Exercice 1 - Les bases de JDBC

Objectif pédagogique

- Comprendre le fonctionnement de base de JDBC
- Maîtriser l'utilisation des PreparedStatement
- Apprendre à gérer correctement les ressources et les exceptions
- Savoir récupérer des clés générées automatiquement

Partie 1 - Implémentation de registerClient

1.1 Analyse et préparation

- Revoir la structure 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)
);
```

1.2 Implémentation

Compléter la méthode `registerClient` dans `VacationBookingApp.java` :

```
public int registerClient(String name, int age) throws SQLException {
    // La requête SQL à utiliser (à compléter)
    String sql = "INSERT INTO client(idc, nom, age) VALUES (?, ?, ?)";

    // 1. Récupérer le prochain ID disponible
    // 2. Insérer le nouveau client
    // 3. Retourner l'ID généré
}
```

Instructions :

1. Créer une requête pour obtenir le prochain ID disponible :
    - Utiliser `SELECT MAX(idc) + 1 FROM client`
    - Gérer le cas où la table est vide (retourner 1)
2. Implémenter l'insertion avec PreparedStatement :
    - Utiliser try-with-resources
    - Définir les paramètres avec les méthodes setXXX
    - Exécuter la requête
    - Gérer les potentielles erreurs SQL

3. Vérifier que l'avoir est bien initialisé par défaut à 2000

1.3 Tests

Tester la méthode avec différents cas :

- Insertion d'un client valide
- Gestion des noms trop longs (> 10 caractères)
- Gestion des âges invalides (négatifs ou ≥ 120)

Partie 2 - Amélioration de handleClientLogin

2.1 Implémentation de la vérification des identifiants

Modifier la méthode `handleClientLogin` dans `ConsoleMenu.java` :

```
private void handleClientLogin() {
    System.out.print("Entrez votre ID client : ");
    try {
        int clientId = Integer.parseInt(scanner.nextLine());
        // 1. Vérifier l'existence du client
        // 2. Récupérer et afficher ses informations
        // 3. Gérer la connexion
    } catch (NumberFormatException e) {
        System.out.println("ID client invalide");
    }
}
```

Instructions :

1. Créer une méthode `verifyClient` dans `VacationBookingApp` :

```
public record ClientInfo(String name, int age, int credit) {}

public ClientInfo verifyClient(int clientId) throws SQLException {
    String sql = "SELECT nom, age, avoir FROM client WHERE idc = ?";
    // Implémenter la vérification
}
```

2. Améliorer les messages utilisateur :
    - Afficher un message de bienvenue avec le nom
    - Afficher l'avoir disponible
    - Gérer les cas d'erreur de manière informative

2.2 Gestion des erreurs

Implémenter une gestion d'erreurs professionnelle :

1. Créer une exception métier :

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

2. Utiliser cette exception dans le code :

```java
public ClientInfo verifyClient(int clientId) throws SQLException, ClientNotFoundExcept
    // Implémenter la logique avec la nouvelle exception
}
```

Exercice 2 - Requêtes et transactions

Objectif pédagogique

- Maîtriser les requêtes SQL complexes en JDBC
- Comprendre et implémenter les transactions
- Apprendre à gérer les contraintes métier
- Pratiquer la gestion des erreurs avancée

Partie 1 - Implémentation de bookVacation

1.1 Analyse du problème

Rappel du cahier des charges :

- Trouver le village le plus cher dans la ville donnée
- Créer le séjour si possible
- Décrémenter l'avoir du client du prix du séjour
- Gérer le cas où aucun village n'est disponible

1.2 Implémentation de base

```
public BookingResult bookVacation(int clientId, String city, int day) throws SQLExcept
    // Désactiver l'autocommit pour gérer la transaction
    boolean originalAutoCommit = connection.getAutoCommit();
    connection.setAutoCommit(false);

    try {
        // 1. Trouver le village le plus cher dans la ville
        String findVillageSql = """
            SELECT idv, prix, activite
            FROM village
            WHERE ville = ?
            ORDER BY prix DESC
            FETCH FIRST 1 ROW ONLY""";

        // 2. Vérifier l'avoir du client
        // 3. Créer le séjour
        // 4. Mettre à jour l'avoir
        // 5. Commit et retourner le résultat

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

    } catch (SQLException e) {
        connection.rollback();
        throw e;
    } finally {
        connection.setAutoCommit(originalAutoCommit);
    }
}
```

Partie 2 - Ajout des vérifications

2.1 Vérification de la disponibilité

```
private boolean isVillageAvailable(int villageId, int day) throws SQLException {
    // Vérifier la capacité du village
    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)) {
        // À compléter
    }
}
```

2.2 Vérification de l'avoir

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

2.3 Génération d'un nouvel ID de séjour

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

Exercice 3 - Gestion des résultats

Objectif pédagogique

- Maîtriser l'utilisation des ResultSet
- Apprendre à formater les données pour l'affichage
- Effectuer des jointures en JDBC
- Pratiquer les calculs et agrégations sur les résultats

Partie 1 - Affichage des séjours

1.1 Implémentation de handleViewBookings

Créer une méthode pour récupérer les séjours avec leurs détails :

```
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<>();

        // TODO : Implémenter la récupération des résultats

        return bookings;
    }
}
```

Dans ConsoleMenu, améliorer l'affichage :

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

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

        // TODO : Implémenter un affichage formaté des réservations
        // Exemple de format attendu :
        // ======================================
        // ID    VILLE       ACTIVITÉ    JOUR   PRIX
        // ---   -----       ---------   ----   -----
        // 1     Paris       Musée       120    50€
        // 2     Nice        Plage       200    75€
        // ======================================
        // Total des dépenses : 125€
        // Nombre de séjours : 2
    } catch (SQLException e) {
        System.out.println("Erreur lors de la récupération de vos réservations : " + e
    }
}
```

Partie 2 - Affichage des villages disponibles

2.1 Implémentation de handleViewVillages

Créer une méthode pour récupérer les villages sans séjour :

```
public class VacationBookingApp {
    public record VillageInfo(
        int villageId,
        String city,
        String activity,
        int price,
        Integer capacity  // null si client
    ) { }

    public List<VillageInfo> getAvailableVillages(boolean isEmployee) throws SQLExcept
        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" : ""
            );

        // TODO : Implémenter la récupération des résultats
    }
}
```

Améliorer l'affichage dans ConsoleMenu :

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

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

        // TODO : Implémenter un affichage formaté
        // Exemple pour un client :
        // ========================================
        // VILLE       ACTIVITÉ          PRIX
        // -----       ---------        ------
        // Nice        Plage            75€
        // Paris       Musée            50€
        // ========================================

        // Exemple pour un employé (avec capacité) :
        // ================================================
        // VILLE       ACTIVITÉ          PRIX     CAPACITÉ
        // -----       ---------        ------    ---------
        // Nice        Plage            75€         250
        // Paris       Musée            50€         100
        // ================================================

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

2.2 Ajouter des statistiques utiles

```
public record VillageStats(
    double averagePrice,
    String mostCommonActivity,
    int totalCapacity
) { }

public VillageStats getVillageStatistics() throws SQLException {
    String sql = """
        SELECT
            AVG(prix) as avg_price,
            MAX(activite) KEEP (DENSE_RANK FIRST ORDER BY cnt DESC) as top_activity,
            SUM(capacite) as total_capacity
        FROM (
            SELECT activite, prix, capacite, COUNT(*) OVER (PARTITION BY activite) as
            FROM village
        )""";

    // TODO : Implémenter la récupération des statistiques
}
```

Exercice 4 - Droits d'accès et sécurité

Objectif pédagogique

- Comprendre la gestion des droits JDBC
- Implémenter un système de contrôle d'accès
- Sécuriser les accès à la base de données
- Protéger l'application contre les attaques courantes

Partie 1 - Gestion des droits d'accès

1.1 Création d'un système de connexion sécurisé

1. Créer une classe pour gérer les utilisateurs :

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

    // Constructeur et getters à implémenter
}

public enum UserRole {
    CLIENT,
    EMPLOYEE
}

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

2. Implémenter un gestionnaire de session :

```
public class SessionManager {
    private User currentUser;

    public void login(String username, String password) throws AuthenticationExceptior
        // TODO: Implémenter la logique de connexion
        // 1. Vérifier les credentials
        // 2. Charger les permissions
        // 3. Créer la session utilisateur
    }

    public boolean hasPermission(Permission permission) {
        // TODO: Vérifier si l'utilisateur courant a la permission
    }

    public void validateAccess(Permission permission) throws AccessDeniedException {
        // TODO: Lever une exception si l'utilisateur n'a pas la permission
    }
}
```

1.2 Implémentation des vues SQL

Créer des vues SQL pour simplifier et sécuriser les accès :

```
-- À implémenter :
-- 1. Vue pour les villages accessibles aux clients
-- 2. Vue pour les séjours d'un client
-- 3. Vue pour les statistiques accessibles aux employés
```

Partie 2 - Sécurisation de l'application

2.1 Protection contre les injections SQL

1. Créer un validateur d'entrées :

```
public class InputValidator {
    public static void validateName(String name) throws ValidationException {
        // TODO: Implémenter la validation
    }

    public static void validateDay(int day) throws ValidationException {
        // TODO: Implémenter la validation
    }

    public static void validateCity(String city) throws ValidationException {
        // TODO: Implémenter la validation
    }
}
```

2. Créer un constructeur de requêtes sécurisé :

```
public class SecureQueryBuilder {
    public static PreparedStatement buildVillageQuery(
        Connection conn,
        Map<String, Object> filters
    ) throws SQLException {
        // TODO: Construire dynamiquement une requête sécurisée
    }
}
```

2.2 Implémentation de la sécurité applicative

1. Ajouter une couche de validation dans les méthodes existantes :

```
public BookingResult bookVacation(int clientId, String city, int day)
    throws SQLException, ValidationException, AccessDeniedException {

    // TODO:
    // 1. Valider les entrées
    // 2. Vérifier les permissions
    // 3. Exécuter la réservation
}
```

2. Sécuriser la connexion employé :

```
private void handleEmployeeLogin() {
    // TODO:
    // 1. Demander le nom d'utilisateur
    // 2. Gérer le mot de passe de manière sécurisée
    // 3. Vérifier les credentials dans la base
    // 4. Créer la session avec les bonnes permissions
}
```

Exercice 5 - Logging et gestion des erreurs

Objectif pédagogique

- Mettre en place un système de logging professionnel
- Créer une hiérarchie d'exceptions métier
- Améliorer la gestion des erreurs
- Rendre l'application plus maintenable

Partie 1 - Système de logging

1.1 Configuration du logging

1. Créer un fichier de configuration pour le logging :

```
public class LogConfig {
    public static void initializeLogging() {
        // TODO:
        // 1. Configurer le logger pour la console
        // 2. Configurer le logger pour les fichiers
        // 3. Définir les niveaux de log appropriés
        // 4. Définir le format des messages de log
    }
}
```

2. Définir des catégories de log :

```
public enum LogCategory {
    SECURITY,
    BUSINESS,
    TECHNICAL,
    AUDIT;

    // TODO: Ajouter les méthodes utilitaires pour le logging
}
```

1.2 Implémentation du logging

1. Créer une classe utilitaire pour le logging :

```
public class AppLogger {
    public static void logBusinessEvent(String message, Object... params) {
        // TODO: Implémenter
    }

    public static void logSecurityEvent(String message, Object... params) {
        // TODO: Implémenter
    }

    public static void logTechnicalError(String message, Throwable error, Object... pa
        // TODO: Implémenter
    }

    public static void logAudit(String action, String user, String details) {
        // TODO: Implémenter
    }
}
```

Partie 2 - Gestion des erreurs

2.1 Création d'une hiérarchie d'exceptions

1. Définir les exceptions de base :

```
public class VacationBookingException extends Exception {
    // TODO: Implémenter
}

public class BusinessException extends VacationBookingException {
    // TODO: Implémenter
}

public class TechnicalException extends VacationBookingException {
    // TODO: Implémenter
}
```

2. Créer les exceptions métier spécifiques :

```
public class InsufficientCreditException extends BusinessException {
    private final int clientId;
    private final int requiredAmount;
    private final int availableCredit;

    // TODO: Implémenter constructeur et méthodes
}

public class VillageFullException extends BusinessException {
    // TODO: Implémenter
}

public class BookingDayException extends BusinessException {
    // TODO: Implémenter
}
```

2.2 Implémentation de la gestion des erreurs

1. Créer un gestionnaire d'erreurs centralisé :

```
public class ErrorHandler {
    public static void handleException(
        VacationBookingException e,
        LogCategory category
    ) {
        // TODO:
        // 1. Logger l'erreur avec le niveau approprié
        // 2. Préparer un message utilisateur adapté
        // 3. Gérer les cas spécifiques
    }

    public static String getUserFriendlyMessage(VacationBookingException e) {
        // TODO: Retourner un message compréhensible pour l'utilisateur
    }
}
```

2. Modifier les méthodes existantes pour utiliser le nouveau système :

```
public BookingResult bookVacation(int clientId, String city, int day)
    throws VacationBookingException {

    try {
        // TODO:
        // 1. Ajouter les logs appropriés
        // 2. Utiliser les nouvelles exceptions
        // 3. Gérer les erreurs de manière cohérente
    } catch (SQLException e) {
        // TODO: Convertir en TechnicalException
    }
}
```

2.3 Amélioration du feedback utilisateur

1. Créer une classe pour les messages utilisateur :

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

    // TODO: Implémenter
}

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