Java

Modules

Module

- Introduced in Java 9
- Modules give a way to describe the relationships and dependencies of the code of an application
- Modules let you control which parts of a module are accessible to other modules and which are not
- Modules are most helpful to large applications
  - To reduce the management complexity of large software
- Small programs also benefit from modules
  - Java API library has now been organized into modules

Module

- It is now possible to specify which parts of the API are required by your program and which are not
- This makes it possible to deploy programs with a smaller runtime footprint
  - Important when creating code for small devices, such as those intended to be part of the Internet of Things (IoT)
- JDK and the run-time system substantially upgraded to support modules
  - Several keywords, enhancements to javac, java, and other JDK tools, new tools and file formats

Module Basics

- Module is a grouping of packages and resources that can be collectively referred to by the module's name
- A module declaration specifies
  - The name of a module
  - Defines the relationship a module and its packages have to other modules
- Module declarations are program statements in a Java source file and are supported by several module related keywords
  - Context-sensitive restricted keywords are recognized as keywords only in the context of a module declaration

Module Basics

- A module declaration is contained in a file called **module-info.java**
  - This file is then compiled by javac into a class file and is known as its module descriptor
- **module-info.java** file must contain only a module definition, cannot contain other types of declarations
- A module declaration begins with keyword **module**

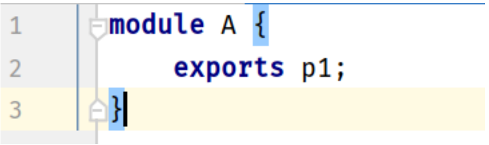
```
module moduleName {
    // module definition (maybe empty, typically not)
}
```

Modules Two Key Features

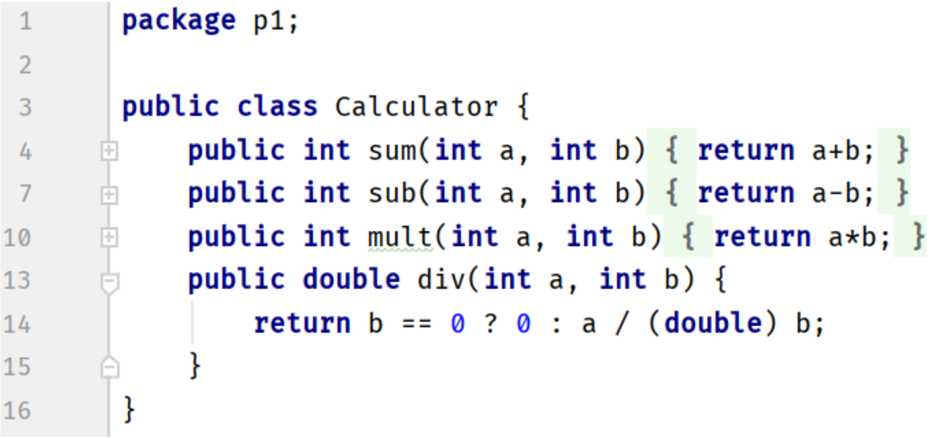
- The first is a module’s ability to specify that it requires another module
  - One module can specify that it depends on another
  - This is accomplished by use of the **requires** keyword
- The second is a module’s ability to control which, if any, of its packages are accessible by another module
  - This is accomplished by use of the **exports** keyword
  - The public and protected types within a package are accessible to other modules only if they are explicitly exported

Simple Module

module-info.java of module A



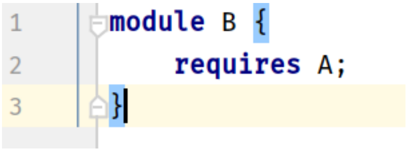
```java
module A {
    exports p1;
}
```



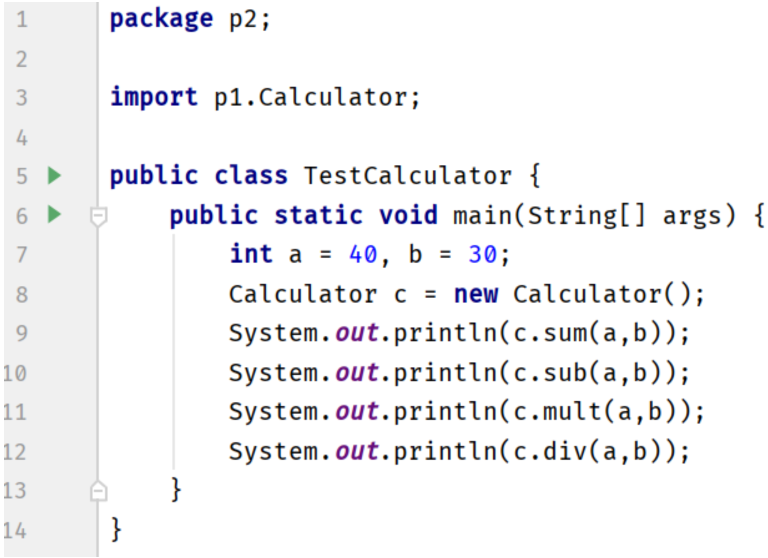
```java
package p1;

public class Calculator {
    public int sum(int a, int b) { return a+b; }
    public int sub(int a, int b) { return a-b; }
    public int mult(int a, int b) { return a*b; }
    public double div(int a, int b) {
        return b == 0 ? 0 : a / (double) b;
    }
}
```

module-info.java of module B



```java
module B {
    requires A;
}
```



```java
package p2;

import p1.Calculator;

public class TestCalculator {
    public static void main(String[] args) {
        int a = 40, b = 30;
        Calculator c = new Calculator();
        System.out.println(c.sum(a,b));
        System.out.println(c.sub(a,b));
        System.out.println(c.mult(a,b));
        System.out.println(c.div(a,b));
    }
}
```

Compile and Run the Module

Project: **JavaModulesSimple** (source code provided)

Go to moduleA\src and run:

```
javac  -d C:\module\A module-info.java p1\Calculator.java
```

Go to moduleB\src and run:

```
javac --module-path C:\module\  -d C:\module\B
module-info.java p2\TestCalculator.java
```

From anywhere run:

```
java --module-path C:\module\ --module B/p2.TestCalculator
```

Closer Look at requires and exports

- **requires** *moduleName*
- Here, *moduleName* specifies the name of a module that is required by the module
- The required module must be present in order for the current module to compile
- When more than one module is required, it must be specified in its own requires statement
- A module declaration may include several different requires statements

Closer Look at requires and exports

- **exports** *packageName*
- Here, packageName specifies the name of the package that is exported by the module in which this statement occurs
- A module can export as many packages as needed, with each one specified in a separate exports statement
- A module may have several exports statements

Closer Look at requires and exports

- When a module exports a package, it makes all of the public and protected types in the package accessible to other modules
  - Public and protected members of those types as well
- If a package within a module is not exported, it is private to that module including all of its public types
- The exports statement makes packages accessible to outside modules
  - Any non-exported package is only for the internal use of its module

Closer Look at requires and exports

- **requires** and **exports** work together
  - If one module depends on another, then it must specify that dependence with requires
  - The module on which another depends must explicitly export the packages that the dependent module needs
  - If either side of this dependence relationship is missing, the dependent module will not compile
- requires and exports statements must occur only within a module statement
- A module statement must occur by itself in a file called **module-info.java**

java.base and the Platform Modules

- Beginning with Java 9 the Java API packages have been incorporated into modules
  - API modules are referred to as platform modules, and their names all begin with the prefix java
  - java.base, java.desktop, java.xml
- By modularizing the API, it becomes possible to deploy an application with only the packages that it requires, rather than the entire Java Runtime Environment (JRE)
  - Very important improvement due to the size of the full JRE

java.base and the Platform Modules

- The most important platform module is java.base
  - It includes and exports those packages fundamental to Java, such as java.lang, java.io, and java.util, among many others
  - java.base is automatically accessible to all modules
  - All other modules automatically require java.base
  - There is no need to include a requires java.base statement in a module declaration
  - It is not wrong to explicitly specify java.base, it’s just not necessary
  - Similar to automatic import of java.lang

Legacy Code and Unnamed Module

- Unnamed module provides support for legacy code
  - When you use code that is not part of a named module, it automatically becomes part of the unnamed module
- Unnamed module has two important attributes
  - all of the packages in the unnamed module are automatically exported
  - unnamed module can access any and all other modules
- When program does not use modules, all API modules are automatically accessible through the unnamed module

Exporting to a Specific Module

- In an exports statement, the to clause specifies a list of one or more modules that have access to the exported package
  - only those modules named in the to clause will have access
  - the to clause creates what is known as a qualified export

exports packageName to moduleNames

- Here, moduleNames is a comma-separated list of modules to which the exporting module grants access

Using requires transitive

- Three modules, A, B, and C
  - B requires A and C requires B
  - C depends on B and B depends on A, C has an indirect dependence on A
- As long as C does not directly use any of the contents of A, the following is fine:

```
module C {
  requires B;
}
```

```
module B {
  exports p;
  requires A;
}
```

p is package exported by B and used by C.

Using requires transitive

- A problem occurs if C does want to access a type in A

Solution 1

```
module C {
   requires B;
   requires A;
}
```

if B will be used by many modules, you must add requires A to all module definitions that require B (tedious)

Solution 2

```
module B {
  exports p;
  requires transitive A;
}
```

You can create an implied dependence on A, any module that depends on B will also, automatically, depend on A. Thus, C would automatically have access to A (better)

Module jar files and jlink

Project: **JavaModules** (source code provided)

Step-1: Compile the modules

javac -d C:\module\A module-info.java p1\Calculator.java (run from moduleA\src)

javac --module-path C:\module\ -d C:\module\B module-info.java p2\TestCalculator.java (run from moduleB\src)

javac --module-path C:\module\ -d C:\module\C module-info.java p3\TestCalculator2.java (run from moduleC\src)

To execute run the following:

java --module-path C:\module\ --module C/p3.TestCalculator2

Module jar files and jlink

Step-2: Create module jar files

Go to C:\module and run:

```
mkdir libs
jar --create --file=libs\A.jar -C A .
jar --create --file=libs\B.jar -C B .
jar --create --file=libs\C.jar --main-class=p3.TestCalculator2 -C C .
```

To execute run the following:

```
java -p C:\module\libs --module C
```

Module jar files and jlink

Step-3: Use jlink to create a custom Java runtime image

- jlink is a tool that generates a custom Java runtime image that contains only the platform modules required for a given application
- Such a runtime image acts exactly like the JRE but contains only the modules we picked and the dependencies they need to function

```
jlink --module-path "%JAVA_HOME%"\jmods;C:\module\libs --add-modules C --output C:\myapp
```

To execute go to C:\myapp\bin and run the following:

```
java --module C/p3.TestCalculator2
```

Final Thoughts on Modules

- Modules are both a recent and significant addition to Java, it is likely that the module system will evolve over time
- Although their use is not required at this time, they offer important benefits for commercial applications that no Java programmer can afford to ignore
- It is likely that module-based development will be in every Java programmer’s future