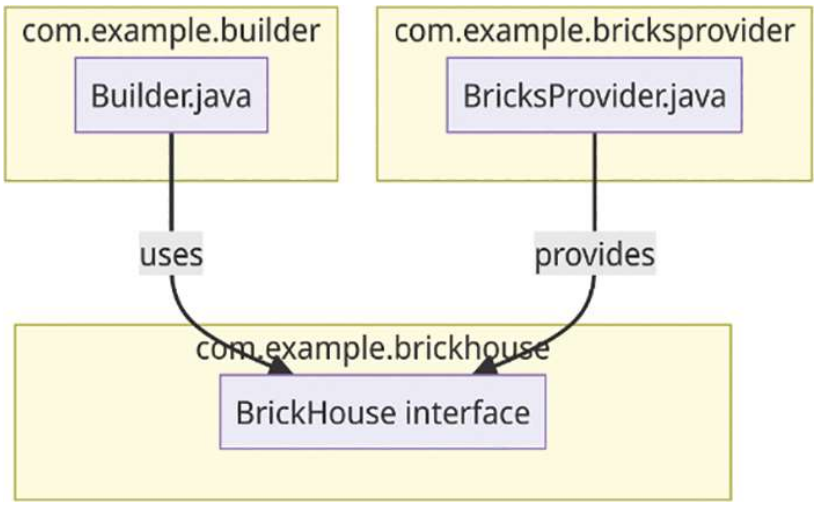

Figure 3.3 Modular Services

Figure 3.3 depicts the relationships between the modules and classes in a module diagram. The module diagram represents the dependencies and relationships between the modules and classes:

- The `com.example.builder` module contains the `Builder.java` class, which uses the `BrickHouse` interface from the `com.example.brickhouse` module.
- The `com.example.bricksprovider` module contains the `BricksProvider.java` class, which implements and provides the `BrickHouse` interface.

Implementation Details

The `ServiceLoader` API is a powerful mechanism that allows the `com.example.builder` module to discover and instantiate the `BrickHouse` implementation provided by the `com.example.bricksprovider` module at runtime. This allows for more flexibility and better separation of concerns between modules. The following subsections focus on some implementation details that can help us better understand the interactions between the modules and the role of the `ServiceLoader` API.

Discovering Service Implementations

The `ServiceLoader.load()` method takes a service interface as its argument—in our case, `BrickHouse.class`—and returns a `ServiceLoader` instance. This instance is an iterable object containing all available service implementations. The `ServiceLoader` relies on the information provided in the `module-info.java` files to discover the service implementations.

Instantiating Service Implementations

When iterating over the `ServiceLoader` instance, the API automatically instantiates the service implementations provided by the service providers. In our example, the `BricksProvider` class is instantiated, and its `build()` method is called when iterating over the `ServiceLoader` instance.

Encapsulating Implementation Details

By using the JPMS, the `com.example.bricksprovider` module can encapsulate its implementation details, exposing only the `BrickHouse` service that it provides. This allows the `com.example.builder` module to consume the service without depending on the concrete implementation, creating a more robust and maintainable system.

Adding More Service Providers

Our example can be easily extended by adding more service providers implementing the `BrickHouse` interface. As long as the new service providers are properly declared in their respective `module-info.java` files, the `com.example.builder` module will be able to discover and use them automatically through the `ServiceLoader` API. This allows for a more modular and extensible system that can adapt to changing requirements or new implementations.

Figure 3.4 is a use-case diagram that depicts the interactions between the service consumer and service provider. It includes two actors: Service Consumer and Service Provider.

- **Service Consumer:** This uses the services provided by the Service Provider. The Service Consumer interacts with the Modular JDK in the following ways:
  - **Discover Service Implementations:** The Service Consumer uses the Modular JDK to find available service implementations.
  - **Instantiate Service Implementations:** Once the service implementations are discovered, the Service Consumer uses the Modular JDK to create instances of these services.
  - **Encapsulate Implementation Details:** The Service Consumer benefits from the encapsulation provided by the Modular JDK, which allows it to use services without needing to know their underlying implementation.
- **Service Provider:** This implements and provides the services. The Service Provider interacts with the Modular JDK in the following ways:
  - **Implement Service Interface:** The Service Provider uses the Modular JDK to implement the service interface, which defines the contract for the service.
  - **Encapsulate Implementation Details:** The Service Provider uses the Modular JDK to hide the details of its service implementation, exposing only the service interface.
  - **Add More Service Providers:** The Service Provider can use the Modular JDK to add more providers for the service, enhancing the modularity and extensibility of the system.

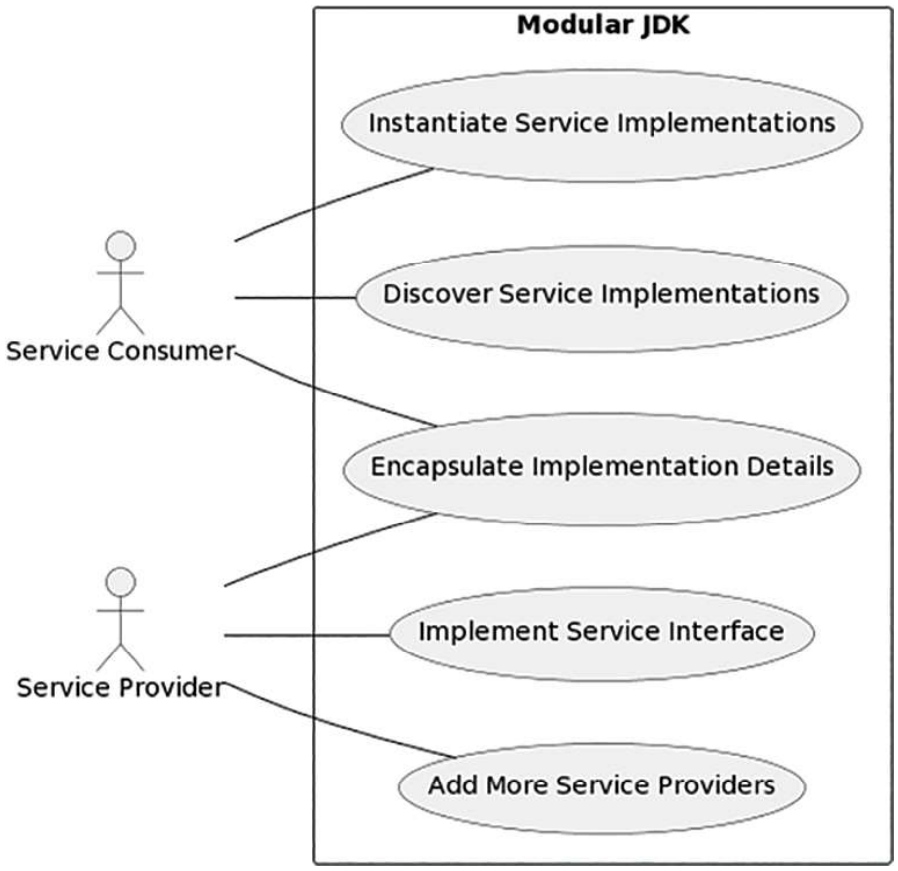

Figure 3.4 Use-Case Diagram Highlighting the Service Consumer and Service Provider

The Modular JDK acts as a robust facilitator for these interactions, establishing a comprehensive platform where service providers can effectively offer their services. Simultaneously, it provides an avenue for service consumers to discover and utilize these services efficiently. This dynamic ecosystem fosters a seamless exchange of services, enhancing the overall functionality and interoperability of modular Java applications.

JAR Hell Versioning Problem and Jigsaw Layers

Before diving into the details of the JAR hell versioning problem and Jigsaw layers, I'd like to introduce Nikita Lipski, a fellow JVM engineer and an expert in the field of Java modularity. Nikita has provided valuable insights and a comprehensive write-up on this topic, which we will be discussing in this section. His work will help us better understand the JAR hell versioning problem and how Jigsaw layers can be utilized to address this issue in JDK 11 and JDK 17.

Java's backward compatibility is one of its key features. This compatibility ensures that when a new version of Java is released, applications built for older versions can run on the new version without any changes to the source code, and often even without recompilation. The same principle applies to third-party libraries—applications can work with updated versions of the libraries without modifications to the source code.

However, this compatibility does not extend to versioning at the source level, and the JPMS does not introduce versioning at this level, either. Instead, versioning is managed at the artifact level, using artifact management systems like Maven or Gradle. These systems handle versioning and dependency management for the libraries and frameworks used in Java projects, ensuring that the correct versions of the dependencies are included in the build process. But what happens when a Java application depends on multiple third-party libraries, which in turn may depend on different versions of another library? This can lead to conflicts and runtime errors if multiple versions of the same library are present on the classpath.

So, although JPMS has certainly improved modularity and code organization in Java, the "JAR hell" problem can still be relevant when dealing with versioning at the artifact level. Let's look at an example (shown in Figure 3.5) where an application depends on two third-party libraries (Foo and Bar), which in turn depend on different versions of another library (Baz).

If both versions of the Baz library are placed on the classpath, it becomes unclear which version of the library will be used at runtime, resulting in unavoidable version conflicts. To address this issue, JPMS prohibits such situations by detecting split packages, which are not allowed in JPMS, in support of its "reliable configuration" goal (Figure 3.6).

While detecting versioning problems early is useful, JPMS does not provide a recommended way to resolve them. One approach to address these problems is to use the latest version of the conflicting library, assuming it is backward compatible. However, this might not always be possible due to introduced incompatibilities.

To address such cases, JPMS offers the *ModuleLayer* feature, which allows for the installation of a module sub-graph into the module system in an isolated manner. When different versions of the conflicting library are placed into separate layers, both of those versions can be loaded by JPMS. Although there is no direct way to access a module of the child layer from the parent layer, this can be achieved indirectly—by implementing a service provider in the child layer module, which the parent layer module can then use. (See the earlier discussion of "Implementing Modular Services with JDK 17" for more details.)

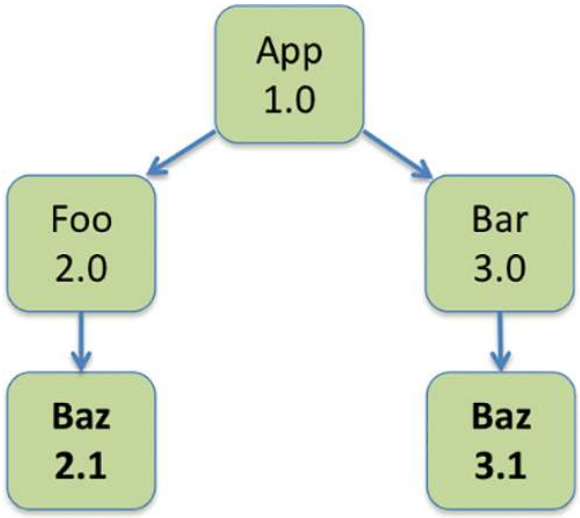

Figure 3.5 Modularity and Version Conflicts

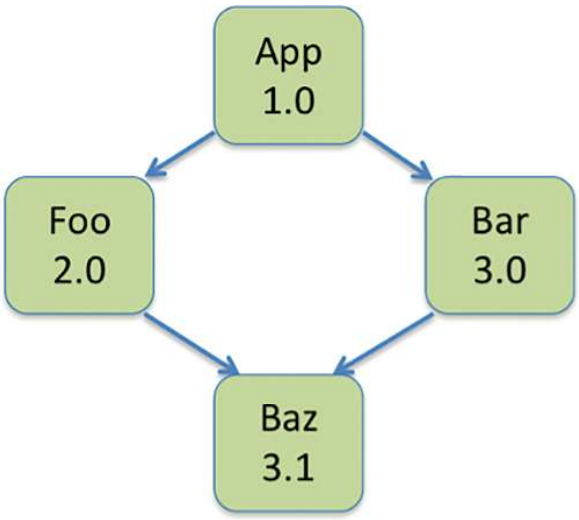

Figure 3.6 Reliable Configuration with JPMS

Working Example: JAR Hell

In this section, a working example is provided to demonstrate the use of module layers in addressing the JAR hell problem in the context of JDK 17 (this strategy is applicable to JDK 11 users as well). This example builds upon Nikita's explanation and the house service provider implementation we discussed earlier. It demonstrates how you can work with different versions of a library (termed basic and high-quality implementations) within a modular application.

First, let's take a look at the sample code provided by Java SE 9 documentation:[1]

```
1  ModuleFinder finder = ModuleFinder.of(dir1, dir2, dir3);
2  ModuleLayer parent = ModuleLayer.boot();
3  Configuration cf = parent.configuration().resolve(finder, ModuleFinder.of(),
   Set.of("myapp"));
4  ClassLoader scl = ClassLoader.getSystemClassLoader();
5  ModuleLayer layer = parent.defineModulesWithOneLoader(cf, scl);
```

In this example:

- At line 1, a `ModuleFinder` is set up to locate modules from specific directories (`dir1`, `dir2`, and `dir3`).
- At line 2, the boot layer is established as the parent layer.
- At line 3, the boot layer's configuration is resolved as the parent configuration for the modules found in the directories specified in line 1.
- At line 5, a new layer with the resolved configuration is created, using a single class loader with the system class loader as its parent.

[1] https://docs.oracle.com/javase/9/docs/api/java/lang/ModuleLayer.html

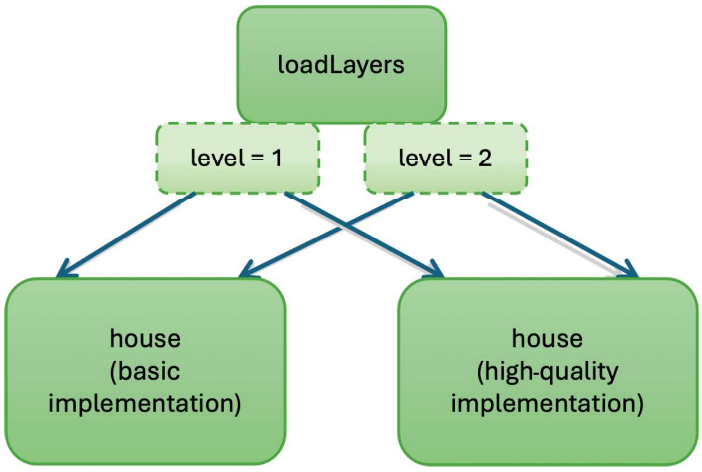

Figure 3.7 JPMS Example Versions and Layers

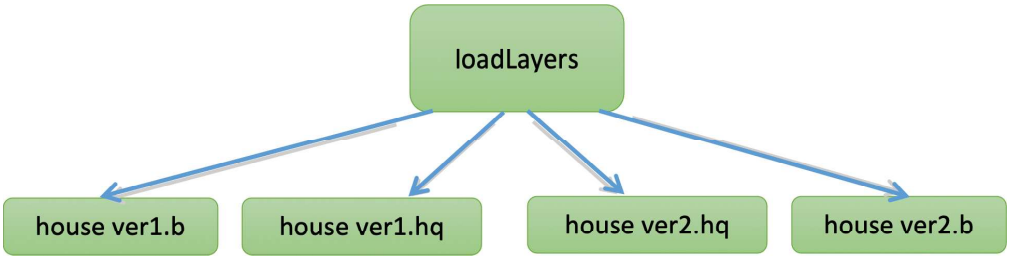

Figure 3.8 JPMS Example Version Layers Flattened

Now, let's extend our house service provider implementation. We'll have basic and high-quality implementations provided in the `com.codekaram.provider` modules. You can think of the "basic implementation" as version 1 of the house library and the "high-quality implementation" as version 2 of the house library (Figure 3.7).

For each level, we will reach out to both the libraries. So, our combinations would be level 1 + basic implementation provider, level 1 + high-quality implementation provider, level 2 + basic implementation provider, and level 2 + high-quality implementation provider. For simplicity, let's denote the combinations as *house ver1.b*, *house ver1.hq*, *house ver2.b*, and *house ver2.hq*, respectively (Figure 3.8).

Implementation Details

Building upon the concepts introduced by Nikita in the previous section, let's dive into the implementation details and understand how the layers' structure and program flow work in practice. First, let's look at the source trees:

```
ModuleLayer
├── basic
│   └── src
│       └── com.codekaram.provider
```

```
│           ├── classes
│           │   ├── com
│           │   │   └── codekaram
│           │   │       └── provider
│           │   │           └── House.java
│           │   └── module-info.java
│           └── tests
├── high-quality
│   └── src
│       └── com.codekaram.provider
│           ├── classes
│           │   ├── com
│           │   │   └── codekaram
│           │   │       └── provider
│           │   │           └── House.java
│           │   └── module-info.java
│           └── tests
└── src
    └── com.codekaram.brickhouse
        ├── classes
        │   ├── com
        │   │   └── codekaram
        │   │       └── brickhouse
        │   │           ├── loadLayers.java
        │   │           └── spi
        │   │               └── BricksProvider.java
        │   └── module-info.java
        └── tests
```

Here's the module file information and the module graph for `com.codekaram.provider`. Note that these look exactly the same for both the basic and high-quality implementations.

```
module com.codekaram.provider {
    requires com.codekaram.brickhouse;
    uses com.codekaram.brickhouse.spi.BricksProvider;
    provides com.codekaram.brickhouse.spi.BricksProvider with com.codekaram.provider.House;
}
```

The module diagram (shown in Figure 3.9) helps visualize the dependencies between modules and the services they provide, which can be useful for understanding the structure of a modular Java application:

- The `com.codekaram.provider` module depends on the `com.codekaram.brickhouse` module and implicitly depends on the `java.base` module, which is the foundational module of every Java application. This is indicated by the arrows pointing from `com.`

`codekaram.provider` to `com.codekaram.brickhouse` and the assumed arrow to `java.base`.

- The `com.codekaram.brickhouse` module also implicitly depends on the `java.base` module, as all Java modules do.

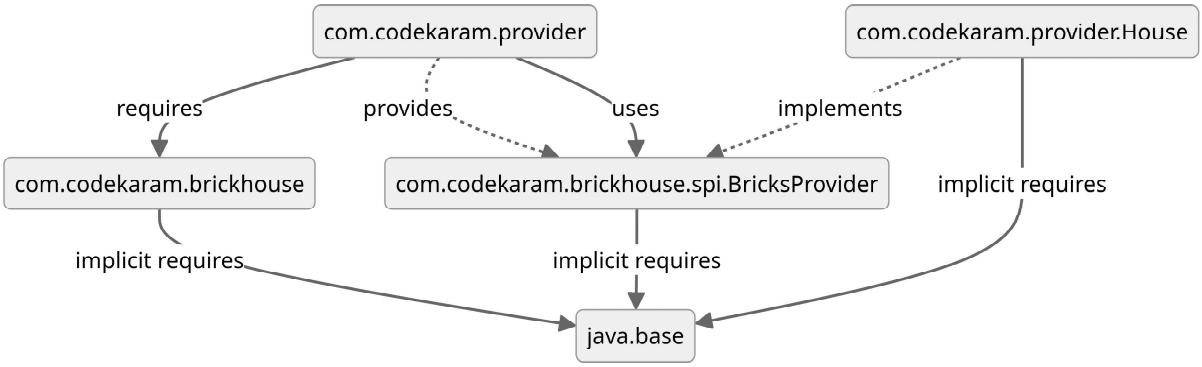

Figure 3.9 A Working Example with Services and Layers

- The `java.base` module does not depend on any other module and is the core module upon which all other modules rely.
- The `com.codekaram.provider` module provides the service `com.codekaram.brickhouse.spi.BricksProvider` with the implementation `com.codekaram.provider.House`. This relationship is represented in the graph by a dashed arrow from `com.codekaram.provider` to `com.codekaram.brickhouse.spi.BricksProvider`.

Before diving into the code for these providers, let's look at the module file information for the `com.codekaram.brickhouse` module:

```
module com.codekaram.brickhouse {
    uses com.codekaram.brickhouse.spi.BricksProvider;
    exports com.codekaram.brickhouse.spi;
}
```

The `loadLayers` class will not only handle forming layers, but also be able to load the service providers for each level. That's a bit of a simplification, but it helps us to better understand the flow. Now, let's examine the `loadLayers` implementation. Here's the creation of the layers' code based on the sample code from the "Working Example: JAR Hell" section:

```
static ModuleLayer getProviderLayer(String getCustomDir) {
   ModuleFinder finder = ModuleFinder.of(Paths.get(getCustomDir));
   ModuleLayer parent = ModuleLayer.boot();
   Configuration cf = parent.configuration().resolve(finder,
     ModuleFinder.of(), Set.of("com.codekaram.provider"));
   ClassLoader scl = ClassLoader.getSystemClassLoader();
   ModuleLayer layer = parent.defineModulesWithOneLoader(cf, scl);
```

```
   System.out.println("Created a new layer for " + layer);
   return layer;
}
```

If we simply want to create two layers, one for house version `basic` and another for house version `high-quality`, all we have to do is call `getProviderLayer()` (from the `main` method):

```
doWork(Stream.of(args)
       .map(getCustomDir -> getProviderLayer(getCustomDir)));
```

If we pass the two directories `basic` and `high-quality` as runtime parameters, the `getProviderLayer()` method will look for `com.codekaram.provider` in those directories and then create a layer for each. Let's examine the output (the line numbers have been added for the purpose of clarity and explanation):

```
1  $ java --module-path mods -m com.codekaram.brickhouse/
   com.codekaram.brickhouse.loadLayers basic high-quality
2  Created a new layer for com.codekaram.provider
3  I am the basic provider
4  Created a new layer for com.codekaram.provider
5  I am the high-quality provider
```

- Line 1 is our command-line argument with `basic` and `high-quality` as directories that provide the implementation of the `BrickProvider` service.
- Lines 2 and 4 are outputs indicating that `com.codekaram.provider` was found in both the directories and a new layer was created for each.
- Lines 3 and 5 are the output of `provider.getName()` as implemented in the `doWork()` code:

```
private static void doWork(Stream<ModuleLayer> myLayers){
myLayers.flatMap(moduleLayer -> ServiceLoader
   .load(moduleLayer, BricksProvider.class)
    .stream().map(ServiceLoader.Provider::get))
  .forEach(eachSLProvider -> System.out.println("I am the " + eachSLProvider.getName() +
" provider"));}
```

In `doWork()`, we first create a service loader for the `BricksProvider` service and load the provider from the module layer. We then print the return `String` of the `getName()` method for that provider. As seen in the output, we have two module layers and we were successful in printing the `I am the basic provider` and `I am the high-quality provider` outputs, where `basic` and `high-quality` are the return strings of the `getName()` method.

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*.

Jlink

Jlink is a tool that helps link modules and their transitive dependencies to create custom modular runtime images. These custom images can be packaged and deployed without needing the entire Java Runtime Environment (JRE), which makes your application lighter and faster to start.

To use the *jlink* command, this tool needs to be added to your path. First, ensure that the `$JAVA_HOME/bin` is in the path. Next, type `jlink` on the command line:

```
$ jlink
Error: --module-path must be specified
Usage: jlink <options> --module-path <modulepath> --add-modules <module>[,<module>...]
Use --help for a list of possible options
```

Here's how you can use *jlink* for the code shown in "Implementing Modular Services with JDK 17":

```
$ jlink --module-path $JAVA_HOME/jmods:build/modules --add-modules com.example.builder --output
consumer.services --bind-services
```

A few notes on this example:

- The command includes a directory called `$JAVA_HOME/jmods` in the `module-path`. This directory contains the `java.base.jmod` needed for all application modules.
- Because the module is a consumer of services, it's necessary to link the service providers (and their dependencies). Hence, the `–bind-services` option is used.
- The runtime image will be available in the `consumer.services` directory as shown here:

```
$ ls consumer.services/
bin conf      include   legal     lib       release
```

Let's now run the image:

```
$ consumer.services/bin/java -m com.example.builder/com.example.builder.Builder
Building a house with bricks...
```

With *jlink*, you can create lightweight, custom, stand-alone runtime images tailored to your modular Java applications, thereby simplifying the deployment and reducing the size of your distributed application.

Conclusion

This chapter has undertaken a comprehensive exploration of Java modules, tools, and techniques to create and manage modular applications. We have delved into the Java Platform Module System (JPMS), highlighting its benefits such as reliable configuration and strong encapsulation. These features contribute to more maintainable and scalable applications.

We navigated the intricacies of creating, packaging, and managing modules, and explored the use of module layers to enhance application flexibility. These practices can help address common challenges faced when migrating to newer JDK versions (e.g., JDK 11 or JDK 17), including updating project structures and ensuring dependency compatibility.

Performance Implications

The use of modular Java carries significant performance implications. By including only the necessary modules in your application, the JVM loads fewer classes, which improves start-up performance and reduces the memory footprint. This is particularly beneficial in resource-limited environments such as microservices running in containers. However, it is important to note that while modularity can improve performance, it also introduces a level of complexity. For instance, improper module design can lead to cyclic dependencies,[2] negatively impacting performance. Therefore, careful design and understanding of modules are essential to fully reap the performance benefits.

Tools and Future Developments

We examined the use of powerful tools like *jdeps*, *jdeprscan*, *jmod*, and *jlink*, which are instrumental in identifying and addressing compatibility issues, creating custom runtime images, and streamlining the deployment of modular applications. Looking ahead, we can anticipate more advanced options for creating custom runtime images with *jlink*, and more detailed and accurate dependency analysis with *jdeps*.

As more developers adopt modular Java, new best practices and patterns will emerge, alongside new tools and libraries designed to work with JPMS. The Java community is continuously improving JPMS, with future Java versions expected to refine and expand its capabilities.

Embracing the Modular Programming Paradigm

Transitioning to modular Java can present unique challenges, especially in understanding and implementing modular structures in large-scale applications. Compatibility issues may arise with third-party libraries or frameworks that may not be fully compatible with JPMS. These challenges, while part of the journey toward modernization, are often outweighed by the benefits of modular Java, such as improved performance, enhanced scalability, and better maintainability.

In conclusion, by leveraging the knowledge gained from this chapter, you can confidently migrate your projects and fully harness the potential of modular Java applications. The future of modular Java is exciting, and embracing this paradigm will equip you to meet the evolving needs of the software development landscape. It's an exciting time to be working with modular Java, and we look forward to seeing how it evolves and shapes the future of robust and efficient Java applications.

[2] https://openjdk.org/projects/jigsaw/spec/issues/#CyclicDependences

This page intentionally left blank

Chapter 4

The Unified Java Virtual Machine Logging Interface

The Java HotSpot Virtual Machine (VM) is a treasure trove of features and flexibility. Command-line options allow for enabling its experimental or diagnostic features and exploring various VM and garbage collection activities. Within the HotSpot VM, there are distinct builds tailored for different purposes. The primary version is the *product* build, optimized for performance and used in production environments. In addition, there are *debug* builds, which include the *fast-debug* and the *slow-debug* variants.

> **NOTE** The asserts-enabled *fast-debug* build offers additional debugging capabilities with minimal performance overhead, making it suitable for development environments where some level of diagnostics is needed without significantly compromising performance. On the other hand, the *slow-debug* build includes comprehensive debugging tools and checks, ideal for in-depth analysis with native debuggers but with a significant performance trade-off.

Until JDK 9, the logging interface of the HotSpot VM was fragmented, lacking unification. Developers had to combine various print options to retrieve detailed logging information with appropriate timestamps and context. This chapter guides you through the evolution of this feature, focusing on the unified logging system introduced in JDK 9, which was further enhanced in JDK 11 and JDK 17.

The Need for Unified Logging

The Java HotSpot VM is rich in options, but its lack of a unified logging interface was a significant shortcoming. Various command-line options were necessary to retrieve specific logging details, which could have been clearer and more efficient. A brief overview of these options illustrates the complexity (many more options were available, aside from those listed here):

- Garbage collector (GC) event timestamp: `-XX:+PrintGCTimeStamps` or `-XX:+PrintGCDateStamps`
- GC event details: `-XX:+PrintGCDetails`
- GC event cause: `-XX:+PrintGCCause`

- GC adaptiveness: `-XX:+PrintAdaptiveSizePolicy`
- GC-specific options (e.g., for the Garbage First [G1] GC): `-XX:+G1PrintHeapRegions`
- Compilation levels: `-XX:+PrintCompilation`
- Inlining information: `-XX:+PrintInlining`
- Assembly code: `-XX:+PrintAssembly` (requires `-XX:+UnlockDiagnosticVMOptions`)
- Class histogram: `-XX:+PrintClassHistogram`
- Information on `java.util.concurrent` locks: `-XX:+PrintConcurrentLocks`

Memorizing each option and determining whether the information is at the info, debug, or tracing level is challenging. Thus, JDK Enhancement Proposal (JEP) 158: *Unified JVM Logging*[1] was introduced in JDK 9. It was followed by JEP 271: *Unified GC Logging*,[2] which offered a comprehensive logging framework for GC activities.

Unification and Infrastructure

The Java HotSpot VM has a unified logging interface that provides a familiar look and approach to all logging information for the JVM. All log messages are characterized using tags. This enables you to request the level of information necessary for your level of investigation, from "error" and "warning" logs to "info," "trace," and "debug" levels.

To enable JVM logging, you need to provide the `-Xlog` option. By default (i.e., when no `-Xlog` is specified), only warnings and errors are logged to *stderr*. So, the default configuration looks like this:

```
-Xlog:all=warning:stderr:uptime,level,tags
```

Here, we see `all`, which is simply an alias for all tags.

The unified logging system is an all-encompassing system that provides a consistent interface for all JVM logging. It is built around four main components:

- **Tags:** Used to categorize log messages, with each message assigned one or more tags. Tags make it easier to filter and find specific log messages. Some of the predefined tags include `gc`, `compiler`, and `threads`.
- **Levels:** Define the severity or importance of a log message. Six levels are available: `debug`, `error`, `info`, `off`, `trace`, and `warning`.
- **Decorators:** Provide additional context to a log message, such as timestamps, process IDs, and more.
- **Outputs:** The destinations where the log messages are written, such as *stdout*, *stderr*, or a file.

[1] https://openjdk.org/jeps/158

[2] https://openjdk.org/jeps/271

Performance Metrics

In the context of JVM performance engineering, the unified logging system provides valuable insights into various performance metrics. These metrics include GC activities, just-in-time (JIT) compilation events, and system resource usage, among others. By monitoring these metrics, developers can identify potential performance issues and preempt necessary actions to optimize the performance of their Java applications.

Tags in the Unified Logging System

In the unified logging system, tags are crucial for segregating and identifying different types of log messages. Each tag corresponds to a particular system area or a specific type of operation. By enabling specific tags, users can tailor the logging output to focus on areas of interest.

Log Tags

The unified logging system provides a myriad of tags to capture granular information. Tags such as `gc` for garbage collection, `thread` for thread operations, `class` for class loading, `cpu` for CPU-related events, `os` for operating system interactions, and many more as highlighted here:

```
add, age, alloc, annotation, arguments, attach, barrier, biasedlocking, blocks, bot, breakpoint,
bytecode, cds, census, class, classhisto, cleanup, codecache, compaction, compilation, condy,
constantpool, constraints, container, coops, cpu, cset, data, datacreation, dcmd, decoder,
defaultmethods, director, dump, dynamic, ergo, event, exceptions, exit, fingerprint, free,
freelist, gc, handshake, hashtables, heap, humongous, ihop, iklass, indy, init, inlining, install,
interpreter, itables, jfr, jit, jni, jvmci, jvmti, lambda, library, liveness, load, loader,
logging, malloc, map, mark, marking, membername, memops, metadata, metaspace, methodcomparator,
methodhandles, mirror, mmu, module, monitorinflation, monitormismatch, nestmates, nmethod,
nmt, normalize, numa, objecttagging, obsolete, oldobject, oom, oopmap, oops, oopstorage, os,
owner, pagesize, parser, patch, path, perf, periodic, phases, plab, placeholders, preorder,
preview, promotion, protectiondomain, ptrqueue, purge, record, redefine, ref, refine, region,
reloc, remset, resolve, safepoint, sampling, scavenge, sealed, setting, smr, stackbarrier,
stackmap, stacktrace, stackwalk, start, startup, startuptime, state, stats, streaming,
stringdedup, stringtable, subclass, survivor, suspend, sweep, symboltable, system, table, task,
thread, throttle, time, timer, tlab, tracking, unload, unshareable, update, valuebasedclasses,
verification, verify, vmmutex, vmoperation, vmthread, vtables, vtablestubs, workgang
```

Specifying `all` instead of a tag combination matches all tag combinations.

If you want to run your application with all log messages, you could execute your program with `-Xlog`, which is equivalent to `-Xlog:all`. This option will log all messages to *stdout* with the level set at `info`, while warnings and errors will go to *stderr*.

Specific Tags

By default, running your application with `-Xlog` but without specific tags will produce a large volume of log messages from various system parts. If you want to focus on specific areas, you can do so by specifying the tags.

For instance, if you are interested in log messages related to garbage collection, the heap, and compressed oops, you could specify the `gc` tag. Running your application in JDK 17 with the `-Xlog:gc*` option would produce log messages related to garbage collection:

```
$ java -Xlog:gc* LockLoops
[0.006s][info][gc] Using G1
[0.007s][info][gc,init] Version: 17.0.8+9-LTS-211 (release)
[0.007s][info][gc,init] CPUs: 16 total, 16 available
…
[0.057s][info][gc,metaspace] Compressed class space mapped at: 0x0000000132000000
-0x0000000172000000, reserved size: 1073741824
[0.057s][info][gc,metaspace] Narrow klass base: 0x0000000131000000, Narrow klass shift: 0, Narrow
klass range: 0x100000000
```

Similarly, you might be interested in the `thread` tag if your application involves heavy multi-threading. Running your application with the `-Xlog:thread*` option would produce log messages related to thread operations:

```
$ java -Xlog:thread* LockLoops
[0.019s][info][os,thread] Thread attached (tid: 7427, pthread id: 123145528827904).
[0.030s][info][os,thread] Thread "GC Thread#0" started (pthread id: 123145529888768, attributes:
stacksize: 1024k, guardsize: 4k, detached).
…
```

Identifying Missing Information

In some situations, enabling a tag doesn't produce the expected log messages. For instance, running your application with `-Xlog:monitorinflation*` might produce scant output, as shown here:

```
…
[11.301s][info][monitorinflation] deflated 1 monitors in 0.0000023 secs
[11.301s][info][monitorinflation] end deflating: in_use_list stats: ceiling=11264, count=2, max=3
…
```

In such cases, you may need to adjust the log level—a topic that we'll explore in the next section.

Diving into Levels, Outputs, and Decorators

Let's take a closer look at the available log levels, examine the use of decorators, and discuss ways to redirect outputs.

Levels

The JVM's unified logging system provides various logging levels, each corresponding to a different amount of detail. To understand these levels, you can use the `-Xlog:help` command, which offers information on all available options:

```
Available log levels:
 off, trace, debug, info, warning, error
```

Here,

- **`off`:** Disable logging.
- **`error`:** Critical issues within the JVM.
- **`warning`:** Potential issues that might warrant attention.
- **`info`:** General information about JVM operations.
- **`debug`:** Detailed information useful for debugging. This level provides in-depth insights into the JVM's behavior and is typically used when troubleshooting specific issues.
- **`trace`:** The most verbose level, providing extremely detailed logging. Trace level is often used for the most granular insights into JVM operations, especially when a detailed step-by-step account of events is required.

As shown in our previous examples, the default log level is set to `info` if no specific log level is given for a tag. When we ran our test, we observed very little information for the `monitorinflation` tag when we used the default `info` log level. We can explicitly set the log level to `trace` to access additional information, as shown here:

```
$ java -Xlog:monitorinflation*=trace LockLoops
```

The output logs now contain detailed information about the various locks, as shown in the following log excerpt:

```
…
[3.073s][trace][monitorinflation      ] Checking in_use_list:
[3.073s][trace][monitorinflation      ] count=3, max=3
[3.073s][trace][monitorinflation      ] in_use_count=3 equals ck_in_use_count=3
[3.073s][trace][monitorinflation      ] in_use_max=3 equals ck_in_use_max=3
[3.073s][trace][monitorinflation      ] No errors found in in_use_list checks.
[3.073s][trace][monitorinflation      ] In-use monitor info:
[3.073s][trace][monitorinflation      ] (B -> is_busy, H -> has hash code, L -> lock status)
```

```
[3.073s][trace][monitorinflation      ]          monitor  BHL            object       object type
[3.073s][trace][monitorinflation      ] ================  ===  ================  ==================
[3.073s][trace][monitorinflation      ] 0x00006000020ec680  100  0x000000070fe190e0  java.
lang.ref.ReferenceQueue$Lock (is_busy: waiters=1, contentions=0owner=0x0000000000000000,
cxq=0x0000000000000000, EntryList=0x0000000000000000)
…
```

In the unified logging system, log levels are hierarchical. Hence, setting the log level to `trace` will include messages from all lower levels (`debug`, `info`, `warning`, `error`) as well. This ensures a comprehensive logging output, capturing a wide range of details, as can be seen further down in the log:

```
…
[11.046s][trace][monitorinflation      ] deflate_monitor: object=0x000000070fe70a58,
mark=0x00006000020e00d2, type='java.lang.Object'
[11.046s][debug][monitorinflation      ] deflated 1 monitors in 0.0000157 secs
[11.046s][debug][monitorinflation      ] end deflating: in_use_list stats: ceiling=11264, count=2,
max=3
[11.046s][debug][monitorinflation      ] begin deflating: in_use_list stats: ceiling=11264,
count=2, max=3
[11.046s][debug][monitorinflation      ] deflated 0 monitors in 0.0000004 secs
…
```

This hierarchical logging is a crucial aspect of the JVM's logging system, allowing for flexible and detailed monitoring based on your requirements.

Decorators

Decorators are another critical aspect of JVM logging. Decorators enrich log messages with additional context, making them more informative. Specifying `-Xlog:help` on the command line will yield the following decorators:

```
…
Available log decorators:
 time (t), utctime (utc), uptime (u), timemillis (tm), uptimemillis (um), timenanos (tn),
uptimenanos (un), hostname (hn), pid (p), tid (ti), level (l), tags (tg)
 Decorators can also be specified as 'none' for no decoration.
…
```

These decorators provide specific information in the logs. By default, `uptime`, `level`, and `tags` are selected, which is why we saw these three decorators in the log output in our earlier examples. However, you might sometimes want to use different decorators, such as

- **`pid` (`p`):** Process ID. Useful in distinguishing logs from different JVM instances.

- **`tid` (`ti`):** Thread ID. Crucial for tracing actions of specific threads, aiding in debugging concurrency issues or thread-specific behavior.
- **`uptimemillis` (`um`):** JVM uptime in milliseconds. Helps in correlating events with the timeline of the application's operation, especially in performance monitoring.

Here's how we can add the decorators:

```
$ java -Xlog:gc*::uptimemillis,pid,tid LockLoops
[0.006s][32262][8707] Using G1
[0.008s][32262][8707] Version: 17.0.8+9-LTS-211 (release)
[0.008s][32262][8707] CPUs: 16 total, 16 available [0.023s][32033][9987] Memory: 16384M
…
[0.057s][32262][8707] Compressed class space mapped at: 0x000000012a000000-0x000000016a000000,
reserved size: 1073741824
[0.057s][32262][8707] Narrow klass base: 0x0000000129000000, Narrow klass shift: 0, Narrow klass
range: 0x100000000
```

Decorators like `pid` and `tid` are particularly valuable in debugging. When multiple JVMs are running, `pid` helps identify which JVM instance generated the log. `tid` is crucial in multi-threaded applications for tracking the behavior of individual threads, which is vital in diagnosing issues like deadlocks or race conditions.

Use `uptime` or `uptimemillis` for a time reference for events, enabling a clear timeline of operations. `uptimemillis` is essential in performance analysis to understand when specific events occur relative to the application's start time. Similarly, `hostname` can be valuable in distributed systems to identify the source machine of log entries.

Decorators enhance log readability and analysis in several ways:

- **Contextual clarity:** By providing additional details such as timestamps, process IDs, and thread IDs, decorators make logs self-contained and more understandable.
- **Filtering and searching:** With relevant decorators, log data can be filtered more effectively, enabling quicker isolation of issues.
- **Correlation and causation analysis:** Decorators allow for correlating events across different parts of the system, aiding in root-cause analysis and system-wide performance optimization.

In summary, decorators in the unified logging system play a pivotal role in enhancing the debugging and performance monitoring capabilities of JVM logs. They add necessary context and clarity, making log data more actionable and insightful for developers.

Outputs

In JVM logging, outputs determine where your log information is directed. The JVM provides flexibility in controlling the output of the logs, allowing users to redirect the output to *stdout*, *stderr*, or a specific file. The default behavior is to route all warnings and errors to *stderr*, while

all `info`, `debug`, and `trace` levels are directed to *stdout*. However, users can adjust this behavior to suit their needs. For instance, a user can specify a particular file to write the log data, which can be useful when the logs need to be analyzed later or when *stdout*/*stderr* is not convenient or desired for log data.

A file name can be passed as a parameter to the `-Xlog` command to redirect the log output to that file. For example, suppose you want to log the garbage collection (`gc`) activities at the `info` level and want these logs to be written to the `gclog.txt` file. Here's an example for the `gc*` tag and level set to `info`:

```
$ java -Xlog:gc*=info:file=gclog.txt LockLoops
```

In this example, all log information about the garbage collection process at the `info` level will be written to the `gclog.txt` file. The same can be done for any tag or log level, providing tremendous flexibility to system administrators and developers.

You can even direct different log levels to different outputs, as in the following example:

```
$ java -Xlog:monitorinflation*=trace:file=lockinflation.txt -Xlog:gc*=info:file=gclog.txt
LockLoops
```

In this case, the `info`-level logs of the garbage collection process are written into `gclog.txt`, and the `trace`-level logs for `monitorinflation` are directed to `lockinflation.txt`. This allows a fine-grained approach to managing log data and enables the segregation of log data based on its verbosity level, which can be extremely helpful during troubleshooting.

NOTE Directing logs to a file rather than *stdout*/*stderr* can have performance implications, particularly for I/O-intensive applications. Writing to a file may be slower and could impact application performance. To mitigate this risk, asynchronous logging can be an effective solution. We will dive into asynchronous logging a bit later in this chapter.

Understanding and effectively managing the outputs in JVM logging can help you take full advantage of the flexibility and power of the logging system. Also, given that the JVM does not automatically handle log rotation, managing large log files becomes crucial for long-running applications. Without proper log management, log files can grow significantly, consuming disk space and potentially affecting system performance. Implementing log rotation, through either external tools or custom scripts, can help maintain log file sizes and prevent potential issues. Effective log output management not only makes application debugging and monitoring tasks more manageable, but also ensures that logging practices align with the performance and maintenance requirements of your application.

Practical Examples of Using the Unified Logging System

To demonstrate the versatility of the unified logging system, let's start by understanding how to configure it. A comprehensive grasp of this system is crucial for Java developers, particularly for efficient debugging and performance optimization. The `-Xlog` option, a central component of this system, offers extensive flexibility to tailor logging to specific needs. Let's explore its usage through various scenarios:

- **Configuring the logging system using the `-Xlog` option:** The `-Xlog` option is a powerful tool used to configure the logging system. Its flexible syntax, `-Xlog:tags:output:decorators:level`, allows you to customize what is logged, where it is logged, and how it is presented.
- **Enabling logging for specific tags:** If you want to enable logging for the `gc` and `compiler` tags at the `info` level, you can do so with the following command:

  ```
  $ java -Xlog:gc,compiler:info MyApp
  ```

- **Specifying different levels for different tags:** It's also possible to specify different levels for different tags. For instance, if you want `gc` logs at the `info` level, but `compiler` logs at the `warning` level, you can use this command:

  ```
  $ java -Xlog:gc=info,compiler=warning MyApp
  ```

- **Logging to different outputs:** By default, logs are written to *stdout*. However, you can specify a different output, such as a file. For example:

  ```
  $ java -Xlog:gc:file=gc.log MyApp
  ```

  This will write all `gc` logs to the file `gc.log`.

- **Using decorators to include additional information:** You can use decorators to include more context in your log messages. For instance, to include timestamps with your `gc` logs, you can use this command:

  ```
  $ java -Xlog:gc*:file=gc.log:time MyApp
  ```

  This will write `gc` logs to `gc.log`, with each log message being prefixed with the timestamp.

- **Enabling all logs:** For comprehensive logging, which is particularly useful during debugging or detailed analysis, you can enable all log messages across all levels with the following command:

  ```
  $ java -Xlog:all=trace MyApp
  ```

  This command ensures that every possible log message from the JVM, from the `trace` to `error` level, is captured. While this provides a complete picture of JVM operations, it can result in a large volume of log data, so it should be used judiciously.

- **Disabling logging:** If you want to turn off logging, you can do so with the following command:

  ```
  $ java -Xlog:off MyApp
  ```

Benchmarking and Performance Testing

The unified logging system is crucial in benchmarking and performance testing scenarios. By thoroughly analyzing the logs, developers can understand how their applications perform under different workloads and configurations. This information can be used to fine-tune those applications for optimal performance.

For instance, the garbage collection logs can provide insights into an application's memory usage patterns, which can guide decisions on adjusting heap size and fine-tuning other GC parameters for efficiency. In addition, other types of logs, such as JIT compilation logs, thread activity logs, or the `monitorinflation` logs discussed earlier can contribute valuable information. JIT compilation logs, for example, can help identify performance-critical methods,[3] whereas the `thread` and `monitorinflation` logs can be used to diagnose concurrency issues or inefficiencies in thread utilization.

In addition to utilizing them during performance testing, integrating these logs into regression testing frameworks is vital. It ensures that new changes or updates do not adversely affect application performance. By doing so, continuous integration pipelines can significantly streamline the process of identifying and addressing performance bottlenecks in a continuous delivery environment.

However, it's important to note that interpreting these logs often requires a nuanced understanding of JVM behavior. For instance, GC logs can be complex and necessitate a deep understanding of memory management in Java. Similarly, deciphering JIT compilation logs requires knowledge of how the JVM optimizes code execution. Balancing log analysis with other performance testing methodologies ensures a comprehensive evaluation of application performance.

Tools and Techniques

Analyzing JVM logs can be a complex endeavor due to the potential influx of data into such logs. However, the complexity of this task is significantly reduced with the aid of specialized tools and techniques. Log analysis tools are adept at parsing logs and presenting the data in a more manageable and digestible format. These tools can filter, search, and aggregate log data, making it easier to pinpoint relevant information. Similarly, visualization tools can help with pattern identification and detection of trends within the log data. They can turn textual log data into graphical representations, making it easier to spot anomalies or performance bottlenecks.

Moreover, based on historical data, machine learning techniques can predict future performance trends. This predictive analysis can be crucial in proactive system maintenance and optimization. However, effective use of these techniques requires a substantial body of historical data and a foundational understanding of data science principles.

These tools are not only instrumental in problem identification but also play a role in operational responses, such as triggering alerts or automated actions based on log analysis results. We will learn about performance techniques in Chapter 5, "End-to-End Java Performance Optimization: Engineering Techniques and Micro-benchmarking with JMH."

[3] We covered performance-critical methods in Chapter 1: "The Performance Evolution of Java: The Language and the Virtual Machine."

Optimizing and Managing the Unified Logging System

Although the unified logging system is an indispensable tool for diagnostics and monitoring, its use demands careful consideration, particularly in regard to applications' performance. Excessive logging can lead to performance degradation, as the system spends additional resources writing to the log files. Moreover, extensive logging can consume significant amounts of disk space, which could be a concern in environments with limited storage. To help manage disk space usage, log files can be compressed, rotated, or moved to secondary storage as necessary.

Thus, balancing the need for detailed logging information and the potential performance impacts is crucial. One approach to do so is to adjust the logging level to include only the most important messages in production environments. For instance, logging some of the messages at the `error` and `warning` levels might be more appropriate than logging all messages at the `info`, `debug`, or `trace` level in high-traffic scenarios.

Another consideration is the output destination for the log messages. During the application's development stages, it's recommended that log messages be directly output on *stdout*—which is why it's the default for the logging system. However, writing log messages to a file might be more suitable in a production environment, so that these messages can be archived and analyzed later as needed.

One advanced feature of the unified logging system is its ability to adjust logging levels at runtime dynamically. This capability is particularly beneficial for troubleshooting issues in live applications. For example, you might typically run your application with standard logging to maximize performance. However, during traffic surges or when you observe an increase in specific patterns, you can temporarily change the logging tag levels to gather more relevant information. Once the issue is resolved and normalcy is restored, you can revert to the original logging level without restarting the application.

This dynamic logging feature is facilitated by the `jcmd` utility, which allows you to send commands to a running JVM. For instance, to increase the logging level for the `gc*` tag to `trace`, and to add some useful decorators, you can use the following command:

```
$ jcmd <pid> VM.log what=gc*=trace decorators=uptimemillis,tid,hostname
```

Here, `<pid>` is the process ID of the running JVM. After executing this command, the JVM will start logging all garbage collection activities at the `trace` level, including the uptime in milliseconds, thread ID, and hostname decorators. This provides us with comprehensive diagnostic information.

To explore the full range of configurable options, you can use the following command:

```
$ jcmd <pid> help VM.log
```

Although the unified logging system provides a wealth of information, it's essential to use it wisely. You can effectively manage the trade-off between logging detail and system performance by adjusting the logging level, choosing an appropriate output, and leveraging the dynamic logging feature.

Asynchronous Logging and the Unified Logging System

The evolution of Java's unified logging system has resulted in another notable advanced (optional) feature: asynchronous logging. Unlike traditional synchronous logging, which (under certain conditions) can become a bottleneck in application performance, asynchronous logging delegates log-writing tasks to a separate thread. Log entries are placed in a staging buffer before being transferred to the final output. This method minimizes the impacts of logging on the main application threads—a consideration that is crucial for modern, large-scale, and latency-sensitive applications.

Benefits of Asynchronous Logging

- **Reduced latency:** By offloading logging activities, asynchronous logging significantly lowers the latency in application processing, ensuring that key operations aren't delayed by log writing.
- **Improved throughput:** In I/O-intensive environments, asynchronous logging enhances application throughput by handling log activities in parallel, freeing up the main application to handle more requests.
- **Consistency under load:** One of the most significant advantages of asynchronous logging is its ability to maintain consistent performance, even under substantial application loads. It effectively mitigates the risk of logging-induced bottlenecks, ensuring that performance remains stable and predictable.

Implementing Asynchronous Logging in Java

Using Existing Libraries and Frameworks

Java offers several robust logging frameworks equipped with asynchronous capabilities, such as Log4j2 and Logback:

- **Log4j2**[4]**:** Log4j2's asynchronous logger leverages the LMAX Disruptor, a high-performance interthread messaging library, to offer low-latency and high-throughput logging. Its configuration involves specifying *AsyncLogger* in the configuration file, which then directs the logging events to a ring buffer, reducing the logging overhead.
- **Logback**[5]**:** In Logback, asynchronous logging can be enabled through the *AsyncAppender* wrapper, which buffers log events and dispatches them to the designated appender. Configuration entails wrapping an existing appender with *AsyncAppender* and tweaking the buffer size and other parameters to balance performance and resource use.

In both frameworks, configuring the asynchronous logger typically involves XML or JSON configuration files, where various parameters such as buffer size, blocking behavior, and underlying appenders can be defined.

[4] https://logging.apache.org/log4j/2.x/manual/async.html

[5] https://logback.qos.ch/manual/appenders.html#AsyncAppender