Design patterns using Spring and Guice

Dependency Injection

Dhanji R. Prasanna

SAMPLE CHAPTER

MANNING

Dependency Injection

by Dhanji R. Prasanna

Chapter 5

brief contents

5 Scope: a fresh breath of state

This chapter covers:

- Understanding what scope means
- Understanding singleton and no scope
- Applying practical scopes for the web

"Still this planet's soil for noble deeds grants scope abounding."

—Johann Goethe

In one sentence, *scope* is a fixed duration of time or method calls in which an object exists. In other words, a scope is a context under which a given key refers to the same instance. Another way to look at this is to think of scope as the amount of time an object's state persists. When the scope context ends, any objects bound under that scope are said to be *out of scope* and cannot be injected again in other instances.

State is important in any application. It is used to incrementally build up data or responsibility. State is also often used to track the context of certain processes, for instance, to track objects in the same database *transaction*.

In this chapter we'll talk about some of the general-purpose scopes: *singleton* and *no scope*. These are scopes that are universally applicable in managing state.

We'll also look at managing state in specific kinds of applications, particularly the web. Managing user-specific state is a major part of scoping for the web, and this is what the *request*, *session*, *flash*, and *conversation* scopes provide. We'll look at a couple of implementations of these with regard to Guice and Spring and how they may be applied in building practical web applications. First, we'll take a primer on scopes.

5.1 What is scope?

The real power of scope is that it lets you model the state of your objects declaratively. By telling the injector that a particular key is bound under a scope, you can move the construction and wiring of objects to the injector's purview. This has some important benefits:

- It lets the injector manage the *latent state* of your objects.
- It ensures that your services get new instances of dependencies as needed.
- It implicitly separates state by context (for example, two HTTP requests imply different contexts).
- It reduces the necessity for *state-aware* application logic (which makes code much easier to test and reason about).

Scope properly applied means that code working in a particular context is oblivious to that context. It is the injector's responsibility to manage these contexts. This means not only that you have an added separation between infrastructure and application logic, but also that the same services can be used for many purposes simply by altering their scopes. Take this example of a family bathroom and its toothpaste:

```
family.give("Joanie", injector.getInstance(Toothpaste.class));
family.give("Jackie", injector.getInstance(Toothpaste.class));
family.give("Sachin", injector.getInstance(Toothpaste.class));
```

Looking at this code we can say that the `Toothpaste` is used by Joanie first, then by Jackie, and finally by Sachin. We might also guess that each family member receives the same tube of toothpaste. If the tube were especially small, Sachin might be left with no toothpaste at all (as per figure 5.1).

This is an example of context: All three family members use the same bathroom and therefore have access to the same instance of `Toothpaste`. It is exactly the same as the following program, using construction by hand:

```
Toothpaste toothpaste = new FluorideToothpaste();

family.give("Joanie", toothpaste);
family.give("Jackie", toothpaste);
family.give("Sachin", toothpaste);
```

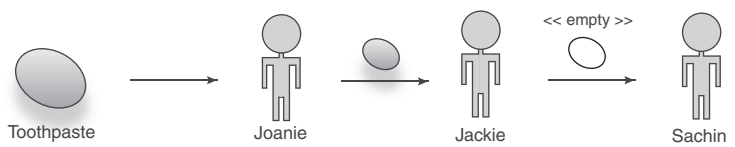

Figure 5.1 The injector distributes the same instance of `Toothpaste` to all family members.

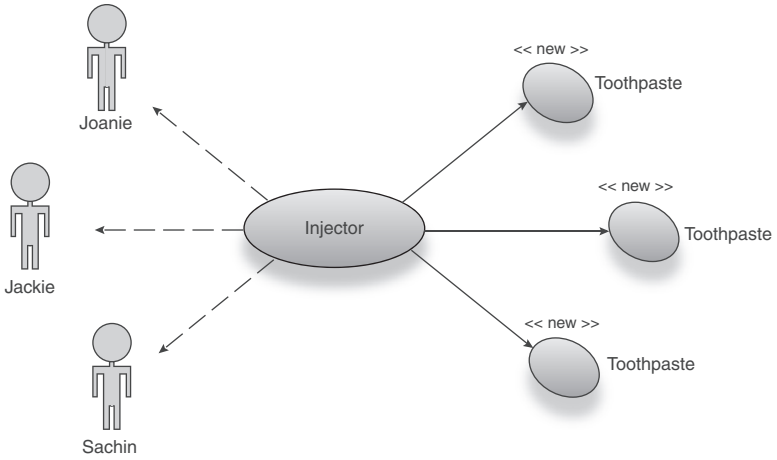

Figure 5.2 The injector creates a new `Toothpaste` instance for each family member.

If this were the whole life of the injector, only one instance of `Toothpaste` would ever be created and used. In other words, `Toothpaste` is bound under *singleton* scope. If each family member had his *own* bathroom (each with its own tube of toothpaste), the semantics would change considerably (figure 5.2).

```
family.give("Joanie", injector.getInstance(Toothpaste.class));
family.give("Jackie", injector.getInstance(Toothpaste.class));
family.give("Sachin", injector.getInstance(Toothpaste.class));
```

Nothing has changed in the code, but now a *new* instance of `Toothpaste` is available to each family member. And now there is no danger of Sachin being deprived of toothpaste by Joanie or Jackie. In this case, the context under which each object operates is unique (that is, its own bathroom). You can think of this as the opposite of singleton scoping. Technically this is like having no scope at all.

5.2 The no scope (or default scope)

In a sense, no scope fulfills the functions of scope, as it

- Provides new instances transparently
- Is managed by the injector
- Associates a service (key) with some context

Or does it? The first two points are indisputable. However, there arises some difficulty in determining exactly what context it represents. To get a better idea of no scope's semantics, let's dissect the example of the toothpaste from earlier. We saw that it took no change in the use of objects to alter their scope. The `family.give()` sequence looks exactly the same for both singleton and no scope:

```
family.give("Joanie", injector.getInstance(Toothpaste.class));
family.give("Jackie", injector.getInstance(Toothpaste.class));
family.give("Sachin", injector.getInstance(Toothpaste.class));
```

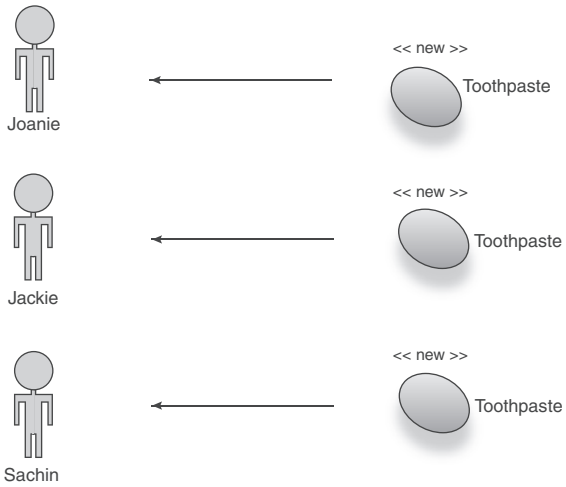

Figure 5.3 Each member of the family receives her own instance of `Toothpaste`.

Or, expanded using construction by hand (modeled in figure 5.3), the same code can be expressed as follows:

```
Toothpaste toothpaste = new FluorideToothpaste();
family.give("Joanie", toothpaste);

toothpaste = new FluorideToothpaste();
family.give("Jackie", toothpaste);

toothpaste = new FluorideToothpaste();
family.give("Sachin", toothpaste);
```

In no scope, every reference to `Toothpaste` implies a new `Toothpaste` instance. We likened this to the family having three bathrooms, one for each member. However, this is not exactly accurate. For instance, if Sachin brushed his teeth twice,

```
family.give("Joanie", injector.getInstance(Toothpaste.class));
family.give("Jackie", injector.getInstance(Toothpaste.class));
family.give("Sachin", injector.getInstance(Toothpaste.class));
family.give("Sachin", injector.getInstance(Toothpaste.class));
```

we would end up with a total of four `Toothpaste` instances (see figure 5.4).

In our conceptual model, there were only three bathrooms. But in practice there were four tubes of toothpaste. This means that no scope cannot be relied on to adhere to any conceptual context. No scope means that every time an injector looks

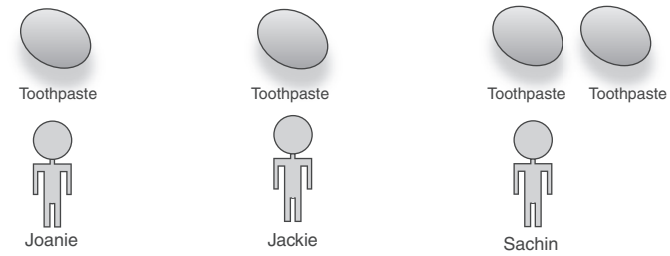

Figure 5.4 There are now four instances of `Toothpaste`, one for each use.

for a given key (one bound under no scope), *it will construct and wire a new instance.* Furthermore, let's say Sachin took Joanie on vacation and only Jackie was left at home. She would brush her teeth once, as follows:

```
family.give("Jackie", injector.getInstance(Toothpaste.class));
```

This would mean only one instance of `Toothpaste` was ever created for the life of the application. This was exactly what happened with singleton scoping, but this time it was purely accidental that it did. Given these two extremes, it is difficult to lay down any kind of strict rules for context with no scope. You could say, perhaps, that no scope is a split-second scope where the context is entirely tied to referents of a key. This would be a reasonable supposition. Contrast singleton and no scope in figure 5.5.

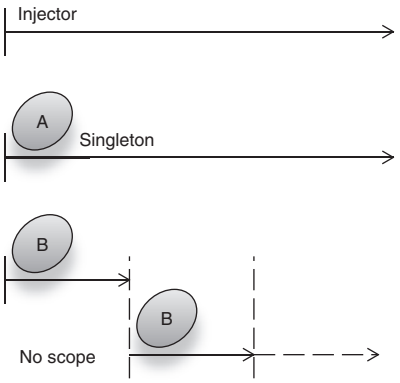

Figure 5.5 Timeline of contexts, contrasting singleton and no scope

No scope is a very powerful tool for working with injector-managed components. This is partly because it allows a certain amount of flexibility in scaling upward. Dependents that exist for longer times (or in *wider* scopes) may safely obtain no-scoped objects as they are required. If you recall the Provider pattern from chapter 4, there is a close similarity. `Granny` obtained new instances of an `Apple` on each use (see listing 5.1, modeled in figure 5.6).

Listing 5.1 An example of no scope using the Provider pattern

```
public class Granny {

    private Provider<Apple> appleProvider;

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

Two new Apples created

In listing 5.1, the `eat()` method uses a provider to retrieve new instances, just as we did for `Toothpaste`, earlier. Here `Apple` is no scoped.

Guice and Spring differ in nomenclature with regard to the no scope. Spring calls it as the *prototype* scope, the idea being that a key (and binding) is a kind of template

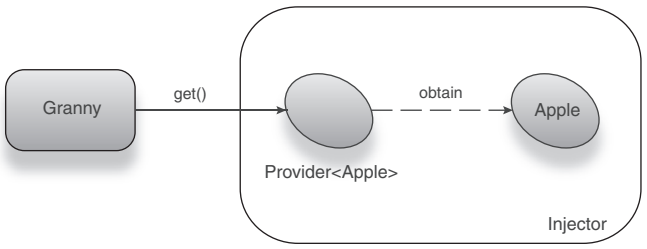

Figure 5.6 `Granny` obtains instances of `Apple` (bound to no scope) from a provider.

(or prototype) for creating new objects. Recall chapter 3, in the section on constructor injection and object validity, where no scoping enabled multiple threads to see independent instances of an object (modeled in figure 5.7):

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

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

This object was actually safe, because any dependents of key *safe* were guaranteed to see new, independent instances of `UnsafeObject`. Like singleton, the name prototype comes from the Gang of Four book, *Design Patterns*. For the rest of this book I will continue to refer to it as, largely because it is a more descriptive name.

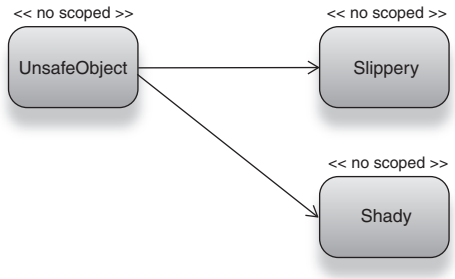

Figure 5.7 `UnsafeObject` and both its dependencies were no scoped.

Like Guice, PicoContainer also assumes the no scope if a key is not explicitly bound to some scope:

```
MutablePicoContainer injector = new DefaultPicoContainer();
injector.addComponent(Toothpaste.class);

family.give("Joanie", injector.getComponent(Toothpaste.class));
family.give("Jackie", injector.getComponent (Toothpaste.class));
...
```

There's almost no difference.

NOTE You will sometimes also hear no scope referred to as the *default* scope. This is a less descriptive name and typically connotes either Guice or PicoContainer (since they *default* to no scope).

While no scope doesn't really lend itself to a context, singleton scope does so quite naturally. Although the design pattern is itself applied in many different ways, we can establish a context quite easily for a singleton.

5.3 The singleton scope

Very simply, a singleton's context is the injector itself. The life of a singleton is tied to the life of the injector (as in figure 5.8).

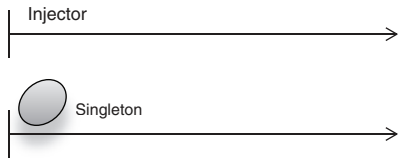

Figure 5.8 Timeline view of an injector and singleton-scoped object `A`

Therefore, only one instance of a singleton is ever created per injector. It is important to emphasize this last point, since it is possible for

multiple injectors to exist in the same application. In such a scenario, each injector will hold a different instance of the singleton-scoped object. This is important to understand because many people mistakenly believe that a singleton means one instance for the entire life of an application. In dependency injection, this is not the case. The distinction is subtle and often confusing. In fact, PicoContainer has dropped the term *singleton* and instead refers to it as *cache scoping*.

I persist with *singleton scope*, however, for a few reasons:

- A *singleton-scoped* object is different from a *singleton-patterned* object (more on this shortly).
- The term *singleton* is well known and reasonably well understood, even if its particular semantics are not.
- *Cache scoping* is a different concept altogether (which you will see in the next chapter).

Identifying which service should be a singleton is quite a divisive issue. It is a design decision that should impinge on the nature of a service. If a service represents a conceptual nexus for clients, then it is a likely candidate. For instance, a database *connection pool* is a central port of call for any service that wants to connect to a database (see figure 5.9). Thus, connection pools are good candidates for singleton scoping.

Similarly, services that are stateless (in other words, objects that have no dependencies or whose dependencies are immutable) are good candidates. Since there is no state to manage, no scoping and singleton scoping are both equally viable options. In such cases, the singleton scope has advantages over no scope for a number of reasons:

- Objects can be created at startup (sometimes called *eager instantiation*), saving on construction and wiring overhead.
- There is a single point of reference when stepping through a debugger.
- Maintaining the lone instance saves on memory (memory complexity is *constant* as opposed to *linear* in the case of no scoping; see figure 5.10).

Business objects are perfect candidates for singleton scoping. They hold no state but typically require data-access dependencies. These services are sometimes referred to as

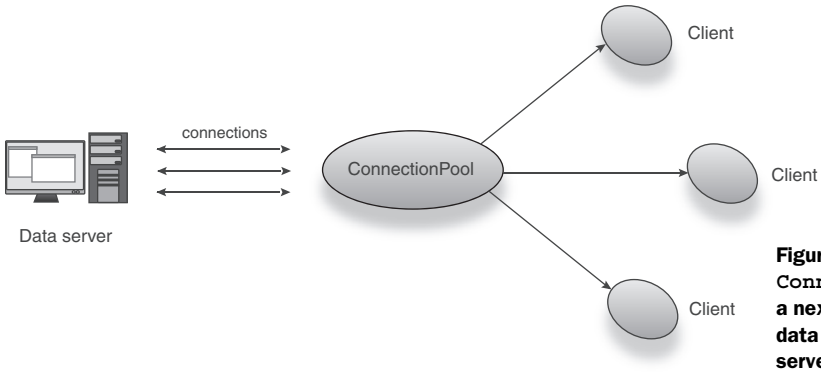

Figure 5.9 `ConnectionPool` is a nexus through which data connections are served to clients.

managers or simply business APIs, as in `PersonnelManager`, `SalaryManager`, and `StockOptionsService` in a hypothetical employee payroll system. Any services-oriented APIs are likewise good candidates to be stateless singletons. We'll talk a bit more about *services-oriented architecture* (SOA) in chapter 10.

Another oft-stated use case for singletons is when dealing with any object graphs that are expensive to construct and assemble. This may be due to any of the following reasons:

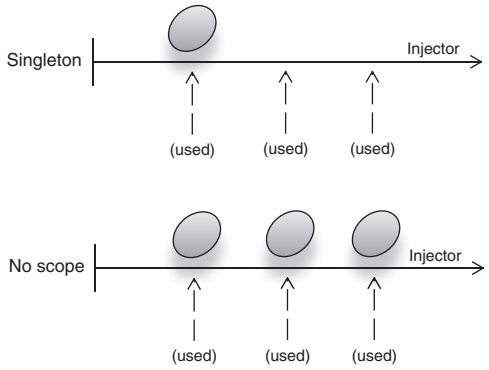

Figure 5.10 Memory usage for a singleton is constant compared to linear for no-scoped instances.

- The object graph itself being very large
- Reliance on slow external resources (like network storage)
- Some difficult computation performed after construction

These aren't good enough reasons in themselves to warrant singleton scoping. Instead you should be asking questions about context. If an external resource is designed to be held open over a long period, then yes, it may warrant singleton scoping, for example, the pool of database connections.

On the other hand, a TCP socket, while potentially expensive to acquire (and cheap to hold onto) may not warrant singleton scoping. If you were writing a game that logs in to a server over a TCP connection when playing against others, you certainly would not want it to be a singleton. If a user happened to log out and back in to a new server in a different network location, a new context for the network service would be established and consequently would need a new instance, not the old singleton instance.

Similarly, the size of the object graph should not play a major role in deciding an object's scope. Object graphs that have several dependencies, which themselves have dependencies, and so on, are not necessarily expensive to construct and assemble. They may have several cheaply buildable parts. So without seeing clear shortcomings in an object's performance, don't be in a rush to optimize it as a singleton. This is a mantra you can repeat to yourself every time you start to speculate and worry about performance in any context.

Computational expense may be a legitimate reason for scoping an object as a singleton, but here too, there are other prevailing concerns. Is the computation a one-time activity? Do its values never change throughout the life of the application? And can you separate the expense of computation from the object itself by, for instance, storing the resultant value in a *memo object*? If you can answer these questions, you may have a singleton on your hands. Remember the singleton litmus test: Should the object exist for the same duration as the injector? In the coming sections, we'll look at how the semantics of the singleton scope affect design in various situations.

Algorithmic (or asymptotic) complexity

Complexity in computing is a measure of the scalability of an algorithm with regard to resources. These resources may be CPU cycles (time complexity), RAM (memory complexity), or any other kind of physical resource that the algorithm may demand (for instance, network bandwidth). Complexity is used to indicate how the algorithm performs with increasing size of input. The input is typically expressed as a number n, indicating the total number of input items. For instance, a text-search algorithm may count the number of characters in a string as its input.

Performance is typically expressed in big Oh notation: $O(n)$ or $O(n^2)$, and so on. Everything is relative to n; the idea is to measure how an algorithm scales for very large values of n. Constant complexity, which is $O(1)$, indicates that the algorithm is independent of n, even for huge values of n. This is considered good because it means the algorithm will only ever consume a fixed amount of resources, regardless of its input.

Going back to the text-search algorithm, it is easy to see that its time complexity is dependent on the size of the input (since every character needs to be searched). This is called *linear complexity* and is usually written as $O(n)$.

In my example of no scoping, the input items are the number of *dependents* of the no-scoped dependency, and the amount of memory allocated scales in proportion to this number. Therefore, its memory complexity is also linear. If I changed to singleton scoping, the memory complexity would be constant, since only the one instance is created and shared by all dependents.

5.3.1 Singletons in practice

The important thing to keep in mind about scoping is that a key is bound under a scope, not an object or class. This is true of any scope, not just the singleton. Take the following example of a master terminal that can see several security cameras in a building:

```
<bean id="terminal" class="MasterTerminal" scope="singleton"/>

<bean id="camera.basement" class="SimpleCamera" scope="prototype">
    <constructor-arg ref="terminal"/>
</bean>

<bean id="camera.penthouse" class="SimpleCamera" scope="prototype">
    <constructor-arg ref="terminal"/>
</bean>
```

Notice that I explicitly declare `terminal` as a singleton (by default, Spring beans are all singletons) and both cameras as no scoped (`scope="prototype"`). In this configuration, both `camera.basement` and `camera.penthouse` share the same instance of `MasterTerminal`. Any further keys will also share this instance. Consider this equivalent in Guice, using a module:

```
public class BuildingModule extends AbstractModule {

    @Override
    protected void configure() {
bind(Camera.class).annotatedWith(Basement.class).
➥   to(SimpleCamera.class);
bind(Camera.class).annotatedWith(Penthouse.class).
➥   to(SimpleCamera.class);

        bind(MasterTerminal.class).in(Singleton.class);
    }
}
```

Here we use combinatorial keys (see chapter 2) to identify the basement and penthouse security cameras. Since Guice binds keys under no scope by default, we need only scope `MasterTerminal`:

```
    bind(MasterTerminal.class).in(Singleton.class);
```

This has the same semantics as the Spring configuration. All security camera instances share the same instance of `MasterTerminal`, as shown in figure 5.11.

Another option is to directly annotate `MasterTerminal` as a singleton:

```
@Singleton
public class MasterTerminal { .. }

public class BuildingModule extends AbstractModule {

    @Override
    protected void configure() {
bind(Camera.class).annotatedWith(Basement.class).
➥   to(SimpleCamera.class);
bind(Camera.class).annotatedWith(Penthouse.class).
➥   to(SimpleCamera.class);
                    }
}
```

This lets you skip an explicit binding (see `BuildingModule`). Figure 5.12 shows how a singleton-scoped instance is shared among no-scoped objects.

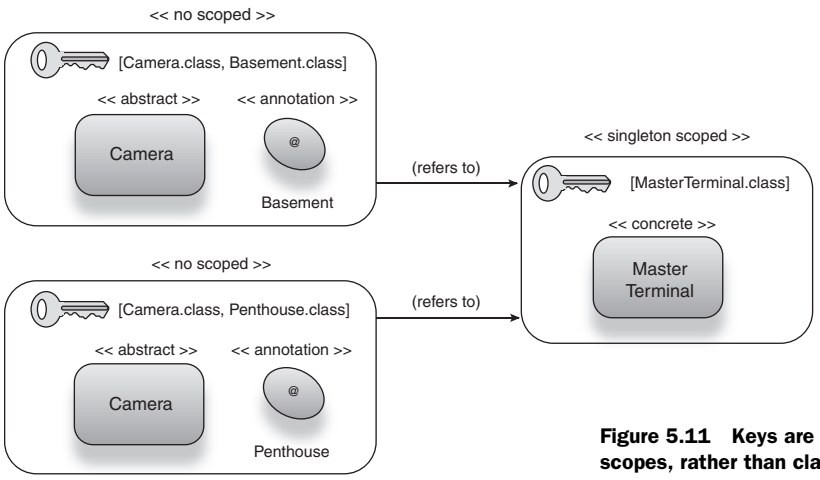

Figure 5.11 Keys are bound under scopes, rather than classes or objects

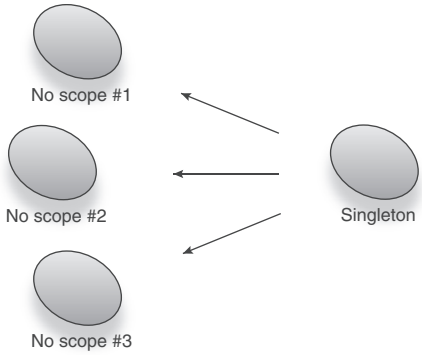

Figure 5.12 A singleton-scoped instance injected into many no-scoped instances

To further illustrate the difference between no scope and singleton scope, let's examine another interesting case. Here is a class `BasementFloor`, which represents a part of the building where security cameras may be installed:

```
public class BasementFloor {
    private final Camera camera1;
    private final Camera camera2;

    @Inject
    public BasementFloor(@Basement Camera camera1,
                         @Basement Camera camera2) {

        this.camera1 = camera1;
         this.camera2 = camera2;
    }
}
```

You might expect that both `camera1` and `camera2` refer to the same instance of security camera, that is, the one identified by key `[Camera, Basement]`. But this is not what happens—`camera1` and `camera2` end up with two different instances of `Camera`. This is because the key `[Camera, Basement]` is bound to no scope (see figure 5.13).

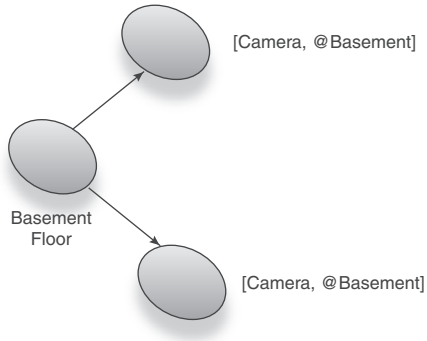

Figure 5.13 `BasementFloor` is wired with two separate instances of no-scoped key `[Camera, @Basement]`.

Similarly, any dependents of `[Camera, Penthouse]` will end up with new, unrelated instances of `Camera`. Consider another class, `Building`, which houses more than one kind of camera:

```
public class Building {
    private final Camera camera1;
    private final Camera camera2;
    private final Camera camera3;
    private final Camera camera4;

    @Inject
    public Building(@Basement Camera camera1,
```

```
                    @Basement Camera camera2,
                    @Penthouse Camera camera3,
                    @Penthouse Camera camera4) {

        this.camera1 = camera1;
        this.camera2 = camera2;
        this.camera3 = camera3;
        this.camera4 = camera4;
    }
}
```

Here, all four fields of `Building` receive different instances of `Camera` even though only two keys are present. This is opposed to the following class, `ControlRoom`, which has four fields that refer to the same instance of `MasterTerminal` but via four references (modeled in figure 5.14):

```
public class ControlRoom {
    private final MasterTerminal terminal1;
    private final MasterTerminal terminal2;
    private final MasterTerminal terminal3;
    private final MasterTerminal terminal4;

    @Inject
    public ControlRoom(MasterTerminal terminal1,
                MasterTerminal terminal2,
                MasterTerminal terminal3,
                MasterTerminal terminal4) {

        this.terminal1 = terminal1;
        this.terminal2 = terminal2;
        this.terminal3 = terminal3;
        this.terminal4 = terminal4;
    }
}
```

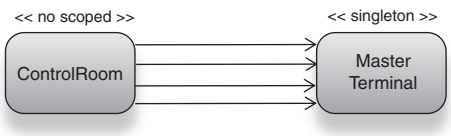

Figure 5.14 All four fields of `ControlRoom` are wired with the same instance of `MasterTerminal`.

If we added a new kind of `MasterTerminal`, bound to a different key, then this would be a different instance. Let's say I add a `MasterTerminal` for the basement only:

```
public class BuildingModule extends AbstractModule {

    @Override
    protected void configure() {
bind(Camera.class).annotatedWith(Basement.class).
➥   to(SimpleCamera.class);
bind(Camera.class).annotatedWith(Penthouse.class).
➥   to(SimpleCamera.class);

bind(MasterTerminal.class).annotatedWith(Master.class).in(Singleton.class);

        bind(MasterTerminal.class).annotatedWith(Basement.class)
            .to(BasementTerminal.class)
            .in(Singleton.class);

    }
}
```

Now any dependents of `[MasterTerminal, Master]` see the same shared instance, but any dependents of key `[MasterTerminal, Basement]` see a different instance. The modified version of `Building` in listing 5.2 illustrates this situation.

Listing 5.2 Object with two different singleton-scoped dependencies

```
public class ControlRoom {
    private final MasterTerminal terminal1;     | Shared instance of
    private final MasterTerminal terminal2;     | [MasterTerminal, Master]

    private final MasterTerminal terminal3;     | Shared instance of
    private final MasterTerminal terminal4;     | [MasterTerminal, Basement]

    @Inject
    public ControlRoom(@Master MasterTerminal terminal1,
                       @Master MasterTerminal terminal2,
                       @Basement MasterTerminal terminal3,
                       @Basement MasterTerminal terminal4) {

        this.terminal1 = terminal1;
        this.terminal2 = terminal2;
        this.terminal3 = terminal3;
        this.terminal4 = terminal4;
    }
}
```

In listing 5.2, `terminal1` and `terminal2` share the same instance of `MasterTerminal`. But `terminal3` and `terminal4` share a different instance. I belabor this point because it is very important to distinguish that a key, rather than an object, is bound under a scope. Singleton scoping allows many instances of the same class to exist and be used by dependents; it allows only one shared instance of a key. This is quite different from the conventional understanding of singletons. This idiom implies a more absolute single instance per application and is definitely problematic. In the following section I'll explain why I go so far as to call it an anti-pattern when compared with the much more erudite singleton scope.

5.3.2 The singleton anti-pattern

You have probably heard a lot of discussion around the web and in technical seminars about the horrors of the singleton as an anti-pattern. Earlier we drew a distinction between *singleton scope*, a feature of DI, and *singleton objects* (or singleton *anti-patterns*), which are the focus of much of this debate. The singleton anti-pattern has several problems. Compounding these problems is the fact that singletons are very useful and therefore employed liberally by developers everywhere.

Its problems however, greatly outweigh its usefulness and should warn you off them for good (especially when dependency injection can save the day with singleton scoping). Let's break this down in code:

```
public class Console {
    private static Console instance = null;

    public static synchronized Console getInstance() {
```

```
        if (null == instance)
            instance = new Console();

        return instance;
    }

    ...
}
```

I'm sure you have seen or written code like this. I certainly have. Its purpose is quite simple; it allows only one instance of `Console` ever to be created for the life of the program (representing the one monitor in a computer, for example). If an instance does not yet exist, it creates one and stores it in a static variable `instance`:

```
    if (null == instance)
        instance = new Console();

    return instance;
```

The static `getInstance()` method is declared synchronized so that concurrent threads don't accidentally attempt to create instances concurrently. In essence, `getInstance()` is a Factory method—but a special type of Factory that produces only one instance, thereafter returning the stored instance every time. You might say this is the Factory equivalent of singleton scoping.

Apart from all the foibles that Factories bring, note that this immediately causes one major problem. This code is not conducive to testing. If an object relies on `Console.getInstance()` to retrieve a `Console` instance and print something to it, there is no way for us to write a unit test that verifies this behavior. We cannot pass in a substitute `Console` in the following code:

```
public class StockTicker {
    private Console console = Console.getInstance();

    public void tick() {
        //print to console
        ...
    }
}
```

`StockTicker` directly retrieves its dependency from the singleton Factory method. In order to test it with a mock `Console`, you'd have to rewrite `Console` to expose a setter method:

```
public class Console {
    private static Console instance = null;

    public static synchronized Console getInstance() {
        if (null == instance)
            instance = new Console();

        return instance;
    }

    public static synchronized void setInstance(Console console) {
        instance = console;
```

```
    }

    ...
}
```

Patterns that force you to add extra code or infrastructure logic purely for the purposes of testing are poor servants of good design. Nonetheless, now you can test `StockTicker`:

```
public class StockTickerTest {

    @Test
    public final void printToConsole() {
        Console.setInstance(new MockConsole());
        ...
    }
}
```

But what if there are other tests that need to create their *own* mocked consoles for different purposes (say, a file listing service)? Leaving the mocked instance in place will clobber those tests. To fix that, we have to change the test again:

```
public class StockTickerTest {

    @Test
    public final void printToConsole() {
        Console previous = Console.getInstance();
        try {
            Console.setInstance(new MockConsole());
            ...
        } finally {
            Console.setInstance(previous);
        }
    }
}
```

I wrapped the entire test in a `try/finally` block to ensure that any exceptions thrown by the test do not subvert the `Console` reset.

TIP Depending on your choice of test framework, there may be other methods of doing this. I like to use *TestNG*,[1] which allows the declaration of setup and teardown methods that run before and after each test. See listing 5.3.

Listing 5.3 A test written in TestNG with setup and teardown hooks

```
public class StockTickerTest {
    private Console previous;

    @BeforeMethod
    void setup() {
```

[1] TestNG is a flexible Java testing framework created by Cedric Beust and others. It takes many of the ideas in JUnit and improves on them. Find out more about TestNG at http://www.testng.org and read Cedric's blog at http://beust.com/weblog.

```
        previous = Console.getInstance();
    }

    @Test
    public final void printToConsole() {
        Console.setInstance(new MockConsole());
        ...
    }

    @AfterMethod
    void teardown() {
        Console.setInstance(previous);
    }
}
```

This is a lot of boilerplate to write just to get tests working. It gets worse if you have more than one singleton dependency to mock. Furthermore, if you have many tests running in parallel (in multiple threads), this code doesn't work at all because threads may crisscross and interfere with the singleton. That would make all your tests completely unreliable. This ought to be a showstopper.

If you have more than one injector in an application, the situation grows worse. Singleton objects are shared even *between* injectors and can cause a real headache if you are trying to separate modules by walling them off in their own injectors (see figure 5.15).

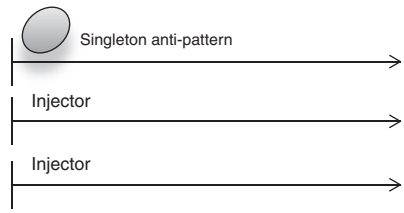

Figure 5.15 Singleton-patterned objects are shared even across injectors.

Moreover, any object created and maintained outside an injector does not benefit from its other powerful features, particularly lifecycle and interception—and the Hollywood Principle. One of the great benefits of DI is that it allows you to quickly bind a key to a different scope simply by changing a line of configuration. That's not possible with the singleton object (figure 5.16).

Its class must be rewritten and retested to introduce scoping semantics. Refactoring between scopes is also a vital part of software development and emerging design. Singleton objects hinder this natural, iterative evolution.

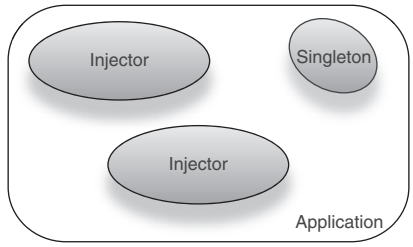

Figure 5.16 Singleton anti-pattern objects sit *outside* dependency injectors.

So to sum up: singleton objects bad, singleton scoping good. Singleton objects make testing difficult if not impossible and are antithetical to good design with dependency injection. Singleton scoping, on the other hand, is a purely injector-driven feature and completely removed from the class in question (figure 5.17).

Singleton scope is thus flexible and grants the usefulness of the Singleton anti-pattern without all of its nasty problems. Scopes have a variety of other uses; they

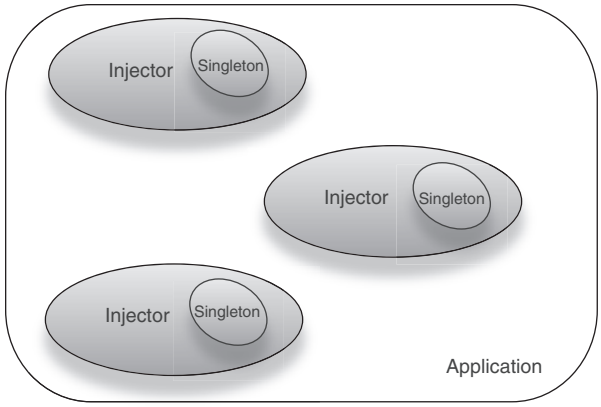

Figure 5.17 Singleton-scoped objects are well-behaved and live inside the injector.

needn't only be applied in the singleton and no scope idioms. In web applications, scopes are extremely useful as unintrusive techniques in maintaining user-specific state. These web-specific scopes are explored in the following section.

5.4 Domain-specific scopes: the web

So far we've seen that scopes are a context in which objects live. When a scope ends, objects bound to it go out of scope. And when it begins again, new objects (of the same keys) come "into scope." Scopes can also be thought of as a period in the objective life of a program where the state of objects is *persistent;* that is, it retains its values for that period. We also examined two of these scopes: the singleton and the no scope. They're rather unique: One sticks around forever, the other a split second. In a sense, these two scopes are universal. Any application has a use for no-scoped objects. And most applications will probably need access to some long-lived service that the singleton context provides.

But there is a whole class of uses that are very specific to a particular problem domain. These are *domain-specific* scopes. They are contexts defined according to the particular behavioral requirements of an application. For example, a movie theater has a specific context for each movie as it is being shown. In a multiplex, several movies may be showing simultaneously, and each of these is a unique context. We can model these contexts as scopes in an injector. A moviegoer watching a showing of the movie *The Princess Bride* is different from one who is watching *Godzilla* in another theater.

An important corollary to this model is that the moviegoer exists for the entire duration of the movie (that is, scope). So if you looked in on the *Godzilla* show, you would expect to see the same members of the audience each time.

The movie scopes are specific to the domain of movie theaters and intimately tied with their semantics. If a moviegoer exits one show just as it is ending and enters another show as it is starting (I used to do this in high school to save money), does it mean the moviegoer is carried across two contexts? Or should its state be lost and a new instance created? We can't answer these questions without getting deeper into

the movie theater analogy. More important, the answers to the questions can't be reused in any other problem domain.

One of the most important sets of domain-specific scopes is those around building *web* applications. Web applications have different contexts that emerge from interaction with users. Essentially, web applications are elaborate document retrieval and transmission systems. These are typically HTML documents, which I am sure you are infinitely familiar with. With the evolution of the web, highly interactive web applications have also arisen—to the point where they now closely resemble desktop applications.

However, the basic protocol for transmission of data between a browser and server has remained fairly unchanged. The contexts for a web application have their semantics in this protocol. Unlike a desktop application, a web application will generally have many users simultaneously accessing it. And these users may enter and leave a chain of communications with the web application at will. For example, when checking email via the popular Gmail service from Google, I sign in first (translated as a request for the inbox document), then open a few unread messages (translated as requests for HTML documents), and finally sign out. All this happens with Gmail running constantly on Google's servers. Contrast this with a desktop client like Mozilla Thunderbird, where I perform the same three steps, but they result in the program starting up from scratch and terminating when I've finished (entirely on my desktop). Furthermore, Google's servers host thousands (if not millions) of users doing very similar things simultaneously. Any service that would normally be singleton scoped, for example, my user credentials in Thunderbird, can no longer be a singleton in Gmail.[2] Otherwise, all users would share the same credentials, and we would be able to read each other's mail.

On the other hand, you can't make everything no scoped either. If you do, classes of the Gmail application would be forced to keep passing around user credentials in order to maintain state in a given request, and that would rewind all the benefits originally reaped from scoping. It would also perform rather poorly. Here's where web-specific scopes really help.

All interaction between a user (via a web browser) and a web application occurs inside an *HTTP request* (figure 5.18).

A *request* is—no surprise—a request for a document or, more generally, a resource. To provide this resource, an application might go to a database or a file or perform some *dynamic* computation. All this happens synchronously, that is, while the user is waiting. This entire transaction forms one HTTP request. Objects that live and operate within this context belong to the *request scope*.

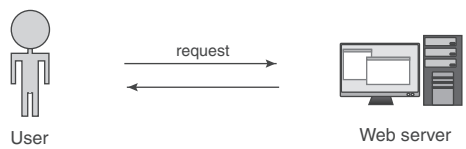

Figure 5.18 Interaction between a web server and a user happens in an HTTP request.

[2] Interesting fact: Gmail's frontend servers use Guice heavily for dependency injection concerns.

5.4.1 HTTP request scope

The request scope is interesting because it isn't strictly a segment in the life of an injector. Since requests can be concurrent, request scopes are also concurrent (but disparate from one another). Let's take the example of an online comic book store. *Sandman* is one of my favorite books, so I do a search for *Sandman* by typing in `"sandman"` at the appropriate page. To the comics store, this is a request for a document containing a list of *Sandman* titles. Let's call the service that generates this document `ComicSearch`. `ComicSearch` must

- Read my search criteria from the incoming request
- Query a database
- Render a list of results

Let's make another service that accesses the database and call this `ComicAccess`. `ComicAccess` will in turn depend on database-specific services like connections and statements in order to deliver results. Listing 5.4 describes the code for the comic store so far.

Listing 5.4 Comic store's search and data access components

```
public class ComicSearch {
    private final ComicAccess comicAccess;

    @Inject
    public ComicSearch(ComicAccess comicAccess) {
        this.comicAccess = comicAccess;
    }

    public HTML searchForComics(String criteria) {
        ...
    }
}
```

Now it makes sense for the `ComicAccess` object to be a singleton—it has no state, connects to a database as needed, and will probably be shared by several clients (other web pages needing to access the comic store). The `searchForComics()` method takes search criteria (typed in by a user) and returns an HTML object.

NOTE Of course, this is a hypothetical class and the exact form of `searchForComics()` may be different depending on the web framework you choose. But its semantics remain the same—it takes in search criteria and converts them to an HTML page displaying a list of matches.

`ComicSearch` is itself stateless (since its only dependency, `ComicAccess`, is immutable), so we could bind it as a singleton. Given this case, it actually works quite well. Since there is no *request-specific* context, binding `ComicSearch` either as a singleton or no scope is viable.

Let's expand this example. Let's say we add a requirement that the store shows me items of interest based on my prior purchases. It's not important how it determines my interests, just that it does. Another service, `UserContext`, will handle this work:

```
public class UserContext {
    private String username;

    private final ComicAccess comicAccess;

    @Inject
    public UserContext(ComicAccess comicAccess) {
        this.comicAccess = comicAccess;
    }

    public List<Comic> getItemsOfInterest() {
        ...
    }

    public void signIn(String username) {
        this.username = username;
    }
}
```

The method `getItemsOfInterest()` scans old purchases using `ComicAccess` and builds a list of suggestions. The interesting part about `UserContext` is its field `username` and method `signIn()`:

```
    public void signIn(String username) {
        this.username = username;
    }
```

When called with an argument, `signIn()` stores the *current* user. You'll notice that `signIn()` is more or less a setter method. But I've deliberately avoided calling it `setUsername()` to distinguish it from a dependency setter. `signIn()` will be called from `ComicSearch`, which itself is triggered by user interaction. Here's the modified code from listing 5.4, reflecting the change:

```
public class ComicSearch {
    private final ComicAccess comicAccess;
    private final UserContext user;

    @Inject
    public ComicSearch(ComicAccess comicAccess, UserContext user) {
        this.comicAccess = comicAccess;
        this.user = user;
    }

    public HTML searchForComics(String criteria, String username) {
        user.signIn(username);

        List<Comic> suggestions = user.getItemsOfInterest();
        ...
    }
}
```

`ComicSearch` is triggered on a request from a user, and method `searchForComics()` is provided with an additional argument, `username`, also extracted from the HTTP request. The `UserContext` object is configured with this username. Now any results it returns will be specific to the current user.

Let's put the `UserContext` to work and expand this another step. We'll add a requirement that the list of results should *not* show any comics that a user has already

purchased. One way to do this is with two queries, one with the entire set of results and the second with a history of purchases, displaying only the difference. That sequence would be:

1. Query all matches for criteria from database.
2. Query history of purchases for current user.
3. Iterate every item in step 1, comparing them to every item in step 2, and remove any matches.
4. Display the remainder.

This works, but it seems awfully complex. It is a lot of code to write and a bit superfluous. Furthermore, if these sets are reasonably large and contain a lot of overlap, it could mean doing a large amount of work to bring up results that are simply thrown away. Worse, two queries are two trips to the database, which is expensive and unnecessary in a high-traffic environment.

Another solution is to create a special *finder* method on `ComicAccess` that accepts the username and builds a database query sensitive to this problem. This is much better, because only the relevant results come back and the impact to `ComicSearch` is very small:

```
    public HTML searchForComics(String criteria, String username) {
        user.signIn(username);

        List<Comic> suggestions = user.getItemsOfInterest();

        List<Comic> results = comicAccess.searchNoPriorPurchases(criteria,
➥   username);
        ...
    }
```

But we can do one better. By moving this work off to `UserContext`, it avoids `ComicSearch` having to know the requisite details for querying comics:

```
public class UserContext {
    private String username;

    private final ComicAccess comicAccess;

    @Inject
    public UserContext(ComicAccess comicAccess) {
        this.comicAccess = comicAccess;
    }

    public List<Comic> getItemsOfInterest() {
        ...
    }

    public List<Comic> searchComics(String criteria) {
        return comicAccess.searchNoPriorPurchases(criteria, username);
    }

    public void signIn(String username) {
        this.username = username;
    }
}
```

Now `ComicSearch` is a lot simpler:

```
public HTML searchForComics(String criteria, String username) {
    user.signIn(username);

    List<Comic> suggestions = user.getItemsOfInterest();

    List<Comic> results = user.searchComics(criteria);
    ...
}
```

The real saving comes with request scoping. We already need to bind `ComicSearch` and `UserContext` in the request scope (because their state is tied to a single user's request). Now let's say that instead of signing in a user in the `searchForComics()` method, we're able to do it at the beginning of a request before the `ComicSearch` page (say, in a servlet *filter*). This is important because it means that authentication logic is separated from business logic. Furthermore, it means code to sign in the user need be written only once and won't have to litter every page:

```
public class ComicSearch {
    private final UserContext user;

    @Inject
    public ComicSearch(UserContext user) {
        this.user = user;
    }

    public HTML searchForComics(String criteria) {
        List<Comic> suggestions = user.getItemsOfInterest();
        List<Comic> results = user.searchComics(criteria);

        ...
    }
}
```

Notice that it is much leaner now and focused on its core purpose. But where has the code for signing in a user gone? Here's one possible way it may have disappeared.

REQUEST SCOPING IN GUICE WITH GUICE-SERVLET

Listing 5.5 shows one implementation using a Java servlet filter and the guice-servlet extension library for Guice.

> **Guice Servlet and Guice**
>
> Guice servlet is an extension to Guice that provides a lot of web-specific functionality, including web-domain scopes (request, session). Guice servlet is registered in web.xml as a filter itself and then later configured using a Guice module. It allows you to manage and intercept servlets or filters via Guice's injector (which is not otherwise possible).
>
> Find out more about guice-servlet at http://code.google.com/p/google-guice.

Listing 5.5 An injector-managed servlet filter using guice-servlet (using Guice)

```
import javax.servlet.Filter;

@Singleton
public class UserFilter implements Filter {
    private final Provider<UserContext> currentUser;

    @Inject
    public UserFilter(Provider<UserContext> currentUser) {
        this.currentUser = currentUser;
     }

    public void doFilter(ServletRequest request, ServletResponse response,
         FilterChain chain) throws IOException, ServletException {

        currentUser.get().signIn(...);      <-- Set up current user

        chain.doFilter(request, response);      <-- Continue processing request
     }

    ...
}
```

There are some interesting things to say about listing 5.5:

- `UserFilter` is a servlet filter applied at the head of every incoming request.
- It is declared under singleton scope (note the `@Singleton` annotation).
- It is injected with a `Provider<UserContext>` so that a *request-scoped* `UserContext` may be obtained each time.
- User credentials are extracted from the request and set on the current `UserContext`.

I use a `Provider<UserContext>` instead of directly injecting a `UserContext` because `UserFilter` is a singleton, and once a singleton is wired with any object, that object gets held onto despite its scope. This is known as *scope-widening injection* and is a problem that I discuss in some detail in the next chapter.

Interestingly enough, in listing 5.4 I was able to use constructor injection to get hold of the `UserContext` provider:

```
@Inject
public UserFilter(Provider<UserContext> currentUser) {
    this.currentUser = currentUser;
 }
```

Ordinarily, this wouldn't be possible for a filter registered in web.xml according to the Java Servlet Specification. However, guice-servlet gets around this by sitting between Java servlets and the Guice injector. Listing 5.6 shows how this is done, with a web.xml that is configured to use guice-servlet.

Listing 5.6 web.xml configured with guice-servlet and Guice

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
```

```
        xmlns="http://java.sun.com/xml/ns/j2ee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
      http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" >

    <listener>
        <listener-class>example.MyGuiceCreator</listener-class
    </listener>

    <filter>
        <filter-name>guiceFilter</filter-name>
        <filter-class>com.google.inject.servlet.GuiceFilter</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>guiceFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

</web-app>
```

web.xml namespace boilerplate

Listener creates injector on web app deploy

Filter all URLs through guice-servlet

Guice-servlet's architecture is depicted in figure 5.19.

Notice that in listing 5.6 a servlet context listener named `MyGuiceCreator` is registered. This is a simple class that you create to handle the job of bootstrapping the injector. It is where you tell guice-servlet what filters and servlets you want the Guice injector to manage. Here's what a `MyGuiceCreator` would look like if it were configured to filter all incoming requests with `UserFilter` (to do our authentication work):

```
public class MyGuiceCreator extends GuiceServletContextListener {

    @Override
    protected Injector getInjector() {
        return Guice.createInjector(new ServletModule() {
@Override
            protected void configureServlets() {
                filter("/*").through(UserFilter.class)

            }
        });
    }
}
```

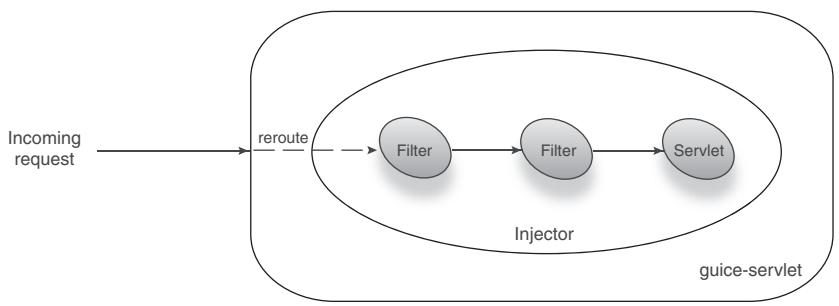

Figure 5.19 Incoming requests are rerouted by guice-servlet to injector-managed filters or servlets.

This code is self-explanatory, but let's go through it anyway. Remember, `MyGuiceCreator` is a class you provide to bootstrap and configure the injector (it must extend `GuiceServletContextListener`). The factory method `Guice.createInjector()` takes instances of Guice's `Module` as argument. You configure filters in guice-servlet via a programmatic API (rather than web.xml):

```
filter("/*").through(UserFilter.class)
```

You could continue adding filters and servlets. Each servlet and filter must be annotated `@Singleton`.

Let's get back to request scoping. By moving code out to the `UserContext` object, every request receives its own instance of `ComicSearch` and `UserContext`. We were able to achieve this transition without any impact to `ComicAccess` and minimal impact to `ComicSearch`. Declarative scoping of objects is thus a very powerful and unintrusive technique.

This is all very well. But how do we actually bind these scopes in Spring, Guice, and others? Let's take a look:

```
import com.google.inject.servlet.RequestScoped;

public class ComicStoreModule extends AbstractModule {

    @Override
    protected void configure() {

    bindComicAccess.class).to(ComicAccessImpl.class).in(Singleton.class);

        bind(UserContext.class).in(RequestScoped.class);
        bind(ComicSearch.class).in(RequestScoped.class);
        ...
    }
}
```

In the example code, I've bound both `UserContext` and `ComicSearch` in `@RequestScoped`. This is a scope made available by guice-servlet and represents the HTTP request scope in the world of Guice.[3] `ComicAccess` is a simple singleton, and so it is a straightforward binding (to its implementation, `ComicAccessImpl`):

```
bind(ComicAccess.class).to(ComicAccessImpl.class).in(Singleton.class);
```

This same effect can be achieved in Spring with its own set of web-specific scoping utilities. The following section explores the techniques involved there.

REQUEST SCOPING IN SPRING

In Spring's XML configuration mode, these bindings are slightly different (see listing 5.7).

Listing 5.7 Spring XML configuration for the online comic store's components

```
<beans ...>
    <bean id="data.comics" class="ComicAccessImpl" scope="singleton">
```

[3] Don't forget that you need to register guice-servlet's `GuiceFilter` in web.xml, as shown previously.

```
    ...
    </bean>

    <bean id="web.user" class="UserContext" scope="request">
        <constructor-arg ref="data.comics"/>
    </bean>

    <bean id="web.comicSearch" class="ComicSearch" scope="request">
        <constructor-arg ref="data.comics"/>
        <constructor-arg ref="web.user"/>
    </bean>
</beans>
```

The only new thing in listing 5.7 is the attribute `scope="request"` on bindings `web.user` and `web.comicSearch`. Of course, none of the three classes need change at all. Like Guice and guice-servlet, Spring requires additional configuration in web.xml. First off, you need to bootstrap a Spring injector when the web application is deployed. In Guice we used a `ServletContextListener` (recall `MyGuiceCreator`, the subclass of `GuiceServletContextListener`). You do this in Spring too:

```
<web-app ...>
    ...
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/comic-store.xml</param-value>
    </context-param>

    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    ...
</web-app>
```

This web.xml is similar to the one we saw earlier with guice-servlet. Notice that we needed to set a context parameter with the name of the XML configuration file. You can think of it as the web equivalent of the following:

```
BeanFactory injector = new FileSystemXmlApplicationContext("WEB-INF/comic-
➥ store.xml");
```

Now we've bootstrapped the injector and told it where to find its configuration. But that's not all; as guice-servlet did for Guice, there's still an integration layer that needs to be configured to get request scopes going:

```
<web-app ...>
    ...
    <listener>
        <listener-class>
            org.springframework.web.context.request.RequestContextListener
        </listener-class>
    </listener>
    ...
</web-app>
```

This listener must appear in addition to the `ContextLoaderListener` shown previously. Now you're set up to shop comics with Spring.

TIP We haven't quite looked at how to configure filters with Spring's injector. Guice-servlet handled this for us via its `GuiceFilter`. Spring does not have this support out of the box. But a sister project, *Spring Security* (formerly *Acegi Security*[4]) does. Spring Security's `FilterToBeanProxy` does a similar job.

Apart from these practical aspects, there are things to keep in mind about request scoping:

- A thread is typically dedicated to a request for the request's entire span. This means that request-scoped objects are not multithreaded.
- It also means that they are generally not thread-safe.
- Integration layers that provide request scoping often cache scoped objects in thread-locals.
- In rare cases, web servers may use multiple threads to service a request (for instance, while processing long-running asynchronous requests). If you are designing a request-scoping library in such a scenario, be aware of thread-local assumptions.

Most of these are pretty low-level and specific to the architecture in question. In the Java servlet world, threads and requests are almost always the same thing (though once completed, a thread may clear its context and proceed to service other requests). So you should be careful to clean up at the end of a request. If you are *designing* request scopes, you should carefully research these potential hazards. Perhaps even more useful than the request scope is the HTTP session scope. This technique allows you to keep objects around between requests in a semantic user session. Using dependency injection, a lot of the glue code to make this happen goes away. Session scoping is thus a powerful tool in the dependency injector's toolbox.

5.4.2 HTTP session scope

An HTTP session scope is the next step up from a request scope. HTTP sessions are an abstraction invented to make up for the fact that the HTTP protocol is stateless. This means that it does not easily allow for long-running interactions with a user to be maintained on the server side. To get around this, developers use clever techniques and string together a series of requests from the same user and call it a session (figure 5.20). An HTTP session has important characteristics:

- A session represents a single, unique user's interaction with a web application.
- A session is composed of one or more requests from the same user.
- Not all requests are necessarily part of a session.
- A session is a kind of store that preserves state between requests.
- The two logical end points of a session are user login and logout.

[4] Find out about Acegi Security at http://www.acegisecurity.org.

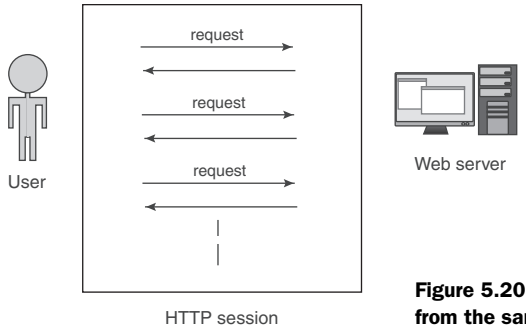

Figure 5.20 A series of related requests from the same user forms an HTTP session.

You can visualize a session as made up of multiple independent requests from the same user (as in figure 5.20). In figure 5.21, each instance of a request-scoped object is unique to that request (R1 to R3). But a *session-scoped* instance is *shared* across all those requests.

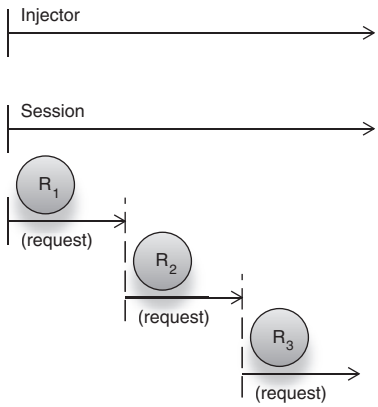

Figure 5.21 A session is composed of independent requests from a user.

Sessions are extremely useful for tracking state relative to a specific user, since a session always exists around one user. These uses include:

- Tracking a user's credentials for security purposes
- Tracking a user's recent activity for quick navigation (for example, breadcrumbs)
- Tracking preferences, to personalize a user's experience
- Caching user-specific information for quick access
- Caching general, constant data for quick access

All these use cases involve storing state temporarily, generally to improve a user's experience through the site. Whether that is about presentation or under the covers, it's about performance. And that's essentially what sessions do. Objects scoped under a session retain their state across requests, essentially continuing from where the last request left off.

Typically, sessions start and end when a user logs in and logs out. This behavior may be customized as necessary. Some requests (for static content, for example) do not participate in a user session and are considered stateless. These requests are independent of sessions, and, generally speaking, services participating in them shouldn't have any user-specific functionality.

Like requests, sessions may also be concurrent. For instance, multiple users who log in at once are said to be in different, unique sessions. While session-scoped instances are shared across requests inside a session, they are independent between sessions. Figure 5.22 shows how this might look in an injector's timeline.

In this figure, `U1` is an object that exists in the first user's session, and `U2` is a different instance of the same key that exists in the second user's session (it also is an aging Irish rock band). `U1` and `U2` are completely independent of one another. But within the first user's session, all requests share the same instance (`U1`)—likewise with the second user and `U2`. Another interesting point is that the second user's session does not start for a while into the application's life. So there is a time when `U2` is out of scope while `U1` is in scope, even though both are instances of the same key (let's call it `U`) bound under session scope.

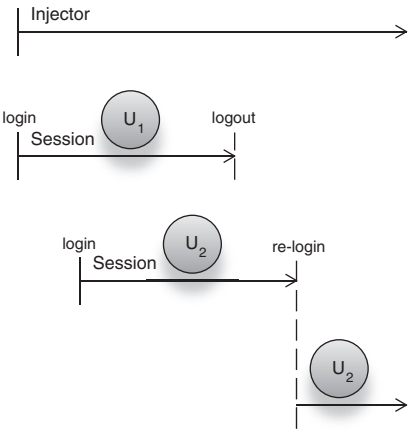

Figure 5.22 Multiple user sessions in the life of an injector

Another interesting thing about figure 5.22 is that the second user actually logs out and logs back in (at the point marked re-login). This means that there are two instances of `U2` for the second user because she started two sessions. The state of `U2` prior to the second login is totally independent from the state after the second login. Contrast this with singleton scoping, where instance state would have been shared for the entire life of the injector, regardless of the number of sessions (or users) involved.

Let's go back to the comic store example. We had three important components:

- `ComicSearch`—A request-scoped service that searched the comic catalog according to given criteria.
- `ComicAccess`—A singleton-scoped data-access service that acted as a bridge between `ComicSearch` and a database.
- `UserContext`—A request-scoped service that was specific to a user; it constructed personalized and filtered search results around a user's behavior.

There was also a filter that set up the `UserContext` each time, by extracting a username from incoming requests. Can you see any room for improvement? Let's revisit class `UserContext` to see if it provides any inspiration:

```
public class UserContext {
    private String username;

    private final ComicAccess comicAccess;

    @Inject
    public UserContext(ComicAccess comicAccess) {
        this.comicAccess = comicAccess;
    }

    public List<Comic> getItemsOfInterest() {
        ...
    }

    public List<Comic> searchComics(String criteria) {
        return comicAccess.searchNoPriorPurchases(criteria, username);
```

```
    }

    public void signIn(String username) {
        this.username = username;
    }
}
```

Straight away there is a clear benefit from making this object session scoped. We wouldn't have to sign in the user on every request! It would be enough to do it once, at the start of the session (logical, since this would be where a user logged in) and simply let it be shared across any further requests from the same user. This would be particularly useful if the application loaded user-specific information (such as a user's full name and date of birth) on sign in, since it would save several unnecessary trips to the database on subsequent requests.

Another saving comes from the use case around caching user-specific information for quick access. If we assume that items of interest for a user are unlikely to change within a single session, there is no need to search and collate them on every request. It can be done once and stored in a memo field for further requests. Here's how you might modify `getItemsOfInterest()` to do just that:

```
public class UserContext {
    private List<Comic> itemsOfInterest;      <-- Memo field

    public List<Comic> getItemsOfInterest() {
        if (null == itemsOfInterest) {        <-- Compute once
            ...                                   and memorize
        }

        return itemsOfInterest;
    }

    ...
}
```

By storing computed suggestions in the `itemsOfInterest` memo field, I save a lot of unnecessary computation so long as subsequent requests come from the same user. Each user has her own memo, in their session-scoped `UserContext`.

Binding `UserContext` to session scope is also laughably simple. As far as the injector is concerned, all you need to do is change its binding (we'll do this in Guice and guice-servlet first):

```
import com.google.inject.servlet.RequestScoped;
import com.google.inject.servlet.SessionScoped;

public class ComicStoreModule extends AbstractModule {

    @Override
    protected void configure() {
    bind(ComicAccess.class).to(ComicAccessImpl.class).in(Singleton.class);

        bind(UserContext.class).in(SessionScoped.class);
        bind(ComicSearch.class).in(RequestScoped.class);
        ...
    }
}
```

Earlier we also saw a shortcut notation, where the class was itself annotated. This is possible with session scope too, but first it requires that you to set up a *scoping annotation*. This must be done manually because the web scopes aren't part of the core Guice distribution. It's fairly easy to do:

```
public class ComicStoreModule extends AbstractModule {

    @Override
    protected void configure() {

        install(new ServletModule());

    bind(ComicAccess.class).to(CommicAccessImpl.class).in(Singleton.class);
        bind(ComicSearch.class).in(ServletScopes.REQUEST_SCOPE);
        ...
    }
}
```

Notice that I've removed any explicit binding of `UserContext`. I can now simply annotate the class, and Guice will correctly bind it to the session scope:

```
@SessionScoped
public class UserContext {
...
}
```

You could do the same with request scopes and guice-servlet's `@RequestScoped` annotation.

The Spring equivalent for binding `UserContext` under session scope is also straightforward (once you have the context listeners set up):

```
<beans ...>
    <bean id="data.comics" class="ComicAccessImpl" scope="singleton">
    ...
    </bean>

    <bean id="web.user" class="UserContext" scope="session">
        <constructor-arg ref="data.comics"/>
    </bean>

    <bean id="web.comicSearch" class="ComicSearch" scope="request">
        <constructor-arg ref="data.comics"/>
        <constructor-arg ref="web.user"/>
    </bean>
</beans>
```

No additional configuration of the web server (or servlet container) is required. Cool! Finally, there are a couple of things to keep in mind when working with objects in the session scope:

- Multiple concurrent requests (from the same user) may belong to the same session.
- This means two threads may hit session-scoped objects at once.
- A session-scoped instance, if wired into a singleton, will stick around even when the user has logged out and the session has ended (*scope-widening injection*). Take care that this does not happen. We'll look at remedies in chapter 6.

So how does a session get scoped?

This is a rather tricky question. I said earlier that HTTP sessions are a hack for the fact that the HTTP protocol is *stateless*. This is more or less accurate—the common way of maintaining a user session is to use a browser *cookie*. A cookie is a small file with unique information that a web application can send to a browser as its identity. So, the next time that web browser hits the same URL, it will send back its cookies. When the web application sees this cookie, it "remembers" who is calling and correctly identifies the user—much like an ATM card reminds the bank of who you are. Cookies are specific to a website and URL, so there is no danger of cookies getting crossed between applications.

Not all browsers support cookies, and since some users consider them a risk, even browsers that do may not have them enabled. In these cases we need an alternative way of tracking sessions. One popular alternative is *URL rewriting*. URL rewriting is quite simple. When a browser requests a document, it does so via a URL. Usually this happens when you click a hyperlink on a previous page. Now, instead of the normal URL, the web application rewrites all the hyperlinks sent to a particular user by adding an identifier to it. Then when you (or any user) click the link, your browser requests the rewritten URL. Incoming requests of this nature are filtered and stripped of the extra bit identifying you as a unique user. This information is used to restore your session, and everything continues as normal.

Most web technologies can be configured to use either URL rewriting or cookies as their session continuation strategy. Some, like the *Java Servlet Framework*, automatically detect browser capability and choose the appropriate strategy. The Java Servlet Framework also provides abstractions for HTTP requestsand sessions to make them easy to work with. Guice-servlet and the SpringFramework both provide integration layers that sit over the servlet APIs and enable transparent web scoping with dependency injection.

5.5 Summary

Scope is about managing the state of objects. Service implementations are bound to keys, which are realized as object instances in an application. Without scoping, keys requested from the injector return new instances each time (and each time they are wired as dependencies). This is known as the no scope. Since there is no way of specifying a duration for the time, state may be preserved in these instances. Singleton-scoped keys, on the other hand, are keys that the injector only wires or returns one instance for. In other words, singletons have one instance per key, per injector. This is different from singleton-patterned objects, which enforce one instance per entire application. I decry this flavor of singleton as an anti-pattern because it is nigh on impossible to test with mock substitution and is tricky in concurrently run tests and environments. Such anti-patterned singletons are also shared between multiple injectors in the same application, which may even violate the configuration of the injector. Always choose singleton scoping over singleton patterning.

Thus, scope may be thought of as the choice of instance each time a key is sought from it—whether new, or old, or shared. The singleton and the no scope are universal and have uses in most applications using dependency injection.

However, there is a whole class of scopes that are specific to particular kinds of applications. These scopes are closely tied to contexts specific to a particular problem domain. One such example is the web. Web-specific scopes are popular with many web frameworks and ship out of the box with many DI libraries. The HTTP request scope is the first and simplest of the web scopes. Keys requested from an injector within the same request always return the same instance. When the request completes, these request-scoped objects are discarded, and subsequent requests force the creation of new instances. Request scoping is interesting since the context that a request purports may be concurrent with several other requests. These requests are all walled off in their own unique scopes and keys, and when sought from the injector they are different across these requests.

HTTP session scope is a step up from request scope and is a natural extension of it. An HTTP session is an artificial construct above a string of requests from the same user. Session scope is thus persistent across all these requests (so long as they are from the same user). Sessions are useful for storing user-specific information temporarily, but they are often abused to store data that is relevant only to specific workflows within the user's session. Try to restrict your use of session scope to keeping around user-relevant information only. Good examples are credentials, preferences, and (relatively) constant biographical data.

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]