# Going Further with Java Modules

## Overview

In this lab you’ll create a Java application that comprises two modules:

* A “library” module that provides useful functionality
* An “application” module that consumes this functionality

## Step 1: Creating the “library” module

In IntelliJ, create a new module named student.module.lib (for example). In the *src* folder, add a module-info.java file so that the module behaves as a proper JPMS module.

Create a package named io and add a class named ConsoleIO. The purpose of this class will be to encapsulate getting user input from the console. Define a public method named getNumber() that asks the user to enter number at the console (e.g., use java.util.Scanner and its nextDouble() method).

Create another package named math and add a class named Stats. In this class, define a public method named getResult() and implement it as follows:

* Call the ConsoleIO getNumber() method twice, to get two numbers from the user.
* Calculate the sum, difference, product, and quotient of the two numbers. Return these 4 values from the method (e.g., define a record type named Result to hold these 4 values).

Now modify the module-info.java file as follows:

* Export the math package from the module, so that it can be read (i.e., consumed) by other modules.
* Don’t export the io package. This package is for internal use only, within the current module. Other modules should be oblivious to the existence of the io package.

## Step 2: Creating the “application” module

In IntelliJ, create another new module named student.module.app (for example). In the *src* folder, add a module-info.java file so that this module too behaves as a proper JPMS module.

This module will utilize the functionality of the “library” module that you created just now. In order for IntelliJ to know how to find the “library” module, you must add it as a dependency as follows:

* Right-click the student.module.app module, and then click *Open Module Settings*.
* In the dialog box, in the right-hand panel, click the *Dependencies* tab.
* Click the + icon, then click *Module Dependency*. A dropdown list appears, showing all the IntelliJ modules available in your project. Select student.module.lib.

You can now crack on writing the code for your student.module.app module:

* Modify module-info.java to formalize the fact that your module “requires” the student.module.lib module. This will enable your module to access the packages exported by the student.module.lib module.
* Define a “main” class that invokes the library’s functionality to get mathematical stats. Display the results on the console.

Build and your “main” class. Verify that it all works smoothly. Take a look in the IntelliJ *Run* panel to see the command it executes to run your application.

## If time permits: Additional suggestions

Play devil’s advocate and see what happens if:

* The “library” module doesn’t indicate that it exports the math package
* The “application” module doesn’t indicate that it requires the “library” module
* The IntelliJ module settings for the “application” module doesn’t specify that the “library” module is a dependency

If you still have some spare time, have a go at defining and using JPMS services.

## Step 2: Using a JPMS module approach

You will now convert your IntelliJ module into a proper JPMS module. To do this, add a module-info.java file in the *src* folder.IntelliJ creates the file as follows (notice that the JPMS module name is the same as the IntelliJ module name):

module student.module.helloworld {

}

Congratulations, you now have a simple JPMS module!

Build and run the application again. Take a look in the IntelliJ *Run* panel; notice that IntelliJ now runs the application in modular fashion as follows (simplified):

java.exe

-p C:\ModernJavaDev\out\production\student.module.helloworld

-m solution.module.helloworld/main.Main

Note the -p and -m options here:

* -p is shorthand for --module-path. It specifies a semicolon-separated list of directories where modules are located. There’s only one directory mentioned in our case, because our simple application only comprises a single module.
* -m is shorthand for --module. It specifies which particular module/mainclass is the entrypoint to execute.

In other words, just the mere presence of the module-info.java file tells IntelliJ to run the application according to the JPMS modular approach. Going forward, all Java applications will eventually be like this!