



# OSGi IN ACTION

**Creating Modular  
Applications in Java**

Richard S. Hall  
Karl Pauls  
Stuart McCulloch  
David Savage

Unedited Draft





**MEAP Edition**  
**Manning Early Access Program**

Copyright 2009 Manning Publications  
For more information on this and other Manning titles go to [www.manning.com](http://www.manning.com)

## **OSGi in Action Table of Contents**

### **Introducing OSGi: Modularity, Lifecycle and Services**

- 1 OSGi Revealed
- 2 Mastering Modularity
- 3 Learning Lifecycle
- 4 Studying Services
- 5 Delving Deeper into Modularity

### **OSGi in Practice**

- 6 Moving towards bundles
- 7 Managing applications
- 8 Testing applications
- 9 Debugging applications
- 10 Integrating Legacy Code

### **Advanced Topics**

- 11 Component Models
- 12 Launching versus Embedding OSGi Frameworks
- 13 Security
- 14 Web services and applications

### **Appendix**

- a) OSGi Glossary
- b) Building with Maven

# 1

## *OSGi Revealed*

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

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

- Programming practices,
- Tricks with multiple class loaders, and
- Serialization between in-process components.

However these techniques are inherently brittle and error prone since they are not enforceable via any specific compile-time or run-time checks. The end result has detrimental impacts on multiple stages of an application's lifecycle:

- Development – you are unable to clearly and explicitly partition development into independent pieces.
- Deployment – you are unable to easily analyze, understand, and resolve requirements imposed by the collection of independently developed pieces that make up the system.
- Execution – you are unable to manage and evolve the constituent pieces of a running system, nor minimize the impact of doing so.

It is definitely possible to manage these issues in Java, and lots of projects do so using the custom techniques mentioned above, but it is 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 would 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. Additionally, it also introduces a new service-oriented programming model, referred to by some as "SOA in a VM." This chapter will give you an overview of the OSGi Service Platform and the issues it is intended to address. Once we have finished this chapter we will have enough background knowledge to start digging into the details in chapter 2.

## **1.1 The what and why of OSGi**

The sixty-four-thousand dollar question is, "What is OSGi?" The simplest answer to this question is it is a modularity layer for the Java platform. Of course, the next question that might 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. If your software is modular, then you can simplify development and improve maintainability by enforcing the logical module boundaries; we will discuss more modularity details in Chapter [ref ch2].

The notion of modularity is not new. The concept actually became fashionable back in the 1970s. So, why is OSGi all the rage right now? To better understand what OSGi can do for you, it is worthwhile to understand what Java is not doing for you with respect to modularity. Once you understand that, then you can see how OSGi can help you.

### **1.1.1 Java's modularity limitations**

Java was never intended to support modular programming, so we admit that criticizing its inability to do so is a little unfair. Java has been promoted as a platform for building all sorts of applications for all sorts of domains ranging from mobile phone to enterprise applications. Most of these endeavors require, or could at least benefit from, modularity, so Java's lack of explicit support does cause some amount of pain for developers. From this point of view, we do feel the following criticisms are valid.

#### **LOW-LEVEL CODE VISIBILITY CONTROL**

While Java provides a fair complement of access modifiers to control visibility (e.g., `public`, `protected`, `private`, and `package private`), these tend to address low-level object-oriented encapsulation and do not really address 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 then this means any dependencies among the packages must be exposed as `public`, which makes it accessible to everyone else too. Often this can expose implementation details, which makes future evolution more difficult since users may end up with dependencies on your non-public API.

To illustrate this, 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 Trivial example of the limitations of Java's object-orientated encapsulation

```
package org.foo.hello;

public interface Greeting {
    void sayHello();
}

-----

package org.foo.hello.impl;

import org.foo.hello.Greeting;

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 + "!");
    }
}

-----

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();
    }
}
```

In Listing 1.1, the author may have intended a third party to only interact with the application via the `Greeting` interface. He or she may mentioned this in Javadoc, tutorials, blogs, or even email rants, but there is nothing actually stopping a third party from constructing a new `GreetingImpl` using its public constructor.

You might argue that the constructor should not be public and 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 very crude tool for ensuring API coherency. Seeing how a supposedly private implementation can be accessed by third-parties developers, now you need to worry about changes to private implementation signatures as well as that of 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 actually do not. 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 are not. Nested packages are largely useful for avoiding name clashes and provide only partial support for the logical code partitioning.

The semantics of packages combined with class-level visibility force you to decide between impairing your application's logical partitioning to avoid exposing non-public API or keeping proper logical partitioning at the expense of exposing non-public API. 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 of the issues that 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 simply returns the first version that 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 1.1 shows the sort of “class path Hell” that can often be 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 merged at run time the Java class path pays no attention to the logical partitioning of the components. This tends to lead to such hard to predict

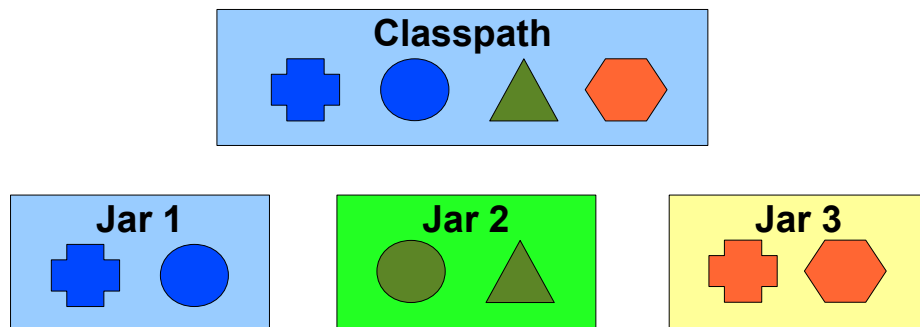


Figure 1.1 Multiple JARs containing the “same” class are merged based on their order of appearance in the class path paying no respect to logical coherency between archives

errors, such as `NoSuchMethodError`, when a class from one JAR file interacts with an incompatible class from another one.

In large applications, created from independently developed components, it is not 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 just one version in such situations, which may not always work for all parts of the application if there are incompatibilities between versions. Worse, if you happen to have multiple versions of the same package on the class path, either on purpose or accidentally, they are 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 just get whatever classes have been made available by the system administrator, which is very likely only an approximation of what the developer actually expected. This hardly inspires confidence.

#### LIMITED DEPLOYMENT AND MANAGEMENT SUPPORT

Java's 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. Likewise 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 so 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 the limitations of Java 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 are wondering if OSGi is something you should be interested in, the answer is most likely, "Yes!" Still not convinced? Here are some common scenarios that you may have encountered where OSGi can be helpful:

- If you ever received `ClassNotFoundException`s when starting your application because the class path was not correct. OSGi can help you here by ensuring that code dependencies are satisfied before allowing the code to execute.
- If you ever encountered run-time errors when executing 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.
- If you ever wanted to share classes between modules without worrying about constraints implied by hierarchical class loading schemes; put in a more concrete way, the dreaded appearance of `"foo instanceof Foo == false"` when sharing

objects between two servlet contexts.

- If you ever wanted to package your application as logically independent JAR files and be able to deploy only those pieces you actually need for a given installation. This pretty much describes the purpose of OSGi.
- If you ever wanted to package your application as logically independent JAR files and also wanted to declare which code is accessible from each JAR file and have this visibility enforced. OSGi enables a new level of code visibility for JAR files that allows you to specify what is and what is not visible externally.
- If you ever wanted to define an extensibility mechanism for your application, like a plugin mechanism. OSGi modularity is particularly suited to providing a powerful extensibility mechanism, including support for run-time dynamism.

As you can see, these scenarios cover a lot of use cases, but are 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 A quick OSGi overview**

The OSGi Service Platform is composed of two parts: the OSGi framework and OSGi standard services. 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 (<http://www.osgi.org>). The alliance is an industry backed non-profit corporation founded in March 1999. The framework specification is now on it's 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 simply a trademark for the technology.

In the bulk of this book we will discuss the OSGi framework, its capabilities, and how to leverage these capabilities. We will not present or discuss in detail all OSGi standard services, but we will discuss relevant services where appropriate. Let's 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 creating OSGi-based applications, since it is 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, Eclipse Equinox, and Knopflerfish. This ultimately benefits you, since you are not tied to a particular vendor and are able to program against the behavior defined in the specification.

The OSGi specification conceptually divides the framework into three layers (see Figure 1.2):

- Module layer – this layer is concerned with packaging and sharing code.
- Lifecycle layer – this layer is concerned with providing run-time module management and access to the underlying OSGi framework.
- Service layer – this layer is concerned with interaction and communication among modules, specifically the components contained in them.

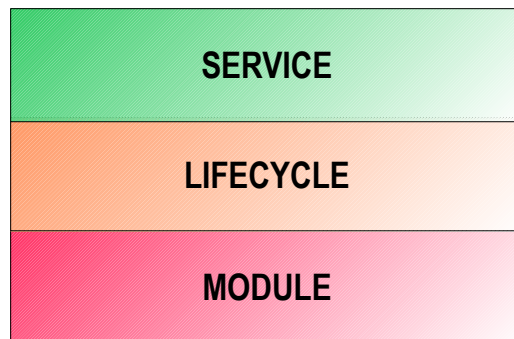


Figure 1.2 OSGi layered architecture

Like typical layered architectures, each layer is dependent upon the layers beneath it. Therefore, it is 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 will give an overview of each here.

#### 1.2.2 Module layer

The module layer defines the OSGi module concept, called a bundle, which is simply a JAR file with extra metadata. A bundle contains your class files and their related resources. Bundles are typically not an entire application packaged into a single JAR file; rather, they are the logical modules that combine to form a given application. Bundles are more powerful

than standard JAR files, since you are able to explicitly declare which contained packages are externally visible (i.e., exported packages). In this sense, bundles extend the normal access modifiers (i.e., `public`, `private`, and `protected`) associated with the Java language.

Another important advantage of bundles over standard JAR files is that you are also able to explicitly declare on which external packages your bundle depends (i.e., imported packages). The main benefit of explicitly declaring your bundles' exported and imported packages is that the OSGi framework can manage and verify the consistency of your bundles automatically; this 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 will discuss in detail in [ref ch2].

### 1.2.3 Lifecycle layer

The lifecycle layer defines how bundles are dynamically installed and managed in the OSGi framework. The lifecycle layer serves two different purposes. External to your application, the lifecycle layer precisely defines the bundle lifecycle operations (e.g., 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 that bundles can be safely added 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 since it allows you to create externally (and remotely) managed applications or completely self-managed applications (or any combination of the two).

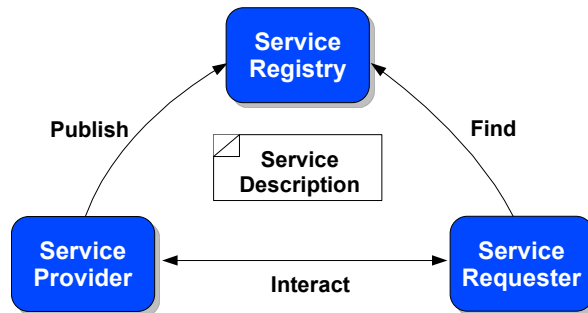


Figure 1.3 The service-oriented interaction pattern. Providers publish services into a registry where requesters can discover which services are available for use.

### 1.2.4 Service layer

Finally, the service layer supports and promotes a flexible application programming model that incorporates concepts popularized by service-oriented computing (although, these concepts were part of the OSGi framework before SOA 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 1.3). Nowadays, 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 very intuitive, since 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 simply Java interfaces that represent a conceptual contract between service providers and service clients. This makes the service layer very lightweight, since service providers are simply Java objects accessed via direct method invocation. Additionally, the service layer expands the bundle-based dynamism of the lifecycle layer with service-based dynamism, i.e., services can appear or disappear at any time. The result is a programming model that eschews the monolithic and brittle approaches of the past, in favor of being modular and flexible.

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

### **1.2.5 Putting it all together**

The OSGi framework is made up of layers, but how do we use these layers in application development? Let's try to make it a little clearer by outlining the general approach you will use when creating an OSGi-based application:

1. Design your application by breaking it down into service interfaces (i.e., 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 of your component JAR files from step 3.

If you are already following an interface-based approach, then the OSGi approach will feel very familiar to you. The main difference will be how you locate your interface implementations (i.e., 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. Once your bundles are installed and started, your application will start and execute like normal, but with several advantages. Underneath, the OSGi framework is providing more

rigid modularity and consistency checking and its dynamic nature opens up a whole 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 for you; 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

Since OSGi functionality is divided over the three layers mentioned previously (modularity, lifecycle, and service), we will show you three different “Hello, world!” examples that illustrate each of these layers.

#### 1.3.1 Modularity layer

The modularity layer is actually not related to code creation as such; rather, it is related to the packaging of your code into bundles. There are certain code-related issues of which you need to be aware when developing, but by and large you prepare your code for the modularity layer by adding packaging metadata to your project's generated JAR files. For example, suppose you want to share the class in Listing 1.2.

#### 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 would compile the source code and put the generated class file into a JAR file. To leverage the OSGi modularity layer you must add some metadata into your JAR file's META-INF/MANIFEST.MF file as shown in Listing 1.3.

#### Listing 1.3 OSGi metadata for greeting implementation

```
Bundle-ManifestVersion: 2 #A
Bundle-Name: Greeting API #B
Bundle-SymbolicName: org.foo.hello #C
Bundle-Version: 1.0 #C
Export-Package: org.foo.hello;version="1.0" #D
#A Indicates the OSGi metadata syntax version
#B Human-readable bundle name, not strictly necessary
```

**#C Symbolic name and version uniquely identify a bundle**  
**#D Make our package visible to other bundles**

In this small example, the bulk of the metadata is largely related to bundle identification. The important part is the `Export-Package` statement, since this 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 the users of it. In this particular example, only the contents of the `org.foo.hello` package are externally visible; if there were other packages in our example, they would not be externally visible. So what does this really mean? It means when you run your application, other modules will not be able to accidentally (or intentionally) depend on packages that your module doesn't explicitly expose.

To use this shared code in another module you would again add metadata, this time using the `Import-Package` statement to explicitly declare which external packages are required by the code contained in the client JAR (see Listing 1.4).

#### Listing 1.4 OSGi metadata for greeting client

```
Bundle-ManifestVersion: 2 #A
Bundle-Name: Greeting Client #B
Bundle-SymbolicName: org.foo.hello.client #C
Bundle-Version: 1.0 #C
Import-Package: org.foo.hello;version="[1.0,2.0)" #D
```

**#A Indicates the OSGi metadata syntax version**  
**#B Human-readable bundle name, not strictly necessary**  
**#C Symbolic name and version uniquely identify a bundle**  
**#D Request this package from another bundle**

To see this example in action, go into the `greeting-example/modularity/` directory for Chapter 1 in the accompanying code and type `ant` to build it and `java -jar main.jar` to run it. Although this example is simple, it illustrates that creating OSGi bundles out of your existing JAR files is a reasonably non-intrusive process. In addition, there are tools that can help you create your bundle metadata, which we will discuss in [ref xx], but in reality no special tools are required to create a bundle other than what you normally use to create a JAR file. Chapter [ref ch2] will go into all of the juicy details of OSGi modularity and how to take advantage of it in your applications.

### 1.3.2 Lifecycle layer

In the last subsection we saw that it is possible to leverage OSGi functionality in a non-invasive way by simply adding metadata to your existing JAR files. Such a simple approach is sufficient for most reusable libraries, but sometimes we need or want to go further to meet specific requirements or to use additional OSGi features. The lifecycle layer moves us 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 will discuss the full lifecycle of a bundle later in chapter [ref ch3],

but first let's look at a simple example to give you an idea of what we are talking about. In Listing 1.5, we extend our previous `Greeting` class to provide a singleton instance.

#### Listing 1.5 Extended greeting implementation

```
package org.foo.hello;

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

    Greeting(String name) {           #B
        m_name = name;
    }

    public static Greeting get() {    #C
        return instance;
    }

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

#A singleton instance to be managed
#B constructor is now package private
#C clients can only use the singleton
```

Listing 1.6 implements a bundle activator that initializes the `Greeting` class singleton when the bundle is started and clears it when it is stopped. The client can now use the pre-configured singleton instead of creating its own instance.

#### Listing 1.6 OSGi bundle activator

```
package org.foo.hello;

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

public class Activator implements BundleActivator {           #A

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

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

You can see at #A that a bundle activator must implement a simple OSGi interface, which comprises the two methods depicted at #B and #C. At execution time, the framework will construct an instance of this class and invoke the `start()` method when the bundle is started and the `stop()` method when the bundle is stopped. What we precisely mean by

“starting” or “stopping” a bundle will become clearer in chapter [ref ch3]. Because the framework uses the same activator instance while the bundle is active, you can share member variables between the `start()` and `stop()` methods.

The inquisitive among you might be wondering 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 is executing. From this context object, the module has access to all of the OSGi functionality for modularity, lifecycle, and services. In short, it is a fairly important object for most bundles, but we will 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 overall lifecycle and gain access to the underlying OSGi framework.

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

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

Notice we also need to import the `org.osgi.framework` package, since our bundle activator has a dependency on it. Otherwise, it is pretty simple to make bundles lifecycle aware. To see this example in action, go into the `greeting-example/lifecycle/` directory for Chapter 1 in the accompanying code and type “ant” to build it and “java -jar main.jar” to run it.

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

### 1.3.3 Service layer

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

```
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 service interface above 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 is not known to the client. For example, one implementation may print its greeting textually, while another may display its greeting in a GUI dialog box. The code in Listing 1.7 depicts a simple text-based implementation.

#### Listing 1.7 Greeting implementation

```
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 might be thinking that nothing in the service interface or Listing 1.7 indicate we are defining an OSGi service. Well, you'd be correct. That's what makes the OSGi's service approach so natural if you are already following an interface-based approach, since your code will largely stay the same. There are two places where your development will be a little different. One is how you make a service instance available to the rest of your application and the other is how the rest of your application discovers the available service.

All service implementations will ultimately be packaged into a bundle and that bundle will need to be lifecycle aware in order to register the service; this means that we need to create a bundle activator for our example service as depicted in Listing 1.8.

#### Listing 1.8 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) {} #B
}

```

This time in the `start()` method at #A, instead of storing the `Greeting` implementation as a singleton, we use the provided bundle context to register it as a service with the service registry. The first parameter we 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 at #B we could unregister the service implementation before stopping the bundle, but in practice you don't need to do this. The OSGi framework will automatically unregister any registered services when a bundle stops.

We've seen how to register a service, but what about discovering a service? Listing 1.9 shows a very simplistic client, which does not handle missing services and suffers from potential race conditions. A more robust way to access services will be discussed in chapter [ref ch4].

#### Listing 1.9 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()); #A

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

    public void stop(BundleContext ctx) {}
}

```

The first thing you'll notice is that accessing a service in OSGi is a two-step process. First at #A, an indirect reference is retrieved from the service registry, which points to the earliest active service that was registered under the given interface name. Second at #B, this indirect reference is used to access the actual service object instance. The service reference can safely be stored in a member variable, but in general you should never store service object instances since this will make your application less dynamic and stop bundles from being cleanly uninstalled. We say "in general" since there are certain advanced use cases discussed in later chapters [ref xx] where you may wish to store a reference to a service. But in these situations you must be very careful to de-reference the service when the OSGi framework tells you it is no longer valid.

Also note that the client code only needs to declare an import for the `Greeting` interface, it has no direct or explicit dependency on the actual service implementation(s). This makes it very easy to swap services dynamically without restarting the client bundle.

Both the service implementation and client should be packaged into separate bundle JAR files. Each bundle will name an activator in their respective bundle metadata, but only the service implementation will export the `org.foo.hello` package, whereas the client will import it. To see this example in action, go into the `greeting-example/service/` directory for Chapter 1 in the accompanying code and type `ant` to build it and `java -jar main.jar` to run it.

Now that we have seen some examples, it is possible for us to 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, then use the modularity layer. If you want a way to initialize your modules and interact with the modularity layer, then use both the modularity and lifecycle layer. If you want a dynamic, interface-based development approach, then 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 will use a simple paint program, whose user interface is depicted in Figure 1.4. The paint program is not intended to be independently useful; rather, it is 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 predefined colors. Available shapes are displayed as

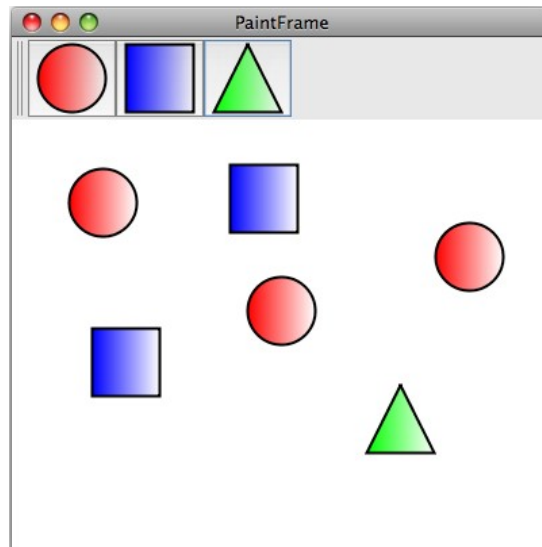


Figure 1.4 Simple paint program user interface

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 value of using a visual program for demonstrating these concepts will become evident when we start introducing run-time dynamism.

We have finished our overview of the OSGi framework and are ready to delve into the details, but before we do let's try to put OSGi in context by discussing similar or related technologies. While no Java technology fills the exact same niche as OSGi, there are several treading similar ground, so it is 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 is actually in a fairly unique position in the Java world. Over the years, no single technology addressed OSGi's exact problem space, but there has been overlaps, complements, and offshoots. While it is not possible to cover how OSGi relates to every conceivable technology, we will try to address some of the most relevant in roughly chronological order.

### **1.4.1 Java Enterprise Edition**

Java2 Platform, Enterprise Edition (J2EE, now simply called JavaEE) has roots dating back to 1997. One might expect a long history of comparison and debate between OSGi and J2EE, but in reality there was never really many comparisons made, since both were targeting opposite ends of the computing spectrum (i.e., enterprise vs. embedded markets). Only within the last couple years has OSGi technology really started to take root in the enterprise space. Taken in total, the J2EE API stack is not really related to OSGi. The Enterprise JavaBeans (EJB) specification is probably the closest comparable technology from the J2EE space, since it defines a component model and packaging format. However, 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 do not address the full component lifecycle, nor did they support clean modularity concepts. OSGi is now moving into the J2EE space to provide a more sophisticated modularity layer beneath these existing technologies. Since the two ignored each other for so long, though, there are some challenges in moving existing J2EE 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 several application servers, such as IBM's Websphere, Redhat's JBoss, BEA's Weblogic, Sun's GlassFish, and ObjectWeb's JOnAS.

### **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. In typical Jini fashion, consumers access clients through some form of proxy using a remote procedure call mechanism, like RMI. The service lookup registry is also a remotely accessible, federated service. The Jini model assumes remote access across multiple VM processes, while OSGi assumes everything occurs in a single VM process. However, in stark contrast to OSGi, Jini does not define any modularity mechanisms and relies on the run-time code loading features of RMI. This is not a criticism of Jini, it is simply intended for a different purpose. 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 run-time platform for Java, has a long history of having a very modular design. Sun purchased NetBeans back in 1999 and has continued to evolve it. The NetBeans platform actually has a lot in common with OSGi. It defines a fairly sophisticated module layer and also promotes interface-based programming using a “lookup” pattern which is quite similar to the OSGi service registry. While OSGi focused on embedded devices and dynamism, the NetBeans platform was originally just an implementation layer for the IDE. Eventually the platform was promoted as a separate tool in its own right, but focused on being a complete GUI application platform with abstractions for file systems, windowing systems, and much more. NetBeans was never really seen as being comparable to OSGi, even though it is; perhaps OSGi's more narrow focus was 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 003, 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 component model for this purpose. If you are wondering how or why a technology for managing and monitoring applications was compared to OSGi, then you're on the right track. JMX is not really comparable to OSGi; it is actually complementary, since 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 so 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, who adopted and extended JMX for use as a modularity layer in its application server (since eliminated in JBoss 5). Nowadays, JMX is not (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 employs 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 is 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 were offering a sophisticated dynamic module layer like OSGi.

There is now significant movement from IoC vendors to port their infrastructure to the OSGi framework. The goal of these endeavors is to make IoC patterns run on top of the OSGi modularity, lifecycle and service layers. There are some unavoidable challenges in moving from a model with global access to the class loader and a static service dependency graph to a modular environment with a dynamic lifecycle, but a lot of progress is being made.

### **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 integration (B2B) solutions. In JBI, "plugin" components provide and consume services once they are plugged into the JBI framework. Components do not directly interact with services, like in OSGi; instead, they communicate indirectly using normalized 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 actually made more sense for JBI to leverage OSGi's more sophisticated modularity, which is ultimately what happened in Project Fuji from Sun and ServiceMix from Apache. Work on JBI 2.0, started in 2007, will further attempt to properly position JBI with respect to OSGi.

### **1.4.7 JSR 277**

In 2005, Sun announced a new JCP specification, called JSR 277, 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, since 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 two-fold: 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 since it started from the OSGi R4 specification and resulted in the R4.1 specification release. During this period OSGi technology continued to gain momentum, while 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, more simply described as a package of packages. The specification of superpackages got bogged down in various details until they were scrapped in favor of simply 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. However, when it became apparent in 2008 that JSR 277 would be put on hold, JSR 294 was pulled back out. Currently, its scope is still evolving. With JSR 277 on hold, Sun introduced an internal project, called Project Jigsaw, to modularize the JDK itself. The details of Jigsaw are still evolving, but this point it is intended to be an implementation detail of Sun's JDK.

#### **1.4.9 Service Component Architecture**

Another comparable technology is Service Component Architecture (SCA), which started 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. It's component model is more advanced since it defines composite components (i.e., a component 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 (e.g. Java, BPEL, EJB, C++) and integrated using various bindings (e.g. SOAP/HTTP, JMS, JCA, IIOP). SCA does define a standard packaging format, but it does not define a sophisticated modularity layer like OSGi provides. OSGi and SCA are complementary in a handful of ways. For example, the SCA specification leaves open the possibility of other packaging formats, which makes it possible to use OSGi as a packaging and modularity layer for Java-based SCA implementations; Apache Tuscany and Newton are examples of an SCA implementation leveraging OSGi. Additionally, 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 is not a Java technology, it deserves mention since it was largely inspired by Java and did improve on it in ways which are similar to how OSGi improves Java. Released in 2002, Microsoft not only learned from Java's example, but they also learned from their 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 is not as flexible as OSGi's. Since dependencies are on specific assemblies, the OSGi notion of provider substitutability is not attainable.

Additionally, at run 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, since multiple assemblies loaded into the same application domain, must all be unloaded if one of them is uninstalled. It is possible to load assemblies into separate domains, but then communication across domains must use inter-process 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 not a significant amount of details in common. Regardless, .NET deserves credit for improving on Java in this area.

## **1.5 Summary**

The Java platform is great for developing all sorts of applications, but it does not do a good job of supporting modularity, which is critical for the long-term success of your projects. 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. New projects can define and enforce separation of concerns from the outset, while existing projects can leverage OSGi modularity by simply modifying how they are packaged with few, if any, changes to the code itself. With this high-level understanding of OSGi technology, we can dive into the details of the modularity layer in chapter 2, which is the foundation of everything else in the OSGi world.