying all of a user's tweets. You may also have noticed a "Sign out" link on the `HomePage`, which a user clicks to leave the session:

```

<a href="http://manning.com/prasanna">Help</a> |
<a href="?action=logout">Sign out</a>

```

`HomePage` reacts to the sign-out action by logging the user out from the `User` service:

```

    @Get("logout")
    public String logout() {
        user.logout();

        return "/login?message=Bye.";
    }

```

Because of the `@On("action")` annotation at the top of the class, Google Sitebricks knows to call a different event handler. This time, method `logout()` is called since it's annotated with the value-matching action (`@Get("logout")`). The `logout()` method then redirects the user to a login page, where they must log in again to continue using the site:

```

    @Get("logout")
    public String logout() {
        user.logout();

        return "/login?message=Bye.";
    }

```

The login page is both the exit and entry point for the crosstalk website and is also modeled as a Google Sitebricks page.

11.5.4 Logging in and out

`LoginPage`, like `HomePage`, has a Java class and companion template, as shown in figure 11.6.

The template is fairly straightforward. It accepts a username and password, as shown in listing 11.9 (see figure 11.6 for the web version).

Listing 11.9 An HTML template for `LoginPage` displays login input fields

```

<html>
<head>
    <title>Login</title>

```

```

<link rel="stylesheet" href="/crosstalk.css"/>
</head>
<body>
  <h2>Please log in to crosstalk.</h2>
  <div class="box">
    <div class="box-content">
      <h3>${message}</h3>
      <form action="/login" method="post">
        <input name="username" type="text" />
        <input name="password" type="password" />
        <input type="submit" value="login"/>
      </form>
    </div>
  </div>
</body>
</html>

```

A dynamic success or fail message

The login form

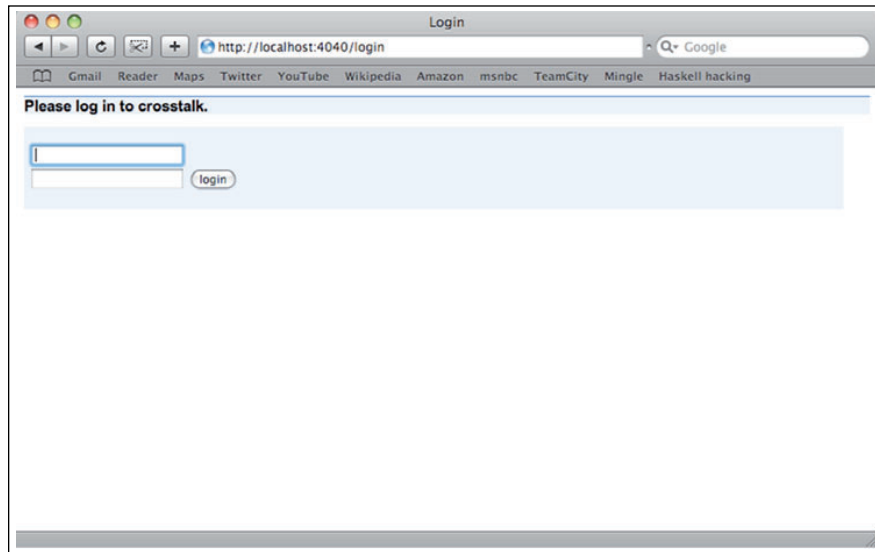


Figure 11.6 All users must log in to use their home page on crosstalk.

This form posts back to LoginPage with the entered credentials. LoginPage then uses this information to authenticate the user against a UserManager, as shown in listing 11.10.

Listing 11.10 Page class LoginPage authenticates users logging into crosstalk

```

@At("/login") @RequestScoped
public class LoginPage {
    private String username;

```



```

private String password;

private String message = "";

//service dependencies
private final UserManager userManager;
private final User user;

@Inject
public LoginPage(UserManager userManager, User user) {
    this.userManager = userManager;
    this.user = user;
}

@Post
public String login() {
    //attempt to authenticate the user
    if (userManager.authenticate(username, password))
        user.login(username);
    else {
        //clear user context from session
        user.logout();

        //stay on this page with error
        return "/login?message=Bad+credentials.";
    }

    //redirect to home page if successfully logged in
    return "/home";
}

//getters/setters...
public void setUsername(String username) {
    this.username = username;
}

public void setPassword(String password) {
    this.password = password;
}

public String getMessage() {
    return message;
}

public void setMessage(String message) {
    this.message = message;
}
}

```

LoginPage has the standard getters and setters to expose its properties to the template and form input. Method login() asks UserManager (a data service) whether the username and password are valid:

```

@Post
public String login() {
    //attempt to authenticate the user
    if (userManager.authenticate(username, password))
        user.login(username);
}

```

```
else {  
    //clear user context from session  
    user.logout();  
  
    //stay on this page with error  
    return "/login?message=Bad+credentials.";  
}  
  
//redirect to home page if successfully logged in  
return "/home";  
}
```

If they are valid, `login()` logs the user in via the session-scoped `User` service shown earlier. If not, a redirect is sent back to the login screen with an error message:

```
@Post  
public String login() {  
    //attempt to authenticate the user  
    if (userManager.authenticate(username, password))  
        user.login(username);  
    else {  
        //clear user context from session  
        user.logout();  
        return "/login?message=Bad+credentials.";  
    }  
  
    //redirect to home page if successfully logged in  
    return "/home";  
}
```

This results in the “Bad credentials” error message as shown in figure 11.7.

Once successfully logged in, the user is sent to their home page to tweet away.

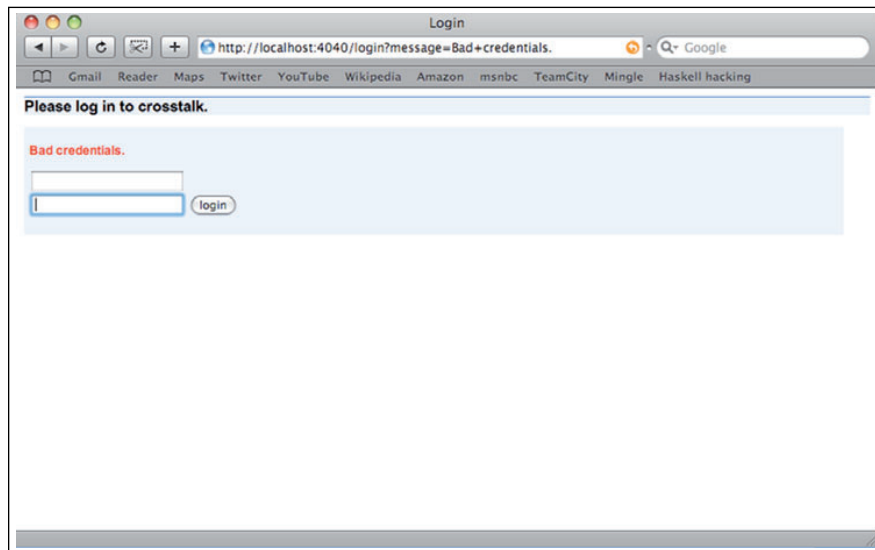


Figure 11.7 Bad usernames or passwords are rejected, appropriately.

11.6 The persistence layer

Data services that crosstalk uses are fairly simple. There's one for managing users (UserManager) and one for managing tweets (TweetManager)—not rocket science. Their implementations are subtly different. Let's look at the TweetManager implementation first, as shown in listing 11.11.

Listing 11.11 Implementation for the TweetManager interface using Hibernate

```
@Immutable @Singleton
class HibernateTweetManager implements TweetManager {

    //the provider pattern helps us prevent scope-widening of sessions
    private final Provider<Session> session;

    @Inject
    public HibernateTweetManager(Provider<Session> session) {
        this.session = session;
    }

    @Finder(query = "from Tweet where author = :author")
    public List<Tweet> tweetsFor(@Named("author") String author) {

        //this method is intercepted by warp-persist DynamicFinders
        // and converted into a query. So you should not see an empty
        // list unless the database contains no Tweets for 'author'.
        return Collections.emptyList();
    }

    public void addTweet(Tweet tweet) {
        session.get().save(tweet);
    }
}
```

HibernateTweetManager implements the TweetManager interface by storing and querying tweets from the database using Hibernate's services. To add a new tweet, we use a session to the database and save a new instance of the Tweet data object:

```
public void addTweet(Tweet tweet) {
    session.get().save(tweet);
}
```

We use the Provider pattern here to avoid scope-widening the session, which is implicitly request scoped. I say implicitly, because its behavior is managed by warp-persist, as you'll see. HibernateTweetManager, like all other classes in the services package, is a thread-safe, immutable singleton. Querying objects is taken care of us by warp-persist's *Dynamic Finders* utility. Annotating the method tweetsFor() with @Finder tells warp-persist to intercept and replace the method with logic that runs the bound query:

```
@Finder(query = "from Tweet where author = :author")
public List<Tweet> tweetsFor(@Named("author") String author) {

    // this method is intercepted by warp-persist DynamicFinders
    // and converted into a query. So you should not see an empty
    // list unless the database contains no Tweets for 'author'.
    return Collections.emptyList();
}
```

The string "from Tweet where author = :author" is an HQL (Hibernate Query Language) query, which tells Hibernate to load all tweets with the matching author. The method parameter `author` is annotated with an `@Named("author")` annotation, which binds the parameter into the query at the label `:author`.

TIP For more information on `warp-persist` and dynamic finders, including tutorials, visit <http://www.wideplay.com>.

The `userManager` service by contrast, does not use Hibernate to store users in the database. This is done as a matter of convenience to get us up and running quickly, as shown in listing 11.12.

Listing 11.12 A UserManager that uses hard-coded user credentials

```
@Immutable @Singleton
class InMemoryUserManager implements UserManager {
    private final Map<String, String> users;

    public InMemoryUserManager() {
        final Map<String, String> users = new HashMap<String, String>();
        users.put("dhanji", "dirocks");

        //freeze the current hardcoded map so that it is thread-safe
        this.users = Collections.unmodifiableMap(users);
    }

    //returns true if a username and password combination is valid
    public boolean authenticate(String username, String password) {
        return users.containsKey(username)
            && users.get(username).equals(password);
    }
}
```

A simple hash table is stored in memory in this implementation, and the `authenticate()` method checks values in this hash table. We ensure that this hash table is immutable by wrapping it in an unmodifiable map. You should recall this best practice from chapter 10.

```
public InMemoryUserManager() {
    final Map<String, String> users = new HashMap<String, String>();
    users.put("dhanji", "dirocks");

    //freeze the current hardcoded map so that it is thread-safe
    this.users = Collections.unmodifiableMap(users);
}
```

This ensures that `InMemoryUserManager` is both highly concurrent and thread-safe (since it is immutable after construction). In a broader implementation, we'd use a Hibernate version of `UserManager` and have a separate set of pages for users to register and manage their accounts.

Now, let's look at interacting with a data store and retrieving and storing tweet data.

11.6.1 Configuring the persistence layer

As you saw in section 11.2, all the components in the services package are separately configured. The ServicesModule takes care of this for us and is shown in listing 11.13.

Listing 11.13 Security and persistence configuration (see also listing 11.1)

```
package com.wideplay.crosstalk.services;

public final class ServicesModule extends AbstractModule {
    private static final String HIBERNATE_CONFIG = "hibernate.properties";

    @Override
    protected void configure() {

        //configure persistence services, using hibernate
        install(PersistenceService
            .usingHibernate()
            .across(UnitOfWork.REQUEST)
            .buildModule()
        );

        //configure hibernate with our tweet data model class
        bind(Configuration.class).toInstance(new AnnotationConfiguration()
            .addAnnotatedClass(Tweet.class)
            .setProperties(loadProperties(HIBERNATE_CONFIG))
        );

        //configure crosstalk data services
        bind(TweetManager.class).to(HibernateTweetManager.class);
        bind(UserManager.class).to(InMemoryUserManager.class);

        //configure the in-memory authenticator (with hard-coded users)
        final HttpSessionAuthenticationManager manager =
            new HttpSessionAuthenticationManager();
        bind(AuthenticationManager.class).toInstance(manager);

        //intercept any @Get or @Post method on any page except LoginPage
        bindInterceptor(
            not(subclassesOf(LoginPage.class)),
            annotatedWith(Get.class).or(annotatedWith(Post.class)),
            new SecurityInterceptor(manager)
        );
    }

    private static Properties loadProperties(String props) {
        ...
    }
}
```

First up, we configure warp-persist to use Hibernate and provide us with a Hibernate unit of work (session to the database) every time an HTTP request arrives:

```
install(PersistenceService
    .usingHibernate()
    .across(UnitOfWork.REQUEST)
    .buildModule()
);
```

Then we configure Hibernate itself by binding the Configuration class to an annotation-based implementation that contains our Tweet domain model class:

```
bind(Configuration.class).toInstance(new AnnotationConfiguration()
    .addAnnotatedClass(Tweet.class)
    .setProperties(loadProperties(HIBERNATE_CONFIG))
);
```

To this configuration, we also add Hibernate configuration properties that tell it which database to use and so on (see the source code accompanying the book for details). The two data services, for users and tweets, are bound directly to their interfaces:

```
bind(TweetManager.class).to(HibernateTweetManager.class);
bind(UserManager.class).to(InMemoryUserManager.class);
```

All this is straightforward. Then we bind another service whose purpose we'll see very shortly:

```
//configure the in-memory authenticator (with hard-coded users)
final HttpSessionAuthenticationManager manager =
    new HttpSessionAuthenticationManager();
bind(AuthenticationManager.class).toInstance(manager);
```

The AuthenticationManager is used by crosstalk's security layer to detect whether a user has been authenticated correctly. This level of indirection is required because we can't always directly inject User into our services. You'll see why in a second. But first, a SecurityInterceptor is bound to all methods annotated with @Get and @Post except the LoginPage:

```
//intercept any @Get or @Post method on any page except LoginPage
bindInterceptor(
    not(subclassesOf(LoginPage.class)),
    annotatedWith(Get.class).or(annotatedWith(Post.class)),
    new SecurityInterceptor(manager)
);
```

This interceptor is used to secure parts of the website by intercepting requests for web pages, allowing us to ensure that only public parts of the site (that is, the login page) are seen by users who are not authenticated and also that only authenticated users may post tweets to their home pages. Let's see how this security interceptor works.

11.7 The security layer

SecurityInterceptor is an AOP interceptor that runs on every event method, as shown in listing 11.14. It's intended to guard web pages and prevent them from being displayed to users who aren't properly authenticated.

Listing 11.14 AopAlliance method interceptor applied across the app for security

```
class SecurityInterceptor implements MethodInterceptor {
    private final AuthenticationManager authenticationManager;

    public SecurityInterceptor(
```

```

        AuthenticationManager authenticationManager) {
            this.authenticationManager = authenticationManager;
        }

        public Object invoke(MethodInvocation methodInvocation)
            throws Throwable {

            //proceed with normal execution if a user is logged in
            if (authenticationManager.isLoggedIn()) {
                return methodInvocation.proceed();
            }

            //redirect to login page if the user is not properly logged in!
            return "/login";
        }
    }
}

```

SecurityInterceptor is an AopAlliance MethodInterceptor,¹ which is triggered every time an HTTP GET or POST handler method is fired by Google Sitebricks (except on LoginPage). If a user is logged in, then the method proceeds as normal. If not, a redirect is sent to the login page:

```

        public Object invoke(MethodInvocation methodInvocation)
            throws Throwable {
            ...

            //redirect to the login page if user is not properly logged in!
            return "/login";
        }
    }
}

```

Method interceptors and their configuration are covered in chapter 8. If you're unsure of how the interception works, consult the section on proxying.

In the next section you'll see how the application's lifespan can be tied to a servlet engine so that it receives notifications about important events in its lifecycle.

11.8 *Tying up to the web lifecycle*

Our database persistence framework has several expensive operations it must perform when starting up and several cleanup operations when shutting down. These are independent of a unit of work and must occur outside the regular operational life of the web application.

Ideally, this coincides with the web application's own lifecycle and the servlet's `init()` and `destroy()` events. Fortunately for us, warp-persist's `PersistenceFilter` handles all this hard work. On an `init()`, it triggers the creation of connection pools, the creation of tables and resources, and so on. In our case, this also kick-starts the Hypersonic HSQL database in memory. On shutdown of the filter (when `destroy()` is called), it closes down the Hibernate `SessionFactory`.

In addition, the `PersistenceFilter` is necessary to open and close Hibernate units of work on every request. This filter must be registered before installing the `SitebricksModule` so that pages can benefit from the session to the database being open.

¹ Refer to chapter 8 for details on AOP and interceptors.

So far, we've looked at presentation, persistence and modularity. We've also used interception to provide security and tied everything up with the web lifecycle. Let's now get crosstalk running and give it the real litmus test.

11.9 Finally: up and running!

To run crosstalk, we'll create a simple class with a main method that fires up *Jetty*. We'll put this class in the `test/` directory to distinguish it from the application proper. It looks like this:

```
public class Crosstalk {
    public static void main(String... args) throws Exception {
        Server jetty = new Server(8080);        //start at port 8080
        server.addHandler(new WebApplicationContext("web", "/"));

        jetty.start();
        jetty.join();
    }
}
```

Class `Crosstalk` creates an instance of `Server`, which represents the *Jetty* web server. We configure it to listen at port 8080 and use the `web/` directory as its resource root. Finally, we also ask it to listen at the root URL context: `"/`.

Once *Jetty* is started, it's ready to begin servicing requests. We must use the `join()` method subsequently to prevent the application from exiting too early:

```
jetty.start();
jetty.join();
```

Now you're free to run the application (see figure 11.8). Point your browser to `http://localhost:8080/login` to see the login page and start using crosstalk!

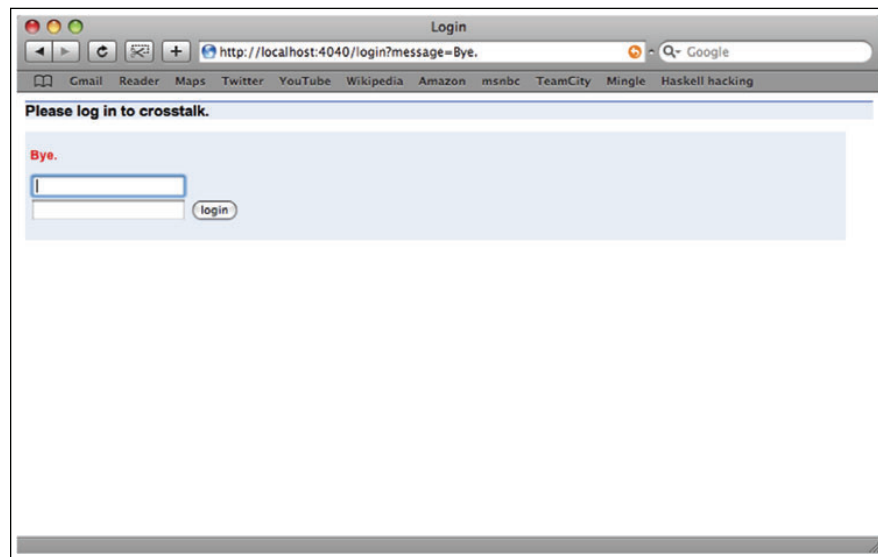


Figure 11.8 See ya!

11.10 Summary

This chapter was an example walkthrough of a real, working web application that embodies many of the concepts you’ve studied thus far in this book. I chose crosstalk as the problem domain, which is a simple clone of the popular microblogging service Twitter.

Crosstalk starts life as a Guice-based application that uses Hibernate for persistence and Google Sitebricks as the web framework to display web pages and react to user input. We also use Mort Bay Jetty as the servlet container. The injector is bootstrapped (as described in chapter 2) by using the guice-servlet integration layer that allows us to create and configure the injector when the web application is deployed to the servlet container.

Crosstalk consists of three main packages:

- web—Contains the web page classes and components
- services—Contains the data and security services
- tweets—Contains the domain model classes for data persistence

All the components of package `services` are exposed via interfaces, and their implementation details are hidden away as package-local. This is an example of good modularity and loose coupling as shown in chapters 4 and 10.

Web pages are all request-scoped and utilize class `User`, which is a session-scoped service that tracks the current user who is logged in. `LoginPage` ensures that a username is attached to the `User` service upon login, and `HomePage` signs a user out as appropriate. These web components exemplify the ideas presented in chapters 5 and 6 on scoping.

Since starting and stopping the persistence engine is an expensive operation and ideally coincides with the start and close of the web application itself, we register a wrapping `PersistenceFilter` that allows us to hook into the servlet lifecycle. Refer to our investigation of lifecycle and its semantics in greater detail in chapter 7.

The `SecurityInterceptor` was added to secure home pages from users who aren’t logged in. This is an example of the behavior modification or AOP interception techniques presented in chapter 8.

Finally, we tied all of these patterns and techniques with the whole application to realize the full benefits of dependency injection!

appendix A: *The Butterfly Container*

Contributed by Jakob Jenkov

This appendix is about the Butterfly DI DSL called *Butterfly Container Script (BCS)*. We include this appendix to illustrate the last of the four configuration mechanisms mentioned in chapter 2.

A.1 A DSL for dependency injection

What we show here is far from a complete listing of the capabilities of Butterfly Container. Rather we show a few of Butterfly Container's solutions; these problems are also mentioned elsewhere in this book:

- Contextual injection via input parameters
- Reinjection via factory injection

This appendix also looks at a few features made possible by a DSL.

A.2 Application configuration

Butterfly Container is an open source project and is part of a growing collection of components called *Butterfly Components*. Should you be interested in learning more, you can find additional information at <http://butterfly.jenkov.com>.

A.2.1 The DSL basics

For reasons too involved to discuss here, Butterfly Container was designed to use a DSL tailored to dependency injection for its configuration. This mechanism is somewhat similar to the XML files of Spring, except BCS is not an XML format.

Butterfly Container Script was designed to look like a cross between a standard property file (name = value) and Java code. This was done to make it easier to read, write, and learn since most Java developers are familiar with both Java and property files. This language design also makes it feasible to use BCS files for application configuration (explained later).

A script file consists of one or more bean factories. It's easiest to explain the basics of BCS by using an example:

```
myBean1 = * com.myapp.MyBean();
```

This configuration consists of five parts:

- A factory ID (`myBean1`)
- An equals sign (`=`)
- An instantiation mode (`*`)
- A Factory chain (`com.myapp.MyBean()`)
- A semicolon (`;`)

The Factory ID identifies this Factory. This ID must be unique throughout the container instance. You supply this ID to the container when obtaining an instance or when referencing that factory from other factories.

The equals sign indicates the beginning of the product part of the factory definition.

The instantiation mode is also sometimes referred to as scope. However, not all bean scopes in Butterfly Container are achieved using the instantiation mode. Sometimes a scope is achieved by combining the instantiation mode with a special bean factory, for instance your own bean factory.

BCS currently includes the following instantiation modes:

- `*`—A new instance is created on every call to a Factory.
- `1`—A singleton is created once and returned on every call to a Factory.
- `1T`—One singleton per thread-calling factory is created once and returned on every call to a Factory.
- `1F`—Flyweight; one instance per provided input parameter is created once and returned on every call to a Factory with same input parameter.

The Factory chain defines what object is to be returned by the Factory. In our example that's an instance of the class `com.myapp.Bean`. A Factory can create more than one object when it executes but return only one of them. We'll explain more about the Factory chain later.

The semicolon signals that the Factory chain is finished.

A.2.2 The Factory chain

In the Factory chain of a Factory definition you can call constructors, call methods, and even define local bean factories not visible outside this factory definition. Here's a somewhat more complex example:

```
myBean1 = * com.myapp.MyBean1();
myBean2 = * com.myapp.MyBean2("A Text", myBean1)
           .setMoreText("more text");
```

This configuration consists of two Factories: `myBean1` and `myBean2`. The definition of `myBean1` is pretty much the same as the first example. `myBean2` shows a few new things. First, it contains two constructor parameters to the constructor of the class

`com.myapp.MyBean2`. The second constructor parameter references the `myBean1` factory, meaning an instance obtained from this factory should be obtained and injected into the constructor.

The Factory chain contains the method call `setMoreText("more text")`. This call further configures the `MyBean2` instance before returning it. In fact, if the `setMoreText()` method had returned an object, it would be this object that was returned by the `myBean2` Factory and not the `MyBean2` instance. Since the `setMoreText()` method returns `void`, it's interpreted as returning the instance the method was called on—the `MyBean2` instance, in other words. That also means that you can chain method calls on methods returning `void`, like this:

```
myBean2 = * com.myapp.MyBean2("A Text", myBean1)
           .setMoreText("more text")
           .setAValue(1)
           .setMoreOfSomething("more...");
```

As you can see, the factory now consists of a chain of constructor/method calls, hence the name Factory chain. The Factory chain can be as long as you need it to be.

This is one of the advantages of having a tailored DSL: the freedom to add features like method chaining on methods returning `void`.

A.2.3 Contextual injection via input parameters

BCS allows contextual injection by enabling factories to receive input parameters. Here's how that looks:

```
myBean1 = * com.myapp.MyBean($0).setValue($1);
myBean2 = * myBean1(com.myapp.SomeBean(), "Value Text");
```

The dollar sign (\$) signals that an input parameter is to be injected. The number (0, 1, and so on) specifies which input parameter to inject. As you can see, input parameters are not named or typed. This can lead to type errors in your Factory definitions if you aren't careful. This is one of the limitations of a custom DSL. It's only as advanced as you make it; you don't get Java's full type checking. In reality this is a big problem, as you might think. You can set up unit tests to check the validity of the factories, if you want to.

The `myBean2` Factory calls the `myBean1` Factory and provides two input parameters to it: a `com.myapp.SomeBean` instance and the string "Value Text". Notice that each input parameter is itself a Factory chain, so you could also have provided input parameters to the `SomeBean` instance and called methods on it, like this:

```
myBean1 = * com.myapp.MyBean($0).setValue($1);
myBean2 = * myBean1(com.myapp.SomeBean("config", $0)
                    .setTitle("The Title"),
                    "Value Text");
```

The `SomeBean` constructor is now given two parameters: the string "config" and the input parameter \$0. This input parameter is not the same as the \$0 of the `myBean1` Factory. The \$0 input parameter of the `myBean1` Factory is the `SomeBean` instance. The \$0 input parameter of the `myBean2` Factory is whatever you provide as first parameter to the `myBean2` Factory.

You can also provide input parameters to a Factory when obtaining beans from the container from Java code, like this:

```
MyBean myBean = (MyBean)
    container.instance("myBean1", new SomeBean(), "Value Text");
```

If the method you're trying to inject parameters into is overloaded, it can be hard or even impossible for the container to tell which of the overloaded methods to call. It's therefore possible to specify the type of the injected parameter, like this:

```
myBean1 = * com.myapp.MyBean((long) $0).setValue((String) $1);
```

A.2.4 Reinjection via factory injection

BCS makes it possible to inject the Factory of a product rather than the product itself. This is done by putting a # in front of the Factory name when referencing it, like this:

```
firstBean = * com.myapp.FirstBean($0).setValue($1);
secondBean = * com.myapp.SecondBean().setFirstBeanFactory(#firstBean);
```

In the `secondBean` Factory, the `firstBean` Factory is injected into the method `setFirstBeanFactory()`, by referencing the injected Factory as `#firstBean`.

While you could declare the type of the Factory inside the `SecondBean` instance of type `com.jenkov.container.itf.factory.IGlobalFactory`, this would result in a hard reference in your code to a Butterfly Container class. You don't want that. Instead, Butterfly Container is capable of adapting the Factory to your own custom interface as long as the interface has an `instance()` method. Here's an example of such an interface:

```
public interface FirstBeanFactory {
    public FirstBean instance();
}
```

In the `SecondBean` class you can now create a field of the type `FirstBeanFactory`, like this:

```
public class SecondBean {
    protected FirstBeanFactory firstBeanFactory = null;

    public void setFirstBeanFactory(FirstBeanFactory factory){
        this.firstBeanFactory = factory;
    }

    public void doSomething(){
        FirstBean firstBean = this.firstBeanFactory.instance();
    }
}
```

Notice how the use of the `FirstBeanFactory` interface inside the `SecondBean` is perfectly type-safe: no casts and no references to any Butterfly Container-specific classes. You can even specify typed parameters to the `instance()` method. If you look at the Factory definition of `firstBean`, you can see that it actually takes two parameters. You can add them to the `instance()` method like this:

```
public interface FirstBeanFactory {  
    public FirstBean instance(long constructorParam, String value);  
}
```

These parameters are typed as well, making it easy for maintenance developers to see what types are required by the firstBean factory.

You could even call the instance() method by the name of the Factory in the container to call, like so:

```
public interface FirstBeanFactory {  
    public FirstBean firstBean(long constructorParam,  
        String value);  
}
```

This can be used to add several Factory methods to the same interface, referencing different Factories in the container, for instance:

```
public interface BeanFactories {  
    public long    numberOfConnectionsInPool();  
    public String dbUrl();  
    public FirstBean firstBean(long constructorParam, String value);  
    // etc...  
}
```

The container would determine at runtime which Factory to obtain the object from by matching the method name with a factory in the container. It doesn't matter which factory you inject. If your method is called instance(), the injected factory is called. If your method is called something else, the Factory having the same name as the method is called.

While this is a powerful feature, you should use it with care. By using named Factory methods in your interface, you create a hard link between the method names and the Factory names. If the Factory names are later changed, the interface methods will stop working and you won't know why, unless you have a unit test determining that it works. You also have the slight disadvantage in that Java doesn't allow the same characters in method names as Butterfly Container allows in Factory names. For instance, if your Factory is named dao/projectDao, you cannot name a method in Java dao/projectDao(). You can still inject and call the Factory, however, using the method name instance(). But then you can have only one method per interface. In most cases, this is sufficient, so it's not a big limitation.

appendix B: SmartyPants for Adobe Flex

Contributed by Josh McDonald

SmartyPants-IOC is a dependency injection framework for use in building Adobe Flex and Flash applications inspired by Google Guice for Java. It's based on key concepts:

- The class+name key
- Injector requests, usually specified via metadata
- Injector rules, specified using an ActionScript-based DSL

B.1 The class+name key

Whether you're telling SmartyPants-IOC what to provide or sitting around with your hand out asking for a dependency, the key is the same: a combination of a class and a name. You must specify the class, but the name is optional.

B.1.1 Injection annotations

How do you actually request that a field be injected? In most circumstances you need to annotate your fields with ActionScript metadata. If you want to request an instance of `IServiceInterface`, the syntax is simple:

```
[Inject]
public var myService:IServiceInterface;
```

Let's say you want to look up a particular string, rather than just anything. For a Web Services Description Language (WSDL) URL, or something along those lines, use this syntax:

```
[Inject(name="mainWSDL")]
public var myServiceWSDL:String;
```

These fields will be injected into your object when it's constructed for you by SmartyPants-IOC or when you call `injector.injectInto(myInstance)`.

SmartyPants-IOC also supports live injections, which when coupled with live rules behave like the Flex SDK's data-binding functionality:

```
//This will be set whenever the source changes
@Inject(name="userName",live)]
public var currentUsername:String;
```

B.2 Injector rules

The injector rules are akin to Guice's bindings, but we use another term to avoid confusion with the Flex SDK's data-binding mechanism. You tell the injector what you'd like it to do, using the DSL. Here are a couple of examples:

```
//Simple singleton rules:
injector.newRule().whenAskedFor(String).named("wsdl")
    .useInstance("http://www.server.com/soap/service.wsdl");
injector.newRule().whenAskedFor(IServiceInterface)
    .useSingletonOf(MyServiceImpl);
injector.newRule().whenAskedFor(MyConcreteClass).useSingleton();

// "Live" rules act just like <mx:Binding>
injector.newRule().whenAskedFor(String)
    .named("userId")
    .useBindableProperty(this, "userId");
```

B.2.1 But how do I kickstart the whole thing?

Good question! Two ways. First, in an MXML component, this code will inject into your component on the `CreationComplete` event:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Canvas xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:smartypants="http://smartypants.expantra.net/2008">
    <smartypants:RequestInjection/>
    <!-- ...Regular code and MXML... -->
</mx:Canvas>
```

And in ActionScript it's even simpler:

```
SmartyPants.injectInto(this);
```

Hopefully this gives a little more insight into the style and ideas behind SmartyPants-IOC. Be sure to check it out on the Google Code site!

SmartyPants-IOC is hosted on Google Code: <http://code.google.com/p/smartypants-ioc/>.

Symbols

@At annotation 298
@Autowired 199
 annotation on constructor 34
@EJB 199
@Entity 301
@Get 298
@GuardedBy 198
@Immutable 246, 260–261
@Inject 199
 @Autowired in place of 35
 skip annotation 93
@On 298
@PostActivate 200
@PostConstruct 199
@PreDestroy 199
@PrePassivate 200
@Remove 199
@Repeat 299
@RequestScoped 298, 300, 303
@ScopingAnnotation 163
@SessionScoped 303
@Singleton 132, 145
 missing 172
@ThreadSafe 168, 246
@Transactional 211, 222

A

Abstract Factory pattern. *See*
 Factory pattern
abstract identity
 referred to as a key 37
Acegi Security 149

ACID transactions 157
Adapter pattern 95, 119,
 175, 180
Adobe Flex 320
advice 212
 avoid too much 219
Algorithmic complexity 131
Amazon 198
annotations
 feature of Java 26
Ant 226
anti-pattern 196, 271
 black box 276–280
 fragile base 277
 lifecycle 276
 rigid configuration 271
 singleton 135, 276
 type unsafe configuration 271
 unwarranted constraints 273
AOP 16, 312
 Guice's approach 18
 intercepting methods 211
 interceptor 311
 make deferred
 modifications 234
 use caution when using 221
AOP injection
 See also method decoration 62
AopAlliance 18, 218
 and Guice 214
Apache 269
Apache Avalon
 earliest DI library 17
 interface injection 61
Apache CXF 280

Apache Excalibur
 Fortress 17
Apache Hadoop
 cluster-computing 184
Apache HiveMind 18–19, 269
Apache James 17
Apache OpenJpa 225
Apache Tapestry 18, 269
Apache Tomcat 290
Apache Wicket 269
application logic 51
 defined 51
 separate from infrastructure
 logic 52
application server 198, 200
AspectJ 18
 build-time weaving 234
 pointcut 215
 Springs library 214
aspect-oriented programming.
 See AOP
Assisted Injection pattern 86
AssistedInject. *See* Assisted
 Injection
atomic operations 263
atomicity 157
autowiring
 constructor 34

B

BCS
 instantiation modes 316
 not an XML format 315
Bean Validation 280

- BeanFactoryAware 84
- behaviorally consistent 109
- Beust, Cedric 137
- binding annotations 255
- bindings
 - determine type 43
 - keys provide 37
 - wrong type 43
- black box
 - anti-patterns 276
 - vs. encapsulation 276
- boilerplate
 - removing 181
- bootstrap
 - combined with
 - configuration 32
- bootstrapping
 - the injector 22, 42
- Builder pattern 16, 88
- Butterfly Components 315
- Butterfly Container 315
 - bean scopes 316
- Butterfly Container Script. *See* BCS

C

- C# 14, 19, 46, 83
- C++ 19
 - destructor 189
 - solves general problems 16
- cache scoping. *See* singleton
- caching
 - in-memory 181
- Castle MicroKernel 19
- CGLib
 - Guice and Spring 218
- circular initialization
 - no injection solution 77
- circular reference 71
 - solve with proxies 75
 - two solutions 72
- circularity 71
 - breaking 73
- class
 - testability 4
- class name
 - fully qualified 273
- classloader 217
- classpath 274
- client
 - definition 3
 - object is 2
 - testing 9
 - verify behavior 10

- client code
 - not invoking injector 24
- clients 198
- Closeable 197
 - definition 197
- cluster computing 184
- coarse grained 157
- code
 - behaviorally focused 22
 - modular design 22
 - separation of infrastructure
 - and application 15
 - testability 22
 - testable 100
 - thread-unsafe 113
- combinatorial bindings
 - build in Spring JavaConfig 50
- combinatorial keys 47, 161
 - benefits 48
 - drawback 48
 - Guice 49
- complexity
 - algorithmic 131
 - linear 131
- concurrency 10, 246, 257–265
 - vs. synchronization 262
- concurrency problems
 - from scope widening 176
- concurrent 136
- concurrent access
 - to counter 178
- configuration
 - combined with bootstrap 32
- confined to single threads. *See* thread confinement
- connection pool 195
- connection validation 195
- consistency 157
- ConstraintFactory. *See* JSR-303
- construction by hand 5, 10
 - creating injector 28, 32
 - DI enables testability 13
 - manual DI 13
 - problems 6
 - toothpaste example 124
 - why so named 6
- constructor
 - autowiring 34
 - initialization hook 56
 - purpose 55
- constructor injection 74, 128
 - advantage 6
 - circular reference
 - problem 71

- contextual 85
 - dominant form of idiom 65
 - first preference 81
 - leverages immutability 67
 - object validity 78
 - vs. setter injection 55, 66
- constructor pyramid 69–71
- constructor wiring 29
 - PicoContainer 33
- context objects
 - set when needed 91
- contextual injection
 - Builders 90
 - variation of partial
 - injection 84
- contract
 - revelation of intent 110
- cookie 154
- Copland 19
- crosscutting concern 210
 - logging 210
- crosstalk 290
- layers 290
 - modularity and services
 - coupling 295
 - presentation layer 296
 - requirements 290
- custom annotations 283
- custom scope 157–166
 - in Guice 160
 - registering in Guice 163
 - Spring 164

D

- DAO 38, 247
- Data Access Object. *See* DAO
- data type
 - scalar 250
- database 111, 123, 224, 292, 312
 - connection pool 129
 - map object to row 224
 - Oracle 111
 - pool of connections 130
 - PostgreSQL 111
- Db4objects
 - warp-persist 222
- decorate. *See* Decorator
- Decorator Design pattern 62
- Decorator pattern 16
- delayed injection. *See* partial injection
- delegator 233
- Delegator pattern 234

- dependencies 24
 - half-constructed 67
 - identifying for injection 36
- dependency 3
- dependency graph. *See* dependency
- dependency injection framework. *See* dependency injector
- dependency injection. *See* DI
- dependency injector 13, 15
 - features 37
- dependent 3
- deployment profiles 118
- destroy lifecycle hook 196
- destruction
 - times and frequency 191
- Destructor anti-pattern 196
 - discourage use of 197
- DI 1
 - DSL used in 16
 - helps loose coupling 111
 - infrastructure code 111
 - manual 13
 - not an invasive technique 28
 - not explicitly knowing 13
 - popular use 15
 - real world use 17
 - scoping 25
 - vs. IoC 15
- DI library 52, 55
 - configuring the injector 26
 - method decoration
 - support 82
- DI solutions
 - fragmentation 267–270
- direct field injection
 - difficult to test 269
- documentation 110
- domain model 296, 301
- Domain Specific Language. *See* DSL
- domain-specific scope 139
 - building web applications 140
- downcast
 - defined 28
 - returned instance 32
- downcasting
 - Provider pattern 83
- DSL 16
 - for dependency injection 315
 - injection-oriented 26
 - solves specific problems 16
- duck-typed 19
- durability 157

- Dynamic Finders 308
- dynamic proxies
 - cannot intercept private methods 231
- See also* proxy

E

- eager instantiation 129
 - vs. lazy instantiation 201
- EasyMock 114–115
- EJB 12, 114, 182
 - does not support constructor injection 199
 - lifecycle hook 200
 - stateful session 269
- EJBs
 - stateful 198
- Emailer
 - change behavior 5
- encapsulation 5, 14, 252
 - class member 30
 - destroyed 253
 - at package and module level 257
 - principles violated 7
 - vs. black box 276
 - warp-persist 256
- Enterprise Java Bean. *See* EJB
- EntityManager. *See* JPA
- escape
 - during construction 245
- execution context 25

F

- Factory
 - equivalent of singleton
 - scoping 136
- Factory chain 316
- Factory pattern 7–11, 24
 - applied to Emailer 7
- faulty configuration 67
 - causes heartache 68
- field injection 65
- filter
 - traces requests 268
- FilterToBeanProxy 149
- final fields
 - attempt to modify 246
- finalization 189
- domain-specific patterns 197
- finalizer
 - can come in handy 190

- finite resources
 - disposal of 189
 - freeing 196
 - modeling as Closeables 197
- Fortress 17
- Fowler, Martin 15
- fragile base class anti-pattern 277
- framework
 - lessons for designers 270–280
- framework design
 - replacement with mocks
 - essential 270
- fully qualified class name 273
- functionality
 - changing in base class 277
 - share 277

G

- Gang of Four 88, 95, 128
- garbage collection 189
 - finalizer before collector runs 189
 - leads to memory leaks 179
- garbage collector. *See* garbage collection
- Gin 18
- Gmail 2
- good key behavior 46
 - rules 47
- Google 12
- Google AdWords 18
- Google Guice. *See* Guice
- Google Sitebricks 269
 - configuring 294
- Google Web Toolkit. *See* GWT
- graphical user interface. *See* GUI
- grid
 - computing 183
 - Oracle Coherence 182
 - sometimes called cluster cache 182
- grid scope 182
- GUI 267
- GUICE
 - AOP 291
- Guice 294–295, 320
 - approach to AOP 18
 - binding scopes 147
 - bindings 45
 - binds under no scope 131
 - CGLib 218
 - combinatorial keys 49
 - compact setter injection 59

Guice (*continued*)
 create Emailer 23
 custom scope 160
 doesn't react to
 @Immutable 168
 as injector 290
 Jabber 254
 key as binding 29
 key as identifier 29
 modules defined 30
 nomenclature 127
 Providers out of the box 83
 request scoping 144
 scopes 124
 setter injection 87
 singletons 246
 testing matchers 213
 transaction scope 161
 warp-persist 222
 Guice servlet 144
 guice-servlet 25, 222, 291, 293
 GWT 18

H

Half-Life 43
 Hammant, Paul 15
 hashtable 241, 244
 no locking 264
 synchronized wrapper
 around 264
 thread-local 162
 Hellesøy, Aslak 15
 Hibernate 266, 290–291
 JPA 225
 warp-persist 222
 Hibernate Query Language. *See*
 HQL
 Hollywood Principle 15, 138
 scopes apply to state of
 objects 181
 See also IoC
 HomePage
 sign out link 304
 HQL 309
 HSQL
 database 312
 See also Hypersonic
 HTML
 web applications 140
 HTTP
 request scoped 251
 HTTP filter
 with Spring Security 225

HTTP protocol
 stateless 154
 HTTP request 124, 276
 scope 141
 HTTP session scope 149
 Hypersonic 290, 312

I

IDE
 helps catch errors 45
 identifier
 combinatorial key 36
 identifying by type 44
 limitations 46
 immutability 246
 pitfalls 258
 immutable
 good design 167
 immutable dependencies
 create 66
 in-construction
 cannot break cycle 75
 in-construction problem
 in a nutshell 76
 setter injection 76
 infrastructure code 116
 separating by area 102
 infrastructure logic 15, 51
 separate from application
 logic 52
 inheritance
 replacing with delegation 279
 initialization 196
 called on 199
 times and frequency 191
 injection
 scope widening 145, 169
 injection idiom
 choosing 65
 injector
 all-purpose 24
 bootstrapping 22
 configuration 26
 injector configuration
 changing at runtime 118
 modifying 100
 integration tests 112, 116
 reveal configuration
 quirks 118
 for web application 117
 IntelliJ IDEA 42, 292
 intercepting methods 211
 interception
 lose 87

 modifying behavior 37
 use cases for 221, 234
 interceptor 214, 217
 bound 213
 with Guice 212
 with Spring 214
 interceptor chain 221
 interface injection 60
 pro and con 62
 invasive technique 28
 Inversion of Control. *See* IoC
 IoC 1, 14–15
 container 15
 several meanings 15
 isolation 157

J

Jabber 253
 Java 14, 46, 83, 197, 291
 annotations 26
 atomic library 263
 birthplace of DI 17
 C# and .NET 19
 class member
 encapsulation 30
 Closeable interface 197
 finalizer 189
 garbage collector 179
 hashtable 241
 mock objects frameworks 114
 no proxy for classes 218
 package-local visibility 232
 package-privacy 254
 proxying interfaces 216
 semantics 44
 servlet filter 144
 Set 38
 statically typed 42
 Strings immutable 261
 volatile keyword 168
 Java Community Process
 Bean Validation 280
 Java EE 17, 198
 IoC 15
 Java library
 binary-tree 38
 hash-table 38
 Java Memory Model 79
 Java Naming and Directory
 Interface. *See* JNDI
 Java Persistence API. *See* JPA
 Java Servlet
 registration of servlets and
 filters 267

Java Servlet Framework 191
 detect browser capability 154
 Java Servlet Specification 145
 Java Swing 196
 Java Virtual Machine. *See* JVM
 java.lang.reflect. *See* reflection
 java.lang.reflect.Proxy. *See* proxy
 java.util.HashMap. *See* hashtable
 JavaConfig 45
 JavaServer Faces. *See* JSF
 JBoss Seam 19
 JDK Proxy. *See* proxy
 JMock 114
 JNDI 12
 Johnson, Rod 15, 17
 JPA
 warp-persist 222
 JSF 269
 JSR-303 280
 Bean Validation 280
 doesn't restrict plug-in
 code 284
 JUnit 137
 JVM
 manages threads 169

K

keys
 bound improperly 13
 class literal 46
 combinatorial 47
 good behavior 40, 43, 46
 keyword
 final 233

L

language
 duck typed 19
 dynamically typed 42
 object-oriented 241
 statically typed 42
 latent state 124
 lazy instantiation 201
 vs. eager instantiation 201
 leaking
 of semantics 252
 Lee, Bob 18
 lexical ordering 242
 libraries
 provide pluggable
 services 271

lifecycle 312
 basic events 187
 closely related to scope 187
 customizing 202
 customizing with
 multicasting 205
 customizing with
 postprocessing 202
 domain-specific 191
 events 191, 205
 hook 191
 notification 203
 notifying of events 37
 lifecycle constrained anti-
 pattern 276
 lifecycle events
 not universal 191
 lifecycle hook
 @PrePassivate 200
 leads to unintuitive
 designs 196
 lifecycle scenarios
 servlets vs. database
 connections 191
 linear complexity 131
 locking 245
 logging 212, 229–230
 loose coupling 16, 270
 example 107
 with DI 111

M

manual dependency injection.
 See construction by hand
 matcher 220
 memory complexity 129, 131
 memory leaks
 and scope-widening 179
 memory reclamation 179
 meta-annotation 163
 metadata
 externalized 93
 injector configuration 26
 method decoration 82
 DI library support 82
 fraught with pitfalls 64
 variation on DI 62
 method interception 214
 enhance 220
 methods
 intercepting 211
 mock objects 8, 114
 useful 103
 Mockito 114

modularity 270
 Module 30, 213
 modules 118
 defined 100
 separation and
 independence 101
 Mort Bay Jetty 290, 313
 Mozilla Thunderbird 2
 multicasting 205
 proxy 207
 mutable singletons 168
 mutual exclusion 178

N

namespaces
 use of 38
 NanoContainer 18
 NASA Mars Rover 12
 native method 190
 .NET 19
 network sockets 197
 Nintendo Wii 43
 no scope 125–128
 vs. singleton 133
 no-scoped 170
 nullary constructor 273, 275

O

object
 good citizen 66
 out-of-scope 170
 role of 100
 object creation 187
 object destruction. *See* finaliza-
 tion
 object graph 69
 assembly 36
 building 4
 changing 13
 complex 47
 described in one place 111
 description 3
 emphasis on unit testing 21
 Factory pattern 7
 how to construct 6
 offload burden 5
 requires new factory 11
 size and scope 130
 structure may change 119
 system of dependencies 3
 tightly coupled 104

object/relational mapping. *See* ORM
 object-oriented languages 157
 object-oriented programming.
 See OOP
 objects
 relationship between 3
 OOP 2
 encapsulation 252
 tight coupling 102
 using constructor injection 6
 Oracle 111, 157
 Oracle Coherence 182, 266
 Oracle TopLink
 JPA 225
 ORM tools
 Hibernate 266
 OSCache 271
 uses reflection 273
 out of scope 123, 139
 out-of-container. *See* integration
 test
 OutOfMemoryError 180
 out-of-scope 170
 between scope contexts 175

P

package-local 232, 296
 partial injection 81
 contextual injection 84
 pattern 127
 patterns
 Abstract Factory 7
 Adapter 95, 119, 175, 180
 Assisted Injection 86
 Builder 16, 88
 Decorator 16
 Decorator Design 62
 Delegator 234
 Factory 7–11, 24
 Provider 82, 127, 161,
 180, 308
 See also individual pattern
 name 25–13
 persistence 157, 272–273,
 291, 296
 persistence.xml. *See* JPA
 PicoContainer 14–15, 19
 all-purpose destroy
 method 196
 bias toward constructor
 injection 18
 cache scoping 129
 greedy 33

 identifying by type 46
 injection 31
 locking mode 245
 multicasting 206
 mutable injector 119
 no scope 128
 prefers constructor wiring 33
 PicoContainer 2.0 32
 PicoContainer Gems 206
 pointcut 215, 226–227, 276
 AspectJ 215
 PostgreSQL 111, 157
 postprocessing 202
 power of scopes
 leveraging 181–184
 precompilation 19
 presentation layer 248–249, 296
 programmatic
 configuration 280
 programming to contract
 108, 113
 prototype scope
 Spring's name for no
 scope 127
 Provider pattern 82, 127, 161,
 180, 308
 implemented in Spring 164
 proxy 230, 312
 of class can be powerful 219
 pitfalls 228
 powerful technique 212
 protected methods 232
 resolving circular
 references 74
 Python 19

R

Rahien, Ayende 16
 RDBMS 118
 rebinding 118
 closely tied to scope 119
 with adapter 120
 refactoring
 possible cost 106
 reflection 273
 reinjection 81
 method decoration 82
 with Provider pattern 82
 reinjection problem 161
 scope widening 173
 use mitigants 175
 relational database 157
 Remote Enterprise Java Bean.
 See EJB

request scoped 251
 each instance unique 150
 objects not
 multithreaded 149
 Request scoping
 in Spring 147
 request scoping
 in Guice 144
 request-scoped. *See* request
 scope
 resource bundle 271
 revelation of intent 108
 rigid configuration anti-
 pattern 271
 role interface
 uses 61
 root object
 obtained from injector 25
 Ruby 19

S

safe publication 241
 guarantee of order 244
 subtlety 243
 synchronization 244
 scalar data type 250
 scope 79–80, 87, 124–125
 apply Hollywood
 Principle 181
 closely related to lifecycle 187
 defined 123
 domain-specific 139
 general purpose 123
 HTTP request 141, 157
 HTTP session 149, 157
 managing state 37
 no scope 123
 singleton 123
 thread-confined 158
 scope widening 172
 form of reinjection
 problem 173
 and memory leaks 179
 with thread-safety 179
 scope-widening 170, 179, 241
 domestic problems 176
 scope-widening injection 145,
 169–180
 not caught with unit test 173
 scoping 270
 annotation 153
 functionality 156
 powerful feature of DI 25

- sealed code
 - Guice injects 93
 - injecting objects in 92
- security 296, 311
- service
 - identity of by key 13
- service clusters
 - Oracle Coherence 266
- Service Location pattern 25
- Service Locator
 - JNDI 12
 - pass it a key 12
 - See also* Service Locator pattern
 - typical use 12
- Service Locator pattern 12–13
- service-client
 - relationship 3
- services 295, 304
 - database-specific 141
 - decomposing into objects 3
 - definition 3
 - objects as 2
 - prefixes 38
- services-oriented architecture.
 - See* SOA
- servlet
 - filter 144
 - lifecycle stages 191
 - restrictive structure 268
- servlet lifecycle
 - DI in conjunction with 193
- session bean
 - stateful 198
- session scope
 - each instance shared 150
 - HTTP 149
 - scoping annotation 153
- setter injection 54, 56, 69, 168
 - can't leverage
 - immutability 67
 - common idiom in use 60
 - dependencies wired 78
 - dominant form of idiom 65
 - false negative 68
 - no additional code 70
 - vs. constructor injection
 - 55, 66
- setter method 54, 136, 250
- Ship, Howard Lewis 18
- single-sign-on semantics 182
- singleton 129
 - abused as a bridge 170
 - litmus test 130
 - must be thread-safe 167
 - mutable 168
 - scoping 160
 - when reclaimed 179
- singleton anti-pattern
 - 135, 138, 276
 - See also* singleton objects
- singleton objects
 - make immutable 168
- Singleton pattern 10
- singleton scope 128
 - advantages over no scope 129
 - context is injector 128
 - vs. no scope 133
 - vs. singleton objects 135
- singleton scoping
 - pool of database connections 130
- singleton-scoped objects
 - cached 201
 - eager instantiation in Spring 202
- Sitebricks
 - as web application framework 290
- SmartyPants 320
 - supports live injections 321
- SOA 130, 183
- source code generation 19
- specification of behavior 108
- spellchecker
 - creating one in French 8
- Spring 14, 19, 83, 269
 - @Autowired 27
 - all-purpose destroy method 196
 - autowiring 34
 - bean tag 268
 - beans are singletons 131
 - binding scopes 147
 - CGLib 218
 - check spelling 26
 - constructor injection 17
 - constructor wiring 29
 - custom scope 164
 - IntelliJ IDEA 42
 - JavaConfig 45
 - lifecycle support method 164
 - make aware of autowired classes 35
 - namespaces 39
 - prototype scope 127
 - request scoping 147
 - scopes 124
 - setter injection 57, 72, 93
 - XML configuration 27
- Spring 2.5 34
- Spring Framework 17
- Spring MVC 193, 269
- Spring Security 149, 228, 270
 - requires HTTP filter 225
 - securing methods 224
 - uses Ant-style paths 226
- Spring XML 104
 - binding explicit 29
- SpringMVC 270
- SQL 195, 290
- Stage 201
- stateful
 - context 156
- stateful behavior
 - a peril 174
- static methods
 - proxies cannot intercept 230
- string identifier. *See* string keys
- string keys 29
 - advantage over plain type 47
 - choose wisely 40
 - flexible but unsafe 47
 - identifying by 37
 - limitations 42
- StructureMap 14, 19
- Struts2 18, 269
- Swing 196
- synchronization 244–245
 - vs. concurrency 262
- synchronized keyword 178

T

- TCP connection 130
- testability 4
 - very important 270
- testable 104
- testable code 100
 - crucial to writing 112
- testing. *See* testability
- TestNG 137
- tests
 - automated unit 112
 - integration 112, 116
- thread confinement 163
- thread of execution 158
- thread-local
 - hash table 162
- thread-safe
 - in a class 169

- thread-safe classes
 - creating 167
- thread-safety
 - eternally difficult 179
 - scope-widening 179
- thread-unsafe 178
- tight coupling 104
 - perils 102
 - refactoring impacts 105
- time complexity 131
- Tirsén, Jon 15
- tracing interceptor
 - with Guice 212
 - with Spring 214
- transaction
 - ACID 157
 - specific context 158
- transaction completes
 - commit 162
 - rollback 162
- transaction scope 158, 181
 - apt fit 160
 - in Guice 161
- transaction scoped
 - object sought outside a transaction 163
 - wired 163
- Tweet 300
- Twitter 290
- type
 - identifying by 44
 - resolution 42
 - specific class of data 42

- type keys 46
 - refers to wrong type 45
 - safe but inflexible 47
- type literals
 - analogous to string literals 44
- type unsafe configuration anti-pattern 271
- type-safety
 - disregard for 271

U

- unit test 109, 112
 - and interception 236
 - first line of defense 118
 - injector configuration 114
 - single concern 113
 - violated rules 117
 - writing 4
- unwarranted constraints anti-pattern 273
- URL rewriting 154
- use cases for interception 234

V

- variant 49
 - using spellcheckers 39
- viral injection 25
- volatile 177
 - keyword 168

W

- warp-persist 222, 256
 - @Transactional 222
 - Dynamic Finders 308
- weaving 212
- web applications 140
 - building practical 124
- Web Services Description Language. *See* WSDL
- WebWork 269
- wiring 245
 - absence of directive 35
 - after instance is constructed 56
 - common forms 55
 - construction order 74
 - dependencies 55
 - repeat code 6
 - satisfies circularity 71
- WSDL 320

X

- XML 94
 - boilerplate schema 28
 - injection in Spring 27

Z

- zombie connection 195
- zombie objects
 - unreclaimable 179