

Creating modular applications in Java

OSGi

IN ACTION



Richard S. Hall
Karl Pauls
Stuart McCulloch
David Savage

FOREWORD BY PETER KRIENS

SAMPLE CHAPTER

 MANNING



OSGi in Action

by Richard S. Hall,
Karl Pauls,
Stuart McCulloch,
and David Savage

Chapter 1

Copyright 2011 Manning Publications

brief contents

PART 1 INTRODUCING OSGi: MODULARITY, LIFECYCLE, AND SERVICES 1

- 1 ■ OSGi revealed 3
- 2 ■ Mastering modularity 24
- 3 ■ Learning lifecycle 69
- 4 ■ Studying services 117
- 5 ■ Delving deeper into modularity 154

PART 2 OSGi IN PRACTICE 189

- 6 ■ Moving toward bundles 191
- 7 ■ Testing applications 230
- 8 ■ Debugging applications 258
- 9 ■ Managing bundles 292
- 10 ■ Managing applications 319

PART 3 ADVANCED TOPICS 343

- 11 ■ Component models and frameworks 345
- 12 ■ Advanced component frameworks 373

13	■	Launching and embedding an OSGi framework	412
14	■	Securing your applications	438
15	■	Web applications and web services	477

Part 1

Introducing OSGi: modularity, lifecycle, and services

The OSGi framework defines a dynamic module system for Java. It gives you better control over the structure of your code, the ability to dynamically manage your code's lifecycle, and a loosely coupled approach for code collaboration. Even better, it's fully documented in a very elaborate specification. Unfortunately, the specification was written for people who are going to implement it rather than use it. In the first part of this book, we'll remedy this situation by effectively creating a user-oriented companion guide to the OSGi framework specification. We'll delve into its details by breaking it into three layers: module, lifecycle, and services. We'll explain what you need to understand from the specification to effectively use OSGi technology.

OSGi revealed

This chapter covers

- Understanding Java's built-in support for modularity
- Introducing OSGi technology and how it improves Java modularity
- Positioning OSGi with respect to other technologies

The Java platform is an unqualified success story. It's used to develop applications for everything from small mobile devices to massive enterprise endeavors. This is a testament to its well-thought-out design and continued evolution. But this success has come in spite of the fact that Java doesn't have explicit support for building modular systems beyond ordinary object-oriented data encapsulation.

What does this mean to you? If Java is a success despite its lack of advanced modularization support, then you may wonder if that absence is a problem. Most well-managed projects have to build up a repertoire of project-specific techniques to compensate for the lack of modularization in Java. These include the following:

- Programming practices to capture logical structure
- Tricks with multiple class loaders
- Serialization between in-process components

But these techniques are inherently brittle and error prone because they aren't enforceable via any compile-time or execution-time checks. The end result has detrimental impacts on multiple stages of an application's lifecycle:

- *Development*—You're unable to clearly and explicitly partition development into independent pieces.
- *Deployment*—You're unable to easily analyze, understand, and resolve requirements imposed by the independently developed pieces composing a complete system.
- *Execution*—You're unable to manage and evolve the constituent pieces of a running system, nor minimize the impact of doing so.

It's possible to manage these issues in Java, and lots of projects do so using the custom techniques mentioned earlier, but it's much more difficult than it should be. We're tying ourselves in knots to work around the lack of a fundamental feature. If Java had explicit support for modularity, then you'd be freed from such issues and could concentrate on what you really want to do, which is developing the functionality of your application.

Welcome to the OSGi Service Platform. The OSGi Service Platform is an industry standard defined by the OSGi Alliance to specifically address the lack of support for modularity in the Java platform. As a continuation of its modularity support, it introduces a service-oriented programming model, referred to by some as *SOA in a VM*, to help you clearly separate interface from implementation. This chapter will give you an overview of the OSGi Service Platform and how it helps you create modular and manageable applications using an interface-based development model.

When we've finished this chapter, you'll understand what role OSGi technology plays among the arsenal of Java technologies and why Java and/or other Java-related technologies don't address the specific features provided by OSGi technology.

1.1 The what and why of OSGi

The \$64,000 question is, "What is OSGi?" The simplest answer to this question is that it's a modularity layer for the Java platform. Of course, the next question that may spring to mind is, "What do you mean by *modularity*?" Here we use *modularity* more or less in the traditional computer-science sense, where the code of your software application is divided into logical parts representing separate concerns, as shown in figure 1.1. If your software is modular, you can simplify development and

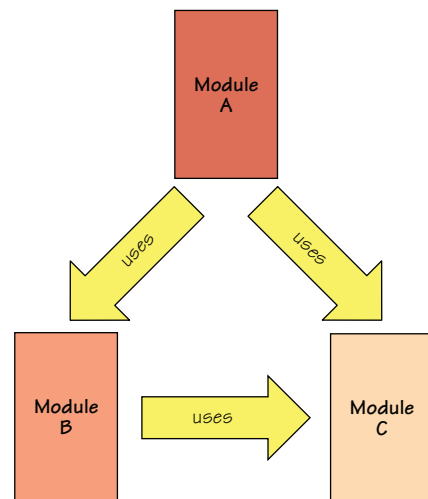


Figure 1.1 Modularity refers to the logical decomposition of a large system into smaller collaborating pieces.

improve maintainability by enforcing the logical module boundaries; we'll discuss more modularity details in chapter 2.

The notion of modularity isn't new. The concept became fashionable back in the 1970s. OSGi technology is cropping up all over the place—for example, as the runtime for the Eclipse IDE and the GlassFish application server. Why is it gaining popularity now? To better understand why OSGi is an increasingly important Java technology, it's worthwhile to understand some of Java's limitations with respect to creating modular applications. When you understand that, then you can see why OSGi technology is important and how it can help.

1.1.1 Java's modularity limitations

Java provides some aspects of modularity in the form of object orientation, but it was never intended to support coarse-grained modular programming. Although it's not fair to criticize Java for something it wasn't intended to address, the success of Java has resulted in difficulty for developers who ultimately have to deal with their need for better modularity support.


Java is promoted as a platform for building all sorts of applications for domains ranging from mobile phone to enterprise applications. Most of these endeavors require, or could at least benefit from, broader support for modularity. Let's look at some of Java's modularity limitations.

LOW-LEVEL CODE VISIBILITY CONTROL

Although Java provides a fair complement of access modifiers to control visibility (such as `public`, `protected`, `private`, and `package private`), these tend to address low-level object-oriented encapsulation and not logical system partitioning. Java has the notion of a *package*, which is typically used for partitioning code. For code to be visible from one Java package to another, the code must be declared `public` (or `protected` if using inheritance). Sometimes, the logical structure of your application calls for specific code to belong in different packages; but this means any dependencies among the packages must be exposed as `public`, which makes them accessible to everyone else, too. Often, this can expose implementation details, which makes future evolution more difficult because users may end up with dependencies on your nonpublic API.

To illustrate, let's consider a trivial “Hello, world!” application that provides a public interface in one package, a private implementation in another, and a main class in yet another.

Listing 1.1 Example of the limitations of Java's object-orientated encapsulation

<pre>package org.foo.hello; public interface Greeting { void sayHello(); }</pre>		<pre>Greeting.java</pre>
<pre>package org.foo.hello.impl; import org.foo.hello.Greeting;</pre>		

GreetingImpl.java

```

public class GreetingImpl implements Greeting {
    final String m_name;

    public GreetingImpl(String name) {
        m_name = name;
    }

    public void sayHello() {
        System.out.println("Hello, " + m_name + "!");
    }
}

```

2 Interface implementation
←

```

package org.foo.hello.main;

import org.foo.hello.Greeting;
import org.foo.hello.impl.GreetingImpl;

public class Main {
    public static void main(String[] args) {
        Greeting greet = new GreetingImpl("Hello World");
        greet.sayHello();
    }
}

```

3 Main method
←

Main.java

Listing 1.1's author may have intended a third party to only interact with the application via the `Greeting` interface ❶. They may mention this in Javadoc, tutorials, blogs, or even email rants, but nothing stops a third party from constructing a new `GreetingImpl` using its public constructor ❷ as is done at ❸.

You may argue that the constructor shouldn't be public and that there is no need to split the application into multiple packages, which could well be true in this trivial example. But in real-world applications, class-level visibility when combined with packaging turns out to be a crude tool for ensuring API coherency. Because supposedly private implementation details can be accessed by third-party developers, you need to worry about changes to private implementation signatures as well as to public interfaces when making updates.

This problem stems from the fact that although Java packages appear to have a logical relationship via nested packages, they don't. A common misconception for people first learning Java is to assume that the parent-child package relationship bestows special visibility privileges on the involved packages. Two packages involved in a nested relationship are equivalent to two packages that aren't. Nested packages are largely useful for avoiding name clashes, but they provide only partial support for the logical code partitioning.

What this all means is that, in Java, you're regularly forced to decide between the following:

- 1 Impairing your application's logical structure by lumping unrelated classes into the same package to avoid exposing nonpublic APIs
- 2 Keeping your application's logical structure by using multiple packages at the expense of exposing nonpublic APIs so they can be accessed by classes in different packages

Neither choice is particularly palatable.

ERROR-PRONE CLASS PATH CONCEPT

The Java platform also inhibits good modularity practices. The main culprit is the Java class path. Why does the class path pose problems for modularity? Largely due to all the issues it hides, such as code versions, dependencies, and consistency. Applications are generally composed of various versions of libraries and components. The class path pays no attention to code versions—it returns the first version it finds. Even if it did pay attention, there is no way to explicitly specify dependencies. The process of setting up your class path is largely trial and error; you just keep adding libraries until the VM stops complaining about missing classes.

Figure 1.2 shows the sort of “class path hell” often found when more than one JAR file provides a given set of classes. Even though each JAR file may have been compiled to work as a unit, when they’re merged at execution time, the Java class path pays no attention to the logical partitioning of the components. This tends to lead to hard-to-predict errors, such as `NoSuchMethodError`, when a class from one JAR file interacts with an incompatible class version from another.

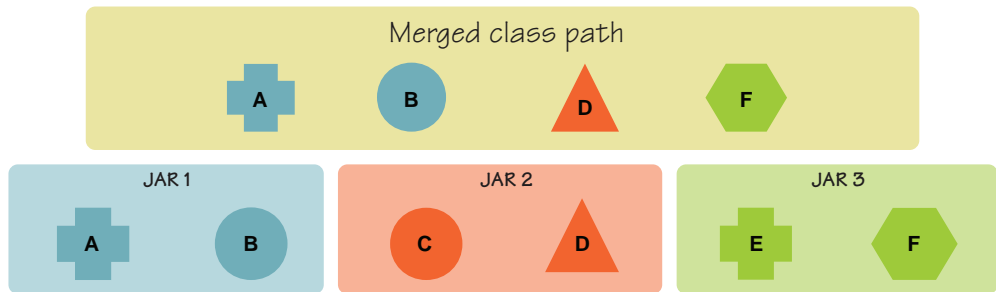


Figure 1.2 Multiple JARs containing overlapping classes and/or packages are merged based on their order of appearance in the class path, with no regard to logical coherency among archives.

In large applications created from independently developed components, it isn’t uncommon to have dependencies on different versions of the same component, such as logging or XML parsing mechanisms. The class path forces you to choose one version in such situations, which may not always be possible. Worse, if you have multiple versions of the same package on the class path, either on purpose or accidentally, they’re treated as split packages by Java and are implicitly merged based on order of appearance.

Overall, the class path approach lacks any form of consistency checking. You get whatever classes have been made available by the system administrator, which is likely only an approximation of what the developer expected.

LIMITED DEPLOYMENT AND MANAGEMENT SUPPORT

Java also lacks support when it comes to deploying and managing your application. There is no easy way in Java to deploy the proper transitive set of versioned code dependencies and execute your application. The same is true for evolving your application and its components after deployment.

Consider the common requirement of wanting to support a dynamic plugin mechanism. The only way to achieve such a benign request is to use class loaders, which are low level and error prone. Class loaders were never intended to be a common tool for application developers, but many of today's systems require their use. A properly defined modularity layer for Java can deal with these issues by making the module concept explicit and raising the level of abstraction for code partitioning.

With this better understanding of Java's limitations when it comes to modularity, we can ponder whether OSGi is the right solution for your projects.

1.1.2 *Can OSGi help you?*

Nearly all but the simplest of applications can benefit from the modularity features OSGi provides, so if you're wondering if OSGi is something you should be interested in, the answer is most likely, "Yes!" Still not convinced? Here are some common scenarios you may have encountered where OSGi can be helpful:

- `ClassNotFoundException`s when starting your application because the class path wasn't correct. OSGi can help by ensuring that code dependencies are satisfied before allowing the code to execute.
- Execution-time errors from your application due to the wrong version of a dependent library on the class path. OSGi verifies that the set of dependencies are consistent with respect to required versions and other constraints.
- Type inconsistencies when sharing classes among modules: put more concretely, the dreaded appearance of `foo instanceof Foo == false`. With OSGi, you don't have to worry about the constraints implied by hierarchical class-loading schemes.
- Packaging an application as logically independent JAR files and deploying only those pieces you need for a given installation. This pretty much describes the purpose of OSGi.
- Packaging an application as logically independent JAR files, declaring which code is accessible from each JAR file, and having this visibility enforced. OSGi enables a new level of code visibility for JAR files that allows you to specify what is and what isn't visible externally.
- Defining an extensibility mechanism for an application, like a plugin mechanism. OSGi modularity is particularly suited to providing a powerful extensibility mechanism, including support for execution-time dynamism.

As you can see, these scenarios cover a lot of use cases, but they're by no means exhaustive. The simple and non-intrusive nature of OSGi tends to make you discover more ways to apply it the more you use it. Having explored some of the limitations of the standard Java class path, we'll now properly introduce you to OSGi.

1.2 An architectural overview of OSGi

The OSGi Service Platform is composed of two parts: the OSGi framework and OSGi standard services (depicted in figure 1.3). The framework is the runtime that implements and provides OSGi functionality. The standard services define reusable APIs for common tasks, such as Logging and Preferences.

The OSGi specifications for the framework and standard services are managed by the OSGi Alliance (www.osgi.org/). The OSGi Alliance is an industry-backed nonprofit corporation founded in March 1999. The framework specification is now on its fourth major revision and is stable. Technology based on this specification is in use in a range of large-scale industry applications, including (but not limited to) automotive, mobile devices, desktop applications, and more recently enterprise application servers.

NOTE Once upon a time, the letters *OSGi* were an acronym that stood for the Open Services Gateway Initiative. This acronym highlights the lineage of the technology but has fallen out of favor. After the third specification release, the OSGi Alliance officially dropped the acronym, and OSGi is now a trademark for the technology.

In the bulk of this book, we'll discuss the OSGi framework, its capabilities, and how to use these capabilities. Because there are so many standard services, we'll discuss only the most relevant and useful services, where appropriate. For any service we miss, you can get more information from the OSGi specifications. For now, we'll continue our overview of OSGi by introducing the broad features of the OSGi framework.

1.2.1 The OSGi framework

The OSGi framework plays a central role when you create OSGi-based applications, because it's the application's execution environment. The OSGi Alliance's framework specification defines the proper behavior of the framework, which gives you a well-defined API to program against. The specification also enables the creation of multiple implementations of the core framework to give you some freedom of choice; there are a handful of well-known open source projects, such as Apache Felix (<http://felix.apache.org/>), Eclipse Equinox (www.eclipse.org/equinox/), and Knopflerfish (www.knopflerfish.org/). This ultimately benefits you, because you aren't tied to a particular vendor and can program against the behavior defined in the specification. It's sort of like the reassuring feeling you get by knowing you can go into any McDonald's anywhere in the world and get the same meal!

OSGi technology is starting to pop up everywhere. You may not know it, but if you use an IDE to do your Java development, it's possible you already have experience with OSGi. The Equinox OSGi framework implementation is the underlying runtime for



Figure 1.3 The OSGi Service Platform specification is divided into halves, one for the OSGi framework and one for standard services.

the Eclipse IDE. Likewise, if you use the GlassFish v3 application server, you're also using OSGi, because the Apache Felix OSGi framework implementation is its runtime. The diversity of use cases attests to the value and flexibility provided by the OSGi framework through three conceptual layers defined in the OSGi specification (see figure 1.4):

- *Module layer*—Concerned with packaging and sharing code
- *Lifecycle layer*—Concerned with providing execution-time module management and access to the underlying OSGi framework
- *Service layer*—Concerned with interaction and communication among modules, specifically the components contained in them

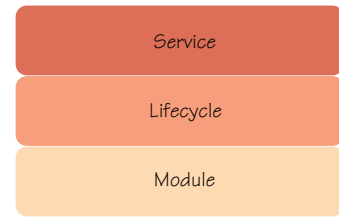


Figure 1.4
OSGi layered architecture

Like typical layered architectures, each layer is dependent on the layers beneath it. Therefore, it's possible for you to use lower OSGi layers without using upper ones, but not vice versa. The next three chapters discuss these layers in detail, but we'll give an overview of each here.

MODULE LAYER

The module layer defines the OSGi module concept, called a *bundle*, which is a JAR file with extra *metadata* (data about data). A bundle contains your class files and their related resources, as depicted in figure 1.5. Bundles typically aren't an entire application packaged into a single JAR file; rather, they're the logical modules that combine to form a given application. Bundles are more powerful than standard JAR files, because you can explicitly declare which contained packages are externally visible (that is, *exported packages*). In this sense, bundles extend the normal access modifiers (public, private, and protected) associated with the Java language.

Another important advantage of bundles over standard JAR files is the fact that you can explicitly declare on which external packages the bundles depend (that is, *imported packages*). The main benefit of explicitly declaring your bundles' exported and imported packages is that the OSGi framework can manage and verify their consistency automatically; this

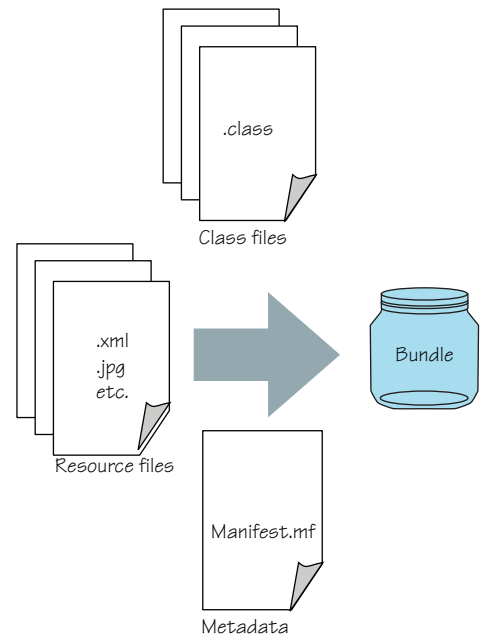


Figure 1.5 A bundle contains code, resources, and metadata.

process is called *bundle resolution* and involves matching exported packages to imported packages. Bundle resolution ensures consistency among bundles with respect to versions and other constraints, which we'll discuss in detail in chapter 2.

LIFECYCLE LAYER

The lifecycle layer defines how bundles are dynamically installed and managed in the OSGi framework. If you were building a house, the module layer would provide the foundation and structure, and the lifecycle layer would be the electrical wiring. It makes everything run.

The lifecycle layer serves two different purposes. External to your application, the lifecycle layer precisely defines the bundle lifecycle operations (install, update, start, stop, and uninstall). These lifecycle operations allow you to dynamically administer, manage, and evolve your application in a well-defined way. This means bundles can be safely added to and removed from the framework without restarting the application process.

Internal to your application, the lifecycle layer defines how your bundles gain access to their execution context, which provides them with a way to interact with the OSGi framework and the facilities it provides during execution. This overall approach to the lifecycle layer is powerful because it lets you create externally (and remotely) managed applications or completely self-managed applications (or any combination).

SERVICE LAYER

Finally, the service layer supports and promotes a flexible application programming model incorporating concepts popularized by service-oriented computing (although these concepts were part of the OSGi framework before service-oriented computing became popular). The main concepts revolve around the service-oriented publish, find, and bind interaction pattern: service providers publish their services into a service registry, while service clients search the registry to find available services to use (see figure 1.6). Nowadays, this service-oriented architecture (SOA) is largely associated with web services; but OSGi services are local to a single VM, which is why some people refer to it as *SOA in a VM*.

The OSGi service layer is intuitive, because it promotes an interface-based development approach, which is generally considered good practice. Specifically, it promotes the separation of interface and implementation. OSGi *services* are Java interfaces representing a conceptual contract between service providers and service clients. This makes the service layer lightweight, because service providers are just Java objects accessed via direct method invocation. Additionally, the service layer expands

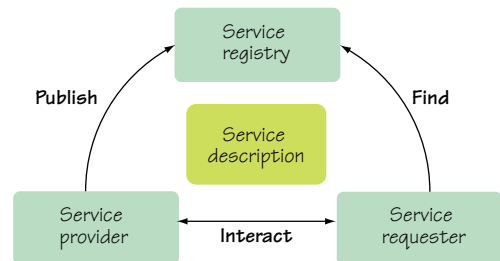


Figure 1.6 The service-oriented interaction pattern. Providers publish services into a registry where requesters can discover which services are available for use.

the bundle-based dynamism of the lifecycle layer with service-based dynamism—services can appear or disappear at any time. The result is a programming model eschewing the monolithic and brittle approaches of the past, in favor of being modular and flexible.

This sounds well and good, but you may still be wondering how these three layers fit together and how you go about using them to create an application on top of them. In the next couple of sections, we'll explore how these layers fit together using some small example programs.

1.2.2 *Putting it all together*

The OSGi framework is made up of layers, but how do you use these layers in application development? We'll make it clearer by outlining the general approach you'll use when creating an OSGi-based application:

- 1 Design your application by breaking it down into service interfaces (normal interface-based programming) and clients of those interfaces.
- 2 Implement your service provider and client components using your preferred tools and practices.
- 3 Package your service provider and client components into (usually) separate JAR files, augmenting each JAR file with the appropriate OSGi metadata.
- 4 Start the OSGi framework.
- 5 Install and start all your component JAR files from step 3.

If you're already following an interface-based approach, the OSGi approach will feel familiar. The main difference will be how you locate your interface implementations (that is, your services). Normally, you might instantiate implementations and pass around references to initialize clients. In the OSGi world, your services will publish themselves in the service registry, and your clients will look up available services in the registry. After your bundles are installed and started, your application will start and execute as normal, but with several advantages. Underneath, the OSGi framework provides more rigid modularity and consistency checking, and its dynamic nature opens up a world of possibilities.

Don't fret if you don't or can't use an interfaced-based approach for your development. The first two layers of the OSGi framework still provide a lot of functionality; in truth, the bulk of OSGi framework functionality lies in these first two layers, so keep reading. Enough talk: let's look at some code.

1.3 *"Hello, world!" examples*

Because OSGi functionality is divided over the three layers mentioned previously (modularity, lifecycle, and service), we'll show you three different "Hello, world!" examples that illustrate each of these layers.

1.3.1 *Module layer example*

The module layer isn't related to code creation as such; rather, it's related to the packaging of your code into bundles. You need to be aware of certain code-related issues

when developing, but by and large you prepare code for the module layer by adding packaging metadata to your project’s generated JAR files. For example, suppose you want to share the following class.

Listing 1.2 Basic greeting implementation

```
package org.foo.hello;

public class Greeting {
    final String m_name;

    public Greeting(String name) {
        m_name = name;
    }

    public void sayHello() {
        System.out.println("Hello, " + m_name + "!");
    }
}
```

During the build process, you compile the source code and put the generated class file into a JAR file. To use the OSGi module layer, you must add some metadata into your JAR file’s META-INF/MANIFEST.MF file, such as the following:

```
Bundle-ManifestVersion: 2
Bundle-Name: Greeting API
Bundle-SymbolicName: org.foo.hello
Bundle-Version: 1.0
Export-Package: org.foo.hello;version="1.0"
```

The first line indicates the OSGi metadata syntax version. Next is the human-readable name, which isn’t strictly necessary. This is followed by the symbolic name and version bundle identifier. The last line shares packages with other bundles.

In this example, the bulk of the metadata is related to bundle identification. The important part is the `Export-Package` statement, because it extends the functionality of a typical JAR file with the ability for you to explicitly declare which packages contained in the JAR are visible to its users. In this example, only the contents of the `org.foo.hello` package are externally visible; if the example included other packages, they wouldn’t be externally visible. This means that when you run your application, other modules won’t be able to accidentally (or intentionally) depend on packages your module doesn’t explicitly expose.

To use this shared code in another module, you again add metadata. This time, you use the `Import-Package` statement to explicitly declare which external packages are required by the code contained in the client JAR. The following snippet illustrates:

```
Bundle-ManifestVersion: 2
Bundle-Name: Greeting Client
Bundle-SymbolicName: org.foo.hello.client
Bundle-Version: 1.0
Import-Package: org.foo.hello;version="[1.0,2.0]"
```

In this case, the last line specifies a dependency on an external package.

To see this example in action, go in the `chapter01/greeting-example/modularity/` directory in the book's companion code, and type `ant to build it` and `java -jar main.jar` to run it. Although the example is simple, it illustrates that creating OSGi bundles out of existing JAR files is a reasonably non-intrusive process. In addition, there are tools that can help you create your bundle metadata, which we'll discuss in appendix A; but in reality, no special tools are required to create a bundle other than what you normally use to create a JAR file. Chapter 2 will go into all the juicy details of OSGi modularity.

1.3.2 Lifecycle layer example

In the last subsection, you saw that it's possible to take advantage of OSGi functionality in a non-invasive way by adding metadata to your existing JAR files. Such a simple approach is sufficient for most reusable libraries, but sometimes you need or want to go further to meet specific requirements or to use additional OSGi features. The lifecycle layer moves you deeper into the OSGi world.

Perhaps you want to create a module that performs some initialization task, such as starting a background thread or initializing a driver; the lifecycle layer makes this possible. Bundles may declare a given class as an *activator*, which is the bundle's hook into its own lifecycle management. We'll discuss the full lifecycle of a bundle in chapter 3, but first let's look at a simple example to give you an idea of what we're talking about. The following listing extends the previous Greeting class to provide a singleton instance.

Listing 1.3 Extended greeting implementation

```
package org.foo.hello;

public class Greeting {
    static Greeting instance;
    final String m_name;

    Greeting(String name) {
        m_name = name;
    }

    public static Greeting get() {
        return instance;
    }

    public void sayHello() {
        System.out.println("Hello, " + m_name + "!");
    }
}
```

Singleton instance

Constructor now package private

Clients must use singleton

Listing 1.4 implements a bundle activator to initialize the Greeting class singleton when the bundle is started and clear it when it's stopped. The client can now use the preconfigured singleton instead of creating its own instance.

Listing 1.4 OSGi bundle activator for our greeting implementation

```
package org.foo.hello;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {

    public void start(BundleContext ctx) {
        Greeting.instance = new Greeting("lifecycle");
    }

    public void stop(BundleContext ctx) {
        Greeting.instance = null;
    }
}
```

A bundle activator must implement a simple OSGi interface, which in this case is composed of the two methods `start()` and `stop()`. At execution time, the framework constructs an instance of this class and invokes the `start()` method when the bundle is started and the `stop()` method when the bundle is stopped. (What we mean by *starting* and *stopping* a bundle will become clearer in chapter 3.) Because the framework uses the same activator instance while the bundle is active, you can share member variables between the `start()` and `stop()` methods.

You may wonder what the single parameter of type `BundleContext` in the `start()` and `stop()` methods is all about. This is how the bundle gets access to the OSGi framework in which it’s executing. From this context object, the module has access to all the OSGi functionality for modularity, lifecycle, and services. In short, it’s a fairly important object for most bundles, but we’ll defer a detailed introduction of it until later when we discuss the lifecycle layer. The important point to take away from this example is that bundles have a simple way to hook into their lifecycle and gain access to the underlying OSGi framework.

Of course, it isn’t sufficient to just create this bundle activator implementation; you have to tell the framework about it. Luckily, this is simple. If you have an existing JAR file you’re converting to be a module, you must add the activator implementation to the existing project so the class is included in the resulting JAR file. If you’re creating a bundle from scratch, you need to compile the class and put the result in a JAR file. You must also tell the OSGi framework about the bundle activator by adding another piece of metadata to the JAR file manifest. For this section’s example, you add the following metadata to the JAR manifest:

```
Bundle-Activator: org.foo.hello.Activator
Import-Package: org.osgi.framework
```

Notice that you also need to import the `org.osgi.framework` package, because the bundle activator has a dependency on it. To see this example in action, go to the `chapter01/greeting-example/lifecycle/` directory in the companion code and type `ant` to build the example and `java -jar main.jar` to run it.

We've now introduced how to create OSGi bundles out of existing JAR files using the module layer and how to make your bundles lifecycle aware so they can use framework functionality. The last example in this section demonstrates the service-oriented programming approach promoted by OSGi.

1.3.3 *Service layer example*

If you follow an interfaced-based approach in your development, the OSGi service approach will feel natural to you. To illustrate, consider the following Greeting interface:

```
package org.foo.hello;
public interface Greeting {
    void sayHello();
}
```

For any given implementation of the Greeting interface, when the sayHello() method is invoked, a greeting will be displayed. In general, a service represents a contract between a provider and prospective clients; the semantics of the contract are typically described in a separate, human-readable document, like a specification. The previous service interface represents the syntactic contract of all Greeting implementations. The notion of a contract is necessary so that clients can be assured of getting the functionality they expect when using a Greeting service.

The precise details of how any given Greeting implementation performs its task aren't known to the client. For example, one implementation may print its greeting textually, whereas another may display its greeting in a GUI dialog box. The following code depicts a simple text-based implementation.

Listing 1.5 Implementation of the Greeting interface

```
package org.foo.hello.impl;
import org.foo.hello.Greeting;

public class GreetingImpl implements Greeting {
    final String m_name;

    GreetingImpl(String name) {
        m_name = name;
    }

    public void sayHello() {
        System.out.println("Hello, " + m_name + "!");
    }
}
```

You may be thinking that nothing in the service interface or listing 1.5 indicates that you're defining an OSGi service. You're correct. That's what makes the OSGi's service approach so natural if you're already following an interface-based approach; your code will largely stay the same. Your development will be a little different in two places: how you make a service instance available to the rest of your application, and how the rest of your application discovers the available service.

All service implementations are ultimately packaged into a bundle, and that bundle must be lifecycle aware in order to register the service. This means you need to create a bundle activator for the example service, as shown next.

Listing 1.6 OSGi bundle activator with service registration

```
package org.foo.hello.impl;

import org.foo.hello.Greeting;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {

    public void start(BundleContext ctx) {
        ctx.registerService(Greeting.class.getName(),
            new GreetingImpl("service"), null);
    }

    public void stop(BundleContext ctx) {}
}
```

This time, in the `start()` method, instead of storing the `Greeting` implementation as a singleton, you use the provided bundle context to register it as a service in the service registry. The first parameter you need to provide is the interface name(s) that the service implements, followed by the actual service instance, and finally the service properties. In the `stop()` method, you could unregister the service implementation before stopping the bundle; but in practice, you don’t need to do this. The OSGi framework automatically unregisters any registered services when a bundle stops.

You’ve seen how to register a service, but what about discovering a service? The following listing shows a simplistic client that doesn’t handle missing services and that suffers from potential race conditions. We’ll discuss a more robust way to access services in chapter 4.

Listing 1.7 OSGi bundle activator with service discovery

```
package org.foo.hello.client;

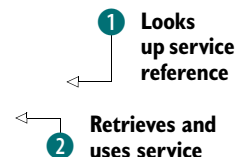
import org.foo.hello.Greeting;
import org.osgi.framework.*;

public class Client implements BundleActivator {

    public void start(BundleContext ctx) {
        ServiceReference ref =
            ctx.getServiceReference(Greeting.class.getName());

        ((Greeting) ctx.getService(ref)).sayHello();
    }

    public void stop(BundleContext ctx) {}
}
```



Notice that accessing a service in OSGi is a two-step process. First, an indirect reference is retrieved from the service registry ❶. Second, this indirect reference is used to

access the service object instance ②. The service reference can be safely stored in a member variable; but in general it isn't a good idea to hold on to references to service object instances, because services may be unregistered dynamically, resulting in stale references that prevent garbage collection of uninstalled bundles.

Both the service implementation and the client should be packaged into separate bundle JAR files. The metadata for each bundle declares its corresponding activator, but the service implementation exports the `org.foo.hello` package, whereas the client imports it. Note that the client bundle's metadata only needs to declare an import for the `Greeting` interface package—it has no direct dependency on the service implementation. This makes it easy to swap service implementations dynamically without restarting the client bundle. To see this example in action, go to the `chapter01/greeting-example/service/` directory in the companion code and type `ant` to build the example and `java -jar main.jar` to run it.

Now that you've seen some examples, you can better understand how each layer of the OSGi framework builds on the previous one. Each layer gives you additional capabilities when building your application, but OSGi technology is flexible enough for you to adopt it according to your specific needs. If you only want better modularity in your project, use the module layer. If you want a way to initialize modules and interact with the module layer, use both the module and lifecycle layers. If you want a dynamic, interface-based development approach, use all three layers. The choice is yours.

1.3.4 Setting the stage

To help introduce the concepts of each layer in the OSGi framework in the next three chapters, we'll use a simple paint program; its user interface is shown in figure 1.7.

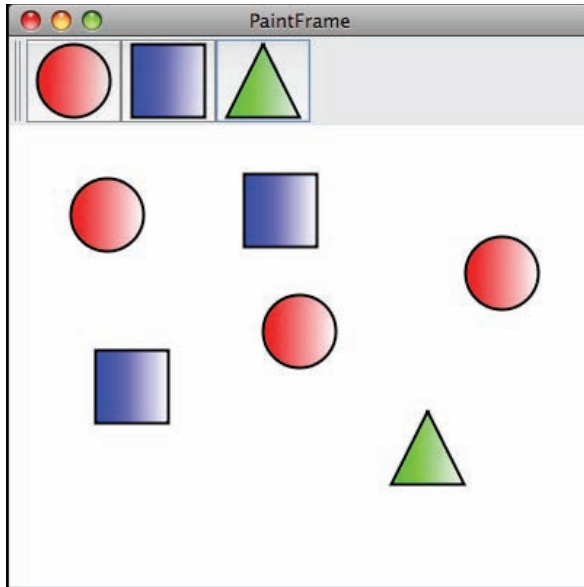


Figure 1.7 Simple paint program user interface

The paint program isn't intended to be independently useful; rather, it's used to demonstrate common issues and best practices.

From a functionality perspective, the paint program only allows the user to paint various shapes, such as circles, squares, and triangles. The shapes are painted in pre-defined colors. Available shapes are displayed as buttons in the main window's toolbar. To draw a shape, the user selects it in the toolbar and then clicks anywhere in the canvas to draw it. The same shape can be drawn repeatedly by clicking in the canvas numerous times. The user can drag drawn shapes to reposition them. This sounds simple enough. The real value of using a visual program for demonstrating these concepts will become evident when we start introducing execution-time dynamism.

We've finished our overview of the OSGi framework and are ready to delve into the details; but before we do, we'll put OSGi in context by discussing similar or related technologies. Although no Java technology fills the exact same niche as OSGi, several tread similar ground, and it's worth understanding their relevance before moving forward.

1.4 Putting OSGi in context

OSGi is often mentioned in the same breath with many other technologies, but it's in a fairly unique position in the Java world. Over the years, no single technology has addressed OSGi's exact problem space, but there have been overlaps, complements, and offshoots. Although it isn't possible to cover how OSGi relates to every conceivable technology, we'll address some of the most relevant in roughly chronological order. After reading this section, you should have a good idea whether OSGi replaces your familiar technologies or is complementary to them.

1.4.1 Java Enterprise Edition

Java Enterprise Edition (Java EE, formerly J2EE) has roots dating back to 1997. Java EE and OSGi began targeting opposite ends of the computing spectrum (the enterprise vs. embedded markets, respectively). Only within the last couple of years has OSGi technology begun to take root in the enterprise space.

In total, the Java EE API stack isn't related to OSGi. The Enterprise JavaBeans (EJB) specification is probably the closest comparable technology from the Java EE space, because it defines a component model and packaging format. But its component model focuses on providing a standard way to implement enterprise applications that must regularly handle issues of persistence, transactions, and security. The EJB deployment descriptors and packaging formats are relatively simplistic and don't address the full component lifecycle, nor do they support clean modularity concepts.

OSGi is also used in the Java EE domain to provide a more sophisticated module layer beneath these existing technologies. Because the two ignored each other for so long, there are some challenges in moving existing Java EE concepts to OSGi, largely due to different assumptions about how class loading is performed. Still, progress is being made, and today OSGi plays a role in all major application servers, such as IBM's WebSphere, Red Hat's JBoss, Oracle's GlassFish, ObjectWeb's JOnAS, and Apache's Geronimo.

1.4.2 *Jini*

An often-overlooked Java technology is Jini, which is definitely a conceptual sibling of OSGi. Jini targets OSGi's original problem space of networked environments with a variety of connected devices.

Sun began developing Jini in 1998. The goal of Jini is to make it possible to administer a networked environment as a flexible, dynamic group of services. Jini introduces the concepts of service providers, service consumers, and a service lookup registry. All of this sounds completely isomorphic to OSGi; where Jini differs is its focus on distributed systems. Consumers access clients through some form of proxy using a remote procedure call mechanism, such as Remote Method Invocation (RMI). The service-lookup registry is also a remotely accessible, federated service. The Jini model assumes remote access across multiple VM processes, whereas OSGi assumes everything occurs in a single VM process. But in stark contrast to OSGi, Jini doesn't define any modularity mechanisms and relies on the execution-time code-loading features of RMI. The open source project Newton is an example of combining OSGi and Jini technologies in a single framework.

1.4.3 *NetBeans*

NetBeans, an IDE and application platform for Java, has a long history of having a modular design. Sun purchased NetBeans in 1999 and has continued to evolve it.

The NetBeans platform has a lot in common with OSGi. It defines a fairly sophisticated module layer and also promotes interface-based programming using a lookup pattern that is similar to the OSGi service registry. Whereas OSGi focused on embedded devices and dynamism, the NetBeans platform was originally an implementation layer for the IDE. Eventually, the platform was promoted as a separate tool in its own right, but it focused on being a complete GUI application platform with abstractions for file systems, windowing systems, and much more. NetBeans has never been seen as comparable to OSGi, even though it is; perhaps OSGi's more narrow focus is an asset in this case.

1.4.4 *Java Management Extensions*

Java Management Extensions (JMX), released in 2000 through the Java Community Process (JCP) as JSR 3, was compared to OSGi in the early days. JMX is a technology for remotely managing and monitoring applications, system objects, and devices; it defines a server and a component model for this purpose.

JMX isn't comparable to OSGi; it's complementary, because it can be used to manage and monitor an OSGi framework and its bundles and services. Why did the comparisons arise in the first place? There are probably three reasons: the JMX component model was sufficiently generic that it was possible to use it for building applications; the specification defined a mechanism for dynamically loading code into its server; and certain early adopters pushed JMX in this direction. One major perpetrator was JBoss, which adopted and extended JMX for use as a module layer in its

application server (since eliminated in JBoss 5). Nowadays, JMX isn't (and shouldn't be) confused with a module system.

1.4.5 Lightweight containers

Around 2003, lightweight or inversion of control (IoC) containers started to appear, such as PicoContainer, Spring, and Apache Avalon. The main idea behind this crop of IoC containers was to simplify component configuration and assembly by eliminating the use of concrete types in favor of interfaces. This was combined with dependency injection techniques, where components depend on interface types and implementations of the interfaces are injected into the component instance. OSGi services promote a similar interface-based approach but employ a service-locator pattern to break a component's dependency on component implementations, similar to Apache Avalon.

At the same time, the Service Binder project was creating a dependency injection framework for OSGi components. It's fairly easy to see why the comparisons arose. Regardless, OSGi's use of interface-based services and the service-locator pattern long predated this trend, and none of these technologies offer a sophisticated dynamic module layer like OSGi. There is now significant movement from IoC vendors to port their infrastructures to the OSGi framework, such as the work by VMware (formerly SpringSource) on the OSGi Blueprint specification (discussed in chapter 12).

1.4.6 Java Business Integration

Java Business Integration (JBI) was developed in the JCP and released in 2005. Its goal was to create a standard SOA platform for creating enterprise application integration (EAI) and business-to-business (B2B) integration solutions.

In JBI, plugin components provide and consume services after they're plugged in to the JBI framework. Components don't directly interact with services, as in OSGi; instead, they communicate indirectly using normalized Web Services Description Language (WSDL)-based messages.

JBI uses a JMX-based approach to manage component installation and lifecycle and defines a packaging format for its components. Due to the inherent similarities to OSGi's architecture, it was easy to think JBI was competing for a similar role. On the contrary, its fairly simplistic modularity mechanisms mainly addressed basic component integration into the framework. It made more sense for JBI to use OSGi's more sophisticated modularity, which is ultimately what happened in Project Fuji from Sun and ServiceMix from Apache.

1.4.7 JSR 277

In 2005, Sun announced JSR 277 ("Java Module System") to define a module system for Java. JSR 277 was intended to define a module framework, packaging format, and repository system for the Java platform. From the perspective of the OSGi Alliance, this was a major case of reinventing the wheel, because the effort was starting from scratch rather than building on the experience gained from OSGi.

In 2006, many OSGi supporters pushed for the introduction of JSR 291 (titled “Dynamic Component Support for Java”), which was an effort to bring OSGi technology properly into JCP standardization. The goal was twofold: to create a bridge between the two communities and to ensure OSGi technology integration was considered by JSR 277. The completion of JSR 291 was fairly quick because it started from the OSGi R4 specification and resulted in the R4.1 specification release. During this period, OSGi technology continued to gain momentum. JSR 277 continued to make slow progress through 2008 until it was put on hold indefinitely.

1.4.8 JSR 294

During this time in 2006, JSR 294 (titled “Improved Modularity Support in the Java Programming Language”) was introduced as an offshoot of JSR 277. Its goal was to focus on necessary language changes for modularity. The original idea was to introduce the notion of a *superpackage* into the Java language—a package of packages.

The specification of superpackages got bogged down in details until it was scrapped in favor of adding a module-access modifier keyword to the language. This simplification ultimately led to JSR 294 being dropped and merged back into JSR 277 in 2007. But when it became apparent in 2008 that JSR 277 would be put on hold, JSR 294 was pulled back out to address a module-level access modifier.

With JSR 277 on hold, Sun introduced an internal project, called *Project Jigsaw*, to modularize the JDK. The details of Jigsaw are still evolving after the acquisition of Sun by Oracle.

1.4.9 Service Component Architecture

Service Component Architecture (SCA) began as an industry collaboration in 2004 and ultimately resulted in final specifications in 2007. SCA defines a service-oriented component model similar to OSGi’s, where components provide and require services. Its component model is more advanced because it defines *composite components* (components made of other components) for a fully recursive component model.

SCA is intended to be a component model for declaratively composing components implemented using various technologies (such as Java, Business Process Execution Language [BPEL], EJB, and C++) and integrated using various bindings (such as SOAP/HTTP, Java Message Service [JMS], Java EE Connector Architecture [JCA], and Internet Inter-Orb Protocol [IIOP]). SCA does define a standard packaging format, but it doesn’t define a sophisticated module layer like OSGi provides. The SCA specification leaves open the possibility of other packaging formats, which makes it possible to use OSGi as a packaging and module layer for Java-based SCA implementations; Apache Tuscany and Newton are examples of an SCA implementation that use OSGi. In addition, bundles could be used to implement SCA component types, and SCA could be used as a mechanism to provide remote access to OSGi services.

1.4.10 .NET

Although Microsoft’s .NET (released in 2002) isn’t a Java technology, it deserves mention because it was largely inspired by Java and did improve on it in ways that are similar

to how OSGi improves Java. Microsoft not only learned from Java's example but also learned from the company's own history of dealing with DLL hell. As a result, .NET includes the notion of an *assembly*, which has modularity aspects similar to an OSGi bundle. All .NET code is packaged into an assembly, which takes the form of a DLL or EXE file. Assemblies provide an encapsulation mechanism for the code contained inside of them; an access modifier, called `internal`, is used to indicate visibility within an assembly but not external to it. Assemblies also contain metadata describing dependencies on other assemblies, but the overall model isn't as flexible as OSGi's. Because dependencies are on specific assembly versions, the OSGi notion of provider substitutability isn't attainable.

At execution time, assemblies are loaded into application domains and can only be unloaded by unloading the entire application domain. This makes the highly dynamic and lightweight nature of OSGi hard to achieve, because multiple assemblies loaded into the same application domain must be unloaded at the same time. It's possible to load assemblies into separate domains; but then communication across domains must use interprocess communication to collaborate, and type sharing is greatly complicated. There have been research efforts to create OSGi-like environments for the .NET platform, but the innate differences between the .NET and Java platforms results in the two not having much in common. Regardless, .NET deserves credit for improving on standard Java in this area.

1.5 Summary

In this chapter, we've laid the foundation for everything we'll cover in the rest of the book. What you've learned includes the following:

- The Java platform is great for developing applications, but its support for modularity is largely limited to fine-grained object-oriented mechanisms, rather than more coarse-grained modularity features needed for project management.
- The OSGi Service Platform, through the OSGi framework, addresses the modularity shortcomings of Java to create a powerful and flexible solution.
- The declarative, metadata-based approach employed by OSGi provides a non-invasive way to take advantage of its sophisticated modularity capabilities by modifying how projects are packaged with few, if any, changes to the code.
- The OSGi framework defines a controlled, dynamic module lifecycle to simplify management.
- Following good design principles, OSGi promotes an interface-based programming approach to separate interfaces from implementations.

With this high-level understanding of Java's limitations and OSGi's capabilities, we can start our adventure by diving into the details of the module layer in chapter 2. This is the foundation of everything else in the OSGi world.

OSGi IN ACTION

Hall • Pauls • McCulloch • Savage

OSGi is a Java-based framework for creating applications as a set of interconnected modules. OSGi lets you install, start, stop, update, or uninstall modules at execution time without taking down your entire system. It's the backbone of the Eclipse plugin system, as well as many Java EE containers, such as GlassFish, Geronimo, and WebSphere.

OSGi in Action provides a clear introduction to OSGi concepts with examples that are relevant both for architects and developers. You'll start with the central ideas of OSGi: bundles, module lifecycles, and interaction among application components. With the core concepts well in hand, you'll explore numerous application scenarios and techniques. You'll learn how to migrate legacy systems to OSGi and how to test, debug, and manage applications.

What's Inside

- Core ideas of OSGi
- Vocabulary, tools, and strategies
- Applying OSGi

This book assumes readers with a working knowledge of Java, but requires no previous exposure to OSGi.

Richard S. Hall, Karl Pauls, Stuart McCulloch, and David Savage are all respected Java developers and committers on the Apache Felix OSGi implementation.

For access to the book's forum and a free ebook for owners of this book, go to manning.com/OSGiinAction



“An impressive book.”

—From the foreword by
Peter Kriens
OSGi Technical Director

“A lucid explanation of an intricate topic.”

—John S. Griffin, Overstock.com

“Easy to read ... explains everything you need to know.”

—Jason Lee, Oracle

“Straight from the experts.”

—Erik Van Oosten, JTeam

“Hit the ground running with this book.”

—David Dossot
Coauthor of *Mule in Action*

ISBN-13: 978-1-933988-91-7
ISBN-10: 1-933988-91-6



9 781933 988917