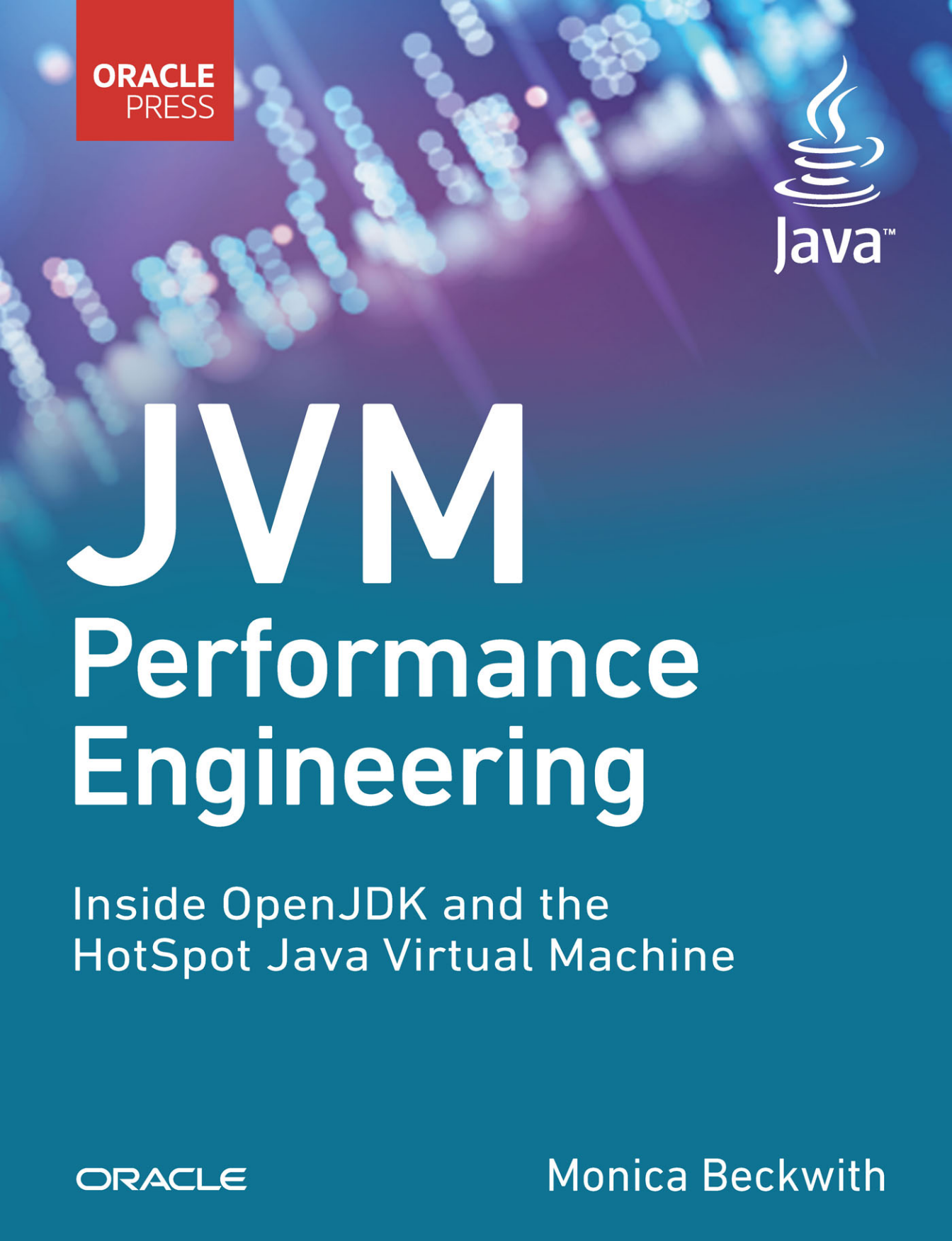
ORACLE
PRESS
Java™
JVM
Performance
Engineering
Inside OpenJDK and the
HotSpot Java Virtual Machine
ORACLE
Monica Beckwith

JVM Performance Engineering

Now, let's visualize the workings of the four layers that we discussed earlier. To do so, we'll create a simple problem statement that builds a quote for basic and high-quality bricks for both levels of the house. First, we add the following code to our `main()` method:

```
int[] level = {1,2};
IntStream levels = Arrays.stream(level);
```

Next, we stream `doWork()` as follows:

```
levels.forEach(levelcount -> loadLayers
         .doWork(…
```

We now have four layers similar to those mentioned earlier (*house ver1.b*, *house ver1.hq*, *house ver2.b*, and *house ver2.hq*). Here's the updated output:

```
Created a new layer for com.codekaram.provider
My basic 1 level house will need 18000 bricks and those will cost me $6120
Created a new layer for com.codekaram.provider
My high-quality 1 level house will need 18000 bricks and those will cost me $9000
Created a new layer for com.codekaram.provider
My basic 2 level house will need 36000 bricks and those will cost me $12240
Created a new layer for com.codekaram.provider
My high-quality 2 level house will need 36000 bricks and those will be over my budget of $15000
```

NOTE The return string of the `getName()` methods for our providers has been changed to return just the `"basic"` and `"high-quality"` strings instead of an entire sentence.

The variation in the last line of the updated output serves as a demonstration of how additional conditions can be applied to service providers. Here, a budget constraint check has been integrated into the high-quality provider's implementation for a two-level house. You can, of course, customize the output and conditions as per your requirements.

Here's the updated `doWork()` method to handle both the level and the provider, along with the relevant code in the main method:

```
private static void doWork(int level, Stream<ModuleLayer> myLayers){
    myLayers.flatMap(moduleLayer -> ServiceLoader
        .load(moduleLayer, BricksProvider.class)
        .stream().map(ServiceLoader.Provider::get))
    .forEach(eachSLProvider -> System.out.println("My " + eachSLProvider.getName()
    + " " + level + " level house will need " + eachSLProvider.getBricksQuote(level)));
}
```

```
public static void main(String[] args) {
    int[] levels = {1, 2};
    IntStream levelStream = Arrays.stream(levels);

    levelStream.forEach(levelcount -> doWork(levelcount, Stream.of(args)
        .map(getCustomDir -> getProviderLayer(getCustomDir))));
}
```

Now, we can calculate the number of bricks and their cost for different levels of the house using the `basic` and `high-quality` implementations, with a separate module layer being devoted to each implementation. This demonstrates the power and flexibility that module layers provide, by enabling you to dynamically load and unload different implementations of a service without affecting other parts of your application.

Remember to adjust the service providers' code based on your specific use case and requirements. The example provided here is just a starting point for you to build on and adapt as needed.

In summary, this example illustrates the utility of Java module layers in creating applications that are both adaptable and scalable. By using the concepts of module layers and the Java `ServiceLoader`, you can create extensible applications, allowing you to adapt those applications to different requirements and conditions without affecting the rest of your codebase.

Open Services Gateway Initiative

The Open Services Gateway Initiative (OSGi) has been an alternative module system available to Java developers since 2000, long before the introduction of Jigsaw and Java module layers. As there was no built-in standard module system in Java at the time of OSGi's emergence, it addressed many modularity problems differently compared to Project Jigsaw. In this section, with insights from Nikita, whose expertise in Java modularity encompasses OSGi, we will compare Java module layers and OSGi, highlighting their similarities and differences.

OSGi Overview

OSGi is a mature and widely used framework that provides modularity and extensibility for Java applications. It offers a dynamic component model, which allows developers to create, update, and remove modules (called bundles) at runtime without restarting the application.

Similarities

- **Modularity:** Both Java module layers and OSGi promote modularity by enforcing a clear separation between components, making it easier to maintain, extend, and reuse code.
- **Dynamic loading:** Both technologies support dynamic loading and unloading of modules or bundles, allowing developers to update, extend, or remove components at runtime without affecting the rest of the application.

- **Service abstraction:** Both Java module layers (with the `ServiceLoader`) and OSGi provide service abstractions that enable loose coupling between components. This allows for greater flexibility when switching between different implementations of a service.

Differences

- **Maturity:** OSGi is a more mature and battle-tested technology, with a rich ecosystem and tooling support. Java module layers, which were introduced in JDK 9, are comparatively newer and may not have the same level of tooling and library support as OSGi.
- **Integration with Java platform:** Java module layers are a part of the Java platform, providing a native solution for modularity and extensibility. OSGi, by contrast, is a separate framework that builds on top of the Java platform.
- **Complexity:** OSGi can be more complex than Java module layers, with a steeper learning curve and more advanced features. Java module layers, while still providing powerful functionality, may be more straightforward and easier to use for developers who are new to modularity concepts.
- **Runtime environment:** OSGi applications run inside an OSGi container, which manages the life cycle of the bundles and enforces modularity rules. Java module layers run directly on the Java platform, with the module system handling the loading and unloading of modules.
- **Versioning:** OSGi provides built-in support for multiple versions of a module or bundle, allowing developers to deploy and run different versions of the same component concurrently. This is achieved by qualifying modules with versions and applying "uses constraints" to ensure safe class namespaces exist for each module. However, dealing with versions in OSGi can introduce unnecessary complexity for module resolution and for end users. In contrast, Java module layers do not natively support multiple versions of a module, but you can achieve similar functionality by creating separate module layers for each version.
- **Strong encapsulation:** Java module layers, as first-class citizens in the JDK, provide strong encapsulation by issuing error messages when unauthorized access to non-exported functionality occurs, even via reflection. In OSGi, non-exported functionality can be "hidden" using class loaders, but the module internals are still available for reflection access unless a special security manager is set. OSGi was limited by pre-JPMS features of Java SE and could not provide the same level of strong encapsulation as Java module layers.

When it comes to achieving modularity and extensibility in Java applications, developers typically have two main options: Java module layers and OSGi. Remember, the choice between Java module layers and OSGi is not always binary and can depend on many factors. These include the specific requirements of your project, the existing technology stack, and your team's familiarity with the technologies. Also, it's worth noting that Java module layers and OSGi are not the only options for achieving modularity in Java applications. Depending on your specific needs and context, other solutions might be more appropriate. It is crucial to thoroughly evaluate the pros and cons of all available options before making a decision for your project. Your choice should be based on the specific demands and restrictions of your project to ensure optimal outcomes.

On the one hand, if you need advanced features like multiple version support and a dynamic component model, OSGi may be the better option for you. This technology is ideal for complex applications that require both flexibility and scalability. However, it can be more difficult to learn and implement than Java module layers, so it may not be the best choice for developers who are new to modularity.

On the other hand, Java module layers offer a more straightforward solution for achieving modularity and extensibility in your Java applications. This technology is built into the Java platform itself, which means that developers who are already familiar with Java should find it relatively easy to use. Additionally, Java module layers offer strong encapsulation features that can help prevent dependencies from bleeding across different modules.

Introduction to Jdeps, Jlink, Jdeprscan, and Jmod

This section covers four tools that aid in the development and deployment of modular applications: *jdeps*, *jlink*, *jdeprscan*, and *jmod*. Each of these tools serves a unique purpose in the process of building, analyzing, and deploying Java applications.

Jdeps

Jdeps is a tool that facilitates analysis of the dependencies of Java classes or packages. It's particularly useful when you're trying to create a module file for JAR files. With *jdeps*, you can create various filters using regular expressions (*regex*); a regular expression is a sequence of characters that forms a search pattern. Here's how you can use *jdeps* on the `loadLayers` class:

```
$ jdeps mods/com.codekaram.brickhouse/com/codekaram/brickhouse/loadLayers.class
loadLayers.class -> java.base
loadLayers.class -> not found
   com.codekaram.brickhouse -> com.codekaram.brickhouse.spi   not found
   com.codekaram.brickhouse -> java.io                        java.base
   com.codekaram.brickhouse -> java.lang                      java.base
   com.codekaram.brickhouse -> java.lang.invoke               java.base
   com.codekaram.brickhouse -> java.lang.module               java.base
   com.codekaram.brickhouse -> java.nio.file                  java.base
   com.codekaram.brickhouse -> java.util                      java.base
   com.codekaram.brickhouse -> java.util.function             java.base
   com.codekaram.brickhouse -> java.util.stream               java.base
```

The preceding command has the same effect as passing the option `-verbose:package` to *jdeps*. The `-verbose` option by itself will list all the dependencies:

```
$ jdeps -v mods/com.codekaram.brickhouse/com/codekaram/brickhouse/loadLayers.class
loadLayers.class -> java.base
loadLayers.class -> not found
   com.codekaram.brickhouse.loadLayers -> com.codekaram.brickhouse.spi.BricksProvider  not found
   com.codekaram.brickhouse.loadLayers -> java.io.PrintStream                  java.base
   com.codekaram.brickhouse.loadLayers -> java.lang.Class                      java.base
```

```
com.codekaram.brickhouse.loadLayers -> java.lang.ClassLoader                java.base
com.codekaram.brickhouse.loadLayers -> java.lang.ModuleLayer                java.base
com.codekaram.brickhouse.loadLayers -> java.lang.NoSuchMethodException      java.base
com.codekaram.brickhouse.loadLayers -> java.lang.Object                     java.base
com.codekaram.brickhouse.loadLayers -> java.lang.String                     java.base
com.codekaram.brickhouse.loadLayers -> java.lang.System                     java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.CallSite            java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.LambdaMetafactory   java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.MethodHandle        java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.MethodHandles       java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.MethodHandles$Lookup java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.MethodType          java.base
com.codekaram.brickhouse.loadLayers -> java.lang.invoke.StringConcatFactory java.base
com.codekaram.brickhouse.loadLayers -> java.lang.module.Configuration       java.base
com.codekaram.brickhouse.loadLayers -> java.lang.module.ModuleFinder        java.base
com.codekaram.brickhouse.loadLayers -> java.nio.file.Path                   java.base
com.codekaram.brickhouse.loadLayers -> java.nio.file.Paths                  java.base
com.codekaram.brickhouse.loadLayers -> java.util.Arrays                     java.base
com.codekaram.brickhouse.loadLayers -> java.util.Collection                 java.base
com.codekaram.brickhouse.loadLayers -> java.util.ServiceLoader              java.base
com.codekaram.brickhouse.loadLayers -> java.util.Set                        java.base
com.codekaram.brickhouse.loadLayers -> java.util.Spliterator                java.base
com.codekaram.brickhouse.loadLayers -> java.util.function.Consumer          java.base
com.codekaram.brickhouse.loadLayers -> java.util.function.Function          java.base
com.codekaram.brickhouse.loadLayers -> java.util.function.IntConsumer       java.base
com.codekaram.brickhouse.loadLayers -> java.util.function.Predicate         java.base
com.codekaram.brickhouse.loadLayers -> java.util.stream.IntStream           java.base
com.codekaram.brickhouse.loadLayers -> java.util.stream.Stream              java.base
com.codekaram.brickhouse.loadLayers -> java.util.stream.StreamSupport       java.base
```

Jdeprscan

Jdeprscan is a tool that analyzes the usage of deprecated APIs in modules. Deprecated APIs are older APIs that the Java community has replaced with newer ones. These older APIs are still supported but are marked for removal in future releases. *Jdeprscan* helps developers maintain their code by suggesting alternative solutions to these deprecated APIs, aiding them in transitioning to newer, supported APIs.

Here's how you can use *jdeprscan* on the `com.codekaram.brickhouse` module:

```
$ jdeprscan --for-removal mods/com.codekaram.brickhouse
No deprecated API marked for removal found.
```

In this example, *jdeprscan* is used to scan the `com.codekaram.brickhouse` module for deprecated APIs that are marked for removal. The output indicates that no such deprecated APIs are found.

You can also use `--list` to see all deprecated APIs in a module:

```
$ jdeprscan --list mods/com.codekaram.brickhouse
No deprecated API found.
```

In this case, no deprecated APIs are found in the `com.codekaram.brickhouse` module.

Jmod

Jmod is a tool used to create, describe, and list JMOD files. JMOD files are an alternative to JAR files for packaging modular Java applications, which offer additional features such as native code and configuration files. These files can be used for distribution or to create custom runtime images with *jlink*.

Here's how you can use *jmod* to create a JMOD file for the `brickhouse` example. Let's first compile and package the module specific to this example:

```
$ javac --module-source-path src -d build/modules $(find src -name "*.java")
$ jmod create --class-path build/modules/com.codekaram.brickhouse com.codekaram.brickhouse.jmod
```

Here, the `jmod create` command is used to create a JMOD file named `com.codekaram.brickhouse.jmod` from the `com.codekaram.brickhouse` module located in the `build/modules` directory. You can then use the `jmod describe` command to display information about the JMOD file:

```
$ jmod describe com.codekaram.brickhouse.jmod
```

This command will output the module descriptor and any additional information about the JMOD file.

Additionally, you can use the `jmod list` command to display the contents of the created JMOD file:

```
$ jmod list com.codekaram.brickhouse.jmod
com/codekaram/brickhouse/
com/codekaram/brickhouse/loadLayers.class
com/codekaram/brickhouse/loadLayers$1.class
…
```

The output lists the contents of the `com.codekaram.brickhouse.jmod` file, showing the package structure and class files.

By using *jmod* to create JMOD files, describe their contents, and list their individual files, you can gain a better understanding of your modular application's structure and streamline the process of creating custom runtime images with *jlink*.