Chapter 13: Java Modules and the JPMS (Java Platform Module System)

Introduction

The modularization of Java applications was introduced with **Java 9** through the **Java Platform Module System (JPMS)**. It was one of the most significant changes to the Java language since generics in Java 5. JPMS offers a powerful mechanism to divide code into **reliable, reusable, and secure components** known as *modules*. These modules define what they expose to other modules and what they encapsulate. JPMS aims to make applications more **scalable, maintainable, and performant** by organizing code into explicit, well-defined modules.

This chapter explores Java Modules, their structure, configuration, usage, benefits, and their role in building robust enterprise applications.

13.1 What is a Module in Java?

A **module** is a self-contained unit of code that groups together related packages and resources. It specifies:

- **Which packages it exports**
- **Which other modules it requires**

Characteristics of a Java Module:

- It has a name.
- It explicitly states dependencies on other modules.
- It encapsulates its internal packages.

13.2 Why JPMS Was Introduced

Before Java 9:

- Java used JAR files to package and distribute code.
- There was no true module system; dependency conflicts (e.g., “JAR Hell”) were common.

- No reliable way to hide internal APIs or detect conflicts between libraries.

JPMS Solves:

- Reliable configuration
- Strong encapsulation
- Scalable platform (small runtime for IoT)
- Better security and maintainability

13.3 Structure of a Module

Each module has a **`module-info.java`** file at its root which acts as a **module descriptor**.

Syntax:

```
javaCopy codemodule com.example.mylibrary {
    requires java.sql;
    exports com.example.mylibrary.api;
}
```

Keywords:

- `module`: declares the module name.
- `requires`: declares dependency on another module.
- `exports`: makes a package accessible to other modules.

13.4 Components of Module System

1. **Module Declaration (`module-info.java`)**

Defines the module and its dependencies.

Example:

```
javaCopy codemodule com.myapp {
    requires java.logging;
    requires com.utils;
    exports com.myapp.api;
}
```

2. `requires` **Directive**

Tells the compiler and runtime that a module depends on another module.

Types:

- `requires`: Compile-time and runtime dependency.
- `requires transitive`: Exposes the dependency to modules that depend on your module.
- `requires static`: Used only at compile time.

3. `exports` Directive

Defines which packages are available to other modules.

```
javaCopy codeexports com.myapp.api;
```

4. `opens` Directive

Opens a package for reflection at runtime (important for frameworks like Spring).

```
javaCopy codeopens com.myapp.internal;
```

5. `uses` and `provides` Directives

Used for **ServiceLoader-based dependency injection**.

```
javaCopy codeuses com.myapp.MyService;

provides com.myapp.MyService with com.myapp.impl.MyServiceImpl;
```

13.5 Types of Modules

Type	Description
Application Module	Modules you write for your application.
Automatic Module	A regular JAR placed on the module path without a `module-info.java`.
Unnamed Module	Classes loaded from the classpath, not explicitly modularized.
Platform Modules	Built-in Java modules (e.g., `java.base`, `java.sql`, `java.xml`).

13.6 Java Platform Modules

Java itself is modularized. Some standard modules include:

- `java.base`: Contains essential classes (automatically required).
- `java.sql`: JDBC API.
- `java.logging`: Java Logging API.
- `java.xml`: XML processing.

You can list all modules via:

```
bashCopy codejava --list-modules
```

13.7 Creating and Using Modules – Example

Folder Structure:

```
arduinoCopy codesrc/
├── com.myapp/
│   ├── module-info.java
│   └── com/myapp/Main.java
├── com.utils/
│   ├── module-info.java
│   └── com/utils/Utils.java
│
```

Sample `module-info.java` for com.myapp:

```
javaCopy codemodule com.myapp {
    requires com.utils;
    exports com.myapp;
}
```

Sample `module-info.java` for com.utils:

```
javaCopy codemodule com.utils {
    exports com.utils;
}
```

Compile:

```
bashCopy codejavac -d out --module-source-path src $(find src -name "*.java")
```

Run:

```
bashCopy codejava --module-path out -m com.myapp/com.myapp.Main
```

13.8 Benefits of JPMS

- **Reliable Configuration**: No more JAR conflicts or classpath issues.

- **Strong Encapsulation**: Prevents unwanted access to internal APIs.
- **Improved Performance**: JVM can optimize startup and memory use.
- **Security**: Explicit access control reduces attack surfaces.
- **Maintainability**: Clear boundaries and dependencies.

13.9 Limitations and Challenges

- Steep learning curve for legacy developers.
- Compatibility issues with non-modular libraries.
- Frameworks like Spring require opens for reflection-based features.
- Not all existing tools and libraries support modules perfectly.

13.10 JPMS vs OSGi

Feature	JPMS	OSGi
Runtime System	Static at compile time	Dynamic at runtime
Complexity	Simpler	Complex
Adoption	Higher (post-Java 9)	Limited to niche areas
Focus	Compile-time modularity	Runtime component model

13.11 Migration from Non-Modular to Modular Code

Steps:

1. Identify and isolate modules in your codebase.
2. Create `module-info.java` for each module.
3. Move third-party libraries to the module path.
4. Use `requires`, `exports`, opens as needed.
5. Refactor reflective access with opens.

Summary

In this chapter, we explored **Java Modules and the Java Platform Module System (JPMS)**, a key feature introduced in Java 9 to enhance modularity, encapsulation, and maintainability. JPMS allows developers to break monolithic applications into **well-defined modules** with controlled dependencies and clear boundaries. We learned about the **structure of modules**, the `module-info.java` descriptor, **directives like `requires`, `exports`, and `opens`**, and how to create and use modules in practice. Despite some **challenges and migration concerns**, JPMS provides a solid foundation for scalable and secure Java applications.
