Exercices

Pour l'exercice qui suit, il est utile de disposer :

- jdk9,
- un outil d'édition de code

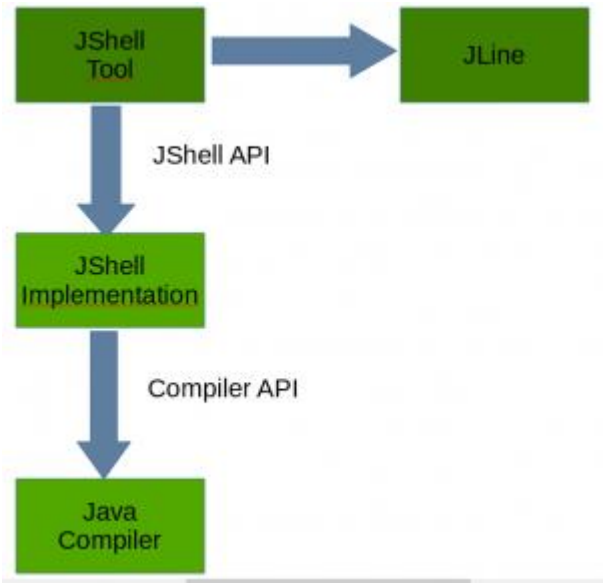
Exercice1

But de l'exercice : utiliser les nouveautés java 9 et surtout jshell

Avant Java 9, tester l'affichage d'un bout de code affichant Hello World, nécessitait un ScrapBook sous éclipse.

Lorsque du code est entré dans la console de `JShell`, il est traité par `JLine`. C'est une librairie Java pour gérer les saisies sur console.
Une fois le code saisi, il est analysé par `JShell` afin d'en déterminer le type (méthode, variable…) puis est encapsulé dans une classe en respectant les règles suivantes :

- Tous les imports sont placés en haut de la classe.
- Les variables, méthodes et déclarations de classes deviennent des membres statiques
- Les expressions et déclarations sont encapsulées dans une méthode à l'intérieur de la classe.



Travail à faire :

1. Configurez votre variable d'environnement JAVA_HOME afin qu'elle pointe vers le répertoire de le JDK 9.
2. Lancez `jshell` qui se trouve dans le répertoire bin\
3. Pour notre 1er test, afficher « exercice1. Pour ce faire, tapez la commande suivante : `System.out.println("exercice1");`
4. Listez les imports chargés au démarrage `/list -start`
5. Déclarer une variable entière `exercice` à 1 ;
6. Créez une interface `Printable` avec :
    a. Une méthode `print private` qui affiche une chaine de caractères s sur un `Printer` passés en paramètre,
    b. Une méthode `error public` qui affiche une `Exception` e sur un `Printer` passés en paramètre,
    c. Une méthode `debug public` qui affiche une `Map` m sur un `Printer` passés en paramètre,
7. Créez une classe `Event` qui implémente `Printable` avec :
    a. Un attribut `title` chaine de caractères
    b. Un attribut `start` date dans le futur par rapport au calendrier courant

   c. Un constructeur, des getters et setters où est contrôlé `start` par rapport à la date locale et le déclenchement de `error`, si la date est passée et le déclenchement de `debug` dans le cas contraire,
8. Faire un programme principal où est instancié un `Event` pour chaque jour de formation.
9. Supprimer la variable `exercice` avec `/drop` ,
10. Lister l'ensemble des variables créées dans JShell avec `/vars` ,
11. Lister l'ensemble des ytpes créées dans JShell avec `/types` ,
12. Importer la classe `LocalDate` pour l'utiliser dans la méthode `debug`.

Pour les exercices qui suivent, il est utile de disposer :

- jdk9, jdk10, jdk11, jdk12
- un outil d'édition de code
- un outil de construction de livrable

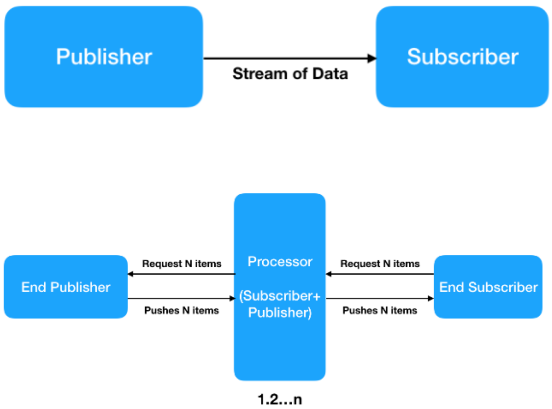
Via un IDE du type Eclipse, les exercices suivants sont à construire par l'emploi d'un builder externe du type Maven (version supérieure à 3.3.7)

Exercice2

But de l'exercice : utiliser les nouveautés l'API Flow

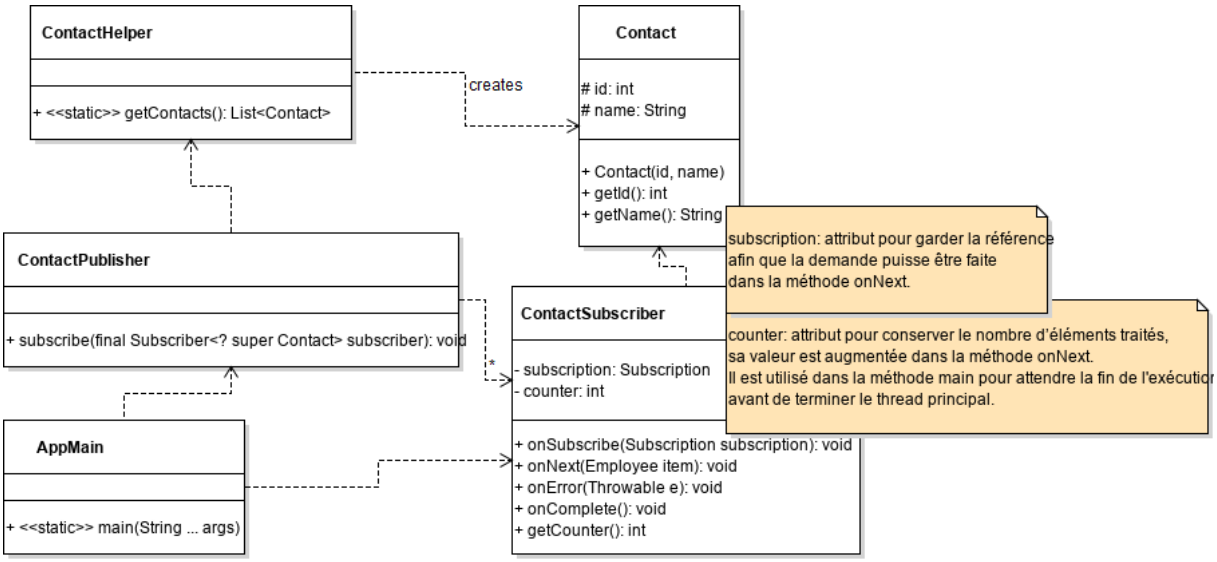
L'utilisation de Stream Reactive est basée sur les définitions issues de https://www.reactivemanifesto.org/, Java 9 a introduit le support des flux réactifs via l'API `java.util.concurrent.Flow`.

Les flux réactifs concernent le traitement asynchrone du flux, il devrait donc y avoir un publisher et un subscriber. Le publisher publie le flux de données et le subscriber utilise les données.
Parfois, les données sont transformées entre le publisher et le subscriber. Le processeur est l'entité située entre le publisher final et le subscriber pour transformer les données reçues du publisher afin que le subscriber puisse les comprendre.
Il peut y avoir une chaîne de processeurs.



Travail à faire :

1. Configurez votre variable d'environnement JAVA_HOME afin qu'elle pointe vers le répertoire de le JDK 9.
2. Via un IDE créez un projet maven, avec une version supérieure à 3.3.7
   a. Déclarez des propriétés pour que le compilateur soir en java 9
   b. Ajoutez un encoding pour les fichiers en utf-8
3. Créez un package `exercice2.streams` afin de coder les classes du diagramme suivant. Il faut connaître les classes de l'API Flow.
   Soit une classe `Contact` utilisée pour créer le flux de messages à envoyer du publisher au subscriber. La classe `ContactHelper` dispose d'une méthode `getContacts` pour retourner une liste de `Contact` qui sera tranformée en stream.



La classe `ContactSubscriber` qui implémente `Subscriber<Contact>` avec:

- La requête de subscription est appelée dans la méthode `onSubscribe` pour démarrer le traitement. Notez également qu'elle a de nouveau appelé la méthode `onNext` après le traitement de l'élément, ce qui oblige le publisher à traiter l'élément suivant.
- `OnError` et `onComplete` affichent des infos, mais dans le scénario réel, ils doivent être utilisés pour effectuer des mesures correctives en cas d'erreur ou le nettoyage des ressources lorsque le traitement est terminé.

La classe `ContactPublisher` qui hérite de la classe `SubscriptionPublisher<Contact>` ne possède qu'une méthode `subscribe` pour tracer dans un log les abonnements.

La classe `AppMain` joue le rôle de scénario de test pour lancer le publisher, le configurer par rapport au stream de Contacts et le subscriber pour les recevoir et les traiter.

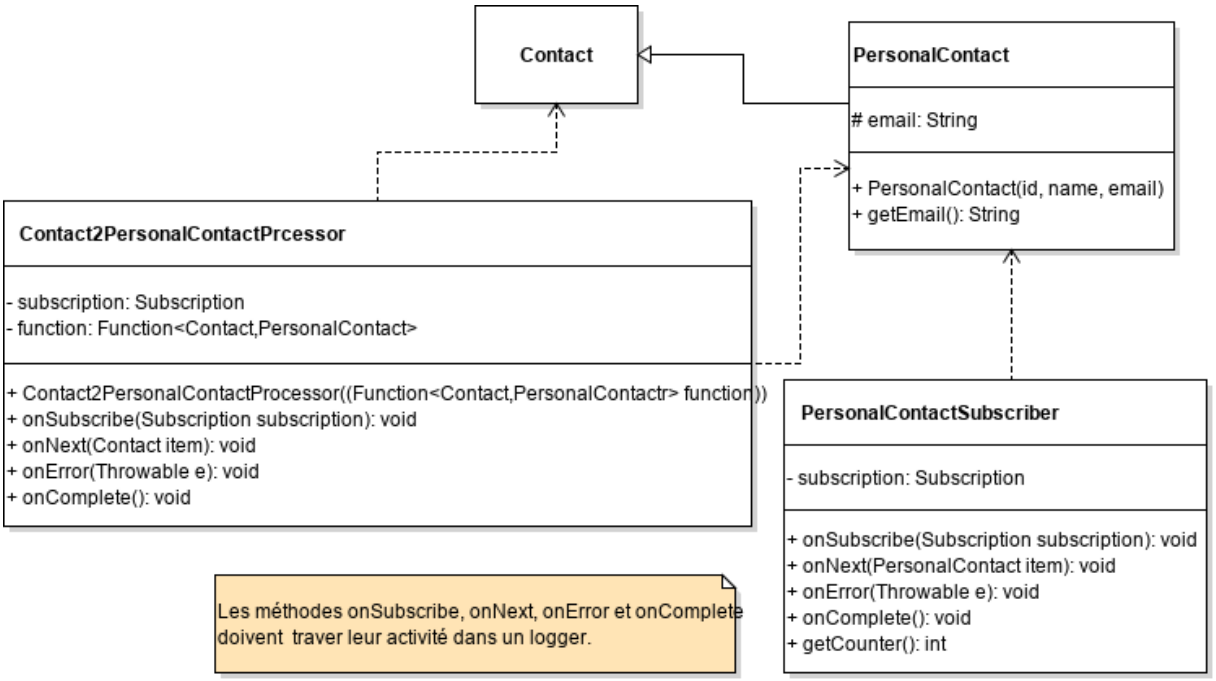
4. Transformation de message via un processeur pour transformer le message entre un publisher et un subscriber. La démarche est :

Faire un autre subscriber nommé `PersonalContactSubscriber` qui s'abonne à un type de message différent à traiter tel que `PersonalContact`.

Faire un processeur nommé `Contact2PersonalContactPrcessor` qui hérite de `SubscriptionPublisher<PersonalContact>` et implémente `Processor<Contact, PersonalContact>`.

- `function` sera utilisée pour convertir l'objet `Contact` en objet `PersonalContact`. Nous convertirons le message `Contact` entrant en message `PersonalContact` via la méthode `onNext`, puis nous utiliserons la méthode d'envoi `SubmissionPublisher` pour l'envoyer au subscriber.
- Puisque `Processor` fonctionne à la fois comme subscriber et comme publisher, nous pouvons créer une chaîne de processeurs entre les publisher finaux et les subscriber.

Refaire la classe `AppMain` avec son scénario de test afin d'utiliser la seconde classe de subscriber avec en amont un le processeur de transformation.



5. Nous voulons utiliser la méthode d'annulation d'abonnement pour ne plus recevoir de message dans le subscriber. Notez que si nous annulons l'abonnement, le subscriber ne recevra pas le signal `onComplete` ou `onError`.
   Enrichir la méthode `onNext(Contact item)` dans lequel le subscriber ne consomme que 3 messages, puis annule l'abonnement.
   Ces 3 messages devrons être écrit dans un fichier conntact.txt dont la gestion sera faite avec un *try with resource*s.
6. Optionnel Pour implémenter un algo de back pressure, il faut enrichir notre implémentation en suivant la démarche de référence sur
   https://github.com/ReactiveX/RxJava/wiki/Backpressure

Exercice3

But de l'exercice : construire des composants respectant jigsaw

En termes simples, la modularité est un principe de conception qui nous aide à atteindre un couplage lâche entre les composants via des contrats clairs et dépendances entre composants implémentation cachée utilisant une encapsulation forte

Travail à faire :

1. Configurez votre variable d'environnement JAVA_HOME afin qu'elle pointe vers le répertoire de le JDK 9.
2. Le but de cet exercice est de construire 2 composants dont le code est fourni dans l'énoncé au fil de l'eau. La construction des modules doit être faite avec Maven et l'usage d'un reactor. La démarche est :
   a. Une bonne décision serait de construire un projet parent `exercice3-parent` contenant l'ensemble des propriétés et plugins partagés par les projets modulaires, tels que :

```
<maven.version>3.3.9</maven.version>
```

```
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
<jdk.version>9</jdk.version>
```

b. Construire un projet Maven enfant nommé `exercice3-persistance` afin de définir le module nommé : `module.persistence` :
Configurer le pom.xml afin de développer en java 9,
Dans `src\main\java`, il faut créer :

1. Un fichier `module-info.java` : pour la description du module. Il utilise une API de log, et expose les classes importantes de cette couche de persistance, à savoir les entités et les repositories.
2. Un package nommé `fr.jva.persistence` : il contient
   a. Un fichier `package-info.java` pour les commentaires de ce package,
   b. Un sous package `entities` contenant la classe `UserEntity`.

```
public class UserEntity
{
    private String firstName;
    private String lastName;

    public UserEntity(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }
}
```

   c. Un sous package `repository` contenant :
      i. L'interface `UserRepository`, pour accéder à tous les objets `UserEntity`,

```
public interface UserRepository
{
    Stream<UserEntity> findAll();
}
```

      ii. La classe `RepositoryFactory`, pour créer des `UserEntity`,

```
public class RepositoryFactory
{
    public UserRepository createUser(){
        return new UserRepositoryImpl();
    }
}
```

      iii. Un sous package `impl` pour définit l'interface `UserRepository` via la classe `UserRepositoryImpl`,

```
public class UserRepositoryImpl implements UserRepository
{
    private static Logger LOGGER =
Logger.getLogger(UserRepositoryImpl.class.getName());
```

```
    public Stream<UserEntity> findAll() {
        LOGGER.info("UserRepositoryImpl : Find All Start");
        UserEntity userEntity1 = new UserEntity("Gerard", "Menvussa");
        UserEntity userEntity2 = new UserEntity("Leopold", "Nord");
        List<UserEntity> all = List.of(userEntity1, userEntity2);
        LOGGER.info("UserRepositoryImpl : Find All End: " +all.size()+" entities");
        return all.stream();
    }
}
```

c. Construire un projet Maven enfant nommé `exercice3-service` afin de définir le module nommé : `module.service` :
Configurer le pom.xml afin de développer en java 9, ajouter la dépendance sur le projet `exercice3-persistance`.
Dans `src\main\java`, il faut créer :
   1. Un fichier `module-info.java` : pour la description du module. Il utilise le module `module-persistence`.
   2. Un package nommé `fr.jva.service` : il contient
      a. Un fichier `package-info.java` pour les commentaires de ce package,
      b. Une classe AppMain ayant une méthode main pour faire un premier test.

```
public class AppMain
{
    public static void main(String[] args) {
        //UserRepositoryImpl is hidden
        RepositoryFactory factory = new RepositoryFactory();
        UserRepository userRepository = factory.createUser();
        Stream<UserEntity> entities = userRepository.findAll();
        entities.forEach(user -> System.out.println(user.getFirstName() + " " +
user.getLastName()));
    }
}
```

d. Construire un projet Maven `exercice3-modulaire` qui est un reactor composé des 2 précédents projets. De plus, il faut s'assurer que les plugins compiler et exec sont présents et configurer avec des versions adaptées à java 9

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>3.7.0</version>
   <configuration>
          <source>9</source>
          <target>9</target>
          <showWarnings>true</showWarnings>
          <showDeprecation>true</showDeprecation>
   </configuration>
</plugin>
<plugin>
   <groupId>org.codehaus.mojo</groupId>
   <artifactId>exec-maven-plugin</artifactId>
   <version>1.6.0</version>
   <configuration>
          <executable>java</executable>
          <arguments>
                 <argument>--module-path</argument>
                 <modulepath/>
```

```
                <argument>--module</argument>
                <argument>...</argument>
            </arguments>
    </configuration>
</plugin>
```

Il est important que l'ensemble des jars générés se trouve dans le même répertoire afin de facilité l'écriture du module-path. Pour ce faire, il faut configurer le plugin qui construit les jars.

e. Effectuer une exécution en utilisant le plugin exec :java et les argument module-path et module.

3. Il est désormais demandé de revoir l'implémentation du composant `module-service`. Le but est de supprimer la classe `AppMain` et la remplacer par une interface `UserService` qui offre une méthode `getUsers(Comparator<UserEntity> c) : Stream<UserEntity>`, afin de retourner un flux d'entité respectant un critère. C'est un module abstrait qui nécessite une implémentation
4. Il est demandé de construire un nouveau projet `exercice3-service-impl` pour offrir une implémentation de du contrat `module-service`.
    a. Dans la fichier `module-info.java`, il est important de disposer de dépendances (`requires`) sur l'interface implémentée et de déclarer le contrat qui est implémenté (`provides … with …`)
    `provides fr.jva.service.UserService with fr.jva.service.impl.UserComparatorService`,
    b. Créer un package `fr.jva.service.impl` et une classe d'implémentation nommée `UserComparatorService`.

    Utiliser le reactor pour reconstruire les 3 projets
5. Faire un module qui consomme le service exposé. Notre module `exercice3-gui` va utiliser au runtime les implémentations disponibles pour afficher les `UserEntity` aux utilisateurs.
    a. Dans le fichier modle-info.java figurent des `requires` sur le module exposant l'interface et tous les modules proposant des implémentations et, très important, faire un `uses` sur l'interface utilisé (déclarée dans le projet exercice3-service) pour que le mécanisme de `ServiceLoader` fonctionne.
    b. Dans un package `fr.jva.gui` , créer une classe `AppMain` avec une méthode `main` pour rechercher un service via `ServiceLoader`.

```
ServiceLoader<UserService> serviceProviders =ServiceLoader.load(UserService.class);
```

    Utiliser le reactor pour reconstruire les 4 projets
    Effectuer une exécution en utilisant le plugin `exec` :java et les argument `module-path` et `module`.
6. Modifier le composant `exercice3-gui`, afin de faire de l'introspection sur la classe d'implémentation du service. La sécurité de la JVM a été renforcée en ce sens. Pour que les types contenus dans un package soient manipulables par réflexion, il faut les ouvrir avec `opens`.

```
module exercice3-gui {
    opens exercice3-service;
}
```

    À noter : il faut également faire un export du package si les types doivent être utilisés normalement. Sinon vous ne pouvez que les utiliser par réflexion.
    Si c'est l'ensemble du module qui doit être ouvert à la réflexion, il faudra écrire `open`

`module` et il n'y aura pas besoin de préciser `opens` sur les packages dans le module-info.java du composant exercice3-service.

7. `jlink` est un outil qui génère une image d'exécution Java personnalisée contenant uniquement les modules de plate-forme requis pour une application donnée. Avec `jlink`, nous pouvons créer notre propre petit JRE qui ne contient que les classes pertinentes que nous voulons utiliser, sans gaspillage de mémoire, ce qui entraîne une augmentation des performances.
    a. Pour utiliser `jlink`, nous avons besoin de connaître la liste des modules JDK utilisés par l'application et que nous devrions inclure dans notre JRE personnalisé. Utilisons la commande `jdeps` pour obtenir les modules dépendants utilisés dans l'application: `exercice3-gui`,

```
jdeps --module-path <repertoire des modules> -s --module <nom du module>
```

    b. Pour créer un environnement JRE personnalisé pour une application basée sur un module, on utilise la commande `jlink`. Voici sa syntaxe de base:

```
jlink [options] –module-path modulepath
  –add-modules module [, module…]  --output <target-directory>
```

    c. Nous avons maintenant notre JRE personnalisé créé par `jlink`. Pour tester notre JRE, essayons d'exécuter notre module en naviguant dans le dossier bin du répertoire en paramètre de `--output` et exécutez la commande ci-dessous:

```
java --module exercice3-gui/fr.jva.exercice3.AppMain
```

    d. Vous pouvez également créer un environnement JRE personnalisé avec des scripts de lancement exécutables. Pour cela, nous devons exécuter la commande `jlink` qui a un paramètre `–launcher` pour créer notre programme de lancement avec notre module et notre classe principale. Cela générera 2 scripts:

```
--launcher exercice3jrelauncher=module-gui/fr.jva.exercice3.AppMain
```

    `exercice3jrelauncher.bat` et `exercice3relauncher` dans notre répertoire `exercice3jre/bin`.

Exercice4

But de l'exercice : faire de 'inférence de type en Java 11

Travail à faire :

Configurez et faire un exemple d'usage de var

```
public class Example {

    public static void main(String[] args) throws Exception {
        var list   = new ArrayList<String>();
        var stream = list.stream();

        var newList = List.of("hello", "John");
        newList.forEach(System.out::println);

        String fileName = "./pom.xml";
        var path  = Paths.get(fileName);
        var bytes = Files.readAllBytes(path);

        System.out.println("info: " + bytes);

        for (var b : bytes) {
```

```
            // TODO
        }

        try (var foo = new FileInputStream(new File(""))) {
            System.out.println(foo);
        } catch (Exception e) {
            // ignore
        }
    }
}
```