7-10.モジュール

モジュールとは

モジュールとは、javaの複数のパッケージをまとめることが出来る機能です。
この機能は、Java9から導入されました。

Java8の公式リファレンス を見てみましょう。
Javaの標準APIとして、数多くのパッケージが存在していました。
コード記述の際、実行に必要なパッケージや、その配下のクラスをimportして読み込むということは行って来ましたね。
java8までは、APIの階層としてはパッケージが最上位でした。
(パッケージ→各クラスやインターフェース→各プロパティやメソッド)

Java11の公式リファレンス を見てみましょう。
新しく、JavaAPIの階層の最上位として、モジュールのカテゴリが出来ています。

試しに、`Java.base`モジュールをクリックして開いてみましょう。

中には、`java.io`や`java.lang`などのパッケージが格納されています。
このように、近い存在の各パッケージを再度グループ分けしている存在がモジュールです。

また、モジュールの中でもこの`Java.base`モジュールは特別な存在となるので、把握しておきましょう(後述)

JavaSilverにおいてモジュールの問題はそこまで多くなく、2〜3問程度です。
よく出題されるのは以下のような項目があり、最低限必要な点について解説します。

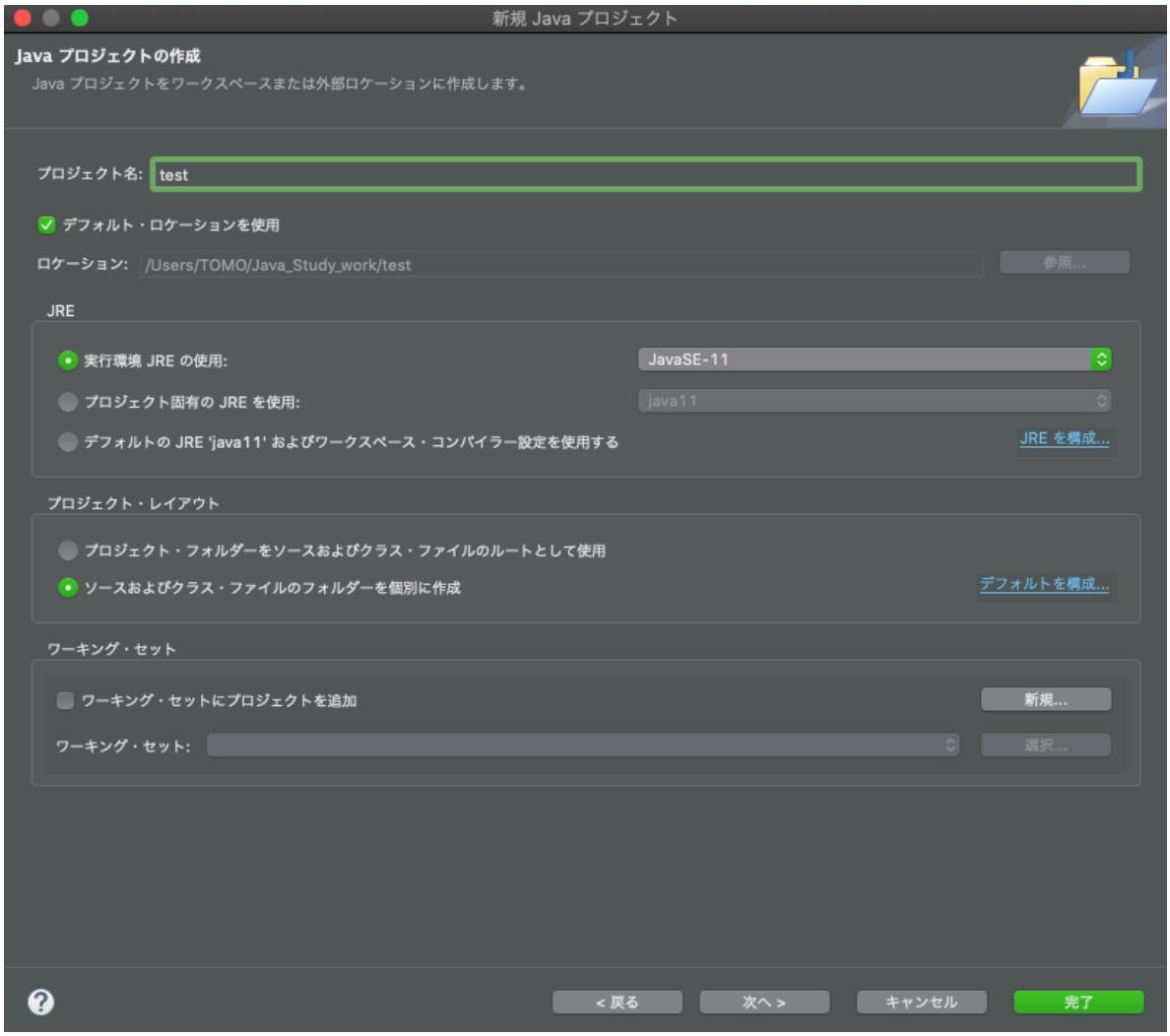
モジュール出題項目
* モジュールの仕様についての文章問題
* モジュールグラフ
* モジュールに関するjavaコマンド

モジュールの設定方法

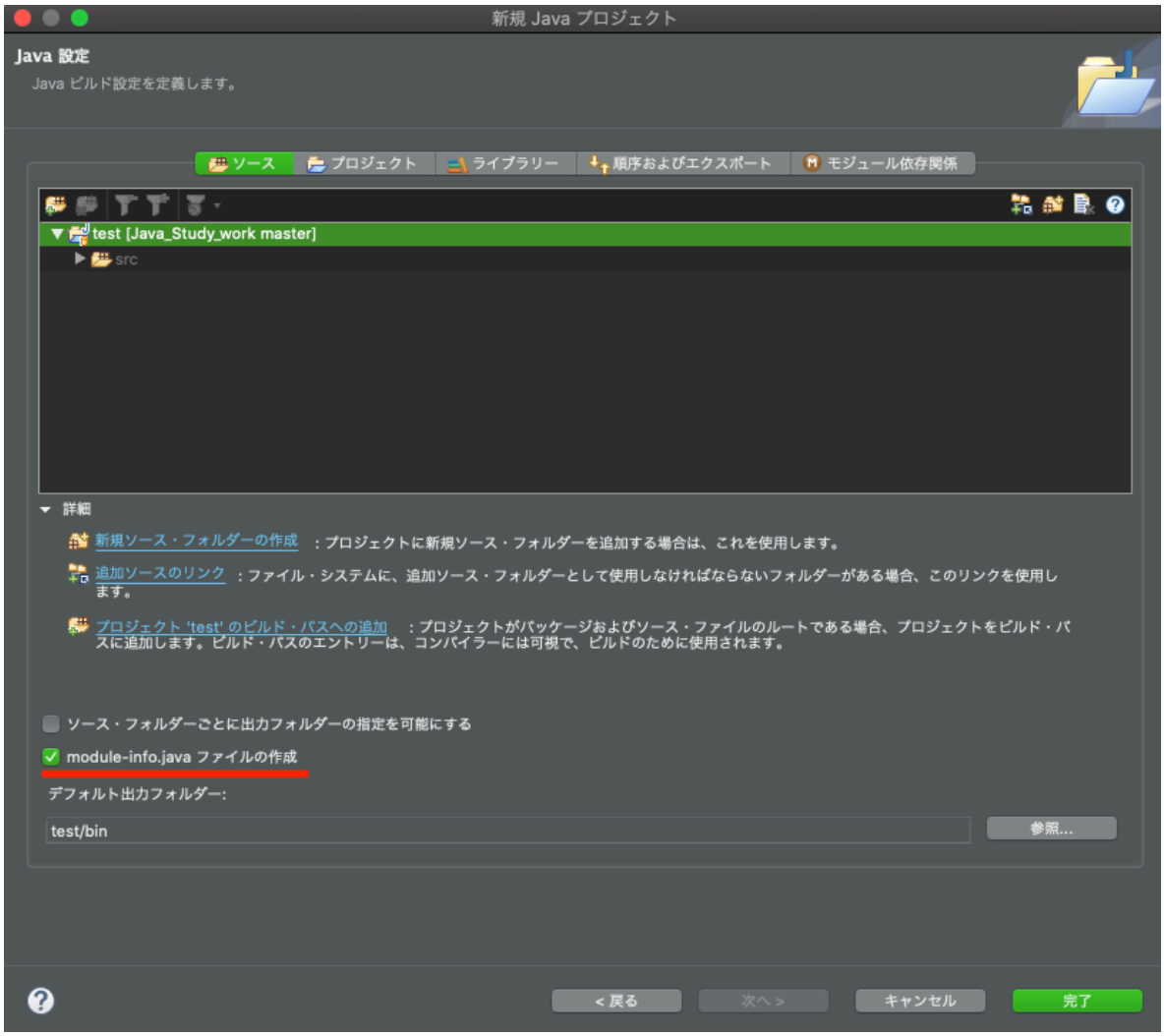
モジュールの設定は、`module-info.java`というファイル内で行います。

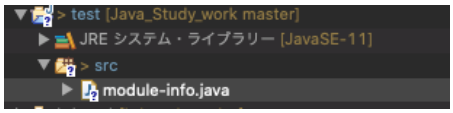
eclipseのプロジェクト一覧の空いているところで右クリック→Javaプロジェクトの作成を行って見ましょう。

?



「次へ」を押した先の設定画面で、`module-info.java`ファイルの作成というチェック欄があります。
ここにチェックを入れることで、プロジェクト作成時に自動的にファイルが作成されます。





`module-info.java`は、各プロジェクトごとに1つのみ使用します。
また、このファイルはプログラム実行時に、設定を読み込むようになっています。

モジュールグラフ

`module-info.java`の中身の記述方法について見ていきます。

主に覚えるべきキーワードは、下記の3つです。

- requires
- requires transitive
- exports

Javaには、アクセス修飾子というものがありましたね。
`public private protected` などで、メソッドや変数を公開する範囲の設定をしていました。
モジュールでは上記のキーワードなどを使用して、 **同じようにモジュールやパッケージを公開する範囲を設定する処理を行っています。**

requires

`requires`は、他のモジュールを読み込むためのキーワードです。

以下のコードは、4つのプロジェクトa,b,c,dがあり、それぞれの`module-info.java`の中身があると想定して下さい。

```
module a {
    requires b;
}
```

```
module b {
    requires c;
    requires d;
}
```

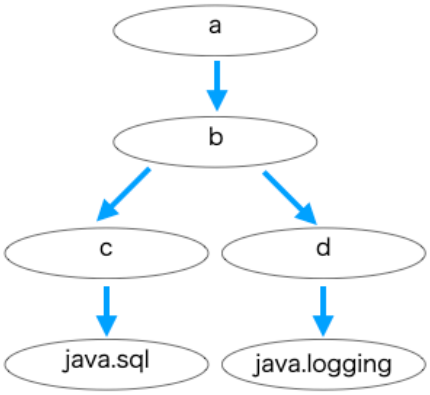
```
module c {
    requires java.sql;
}
```

```
module d {
    requires java.logging;
}
```

`requires`で、そのプロジェクトで読み込む他のモジュールを設定します。
公式リファレンスに載っている、`java.sql`などのJava標準のモジュールももちろん読み込むことができますし、オリジナルで作成したモジュールも、`requires b`のようにして読み込むことができます。

モジュールを読み込むことを、モジュールに依存する、とも言いますので覚えておきましょう。
上記の例だと、モジュールaはモジュールbに依存している。
モジュールbはモジュールcとdに依存しています。。

このような依存関係を図に表したものを、モジュールグラフと呼びます。



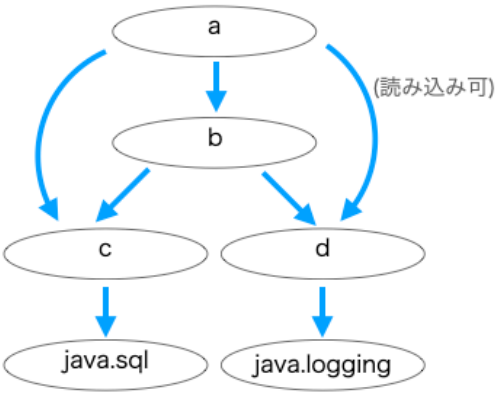
`requires`について1点注意なのが、 **この図において、モジュールから直接矢印が指しているところしか内容を読み込めません。**

aからb、bからまたc,dを読み込んでいますが、 **aとc,dは直接繋がっていないので読み込むことが出来ません。**

requires transitive

もし、間にモジュールを挟んだ孫モジュール等を読み込みたい場合は、`requires transitive`を使用します。
これを間接エクスポートと呼びます。

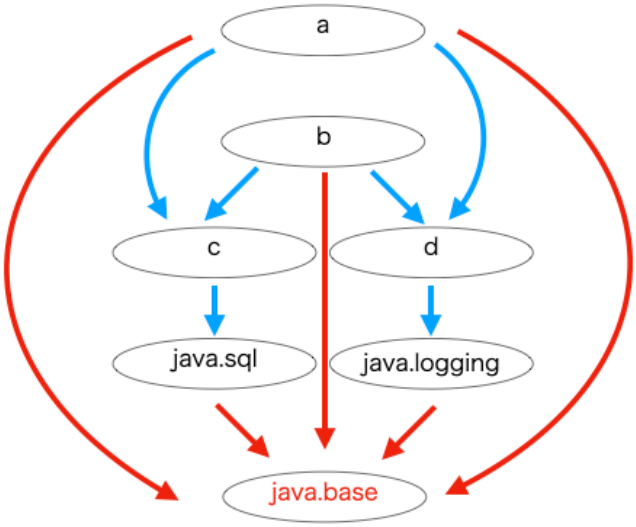
```
module b {
    //requires c;
    //requires d;
    //上記の代わりに、以下のコードを記述する。
    requires transitive c;
    requires transitive d;
}
```



モジュールbを読み込んだ場合、自動的にモジュールcとdも読み込ませるという処理が表現できます。

また、標準モジュールの中で`java.base`だけは特別で、`requires`を明記していなくても、全てのモジュールで自動的に`requires`がなされ、読み込むことができるようになっています。
暗黙的に依存している、とも言います。

すなわち、少し見づらいですが以下のようなモジュールグラフになります。



exports

上記の例で、プロジェクトaのモジュールaからモジュールbを読み込みますが、それだけではプロジェクトb内にある各パッケージを読み込むことはできません。

読み込ませたいパッケージは、`exports`を使用して必要な分を公開する必要があります。

```
module b {
    //プロジェクト内にあるcom.sampleパッケージを公開して読み込み可にする
    exports com.sample;

    requires transitive c;
    requires transitive d;
}
```

また、特定のモジュールからのみアクセス可能にしたい場合は、末尾に　`to モジュール名`を付け加えます。

```
module b {
    //プロジェクト内にあるcom.sampleパッケージを公開して、モジュールaからのみ読み込み可にする
    exports com.sample to a;

    requires transitive c;
    requires transitive d;
}
```

モジュールに関するJavaコマンド

ターミナルやコマンドプロンプトで行うJavaコマンドに、モジュールに関するコマンドが存在し、
必要なコマンドを問う問題も頻出です。

モジュールを実行する

```
//構文
> java --module-path(または-p) モジュールのルートディレクトリ -m 実行したいモジュールのクラス

//コマンド・出力例
> java --module-path mods -m hello/com.sample.Main
Hello

//解説
modsディレクトリ内にhelloモジュールが入っている。
-mコマンドを入力、モジュール名の後をスラッシュで区切り、モジュールに含まれる`パッケージ.クラス名`を記述。
```

モジュールの設定情報を調べる

```
//構文
> java --module-path モジュールのルートディレクトリ --describe-module モジュール名

//コマンド・出力例
> java --module-path mods --describe-module b
b file: ///Users/user/Desktop/java/mods/b/
exports com.sample;
requires transitive c;
requires transitive d;
requires java.base mandated

//解説
モジュールbの場所と、モジュールの設定内容を出力。
```

jmodファイルの設定情報を調べる

```
//構文
> jmod describe jmodファイルのパス

//コマンド・出力例
> jmod describe foo.jmod
foo
exports com.test;
requires java.base mandated

//解説
jmodファイルとは、使用するモジュールをまとめて圧縮したファイルです。
中に含まれるモジュール名とその詳細を出力します。
```

モジュールの依存関係を調べる

```
//構文
> jdeps --list-deps クラスファイルやJARファイル

//コマンド・出力例
> jdeps --list-deps hello.jar
java.base

//解説
jarファイルとは、コンパイルされたクラスファイル等を集めて圧縮したファイルです。
jdepsコマンドによって、jarファイルの中のファイルが、どのモジュールに依存しているかを出力します。
```

```
//構文b
> java --show-module-resolution ～～ディレクトリパスや実行クラス名など

//コマンド・出力例
> java --show-module-resolution 【省略】 foo/foo.Main
root foo file:///.../foo/classes/
foo requires a file:///.../a/classes/
bar requires b file:///.../b/classes/
bar requires c file:///.../c/classes/
java.base binds jdk.localedata jrt:/jdk.localedata
java.base binds java.desktop jrt:/java.desktop

//解説
javaでクラスファイルを実行する際、`--show-module-resolution`オプションをつけることによって、
モジュールの依存関係の構築やモジュールの読み込みの工程が出力されるようになります。
```

コンパイル時にエクスポート設定を追加する

```
//構文
> javac --add-exports モジュール名/エクスポートするパッケージ=利用するパッケージ

//コマンド・出力例
> javac --module-path mods --add-exports foo/com.test=hello 【～省略～】 Main.java

//解説
Main.javaファイルをコンパイルする際、追加でエクスポートして利用できるようにする。
fooモジュールのcom.testパッケージをエクスポートし、helloモジュールから読み取れるようにして、コンパイル。
あくまでコンパイル時に一時的に追加するものであり、コードが書き換わるわけではないので注意です。
```

課題

満点を取れるまで小テストを受験して下さい。

制限時間: 30 分

評定方法: 最高評点

合格点: 100 / 100

受験件数: 96