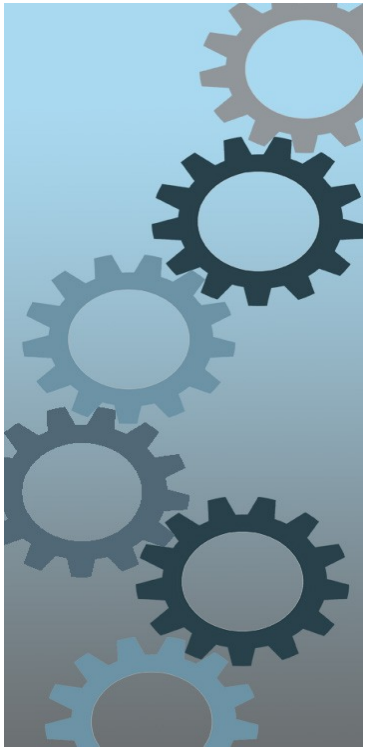
HSLU Hochschule Luzern

Programming Concepts And Paradigms

Modular Programs
Packages and Modules



Marcel Baumann

CC BY

Content

- Module History
- Module Definitions
  - Types
  - Directives
  - Rules
  - Example
- Building and Migrating Applications

Why Modules?

- Modular applications
- Modular JVM and Custom Java Builds
- Better Access Control Features (*and unique package enforcement)*
- Clearer Dependency Management and improved performance
- Services instead of Singletons

Dependency Hell

- Conflicting dependencies
- Diamond dependencies
- Long chain of dependencies
- *Birth of Maven central repository*
- *Pain of laggard maintainers and associated security risks*

Jar Approaches (1/2)

- Packages and classes have visibility
  - public
  - protected
  - package private
  - private

Jar Approaches (2/2)

- Jar have no clear rules
  - Same package can be defined in multiple jar files
  - Packages are top-level containers, they do not have a hierarchy
  - Through reflection you can access any private class, method and instance variable

Module Approaches (1/2)

- Only exported packages are visible outside the module
- It is prohibited to declare a package in multiple modules
- Per default reflection access is disabled
- *A soft migration path is supported (and delaying the adoption of modules)*

Module Approaches (2/2)

- Versioning is not part of the module concept
- Module concept has killed OSGI
  - In 99.999% of all business cases (*and Docker killed the last 0.001%)*
- Modules have paved the way to smaller images and faster applications

Java Modules

When we create a module, we include a descriptor file that defines several aspects of our new module:

- **Name** – the name of our module
- **Dependencies** – a list of other modules that this module depends on
- **Public Packages** – a list of all packages we want accessible from outside the module
- ***Services Offered*** – we can provide service implementations that can be consumed by other modules
- ***Services Consumed*** – allows the current module to be a consumer of a service
- ***Reflection Permissions*** – explicitly allows other classes to use reflection to access the private members of a package

Module Types

- **System Modules –** These are the modules listed when we run the list-modules command *java --list-modules*. They include the Java SE and JDK modules.
- **Application Modules –** These modules are what we usually want to build when we decide to use Modules. They are **named** and defined in the compiled module-info.class file included in the assembled JAR.
- **Automatic Modules –** We can include unofficial modules by adding existing JAR files to the module path. The name of the module will be derived from the name of the JAR. Automatic modules will have full read access to every other module loaded by the path.
- **Unnamed Module –** When a class or JAR is loaded onto the classpath, but not the module path, it's automatically added to the unnamed module. It's a catch-all module to maintain backward compatibility with previously-written Java code.

Control Questions

1. What are the advantages of a Java modules?
2. What is an unnamed module?
3. What is an automatic module?
4. What is a named module?

Requires, Requires Static, transitive

- Describes your dependencies to requested modules
- Support your clients by providing transitive dependencies
- Requires static only needed during compilation but not during runtime
  - e.g. some annotations

Exports, exports to

- Only exported packages are visible
- The solution to the *Unsafe* design problem
- A concept similar to *friend* in C++
- Language support for domain driven design approach
  - The exported components are your domain interface.

Provides and Uses

- Service provider pattern through module
- Support multiple implementation of a service
- Identifies all clients of a service

Open, opens, opens to

- A cleaner support of reflection
  - Open whole module is for legacy
  - Opens *packageA* is for limited access
  - Opens *packageA* to *moduleX* is for security

Directives Summary

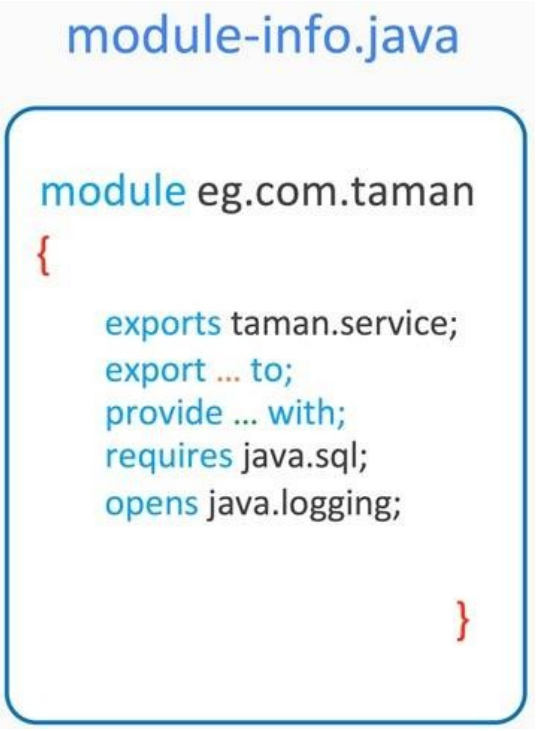
Derivative	Expression
export <package>	Allows all modules to access the package
export <package> to <module>	Allows a specific module to access the package
requires <module>	Indicates module is dependent on another module
requires transitive <module>	Indicates the module is dependent on another module and all modules this module uses
uses <interface>	Indicates that a module uses a service
provides <interface> with <class>	Indicates that a module provides an implementation of a service
open module {}	Opens whole module for reflection *(do not use)*
opens <package>	Opens package for reflection
opens <package> to <module>	Opens package for reflection through module

Control Questions

1. What does a requires directive?
2. What does a exports directive?
3. How can you limit access to exported packages?
4. When should you use requires transitive directive?

module-info.java

- No cyclic dependencies
- At root of your package hierarchy
- Compile module options
  - Module path **–module-path, -p**
- Find information with jdeps



Example Simple

```
module net.tangly.fsm {
    exports net.tangly.fsm;
    exports net.tangly.fsm.dsl;
    exports net.tangly.fsm.utilities;

    requires org.slf4j;
    requires static transitive org.jetbrains.annotations;
}
```

Example Service

```
module ch.hslu.service {
    exports ch.hslu.services;
    opens ch.hslu.services.factory
            to Logger;
}
```

```
module ch.hslu.provider {
    requires ch.hslu.service;
    provides ch.hslu.services.Service
            with ch.hslu.provider.ServiceImpl;
}
```

```
module ch.hslu.consumer {
    requires ch.hslu.service;
    requires ch.hslu.locator;
}
```

```
module ch.hslu.locator {
    export ch.hslu.locators to ch.hslu.consumer
    requires ch.hslu.service;
    uses ch.hslu.services.Service;
}
```

PCP Service Example

```
module ch.hslu.pcp.services {
    exports ch.hslu.pcp.services;
}
```

```
package ch.hslu.pcp.services;

public record Person(String firstname, String lastname, String identifier) {}
```

```
package ch.hslu.pcp.services;

import java.util.Optional;

public interface Service {
    public boolean isResponsibleFor(String country);

    public Optional<Person> getPerson(String identifier);
}
```

PCP Service Impl Example

```
module ch.hslu.pcp.serviceSwiss {
    requires ch.hslu.pcp.services;
    provides ch.hslu.pcp.services.Service with
                ch.hslu.pcp.serviceSwiss.ServiceSwiss;
}
```

```
package ch.hslu.pcp.serviceSwiss;

import java.util.HashMap;import java.util.Locale;
import java.util.Map;import java.util.Optional;
import ch.hslu.pcp.services.Service;
import ch.hslu.pcp.services.Person;

public class ServiceSwiss implements Service {
    private static final String SWITZERLAND ="CH";
    private Map<String, Person> persons;

    public ServiceSwiss() {
        persons = new HashMap<>();
        persons.put("007", new Person("John", "Doe", "007"));
    }

    public boolean isResponsibleFor(String country) { return SWITZERLAND.equals(country);}

    public Optional<Person> getPerson(String identifier) { return
Optional.ofNullable(persons.get(identifier)); }
}
```

PCP Service Locator

```
module ch.hslu.pcp.locator {
    requires ch.hslu.pcp.services;
    uses ch.hslu.pcp.services.Service;
    exports ch.hslu.pcp.locator;
}
```

```
package ch.hslu.pcp.locator;

import java.util.Optional;
import java.util.ServiceLoader;
import ch.hslu.pcp.services.Service;

public class ServiceLocator {
    public static Optional<Service> find(String country) {
        ServiceLoader<Service> loader = ServiceLoader.load(Service.class);
        return loader.stream().map(ServiceLoader.Provider::get).filter(o ->

                        o.isResponsibleFor(country)).findAny();
    }
}
```

PCP Service Consumer Example

```
module ch.hslu.pcp.consumer {
    requires ch.hslu.pcp.services;
    requires ch.hslu.pcp.locator;
}
```

```
package ch.hslu.pcp.consumer;

import java.util.Optional;
import ch.hslu.pcp.locator.ServiceLocator;
import ch.hslu.pcp.services.Person;
import ch.hslu.pcp.services.Service;

public class ServiceConsumer {
    public Optional<Person> findSwissPerson(String identifier) {
        Optional<Service> service = ServiceLocator.find("CH");
        return (service.isPresent()) ? service.get().getPerson(identifier) :
        Optional.empty();
    }

    public static void main(String[] args) {
        var consumer = new ServiceConsumer();
        consumer.findSwissPerson("007").ifPresent(System.out::println);
    }
}
```

Build and Run Instructions

- javac -d serviceProviderInterfaceModule
  consumer/ch/hslu/pcp/consumer/*.java
  consumer/module-info.java
- jar -cvf mods/ch.hslu.pcp.consumer.jar -C consumer/ .
- java -p ./mods/ -m
  ch.hslu.pcp.consumer/ch.hslu.pcp.consumer.ServiceConsumer

Gradle File

```
Plugins {
    id 'java-library'
}



Dependencies {
    implementation project(':ch.hslu.pcp.locator')
    implementation project(':ch.hslu.pcp.services')

    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.7.0'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.7.0'

}
```

Automatic Modules

- Automatic modules exist due to legacy jar files
  - The ugly approach:
    - *let Java infer the module name*
  - The acceptable approach:
    - *define the module name in the manifest file MANIFEST.MF*
      - *Automatic-Module-Name: module-name*
        (please use a globally unique name schema, ideally the same approach as for package names)

Access Rules

- An **unnamed** module can access any jar file in class path and any module in module path.
- All packages of an **automatic** module are accessible from other modules. An automatic module **requires** all other modules. They can only access the module path.
- A **named** module have access to named and automatic modules with **requires** directive. They can only access the module path.

Prepare Migration

- Draw your dependency diagram for modules and packages
  - Extract the dependencies from your gradle or maven file
  - Define an acyclic graph – *meaning remove cycles*
  - Put all your jar file in the classpath

Migrating to Modules bottom-up

- Pick the lowest-level project that has not yet been migrated. (Remember the way we ordered them by dependencies in the previous section?)
- Add a **module-info.java** file to that project. Be sure to add any exports to expose any package used by higher-level JAR files. Also, add a requires directive for any modules it depends on.
- Move this newly migrated named module from the classpath to the module path.
- Ensure any projects that have not yet been migrated stay as unnamed modules on the classpath.
- Repeat with the next-lowest-level project until you are done.

Migrating to Modules top-down

- Place all projects on the module path.
- Pick the highest-level project that has not yet been migrated.
- Add a module-info file to that project to convert the automatic module into a named module. Again, remember to add any exports or requires directives. You can use the automatic module name of other modules when writing the requires directive since most of the projects on the module path do not have names yet.
- Repeat with the next-lowest-level project until you are done.

Legacy Systems

- Use tools such as maven plugin
  com.github.ben-manes.versions
- If you are still working in a legacy environment use tools such as ArchUnit to ensure some rules
- If you have dirty code use jdeps –jdk-internals <jar file>
- Start planning your migration to modules: It is “**a *when and not an if***” decision

Exercises

- Module definition of a jar file
- Creation of services with implementation and consumer