QA 76
.73
.J38
0827
2011
Copy 1



Creating modular applications in Java

OSGi IN ACTION

Richard S. Hall
Karl Pauls
Stuart McCulloch
David Savage

FOREWORD BY PETER KRIENS

MANNING

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

OSGi in Action

Creating Modular Applications in Java

RICHARD S. HALL
KARL PAULS
STUART McCULLOCH
DAVID SAVAGE



MANNING
Greenwich
(74° w. long.)

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
180 Broad Street, Suite 1323
Stamford, CT 06901
Email: orders@manning.com

©2011 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

♾ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.


Manning Publications Co.
180 Broad Street, Suite 1323
Stamford, CT 06901

Development editor: Cynthia Kane
Copyeditor: Tiffany Taylor
Typesetter: Gordan Salinovic
Illustrator: Martin Murtonen
Cover designer: Marija Tudor

ISBN 9781933988917
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 16 15 14 13 12 11



Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

about this book

The OSGi specifications are well written and elaborate, so if you need to know details about OSGi technology, the specifications are the place to look. If you do, you'll discover that they were written for someone who is going to implement the specifications, not use them. This book started out as an attempt to remedy this situation by creating a user-oriented companion guide for the specifications. Our goal wasn't to create an OSGi cookbook but to thoroughly describe the important aspects of OSGi and show how to use them. Our main idea was to more simply explain the OSGi specifications by ignoring the implementation details and including additional usage information.

To that end, we've tried to limit ourselves to discussing the most common concepts, features, and mechanisms needed to work with OSGi technology throughout the book. That doesn't mean we were able to avoid all the esoteric details. As you'll find when you begin working with OSGi, it enforces a new level of strictness when it comes to modularity, which will likely break some of your old practices. In the end, you need to understand what's going on under the covers in some places to be able to effectively debug and diagnose the situations in which you find yourself.

As our writing progressed, the book chapters began to separate naturally into three parts:

1. Explaining the core OSGi specification
2. Describing how to work with the specification in practice
3. Introducing advanced OSGi-related topics

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

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

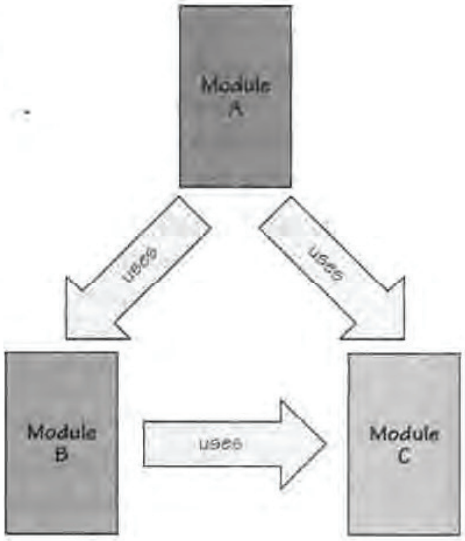


Figure 1.1 *Modularity* refers to the logical decomposition of a large system into smaller collaborating pieces.

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

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

Greeting.java

```
package org.foo.hello;

public interface Greeting {
  void sayHello();
}
```

❶ Simple interface

GreetingImpl.java

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

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

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

In addition to merging the exported and imported packages and required bundles, the bundle class paths are also merged. This impacts the overall class search order for the bundle, like this:

1. Requests for classes in `java.` packages are delegated to the parent class loader; searching stops (section 2.5.4).
2. Requests for classes in an imported package are delegated to the exporting bundle; searching stops (section 2.5.4).
3. Requests for classes in a package from a required bundle are delegated to the exporting bundle; searching continues with a failure (section 5.3.1).
4. The host bundle class path is searched for the class; searching continues with a failure (section 2.5.4).
5. Fragment bundle class paths are searched for the class in the order in which the fragments were installed. Searching stops if found but continues through all fragment class paths and then to the next step with a failure.
6. If the package in question isn't exported or required, requests matching any dynamically import package are delegated to an exporting bundle if one is found. Searching stops with either a success or a failure (section 5.2.2).

This is the complete bundle class search order, so you may want to mark this page for future reference! This search order makes it clear how fragments support split packages, because the host and all fragment class paths are searched until the class is found.

Fragments and the `Bundle-ClassPath`

The bundle class path search order seems fairly linear, but fragments do introduce one exception. When specifying a bundle class path, you can specify embedded JAR files, such as

```
Bundle-ClassPath: .,specialized.jar,default.jar
```

Typically, you'd expect both of these JAR files to be contained in the JAR file of the bundle declaring them, but this need not be the case. If fragments aren't involved, the framework ignores a non-existent JAR file on the bundle class path. But if the bundle has fragments attached to it, the framework searches the fragments for the specified JAR files if they don't exist in the host bundle.

In the example, imagine that the host contains default.jar but doesn't contain specialized.jar, which is contained in an attached fragment. The effect this has on the class search order is that the specified fragment content is searched before some of the host bundle content.

Sometimes this is useful if you want to provide default functionality in the host bundle but be able to override it on platforms where you have specialized classes (for example, using native code). You can also use this approach to provide a means for issuing patches to bundles after the fact, but in general it's better to update the whole bundle.

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

8.3 Tracking down memory leaks

Memory leaks occur in OSGi applications as in any other Java application. All you need is something like a rogue thread or static field hanging on to one end of a spaghetti ball of references to stop the garbage collector from reclaiming the objects. In a desktop Java application, you may not notice any memory leaks because you don't leave the application running for a long time. As soon as you restart the JVM, your old application with its ever-growing heap of objects is gone, and you get a brand-new empty heap to fill.

Server-side OSGi applications, on the other hand, can have longer lifetimes; an uptime of many months isn't unreasonable. One of the strengths of OSGi is that you're able to install, update, and uninstall bundles without having to restart the JVM. Although this is great for maximizing uptime, it means you have to be careful not to introduce memory leaks in your bundles. You can't always rely on the process being occasionally restarted. Furthermore, updating a bundle introduces a new class loader to hold the updated classes. If there's anything holding on to objects or classes from the old class loader, it won't be reclaimed, and your process will use more and more class loaders each time the bundle is updated or reinstalled.

Introducing the PermGen heap

Class-loader leaks can be more problematic than simple object leaks because some Java runtimes, like Sun's HotSpot JVM, place classes in a separate heap space called the Permanent Generation, or PermGen for short. This class heap is much smaller than the main object heap, and its capacity is controlled by a different GC setting: `-XX:MaxPermSize`. If every bundle update adds hundreds of new class revisions without unloading anything, you'll probably exhaust the PermGen before you run out of object heap space.

Any leak is a cause for concern, but depending on your requirements, not all leaks warrant investigation. You may not even notice certain leaks if they add only a few bytes to the heap every now and then. What's the best way to find leaks in an OSGi application?

8.3.1 Analyzing OSGi heap dumps

As with debugging, you can continue to use your existing heap analysis skills to examine OSGi applications. Sure, there are more class loaders than in a normal Java application; but standard Java EE containers also contain multiple class loaders, and that doesn't stop developers from finding memory leaks inside web applications.

Let's see what an OSGi application heap dump looks like. The leaky application is under chapter08/memory-leaks in the online examples. It consists of a single bundle that creates and accesses a `ThreadLocal` variable every time the bundle starts and fails to remove it when the bundle stops. Here's the bundle activator.

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

To be able to show all of what deployment packages have to offer, let's assume you want to provide a core version of the program containing the drawing frame and the shape API bundles. This way, you're able to deploy the actual shape implementations separately via an extension pack. The extension pack contains the square, circle, and triangle bundles. Let's go with this approach and explore the different ways you can use deployment packages to make it work.

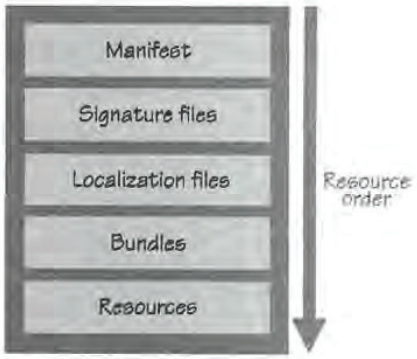


Figure 10.4 Structure of a deployment package JAR file

The general structure of a deployment package is shown in figure 10.4. This ordering is carefully designed to allow deployment packages to be streamed in such a way that the contents can be processed without needing to download the entire JAR file.

The deployment package design has a few other desirable characteristics. First, the deployment package puts metadata in its manifest, similar to bundles, which allows you to turn it into a named and versioned set of resources. Second, by taking advantage of the fact that JAR files can be signed, you can use signed JAR files to make your deployment packages tamperproof.

For this example, you can do either of the following (see figure 10.5):

- Create a deployment package for the core bundles and one package for all shape bundles.
- Create a deployment package for the core bundles and individual deployment packages for each shape bundle.

The difference is obviously that in the first case, you'll deploy either all shapes or none; and in the second case, you can extend the core bundle piecemeal. The important point to understand, though, is that you can't use both approaches at the same time: you must choose one.

In terms of the example, you need to make a decision. In this case, you'll go with the first approach and create a single deployment bundle for all shapes. But because deployment packages can be updated, you can gain some flexibility by starting with only one shape in the deployment package and then adding another one in an updated version and

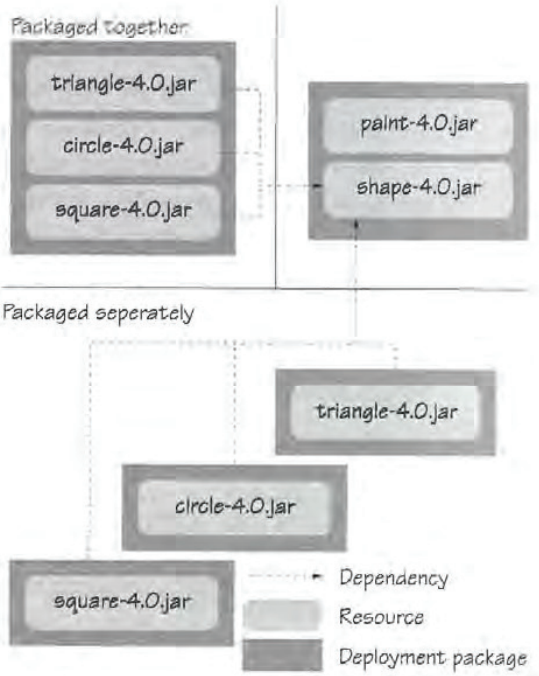


Figure 10.5 Paint program packaging alternatives

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

Deployment packages are greedy

These two different packaging strategies can't be used simultaneously. The specification only allows resources to belong to a single resource package. Using both approaches at the same time or changing your approach after the fact would move ownership of the bundle resources to another deployment package and thus violate the specification.

A deployment package is defined as a set of resources that must be managed as a unit. The resources in a deployment package are assumed to be tightly coupled, such as a bundle and its configuration data. As a consequence, a resource can belong to only one deployment package; otherwise, for example, you could run into situations where you had two different, conflicting configurations for the same bundle.

another for the third or other combinations. When you create an update that adds or removes resources from a previous version, you don't even have to package the resources inside the update; instead, you can use *fix packages.*

FIX PACKAGE A deployment package that minimizes download time by excluding resources that aren't required to upgrade or downgrade a deployment package. It can only be installed if a previous version of that deployment package is already installed. A fix package contains only the changed and new resources. A fix package (called the *source*) therefore must specify the range of versions that the existing deployment package (called the *target*) must have installed. You'll see this shortly when we walk through the example.

Let's assume that you want to be able to add new shapes to the application when they become available. In this scenario, it makes sense to start with a core deployment package and create fix packages, adding new shapes as they become available.

Now that you've figured out your packaging approach, how do you proceed? You need to create a manifest for the target that contains the paint frame and shape API bundles; you'll use this to provision the paint program core. Then you need to create the manifest of the fix package that you'll use to add the three shape bundles to the core. When you have your manifests, you need to create two JAR files with the corresponding manifests and your bundles, you can optionally sign them, and you're good to go. Here's the manifest of the core deployment package:

```
Manifest-Version: 1.0
DeploymentPackage-SymbolicName: org.foo.paint
DeploymentPackage-Version: 1.0.0

Name: paint-4.0.jar
Bundle-SymbolicName: org.foo.paint
Bundle-Version: 4.0.0

Name: shape-4.0.jar
Bundle-SymbolicName: org.foo.shape
Bundle-Version: 4.0.0
```

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

You first specify the deployment package's symbolic name and version. Next, you specify the list of resources contained in the JAR file. You specify the name of a resource, its symbolic name, and its version; you must do this for each resource. For this example, you only have bundle resources. To finish, you need to use the `jar` tool to create the JAR file with the appropriate content, and you're finished with your first deployment package.

Signing deployment packages

In this example, you don't sign your deployment package, nor is it required for you to do so. If you want to create a signed deployment package, you use the `jarsigner` tool from the standard Java SDK. The signing process is no different than signing a normal JAR file; it results in the signatures being placed in the deployment package JAR file in the META-INF directory and after the MANIFEST.MF file. Additionally, each entry section in the manifest contains a digest entry.

Now you need to create the manifest for your fix package containing the shape bundles. This manifest is as follows:

```
Manifest-Version: 1.0
DeploymentPackage-Symbolicname: org.foo.paint
DeploymentPackage-Version: 2.0
DeploymentPackage-FixPack: [1,2)

Name: paint-4.0.jar
Bundle-SymbolicName: org.foo.paint
Bundle-Version: 4.0.0
DeploymentPackage-Missing: true

Name: shape-4.0.jar
Bundle-SymbolicName: org.foo.shape
Bundle-Version: 4.0.0
DeploymentPackage-Missing: true

Name: triangle-4.0.jar
Bundle-SymbolicName: org.foo.shape.triangle
Bundle-Version: 4.0.0

Name: circle-4.0.jar
Bundle-SymbolicName: org.foo.shape.circle
Bundle-Version: 4.0.0

Name: square-4.0.jar
Bundle-SymbolicName: org.foo.shape.square
Bundle-Version: 4.0.0
```

Because the fix package is an update to your core package, the symbolic name stays the same, but the version is upgraded to `2.0.0`. The `DeploymentPackage-FixPack` header indicates that this is a fix package; you use version-range syntax to indicate that the fix package can be applied to any previously installed version of the deployment package from `1.0.0` inclusive to `2.0.0` exclusive. This version-numbering scheme expresses the assumption that only major version-number changes indicate added

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

bundles. You don't need to package the bundles already present in the core package, but you still need to mention them in the manifest. You use the `DeploymentPackage-Missing` header to do this. Then you specify the shape bundles in the same fashion as before. To use the deployment packages, you need to make each available via a URL.

> **NOTE** If you make deployment packages available via a protocol that supports MIME types, the standard MIME type for deployment packages is `application/vnd.osgi.dp`.

Next, you can use the provided `DeploymentAdmin` service in your management agent to install, update, and uninstall deployment packages.

MANAGING DEPLOYMENT PACKAGES

To demonstrate how a management agent can use Deployment Admin, you'll again return to the shell and create a new `dpa` shell command to list, install, and uninstall deployment packages. This command will use the `DeploymentAdmin` service, which is represented by the following interface:

```
public interface DeploymentAdmin {
  DeploymentPackage installDeploymentPackage(InputStream in)
    throws DeploymentException;
  DeploymentPackage[] listDeploymentPackages();
  DeploymentPackage getDeploymentPackage(String symbName);
  DeploymentPackage getDeploymentPackage(Bundle bundle);
  boolean cancel();
}
```

The following listing shows the implementation of the command.

Listing 10.4 Deployment Admin shell command example

```
public class DeploymentPackageCommand extends BasicCommand {

  public void exec(String args, PrintStream out, PrintStream err)
    throws Exception {
    DeploymentAdmin admin = getDeploymentAdmin();

    if (admin == null) {
      out.println("No DeploymentAdmin service found.");
      return;
    }
    if (args != null) {
      if (args.trim().equalsIgnoreCase("list")) {
        for (DeploymentPackage dp : admin.listDeploymentPackages()) {
          out.println(dp.getName() + " " + dp.getVersion());
        }
      } else if (args.trim().startsWith("uninstall ")) {
        DeploymentPackage dp = admin.getDeploymentPackage(
          args.trim().substring("uninstall ".length()));
        if (dp != null) {
          dp.uninstall();
        } else {
          out.println("No such package");
        }
```

❶ Gets installed deployment packages

❷ Uninstalls deployment package

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

14 Securing your applications

This chapter covers

- Providing an overview of the Java security architecture
- Using OSGi Conditional Permission Admin to manage permissions
- Signing bundles and granting permissions based on bundle signers
- Creating custom permission conditions for advanced use cases

OSGi allows you to create loosely coupled, extensible applications. In such applications, bundles can come and go at any time, and it's easy to allow third parties to extend your application in a well-defined way. But as with most things in life, there's a downside to this flexibility: you open yourself (or your users) to security vulnerabilities because third-party bundles can't be completely trusted.

Luckily, the Java platform has built-in support for secure sandboxes, and the OSGi framework is designed to take advantage of them. Unfortunately, secure

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

sandboxes and their restrictions are difficult to get right and often hard to manage. This is especially true in an environment as dynamic as the OSGi framework. To help with this situation, OSGi defines an extensive and powerful security model to ease security management.

In this chapter, we'll familiarize you with the Java security model and how OSGi uses it to securely deploy and manage applications. You'll learn how to create secure applications and well-behaved bundles for security-enabled OSGi frameworks. Before we start with that, let's look at some general issues you'll need to consider when trying to secure your applications.

14.1 To secure or not to secure

Modern applications and software solutions increasingly center around loosely coupled, extensible architectures. Component and service orientation are applied to almost all areas of application development including distributed systems, ubiquitous computing, embedded systems, and client-side applications. One of the main drawbacks of loosely coupled, extensible applications is the potential security issues around executing untrusted code. There are two common reasons for running untrusted software:

- *Permission management is often extremely complicated.* Often, users are left to make security policy decisions, and they're typically unable to assess the impact of granting a given permission. Further, because the user is typically using an application to perform some task, security is largely viewed as an obstacle because it doesn't contribute to the task at hand.
- *It's inherently tricky to establish meaningful identity of third-party software providers.* It's often necessary to differentiate between providers or types of providers to properly grant or deny permissions. Often the origin of the software artifact (where it came from) is used for this purpose, but techniques like digital signatures are also needed to ensure that the software hasn't been tampered with. Digital signatures introduce the complicated process of creating and maintaining certificates and trust between certificates, which can be onerous for both users and developers.

This raises perhaps the biggest issue with securing code: it adds another burden to development. Even if you don't plan to run with security enabled, your code has to be aware of security if you want it to be possible for other people to use the code when security is enabled. Then, to make matters worse, if you decide to enable security, the fine-grained security checks impose an execution-time performance penalty.

Despite these issues, security isn't something that can or should be ignored, because plenty of people are willing to take advantage of software vulnerabilities. What do you need to do? Providing meaningful security management involves three key aspects:

- Establishing identity
- Establishing policies to manage permissions
- Performing permission checks and privilege management

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

As we mentioned previously, identity can be established by the location or origin of the software artifact or by cryptographic measures using digital certificates. Especially for the latter approach, the software provider generally needs to make the code available in such a way that you can establish the needed credentials. When you have identity established, you need to define the permissions that code should have. For OSGi, this is the responsibility of whoever is managing the framework, which can be a gateway operator, a server administrator, or even an end user. As a consequence, permission management should be kept as simple as possible. Last but not least, security must be built into the code itself. You have to think about internal security checks to prevent external code from performing undesired operations and also how to limit privileges so external code can perform potentially sensitive operations in a safe way.

Assuming you're able to develop your code with all the security checks in the right place, define a reasonably policy to manage permissions, and sign it using a trusted certificate for establishing identity, is all the work worth it?

Clearly, it depends on your specific situation. In some cases, it's not within the scope of your application. Either the performance impact is too great or the development costs are too high. Often, these issues serve as the determining factor for creating security-enabled applications. This is compounded by the fact that if code isn't *designed* to be usable in security-enabled environments, it's unlikely to happen by accident. This results in a catch-22 type of situation, where the difficulty associated with creating secure code results in security being ignored, which makes it next to impossible to use such code with security enabled, thus further raising the barriers for deciding to develop with security in mind in the first place.

All hope isn't lost. In the remainder of this chapter, we'll show you that taking advantage of the security capabilities of the OSGi framework needn't be too difficult. In the next section, we'll start by taking a high-level view of Java and OSGi security.

14.2 Security: just do it

So you want to secure your OSGi-based application. Great—but where do you start? Let's begin at the beginning and look at the Java security architecture and its permission model, on which the OSGi security model is based.

14.2.1 Java and OSGi security

It's important to understand the Java security architecture; but to keep this chapter tightly scoped, we'll introduce only the parts needed to understand the remainder of the chapter. Welcome to Java security boot camp!

The Java security architecture is fundamentally about the assignment of `java.security.Permission` subclass objects to code. Specific permissions allow access to specific, sensitive operations, such as file system or network socket access. How do you grant `Permission` objects to code? The Java security architecture is based on the two fundamental concepts of domain- and role-based security:

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

- Domain-based security revolves around granting code permissions based on its origins (also referred to as its *code base*).
- Role-based security revolves around authenticating users or processes and granting them permissions based on who they are.

The OSGi framework security model relies on Java's domain-based approach; the role-based approach is possible, but only as a layer on top. In standard Java, role-based security is provided by the Java Authentication and Authorization Service (JAAS) framework, but OSGi also provides its own API in the form of the User Admin Service. We won't deal with role-based security in this chapter; for more information on the User Admin Service, refer to the OSGi compendium specification. Now, let's delve a little deeper into domain-based security.

PERMISSIONS

The Java permission model is fairly simple. The `Permission` class is a base class from which more specific permissions can be derived via subclassing. You grant `Permission` objects to code to give it the ability to perform sensitive operations. Additionally, the `Permission` class has a method called `implies()` that accepts another `Permission`. This method checks to see if the supplied permission is implied by the target permission (similar to being a subset). Thus, `Permission` objects are used to both grant and check permissions.

PROTECTION DOMAINS

You grant permissions to code, but how are they associated with it? For domain-based security, Java uses a special concept called a *protection domain*, which is represented by the `java.security.ProtectionDomain` class, to encapsulate the security characteristics of a domain. Permissions are granted to protection domains, and all classes belong to a single protection domain. Sound complicated? Actually, in OSGi it's pretty simple, because a domain is mapped one-to-one with a bundle; you can think of it as a bundle protection domain. All classes originating from a given bundle are members of the same bundle protection domain.

> **BUNDLE PROTECTION DOMAIN** Maintains a set of permissions granted to a given bundle. All classes loaded from a bundle are associated with the bundle's protection domain, thus granting them the permissions granted to the bundle.

To understand how protection domains enable permission checking, consider code that performs a sensitive operation, such as creating a file. The code in the JRE for file access performs security checks internally to make sure the invoking code has permission to perform the operation. Internally, the code associated with performing file system operations triggers a specific permission check by using the security-checking methods of `SecurityManager` or `AccessController`. When triggered, the JVM collects the `ProtectionDomains` of all classes on the call stack leading to the invocation of the sensitive operation. It checks that each protection domain on the call stack has at least one permission implying (granting) the specific permission being checked by the method. Figure 14.1 shows how this looks in practice.

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

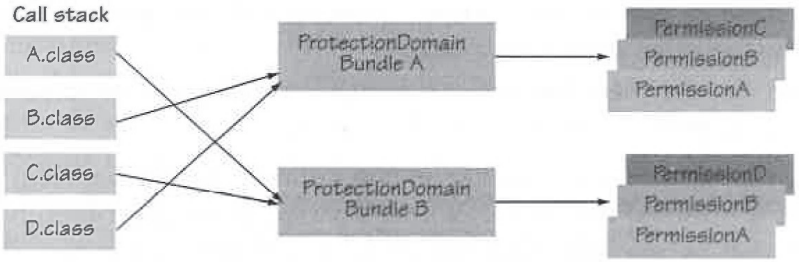


Figure 14.1 The JVM checks permissions by collecting all protection domains associated with classes on the call stack and seeing if each involved protection domain has the specified permission granted to it.

In this case, assume that class `D` performs a sensitive operation that triggers an internal permission check using the `AccessController.checkPermission(Permission p)` method. This checks whether at the point of the permission check, all protection domains on the call stack have permission `p`. Looking at figure 14.1, the JVM performs a stack walk from the class performing the security check and determines that classes `A`, `B`, `C`, and `D` are involved. Subsequently, it determines that classes `A` and `C` originate from the protection domain of Bundle B, and classes `B` and `D` originate from the protection domain of Bundle A. With this information, the JVM checks whether all protection domains have some permission implying the checked permission. If not, then a security exception is thrown.

This provides a good foundation for understanding the Java security architecture, but there's one final piece to the puzzle: privileged calls.

PRIVILEGED CALLS

You now know that checking a specific permission triggers a complete stack walk to collect all involved protection domains and verify that they all imply that permission. This is useful, but it's too restrictive by itself. For example, assume you have a service with a method for appending a message to a log file. Because disk operations trigger file system–related permission checks, all code on the call stack must have permission to write to the file. This may be fine if only trusted code is involved; but in an extensible and collaborative environment like OSGi, you generally want to allow arbitrary bundles to share code and services, so it's likely that some code on the call stack won't be trusted.

In such cases, if a permission check always walks up the entire call stack, you either have to disallow all nontrusted code or grant code-sensitive permissions to untrusted code. Neither choice is palatable, which is why Java supports privileged calls. A *privileged call* is a mechanism to short-circuit the stack walk when performing a permission check. In practice, this allows trusted code to perform sensitive operations on behalf of code with insufficient permissions.

You achieve this by using the `AccessController.doPrivileged()` method, which takes a `PrivilegedAction` instance (it has a `run()` method similar to a `Runnable`). When the `doPrivileged()` method is invoked, it invokes the `run()` method of the passed-in `PrivilegedAction`. Any subsequent permission checks triggered by the `PrivilegedAction` stop walking the call stack at the last privileged call. Thus, only the protection domains from the privileged action onward are considered by subsequent permission checks.

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

Returning to the example of a service for appending a message to a log file, you trust the bundle containing the service implementation, but you don't want to give direct file system access to anyone else. To do this, your service must encapsulate its file system operations inside a `PrivilegedAction` and use `doPrivileged()` like this:

```
public void append(String msg) {
  SecurityManager sm = System.getSecurityManager();
  if (sm != null) {
    AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
        doFileAppend();
      }
    });
  }
  else {
    doFileAppend();
  }
}
```

Any triggered permission checks stop walking the call stack at the `run()` method, which means nontrusted code further up the stack won't have its protection domain checked for the triggered permissions. Pushing this example further, you may decide to limit which code can call the `append()` method. To so this, you can create your own `Permission` subclass, which you can grant to code. For the append method, if you create an `AppendPermission`, it can check the permission before performing the privileged call:

```
public void append(String msg) {
  SecurityManager sm = System.getSecurityManager();
  if (sm != null) {
    sm.checkPermission(new AppendPermission());
    AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() { doFileAppend(); }});
  } else {
    doFileAppend();
  }
}
```

Here your service asks the `SecurityManager` to check whether the code on the call stack has been granted the custom `AppendPermission`. If so, it can continue to perform the file-append operation; otherwise, a security exception is thrown.

WARNING You may have noticed that you check whether the security manager is `null` before performing security checks. You do it this way because you want to perform security checks only if security is enabled, to avoid performance penalties when it's not enabled.

That pretty much sums up the important pieces of the Java security architecture. These mechanisms provide for flexible, fine-grained security management. A potential downside is that managing all these permissions can be complex. Luckily, the OSGi specification lessens some of this complexity by defining services to help you perform

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

permission management. We'll look at these services shortly; first, let's examine OSGi-specific permissions defined by the OSGi specification.

14.3 OSGi-specific permissions

Certain methods in the OSGi framework API perform sensitive operations or provide access to sensitive information. To control which code can access these sensitive methods, the OSGi specification defines a few custom permissions, as you learned about in the last section. You can group these permissions by the layers of the OSGi framework, as shown in table 14.1.

Table 14.1 Permissions defined by the OSGi specification

Layer	Permission(s)
Module	`PackagePermission`—Controls which packages a bundle is allowed to import and/or export
	`BundlePermission`—Controls which bundles a bundle is allowed to require
Lifecycle	`AdminPermission`—Controls which bundles are allowed to perform sensitive lifecycle operations
Services	`ServicePermission`—Controls which services a bundle is allowed to publish and/or use

We'll introduce these OSGi permissions in the following subsections, and you'll subsequently use them when we discuss permission management.

Names and actions

Standard Java permissions typically have constructors that accept two parameters: a `name` string and an `actions` string. The meaning of these parameters is determined by the specific permission. For example, `java.io.FilePermission` expects a file path for `name` and a comma-delimited value that may include `READ`, `WRITE`, `EXECUTE`, or `DELETE` for actions. The combination of `name` and `actions` allows you to express everything the permission allows you to control. All of the OSGi-specific permissions follow this pattern, as you'll see.

14.3.1 PackagePermission

`PackagePermission` is a module-layer permission giving you the ability to limit which packages a bundle can import or export. For example, we discussed how methods can use `AccessController.doPrivileged()` to allow code with insufficient privileges to perform sensitive operations. You may not want any arbitrary code using the packages containing these privileged operations. In that case, you can use `PackagePermission` to limit which bundles can import the packages containing the associated classes. Likewise, you can use `PackagePermission` to control which bundles can export a given package, because you may only want trusted bundles providing some packages.

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

To grant a specific `PackagePermission`, you need to supply the `name` and `actions` parameters for its constructor; these parameters are described in table 14.2.

Table 14.2 `PackagePermission` constructor parameters

Parameter	Description
`String name`	Name of the package or packages to which this permission applies
`String actions`	Comma-delimited list of the actions granted by the permission (`export`, `import`, or `exportonly`)

For convenience, you can use `*` or a trailing `.*` as a wildcard to target several packages with a single permission. For the actions, `import` gives a bundle permission to import the named packages, `export` gives a bundle permission to export and import the package, and `exportonly` does as its name implies. You may wonder why `export` also gives permission to import the named packages. It's to support bundles' ability to import packages they export (that is; substitutable exports), as described in section 5.1.1.

To get an idea of how `PackagePermission` works, let's take a conceptual look at how the framework uses it. Assume you have a bundle with the following imports and exports:

```
Import-Package: org.foo, org.bar
Export-Package: org.bar
```

When the framework resolves this bundle, it checks to see whether the bundle has the following permissions granted to it:

- `PackagePermission.IMPORT` permission for the `org.foo` package
- `PackagePermission.IMPORT` permission for the `org.bar` package
- `PackagePermission.EXPORT` permission for the `org.bar` package

For these checks to succeed, you'd have to grant the necessary permissions, such as

```
new PackagePermission("org.foo", PackagePermission.IMPORT);
new PackagePermission("org.bar", PackagePermission.EXPORT);
```

Notice that you don't need to grant the bundle permission to import `org.bar`, because it's implied by the export action.

That's the basics for `PackagePermission`; let's move on to the next OSGi permission.

14.3.2 *BundlePermission*

Similar to `PackagePermission`, `BundlePermission` is a module-layer permission for controlling bundle and fragment dependencies. To grant a `BundlePermission`, you need to construct it with the parameters shown in table 14.3.

As with `PackagePermission`, you can use `*` or a trailing `.*` as a wildcard to target several packages with a single permission. To control bundle dependencies, the `provide` action gives a bundle permission to be required by bundles matching the supplied

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

Table 14.3 `BundlePermission` constructor parameters

Parameter	Description
`String symbolicName`	Symbolic name of the bundle to which this permission applies
`String actions`	Comma-delimited list of the actions granted by the permission (`provide`, `require`, `host`, or `fragment`)

symbolic name, whereas `require` gives it permission to require matching bundles. Fragment dependencies are controlled by the `host` and `fragment` actions, which give a bundle the ability to be a host bundle for matching fragments or be a fragment for matching hosts, respectively. Another similarity to `PackagePermission` is that the `provide` action implies the `require` action.

Using `BundlePermission` isn't sufficiently different from using `PackagePermission`, so we won't look into it any further. Instead, we'll move on to the next OSGi permission.

14.3.3 AdminPermission

`AdminPermission` is a lifecycle-layer permission to control access to sensitive framework operations and information. The operations and information protected by `AdminPermission` are diverse, which makes it somewhat complex but fairly powerful. Table 14.4 shows the parameters needed to create such a permission.

Table 14.4 `AdminPermission` constructor parameters

Parameter	Description
`String filter`	LDAP filter to specify matching bundles
`String actions`	Comma-delimited list of the actions granted by the permission (`class`, `execute`, `extensionLifecycle`, `lifecycle`, `listener`, `metadata`, `resolve`, `resource`, `startlevel`, or `context`)

When you grant `AdminPermission` to a bundle, that bundle is allowed to perform the specified actions on the bundles matching the filter. The filter uses the same LDAP filter syntax used by the OSGi service registry, but only the following attributes are defined for matching:

- `signer`—Identity information about the bundle provider
- `location`—Bundle location
- `id`—Bundle identifier
- `name`—Bundle symbolic name

We'll give some filter examples shortly. First we'll briefly describe what the granted actions allow on the matching bundles:

- `class`—Load a class
- `execute`—Start, stop, and set the start level

Exhibit 1009
Microsoft Corporation v. Edge Networkring Systems, LLC

- `extensionLifecycle`—Manage extension bundles
- `lifecycle`—Install, update, and uninstall
- `listener`—Add/remove a synchronous bundle listener
- `metadata`—Retrieve bundle headers and location
- `resolve`—Resolve and refresh
- `resource`—Get resources
- `startlevel`—Set the start level, and initial the start level
- `context`—Get the bundle context

The special action `*` represents all actions. As you can see, `AdminPermission` gives you fine-grained control over which bundles can do what on which bundles. For example, assume a bundle wants to install another bundle using code like this:

```
context.installBundle("file:bundle.jar").start();
```

This triggers the framework to check whether all code on the call stack has the following:

- `AdminPermission.LIFECYCLE` permission for the installed bundle
- `AdminPermission.EXECUTE` permission for the installed bundle

This is relatively straightforward, although granting the permission can be a little confusing. The thing to remember about `AdminPermission` is that you use it to grant a bundle the right to perform specific operations on other bundles. The `filter` constructor parameter is how you specify the bundles that can be controlled. For a more complicated example, you can grant an `AdminPermission` like this:

```
new AdminPermission("(&(signer=CN=core,O=baz,C=de)(name=org.foo.*)
  (location=file://*)(id>=10))", AdminPermission.LIFECYCLE + "," +
  AdminPermission.EXECUTE);
```

The bundle granted this permission can perform the operations associated with `AdminPermission.LIFECYCLE` and `AdminPermission.EXECUTE` on bundles with a signer matching `CN=core,O=baz,C=de` (more on signing later), a symbolic name starting with `org.foo`, a location starting with `file://`, and a bundle identifier greater than 10. Granted, this is completely contrived, but it illustrates the possibilities.

You've now seen the module- and lifecycle-layer permissions, which means you have one framework layer to go—services.

14.3.4 ServicePermission

`ServicePermission` is a service-layer permission for controlling which services a bundle can provide or use. As with the other permissions, the actual permission granted is controlled by its constructor parameters, as shown in table 14.5.

You can use `*` or a trailing `.*` as a wildcard to target several service interfaces or classes with a single permission. The `get` action grants the ability to use the specified services, whereas the `register` action grants the ability to provide the specified services. For the `get` action, you can also use an LDAP filter for `name`, which matches

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

Table 14.5 `ServicePermission` constructor parameters

Parameter	Description
`String name`	Service interface or class name(s)
`String actions`	Comma-delimited list of the actions granted by the permission (`get` or `register`)

against the associated service properties in addition to the same set of bundle properties described for `BundlePermission` (`signer`, `location`, `id`, and `name`).

To get a better understanding of how this permission is used, consider the following snippet of code a bundle can use to find a service and to register a service:

```
context.getServiceReference("org.foo.Service");
context.registerService("org.bar.Service", new Service(), null);
```

Here you find an `org.foo.Service` service and register an `org.bar.Service` service. This triggers the framework to check whether all code on the call stack has the following:

- `ServicePermission.GET` for the `org.foo.Service` interface
- `ServicePermission.REGISTER` for the `org.bar.Service` interface

These permission checks are straightforward. For the associated bundle to perform these tasks, you can grant it these permissions:

```
new ServicePermission("org.foo.*", ServicePermission.GET);
new ServicePermission("org.bar.Service", ServicePermission.REGISTER);
```

In the first permission, you use a wildcard to allow it to access all services in the `org.foo` package. In the second permission, you specifically allow it to register `org.bar.Service` services.

That completes the OSGi-specific permissions you can grant to bundles. Before we move on to discussing permission management, let's briefly discuss file permissions, because they behave slightly differently in an OSGi environment.

14.3.5 Relative file permissions

Although `java.io.FilePermission` isn't defined by the OSGi specification, it's still impacted by how the framework interprets it. In a standard Java environment, a file permission created with a relative path is interpreted as being relative to the directory from which the Java process was started (the current working directory). This isn't the case in an OSGi environment. Instead, it's treated as relative to the root of the private data area of the associated bundle.

Typically, this doesn't have much of an effect, especially because the framework automatically grants bundles permission to read, write, and delete files in their own private area. The main thing this enables is the ability to grant a bundle additional permissions for files in its private data area, such as the execute permission.

Enough of describing permissions. Next, we'll discuss how you manage them with the Conditional Permission Admin Service.

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

```
    + context.getBundle().getLocation() + "\"]"
    + "(java.security.AllPermission \"*\" \"*\")"
    + "} \"Management Agent Policy\"");
  ConditionalPermissionAdmin cpa =
    getConditionalPermissionAdmin(context);
  ConditionalPermissionUpdate u = cpa.newConditionalPermissionUpdate();
  List infos = u.getConditionalPermissionInfos();
  infos.clear();
  for (String encodedInfo : encodedInfos) {
    infos.add(cpa.newConditionalPermissionInfo(encodedInfo));
  }
  if (!u.commit()) {
    throw new ConcurrentModificationException(
      "Permissions changed during update");
  }
}
```

This method combines most of your knowledge about the `ConditionalPermissionAdmin` service. You get the policy file, which defaults to a file called `security.policy` but can be configured. Next, you read in the encodings of the `ConditionalPermissionInfo` objects contained in the policy file and add an encoding for an `AllPermission` at the beginning of the list for the policy-file reader bundle. The previous step is necessary to make sure the policy-reader bundle has sufficient permission to make future changes to the security policy.

After this, you get the `ConditionalPermissionAdmin` service, create a new `ConditionalPermissionUpdate`, and use it to get the current list of `ConditionalPermissionInfo` objects. You clear the existing policy to make sure you're starting with a clean slate, and then loop through the encoded permissions to decode them and add them to your list of objects. The only thing left to do is commit the update. Because the update may fail if the permissions were changed concurrently, you throw an exception in this case.

To see the full details of this bundle, go to the chapter14/combined-example/org.foo.policy/ directory of the book's companion code. This bundle is generic and can be used in any security-enabled framework to put a policy file into effect. You'll see it in action a little later when we show a complete example with digitally signed bundles and a custom condition. We'll introduce bundle signing next.

14.5 Digitally signed bundles

Defining a security policy by assigning permissions to bundles is a workable approach, but being able to step up a level can simplify things. For example, you may trust a particular provider, so it's nice to be able to assign permissions based on the provider rather than individual bundles. Doing so simplifies defining a security policy, because it raises the level of abstraction. Digitally signed bundles can help you achieve this; specifically, they help you do two things:

- Authenticate the provider of a bundle
- Ensure that bundle content hasn't been modified

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

The former provides the ultimate goal, but without the latter, the former would be meaningless. You'll learn about both as we discuss digital signing and certificates. We'll show you how to create certificates and use them to digitally sign your bundles. Then we'll introduce `BundleSignerCondition`, which gives you the ability to grant permissions based on the identity established via certificates. First, let's get some terminology out of the way.

14.5.1 Learning the terminology

The domain of digital cryptography is complex and sophisticated. Providing a complete and detailed description is beyond the scope of the book, so we'll focus on describing just enough to have it make sense. With that in mind, table 14.6 introduces some relevant terms we'll use throughout the remainder of the chapter.

Table 14.6 Digital cryptography terminology

Term	Definition
Digital signing	A mathematical approach for verifying the authenticity of digital data. Specifically, used to verify the identity of the provider and that the data hasn't been modified.
Signature	A unique value calculated when data is digitally signed.
Public key cryptography	A form of digital signing using two mathematically related keys: a public key and a private key. The private key is a guarded secret used to sign data. The public key is shared with others in the form of a certificate, which they can use to verify that a signature was generated with the private key. This allows you to infer the identity of the provider and determine whether someone has tampered with the data.
Certificate	A form of metadata about a public key, binding it to the identity of the private key holder. This binding is achieved by having a well-known (trusted) third party sign the public key/identity pair.
Distinguished name	The identification portion of a certificate; specifically, as defined by the X.509 ITU-IT standard. Identifies the holder of the private key.
Certificate chain	A certificate has a reference to the certificate of its third-party signer, which includes a reference to the certificate of its signer, and so on, until the root. This is a certificate chain. The root of the certificate chain is a self-signed certificate.

A general understanding of these terms should be sufficient. You don't need a complete understanding of digital cryptography to use the technology effectively. We'll start looking at the basics.

14.5.2 Creating certificates and signing bundles

You'll be using digital signing based on public key cryptography, which involves a public key and a private key. The public key is shared with the world in the form of a certificate. The private key is kept secret and used to sign data by performing a computation over it. The resulting value can be verified by performing another calculation over the data using the public key. This verifies that the signer has access to the private key and that the data hasn't been modified.

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

In OSGi, the signer of a bundle is associated with it. With this association, you can grant permissions to a bundle based on its signers. For example, you can assign permissions to all bundles from a particular company, if the company signs its bundles. You can also grant permissions to perform operations on bundles signed by specific principles. These approaches provide a simple yet powerful way to control who can do what inside your application.

Effectively, bundle signing creates a powerful delegation model. An administrator can grant a restricted set of permissions to a signer, after which the signer can create bundles that use those permissions or some subset, without any intervention or communication with the administrator for each particular bundle. To understand how this all fits together, consider the following scenario.

Assume you have a system that features a set of core bundles (which we'll call the *core domain*) and an arbitrary number of third-party plugin bundles (which we'll call the *third-party domain*). This means you expect fully trusted bundles and not completely trusted bundles to exist in your system, but you want to provide a level of isolation between them. Your goal is to create a simple security policy that lets you manage core and third-party bundle domains without knowing the precise bundles in each set. We'll delve into the details of doing this next.

CERTIFICATES AND KEYSTORES

To implement the desired security policy for this scenario, you need to create two root certificates for the core and third-party domains. These root certificates will be used by the framework to establish a chain of trust for the two domains. With these two certificates, you can then sign certificates of core and third-party providers with the appropriate certificate. When they use their individual certificates to sign bundles they've created, the framework can use the root certificates to establish a chain of trust and determine to which domain a bundle belongs.

The details of all this are based on Java 2 JAR file signing, which means the same tools you use to sign JAR files can be used for OSGi. To create the needed certificates and their associated public and private keys, you'll use the `keytool` command provided by the JDK. It can create and manage certificates and keys inside a *keystore*, which is an encrypted file defined by Java for this purpose. For this scenario, you use `keytool` to create two certificates and their associated public/private keys for the core and third-party domains like this:

```
keytool -genkey -keystore keys.ks -alias core -storepass foobar  \
  -keypass barbaz -dname "CN=core,O=baz,C=de"
keytool -genkey -keystore keys.ks -alias third-party \
  -storepass foobar -keypass barbaz -dname "CN=third-party,O=baz,C=de"
```

This creates a keystore called `keys.ks` containing two new key pairs and a certificate for each pair with aliases of `core` and `third-party`. The keystore is protected by the password `foobar`, and the keys themselves have the password `barbaz`. The `-dname` switch allows you to specify the distinguished name you use to identify yourself, which in this case is the `baz` organization in Germany (`de`).

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

DISTINGUISHED NAME A standard X.509 structured name, officially identifying a node in a hierarchical namespace. For our purposes, it's sufficient to recognize a distinguished name (DN) as a set of comma-delimited attributes, such as in the example: `CN=core,O=baz,C=de`. These attributes specify the common name, organization, and country, respectively. The hierarchical aspect of this namespace is that it goes from the least significant (but most specific) attribute to the most significant. The root of the tree for these attributes is the country, which is then divided into organization, and further divided into common names within an organization. Order is significant. Two DNs with the same attributes but different order are different DNs.

The next thing to do is sign your key pair certificates with themselves. It may sound a little strange, but this is how you make them root certificates. It's a common thing to do, as you can see by the fact that the `keytool` command has support for it:

```
keytool -selfcert -keystore keys.ks -alias core -storepass foobar \
  -keypass barbaz -dname "CN=core,O=baz,C=de"
keytool -selfcert -keystore keys.ks -alias third-party \
  -storepass foobar -keypass barbaz -dname "CN=third-party,O=baz,C=de"
```

The only difference from the previous command is that you use `-selfcert` instead of `-genkey`.

Now you have key pairs that you can use to sign other certificates or bundles to make them part of your trusted certificate chain. To allow other people to verify your signatures, you need to extract the certificates from the `keys.ks` keystore and import them into a new keystore called `certificates.ks`. Why? Because the `keys.ks` keystore contains your private keys; you need another keystore that contains only your public keys to share with the outside world. Currently, your certificates are saved as key entries (a public/private key pair and its certificate) in the keystore. You need to export them and re-import them as certificate-only entries, which you do like this:

```
keytool -export -v -keystore keys.ks -alias core \
  -file core.cert -storepass foobar -keypass barbaz
keytool -export -v -keystore keys.ks -alias third-party \
  -file third-party.cert -storepass foobar -keypass barbaz
keytool -import -v -keystore certificates.ks -alias core-cert \
  -file core.cert -storepass foobar -keypass barbaz
keytool -import -v -keystore certificates.ks -alias third-party-cert \
  -file third-party.cert -storepass foobar -keypass barbaz
```

You can verify the contents of your keystores like this:

```
> keytool -list -keystore certificates.ks -storepass foobar

third-party-cert, 08.01.2010, trustedCertEntry,
fingerprint (MD5): 15:9B:EE:BE:E7:52:64:D4:9C:C1:CB:5D:69:66:BB:29
core-cert, 08.01.2010, trustedCertEntry,
fingerprint (MD5): CE:37:F8:71:C9:37:12:D0:F1:C8:2B:F9:85:BE:EA:61

> keytool -list -keystore keys.ks -storepass foobar

core, 08.01.2010, PrivateKeyEntry,
fingerprint (MD5): CE:37:F8:71:C9:37:12:D0:F1:C8:2B:F9:85:BE:EA:61
third-party, 08.01.2010, PrivateKeyEntry,
fingerprint (MD5): 15:9B:EE:BE:E7:52:64:D4:9C:C1:CB:5D:69:66:BB:29
```

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

You have everything in place now, which means we can look into signing bundles to make them members of one of your domains.

SIGNING BUNDLES

A bundle JAR file can be signed by multiple signers; the signing follows normal Java JAR signing rules. The only additional constraint for a bundle is that all entries inside the bundle must be included in the signature, but entries below the META-INF/ directory aren't included. Normal Java JAR file signing allows for partially signed JAR files, but OSGi doesn't. It's lucky that signing all entries except those below META-INF/ is the default in JAR signing, so you can use the `jarsigner` tool included in the JDK. The following will sign a bundle with your core private key:

```
jarsigner -keystore file:keys.ks \
  -storepass foobar -keypass barbaz core-bundle.jar core
```

Signing another bundle with your third-party private key looks very similar. You specify the appropriate alias:

```
jarsigner -keystore file:keys.ks \
  -storepass foobar -keypass barbaz third-party-bundle.jar third-party
```

For verification, you need the keystore containing the certificates. You can use the `jarsigner` tool for verification as well:

```
jarsigner -verify -keystore file:certificates.ks core-bundle.jar
jarsigner -verify -keystore file:certificates.ks third-party-bundle.jar
```

This command should output `jar verified` if you've correctly signed the bundles. Assuming you have, you now have one bundle in the core domain and one in the third-party domain. This makes it easy for you to grant permissions to either, based on the signer of a bundle, as you'll see next.

14.5.3 BundleSignerCondition

To assign permissions to bundles based on who signed them, you need a condition. The OSGi specification defines the `BundleSignerCondition` for certificate matching, which is specifically based on DN matching. DN matching can seem somewhat complicated, but it needn't be. We'll discuss the details of it shortly; first, let's look at how you construct a `BundleSignerCondition`.

The `BundleSignerCondition` is initialized with a DN matching expression as its first argument and an optional second argument of `!`. If you specify the exclamation mark, it negates the result of the DN matching expression. Consider the following snippet of an encoded `ConditionalPermissionInfo` object containing a `BundleSignerCondition`:

```
ACCEPT {
[org.osgi.service.condpermadmin.BundleSignerCondition "CN=core,O=baz,C=de"]
...
}
```

This matches a bundle on the call stack if it's signed by the core certificate of the example, which means any permissions associated with this entry will be granted to

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

the bundle. On the other hand, the following won't match a bundle if it was signed by the core certificate:

```
ACCEPT {
[org.osgi.service.condpermadmin.BundleSignerCondition
  "CN=core,O=baz,C=de" "!"]
...
}
```

The DN matching expression in these two examples illustrates how simple DN matching can be. It can also be sophisticated, because it supports various flavors of wildcard matching. We'll describe that next.

DISTINGUISHED NAME MATCHING DETAILS

You saw that a DN is composed of multiple attributes, like country, organization, and common name. When performing DN matching, you're matching against these attributes using a comma-delimited list, such as what you saw earlier with `CN=core, O=baz,C=de` to match the core certificate. Additionally, because certificates can be signed by other certificates, you can match against the other certificates in the chain: you delimit different certificates with a semicolon.

To match certificates in a chain, use DN matching expressions against the DN associated with each certificate you're trying to match. For example, consider the following DN matching expression:

```
CN=extensions,O=bar,C=fr;CN=core,O=baz,C=de
```

This matches a bundle that was signed by the bar organization from France using its extensions certificate, which was signed by your core certificate. You need to understand two important points about chain matching:

- Matching occurs naturally against the most specific certificate. Certificates further up the chain that aren't mentioned are ignored.
- Order is important, because reversing it indicates the opposite signing relationship.

When you match certificate chains, you're specifying an interest from the most specific certificate of the chain onward.

Both attribute matching and certificate chain matching support wildcards, but the rules for comparison are more complicated than string-based wildcard matching. The different cases are described in table 14.7.

Table 14.7 Certificate DN wildcard matching

Case	Description
Specific attribute wildcards	If a wildcard is used as part of the right-hand argument of an attribute, such as `CN=*,O=baz,C=de` this matches either of the two certificates (core and third-party). You can also use a wildcard for more than one attribute: `CN=*,O=baz,C=*` This matches any certificate from the `baz` organization from any country.

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

Table 14.7 Certificate DN wildcard matching *(continued)*

Case	Description
Arbitrary attribute wildcards	If a wildcard is used standalone, such as `*,O=baz,C=de` this matches any attributes coming before `O` and `C`, regardless of their name or value. For the example, it's another way to match both of your certificates. This kind of wildcard can also be combined with the previous: `*,O=baz,C=*` This also matches all certificates from the `baz` organization from any country.
Certificate chain wildcards	The attribute wildcard can be used in a certificate chain and behaves as described earlier, but when used standalone it matches at most one certificate. For example: `*;CN=core,O=baz,C=de` This matches either a bundle signed by another certificate that was signed by your core certificate or a bundle signed directly by your core certificate. The hyphen wildcard matches zero or more certificates: `-;CN=core,O=baz,C=de` This matches any bundle signed by your core certificate anywhere in the certificate chain.

The rules for certificate matching are also relevant to `AdminPermission`, discussed in section 14.3.3. If you recall, `AdminPermission` accepts an LDAP filter over a limited number of attributes to describe target bundles. The value for the `signer` attribute of the LDAP filter is a DN matching expression.

ESTABLISHING TRUST

In addition to DN matching, a `BundleSignerCondition` will only match if all the certificates in the chain are trusted or are signed by a trusted certificate. Certificates are *trusted* when they're known by the OSGi framework. How do they become known? Prior to OSGi R4.2 specification, this was implementation-specific; but now the standard way is to specify keystores containing trusted certificates using the `org.osgi.framework.trust.repositories` framework configuration property. For example:

```
org.osgi.framework.trust.repositories=\
  /var/trust/keystore.jks:~/.cert/certs.jks
```

The value is a list of file paths, where the paths are separated by the system-specific `File.pathSeparator`. Each file path must point to a JKS keystore, which can't have a password. The framework uses the keystores as trust repositories to authenticate certificates of trusted signers. The stores must be used only as read-only trust repositories to access public keys.

You should now understand how to use certificates to sign your bundles and grant permissions based on the bundle signer. With that out of the way, let's look at how you can use local permissions to know which permissions a bundle needs.

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

14.6 Local permissions

Bundle signing provides a powerful yet fairly simple mechanism for creating desired security policies. But it doesn't help address one nagging issue: how do you know which permissions to grant a bundle? Even if you've verified a bundle's signature and know that the bundle comes from a trusted provider, you still need to answer this question. Even if you fully trust a provider, it's better to limit a bundle's permissions to a precise set of required permissions to further prevent intended or unintended security breaches.

The standard Java security architecture doesn't help you here; instead, you must rely on prior knowledge about the code's requirements or trial and error. OSGi specifically addresses this issue with a concept called *local permissions*. Local permissions are defined by a resource contained in the bundle, which describes the maximum set of permissions required by the bundle. This set of permission is enforced by the OSGi framework. A bundle can be granted fewer permissions than its local permissions, but it never gets more permissions.

At first blush, it may seem a little odd to have a bundle define its own permissions, but the purpose is more for the deployer to audit and analyze a bundle. Bundles aren't guaranteed to receive the permissions they request and therefore should be programmed to degrade gracefully when they receive less. As a deployer, though, local permissions simplify your life because you can easily determine what permissions are required and which you're willing to give. For example, if the local permissions request the use of network sockets, it's clear that the bundle has the potential to access the wider internet. If you decide this is acceptable, you can trust this audit because it's enforced by the framework at execution time.

What do local permissions look like in practice? The bundle-permission resource is a file in the bundle's OSGI-INF/ directory called permissions.perm. It contains a listing of all of the bundle's required permissions. As a simple example, let's assume you provide a bundle that only wants to export a single package, org.foo. This bundle's OSGi-INF/permissions.perm file is as follows:

```
# Tuesday, Dec 28 2009
# Foo Bundle
( org.osgi.framework.PackagePermission "org.foo" "IMPORT,EXPORT" )
```

Lines that start with a # are comments. All other non-empty lines describe required permissions as encoded `PermissionInfo` objects. This is simple but effective when it comes to auditing the security impact of a given bundle.

You've now learned about some powerful tools for defining a security policy; but in the infamous words of many infomercials, "Wait! There's still more!" In the next section, we'll cover the most advanced tool available: the ability to create custom conditions for your security policy. We'll explore *why* you may want to do this and show you *how* to do it by implementing two custom conditions.

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

Here you check whether you've already asked the user by looking at the `m_alreadyAsked` flag, which is only necessary to avoid a race condition if multiple threads are trying to set the initial value; after that, the framework will cache the result because the condition is immutable. If the user hasn't already been asked, you create a new `AskTheUser` object with your question ❷ and call its `ask()` method. When you get the result, you set the `alreadyAsked` flag to `true` ❸ to make sure that the user is asked only one time for the given bundle. Finally, you return the result or invert the result if `"!"` was specified in the `ConditionInfo`.

A slightly complicated part of this example is that you need to perform the call to the `ask()` method using `AccessController.doPrivileged()` ❶. This is because the use of Swing will result in a lot of additional permission checks, so you must limit the protection domains involved to the protection domain of the condition itself. Because the condition must be on the class path, it'll have the protection domain of the framework, which needs to have `AllPermission`. If, for whatever reason, you get an exception, you return `false`.

This completes the `AskTheUserCondition` implementation. If you package it as an extension bundle or add it to the framework class path, you can use it to let the user make security decisions by including it in your policy file like this:

```
ACCEPT {
  [ org.foo.AskTheUserCondition "Do you want to allow $symbolic-name to
  provide a shape?" ]
  ( org.osgi.framework.ServicePermission "org.foo.shape.SimpleShape"
  "register")
}
```

We've covered a lot of ground in this chapter, so you should be commended for making it this far. To wrap up the discussion on security, we'll look at an example that pulls everything together.

14.8 Bringing it all back home

What's left is to show you how to start a framework with security enabled. You need to make sure a security manager is installed in the system and tell the framework where it can find the trusted root certificates. You can either set a custom security manager or have the framework install its own security manager. Typically, you need to set the following two framework-configuration properties:

- `org.osgi.framework.security`—Tells the framework to set the security manager
- `org.osgi.framework.trust.repositories`—As mentioned earlier, specifies the repositories of trusted certificates

The `org.osgi.framework.security` property value can be either an empty string or `osgi`. In either case, the framework sets the JVM security manager to an implementation-specific one when started. If the property isn't specified, the framework doesn't set the security manager; security will still work if a security manager is already set, but not all features of OSGi security may work. In particular, if the existing security manager uses `AccessController`, postponed conditions won't work.

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

Even though some aspects of enabling security are standardized, not all aspects are. As a result, enabling security is handled a little differently by different framework implementations. We'll use the Apache Felix framework to show a concrete example. The Felix framework is special because it provides its Conditional Permission Admin Service implementation as an extension bundle. This means that in addition to setting the previous properties, you also need to deploy the security provider bundle. Luckily, this is easy to do with the bundle launcher; add it to the directory containing the bundles you want to launch.

This gets you a framework with security enabled and an initial security policy. If you want to allow for bundles signed by trusted certificates, you can use the `org.osgi.framework.trust.repositories` property to point to the keystore containing the certificates you trust, but typically a keystore requires the use of a password. Because keystores used with this configuration property can't have passwords, you need to resort to an implementation-specific means to give you what you want. For the Felix framework, you do the following:

```
java -Dorg.osgi.framework.security=osgi \
    -Dfelix.keystore=file:certificates.ks \
    -Dfelix.keystore.pass=foobar \
    -Dfelix.keystore.type=jks \
    -jar launcher.jar bundles
```

In this case, the initial security policy file contains the following:

```
grant { permission java.security.AllPermission; };
```

This sets up your framework with a keystore and the password necessary to access it.

To illustrate what you can do with all of this, let's add to the paint program a security policy that uses the security features you've learned about. The security policy will allow core providers to provide shapes automatically; all others will require explicit approval from the user. Other than provide shape services, bundles are allowed to do anything. Start by creating a policy file with an entry to grant `AllPermission` to bundles signed by the core certificate:

```
ALLOW {
  [ org.osgi.service.condpermadmin.BundleSignerCondition
    "CN=core,O=baz,C=de" ]
   ( java.security.AllPermission "*" "*" )
} "Signed by core"
```

Next, create an entry to grant all other bundles permission to register a shape service based on the condition that the user approves it. For this, use your custom condition like this:

```
ALLOW {
  [ org.foo.condition.ask.AskUserCondition
    "Do you want to allow ${symbolic-name} to provide a shape?" ]
  ( org.osgi.framework.ServicePermission
    "org.foo.shape.SimpleShape" "register" )
} "Ask the user"
```

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

If a bundle that isn't signed by your core certificate tries to register a shape service, the user is asked to grant or deny that request. Pretty simple, right? This entry only deals with asking the user to approve shape services from non-core bundles; you still need to create an entry to grant these bundles the ability to do everything else. In that case, you need to grant them `AllPermission` except for registering shape services. You can use a `DENY` access decision, like this:

```
DENY {
  ( org.osgi.framework.ServicePermission
    "org.foo.shape.SimpleShape" "register" )
} "Deny register"
ALLOW {
  ( java.security.AllPermission "*" "*" )
} "All other"
```

All these entries combined form your defined security policy. You may wonder whether the policy is correct. If you follow the rule ordering in the policy file, it looks like it will always prompt the user if a bundle isn't signed by the core. But this isn't the case, because `AskUserCondition` is a postponed condition. That means it's evaluated only if no other rule with an immediate condition implies the permission. Your security policy is evaluated like this:

1. If a bundle is signed by the core certificate, it immediately matches the first rule, which allows it to do anything.
2. If a non-core signed bundle performs any secure operation other than registering a shape service, the first rule doesn't apply, the second rule is postponed, the third rule doesn't apply, and ultimately the fourth rule is matched that allows the bundle to do anything.
3. If a non-core signed bundle tries to register a shape service, the first rule doesn't apply, the second rule is postponed, the third rule applies because the permission is implied, and this rule fails due to the `DENY` access decision. This causes the framework to evaluate the postponed second rule because it logically came before the failed entry, which prompts the user and grants the permission based on the user's reply.

As you can see, you only ask the user at the end if all other rules don't provide the needed permission. The `DENY` access decision of the third rule provides a way to short-circuit the rule evaluation. To see this security policy in action, go into the chapter14/combined-example/ directory of the book's companion code; type `ant` to build it and the following to run it:

```
java -Dorg.osgi.framework.security=osgi -Djava.security.policy=all.policy \
  -Dfelix.keystore=file:certificates.ks -Dfelix.keystore.pass=foobar  \
  -Dfelix.keystore.type=jks -jar launcher.jar bundles
```

This starts your shell in a security-enabled framework running your security policy. You first need to install the bundles of the paint program:

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

```
-> install file:paint-example/shape-4.0.jar
Bundle: 8
-> install file:paint-example/paint-4.0.jar
Bundle: 9
-> start 9
```

These bundles were signed by the core certificate and have `AllPermission`, so you should see an empty paint program (one with no shapes) after starting the paint bundle. Next, install and start the unsigned circle bundle:

```
-> install file:paint-example/circle-4.0.jar
Bundle: 10
-> start 10
```

Doing this causes the user to be prompted to grant the permission, as shown in figure 14.3. If you grant the bundle permission, you get a circle shape in the paint program, as shown in figure 14.4.

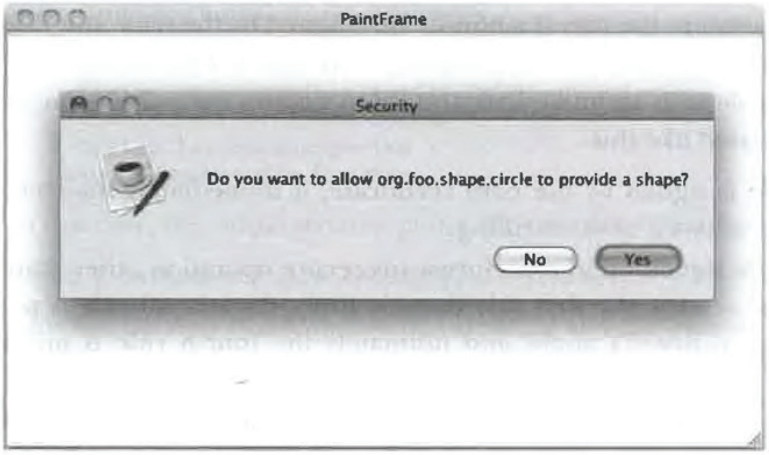


Figure 14.3 Secured paint program prompting the user to grant the unsigned circle bundle permission to provide a shape service



Figure 14.4 Secured paint program after the user has granted permission to the unsigned circle bundle

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

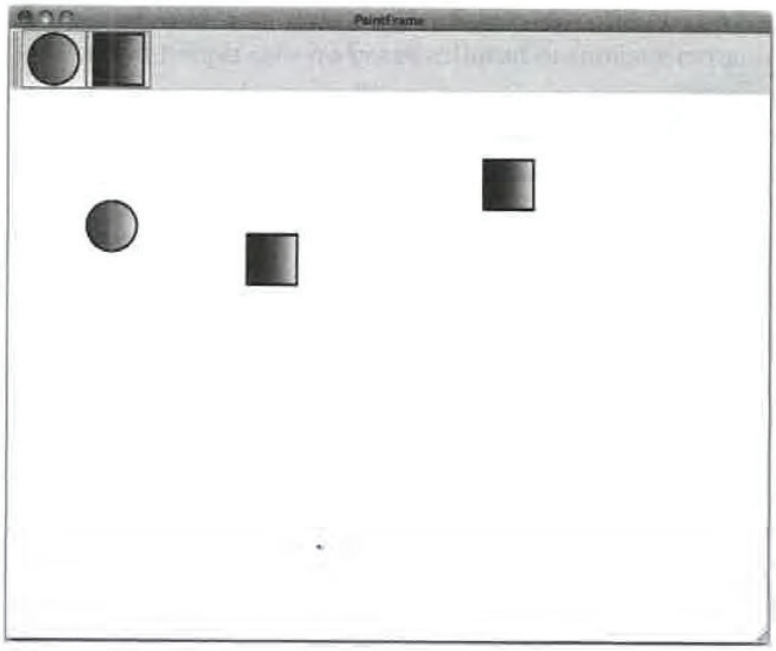


Figure 14.5 Secured paint program with an unsigned circle bundle and a signed square bundle

Finally, to show that core-signed bundles can provide shapes without prompting the user, install and start the square bundle:

```
-> install file:paint-example/square-4.0.jar
Bundle: 11
-> start 11
```

You should now have a paint program with circle and square shapes, as shown in figure 14.5.

Congratulations! If you've made it this far, you know just about everything there is to know about securing your OSGi-based applications. Just do it!

14.9 Summary

In this chapter, we introduced you to the Java security model and showed how OSGi uses it to provide the infrastructure to deploy and manage applications that must run in secure environments. You've learned the following:

- It's important to have security in mind when you're writing bundles, because otherwise they probably won't be good citizens in a security-enabled environment.
- Java security provides the foundation of the OSGi security model.
- OSGi provides the necessary permissions to express security policies for bundles with respect to the three key layers of the OSGi framework: module, lifecycle, and services.
- The Conditional Permission Admin Service introduces a new way of managing security by means of conditions that must be satisfied in order for certain permissions to be applicable.

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

- You can make your life a lot simpler by signing your bundles with certificates and assigning permissions to bundles based on who signed them.
- Specifying local permission inside of bundles provides a convenient and simple way to audit and enforce the permissions needed by a bundle.
- It's easy to implement and provide custom conditions for use in your security policies.
- Postponed conditions let you defer expensive condition evaluation until the end, which allows you to fine-tune your security policies for efficiency.

With this knowledge under your belt, you can secure your framework according to your specific security policies and develop bundles that can work in security-enabled frameworks. In the next chapter, we'll look into how you can use and provide web services in OSGi as well as how to build web applications on top of OSGi.

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC

JAVA

LC CIP
0 028 813 668 3

OSGi IN ACTION

Hall • Pauls • McCulloch • Savage

Free eBook
SEE INSERT

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
54999
9 781933 988917

MANNING $49.99 / Can $61.99 [INCLUDING eBOOK]

Exhibit 1009
Microsoft Corporation v. Edge Netwokring Systems, LLC