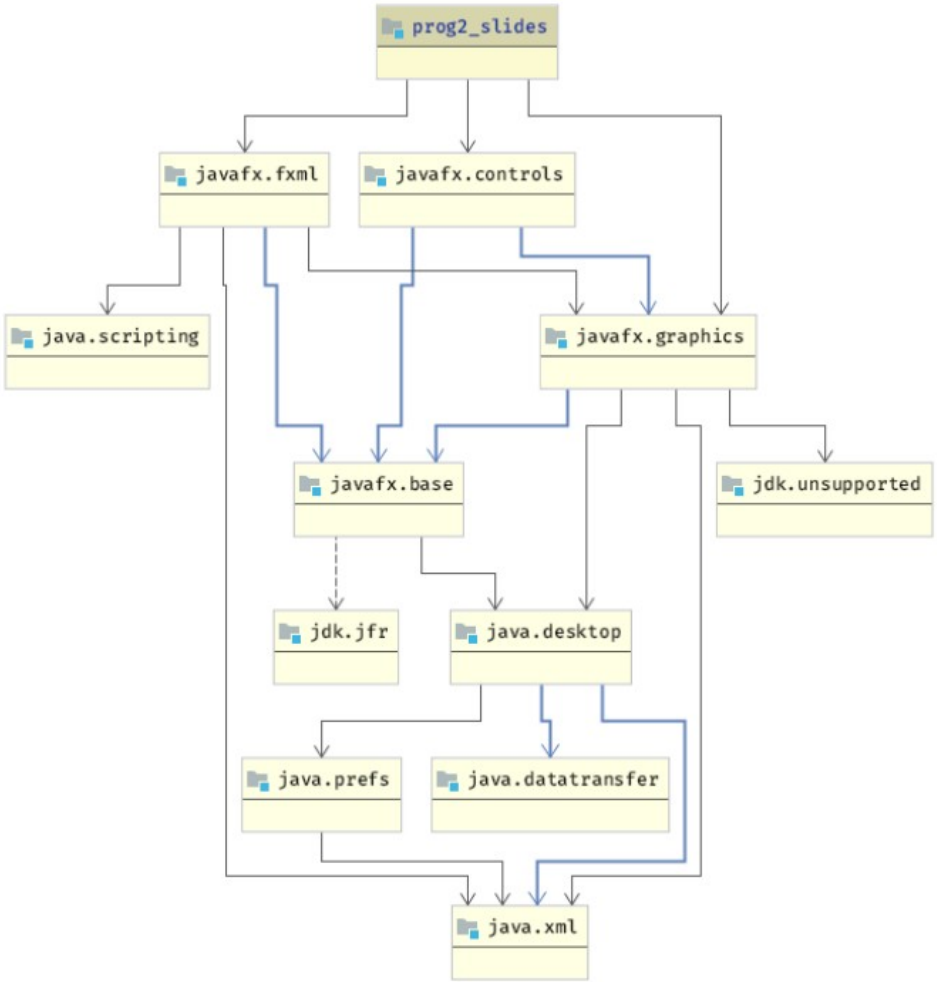
Some notes to Java9 Modules

10.2.5. Modules: Dependencies

- These dependencies between modules can also be visualized, e.g. within IntelliJ
  - Regular shaped lines represent a dependency on a module: all types can read the exported types of the required module
  - Blue lines represent an *implied readability*: all modules depending on the module with a transitively required module will also be able to read contents of that module



1. prog2_slides

This module depends on javafx.fxml, javafx.controls, and indirectly on javafx.base, java.desktop, and others.

```
module prog2_slides {
    requires javafx.fxml;
    requires javafx.controls;
}
```

---

2. javafx.fxml

This module depends on javafx.base.

```
module javafx.fxml {
    requires transitive javafx.base;
    exports javafx.fxml;
}
```

---

3. javafx.controls

This module depends on javafx.base, javafx.graphics, and indirectly on java.desktop.

```
module javafx.controls {
    requires transitive javafx.base;
    requires transitive javafx.graphics;
    exports javafx.controls;
}
```

---

4. javafx.graphics

This module depends on javafx.base, java.desktop, and jdk.unsupported.

```
module javafx.graphics {
    requires transitive javafx.base;
    requires java.desktop;
    requires jdk.unsupported;
    exports javafx.graphics;
}
```

---

5. javafx.base

This module depends on java.prefs and java.datatransfer.

```
module javafx.base {
    requires java.prefs;
    requires java.datatransfer;
    exports javafx.base;
}
```

---

6. java.desktop

This module depends on java.datatransfer.

```
module java.desktop {
    requires java.datatransfer;
    exports java.desktop;
}
```

---

7. java.prefs

This module does not have any direct dependencies in the diagram.

```
module java.prefs {
    exports java.prefs;
}
```

---

8. java.datatransfer

This module depends on java.xml.

```
module java.datatransfer {
    requires java.xml;
    exports java.datatransfer;
}
```

---

9. java.xml

This module does not have any dependencies in the diagram.

```
java
Code kopieren
module java.xml {
    exports java.xml;
}
```

Key Points:

1. **requires transitive**:
    - Used when a module wants to expose **its dependencies to its dependents**. For example, javafx.fxml declares requires transitive javafx.base, meaning any module requiring javafx.fxml will automatically require javafx.base.
2. **exports**:
    - Specifies the packages that are available to other modules.
3. **Dependencies**:
    - The blue lines in the diagram represent transitive dependencies, while the black lines represent direct dependencies.

Further restriction „to“

If you want to limit access to a specific package to only certain modules, use `exports ... to`.

Example:

```
module my.module {
    exports com.example.api to specific.module;
// Only specific.module can access this package
}
```

In this case:

- The `com.example.api` package will only be accessible to `specific.module`.
- Other modules won't be able to access it.

The unnamed module

Used when no module-info.java is provided

```
module my.unnamed.module {
   // Require all common Java platform modules
   requires transitive java.base; // Always required
   requires transitive java.logging;
   requires transitive java.sql;
   requires transitive java.xml;
   requires transitive java.desktop;
   requires transitive java.management;
   requires transitive java.naming;
   requires transitive java.net.http;
   requires transitive java.prefs;
   requires transitive java.scripting;
   requires transitive java.security.jgss;
   requires transitive java.instrument;

   // Export all packages (replace with actual package names in your application)
   exports com.example.package1;
   exports com.example.package2;

   // Open all packages for reflection (replace with actual package names in your
application)
   opens com.example.package1;
   opens com.example.package2;
}
```

Key Differences Between `exports` and `opens`

Directive	Purpose	Access Level
`exports`	Makes the package accessible to other modules	Compile-time and runtime (no reflection)
`opens`	Makes the package accessible for reflection only	Runtime (reflective access to private/protected members)
Both	Used together for combined functionality	Compile-time, runtime, and reflective access

```
module my.module {
    opens com.example.myapp;
}
```

```
package com.example.reflect;

import com.example.myapp.MyClass;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class ReflectExample {
    public static void main(String[] args) throws Exception {
        MyClass obj = new MyClass();

        // Access private field
        Field secretField = MyClass.class.getDeclaredField("secret");
        secretField.setAccessible(true); // Requires the package to be opened
        System.out.println("Secret: " + secretField.get(obj));

        // Access private method
        Method getSecretMethod = MyClass.class.getDeclaredMethod("getSecret");
        getSecretMethod.setAccessible(true); // Requires the package to be opened
        System.out.println("Method Output: " + getSecretMethod.invoke(obj));
    }
}
```

Behavior Without module-info.java (Unnamed Module):

- All packages in your code are **implicitly exported** and available to other classes and libraries.
- This is how Java worked prior to Java 9 Platform Module System(JPMS) (Java 8 and earlier).

Behavior With module-info.java:

- **Encapsulation by Default**:
  - No packages are exported unless explicitly declared in module-info.java.
  - This means other modules cannot access your packages unless you explicitly export them.

How to Export Packages in module-info.java

You need to explicitly declare which packages to export using the exports directive.

Example

```
module my.module {
    exports com.example.mypackage; // Only this package is accessible to other modules.
}
```

Key Points:

- Only the com.example.mypackage is accessible to other modules.
- Other packages in the module remain encapsulated and cannot be accessed by other modules.

Opening Packages for Reflection

If you are using frameworks (e.g., Spring, Hibernate) that rely on reflection to access your classes, you also need to **open** the relevant packages.

Example

```
module my.module {
    exports com.example.mypackage; // Accessible at compile-time and runtime.
    opens com.example.mypackage;  // Accessible via reflection at runtime.
}
```

Key Points:

- exports: Makes the package accessible at compile-time and runtime.

- opens: Allows runtime reflection but does not make the package accessible at compile-time.

Practical Example

If you have the following structure:

csharp
Code kopieren

```
src
└── com
    └── example
        ├── publicapi
        │   └── MyPublicClass.java
        └── internal
            └── MyInternalClass.java
```

You can define your module-info.java as:

```
module my.module {
    exports com.example.publicapi; // This is part of the module's public API.
}
```

***A good overview is found in* https://github.com/goxr3plus/java9-modules-tutorial**

More modifiers

Comparison

Directive	Scope	Purpose	Usage
`provides`	Specific Service	Declares a module provides a service.	Used for service implementation.
`uses`	Specific Service	Declares a module consumes a service.	Used for service discovery.
`open`	Entire Module	Allows reflection access for all packages.	Used for reflective applications.

1. `provides`

- **Purpose**: Declares that the module provides a specific implementation of a service.
- **Used with**: The **Service Provider Interface (SPI)** mechanism.
- **Syntax**:

```
provides <ServiceInterface> with <ServiceImplementation>;
```

- **Example**:

```
module com.example.provider {
    provides com.example.MyService with com.example.MyServiceImpl;
}
```

- **Explanation**:
  - Indicates that the module includes a concrete implementation of the `MyService` interface.
  - Enables **dynamic service discovery** using `ServiceLoader`.

2. `uses`

- **Purpose**: Declares that the module depends on a service but does not know which implementation will be provided.
- **Used with**: The **Service Provider Interface (SPI)** mechanism.
- **Syntax**:

```
uses <ServiceInterface>;
```

- **Example**:

```
java
Code kopieren
module com.example.consumer {
    uses com.example.MyService;}
```

- **Explanation**:
  - Indicates that the module consumes the `MyService` service.
  - The actual implementation is discovered at runtime using `ServiceLoader`.

3. open

- **Purpose**: Declares that the entire module is "open" for runtime reflection.
- **Used for**: Applications or frameworks that need reflection access to all packages in the module.
- **Syntax**:

```
open module <module_name> {
    ...
}
```

- **Example**:

```
open module com.example {
    requires some.framework;
}
```

- **Explanation**:
  - Makes all packages in the module accessible for reflection.
  - Useful when migrating non-modular applications to JPMS that rely on reflection.

Together:

```
module com.example.consumer {
    uses com.example.MyService;
}
```

- Then, the `ServiceLoader` can be used to dynamically discover implementations:

```
ServiceLoader<MyService> loader = ServiceLoader.load(MyService.class);
for (MyService service : loader) {
    service.performTask(); // Use the service
}
```

```
module com.example.provider {
    provides com.example.MyService with com.example.MyServiceImpl;
    exports com.example; // Makes `MyServiceImpl` directly
accessible (not recommended for encapsulation).
}
```

Repetition – Access Modifiers

Access Modifier Summary Table

Modifier	Same Class	Same Package	Subclass (Other Package)	Other Package
`public`	✅	✅	✅	✅
`protected`	✅	✅	✅ (via inheritance)	❌
`default`	✅	✅	❌	❌
`private`	✅	❌	❌	❌

Further considerations

Using Java's module system (JPMS) does allow you to manage dependencies more explicitly by leveraging module declarations and `module-info.java` files. However, getting rid of Maven entirely depends on how you want to manage external dependencies and build processes. Let's break this down:

What Maven Provides

1. **Dependency Management**:
    - Resolves and downloads external libraries (e.g., JAR files) from central repositories.
    - Manages transitive dependencies.
2. **Build Automation**:
    - Handles tasks like compiling, packaging, testing, and deployment.
3. **Standard Project Structure**:
    - Enforces conventions like `src/main/java` and `src/test/java`.

What JPMS Provides

- **Explicit Dependencies**:
    - The `requires` directive in `module-info.java` explicitly declares module dependencies.
- **Encapsulation**:
    - You control what is accessible to other modules using `exports` and `opens`.
- **Reduced Classpath Issues**:
    - Modules help reduce "classpath hell" by organizing code into logical units.

However, **JPMS itself does not manage external dependencies**, which is where Maven (or Gradle, or similar tools) still plays a crucial role.

When You Can Get Rid of Maven

1. **You Have No External Dependencies**:

- If your project only uses the Java standard library and your own modules, you can use JPMS alone.
- Compile and run modules using the `javac` and `java` commands directly.

Example Build and Run with JPMS:

```
javac -d out --module-source-path src $(find src -name "*.java")
java --module my.module/com.example.Main
```

2. **You Manage JAR Files Manually**:
    - If you manually download JAR files (e.g., from Maven Central) and organize them in a local `lib/` folder, you can use the `--module-path` option to include them.

    Example:

    ```
    javac -d out --module-path lib --module-source-path src $(find src -name
    "*.java")
    java --module-path lib:out --module my.module/com.example.Main
    ```

    This approach works but can quickly become unwieldy for larger projects with many dependencies or transitive dependencies.

When You Still Need Maven

1. **For Managing External Libraries**:
    - If your project depends on libraries like Apache Commons, Jackson, or Hibernate, Maven automates their download and resolution, especially for transitive dependencies.
2. **For Build Automation**:
    - Maven integrates with testing frameworks (e.g., JUnit), packaging tools, and deployment workflows.
3. **For Multi-Module Projects**:
    - Maven simplifies building and linking multiple modules with proper dependency resolution.
4. **For Maintaining Compatibility with Other Tools**:
    - Many IDEs and CI/CD systems (e.g., Jenkins) work seamlessly with Maven projects.

How JPMS and Maven Work Together

JPMS and Maven complement each other:

- JPMS defines modular boundaries, encapsulation, and dependencies at runtime.
- Maven automates dependency resolution, fetching libraries, and project builds.

Using JPMS with Maven

Maven can work with JPMS projects by:

1. Including `module-info.java` in your source files.
2. Using the `<module>` tag in the POM for module naming.

Example POM for JPMS Project:

```
xml
Code kopieren
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>my.module</artifactId>
    <version>1.0</version>

    <dependencies>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.12.0</version>
        </dependency>
    </dependencies>
</project>
```

Key Differences Between JPMS Alone and Maven

Feature	JPMS Alone	Maven with JPMS
Dependency Management	Manual (download and organize JARs)	Automated (resolves dependencies)
Build Automation	Manual (write scripts)	Automated (standardized process)
Transitive Dependencies	Manual (add all dependencies)	Automatic
IDE Integration	Limited	Seamless

Conclusion

- If your project is simple and does not require external libraries, you can use **JPMS alone** without Maven.
- For anything more complex (e.g., external libraries, testing frameworks, CI/CD), Maven remains invaluable.

Even with JPMS, Maven or a similar tool like Gradle is still the preferred way to manage larger projects effectively. You can think of Maven as a complement to JPMS, not a replacement.