Design patterns using Spring and Guice

Dependency Injection

Dhanji R. Prasanna

SAMPLE CHAPTER

MANNING

Dependency Injection

by Dhanji R. Prasanna

Chapter 7

brief contents

7 From birth to death: object lifecycle

This chapter covers:

- Notifying objects of significant events
- Understanding domain-specific lifecycle
- Initializing lazy and eager singletons
- Customizing lifecycle with multicasting

"I agree with everything you say, but I would attack to the death your right to say it."
—Tom Stoppard

Whether or not lifecycle is a part of dependency injection is a divisive issue. However, like scoping, it can be a powerful tool when used correctly, and it fits closely with dependency injection. In this chapter we'll look at the basic form of lifecycle offered by the language runtime—constructors. We'll also look at managed lifecycle as offered by more elaborate frameworks like servlets and EJBs, to illustrate how lifecycle is domain specific. On the way we'll examine the pitfalls of relying on one-size-fits-all lifecycle models.

Finally, we'll look at how to design and implement a custom lifecycle strategy and design classes that are simple and easy to test. First, let's start by looking at what lifecycle is: events in the life of an application that an object is notified about.

7.1 Significant events in the life of objects

Lifecycle and scope are closely related concepts, so much so that they are occasionally confused with each other. An object's longevity can be seen in these two dimensions. Object longevity with regard to some context (duration, users, threads, and so on) is its scope. Periods in this context indicate various things about an object, for instance, whether it is ready to begin serving clients, or if the object represents a network resource, whether that resource has been closed or abnormally interrupted.

Clearly many of these *states* are specific to the nature of the object. A movie may have a *paused* state in its life; a web application has a *deployed* state, a game has a *game-over* state, and so on. The various states an object goes through in its life are collectively known as its lifecycle. When an object transitions between such states, it is said to undergo lifecycle changes. These are often modeled as events that an object responds to (see figure 7.1).

The most basic lifecycle events are *create* and *destroy*.

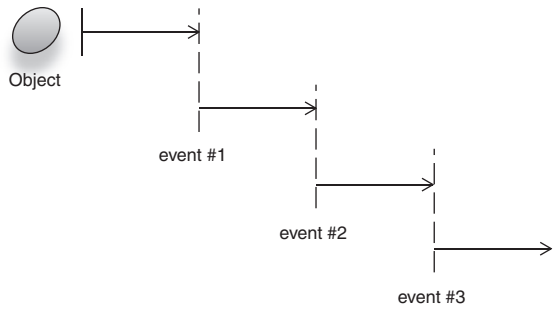

Figure 7.1 An object is notified of events as parts of its lifecycle.

7.1.1 Object creation

Objects come into instance when they are created and are notified about this event by a constructor. Language runtimes support this construct naturally, so this isn't really rocket science. After memory is allocated to an object, its constructor is called, which can perform some initialization logic. Ideally, one would do everything that's necessary on construction (any computation, dependency wiring, and the like) to put an object into a usable state. However, this isn't always possible for reasons that we'll discuss shortly.

In chapter 3 we saw how the constructor is used to wire an object with its dependencies. We saw that it is a powerful technique for creating objects that are good citizens. Good citizenry naturally extends to the lifecycle of an object. Constructing an object (with its dependencies) puts it in a usable state, and thus begins its lifecycle.

Listing 7.1's `Transporter` is notified via its constructor, so that it can prepare itself for use.

Listing 7.1 `Transporter`'s lifecycle is begun with a call to its constructor

```
public class Transporter {
    private final ControlPanel control;
    private final PowerCell cell;
```

```
    public Transporter(ControlPanel control, PowerCell cell) {
        this.control = control;
        this.cell = cell;

        cell.charge();
        control.activate();
}

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

Constructor is notified, begins lifecycle

Ready to be used

Figure 7.2 models these classes. In `Transporter`'s case, it calls into its dependencies first (as per figure 7.3).

Following this, `Transporter` is in the ready stage of its life. Method `energize()` can be called safely, to send crew members off on wild planetary adventures (see figure 7.4).

Furthermore, any preparatory computations are also a natural fit to the constructor:

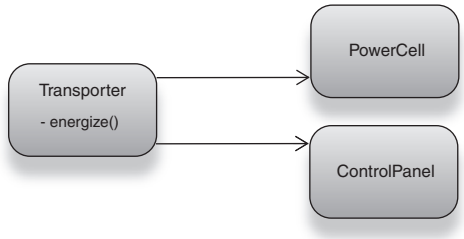

Figure 7.2 A `Transporter` depends on a `PowerCell` and `ControlPanel`.

```
public Transporter(ControlPanel control, PowerCell cell) {
    this.control = control;
    this.cell = cell;

    cell.charge();
    control.activate();
}
```

There's nothing spectacular about this example or about using constructors to prepare objects, but the example serves to illustrate the idea behind lifecycle with states

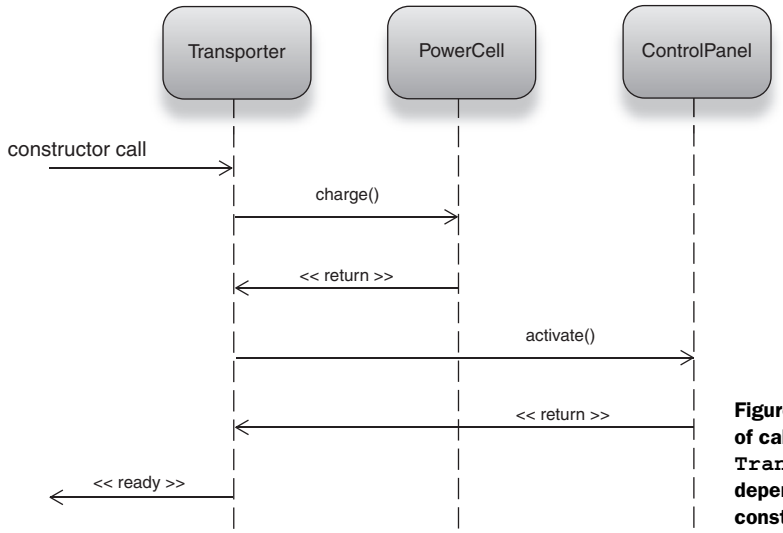

Figure 7.3 Sequence of calls to initialize the `Transporter` and its dependencies in a constructor

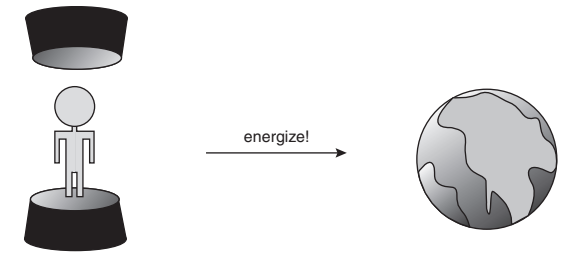

Figure 7.4 Once the constructor lifecycle method runs, we are ready for adventuring!

and about certain tasks that need to be performed to put an object into such states. Similarly, when a service is shut down, resources often need to be relinquished (memory reclaimed, sockets closed, and so on). This occurs when a service is destroyed.

7.1.2 Object destruction (or finalization)

In Java, the explicit task of allocating and reclaiming memory is taken care of by the runtime environment. This frees you from having to worry about when to reclaim allocated objects. Premanaged languages like C++ support an explicit *destructor*, which acts as a counterpart to constructors. The destructor is a lifecycle event method, which runs once, just prior to memory being freed. Java has something analogous: *finalizers*. A finalizer is a method that runs just prior to an object being reclaimed by the runtime garbage collector.

Unlike in C++, in Java we cannot guarantee when an object will be reclaimed and therefore when its finalizer runs. The only certainty is that a finalizer *will* be run before the object is reclaimed. This may be often and early (in the case of aggressive garbage collection), or rare and late, or even right at the shutdown of the application itself, or never if the application exits abnormally. For these reasons, disposal of finite resources (such as network sockets or database connections) inside a finalizer is not a good idea.

To illustrate why, let's take the example of a file service that displays the contents of various images in a directory (like a filmstrip preview). This service is described in listing 7.2.

Listing 7.2 This application displays impressions of images on a disk

```
public class Thumbnail {
    private final FileSystem fileSystem;
    private InputStream data;

    public Thumbnail(FileSystem fileSystem) {
        this.fileSystem = fileSystem;
    }

    public Image display(String path) throws IOException {
         data = fileSystem.getPath(path).newInputStream();
         ...
```

InputStream opened on display

```
    }

    @Override
    protected void finalize() throws Throwable {
        data.close();
        data = null;
        super.finalize();
    }
}
```

InputStream closed in finalizer (bad!)

Each time the user flips to a new image, the `Thumbnail` object opens a new file and displays it. However, the input stream is closed only inside its `finalize()` method. Since there is no guarantee of when an object may be reclaimed, it's a real possibility that the program will open too many files from the operating system and thus run out of resources (see figure 7.5). This is obviously a poor use of the finalizer lifecycle.

Most times, you will find a better lifecycle method to release finite resources than a finalizer. So, should you never use finalizers? Not quite—there are some rare cases when they come in handy. While you can't rely on a finalizer to release finite resources, you can use a finalizer to *ensure* that a resource has been released previously:

```
    @Override
    protected void finalize() throws Throwable {
        if (null != data) {
        logger.warning("unclosed thumbnail: " + this);
            data.close();
        }

        data = null;
        super.finalize();
    }
```

This is a sanity check that can help in cases where you mistrust clients of your code. Strictly speaking, this isn't common, but it's occasionally useful. Alternatively, you might use logging or profiling APIs inside a finalizer to test predictions about your program's performance.

One other use case for finalizers is releasing memory in collaborating libraries that are not managed. For example, memory resources of a native C library running inside a Java program can be released inside a finalizer by calling a native method (see figure 7.6).

These are corner cases that you are unlikely to encounter in everyday programming. However, it's important to understand the role of finalizers and more important to avoid *abusing* them. More on the pitfalls of finalization is presented in chapter 9.

Figure 7.5 Method `finalize()` may run very late, so it's unsuitable for closing finite resources.

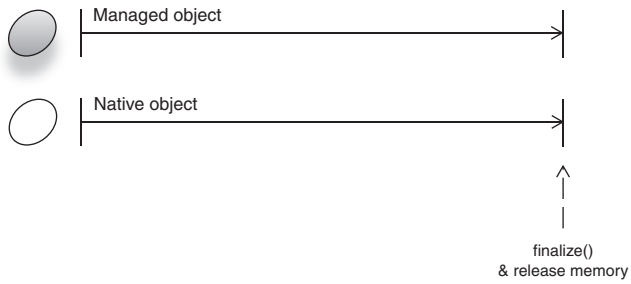

Figure 7.6 It's sometimes useful to release memory in collaborating native libraries with `finalize()`.

Construction and finalization are universal events that apply to any kind of object. But most types of lifecycle apply only to specific classes of objects—and then only in certain application domains. Remember that not all lifecycle is universal and that events are typically specific to particular problem domains.

7.2 One size doesn't fit all (domain-specific lifecycle)

We've seen that constructors are a lifecycle hook on creating objects and that finalizers are their complement prior to destruction. We've also seen that finalizers don't fit all clean-up use cases. Constructors are a bit better, but they too don't fit all initialization use cases. For example, a web application is deployed, and all its objects are constructed quite early. But it may only begin servicing requests at a much later time.

The application doesn't require many of its (finite) resources until that point. In the Java Servlet Framework, this directly translates to a servlet's `init()` method, where resources such as database connections can be acquired and held. Similarly, a network service may be created and configured but is not fully ready until a socket is opened.

This leads to the conclusion that lifecycle events are *not* universal and that the stages in an object's life are *specific* to a problem domain. For instance, the lifecycle of a servlet is very different from the lifecycle of a database connection or that of a movie player. Consequently, the times and frequency of initialization and destruction are dependent on the nature of the service. Web pages are created and destroyed frequently (on each request), while database connection pools are created once and held open almost indefinitely.

In this chapter, we'll look at some of these domains, how they differ, and how to design with them in mind. Let's start by contrasting two very common problem domains that have very different lifecycles: web servlets and database connections.

7.2.1 Contrasting lifecycle scenarios: servlets vs. database connections

A *servlet* is a web component used to render web pages that has three major stages in its lifecycle: *constructed*, *ready to service*, and *destroyed*. These are demarcated by two lifecycle events: *init* and *destroy*. A servlet's `init()` method is the lifecycle *hook* that's called to notify it of initialization. In other words, when the `init()` method returns, the servlet is expected to be ready for service. Similarly, method `destroy()` is called to notify a servlet that its life has ended. Any cleanup of the servlet's dependencies should be

performed here. Once destroyed, a servlet object is never called on to service requests again. Listing 7.3 shows a servlet object that initializes itself in its constructor and opens a connection to the database when notified of the init event. It releases this connection on servlet destruction. This sequence is illustrated in figure 7.7.

Listing 7.3 A Java servlet that starts and cleans up its resources inside lifecycle hooks

```
public class NewsServlet extends HttpServlet {
     private Connection con;
     private final NewsService newsService;

     public NewsServlet() {
         newsService = new NewsService();
     }

     @Override
     public void init() throws ServletException {
         try {
             con = DriverManager.getConnection(..);
         } catch (SQLException e) {
             ...
         }
     }

     @Override
     public void service(ServletRequest req, ServletResponse res) { .. }

     @Override
     public void destroy() {
         try {
             con.close();
             con = null;
         } catch (SQLException e) {
             ...
         }
     }
}
```

Create nonfinite dependencies

Open database connection

Close and release connection

There are some oddities about `NewsServlet`: Because it is managed directly by the servlet framework, it cannot benefit from dependency injection. Its lifecycle (that is, `init()` and `destroy()`) is managed entirely by the servlet container. It introduces the need to create dependencies by hand (or use a factory) and tightly couples the servlet to the underlying data layer. By inserting an integration layer like guice-servlet, we can change all that and have `NewsServlet` benefit from dependency injection as well as receive the servlet lifecycle events. Listing 7.4 reimagines `NewsServlet` in this light.

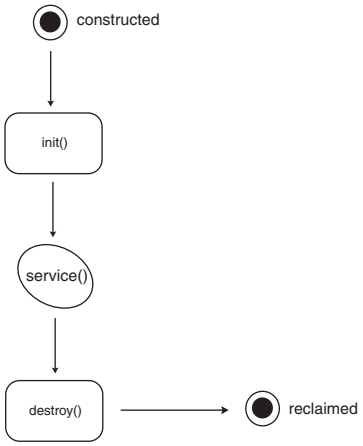

Figure 7.7 Stages in a servlet's lifecycle

Listing 7.4 A version of `NewsServlet` managed by Guice and guice-servlet

```
@Singleton
public class NewsServlet extends HttpServlet {
     private final PersistenceService persistence;
     private final NewsService newsService;

     @Inject
     public NewsServlet(PersistenceService persistence,
        NewsService newsService) {
         this.persistence = persistence;
         this.newsService = newsService;
     }

     @Override
     public void init() {
         persistence.start();
     }

     @Override
     public void service(ServletRequest req, ServletResponse res) { .. }

     @Override
     public void destroy() {
         persistence.shutdown();
     }
}
```

Abstract, nonfinite dependencies are injected

Start persistence service

Shut down persistence service

This version of `NewsServlet` is much neater, not simply because it is injected, but also because its lifecycle can naturally cascade to abstractions underneath. For example, `PersistenceService` may represent a database connection, flat disk file, or even in-memory network storage. Dependency injection in conjunction with the servlet lifecycle makes for simpler, more testable code.

Spring's `DispatcherServlet` provides a similar facility for routing requests from the servlet container to its own custom MVC framework. First you set up the `DispatcherServlet` in web.xml. Then you hook it up to a custom Spring MVC controller that does what you want (and is dependency injected by Spring):

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

    <bean name="/news.html" init-method="init" destroy-method="destroy"
➥   class="c7.NewsController">
      <constructor-arg ref="persistence"/>
      <constructor-arg ref="newsService"/>
    </bean>
    ...
</beans>
```

Here instead of implementing `HttpServlet` directly, we implement Spring's `Controller` interface:

```
import org.springframework.web.servlet.mvc.Controller;
...
public class NewsController implements Controller {
     private final PersistenceService persistence;
     private final NewsService newsService;

     public NewsController(PersistenceService persistence,
        NewsService newsService) {
         this.newsService = newsService; this.persistence = persistence;
     }
     public void init() {
         persistence.start();
     }

     public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) { .. }
     public void destroy() {
         persistence.shutdown();
     }
}
```

Start persistence service

Shut down persistence service

Except for the variant `handleRequest()` method, this is very similar to the previous example.

NOTE Notice that we had to specify `init-method` and `destroy-method` in the XML configuration; this is necessary since we're moving from the servlet container to Spring's injector. While these do not exactly coincide (as in the case of Guice Servlet), they perform the same function.

Contrast this with the lifecycle of a different service, such as a database connection. Both can be managed by DI, and both have a role to play in web applications, but they lead very different lives. Listing 7.5 describes a fictional, pooled database connection class.

Listing 7.5 `Connection` wraps a raw database-driver connection for pooling purposes

```
public class PooledConnection {
    private Connection conn;
    private ConnectionState state;

    public synchronized void open() throws SQLException {
        conn = DriverManager.getConnection(..);
    }

    public synchronized void onCheckout() {
        this.state = IN_USE;
    }

    public synchronized void onReturn() {
        this.state = IDLE;
    }

public synchronized void close() throws SQLException {
        if (IN_USE == state)
```

Connection is established

Connection is checked out for use

Connection is checked back in

Connection is disposed, if not in use

```
            throw new IllegalStateException();

        conn.close();
    }
}
```

In listing 7.5, there are four lifecycle hooks: `open()`, `close()`, `onCheckout()`, and `onReturn()`. These refer to events in the life of the database connection. `open` and `close` are self-explanatory; `onCheckout()` refers to the connection being taken out of its pool for use. Think of this as checking out a book from your local public library. When notified, the pool-aware connection puts itself in an `IN_USE` state, which is used to ensure that it isn't accidentally closed while serving data. Similarly, `onReturn()` is called when the connection is checked back into its pool, setting it into an `IDLE` state. Figure 7.8 is a flow chart describing the sequence of events in `Connection`'s lifecycle.

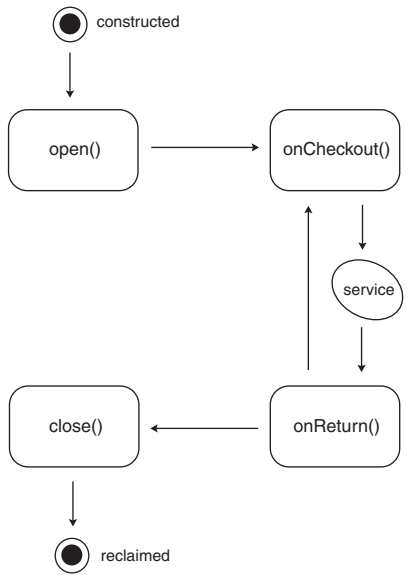

Figure 7.8 Flow of stages in an example database connection's lifecycle

Idle connections are safe to close or to perform other background activity on (for example, connection *validation*).

The connection's lifecycle hooks are called from an owning connection pool. As previously said, its usage profile is significantly different from a servlet's—and indeed any other object managed by dependency injectors. The pool itself may be managed by a servlet and cohere with its lifecycle; however, this is not particularly relevant to the connection's lifecycle.

Connection pooling and validation

Connections to RDBMS databases often have timeout values. After a certain period of inactivity, a driver automatically releases a held connection in order to prevent zombie connections from dragging down database performance. Since establishing new connections is potentially expensive, applications create and hold them in pools (at startup), taking the cost up front. Connections are then distributed from the pools to clients and returned after use, avoiding the need to dispose and reestablish connections during the life of busy applications. This is known as *connection pooling*.

If a connection remains idle in a pool for any length of time, it's in danger of being timed out. To prevent this, connection pools execute dummy SQL statements at periodic delays to reset the timer and keep the connection alive. Often these are as simple as `"SELECT 1 FROM DUAL"`, which returns the number `1` from a numerical sequence. This process is known as *connection validation*.

Just as it is difficult to fit connections and servlet into the same lifecycle, it is also difficult for any one lifecycle model to fit all use cases. Many programmers lead themselves into trouble by trying to do just that. One such anti-pattern is trying to apply a converse of the initialization lifecycle event to all services. I call this the destructor anti-pattern.

7.2.2 The Destructor anti-pattern

As we said earlier, finalizers are not well suited to freeing up finite resources in languages with managed memory models. To solve this, many developers use an all-purpose *destroy* lifecycle hook, which is called when the injector (or application itself) is shut down. Spring and PicoContainer provide these all-purpose destroy methods, which are called at some point in the application's lifecycle but not necessarily just when an object is being reclaimed. Here is the file service from earlier in the chapter, modeled using Spring's built-in destroy event:

```
<bean id="thumbnail" class="files.Thumbnail" destroy-method="close">
        <constructor-arg ref="fileSystem"/>
</bean>
```

The attribute `destroy-method=".."` refers to the name of a no-argument method to be called when the injector is shut down. While these are often useful in particular problem domains (such as the servlet lifecycle we saw previously), generalizing this event is not always appropriate. Take the `Thumbnail` example at the head of the chapter; the same problem occurs here as we saw with finalizers. Method `close()` is called when the application exits (Spring's injector calls destroy hooks when the JVM exits). This means that file handles will be held open while images are being viewed. Once again, you can very easily run out of finite OS resources without really using them.

A more sensible solution would be to close each file handle after the image preview has been extracted from it. There isn't a clear way to do that using only the `destroy-method` mechanism or, more generally, using the Destructor pattern.

Thus, indiscriminate use of this all-purpose lifecycle hook can lead to unintuitive designs. It's not uncommon to see Java Swing windows being provided a destructor. Often these are used to dispose of graphical widgets no longer needed within the window. The irony is that Swing widgets don't need to be disposed explicitly. Like any other Java object, Swing widgets are managed and reclaimed by the language runtime and require no explicit memory management.

Without a well-understood destruction hierarchy, it becomes difficult to predict the order in which objects are destroyed. For example:

```
<bean id="thumbnail" class="files.Thumbnail" destroy-method="close">
    <constructor-arg ref="fileSystem"/>
</bean>

<bean id="fileSystem" class="files.ZipFileSystem" destroy-method="close"/>
```

It's difficult to predict the order in which `ZipFileSystem.close()` and `Thumbnail.close()` will be called. If one depends on the other while shutting down, this may lead to accidental illegal states where a dependency has already been closed but a

dependent still needs it to shut itself down. It's not unusual to find conditions like the following, to account for just such a scenario:

```
public void close() {
    if (null != reader) {
        if (reader.isOpen())
            reader.someLastMinuteLogic();

        reader = null;
    }

    shutdownMyself();
}
```

Such contortions are abstruse and unnecessary, especially when we have managed memory models and patterns like DI to fall back on. In this light, I strongly discourage using the jack-of-all-trades destroy method or the Destructor anti-pattern. A good alternative is to pick domain-specific finalization patterns. For I/O, Java's `Closeable` interface functions nicely.

7.2.3 Using Java's Closeable interface

Not all objects can be left to garbage collection to be disposed of. Often, an external resource needs to be closed explicitly, at a time other than the application's exit, as we saw in the case of the thumbnail image viewer. Global, one-time destructors are ill-suited for the reasons we've just seen.

An object that acts as a data endpoint can be designed with *closeability* in mind if it exposes the `java.io.Closeable` interface. Here's a definition from the API documentation:

> *"A* `Closeable` *is a source or destination of data that can be closed. The close method is invoked to release resources that the object is holding (such as open files)."*

Modeling finite resources (files, network sockets, and so on) as `Closeable`s allows you to redact them into lifecycle stages. For example, a look-ahead caching service could open and read image data in the background. When they are finished being read, these files would be placed into a queue to be closed periodically by a lower-priority thread:

```
@ThreadSafe
public class ResourceCloser implements Runnable {
    @GuardedBy ("lock")
    public final List<Closeable> toClose = new ArrayList<Closeable>();

    private final Object lock = new Object();

    public void run() {
        synchronized(lock) {
            for (Closeable resource : toClose) {
                try {
                    resource.close();
                } catch (IOException e) {
                    ...
                }
```

```
                }
            }
        }

        public void schedule(Closeable resource) {
            synchronized(lock) {
                toClose.add(resource);
            }
        }
}
```

NOTE The annotation `@GuardedBy` indicates that access to mutable list `toClose` is guarded by a field named `lock`. Like `@ThreadSafe`, `@GuardedBy` is simply a documenting annotation and has no effect on program semantics.

Resources to be closed are enqueued via the `schedule()` method and closed periodically by the `run()` method. We won't worry too much about how this method is scheduled to run or how often. Let it suffice to say that it happens periodically and at a lower priority than other threads in the application. This satisfies the close lifecycle of a file handle and doesn't suffer the problems of destructors. So far, we've looked at how lifecycle is domain-specific and that while objects share common events like construction and finalization, they cannot be shoehorned into initialization and destruction patterns universally. To illustrate, let's examine a scenario that highlights how complex and specific lifecycle can get. Let's look at stateful EJBs, which are common in many real-world business applications.

7.3 A real-world lifecycle scenario: stateful EJBs

So far we've seen fairly simple lifecycle hooks in the form of constructors, some more involved lifecycles with servlets and pooled database connections, and some pitfalls in generalizing these models. Now let's take a look at a more complex case: *stateful EJBs*. EJBs are a programming model for business services. They are managed directly by an application server (commonly, a Java EE application server), and as such their lifecycle is also controlled by it. There are several types of EJBs, suited to various purposes such as persistence, message-passing, and business services. Stateful EJBs at a particular kind of business service are sometimes called *stateful session beans*.

These are basically objects that interact with a client around some specific context. They are called stateful because they can maintain this context across multiple interactions from the same client, in much the same manner as an HTTP session does with a browser client. As of this writing, the EJB specification is in version 3.0, which has some dependency injection features, where EJBs can be provided to one another via annotated fields. Similarly, they also have a controlled lifecycle, where the bean is notified of major events by the application server.

Listing 7.6 is an example of a stateful EJB representing a shopping cart. Its clients may be a website front end (like Amazon.com), a desktop application at a checkout counter, or even another EJB.

Listing 7.6 A stateful EJB representing a returning customer's shopping cart

```
@Remote
public interface ShoppingCart {
    void add(Item item);
    List<Item> list();
}

@Stateful
public class ShoppingCartEjb implements ShoppingCart {
    private List<Item> items = new ArrayList<Item>();
    private double discount;

    @EJB
    private InventoryEjb inventory;     <-- Injected by EJB container

    @PostConstruct
    public void prepareCart() {
        discount = inventory.todaysDiscount() * items.size();     <-- Compute initial discount
    }

    @Remove
    public Status purchase() {
        return inventory.process(items, discount);     <-- Process purchase and dispose EJB
    }

    public void add(Item item) { items.add(item); }
    public List<Item> list() { return items; }     <-- Shopping cart public methods
}
```

In listing 7.6's `ShoppingCartEjb`, all methods except `add()` and `list()` are EJB lifecycle hooks. Method `prepareCart()` is marked with `@PostConstruct`, indicating to the EJB container that it should be called on initialization. `purchase()` performs a dual role: It processes the order in the shopping cart, returning a status code, and also tells the EJB container to end the stateful bean's lifecycle. Another lifecycle hook, `@PreDestroy`, is available for any explicit cleanup actions that need to occur on a *remove* event.

NOTE The annotation `@EJB` is used in much the same manner as Guice's `@Inject` or Spring's `@Autowired` but only on a field. This tells the EJB container to look up the appropriate EJB and inject an instance. EJB provides basic dependency injection in this fashion. This is undesirable because of the inability to replace with mock objects and for several other reasons outlined in chapters 3 and 4 (EJB does not support constructor injection as of version 3.0).

This is certainly quite a complex set of semantics for a simple shopping cart service. But it gets better—because a stateful EJB is meant to keep a running conversation with its client (until `@Remove` is reached), two more lifecycle hooks are available. These are used to transition a stateful EJB from its active servicing state to a passive state (while the client is off doing other things) and then back again to an active state when the client has returned.

During this passive time, it may be useful to have the shopping cart data stored in a more permanent storage medium like a database or disk cache. EJB provides a `@PrePassivate` lifecycle hook for this purpose:

```
@PrePassivate
public void passivate() {
    cache.store(items);
}
```

This is useful if there is some kind of failure in the application server. On reactivation, you may want to refresh the items in the shopping cart or recompute discounts to ensure that associated data have not become stale. An administrator may have modified the item's price while the stateful EJB was passive. The `@PostActivate` lifecycle hook is called by an EJB container every time a stateful EJB is reactivated:

```
@PostActivate
public void activate() {
    items = inventory.refresh(items);
}
```

While the EJB programming model is not a direct analog of dependency injectors, it nonetheless serves to illustrate how complex, managed lifecycle can work. This entire flow is illustrated in figure 7.9.

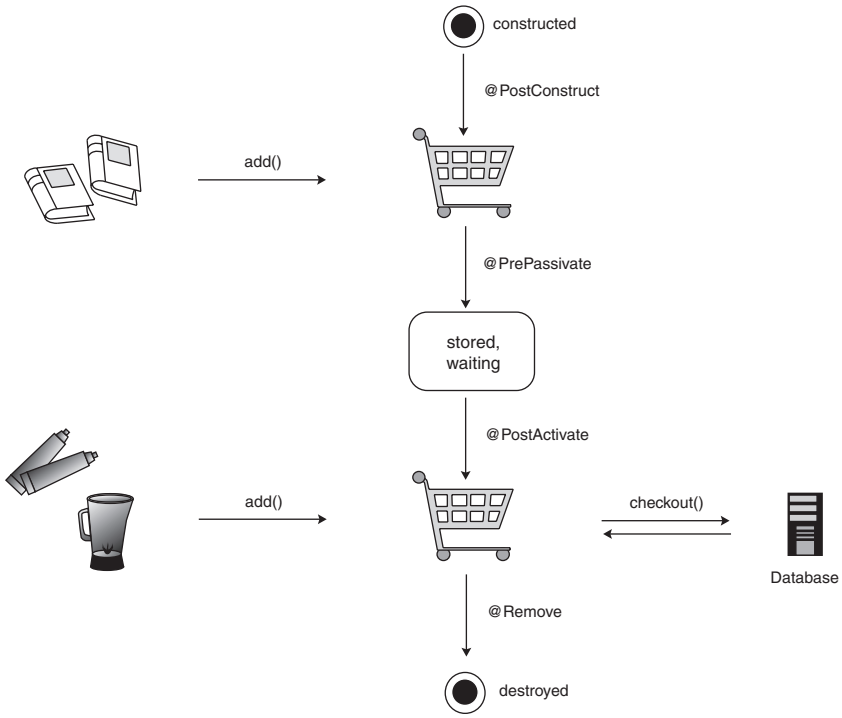

Figure 7.9 The lifecycle of a stateful EJB that models a shopping cart for *returning* clients

You will frequently find the need to apply similar patterns to your own environment, where objects are notified of significant events in their lifespan by an external actor or framework. Keep these examples in mind when designing lifecycle models for your own applications. Another important consideration when designing lifecycle is the timing of events. Not all objects are constructed at the same time. It is often desirable to delay construction until an object is needed. Among other things, this allows for better startup performance and lower memory usage. This type of delayed construction is called *lazy instantiation*.

7.4 Lifecycle and lazy instantiation

Another concept closely related to lifecycle is *on-demand* object construction. We saw this earlier with bootstrapping injectors on demand, per use. It's more common to find dependencies being constructed this way, particularly singleton-scoped objects. Rather than construct a dependency on injector startup, it's created when first needed for injection. In other words, it's created lazily (see figure 7.10).

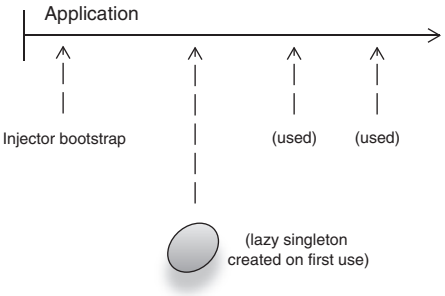

Figure 7.10 Lazily bound objects are created when first used.

Lazy creation of instances is useful when one is unsure whether all injector-managed services will be used early or used at all. If only a few objects are likely to be used early on and the remainder may be used infrequently or at a much later time, it makes sense to delay the work of creating them. This not only saves the injector from a lot of up-front cost but potentially also saves on the amount of memory used. It is especially useful during testing or debugging, where several configured services will probably never be used, or in a desktop or embedded application where fast startup is important.

The converse of lazy instantiation is *eager* instantiation. This is where singleton-scoped objects (in particular) are created along with the injector's bootstrap and cached for later use. Eager instantiation is very useful in large production systems where the performance hit can be taken up front, to make dependencies quickly available subsequently, as illustrated in figure 7.11.

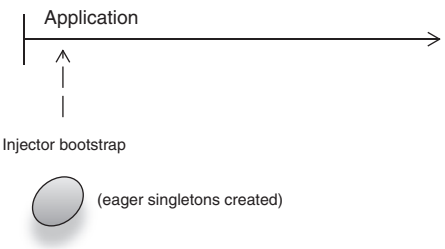

Figure 7.11 Eagerly bound singletons are created along with the injector itself.

The Guice injector can be built with a `Stage` directive, which tells it whether or not to eagerly instantiate singletons:

```
Guice.createInjector(Stage.PRODUCTION, new MyModule());
```

Singletons eagerly created

Or the alternative:

```
Guice.createInjector(Stage.DEVELOPMENT, new MyModule());
```

Singletons lazily created

You can also force certain keys to bind eagerly:

```
bind(MyServiceImpl.class).asEagerSingleton();
```

Or they can bind implicitly, by binding to a preexisting instance:

```
bind(MyService.class).toInstance(new MyServiceImpl());
```

Note that in the latter case, `MyServiceImpl` does not benefit from dependency injection itself, so this method should be avoided as much as possible. In Spring, this is achieved using the `lazy-init=".."` attribute:

```
<bean id="slothful" class="sins.seven.Sloth" lazy-init="true">
...
</bean>
```

Singleton instance `slothful` is created only when it is first needed by the application. By default, all singleton-scoped objects are created eagerly in Spring. Lazy instantiation as a design idiom is quite common. Before you decide to use it, you should consider issues of timing and performance, particularly during startup.

This chapter has thus far dealt with lifecycle events provided by frameworks like Spring and EJB. Now let's explore how we can create our own kinds of lifecycle. The simplest way to do this using a dependency injector is with post-processing.

7.5 Customizing lifecycle with postprocessing

As you've seen, lifecycle is specific to particular use cases and applications. In this case, the injector cannot help you directly, since libraries cannot ship with all possible lifecycle models (obviously). They can help, however, by providing a means for extending the lifecycle model to suit your own requirements.

Libraries like Spring and Guice allow an instance, once created, to be further processed by some general logic. This may include additional wiring, computation, or registering the instance for later retrieval. And it happens before the instance is injected on dependents. This is called *postprocessing* and is perfect for creating a custom lifecycle model (see figure 7.12).

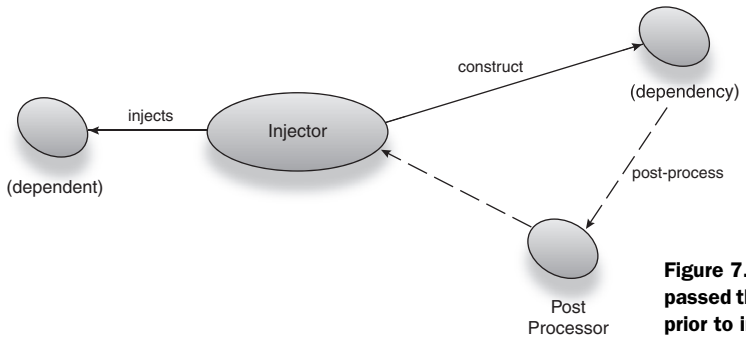

Figure 7.12 Dependencies are passed through the postprocessor prior to injection.

Let's take the example of a shopping arcade where there are several screens showing advertisements and shopping-related information. At midnight, when the arcade closes, these screens need to stop showing ads and display a notice saying that the mall is closed. Each screen's display is controlled from a separate set of feeds, based on their location. Modeling this as lifecycle of a feed (or screen) gives us a flexible and simple solution to the problem.

At midnight a notification is sent to all screens, putting them into a suspended state. Another notification is sent out the next morning when the mall reopens, so that normal programming can resume. Listing 7.7 shows how this `Screen` class would look in Java.

Listing 7.7 A screen with a timed lifecycle, driven by video feeds

```
package arcade;

public class Screen {
    private final Feed daytimeFeed;
    private final Feed overnightFeed;

    public Screen(Feed daytime, Feed overnight) { .. }

    public void suspend() {
        show(overnightFeed);
    }

    public void resume() {
        show(daytimeFeed);
    }

    ...
}
```

Class `Screen` has two interesting methods that embody its lifecycle:

- `suspend()`—Switches the screen to its overnight "mall closed" display
- `resume()`—Restores normal commercial programming

Both methods are specific to the shopping arcade problem domain. We need them to be called at specific times (midnight and early morning) to manage the activity of screens around the mall. The lifecycle sequence for this class is illustrated in figure 7.13.

If all instances of `Screen` are created by a dependency injector, it's possible to postprocess each instance and *register* it for subsequent lifecycle notification. Here's how a postprocessor might look in Spring, along with three independent `Screens` around the arcade:

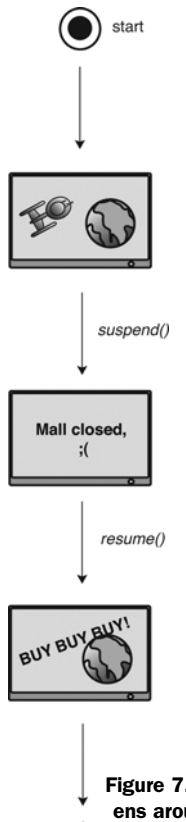

Figure 7.13 Screens around the mall move to a different feed when

```
<beans ...>
    ...
    <bean class="arcade.LifecyclePostProcessor"/>
    <bean id="screen.corridor" class="arcade.Screen">
        <constructor-arg ref="daytimeFeed"/>
        <constructor-arg ref="overnightFeed"/>
    </bean>
    <bean id="screen.foodcourt" class="arcade.Screen">
        <constructor-arg ref="daytimeFeed"/>
        <constructor-arg ref="overnightFeed"/>
    </bean>
    <bean id="screen.entrance" class="arcade.Screen">
        <constructor-arg ref="daytimeFeed"/>
        <constructor-arg ref="overnightFeed"/>
    </bean>
</beans>
```

And the postprocessor implementation, which registers available screens for lifecycle management, is as follows:

```
package arcade;

import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.beans.BeansException;

public class LifecyclePostProcessor implements BeanPostProcessor {
    private final List<Screen> screens = new ArrayList<Screen>();

    public Object postProcessAfterInitialization(
        Object object, String key) throws BeansException {

           if (object instanceof Screen)
               screens.add((Screen)object);

           return object;
    }

    public Object postProcessBeforeInitialization(
        Object object, String key) throws BeansException { .. }
}
```

To understand this, let's dissect the `BeanPostProcessor` interface:

```
package org.springframework.beans.factory.config;

import org.springframework.beans.BeansException;

public interface BeanPostProcessor {
    Object postProcessAfterInitialization(
        Object object, String key) throws BeansException;

    Object postProcessBeforeInitialization(
        Object object, String key) throws BeansException;

}
```

Classes exposing this interface must implement the postprocess methods. Their signatures are identical: Both take an instance to postprocess and the string key it is bound to. They also return an `Object`, which is expected to be the *postprocessed* instance. To

return the instance unaffected (that is, as per normal), you simply return the object that was passed in. For our purpose, this is exactly what we want. But first we register the instance in a collection of `Screens` to be used later by the lifecycle system:

```
public Object postProcessAfterInitialization(
    Object object, String key) throws BeansException {

        if (object instanceof Screen)
            screens.add((Screen)object);

        return object;
    }
```

Notice that we're only interested in instances of the `Screen` class. Subsequently, lifecycle events can be fired on a timer trigger to all screens as necessary:

```
public class LifecyclePostProcessor implements BeanPostProcessor {
    private final List<Screen> screens = new ArrayList<Screen>();

    public void suspendAllScreens() {    <-- Called at midnight, by timer
        for (Screen screen : screens)
            screen.suspend();            <-- Suspend each screen, in turn
    }

    public Object postProcessAfterInitialization(
        Object object, String key) throws BeansException { .. }

    public Object postProcessBeforeInitialization(
        Object object, String key) throws BeansException { .. }
}
```

Consider its complement, which is called by another timer, when the arcade reopens next morning:

```
    public void resumeAllScreens() {
        for (Screen screen : screens)
            screen.resume();
    }
```

This process can go on continuously every day, so long as the timer thread is in good health, and the screens will correctly switch feeds at the appropriate times. We didn't have to write a timer for each screen, nor did `Screens` have to keep track of the time themselves. Our single lifecycle manager was sufficient to push events out at the correct time to all `Screens`. This means not only less code to write but also far less code to *test* and fewer possible points of failure.

Custom post-processing is thus a powerful and flexible technique for creating lifecycle models suited to your application and its particular requirements. Another variant to building custom lifecycle is by multicasting a single method call to many recipients. This system has the advantage of being able to pass arbitrary arguments and even process returned values.

7.6 Customizing lifecycle with multicasting

Multicasting is very similar to what you just saw in section 7.5. Specifically, *multicasting* is the process of dispatching a single method call to multiple recipients. As such, it is

perfectly suited to sending lifecycle event notifications to objects managed by an injector. The primary difference between multicasting and the method we used with post-processors is that the framework takes care of broadcasting the event across instances in the injector. This is illustrated in figure 7.14.

This means that you can design your classes to be slightly more decoupled, with less effort. And you can model them according to the traits they embody in a lifecycle model. Listing 7.8 shows how the `Screen` would look if it were designed with multicasting in mind.

Listing 7.8 A screen with a timed lifecycle, designed with lifecycle traits

```
package arcade;

public class Screen implements Suspendable, Resumable {
    private final Feed daytimeFeed;
    private final Feed overnightFeed;

    public Screen(Feed daytime, Feed overnight) { .. }

    public void suspend() {
        show(overnightFeed);
    }

    public void resume() {
        show(daytimeFeed);
    }

    ...
}

public interface Suspendable {
    void suspend();
}

public interface Resumable {
    void resume();
}
```

Here `Suspendable` and `Resumable` are two *traits* that a `Screen` possesses. In other words, they are roles that the `Screen` can embody. This is very similar to the role interfaces that you saw with interface injection in chapter 3. Multicasting is supported in PicoContainer's Gems extension.[1] Listing 7.9 describes a lifecycle manager that uses multicasting to notify `Screens`.

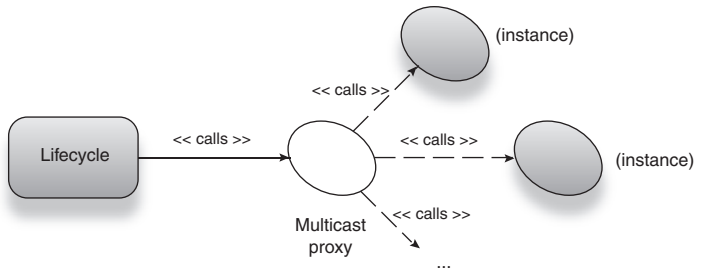

Figure 7.14 A multicast proxy promulgates a single lifecycle call across all managed instances.

[1] Gems are simple, intuitive extensions to the PicoContainer core framework. Find out more at: http://www.picocontainer.org.

Listing 7.9 Lifecycle manager that suspends and resumes services via multicasting

```
package arcade;

import com.thoughtworks.proxy.factory.StandardProxyFactory;
import org.picocontainer.gems.util.Multicaster;
import org.picocontainer.PicoContainer;

public class LifecycleManager {
    private final PicoContainer injector = ..;

    public void suspendAll() {
        Suspendable target = (Suspendable)
                    Multicaster.object(injector, true,
                    new StandardProxyFactory());

        target.suspend();
    }
}
```

First, we create a *multicasting proxy*. This is a dynamically generated implementation of `Suspendable` that transparently delegates its method calls across every instance of `Suspendable` available to the injector:

```
        Suspendable target = (Suspendable)
                    Multicaster.object(injector, true, new
                    StandardProxyFactory());
```

The first argument is the injector to inspect for `Suspendables`. The second tells the multicaster to proceed in the order in which `Suspendables` were originally created by the injector. The last argument is a factory utility, which is used to dynamically create the proxy implementation of `Suspendable`:

```
        Suspendable target = (Suspendable)
                    Multicaster.object(injector, true, new
                    StandardProxyFactory());
```

`StandardProxyFactory` simply uses the JDK tool `java.lang.reflect.Proxy` to create dynamic classes of type `Suspendable`.

This is cool because now the same lifecycle manager can suspend and resume *any* services that expose `Suspendable` and `Resumable`, not just `Screens`. And if the implementation logic of a screen changes (say you move the feed control out into a `Hub`), you don't have to worry about breaking the lifecycle model. Multicasting is thus an evolved form of customizing lifecycle and probably the best choice in most cases.

7.7 Summary

While lifecycle isn't an immediate part of dependency injection, the two are nonetheless closely related. Every object goes through a series of states from construction to destruction that are demarcated by lifecycle *events*. In certain applications objects can be notified of these events by a framework or assisting library.

A class's constructor is the most straightforward and ready form of lifecycle. It is called after memory has been allocated for the object and used to perform some initialization logic, to put the object into a usable state. Dependency injectors often use

constructors to provide objects with their dependencies. This is detailed further in chapter 3.

In Java, when an object is about to be reclaimed by the garbage collector, its *finalizer* is called. A finalizer is called at an arbitrary, uncontrollable time, in a separate thread. Finalizers may not run until an application shuts down, or not even then! As a result, finalizers are unreliable for performing cleanup of finite system resources such as file handles or network sockets. Finalizers may be useful in rare cases for sanity checks or for reclaiming memory used in native libraries.

Lifecycle is not a universal concept. Different applications have different requirements for the transitions in state of their services. Lifecycle is thus specific to particular *problem domains*. For example, servlets undergo a separate initialization step well after deployment, to put them into a service-ready state. When a web application is stopped, an explicit destroy event is sent to a servlet. This is very different from a database connection's lifecycle, which may include notifications to set it "idle" or "in use," so validations or disconnects can be performed safely.

Some DI libraries provide a *destroy-method* hook, which is called by the injector on application shutdown. This is unsuitable for most cases for the same reasons that finalizers are. And more specific lifecycle models should be sought, rather than attempt to use this "destructor" anti-pattern. Java's `Closeable` interface is a suitable alternative for releasing finite resources that are data sources or producers. This also indicates that a service is designed with *closeability* in mind, as opposed to being an afterthought as with destructors.

Stateful EJBs are a programming model that supports a very complex lifecycle model, where EJBs are notified by an application server (or EJB container) periodically. Stateful EJBs are akin to HTTP sessions or "conversations" (see chapter 5), where clients may resume a previous interaction with the EJB. These points of resumption and suspension are modeled as lifecycle events to the EJB: postactivate and prepassivate.

An object's lifecycle is also related to timing—singleton-scoped objects can be created immediately, upon injector bootstrap, or lazily, when first needed. Lazy instantiation is useful when startup time is important, such as in a desktop application or when debugging. Eager instantiation is the opposite form, where all singletons are created when the injector starts up. This is useful in production servers where a performance hit can be taken up front, if it means that services can be obtained faster during an application's active life.

Customizing lifecycle is important, and it is useful in many applications. Post-processing is an idiom where an object is passed to a post-processor, prior to being made available for injection. These instances can be inspected and held in a registry for later reference. When a notification needs to be sent, you simply iterate the registry and notify each instance in turn. Spring offers postprocessing via an interface `BeanPostProcessor`.

Lifecycle can also be customized using multicasting. This is very similar to the post-processing technique, except that the framework is responsible for promulgating

events to instances managed by the injector. A single method call on an interface is transparently multicast to eligible instances. This is useful in designing more decoupled services that are easier to test and more amenable to refactoring.

Custom lifecycle is a powerful and flexible idiom for reducing boilerplate code and making your code tighter. It allows you to design classes that are more focused and easier to test and maintain. Almost all problem domains have some kind of use for framework lifecycle management. In the next chapter, we'll look at modifying object behavior by intercepting method calls. This can be thought of as the converse of lifecycle, since it involves changing how objects react to their callers rather than propagating events down to them as this chapter showed. As you will see, method interception can be a powerful technique for achieving focused goals across a broad spectrum of services.

JAVA/SOFTWARE ENGINEERING

Dependency Injection DESIGN PATTERNS USING SPRING AND GUICE

Dhanji R. Prasanna

In object-oriented programming, a central program normally controls other objects in a module, library, or framework. With dependency injection, this pattern is inverted—a reference to a service is placed directly into the object which eases testing and modularity. Spring or Google Guice use dependency injection so you can focus on your core application and let the framework handle infrastructural concerns.

Dependency Injection explores the DI idiom in fine detail, with numerous practical examples that show you the payoffs. You'll apply key techniques in Spring and Guice and learn important pitfalls, corner-cases, and design patterns. Readers need a working knowledge of Java but no prior experience with DI is assumed.

What's Inside

- How to apply it (Understand it first!)
- Design patterns and nuances
- Spring, Google Guice, PicoContainer ...
- How to integrate DI with Java frameworks

Dhanji R. Prasanna is a Google software engineer who works on Google Wave and represents Google on several Java expert groups. He contributes to Guice, MVEL, and other open source projects.

"The most comprehensive coverage of DI that I have seen."
—Frank Wang, Chief Software Architect, DigitalVelocity LLC

"A handy manual for writing better programs with less code."
—Jesse Wilson
Guice 2.0 Lead, Google Inc.

"Dependency injection is not just for gurus—this book explains all."
—Paul King, Director, ASERT

"A fantastic book ... makes writing great software much easier."
—Rick Wagner, Enterprise Architect
Acxiom Data Products

"I am recommending this book to my staff."
—Robert Hanson, Manager
Applications Development
Quality Technology

For online access to the author, code samples, and a free ebook for owners of this book, go to manning.com/DependencyInjection

ISBN 13: 978-1-933988-55-9
ISBN 10: 1-933988-55-X
54999
9 781933 988559

MANNING $49.99 / Can $62.99 [INCLUDING eBOOK]