3 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.

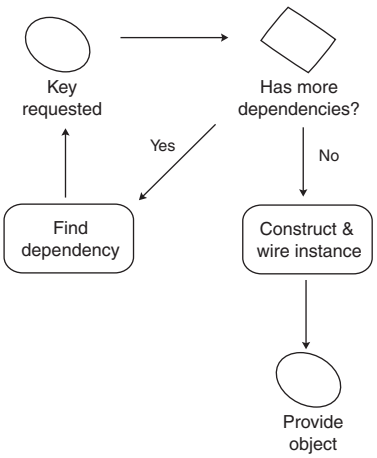

Figure 3.1 Sequence of operations in constructor injection

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).

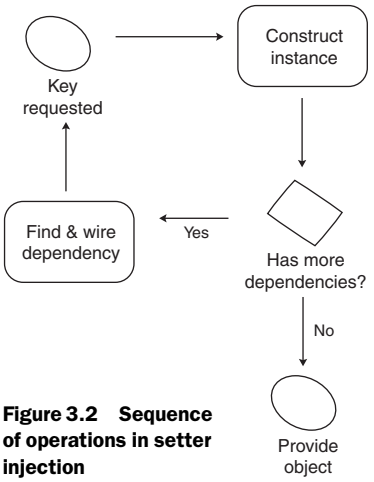

Figure 3.2 Sequence of operations in setter injection

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

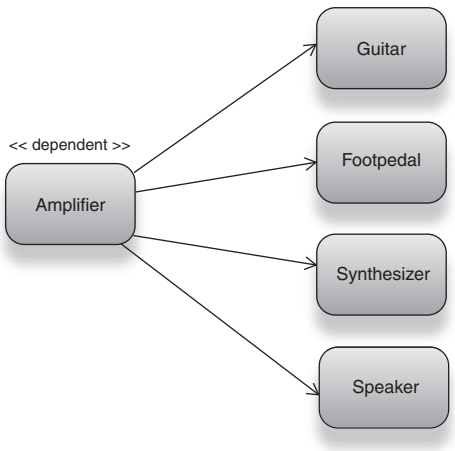

Figure 3.3 Class model of the amplifier and its four dependencies

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">
            <bean class="equipment.Guitar"/>
        </property>

        <property name="speaker">
            <bean class="equipment.Speaker"/>
        </property>

        <property name="footpedal">
            <bean class="equipment.Footpedal"/>
        </property>

        <property name="synthesizer">
            <bean class="equipment.Synthesizer"/>
        </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);
}
```

New role interfaces for each additional dependency

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) {
```

Interface injection methods

```
        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 that 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 Decorator[1] 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*.

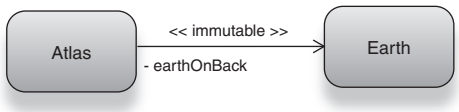

Figure 3.4 `Atlas` depends on `Earth` and cannot be changed once wired.

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

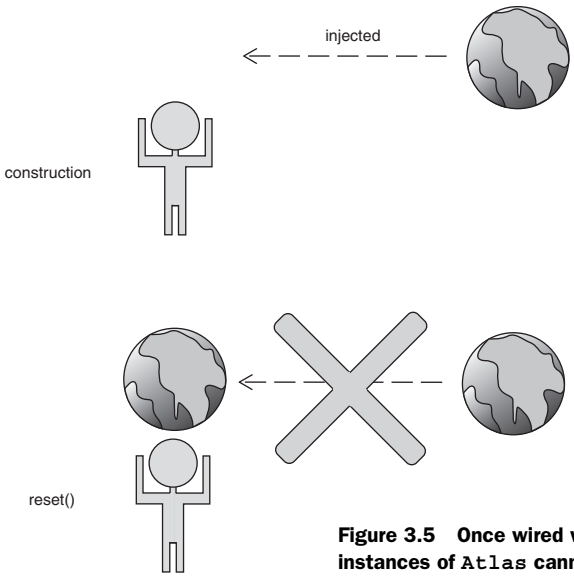

Figure 3.5 Once wired with an `Earth`, instances of `Atlas` cannot be wired again.

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.

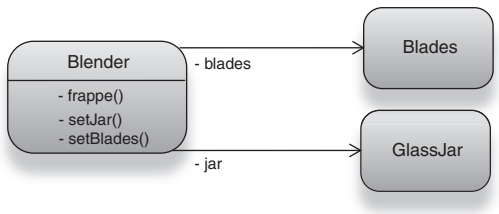

Figure 3.6 `Blender` is setter injected with `Blades` and `GlassJar` (see listing 3.12).

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.

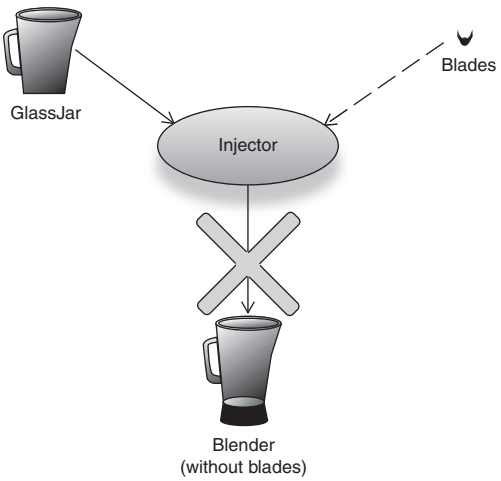

Figure 3.7 `Blender` is incompletely wired, an inadvertent side effect of setter injection.

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.

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.

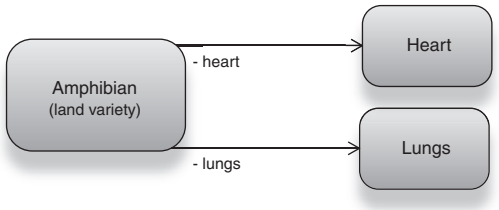

Figure 3.8 The land-dwelling variety of `Amphibian` depends on a `Heart` and `Lungs`.

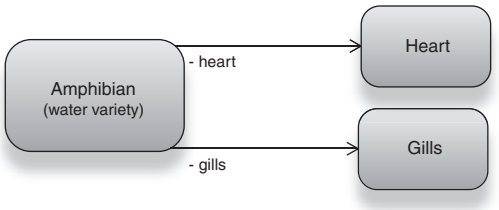

Figure 3.9 The aquatic variety of `Amphibian` depends on a `Heart` and `Gills`.

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

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.

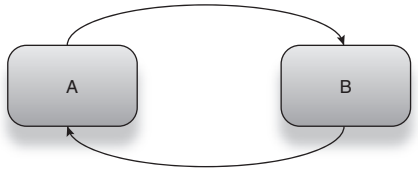

Figure 3.10 `A` and `B` are dependent on each other (circular dependency).

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

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).

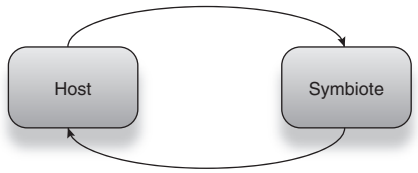

Figure 3.11 `Host` and `Symbiote` are circularly interdependent.

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).

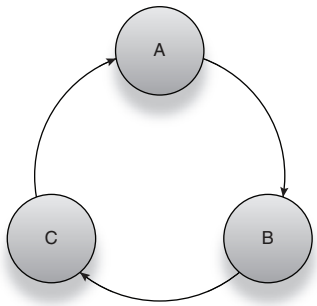

Figure 3.12 Triangular circularity: A depends on B, which depends on C, which depends on A.

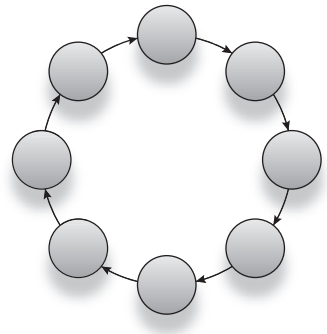

Figure 3.13 Any number of objects may be part of a circular dependency.

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:

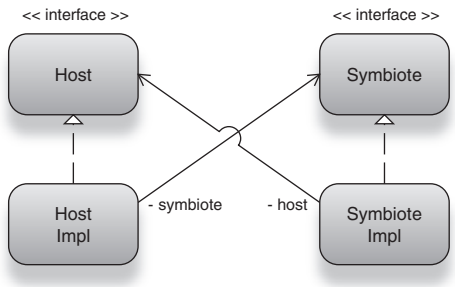

Figure 3.14 A class model for decoupled `Host` and `Symbiote` services (see listing 3.16)

```
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>
```

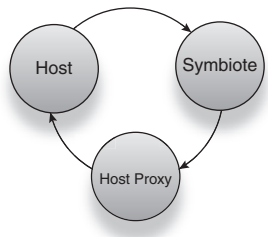

Figure 3.15 Injecting circular referents via a proxy

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);

    }
}
```

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.

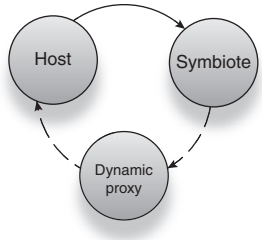

Figure 3.16 Injecting circular referents in Guice with a dynamic proxy

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 prove 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="unsafe1" 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"/>
```

```
        <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.

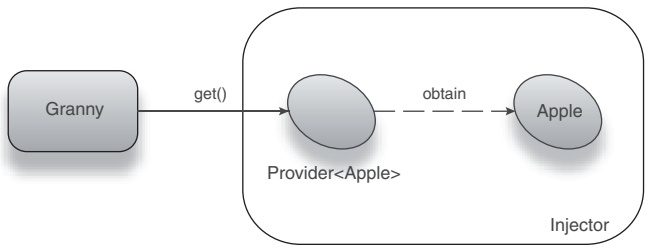

Figure 3.17 `Granny` obtains `Apple` instances from the injector via a provider

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).

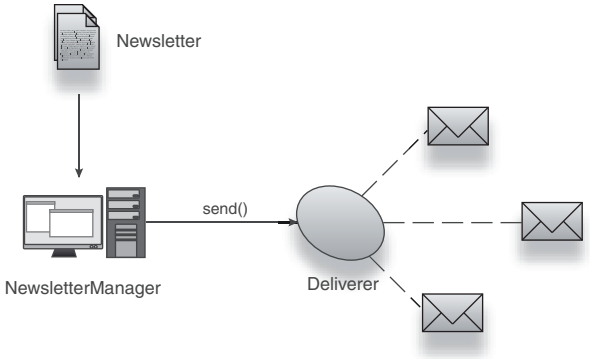

Figure 3.18 A `Deliverer` sends `Newsletters` out to recipients, on prompting by `NewsletterManager`

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.

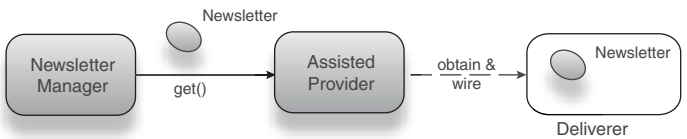

Figure 3.19 `AssistedProvider` creates a contextualized `Deliverer` with the current `Newsletter`.

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;

    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();

            d.deliverTo(recipient);
        }
     }
}

public class DelivererBuilder {
    private Newsletter letter;

    public void letter(Newsletter letter) {
        this.letter = letter;
    }

    public Deliverer buildDeliverer() {
        return new Deliverer(letter);
    }
}
```

Inject a Builder for dependency

Set context and build dependency

Setter method accepts context

Construct with context

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 DeliverерBuilder {
    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

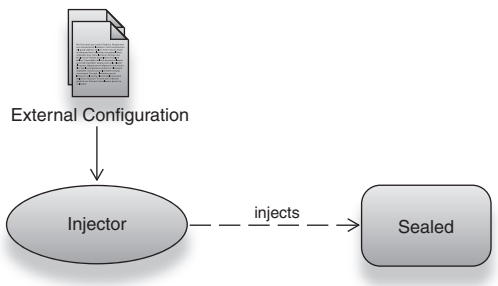

Figure 3.20 External metadata allows you to inject classes in sealed code easily.

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:

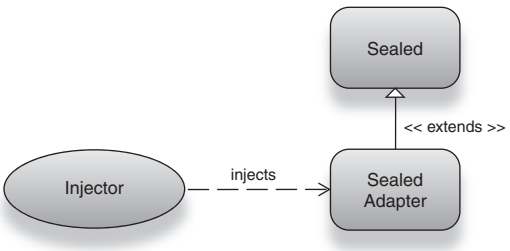

Figure 3.21 `SealedAdapter` helps `Sealed` to be injected transparently.

```
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);
    }
}
```

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.