Exercice 1 - Les bases de JDBC

Partie 1 : Implémentation de `registerClient`

L'objectif est d'insérer un nouveau client dans la table `CLIENT` en attribuant un **ID unique** et en respectant les contraintes définies dans la table.

Rappel de la table CLIENT :

```
CREATE TABLE client (
    idc INT PRIMARY KEY,
    nom VARCHAR2(10) NOT NULL,
    age INT NOT NULL CHECK (0<=age AND age<120),
    avoir INT DEFAULT 2000 NOT NULL CHECK (0<=avoir AND avoir<=2000)
);
```

Étape 1 : Récupérer le prochain ID disponible

Pour attribuer un ID unique :

1. On utilise `SELECT MAX(idc) + 1` pour obtenir le prochain ID.
2. On gère le cas où la table est vide : l'ID commence à 1 grâce à `COALESCE`.

Étape 2 : Insérer le client avec `PreparedStatement`

- La requête SQL pour l'insertion est **paramétrée** pour éviter les injections SQL.
- On utilise `try-with-resources` pour assurer la fermeture automatique des ressources.

Solution complète :

```java
public int registerClient(String name, int age) throws SQLException {
    // Étape 1 : SQL pour récupérer le prochain ID
    String getNextIdSql = "SELECT COALESCE(MAX(idc) + 1, 1) FROM client";

    // Étape 2 : SQL pour insérer un nouveau client
    String insertClientSql = "INSERT INTO client(idc, nom, age) VALUES (?, ?, ?)";

    int newId = 1; // Par défaut, on commence à 1

    // Étape 3 : Récupérer le prochain ID disponible
    try (PreparedStatement getIdStmt = connection.prepareStatement(getNextIdSql);
         ResultSet rs = getIdStmt.executeQuery()) {
        if (rs.next()) {
            newId = rs.getInt(1); // Lire l'ID calculé
        }
    }

    // Étape 4 : Insérer le client dans la base de données
    try (PreparedStatement insertStmt = connection.prepareStatement(insertClientSql)
        insertStmt.setInt(1, newId);   // ID du client
        insertStmt.setString(2, name); // Nom
        insertStmt.setInt(3, age);     // Âge
        insertStmt.executeUpdate();    // Exécuter l'insertion
    }

    return newId; // Retourner l'ID généré
}
```

Explication de chaque étape :

1. **SQL pour l'ID :**
   - La requête utilise `MAX(idc) + 1` pour déterminer l'ID suivant.
   - `COALESCE(..., 1)` garantit que si la table est vide, l'ID sera 1.
2. **`PreparedStatement` :**
   - Utilise des paramètres **?** pour éviter les injections SQL.
   - On utilise `setInt` et `setString` pour passer les valeurs.
3. **Gestion des ressources :**
   - Le bloc `try-with-resources` assure que les objets `PreparedStatement` et `ResultSet` sont correctement fermés, même en cas d'erreur.
4. **Retour de l'ID :**
   - L'ID généré est retourné pour vérification ou affichage ultérieur.

Tests à effectuer :

1. **Insertion réussie** : Ajouter un client valide.
   - Exemple : `registerClient("Alice", 25)`
     Résultat attendu : Un ID est généré et inséré dans la table.
2. **Erreur pour nom trop long** :

- Exemple : `registerClient("LongNameExceed", 30)`
  Résultat attendu : Erreur SQL (nom limité à 10 caractères).

3. **Erreur pour âge invalide** :
   - Exemple : `registerClient("Bob", 150)`
     Résultat attendu : Erreur SQL (CHECK sur l'âge).

4. **Vérification de l'avoir** :
   - Après insertion, l'avoir doit être initialisé automatiquement à `2000`.

Exemple de test :

```
public static void main(String[] args) {
    try {
        VacationBookingApp app = new VacationBookingApp();
        int clientId = app.registerClient("JohnDoe", 28);
        System.out.println("Client enregistré avec l'ID : " + clientId);
    } catch (SQLException e) {
        System.err.println("Erreur : " + e.getMessage());
    }
}
```

Sortie attendue :

```
Client enregistré avec l'ID : 1
```

Partie 2 : Amélioration de `handleClientLogin`

L'objectif est de vérifier l'existence d'un client via son ID et d'afficher ses informations.

Récapitulatif des tâches :

1. Créer une méthode `verifyClient` pour récupérer les informations d'un client via son ID.
2. Lever une exception `ClientNotFoundException` si l'ID est invalide.

Solution pour `verifyClient` :

```
public ClientInfo verifyClient(int clientId) throws SQLException, ClientNotFoundExce
    String sql = "SELECT nom, age, avoir FROM client WHERE idc = ?";

    try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
        pstmt.setInt(1, clientId); // Paramètre ID

        try (ResultSet rs = pstmt.executeQuery()) {
            if (rs.next()) {
                return new ClientInfo(
                    rs.getString("nom"),
                    rs.getInt("age"),
                    rs.getInt("avoir")
                );
            } else {
                throw new ClientNotFoundException(clientId);
            }
        }
    }
}
```

Gestion de l'exception `ClientNotFoundException` :

```
public class ClientNotFoundException extends Exception {
    public ClientNotFoundException(int clientId) {
        super("Client avec l'ID " + clientId + " non trouvé.");
    }
}
```

Appel depuis `handleClientLogin` :

```
private void handleClientLogin() {
    System.out.print("Entrez votre ID client : ");
    try {
        int clientId = Integer.parseInt(scanner.nextLine());
        ClientInfo client = app.verifyClient(clientId);

        System.out.println("Bienvenue " + client.name());
        System.out.println("Votre crédit actuel est de : " + client.credit() + "€");
    } catch (NumberFormatException e) {
        System.out.println("ID client invalide.");
    } catch (ClientNotFoundException e) {
        System.out.println(e.getMessage());
    } catch (SQLException e) {
        System.out.println("Erreur technique : " + e.getMessage());
    }
}
```

Tests à effectuer :

1. **ID existant :**

- Entrée : 1
  Sortie attendue :

```
Bienvenue JohnDoe
Votre crédit actuel est de : 2000€
```

2. **ID inexistant :**
   - Entrée : 99
     Sortie attendue :

```
Client avec l'ID 99 non trouvé.
```

3. **ID invalide :**
   - Entrée : abc
     Sortie attendue :

```
ID client invalide.
```

Exercice 2 - Requêtes et transactions

Partie 1 : Implémentation de bookVacation

L'objectif est de permettre à un client de réserver un séjour dans un village en :

1. **Trouvant le village le plus cher dans une ville donnée**.
2. **Vérifiant l'avoir disponible du client**.
3. **Insérant un nouveau séjour**.
4. **Décrémentant l'avoir du client**.

Étape 1 : SQL pour trouver le village le plus cher

On utilise une requête avec `ORDER BY` pour trier les villages en fonction du prix et `FETCH FIRST 1 ROW ONLY` pour récupérer uniquement le plus cher.

Étape 2 : SQL pour vérifier l'avoir du client

On vérifie que l'avoir (`credit`) du client est suffisant pour couvrir le prix du village.

Étape 3 : Insérer le séjour

On insère un nouveau séjour dans la table `sejour` avec un ID unique.

Étape 4 : Mettre à jour l'avoir

On décrémente l'avoir du client du montant correspondant au prix du village.

Solution complète :

```java
public BookingResult bookVacation(int clientId, String city, int day) throws SQLExce
    // Requêtes SQL
    String findVillageSql = """
        SELECT idv, prix, activite
        FROM village
        WHERE ville = ?
        ORDER BY prix DESC
        FETCH FIRST 1 ROW ONLY""";

    String checkCreditSql = "SELECT avoir FROM client WHERE idc = ?";
    String insertSejourSql = "INSERT INTO sejour(ids, idc, idv, jour) VALUES (?, ?,
    String updateCreditSql = "UPDATE client SET avoir = avoir - ? WHERE idc = ?";

    // Désactiver l'autocommit pour gérer la transaction manuellement
    connection.setAutoCommit(false);

    try {
        // Étape 1 : Trouver le village le plus cher
        int villageId, price;
        String activity;
        try (PreparedStatement findVillageStmt = connection.prepareStatement(findVil
            findVillageStmt.setString(1, city);
            ResultSet villageRs = findVillageStmt.executeQuery();

            if (!villageRs.next()) {
                throw new SQLException("Aucun village trouvé dans la ville : " + cit
            }

            villageId = villageRs.getInt("idv");
            price = villageRs.getInt("prix");
            activity = villageRs.getString("activite");
        }

        // Étape 2 : Vérifier l'avoir du client
        int currentCredit;
        try (PreparedStatement checkCreditStmt = connection.prepareStatement(checkCr
            checkCreditStmt.setInt(1, clientId);
            ResultSet creditRs = checkCreditStmt.executeQuery();

            if (!creditRs.next()) {
                throw new SQLException("Client introuvable.");
            }

            currentCredit = creditRs.getInt(1);
            if (currentCredit < price) {
                throw new SQLException("Crédit insuffisant pour réserver.");
            }
        }

        // Étape 3 : Insérer le séjour
        int newSejourId = getNextBookingId(); // Génération d'un nouvel ID
        try (PreparedStatement insertSejourStmt = connection.prepareStatement(insert
            insertSejourStmt.setInt(1, newSejourId);
            insertSejourStmt.setInt(2, clientId);
            insertSejourStmt.setInt(3, villageId);
            insertSejourStmt.setInt(4, day);
            insertSejourStmt.executeUpdate();
        }
```

```
            // Étape 4 : Mettre à jour l'avoir
            try (PreparedStatement updateCreditStmt = connection.prepareStatement(update
                updateCreditStmt.setInt(1, price);
                updateCreditStmt.setInt(2, clientId);
                updateCreditStmt.executeUpdate();
            }

            // Commit de la transaction
            connection.commit();
            return new BookingResult(villageId, newSejourId, activity);

        } catch (SQLException e) {
            connection.rollback(); // Annuler la transaction en cas d'erreur
            throw e; // Renvoyer l'erreur
        } finally {
            connection.setAutoCommit(true); // Réactiver l'autocommit
        }
    }
```

Fonction utilitaire pour générer un ID de séjour :

```
    private int getNextBookingId() throws SQLException {
        String sql = "SELECT COALESCE(MAX(ids) + 1, 1) FROM sejour";

        try (PreparedStatement stmt = connection.prepareStatement(sql);
             ResultSet rs = stmt.executeQuery()) {
            if (rs.next()) {
                return rs.getInt(1);
            }
        }
        return 1; // Si la table est vide, retourner 1
    }
```

Explications de chaque étape :

1. **Trouver le village le plus cher** :
    - On utilise `ORDER BY prix DESC` pour trier par prix décroissant.
    - La clause `FETCH FIRST 1 ROW ONLY` permet de récupérer uniquement le village le plus cher.
2. **Vérifier l'avoir du client** :
    - On vérifie si l'avoir est suffisant avant de procéder à l'insertion.
3. **Insérer un nouveau séjour** :
    - La méthode `getNextBookingId()` garantit un ID unique en récupérant le maximum existant dans la table.
4. **Mettre à jour l'avoir** :
    - L'avoir du client est décrémenté par `avoir = avoir - ?`.
5. **Transaction** :
    - `setAutoCommit(false)` permet de désactiver l'autocommit pour exécuter plusieurs opérations dans une seule transaction.
    - En cas d'erreur, `connection.rollback()` annule toutes les opérations.

6. **Réactiver l'autocommit** :
    - Dans le bloc `finally`, on réactive l'autocommit pour ne pas affecter les autres opérations.

Exemple de test :

```
public static void main(String[] args) {
    try {
        VacationBookingApp app = new VacationBookingApp();
        BookingResult result = app.bookVacation(1, "Paris", 100);
        System.out.println("Séjour réservé avec succès :");
        System.out.println("ID Séjour : " + result.bookingId());
        System.out.println("Village ID : " + result.villageId());
        System.out.println("Activité : " + result.activity());
    } catch (SQLException e) {
        System.err.println("Erreur : " + e.getMessage());
    }
}
```

Scénarios de test :

1. **Client avec crédit suffisant** :
    - Entrée : `clientId = 1`, `ville = "Paris"`, `jour = 100`.
    - **Sortie attendue** :

      ```
      Séjour réservé avec succès :
      ID Séjour : 1
      Village ID : 2
      Activité : Musée
      ```

2. **Client avec crédit insuffisant** :
    - Entrée : `clientId = 2` (avoir = 50), `ville = "Nice"`, `jour = 101`.
    - **Sortie attendue** :

      ```
      Erreur : Crédit insuffisant pour réserver.
      ```

3. **Aucun village disponible** :
    - Entrée : `ville = "Lyon"` (aucun village enregistré).
    - **Sortie attendue** :

      ```
      Erreur : Aucun village trouvé dans la ville : Lyon
      ```

Passons maintenant à la **Partie 2 : Ajout des vérifications** de l'exercice 2. Cette partie introduit des méthodes de contrôle supplémentaires pour assurer que les réservations respectent les contraintes métier. Les vérifications incluent :

1. **Disponibilité d'un village** à une date donnée.
2. **Suffisance du crédit** du client pour effectuer la réservation.
3. **Génération d'un nouvel ID** pour les séjours.

2.1 Vérification de la disponibilité

L'objectif est de vérifier si un village a suffisamment de capacité pour accueillir un séjour à une date donnée.

Solution :

```
private boolean isVillageAvailable(int villageId, int day) throws SQLException {
    String sql = """
        SELECT v.capacite > (
            SELECT COUNT(*)
            FROM sejour s
            WHERE s.idv = v.idv
            AND s.jour = ?
        ) AS disponible
        FROM village v
        WHERE v.idv = ?""";

    try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
        pstmt.setInt(1, day);
        pstmt.setInt(2, villageId);

        try (ResultSet rs = pstmt.executeQuery()) {
            if (rs.next()) {
                return rs.getBoolean("disponible");
            }
        }
    }
    return false; // Si aucun résultat, considérer comme non disponible
}
```

Explication :

- La sous-requête compte le nombre de séjours existants pour un village donné à une date précise.
- On compare ce nombre à la capacité maximale du village pour déterminer s'il reste des places (`capacite > COUNT(*)`).
- On utilise un alias `disponible` pour récupérer directement le résultat sous forme de booléen.

2.2 Vérification de l'avoir

L'objectif est de s'assurer que le client dispose d'assez de crédit pour effectuer une réservation.

Solution :

```java
private boolean hasEnoughCredit(int clientId, int price) throws SQLException {
    String sql = "SELECT avoir >= ? AS enough_credit FROM client WHERE idc = ?";

    try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
        pstmt.setInt(1, price);
        pstmt.setInt(2, clientId);

        try (ResultSet rs = pstmt.executeQuery()) {
            if (rs.next()) {
                return rs.getBoolean("enough_credit");
            }
        }
    }
    return false; // Par défaut, crédit insuffisant
}
```

Explication :

- La requête vérifie si l'avoir du client est supérieur ou égal au prix du séjour.
- L'alias `enough_credit` permet de récupérer directement un booléen.
- La méthode retourne `false` si aucune ligne ne correspond (client introuvable).

2.3 Génération d'un nouvel ID de séjour

Pour garantir un **ID unique** lors de l'insertion dans la table `sejour`, on utilise `MAX` pour calculer l'ID suivant.

Solution :

```java
private int getNextBookingId() throws SQLException {
    String sql = "SELECT COALESCE(MAX(ids) + 1, 1) FROM sejour";

    try (PreparedStatement pstmt = connection.prepareStatement(sql);
         ResultSet rs = pstmt.executeQuery()) {
        if (rs.next()) {
            return rs.getInt(1); // Retourner le nouvel ID
        }
    }
    return 1; // Par défaut, si la table est vide
}
```

Explication :

- La fonction `COALESCE` retourne 1 si la table `sejour` est vide.
- `MAX(ids) + 1` garantit que le nouvel ID est supérieur au maximum existant.

Intégration des vérifications dans bookVacation

On intègre maintenant les trois nouvelles méthodes (`isVillageAvailable`, `hasEnoughCredit`, `getNextBookingId`) dans la méthode `bookVacation`.

Solution finale :

```
public BookingResult bookVacation(int clientId, String city, int day) throws SQLExce
    String findVillageSql = """
        SELECT idv, prix, activite
        FROM village
        WHERE ville = ?
        ORDER BY prix DESC
        FETCH FIRST 1 ROW ONLY""";

    String insertSejourSql = "INSERT INTO sejour(ids, idc, idv, jour) VALUES (?, ?,
    String updateCreditSql = "UPDATE client SET avoir = avoir - ? WHERE idc = ?";

    connection.setAutoCommit(false);

    try {
        // Étape 1 : Trouver le village le plus cher
        int villageId, price;
        String activity;

        try (PreparedStatement findVillageStmt = connection.prepareStatement(findVil
            findVillageStmt.setString(1, city);
            ResultSet villageRs = findVillageStmt.executeQuery();

            if (!villageRs.next()) {
                throw new SQLException("Aucun village trouvé dans la ville : " + cit
            }

            villageId = villageRs.getInt("idv");
            price = villageRs.getInt("prix");
            activity = villageRs.getString("activite");
        }

        // Étape 2 : Vérifier la disponibilité du village
        if (!isVillageAvailable(villageId, day)) {
            throw new SQLException("Le village est complet à cette date.");
        }

        // Étape 3 : Vérifier l'avoir du client
        if (!hasEnoughCredit(clientId, price)) {
            throw new SQLException("Crédit insuffisant pour réserver.");
        }

        // Étape 4 : Générer un nouvel ID et insérer le séjour
        int newSejourId = getNextBookingId();
        try (PreparedStatement insertSejourStmt = connection.prepareStatement(insert
            insertSejourStmt.setInt(1, newSejourId);
            insertSejourStmt.setInt(2, clientId);
            insertSejourStmt.setInt(3, villageId);
            insertSejourStmt.setInt(4, day);
            insertSejourStmt.executeUpdate();
        }

        // Étape 5 : Mettre à jour l'avoir
        try (PreparedStatement updateCreditStmt = connection.prepareStatement(update
            updateCreditStmt.setInt(1, price);
            updateCreditStmt.setInt(2, clientId);
            updateCreditStmt.executeUpdate();
        }

        // Commit de la transaction
```

```
            connection.commit();
            return new BookingResult(villageId, newSejourId, activity);

        } catch (SQLException e) {
            connection.rollback();
            throw e; // Relancer l'exception après rollback
        } finally {
            connection.setAutoCommit(true);
        }
    }
```

Tests à effectuer :

1. **Village disponible, crédit suffisant :**
    - Entrée : `clientId = 1`, `ville = "Paris"`, `jour = 100`.
    - **Résultat attendu :** Réservation réussie.
2. **Village complet :**
    - Entrée : `ville = "Nice"`, date où la capacité est atteinte.
    - **Résultat attendu :** Erreur : "Le village est complet à cette date."
3. **Crédit insuffisant :**
    - Entrée : `clientId = 2`, `ville = "Paris"`, `jour = 100`.
    - **Résultat attendu :** Erreur : "Crédit insuffisant pour réserver."
4. **Aucun village disponible :**
    - Entrée : `ville = "Lyon"`.
    - **Résultat attendu :** Erreur : "Aucun village trouvé dans la ville."

Exercice 3 - Gestion des résultats

Partie 1 : Implémentation de `getClientBookings`

L'objectif est de récupérer les réservations d'un client avec leurs détails en utilisant une **jointure** entre les tables `SEJOUR` et `VILLAGE`.

Requête SQL attendue :

On effectue une jointure entre :

- `SEJOUR` (qui contient les réservations) et
- `VILLAGE` (qui contient les détails des villages).

On filtre les résultats par l'ID client (`idc`) et on ordonne les séjours par la date (`jour`).

Solution :

```
public class VacationBookingApp {

    public record BookingDetails(
        int bookingId,
        String city,
        String activity,
        int day,
        int price
    ) { }

    public List<BookingDetails> getClientBookings(int clientId) throws SQLException
        String sql = """
            SELECT s.ids, v.ville, v.activite, s.jour, v.prix
            FROM sejour s
            JOIN village v ON s.idv = v.idv
            WHERE s.idc = ?
            ORDER BY s.jour""";

        List<BookingDetails> bookings = new ArrayList<>();

        try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
            pstmt.setInt(1, clientId);

            try (ResultSet rs = pstmt.executeQuery()) {
                while (rs.next()) {
                    bookings.add(new BookingDetails(
                        rs.getInt("ids"),
                        rs.getString("ville"),
                        rs.getString("activite"),
                        rs.getInt("jour"),
                        rs.getInt("prix")
                    ));
                }
            }
        }
        return bookings;
    }
}
```

Explications :

1. **Requête SQL** :
   - On sélectionne les informations des tables `SEJOUR` et `VILLAGE` via une **jointure interne** (`JOIN`).
   - La clause `WHERE` limite les résultats aux réservations du client spécifié.
   - Les résultats sont triés par la colonne `jour`.
2. **ResultSet** :
   - La méthode `rs.next()` parcourt chaque ligne des résultats.
   - On récupère les valeurs des colonnes avec `rs.getInt`, `rs.getString`, etc., en utilisant les noms des colonnes (`ids`, `ville`, etc.).
3. **Utilisation de `record`** :

- Le `record BookingDetails` est utilisé pour stocker chaque réservation de manière concise et immuable.

4. **Liste des résultats** :
    - Les réservations sont stockées dans une `ArrayList` de `BookingDetails` et renvoyées.

Amélioration de l'affichage dans `handleViewBookings`

La méthode suivante permet d'afficher les réservations d'un client dans un format **tabulaire** clair.

Solution :

```
private void handleViewBookings() {
    try {
        var bookings = app.getClientBookings(currentClientId);

        if (bookings.isEmpty()) {
            System.out.println("Vous n'avez aucune réservation.");
            return;
        }

        System.out.println("=======================================");
        System.out.printf("%-5s %-15s %-15s %-5s %-5s\n",
                          "ID", "VILLE", "ACTIVITÉ", "JOUR", "PRIX");
        System.out.println("=======================================");

        int total = 0;
        for (var booking : bookings) {
            System.out.printf("%-5d %-15s %-15s %-5d %-5d€\n",
                              booking.bookingId(),
                              booking.city(),
                              booking.activity(),
                              booking.day(),
                              booking.price());
            total += booking.price();
        }
        System.out.println("=======================================");
        System.out.println("Total des dépenses : " + total + "€");
        System.out.println("Nombre de séjours : " + bookings.size());

    } catch (SQLException e) {
        System.out.println("Erreur lors de la récupération des réservations : " + e.
    }
}
```

Explications :

1. **Formatage des colonnes** :
    - On utilise `String.format` avec des tailles fixes (`%-5s`, `%-15s`, etc.) pour aligner les colonnes.
    - Cela garantit un affichage lisible même si les valeurs varient.

2. **Calcul des statistiques** :
    - On calcule le **total des dépenses** en additionnant les prix des séjours.

- Le **nombre de séjours** correspond à la taille de la liste.

3. **Gestion des cas vides** :
    - Si aucune réservation n'est trouvée, un message est affiché.

Exemple d'affichage attendu :

```
=====================================
ID    VILLE           ACTIVITÉ        JOUR  PRIX
=====================================
1     Paris           Musée           120   50€
2     Nice            Plage           200   75€
3     Lyon            Parc Aventure   300   100€
=====================================
Total des dépenses : 225€
Nombre de séjours : 3
```

Partie 2 : Affichage des villages disponibles

L'objectif est de lister les villages sans séjour existant. Les employés peuvent également voir la **capacité** de chaque village.

Solution :

```
public List<VillageInfo> getAvailableVillages(boolean isEmployee) throws SQLExceptio
    String sql = """
        SELECT v.idv, v.ville, v.activite, v.prix %s
        FROM village v
        WHERE NOT EXISTS (
            SELECT 1
            FROM sejour s
            WHERE s.idv = v.idv
        )
        ORDER BY v.ville, v.prix DESC""".formatted(isEmployee ? ", v.capacite" : "")

    List<VillageInfo> villages = new ArrayList<>();

    try (PreparedStatement pstmt = connection.prepareStatement(sql);
         ResultSet rs = pstmt.executeQuery()) {
        while (rs.next()) {
            villages.add(new VillageInfo(
                rs.getInt("idv"),
                rs.getString("ville"),
                rs.getString("activite"),
                rs.getInt("prix"),
                isEmployee ? rs.getInt("capacite") : null
            ));
        }
    }
    return villages;
}
```

Explication :

- La clause `NOT EXISTS` filtre les villages qui n'ont **aucun séjour associé**.
- Pour les employés, la colonne `capacite` est incluse via un paramètre conditionnel.

Amélioration de l'affichage :

```
private void handleViewVillages() {
    try {
        var villages = app.getAvailableVillages(!isEmployee);

        if (villages.isEmpty()) {
            System.out.println("Aucun village disponible.");
            return;
        }

        System.out.println("========================================");
        if (isEmployee) {
            System.out.printf("%-15s %-15s %-5s %-10s\n", "VILLE", "ACTIVITÉ", "PRIX
        } else {
            System.out.printf("%-15s %-15s %-5s\n", "VILLE", "ACTIVITÉ", "PRIX");
        }
        System.out.println("========================================");

        for (var village : villages) {
            if (isEmployee) {
                System.out.printf("%-15s %-15s %-5d %-10d\n",
                                  village.city(), village.activity(), village.price(
            } else {
                System.out.printf("%-15s %-15s %-5d\n",
                                  village.city(), village.activity(), village.price(
            }
        }

    } catch (SQLException e) {
        System.out.println("Erreur lors de la récupération des villages : " + e.getM
    }
}
```

Exemple d'affichage :

Pour un **client** :

```
========================================
VILLE           ACTIVITÉ        PRIX
========================================
Paris           Musée           50€
Nice            Plage           75€
========================================
```

Pour un **employé** :

```
========================================
VILLE           ACTIVITÉ        PRIX    CAPACITÉ
========================================
Paris           Musée           50€     100
Nice            Plage           75€     250
========================================
```

Exercice 4 - Droits d'accès et sécurité

Partie 1 : Gestion des droits d'accès

L'objectif est de créer un système de connexion sécurisé avec une gestion des rôles et des permissions.

1.1 Création d'une classe utilisateur

On définit la structure de l'utilisateur (User) avec son rôle et ses permissions.

```java
public class User {
    private final int id;
    private final String username;
    private final UserRole role;
    private final Set<Permission> permissions;

    public User(int id, String username, UserRole role, Set<Permission> permissions)
        this.id = id;
        this.username = username;
        this.role = role;
        this.permissions = permissions;
    }

    public int getId() { return id; }
    public String getUsername() { return username; }
    public UserRole getRole() { return role; }
    public boolean hasPermission(Permission permission) {
        return permissions.contains(permission);
    }
}

public enum UserRole {
    CLIENT,
    EMPLOYEE
}

public enum Permission {
    VIEW_VILLAGES,
    BOOK_VACATION,
    VIEW_OWN_BOOKINGS,
    CREATE_VILLAGE,
    MODIFY_VILLAGE,
    VIEW_ALL_BOOKINGS,
    CLEANUP_BOOKINGS
}
```

1.2 Implémentation du gestionnaire de session

Le `SessionManager` gère les sessions utilisateurs, l'authentification et la validation des permissions.

Solution :

```
public class SessionManager {
    private User currentUser;

    public void login(String username, String password) throws AuthenticationExcepti
        String sql = """
            SELECT idu, role
            FROM users
            WHERE username = ? AND password = ?""";

        try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
            pstmt.setString(1, username);
            pstmt.setString(2, password);

            try (ResultSet rs = pstmt.executeQuery()) {
                if (rs.next()) {
                    int userId = rs.getInt("idu");
                    UserRole role = UserRole.valueOf(rs.getString("role"));
                    Set<Permission> permissions = loadPermissions(role);

                    currentUser = new User(userId, username, role, permissions);
                    System.out.println("Connexion réussie. Bienvenue " + username);
                } else {
                    throw new AuthenticationException("Nom d'utilisateur ou mot de p
                }
            }
        }
    }

    private Set<Permission> loadPermissions(UserRole role) {
        Set<Permission> permissions = new HashSet<>();
        switch (role) {
            case CLIENT -> {
                permissions.add(Permission.VIEW_VILLAGES);
                permissions.add(Permission.BOOK_VACATION);
                permissions.add(Permission.VIEW_OWN_BOOKINGS);
            }
            case EMPLOYEE -> {
                permissions.addAll(Arrays.asList(Permission.values()));
            }
        }
        return permissions;
    }

    public boolean hasPermission(Permission permission) {
        return currentUser != null && currentUser.hasPermission(permission);
    }

    public void validateAccess(Permission permission) throws AccessDeniedException {
        if (!hasPermission(permission)) {
            throw new AccessDeniedException("Accès refusé : vous n'avez pas la permi
        }
    }

    public User getCurrentUser() {
        return currentUser;
    }
}
```

Explications :

1. **Authentification :**
   - Vérifie les identifiants (`username` et `password`) dans la table `users`.
   - Charge les permissions associées au rôle (`CLIENT` ou `EMPLOYEE`).
2. **Permissions :**
   - Les clients ont des permissions limitées (`VIEW_VILLAGES`, `BOOK_VACATION`, etc.).
   - Les employés disposent de toutes les permissions.
3. **Validation :**
   - `validateAccess` vérifie si l'utilisateur courant possède la permission requise.
   - Si ce n'est pas le cas, une exception `AccessDeniedException` est levée.

1.3 Sécurisation des accès

Appel de la validation des permissions :

Par exemple, dans la méthode `bookVacation` :

```
public BookingResult bookVacation(int clientId, String city, int day)
        throws SQLException, AccessDeniedException {
    sessionManager.validateAccess(Permission.BOOK_VACATION);
    // Logique de réservation ici...
}
```

Partie 2 : Sécurisation contre les injections SQL

L'objectif est de sécuriser toutes les entrées utilisateur pour prévenir les attaques SQL injection.

2.1 Validation des entrées utilisateur

On utilise une classe `InputValidator` pour valider les entrées avant de les traiter.

```
public class InputValidator {
    public static void validateName(String name) throws ValidationException {
        if (name == null || !name.matches("[A-Za-z\\s]{1,10}")) {
            throw new ValidationException("Le nom doit contenir uniquement des lettr
        }
    }

    public static void validateCity(String city) throws ValidationException {
        if (city == null || !city.matches("[A-Za-z\\s]{1,20}")) {
            throw new ValidationException("La ville est invalide.");
        }
    }

    public static void validateDay(int day) throws ValidationException {
        if (day <= 0 || day > 365) {
            throw new ValidationException("Le jour doit être compris entre 1 et 365.
        }
    }
}
```

2.2 Utilisation de PreparedStatement

On remplace toutes les requêtes dynamiques par des **requêtes paramétrées** pour éviter les injections SQL.

Exemple de requête sécurisée pour récupérer les villages disponibles :

```
String sql = "SELECT idv, ville, activite, prix FROM village WHERE ville = ?";
try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
    pstmt.setString(1, city);
    try (ResultSet rs = pstmt.executeQuery()) {
        while (rs.next()) {
            // Logique pour récupérer les résultats
        }
    }
}
```

Partie 3 : Création des vues SQL

Pour sécuriser davantage les accès, on crée des **vues SQL** pour restreindre les données visibles selon le rôle de l'utilisateur.

Exemple de vue pour les clients :

```
CREATE VIEW client_villages AS
SELECT idv, ville, activite, prix
FROM village;
```

Exemple de vue pour les employés (avec capacité) :

```sql
CREATE VIEW employee_villages AS
SELECT idv, ville, activite, prix, capacite
FROM village;
```

Tests à effectuer :

1. **Connexion avec un client valide** :
   - Tester les permissions disponibles (`BOOK_VACATION`, `VIEW_VILLAGES`).
2. **Connexion avec un employé** :
   - Accéder aux fonctionnalités avancées comme `CLEANUP_BOOKINGS`.
3. **Accès refusé** :
   - Un client tente d'accéder à une fonction réservée aux employés.
4. **Validation des entrées** :
   - Nom contenant des caractères spéciaux → erreur.
   - Jour hors plage (0 ou > 365) → erreur.
5. **Test d'injection SQL** :
   - Entrée : `city = "Paris' OR '1'='1"`
     Résultat attendu : Aucune injection possible grâce à `PreparedStatement`.

Exercice 5 - Logging et gestion des erreurs

Partie 1 : Configuration et mise en place du système de logging

On utilise **Java Util Logging** (`java.util.logging`) pour mettre en place un système de log.

1.1 Configuration du logger

Classe de configuration :

```java
import java.io.IOException;
import java.util.logging.*;

public class LogConfig {
    public static void initializeLogging() {
        try {
            // Créer le gestionnaire de log pour les fichiers
            FileHandler fileHandler = new FileHandler("app.log", true);
            fileHandler.setFormatter(new SimpleFormatter());

            // Créer le gestionnaire pour la console
            ConsoleHandler consoleHandler = new ConsoleHandler();
            consoleHandler.setFormatter(new SimpleFormatter());

            // Obtenir le logger par défaut
            Logger logger = Logger.getLogger("");
            logger.setLevel(Level.ALL);
            logger.addHandler(fileHandler);
            logger.addHandler(consoleHandler);

            // Désactiver les logs par défaut du gestionnaire root
            logger.setUseParentHandlers(false);

        } catch (IOException e) {
            System.err.println("Erreur d'initialisation du logging : " + e.getMessag
        }
    }
}
```

1.2 Création d'une classe utilitaire pour les logs

Pour uniformiser l'enregistrement des logs, on définit des méthodes utilitaires.

```
import java.util.logging.Level;
import java.util.logging.Logger;

public class AppLogger {
    private static final Logger logger = Logger.getLogger(AppLogger.class.getName())

    public static void logBusinessEvent(String message, Object... params) {
        logger.log(Level.INFO, message, params);
    }

    public static void logSecurityEvent(String message, Object... params) {
        logger.log(Level.WARNING, "Sécurité : " + message, params);
    }

    public static void logTechnicalError(String message, Throwable error) {
        logger.log(Level.SEVERE, message, error);
    }

    public static void logAudit(String action, String user, String details) {
        logger.log(Level.INFO, "Audit : Action={0}, Utilisateur={1}, Détails={2}",
                   new Object[]{action, user, details});
    }
}
```

Exemple d'utilisation :

```
public void bookVacation(int clientId, String city, int day) {
    try {
        AppLogger.logBusinessEvent("Réservation demandée pour ClientID={0}, Ville={1
                                   clientId, city, day);

        // Logique métier...
        AppLogger.logBusinessEvent("Réservation confirmée pour ClientID={0}", client
    } catch (Exception e) {
        AppLogger.logTechnicalError("Erreur lors de la réservation pour ClientID=" +
    }
}
```

Partie 2 : Création d'une hiérarchie d'exceptions métier

On définit une hiérarchie d'exceptions pour différencier les erreurs techniques et les erreurs métier.

2.1 Définition des exceptions de base

```
public class VacationBookingException extends Exception {
    public VacationBookingException(String message) {
        super(message);
    }
}

public class BusinessException extends VacationBookingException {
    public BusinessException(String message) {
        super(message);
    }
}

public class TechnicalException extends VacationBookingException {
    public TechnicalException(String message, Throwable cause) {
        super(message);
        initCause(cause);
    }
}
```

2.2 Exceptions spécifiques

On définit des exceptions pour des cas métiers précis.

```
public class InsufficientCreditException extends BusinessException {
    public InsufficientCreditException(int clientId, int required, int available) {
        super("Client " + clientId + " : crédit insuffisant (Requis=" + required
              + ", Disponible=" + available + ")");
    }
}

public class VillageFullException extends BusinessException {
    public VillageFullException(int villageId) {
        super("Le village " + villageId + " est complet.");
    }
}
```

Partie 3 : Gestion centralisée des erreurs

On centralise la gestion des erreurs pour les traiter de manière uniforme.

Classe ErrorHandler :

```
import java.util.logging.Level;

public class ErrorHandler {
    public static void handleException(VacationBookingException e) {
        if (e instanceof BusinessException) {
            AppLogger.logBusinessEvent("Erreur métier : " + e.getMessage());
            System.out.println("Erreur : " + e.getMessage());
        } else if (e instanceof TechnicalException) {
            AppLogger.logTechnicalError("Erreur technique : ", e);
            System.out.println("Une erreur technique est survenue. Veuillez réessaye
        }
    }
}
```

Exemple d'utilisation :

Dans bookVacation :

```
public BookingResult bookVacation(int clientId, String city, int day) {
    try {
        // Logique métier
        if (!hasEnoughCredit(clientId, 100)) {
            throw new InsufficientCreditException(clientId, 100, 50);
        }

        return new BookingResult(1, 1, "Musée");
    } catch (VacationBookingException e) {
        ErrorHandler.handleException(e);
        return null;
    } catch (SQLException e) {
        ErrorHandler.handleException(new TechnicalException("Erreur SQL", e));
        return null;
    }
}
```

Partie 4 : Feedback utilisateur

L'utilisateur reçoit des messages adaptés en fonction du type d'erreur.

Amélioration du feedback :

On utilise une classe pour uniformiser les messages.

```
public class UserMessage {
    private final String message;
    private final MessageType type;

    public UserMessage(String message, MessageType type) {
        this.message = message;
        this.type = type;
    }

    public void display() {
        System.out.println("[" + type + "] " + message);
    }

    public enum MessageType {
        INFO, SUCCESS, WARNING, ERROR
    }
}
```

Exemple d'affichage :

```
new UserMessage("Crédit insuffisant pour effectuer cette réservation.",
                UserMessage.MessageType.ERROR).display();
```

Tests à effectuer :

1. **Logger les événements métier :**
    - Vérifier que les réservations réussies sont loggées dans `app.log`.
2. **Logger les erreurs techniques :**
    - Forcer une erreur SQL et vérifier le log de type SEVERE.
3. **Lever et gérer des erreurs métier :**
    - Tester `InsufficientCreditException` et s'assurer qu'elle est correctement affichée.
4. **Gestion centralisée des erreurs :**
    - Lever plusieurs types d'exceptions et vérifier les logs et les messages utilisateur.
5. **Validation des messages utilisateur :**
    - Vérifier que les messages affichés sont clairs et adaptés à chaque situation.