

Examen

durée 2h, tiers temps additionnel 40 minutes

Consignes l'examen dure 2 heures et est noté sur 20 points. Il comporte trois exercices sur 8, dont un aide mémoire se trouvant à la page 7. Les notes manuscrites sont les seuls documents autorisés. Les barème n'est donné qu'à titre indicatif et reflète la difficulté des exercices (et donc le temps qu'il est conseillé d'y consacrer).

Attention : Dans toutes les questions de code, les endroits à compléter sont dénotés par un (n). Reportez ce numéro sur votre copie, suivi du code Java correspondant. Il est **interdit** de modifier du code ailleurs (en particulier de rajouter de nouveaux attributs aux classes, ou de modifier les signatures des méthodes).

1 Questions de cours (3 points)

Pour chacune des questions suivantes, reporter sur votre copie l'ensemble des lettres des réponses correctes (il peut y en avoir plusieurs ou aucune). Si aucune réponse n'est correcte selon vous, indiquez le par \emptyset . Un point est accordé si l'ensemble exact des bonnes réponses est sélectionné, sinon aucun point (pas de points négatifs).

1. En Java/JSP :

- (a) la session permet de partager des données entre plusieurs clients
- (b) la session nécessite l'utilisation de *cookies*
- (c) les fichiers *.jsp* font partie du modèle (au sens MVC)

Réponse: Bonne réponse 2. (La session permet de partager des données entre plusieurs pages pour un même client et les fichiers JSP font partie de la Vue).

2. Le mot clé *final* de Java :

- (a) Permet d'empêcher l'extension d'une classe par héritage
- (b) Permet d'empêcher la surcharge d'une méthode
- (c) Permet d'empêcher la modification d'une variable

Réponse: Bonnes réponses 1, 3. Il permet d'empêcher la redéfinition d'une méthode dans une sous-classe, pas la surcharge.

3. Soit le code d'une classe étendant *Thread* et le code l'utilisant :

```
class MyThread {
    int[] values; /* tableau d'entiers */
    MyThread(int[] v) { values = v; }

    void run() {
        for(int i = 0; i < values.length; i++) {
            values[i]++;
        }
    }
}

/* initialise un tableau de quatre 0 */
int [] tab = new int[4];

(new MyThread(tab)).start();
(new MyThread(tab)).start();
(new MyThread(tab)).start();
(new MyThread(tab)).start();

/* affiche les quatre cases */
for(int i : tab) System.out.print(i + " ");
```

- (a) Le programme peut afficher 4 4 4 4
- (b) Le programme peut afficher 1 2 3 4
- (c) Le programme peut afficher 0 0 0 0

(d) Le programme peut afficher 1 2 1 2

Réponse: Bonnes réponses 1, 2, 3, 4. En effet les 4 threads s'exécutent en parallèle et en parallèle de l'affichage. Réponse 1, les 4 threads se sont terminés avant l'affichage. Réponse 3 les 4 threads n'ont pas commencé avant l'affichage. Pour les réponses 2 et 4, il peut y avoir un entrelacement plus fin, avec problème d'accès. Par exemple, pour obtenir la réponse 2

- Case 1, les quatre thread lisent la valeur de la case (0) en même temps. Ils font tous une incrémentation (ils calculent tous la valeur 1) et ils écrivent tous dans la case (en écrasant les valeur les unes des autres).
- Même chose, mais un des threads fait son incrémentation après l'écriture des trois autres threads
- Même chose que le 2, avec deux threads en conflits et deux qui se suivent
- Case 4, tous les threads en séquence

2 Annuaire de machines (12 points)

Le contexte de cet exercice est celui d'un annuaire de machines sur un réseau.

2.1 TCP (7 points)

Dans cette première partie codée en TCP, un serveur est à l'écoute de connexions. Le code partiel du serveur est donné à la figure 1. Un client peut se connecter au serveur pour trois raisons :

- être ajouté à l'annuaire en envoyant le message "REGISTER"
- être retiré de l'annuaire en envoyant le message "UNREGISTER"
- connaître la liste de toutes les machines enregistrées avec LIST

Sur réception de ces messages, le serveur réagit comme suit :

- Pour "REGISTER", le serveur récupère à partir de la socket l'adresse IP du client qui communique avec lui, et l'ajoute dans un ensemble global (de type `Set<String>`). Si le client n'était pas enregistré, le serveur renvoie "OK", s'il l'était déjà il renvoie "ERROR"
- Pour "UNREGISTER", le serveur récupère à partir de la socket l'adresse IP du client qui communique avec lui, et la retire de l'ensemble. Si le client était enregistré, le serveur renvoie "OK", s'il ne l'était pas il renvoie "ERROR"
- Pour "LIST" le serveur renvoie :

```
BEGIN
ip1
...
ipn
END
```

où les ip_i sont les adresses IP contenue dans l'ensemble.
- Pour tout autre chaîne envoyée par le client, le serveur renvoie "ERROR".

Questions

1. (1 point) Compléter le code du constructeur `TCPRegistryConnection` (cf. ①). Ce dernier utilise les paramètres `s` et `r` pour initialiser les attributs `out`, `in`, `toClient` et `registry`.
2. (2 points) Compléter le code de la méthode `add` (cf. ②) et `remove` (cf. ③). La méthode `add` (resp. `remove`) ajoute (resp. retire) l'adresse `ip` à l'ensemble `registry` et renvoie `true` si l'élément était absent (resp. présent), et `false` s'il était présent (resp. absent). De plus, les méthodes doivent être sûres du point de vue de l'accès concurrent (1 point).
3. (1 point) Compléter le code de la méthode `getClientIP` (cf. ④). Cette dernière renvoie l'adresse IP du client connecté à la socket `toClient` sous forme d'une chaîne de caractères.
4. (3 points) Compléter le code de la méthode `run` (cf. ⑤). Cette dernière lit une ligne de texte sur `in` et implémente le protocole décrit. Dans le cas de "LIST", le code provoquant l'envoi de la liste des IPs devra être sûr du point de vue de la concurrence.

<pre> class TCPRegistryConnection extends Thread { private PrintStream out; private BufferedReader in; private Socket toClient; private Set<String> registry; TCPRegistryConnection(Socket s, Set<String> r) throws IOException { ① } boolean add(String ip) { ② } boolean remove(String ip) { ③ } String getClientIP() { ④ } public void run() { try { ⑤ } catch (IOException e) { } } } </pre>	<pre> public class TCPRegistryServer { private ServerSocket serverSocket; private Set<String> registry; public TCPRegistryServer() { registry = new HashSet<String>(); }; public void start(int port) throws IOException { if (serverSocket == null) { serverSocket = new ServerSocket(); } serverSocket.bind(new InetSocketAddress(port)); } public void mainLoop() { while (true) { try { Socket s = serverSocket.accept(); TCPRegistryConnection c = new TCPRegistryConnection(s, registry); c.start(); } catch (IOException e) { return; } } } } </pre>
---	---

FIGURE 1 – Code partiel du Serveur d'annuaire

Réponse: Voir le fichier TCPRegistryServer.java

2.2 UDP (5 points)

On suppose que le serveur possède un morceau de code UDP qui lui permet de diffuser son adresse IP via un datagramme UDP contenant : "REGISTRY:1.2.3.4" (où 1.2.3.4 est l'adresse IP du serveur). Les client se mettent en écoute sur le port 10000 et essayent, pendant une seconde de recevoir un datagramme sur ce port. Si au bout de 10 essais d'une seconde aucun datagramme n'arrive, les client termine. Si un datagramme arrive, alors le client le decode, récupère l'adresse IP contenue dans le message et appelle la méthode launchClient(String ip) en lui passant l'IP en paramètre. Le code du client est donné ci-dessous :

```

public void launchClient(String ip) { ... }

public void waitForServer() {
    try {
        DatagramSocket serverSocket = new DatagramSocket(null);
        ⑥
        while (true) {
            try {
                ⑦
            } catch (SocketTimeoutException e) {
                ⑧
            }
        }
    }
}

```

```
} catch (Exception e) {}  
}
```

Questions

- (4 points) Compléter le code du client UDP aux endroits indiqués
 - (1.5 point) place `serverSocket` en mode écoute sur le port `10000` et lui définit un *timeout* de `1000ms`, initialise un compteur entier à `10` et un `DatagramPacket` de taille suffisante pour recevoir le message.
 - (2 points) reçoit le datagramme, extrait la chaîne reçue, extrait de la chaîne l'adresse IP, appelle `launchClient` et termine la méthode.
 - (0.5 point) décrémente le compteur. S'il vaut `0`, termine la méthode.
- (1 point) A t'on la garantie que si un client quitte après 10 essais infructueux, c'est par ce qu'il n'y a pas de serveur ? (justifier brièvement).

Réponse: Voir le fichier `UDPDiscoverRegistry.java` pour le code. Pour la question 2 : On n'a aucune garantie. En effet comme on est en UDP, il est possible qu'il y ait un serveur mais que ses 10 réponses aient été perdues.

3 JSP Changement de mot de passe (5 points)

On souhaite réaliser un petit formulaire permettant à un utilisateur de changer de mot de passe. On dispose pour cela :

- D'une page `change.html` contenant quatre éléments `input` dont les noms sont `login`, `old`, `new1`, `new2` et qui correspondent respectivement au login de l'utilisateur, son ancien mot de passe, son nouveau mot de passe et son mot nouveau mot de passé entré une seconde fois pour confirmation
- Un *servlet* `UpdateDBServlet` servant de contrôleur
- Un modèle `PasswordDB` servant de modèle
- Une page `success.jsp`, qu'on ne détaille pas dans cet exercice.

Questions

- (2.5 points) On considère le code ci-dessous pour le modèle :

```
class PasswordDB {  
    private Connection conn;  
    /* Effectue la connexion à la base */  
    LogDB() throws SQLException, ClassNotFoundException { ... }  
  
    boolean updatePW(String login, String old, String new) { ⑨ }  
}  
associé à une table :  
CREATE TABLE USER(login VARCHAR(20) PRIMARY KEY, password VARCHAR(20));
```

Donner le code de la méthode `updatePW` (cf. ⑨) pour qu'elle mette à jour le mot de passe dans la table `USER`. Si l'utilisateur `login` n'existe pas, si `old` n'est pas le mot de passe actuellement stocké ou si le code SQL lève une exception, alors la méthode renvoie `false`. Sinon, la méthode effectue la mise à jour et renvoie `true`. On rappelle que la syntaxe pour une mise à jour est :

UPDATE *t* SET *u*₁, . . . , *u*_{*n*} WHERE *c*

où *t* est un nom de table, *c* est une condition et les *u*_{*i*} des mises à jours de la forme *a* = *e* ou *a* est un attribut et *e* une expression.

Réponse:

```

class PasswordDB {
    private Connection conn;
    /* Effectue la connexion à la base */
    LogDB() throws SQLException, ClassNotFoundException { ... }

    boolean updatePW(String login, String old, String new) {
        try {
            Statement stmt = conn.createStatement();
            int numUpdates = stmt.executeUpdate ("UPDATE USER SET password =
            ''' + new + ''' WHERE login = ''' + login + ''' AND password = ''' +
            old + '''");
            return numUpdates == 1;
            /* on a mis à jour exactement une
            ligne, lorsque le login existe et que le password vaut
            l'ancienne valeur */;
        } catch (Exception e) {
            return false;
        }
    }
}

```

2. (2.5 points) Donner le corps d'une méthode

`doGet(HttpServletRequest req, HttpServletResponse res)`

qui effectue les actions suivantes :

- récupère les paramètres GET login, old, new1, new2.
- si new1 est différent de new2 (mauvaise confirmation), la méthode effectue une redirection HTTP vers `change.html`
- récupère dans la session un objet PasswordDB et le crée au besoin
- appelle la méthode `update` avec les bons arguments
- si la méthode `update` renvoie `false` effectue une redirection HTTP vers `change.html`. Si elle renvoie `true` effectue une redirection interne vers `success.jsp`

Remarque : vous n'avez pas à gérer les exceptions dans votre code (*i.e.* pas besoin de mettre un try/catch global).

Réponse:

```

void doGet(HttpServletRequest req, HttpServletResponse res) {
    String login = req.getParameter("login");
    login = (login == null) ? "" : login; /* si absent, chaine vide */

    String old = req.getParameter("old");
    old = (old == null) ? "" : old;

    String new1 = req.getParameter("new1");
    new1 = (new1 == null) ? "" : new1;

    String new2 = req.getParameter("new2");
    new2 = (new2 == null) ? "" : new2;

    if (! new1.equals(new2) ) {
        res.sendRedirect("change.html");
        return;
    }

    PasswordDB db = (PasswordDB) req.getSession().getAttribute("db");

```

```
    if (db == null) {
        db = new PasswordDB();
        req.getSession().setAttribute("db", db);
    }

    if (db.update(login, old, new1)) {
        RequestDispatcher rd = res.getRequestDispatcher("success.jsp");
        rd.forward();
    } else {
        res.sendRedirect("change.html");
    }
}
```

Aide-mémoire

Général

En Java, l'opérateur `+` sert à concaténer des chaînes de caractères (ou d'autres objets) et ces derniers sont convertis automatiquement en chaînes.

Integer.parseInt(s) Renvoie l'entier représenté par la chaîne de caractères `s`. Si ce n'est pas une chaîne valide, lève l'exception `NumberFormatException`

Thread Classe abstraite (qu'il faut étendre) représentant un *thread* (sous processus exécuté de manière concurrente). La classe héritant de `Thread` doit implémenter la méthode `.run()` qui contient le code à exécuter en parallèle. Un *thread* est démarré avec `.start()`.

new Date() crée un objet `Date` initialisé avec l'heure courante. La méthode `.toString()` renvoie une chaîne de caractères représentant l'heure.

PrintWriter Classe permettant d'écrire dans un buffer (fichier, socket, terminal, ...). Possède les méthodes `.print(s)` et `.println(s)` pour écrire une chaîne. On peut créer un `PrintWriter` à partir d'un `OutputStream o` :

```
PrintWriter p = new PrintWriter(new BufferedWriter(o));
```

BufferedReader Classe permettant de lire dans un buffer sous-jacent (fichier, socket, terminal, ...). Possède la méthode `.readLine()` pour lire une chaîne jusqu'au prochain « `\n` » (qui est supprimé de la chaîne renvoyée). On peut créer un `BufferedReader` à partir d'un `InputStream i` avec :

```
BufferedReader b = new BufferedReader(new InputStreamReader(i));
```

UDP

classe DatagramSocket Représente un *socket* UDP sur lequel des `DatagramPacket` peuvent être envoyés ou reçus. Méthodes :

- .bind(InetAddress addr)** Place la socket en mode écoute sur l'adresse de socket donnée. Peut créer une adresse d'écoute de la manière suivante : `new InetAddress(port)`; où `port` est un numéro de port sur lequel écouté. Une telle adresse est associée par défaut à tous les périphériques réseau de la machine.
- .connect(InetAddress addr, int port)** place la socket en mode connexion sur l'adresse donnée et le port donné.
- .send(DatagramPacket p)** Envoie le message contenu dans le datagramme `p` à l'autre bout de la socket.
- .receive(DatagramPacket p)** Reçoit un message sur la socket et le stocke dans le datagramme `p`.

classe DatagramPacket Représente un paquet UDP utilisé pour envoyer ou recevoir des messages. Méthodes :

- new DatagramPacket(buffer, len)** Crée un nouveau `DatagramPacket`, utilisant `len` cases du tableau java `buffer` sous-jacent (déclaré avec `byte[] buffer`;). Les méthodes `.send` et `.receive` de la classe `DatagramSocket` écrivent ou lisent le message envoyé ou reçu dans de tels objets.
- .getOffset()** Renvoie l'indice dans le tableau sous-jacent à partir duquel la donnée est écrite.
- .getLength()** Renvoie la longueur de la donnée reçue ou écrite.

TCP

classe Socket Représente une connexion entre deux machines (un client et le serveur). Méthodes :

- .getOutputStream()** renvoie un `OutputStream` de sortie, connecté au flux d'entrée de la machine de l'autre côté de la socket.
- .getInputStream()** renvoie un `InputStream` d'entrée, connecté au flux de sortie de la machine de l'autre côté de la socket.
- .close()** Permet de fermer la socket
- .getInetAddress()** renvoie un objet `InetAddress` représentant l'adresse IP du client connecté à la socket. Peut être converti en chaîne avec `.toString()`.

JSP

JDBC

classe Connexion Représente une connexion à la base de données. Possède une méthode `.createStatement()` pour créer un objet `Statement`.

classe Statement Représente une requête de base de données. Méthodes :

.executeQuery(q) exécute la requête `q` (typiquement un `SELECT ...`) et renvoie le résultat sous la forme d'un `ResultSet`.

.executeUpdate(q) exécute la mise à jour `q` (exemple : `CREATE TABLE, INSERT, UPDATE, ...`). Renvoie le nombre de lignes modifiées comme un entier.

classe ResultSet Représente un résultat de requête, initialisé à une position fictive avant le premier résultat. Méthodes :

.next() positionne le curseur interne sur le prochain résultat et renvoie `true` si c'est possible et `false` s'il ne reste plus de résultat.

.getXXX(col) famille de méthodes qui renvoient le contenu de la colonne `col`. `XXX` est un type Java comme `Int`, `String`, `Boolean`. La colonne peut être référencée par son nom ou par sa position à partir de 1.

Servlet

classe HttpServlet Classe abstraite à étendre. Les classes en dérivant doivent implémenter la méthode `.doXxx` pour traiter la requête HTTP de type `Xxx` (par exemple `Get`, `Post`, ...). Leur prototype est :

```
protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException
```

classe HttpServletRequest classe représentant la requête HTTP qui visite le servlet. Méthodes :

.getParameter(p) Renvoie sous forme de chaîne de caractères la valeur du paramètre de nom `p` passé par un formulaire dans la requête HTTP.

.getAttribute(a) Renvoie un `Object` représentant l'objet stocké dans la requête avec le nom `a`. Renvoie `null` si l'objet n'est pas trouvé. Il faut caster l'objet vers le bon type.

.setAttribute(a, o) stocke l'objet `o` dans la requête avec le nom `a`.

.getSession() Renvoie la session HTTP. L'objet *session* possède aussi deux méthodes `.getAttribute` et `.setAttribute` identiques au précédente mais qui stocke dans la session et non pas dans la requête.

.getRequestDispatcher(url) Renvoie un objet `RequestDispatcher` permettant de charger la page dont l'URL est donnée. On utilise la méthode `.forward(request, response)` du `RequestDispatcher` pour rediriger.

classe HttpServletResponse classe représentant la réponse HTTP. Méthodes :

.sendRedirect(url) Effectue une redirection HTTP vers l'URL donnée.

Balises JSTL

<c:out value="..." /> Écrit dans la page le résultat de l'expression contenue dans `value`

<c:forEach var="i" items="..."> Effectue une boucle sur tous les éléments de la collection Java résultant de l'évaluation de `value`. L'indice de boucle est donné par `var`

<c:choose> Dénote une conditionnelle multiple. Il doit contenir un nombre arbitraire de `c:when` et éventuellement un `c:otherwise` final.

<c:when test="..."> Dénote un cas qui est choisi si l'expression contenue dans `test` est vraie.

<c:otherwise> Représente le cas par défaut pour un ensemble de choix

<c:if test="..."> Permet d'effectuer un test