Quiz yourself: Declaring and accessing Java modules

May 27, 2022 | 3 minute read

How do you provide access to a module's classes to code outside that module?

Imagine you are developing a date/time manipulation framework. The framework module is named `date.time.utils` and contains classes in the `com.acme.utils` package. The Java Date/Time API is located in the `java.time` package, which is in the `java.base` module. **Which of the following steps is mandatory to properly define your module for your clients?** Choose one.

A. Add the following to the `date.time.utils` module descriptor:
```
requires transitive java.base;
```

B. Add the following to the `date.time.utils` module descriptor:
```
requires java.base;
```
Also inform your clients to add the following in their modules:
```
requires java.base;
```

C. Add the following to the `date.time.utils` module descriptor:
```
exports date.time.utils;
```

D. Add the following to the `date.time.utils` module descriptor:
```
exports com.acme.utils;
```

Answer. The question provides the required information that the Java Date/Time API belongs to the `java.base` module, and you should already know that the `java.base` module is always *implicitly* required by any other module.
The question investigates how a module makes elements of itself available for other modules to use. It's important to know that elements declared in a module are effectively hidden inside that module unless they are deliberately and expressly made accessible to clients of the module.

Option A suggests adding a `requires transitive` dependency on the `java.base` module. This directive would declare that the `date.time.utils` module, and any module using it, requires `java.base`. However, `java.base` is always implicitly required for all modules. The `requires transitive` directive also states that clients of this module automatically have access to the exported features of the module required by this module. But again, since all modules have implicit access to `java.base`, this directive has no value in this case.
In addition to the lack of effect of the suggested directive, nothing in option A actually grants clients of this module the chance to use anything inside the module. Because of this, option A is incorrect.

Option B is very similar to option A, only differing in the absence of the `transitive` modifier and the admonition that users of this library should expressly add a

dependency on `java.base`. Since `java.base` is always implicitly required, both actions listed here are without effect. Further, neither action does anything to grant access to elements of the module to users, so option B is also incorrect.

Option C suggests adding an `exports` directive, but that directive appears to refer to a module name. The compiler treats what follows the `exports` keyword as a *package*. It is an error to try to export a package that does not exist, and because no package in this example has the same name as that module name, option C is incorrect.

Option D correctly describes the only required step for the described scenario—and, in general, for any module whose classes are expected to be used outside that module—which is to add one or more `exports` directives that specify a package that should be accessible to external clients of the module. The simplest module definition for this question's scenario will follow the guidance provided by option D and will look like the following:

```
module date.time.utils {
    exports com.acme.utils ;
}
```

Conclusion. The correct answer is option D.