2019

Compte rendu de TP : Interface fonctionnelle, la puissance du déclaratif

Java avancé – TP n°3

Jonathan CRÉTÉ – Filière Informatique

Exercice 1 - Docteur, je me sens mal ?

1) Écrire la méthode healthCheck sachant que dans notre cas, le contenu de la réponse HTTP ne nous intéresse pas, seul un code de réponse de 200 est suffisant.

Note : la doc de la classe HttpClient vous donne un exemple d'utilisation.

```
public static boolean healthCheck(URI uri) throws InterruptedException {
	Objects.requireNonNull(uri);
	if(!uri.getScheme().equals("http")) {
		throw new IllegalArgumentException("URI scheme is not
HTTP!");
	}
	try {
		HttpClient client = HttpClient.newBuilder()
				.version(Version.HTTP_1_1)
				.connectTimeout(Duration.ofSeconds(20))
				.build();

		HttpRequest req = HttpRequest.newBuilder()
				.uri(uri)
				.build();

		HttpResponse<String> response = client.send(req,
BodyHandlers.ofString());
		System.out.println(response.statusCode());
		if(response.statusCode() == 200) {
			return true;
		}

	} catch (IOException e) {
		System.err.println(e);
	}
	return false;

}
```

2) Quelle est la classe en Java qui correspond à un résultat ou pas et qui va donc servir de valeur de retour pour notre fonction condition ?

Définir l'interface fonctionnelle URIFinder qui possède une méthode find pour que le code d'utilisation d'un URIFinder ci dessous fonctionne

Notre valeur de retour sera un objet de type Optional. Ont défini notre interface fonctionnel avec l'annotation @FunctionalInterface et sa seule méthode abstraite *find()*

```
@FunctionalInterface
public interface URIFinder {
	// implicite abstract
	public  Optional<URI> find();
```

3) On cherche maintenant à écrire la méthode fromArguments qui prend en paramètre un tableau de String (celui fourni par le main) et renvoie un URIFinder qui considère le premier argument, et si il existe, permet de le transformer en URI (en utilisant URI.create).

Où doit-on placer la méthode fromArguments et quels sont les modificateurs (public etc) de celle-ci ?

Écrire la méthode fromArguments.

Une *factory method* est une méthode qui permet de créer un objet mais qui n'est pas un constructeur

La méthode fromArguments doit être écrite dans l'interface UriFinder et elle doit être public et static (L'interface fonctionnel doit posséder une seule méthode abstraite ne possédant pas d'implémentation par défaut.).

Dans un premier temps, on créer un nouvel objet de la classe anonyme qui implémente URIFinder.

```
public static URIFinder fromArguments(String[] args) {
	Objects.requireNonNull(args);
	return new URIFinder() {
		@Override
		public Optional<URI> find() {
			if(args.length == 0) {
				return Optional.empty();
			}
			// create -> factory
			return Optional.of(URI.create(args[0]));
		}
	};
}
```

Puis on utilise une lambda sans arguments pour implémenter l'interface fonctionnelle :

```
public static URIFinder fromArguments(String[] args) {
	Objects.requireNonNull(args);
	return () -> {
		if(args.length == 0) {
			return Optional.empty();
		}
		return Optional.of(URI.create(args[0]));
	};
}
```

Enfin, on développe une méthode ***buildAnUri(...)*** qui construis , notre objet URI à partir d'une chaîne de caractère (Peut lever une ***URISyntaxException*** et retourner un objet *Optional empty*)

```
public static Optional<URI> buildAnURI(String s) {
	try {
		return Optional.of(new URI(s));
	} catch (URISyntaxException e) {
		return Optional.empty();
```

```
	}
}

public static URIFinder fromArguments(String[] args) {
	Objects.requireNonNull(args);
	return () -> {
		return Optional.of(args)
			.filter(a -> a.length != 0)
			// Optional de Optional
			.flatMap(a -> buildAnURI(a[0]));
	};
}
```

On vérifie tout d'abord que le tableau de Strings en paramètre n'est pas null ;

Ensuite, avec une *lambda bloc*, on retourne un objet **Optional** du tableau auquel on applique deux opérations (un peu comme avec un stream).

On filtre de manière à ne pas avoir un élément du tableau de longueur 0. Enfin, on utilise la méthode *flapMap(...)* de manière à aplatir notre objet **Optional de Optional** , tout en construisant notre URI avec le premier argument du tableau.

4) On cherche maintenant à écrire une méthode fromURI qui prend une chaîne de caractères et la transforme en URI.

Que se passe-t-il si la chaîne de caractères n'est pas une URI valide ?

Comment faire si l'on souhaite ignorer les chaînes de caractères qui ne sont pas des URI valides ?

Écrire le code de la méthode fromURI et réviser le code de fromArguments pour que lui aussi ignore les chaînes de caractères qui ne sont pas des URI valides.On appel la méthode close()

Si la chaîne de caractères n'est pas une URI valide, alors une **UriSyntaxException** est levée.

Notre méthode *buildAnURI(...)* retourne un Optional.

```
public static URIFinder fromURI (String st) {
	Objects.requireNonNull(st);
	return () -> buildAnURI(st);
}
```

5) On souhaite maintenant écrire une méthode or qui permet de combiner deux URIFinder de telle façon que si le premier URIFinder ne trouve pas d'URI, alors le second essaie de trouver son URI.

Où doit-on placer la méthode or et quels sont les modificateurs de celle-ci ?

Écrire la méthode or.

La méthode or est une méthode d'instance. Toutefois on rappelle que dans le cas de notre interface fonctionnelle URIFinder, on doit n'avoir qu'une seule méthode abstraite ne possédant pas d'implémentation par défaut. Pour palier à ce problème notre méthode *or(...)* sera une méthode par défaut et toujours dans notre interface.

```
default URIFinder or(URIFinder uri){
	Objects.requireNonNull(uri);
	return () -> {
		return find().or(() -> uri.find());
	};
}
```

6) On souhaite écrire la méthode fromMapGetLike qui prend comme premier paramètre un nom de clé/propriété et comme second paramètre une fonction ayant la même signature et la même sémantique que map::get (avec des String comme type de clé et de valeur dans la Map). Cette méthode devra renvoyer un URIFinder permettant de chercher la valeur de la clé dans la Map.

Voici un exemple d'utilisation.

Écrire le code de la méthode fromMapGetLike et vérifier aussi que le code fromMapGetLike("HEALTH_CHECK_URI", System::getenv) est valide.

Rappel: map.get renvoie null si il n'y a pas de valeur associée à une clé donnée.

```
public static URIFinder fromMapGetLike(String key , Function<? super
String, String> get ) {
	Objects.requireNonNull(key);
	Objects.requireNonNull(get);
	return () -> {
		return Optional.ofNullable(get.apply(key))
				.flatMap(URIFinder::buildAnURI);
	};
}
```

7) En fait, la méthode fromMapGetLike pourrait avoir des types de paramètre (une signature) acceptant plus de cas valides. Par exemple, le code suivant dans lequel les clés sont des Integer devrait aussi fonctionner.

On modifie notre méthode avec une variable de type, de manière à obtenir une méthode *fromMapGetLike(..)* paramétrée (Le but est d'ajouter une contrainte sur les types des paramètres)

```
public static<E> URIFinder fromMapGetLike(E key , Function<? super E,
String> get ) {
	Objects.requireNonNull(key);
	Objects.requireNonNull(get);
	return () -> {
		return Optional.ofNullable(get.apply(key))
				.flatMap(URIFinder::buildAnURI);
};
}
```

8) Enfin, on souhaite ajouter une méthode fromPropertyFile qui prend en paramètre le chemin d'un fichier et le nom d'une clé et renvoie un URIFinder qui renvoie l'URI associé à la clé dans le fichier de properties si l'association existe et que la valeur associée est bien une URI valide.

Écrire la méthode fromPropertyFile.

Note : il existe une méthode Properties.load().

Note 2 : vous vous rappelez surement qu'une bonne façon de lire un fichier de caractères est d'utiliser un BufferedReader...

```
	public static URIFinder fromPropertyFile(Path path, String key) {
		Objects.requireNonNull(path);
		Objects.requireNonNull(key);
		var file = path.toFile();

		try {
			FileReader reader = new FileReader(file);
			BufferedReader buffer = new BufferedReader(reader);
			Properties ppt = new Properties();
			ppt.load(buffer);
			ppt.forEach((x, value) -> System.out.println("Key : " +
x + ", Value : " + value));
			var value =  ppt.getProperty(key);
			buffer.close();
			/*
			if(value != null) {
				return () -> buildAnURI(value);
			}
			else {
				return () -> Optional.empty();
			}
			*/

			return () -> {
				return Optional.ofNullable(value)
						.flatMap(elt -> buildAnURI(value));
```

```
                };
            }
            catch (IOException io) {
                io.printStackTrace();
                return () -> Optional.empty();
            }

        }
```

Conclusion

Durant ce troisième TP de JAVA avancé, j'ai pu comprendre qu'une interface peut avoir différents buts :

- Donner un type commun et abstraire des classes représentant les données
- Abstraire la façon dont le code s'exécute.

J'ai pu encore une fois revoir les *lambda avec le type Optional*, cela devient un automatisme lorsque l'on veut s'assurer que la valeur de retour null est encapsulée. En termes de difficultés rencontrées , j'ai toujours un peu de mal avec la structure de données *Map*, il faut que je m'entraine de ce côté-là. J'ai toutefois bien compris qu'une interface fonctionnelle doit posséder une seule méthode abstraite ne possédant pas d'implémentation par défaut. Par conséquent, je dois vraiment travailler sur l'implémentation de la structure Map et sur les lambda.