Contrôle final de Compléments en Programmation Orientée Objet
(Correction)

- **Durée : 1 heures 50 minutes.**
- **Tout document ou moyen de communication est interdit. Seule exception : une feuille A4 recto/verso avec vos notes personnelles.**
- **Le rendu de la Section I) consiste en un programme entier (constitué de plusieurs fichiers sources). Celui-ci sera à présenter le mieux possible, sans faire de réponse séparée pour chaque question ou même suivre l'ordre des questions. Le programme est à réaliser en Java 11.**
- **Répondez au dernier exercice directement sur le sujet. Il est imprimé sur une feuille séparée à joindre à la copie que vous rendrez.**
- **Le barême est seulement donné à titre indicatif.**

Quelques rappels et indications :

- L'interface fonctionnelle `BiConsumer` du package `java.util.function` :

```
1  @FunctionalInterface public interface BiConsumer<T, U> { public void accept(T var1, U var2); }
```

- Les interfaces `java.lang.Runnable` et `java.util.concurrent.Callable` :

```
1  @FunctionalInterface public interface Runnable { void run(); }
```

```
1  @FunctionalInterface public interface Callable<V> { V call(); }
```

- À propos de `ForkJoinPool` et de `ForkJoinTask` (`java.util.concurrent`) :
  - créer une tâche (`ForkJoinTask<V>`) à partir d'une lambda-expression (`Runnable` ou `Callable<V>`) : `ForkJoinTask.adapt(()-> { /*instructions */})`. Attention, ceci crée la tâche sans l'exécuter.
  - créer un *pool* borné à 16 *threads* : `new ForkJoinPool(16)`
  - exécuter une tâche sur un *pool* donné : `pool.submit(task)`
  - attendre la fin d'une tâche : `task.join()` (retourne le résultat, de type `V`, de la tâche)
  - exécuter une tâche sur le *pool* courant : `task.fork()` (*pool* courant : c'est celui qui gère le *thread* courant ou bien, à défaut, `ForkJoinPool.commonPool()`, si le *thread* courant n'est pas géré par un `ForkJoinPool`).

I) Serveur web

Nous voulons écrire une bibliothèque pour faciliter l'implémentation d'applications web côté serveur. En utilisant celle-ci, une application serveur pourrait être paramétrée et démarrée de la façon suivante :

```
1  package sampleapp;
2
3  import mywebframework.Server;
4
5  public class App {
6      public static void main(String[] args) {
7          var server = new Server();
8          server.get("/", (req, res) -> {
9              res.code = 200; // OK
```

```
10          res.body = "<html><head><title>Page d'accueil</title></head><body><h2>Ceci est la
                racine.</h2></body></html>";
11      });
12      server.get("/about", (req, res) -> {
13          res.code = 200; // OK
14          res.body = "<html><head><title>À propos de mon site</title></head><body><h2>Ce site est
                motorisé par MyWebFrameWork 0.1.</h2></body></html>";
15      });
16      server.use(null, (req, res) -> { // route par défaut
17          res.code = 404; // NOT FOUND
18          res.body = "<html><head><title>Page introuvable</title></head><body><h2>La page que vous
                cherchez est introuvable sur ce site.</h2></body></html>";
19      });
20      server.start(8080);
21  }
22 }
```

Résumé du fonctionnement : les appels `get` et `use` enregistrent des « routes » dans une liste gérée par l'instance de `Server`. La méthode `start` demande au serveur d'attendre les futures connexions sur le port 8080 et d'y répondre à l'aide des routes enregistrées.

Les connexions prennent la forme de requêtes HTTP, décrites par une « méthode »[1] HTTP (une valeur parmis GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE ou PATCH) et une URI (c'est-à-dire le chemin du document demandé).

Pour chaque requête reçue, le serveur cherche, dans sa liste de routes, la première capable de traiter la requête, puis l'exécute. L'exécution de la route construit une réponse que le serveur envoie ensuite sur la *socket* de la connexion. Typiquement, la réponse contient un code de statut et du texte (une page web) qui sera affiché par le client (navigateur web) qui s'était connecté.

On donne d'ores et déjà la classe `Request`, représentant une requête HTTP :

```
1  package mywebframework;
2
3  public final class Request {
4      private final HttpMethod type;
5      private final String uri;
6      public Request(HttpMethod type, String url) {
7          this.type = type;
8          this.uri = url;
9      }
10     public HttpMethod getType() {
11         return type;
12     }
13     public String getUri() {
14         return uri;
15     }
16 }
```

Remarque : les classes et interfaces à programmer seront placées dans le *package* `mywebframework`.

Exercice 1 : Diagramme de classes (2 points)

Lisez tout le sujet et dessinez, sur un brouillon, un diagramme de toutes les classes et interfaces évoquées dans le sujet. À la fin de l'épreuve, mettez à jour ce diagramme et recopiez-le sur votre copie. Suivez, tant que possible, les conventions UML.

1. C'est le terme consacré. À ne surtout pas confondre avec les méthodes de Java. Ainsi, dans ce sujet, nous garderons les guillemets à chaque fois que nous parlerons d'une « méthode » HTTP, alors que le mot méthode pour les méthodes Java sera sans guillemets.

Exercice 2 : Représenter les « méthodes » HTTP (1 point)

Écrire le type `HttpMethod` dont les instances représentent les différentes « méthodes » HTTP. Remarque : il y a un ensemble fini et fixe de « méthodes » HTTP.

Exercice 3 : Construire une réponse (2 points)

Pour simplifier, nous considérons qu'une réponse HTTP est constituée d'un code de réponse (200 pour OK, 400 pour une requête mal formée, 404 pour document non existant, etc.) et d'un contenu (du texte).

1. Pour des raisons de fiabilité, on veut que le type `Response` soit immuable : les attributs (`code` et `body`) sont non modifiables et la classe n'est pas extensible [2].

   Écrire la classe `Response` dont les instances sont des réponses à des requêtes HTTP.

   Les attributs seront tous **`private`** mais auront des *getters* **`public`**. Un unique constructeur prendra en argument les paramètres nécessaires pour initialiser les attributs.

2. En réalité, une réponse contient aussi des méta-données de toutes sortes (et pas forcément pertinentes pour tous les types de réponse) et aurait donc bien plus que 2 attributs.

   Plutôt que de devoir utiliser directement un constructeur avec de trop nombreux paramètres (ou de créer de multiples contructeurs), nous allons utiliser le patron monteur/*builder*.

   Écrire la classe `ResponseBuilder` qui
   — contient les mêmes attributs que `Response`, mais qui sont, cette fois-ci, **`public`** et modifiables,
   — et a une méthode **`public`** `Response build()` qui retourne une nouvelle instance de `Response` dont les attributs sont initialisés avec les mêmes valeurs que les attributs de même nom de l'instance courante de `ResponseBuilder`.

Exercice 4 : Déclarer les routes (4 points)

Une route est une règle permettant de traiter des requêtes correspondant à des critères définis (selon sa « méthode » HTTP et son URI ; là encore, il s'agit d'une simplification).

Une application web est définie par une liste de telles routes (l'ordre importe, car c'est la première règle correspondant à la requête qui sera exécutée).

On donne l'interface `mywebframework.Route` :

```
1  package mywebframework;
2
3  public interface Route {
4      boolean handles(Request request); // dit si cette route peut traiter request
5      Response handle(Request request); // traite request en construisant une réponse qu'elle retourne
6  }
```

Écrire la classe `mywebframework.Server` en y mettant :

1. comme seul attribut (privé, final), une liste (modfiable) de routes,
2. un seul constructeur, sans paramètre,
3. la méthode **`private`** `Response handle(Request req)` qui parcourt les routes enregistrées jusqu'à trouver la première qui convient pour `req`, puis exécute cette route et retourne la réponse fabriquée par cette route,

2. On pourrait admettre qu'elle ait des sous-classes à condition qu'elles soient toutes immuables. Pour cela, il faut qu'elle soit « scellée ». Cela dit, pour `Response` nous n'avons pas besoin de sous-classes.

4. les méthodes suivantes :

```
1  public void use(String uri, BiConsumer<Request, ResponseBuilder> handler); // ajoute une route
       traitant toutes les requêtes dont l'URI est uri.
2  public void get(String uri, BiConsumer<Request, ResponseBuilder> handler); // ajoute une route
       traitant toutes les requêtes de méthode GET dont l'URI est uri.
3  public void post(String uri, BiConsumer<Request, ResponseBuilder> handler); // ajoute une route
       traitant toutes les requêtes de méthode POST dont l'URI est uri.
```

Par convention, si la valeur `null` est passée pour le paramètre `uri`, la route doit traiter toutes les requêtes (ayant la bonne « méthode » HTTP) sans distinction d'URI.

Remarques :

— *Ces méthodes doivent instancier des routes avant de les ajouter à la liste. Mais pour instancier une interface, il faut l'implémenter ! (définir une ou des classes)*

— *Attention, le paramètre `handler` est un `BiConsumer<Request, ResponseBuilder>`, représentant une fonction ne retournant rien (mais `ResponseBuilder` est une structure mutable, dont l'instance passée fait office de valeur de retour).*
Par ailleurs la méthode `handle` de l'interface `Route` retourne bien `Response`, et non `ResponseBuilder`.

— *Le sujet ne demande pas d'écrire les autres méthodes `put`, `delete` ... pour les « méthodes » HTTP de même nom. Évidemment, le logiciel complet devrait contenir ces méthodes.*

La classe `Server` sera finalisée dans l'exercice suivant (écriture de la méthode `start`).

Exercice 5 : Orchestrer les connexions (5 points)

Vous allez maintenant écrire la méthode `public void start(int port)` de la classe `Server`.

Son fonctionnement est le suivant : d'abord on instancie `ServerSocket`, puis, dans une boucle infinie (exécutée dans un *thread* dédié), on écoute les connections sur cette *socket* (méthode `listen`) et on récupère des *sockets* de connexion (`ConnectionSocket`) quand `listen` retourne.

Dès qu'une *socket* de connexion est récupérée, on lance le traitement de la connexion dans un autre *thread*. Le traitement consiste à lire une requête HTTP (`readRequest` de `ConnectionSocket`), à construire la réponse en exécutant la méthode `handle` de `Server` et l'envoyer sur la *socket* (`sendResponse` de `ConnectionSocket`), puis fermer la *socket* (`close`).

Les classes suivantes[3] sont supposées fournies :

```
1  package mywebframework;
2
3  public class ServerSocket implements AutoCloseable {
4      public ServerSocket(int port) { ... } // construit une socket communiquant sur le port port
5      public ConnectionSocket listen() { ... } // attend une connexion et retourne son descripteur
6      @Override public void close() { ... } // fermer la socket
7  }
```

```
1  package mywebframework;
2
3  public class ConnectionSocket implements Autocloseable {
4      public Request readRequest() { ... } // reçoit une requête (bloquante jusqu'à  réception)
5      public void sendResponse(Response resp) { ... } // envoie la réponse (bloquante jusqu'à  envoi).
6      @Override public void close() { ... } // fermer la socket
7  }
```

3. Remarque : dans la « vraie » vie, les sockets ne gèrent pas HTTP directement. On récupère et on envoie des données brutes via le protocole TCP. Pour gérer HTTP, il faut une classe de plus pour analyser les paquets reçus et pour formatter les réponses à envoyer. Mais pour le sujet, cette socket dédiée à la gestion du protocole HTTP est bien pratique !

À faire : programmer la méthode `start`. Il est possible d'utiliser les *threads* directement, mais préférable d'utiliser un *thread pool* borné (en effet : au delà d'un certain nombre de connexions concurrentes, la qualité de service se dégrade).

Nom, prénom :

II) Questions de cours

Exercice 6 : Questionnaire (6 points)

*Pour chaque case, inscrivez soit "**V**"(rai) soit "**F**"(aux), ou bien ne répondez pas.*
Note = max(0, nombre de bonnes réponse - nombre de mauvaises réponses), ramenée au barême.
Les questions concernent Java 11.

1. [F] Tout seul, le fichier `Z.java`, ci-dessous, compile :

```java
import java.util.function.*;
public class Z { Function<Object, Boolean> f = x -> { System.out.println(x); }; }
```

   Correction : La lambda expression donnée ici ne peut pas implémenter la méthode `apply` de `Function`, car son type de retour est `void` (ou plutôt : elle ne retourne rien et, en tout cas, certainement pas `Boolean`).
   Donc l'inférence vers `Function<Object, Boolean>` n'est pas possible.

2. [F] Une méthode déclarée à la fois `private` et `final` ne compile pas.

   Correction : Il n'y a pas de contradiction entre les deux modificateurs. Cependant, `final` est inutile pour une méthode `private`.

3. [F] Une classe `final` peut contenir une méthode `abstract`.

   Correction : Non car la méthode `abstract` ne pourrait jamais être implémentée.

4. [V] `HashSet<Integer>` est sous-type de `Set<Integer>`.

   Correction : Oui : d'une part `HashSet` implémente `Set`, d'autre part, les deux types génériques sont ici paramétrés avec le même paramètre (`Integer`).

5. [V] Une `enum` peut contenir une `interface` membre.

   Correction : C'est autorisé par le langage.

6. [V] Quand, on initialise un attribut de type `int`, sa valeur est stockée dans le tas.

   Correction : Les attributs primitifs font partie intégrante de l'objet, qui est stocké dans le tas (quant aux non-primitifs : la référence est partie intégrante de l'objet, celle-ci pointe vers un autre objet, aussi dans le tas).

7. [V] Tout objet existant à l'exécution est instance de `Object`.

   Correction : `Object` est par définition le type de tous les objets... et de `null`.

8. [F] Pour faire un *upcasting*, on doit demander explicitement le transtypage.

Correction : L'*upcasting* automatique est à la base du polymorphisme par sous-typage. L'opération est donc totalement tranpsarente (et sans danger).

9. F La classe d'un objet donné peut être modifiée à l'exécution.

Correction : Non, on ne modifie pas la classe d'un objet. Tout au plus, on peut *caster* une expression qui le référence.

10. F Toute variable locale de Java peut être une variable partagée (entre plusieur *threads*).

Correction : Non justement, les variables locales sont les seules qui ne peuvent pas être partagées ! C'est logique : elles sont stockées en pile, or chaque *thread* dispose de sa propre pile privée.

11. F Le programme suivant compile et affiche soit rien, soit 1 :

```
1  public class Truc {
2      public final int contenu = 1;
3
4      public static void main(String args[]) {
5          Truc truc = null;
6          new Thread(() -> { if (truc != null) System.out.println(truc.contenu); }).start();
7          truc = new Truc();
8      }
9  }
```

Correction : Non, il ne compile pas car la variable `truc` n'est pas effectivement finale. Elle n'est donc pas utilisable dans une lambda-expression.

12. V Le programme suivant compile et affiche soit rien, soit 1 :

```
1  public class Truc {
2      public final contenu = 1;
3      public static Truc instance = null;
4
5      public static void main(String args[]) {
6          new Thread(() -> { if (instance != null) System.out.println(instance.contenu); }).start();
7          instance = new Truc();
8      }
9  }
```

Correction : Oui, car l'attribut `contenu` est final, et donc son initialisation "arrive-avant" toute lecture ultérieure.
Sinon, en règle générale, même si `instance` n'est pas `null`, le *thread* créé n'est pas certain de voir l'objet totalement initialisé (il manquerait cette relation arrivé-avant, qu'on peut imposer en déclarant `contenu` comme `final` ou `volatile`).