Héritage avec Spring Data JPA

Introduction

L'héritage est un concept fondamental de la POO qui peut être mappé en base de données de différentes manières avec JPA. JPA propose **3 stratégies principales** définies par l'annotation `@Inheritance`.

Les Stratégies d'Héritage JPA

1. SINGLE_TABLE (Table unique)

Principe

Toutes les classes de la hiérarchie sont stockées dans **une seule table** avec une colonne discriminante.

Code

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "type_personne", discriminatorType =
DiscriminatorType.STRING)
public abstract class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    private String prenom;
    // getters/setters
}

@Entity
@DiscriminatorValue("ETUDIANT")
public class Etudiant extends Personne {
    private String numeroEtudiant;
    private Integer promotion;
    // getters/setters
}

@Entity
@DiscriminatorValue("PROFESSEUR")
public class Professeur extends Personne {
    private String specialite;
    private Double salaire;
    // getters/setters
}
```

Table résultante

```
CREATE TABLE personne (
    id BIGINT PRIMARY KEY,
    type_personne VARCHAR(31),      -- Colonne discriminante
    nom VARCHAR(255),
    prenom VARCHAR(255),
    numero_etudiant VARCHAR(255),   -- NULL pour professeurs
    promotion INT,                  -- NULL pour professeurs
    specialite VARCHAR(255),        -- NULL pour étudiants
    salaire DOUBLE                  -- NULL pour étudiants
);
```

Avantages :

- Performance excellente (pas de JOIN)
- Requêtes polymorphiques simples
- Une seule table à gérer

Inconvénients :

- Beaucoup de colonnes NULL
- Violation de la normalisation
- Contraintes d'intégrité difficiles

Cas d'usage : Hiérarchies simples, peu de champs spécifiques par sous-classe.

2. JOINED (Table par classe)

Principe

Une table pour chaque classe de la hiérarchie, liées par **clé étrangère** vers la table parent.

Code

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
    private String nom;
    private String prenom;
    // getters/setters
}

@Entity
public class Etudiant extends Personne {
    private String numeroEtudiant;
    private Integer promotion;
    // getters/setters
}

@Entity
public class Professeur extends Personne {
    private String specialite;
    private Double salaire;
    // getters/setters
}
```

Tables résultantes

```
CREATE TABLE personne (
    id BIGINT PRIMARY KEY,
    nom VARCHAR(255),
    prenom VARCHAR(255)
);

CREATE TABLE etudiant (
    id BIGINT PRIMARY KEY,
    numero_etudiant VARCHAR(255),
    promotion INT,
    FOREIGN KEY (id) REFERENCES personne(id)
);

CREATE TABLE professeur (
    id BIGINT PRIMARY KEY,
    specialite VARCHAR(255),
    salaire DOUBLE,
    FOREIGN KEY (id) REFERENCES personne(id)
);
```

Avantages :

- Normalisation respectée
- Pas de colonnes NULL
- Structure claire et maintenable
- Contraintes d'intégrité faciles

❌ Inconvénients :

- Performances (JOINs nécessaires pour chaque requête)
- Requêtes plus complexes

✅ Cas d'usage : Hiérarchies complexes, intégrité des données importante, nombreux champs spécifiques.

3. TABLE_PER_CLASS (Table par classe concrète)

Principe

Une table complète pour **chaque classe concrète** (pas de table pour la classe abstraite). Les champs du parent sont **dupliqués** dans chaque table enfant.

Code

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)  // Attention au type de
génération
    private Long id;
    private String nom;
    private String prenom;
    // getters/setters
}

@Entity
public class Etudiant extends Personne {
    private String numeroEtudiant;
    private Integer promotion;
    // getters/setters
}

@Entity
public class Professeur extends Personne {
    private String specialite;
    private Double salaire;
    // getters/setters
}
```

Tables résultantes

```
-- Pas de table personne !

CREATE TABLE etudiant (
    id BIGINT PRIMARY KEY,
    nom VARCHAR(255),                -- Dupliqué
    prenom VARCHAR(255),             -- Dupliqué
    numero_etudiant VARCHAR(255),
    promotion INT
);

CREATE TABLE professeur (
    id BIGINT PRIMARY KEY,
    nom VARCHAR(255),                -- Dupliqué
    prenom VARCHAR(255),             -- Dupliqué
    specialite VARCHAR(255),
    salaire DOUBLE
);
```

Avantages :

- Pas de colonnes NULL
- Bonnes performances pour requêtes sur une classe spécifique
- Isolation complète des données

Inconvénients :

- Duplication des colonnes communes
- Requêtes polymorphiques très lentes (nécessite des UNION)
- Gestion des IDs complexe
- Difficile à maintenir

Cas d'usage : Classes très différentes, peu ou pas de requêtes polymorphiques.

@MappedSuperclass (Alternative à l'héritage)

Principe

La classe parente **n'est PAS une entité**, elle sert uniquement de modèle pour partager des champs communs. Aucune table n'est créée pour elle, et **aucune requête polymorphique** n'est possible.

Code

```
@MappedSuperclass
public abstract class EntiteAuditee {
    @CreatedDate
    private LocalDateTime dateCreation;
    @LastModifiedDate
    private LocalDateTime dateModification;
    @CreatedBy
    private String creerPar;
    @LastModifiedBy
    private String modifierPar;
    // getters/setters
}

@Entity
public class Produit extends EntiteAuditee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    private Double prix;
    // getters/setters
}

@Entity
public class Commande extends EntiteAuditee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String numero;
    private Double total;
    // getters/setters
}
```

Tables résultantes

```
-- Pas de table entite_auditee !

CREATE TABLE produit (
    id BIGINT PRIMARY KEY,
    nom VARCHAR(255),
    prix DOUBLE,
    date_creation TIMESTAMP,
    date_modification TIMESTAMP,
    creer_par VARCHAR(255),
    modifier_par VARCHAR(255)
);

CREATE TABLE commande (
    id BIGINT PRIMARY KEY,
    numero VARCHAR(255),
    total DOUBLE,
    date_creation TIMESTAMP,
    date_modification TIMESTAMP,
```

```
    creer_par VARCHAR(255),
    modifier_par VARCHAR(255)
);
```

Avantages :

- Réutilisation du code (DRY)
- Pas de complexité d'héritage en base
- Chaque table est indépendante
- Performances optimales

Inconvénients :

- Pas de requêtes polymorphiques possibles
- Pas de relations vers la classe parente
- Pas d'identité commune en base

Cas d'usage : Partage de champs communs (audit, timestamps, champs techniques), sans relation d'héritage métier.

Repositories Spring Data JPA

Repositories de base

```
// Repository pour le parent (requêtes polymorphiques)
public interface PersonneRepository extends JpaRepository<Personne, Long> {
    List<Personne> findByNom(String nom);
    // Retourne TOUS les types (Etudiant + Professeur)
    List<Personne> findByPrenomContaining(String prenom);
}

// Repository spécifique pour Etudiant
public interface EtudiantRepository extends JpaRepository<Etudiant, Long> {
    List<Etudiant> findByPromotion(Integer promotion);
    @Query("SELECT e FROM Etudiant e WHERE e.numeroEtudiant = :numero")
    Optional<Etudiant> findByNumero(@Param("numero") String numero);
    // Requête spécifique aux étudiants
    List<Etudiant> findByPromotionGreaterThan(Integer promotion);
}

// Repository spécifique pour Professeur
public interface ProfesseurRepository extends JpaRepository<Professeur, Long> {
    List<Professeur> findBySpecialite(String specialite);
    List<Professeur> findBySalaireGreaterThan(Double salaire);
```

```
    @Query("SELECT p FROM Professeur p WHERE p.specialite LIKE %:mot%")
    List<Professeur> rechercherParSpecialite(@Param("mot") String mot);
}
```

Bonne pratique : Créer un repository pour chaque type (parent et enfants) permet d'avoir des méthodes spécifiques tout en conservant les requêtes polymorphiques.

Requêtes polymorphiques

```
@Service
public class PersonneService {
    @Autowired
    private PersonneRepository personneRepository;
    @Autowired
    private EtudiantRepository etudiantRepository;
    @Autowired
    private ProfesseurRepository professeurRepository;
    // Récupère TOUS les types (Etudiant + Professeur)
    public List<Personne> toutesLesPersonnes() {
        return personneRepository.findAll();
    }
    // Filtrage par type avec instanceOf
    public List<Etudiant> seulementLesEtudiants() {
        return personneRepository.findAll().stream()
            .filter(p -> p instanceof Etudiant)
            .map(p -> (Etudiant) p)
            .collect(Collectors.toList());
    }
    // Meilleure approche : utiliser le repository spécifique
    public List<Etudiant> tousLesEtudiants() {
        return etudiantRepository.findAll();
    }
    // Utilisation de TYPE() dans JPQL
    @Query("SELECT p FROM Personne p WHERE TYPE(p) = Etudiant")
    List<Etudiant> findAllEtudiants();
    @Query("SELECT p FROM Personne p WHERE TYPE(p) = Professeur")
    List<Professeur> findAllProfesseurs();
    // Filtrage multiple
    @Query("SELECT p FROM Personne p WHERE TYPE(p) IN (Etudiant, Professeur)")
    List<Personne> findAllPersonnesSpecifiques();
    // Requête polymorphique avec condition
    @Query("SELECT p FROM Personne p WHERE p.nom = :nom AND TYPE(p) = Etudiant")
    List<Etudiant> findEtudiantsByNom(@Param("nom") String nom);
}
```

Attention : Les requêtes polymorphiques sur `PersonneRepository` retournent **tous** les sous-types. Utilisez les repositories spécifiques pour des requêtes ciblées.

Annotations importantes

@Inheritance

```
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)  // ou JOINED,
TABLE_PER_CLASS

// Paramètres possibles :
// - InheritanceType.SINGLE_TABLE : table unique (défaut)
// - InheritanceType.JOINED : table par classe
// - InheritanceType.TABLE_PER_CLASS : table par classe concrète
```

@DiscriminatorColumn (pour SINGLE_TABLE)

```
@DiscriminatorColumn(
    name = "type_entite",                          // Nom de la colonne (défaut :
DTYPE)
    discriminatorType = DiscriminatorType.STRING,   // STRING, INTEGER, CHAR
    length = 50                                    // Taille (pour STRING)
)

// Exemples :
@DiscriminatorColumn(name = "type")                // VARCHAR(31) par défaut
@DiscriminatorColumn(name = "categorie", length = 100)
@DiscriminatorColumn(name = "type_id", discriminatorType =
DiscriminatorType.INTEGER)
```

Obligatoire avec SINGLE_TABLE (créée automatiquement avec le nom DTYPE si non spécifiée).

Non utilisée avec JOINED et TABLE_PER_CLASS.

@DiscriminatorValue

```
@DiscriminatorValue("ETUDIANT")  // Valeur dans la colonne discriminante

// Si non spécifié, utilise le nom de la classe par défaut
@Entity
@DiscriminatorValue("PROF")      // Personnalisé
public class Professeur extends Personne { }

@Entity                          // Utilisera "Etudiant" par défaut
public class Etudiant extends Personne { }
```

@PrimaryKeyJoinColumn (pour JOINED)

```
@Entity
@PrimaryKeyJoinColumn(name = "etudiant_id")  // Personnalise le nom de la FK
public class Etudiant extends Personne {
    // Par défaut, la FK aurait été nommée "id"
}

// Avec plusieurs colonnes (cas rare)
@Entity
@PrimaryKeyJoinColumn(name = "id_etudiant", referencedColumnName = "id")
public class Etudiant extends Personne { }
```

Exemples Pratiques Complets

Exemple 1 : Système de paiement (SINGLE_TABLE)

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "payment_type")
public abstract class Paiement {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Double montant;
    private LocalDateTime date;
    @ManyToOne
    @JoinColumn(name = "commande_id")
    private Commande commande;
    // Méthode abstraite pour le polymorphisme
    public abstract String getMethodeDescription();
}

@Entity
@DiscriminatorValue("CARTE")
public class PaiementCarte extends Paiement {
    private String numeroCarte;
    private String cvv;
    private String nomTitulaire;
    @Override
    public String getMethodeDescription() {
        return "Carte bancaire";
    }
}

@Entity
@DiscriminatorValue("PAYPAL")
public class PaiementPaypal extends Paiement {
    private String email;
```

```
    private String transactionId;
    @Override
    public String getMethodeDescription() {
        return "PayPal";
    }
}

@Entity
@DiscriminatorValue("VIREMENT")
public class PaiementVirement extends Paiement {
    private String iban;
    private String bic;
    private String reference;
    @Override
    public String getMethodeDescription() {
        return "Virement bancaire";
    }
}

// Repositories
public interface PaiementRepository extends JpaRepository<Paiement, Long> {
    List<Paiement> findByMontantGreaterThan(Double montant);
    @Query("SELECT p FROM Paiement p WHERE TYPE(p) = PaiementCarte")
    List<PaiementCarte> findAllPaiementsCarte();
    @Query("SELECT p FROM Paiement p WHERE p.date BETWEEN :debut AND :fin")
    List<Paiement> findByDateBetween(@Param("debut") LocalDateTime debut,
                                     @Param("fin") LocalDateTime fin);
}

public interface PaiementCarteRepository extends JpaRepository<PaiementCarte, Long>
{
    Optional<PaiementCarte> findByNumeroCarte(String numeroCarte);
}

// Service
@Service
public class PaiementService {
    @Autowired
    private PaiementRepository paiementRepository;
    public void traiterPaiement(Paiement paiement) {
        paiementRepository.save(paiement);
        // Traitement spécifique selon le type
        if (paiement instanceof PaiementCarte carte) {
            // Logique spécifique carte
            validerCarte(carte);
        } else if (paiement instanceof PaiementPaypal paypal) {
            // Logique spécifique PayPal
            verifierTransactionPaypal(paypal);
        } else if (paiement instanceof PaiementVirement virement) {
            // Logique spécifique virement
            validerIban(virement);
        }
    }
    public Map<String, Long> statistiquesParType() {
        return paiementRepository.findAll().stream()
            .collect(Collectors.groupingBy(
```

```
                p -> p.getClass().getSimpleName(),
                Collectors.counting()
            ));
    }
    public Double montantTotalParType(String type) {
        return paiementRepository.findAll().stream()
            .filter(p -> p.getClass().getSimpleName().equals(type))
            .mapToDouble(Paiement::getMontant)
            .sum();
    }
    private void validerCarte(PaiementCarte carte) {
        // Logique de validation
    }
    private void verifierTransactionPaypal(PaiementPaypal paypal) {
        // Logique de vérification
    }
    private void validerIban(PaiementVirement virement) {
        // Logique de validation
    }
}
```

Pourquoi SINGLE_TABLE ici ?

- Peu de différences entre les types de paiement
- Requêtes fréquentes sur tous les paiements
- Performance importante pour le traitement

Exemple 2 : Système de documents (JOINED)

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Document {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String titre;
    private LocalDateTime dateCreation;
    private String description;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "utilisateur_id")
    private Utilisateur auteur;
    @Enumerated(EnumType.STRING)
    private StatutDocument statut;
    // getters/setters
}

@Entity
public class DocumentTexte extends Document {
    @Lob
    @Column(columnDefinition = "TEXT")
```

```
    private String contenu;
    private Integer nombreMots;
    private Integer nombreParagraphes;
    @Enumerated(EnumType.STRING)
    private FormatTexte format; // MARKDOWN, HTML, PLAIN
    // getters/setters
}

@Entity
public class DocumentImage extends Document {
    private String url;
    private Integer largeur;
    private Integer hauteur;
    private String format; // PNG, JPG, GIF, etc.
    private Long tailleFichier;
    @Column(columnDefinition = "TEXT")
    private String altText;
    // getters/setters
}

@Entity
public class DocumentPdf extends Document {
    private String cheminFichier;
    private Integer nombrePages;
    private Long tailleFichier;
    private Boolean estProtege;
    @Column(columnDefinition = "TEXT")
    private String metadonnees;
    // getters/setters
}

@Entity
public class DocumentVideo extends Document {
    private String url;
    private Integer dureeSecondes;
    private String codec;
    private String resolution;
    private Long tailleFichier;
    // getters/setters
}

// Repositories
public interface DocumentRepository extends JpaRepository<Document, Long> {
    List<Document> findByAuteur(Utilisateur auteur);
    @Query("SELECT d FROM Document d WHERE d.dateCreation BETWEEN :debut AND :fin")
    List<Document> findByPeriode(@Param("debut") LocalDateTime debut,
                                 @Param("fin") LocalDateTime fin);
    @Query("SELECT TYPE(d), COUNT(d) FROM Document d GROUP BY TYPE(d)")
    List<Object[]> countByType();
}

public interface DocumentTexteRepository extends JpaRepository<DocumentTexte, Long>
{
    List<DocumentTexte> findByNombreMotsGreaterThan(Integer nombreMots);
    @Query("SELECT d FROM DocumentTexte d WHERE d.contenu LIKE %:mot%")
    List<DocumentTexte> rechercherParContenu(@Param("mot") String mot);
```

```
    List<DocumentTexte> findByFormat(FormatTexte format);
}

public interface DocumentImageRepository extends JpaRepository<DocumentImage, Long>
{
    List<DocumentImage> findByFormat(String format);
    List<DocumentImage> findByLargeurGreaterThanAndHauteurGreaterThan(
        Integer largeurMin, Integer hauteurMin);
}

public interface DocumentPdfRepository extends JpaRepository<DocumentPdf, Long> {
    List<DocumentPdf> findByNombrePagesGreaterThan(Integer nombrePages);
    List<DocumentPdf> findByEstProtege(Boolean protege);
}

// Service
@Service
public class DocumentService {
    @Autowired
    private DocumentRepository documentRepository;
    @Autowired
    private DocumentTexteRepository documentTexteRepository;
    @Autowired
    private DocumentImageRepository documentImageRepository;
    public List<Document> documentsParUtilisateur(Utilisateur utilisateur) {
        return documentRepository.findByAuteur(utilisateur);
    }
    public Map<String, Long> statistiquesParType() {
        List<Object[]> results = documentRepository.countByType();
        return results.stream()
            .collect(Collectors.toMap(
                arr -> ((Class<?>) arr[0]).getSimpleName(),
                arr -> (Long) arr[1]
            ));
    }
    public List<DocumentTexte> rechercherTexte(String recherche) {
        return documentTexteRepository.rechercherParContenu(recherche);
    }
    public Long calculerEspaceUtilise(Utilisateur utilisateur) {
        return documentRepository.findByAuteur(utilisateur).stream()
            .filter(d -> d instanceof DocumentImage || d instanceof DocumentPdf ||
d instanceof DocumentVideo)
            .mapToLong(d -> {
                if (d instanceof DocumentImage) {
                    return ((DocumentImage) d).getTailleFichier();
                } else if (d instanceof DocumentPdf) {
                    return ((DocumentPdf) d).getTailleFichier();
                } else if (d instanceof DocumentVideo) {
                    return ((DocumentVideo) d).getTailleFichier();
                }
                return 0L;
            })
            .sum();
    }
}
```

Pourquoi JOINED ici ?

- Beaucoup de champs spécifiques à chaque type
- Intégrité des données importante
- Différences significatives entre les types
- Structure évolutive (facile d'ajouter de nouveaux types)

Exemple 3 : Audit avec @MappedSuperclass

```
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public abstract class EntiteAuditee {
    @CreatedDate
    @Column(nullable = false, updatable = false)
    private LocalDateTime dateCreation;
    @LastModifiedDate
    @Column(nullable = false)
    private LocalDateTime dateModification;
    @CreatedBy
    @Column(nullable = false, updatable = false)
    private String creerPar;
    @LastModifiedBy
    @Column(nullable = false)
    private String modifierPar;
    @Version
    private Integer version; // Pour l'optimistic locking
    // getters/setters
}

// Configuration Spring pour l'audit
@Configuration
@EnableJpaAuditing
public class JpaAuditConfig {
    @Bean
    public AuditorAware<String> auditorProvider() {
        return () -> {
            // Récupère l'utilisateur connecté depuis Spring Security
            Authentication authentication = SecurityContextHolder
                .getContext()
                .getAuthentication();
            if (authentication == null || !authentication.isAuthenticated()) {
                return Optional.of("system");
            }
            return Optional.of(authentication.getName());
        };
    }
}

// Utilisation dans les entités
@Entity
```

```
public class Produit extends EntiteAuditee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    private String reference;
    private Double prix;
    @Enumerated(EnumType.STRING)
    private CategorieProduit categorie;
    // getters/setters
}

@Entity
public class Commande extends EntiteAuditee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String numero;
    private Double total;
    @Enumerated(EnumType.STRING)
    private StatutCommande statut;
    @OneToMany(mappedBy = "commande", cascade = CascadeType.ALL)
    private List<LigneCommande> lignes = new ArrayList<>();
    // getters/setters
}

@Entity
public class Client extends EntiteAuditee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    private String prenom;
    private String email;
    @OneToMany(mappedBy = "client")
    private List<Commande> commandes = new ArrayList<>();
    // getters/setters
}

// Service utilisant l'audit
@Service
public class AuditService {
    @Autowired
    private ProduitRepository produitRepository;
    public List<Produit> produitsModifiesRecemment(int heures) {
        LocalDateTime limite = LocalDateTime.now().minusHours(heures);
        return produitRepository.findAll().stream()
            .filter(p -> p.getDateModification().isAfter(limite))
            .collect(Collectors.toList());
    }
    public Map<String, Long> statistiquesModificationsParUtilisateur() {
        return produitRepository.findAll().stream()
            .collect(Collectors.groupingBy(
                EntiteAuditee::getModifierPar,
                Collectors.counting()
            ));
```

```
    }
}
```

Avantages de @MappedSuperclass pour l'audit :

- Code DRY : champs d'audit définis une seule fois
- Appliqué automatiquement à toutes les entités qui héritent
- Pas de complexité en base de données
- Fonctionne avec toutes les stratégies d'héritage

Comparaison des Stratégies

Critère	SINGLE_TABLE	JOINED	TABLE_PER_CLASS
Performance lecture	□□□	□□	□□
Performance écriture	□□□	□□	□□□
Normalisation	□	□	⚠
Colonnes NULL	Beaucoup	Aucune	Aucune
Requêtes polymorphiques	□□□	□□	□ (UNION)
Complexité	Simple	Moyenne	Complexe
Évolutivité	□□	□□□	□
Intégrité référentielle	⚠	□	⚠
Taille base de données	Moyenne	Grande	Grande

Bonnes Pratiques

□ À FAIRE

1. Choisir la bonne stratégie selon le contexte

```
// Peu de différences entre sous-classes → SINGLE_TABLE
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Notification { }

// Beaucoup de champs spécifiques → JOINED
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Document { }

// Classes très distinctes → TABLE_PER_CLASS (rare)
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Support { }
```

2. Utiliser des classes abstraites pour les parents

```
// ✅ BON
public abstract class Personne { }

// ❌ MAUVAIS (sauf cas spécifique)
public class Personne { }
```

3. Nommer explicitement les colonnes discriminantes

```
// ✅ BON
@DiscriminatorColumn(name = "type_entite")
@DiscriminatorValue("ETUDIANT")

// ⚠️ Moins clair
@DiscriminatorValue("E")  // Trop court, peu lisible
```

4. Créer des repositories pour chaque type

```
// ✅ BON
public interface PersonneRepository extends JpaRepository<Personne, Long>
{ }
public interface EtudiantRepository extends JpaRepository<Etudiant, Long>
{ }
public interface ProfesseurRepository extends JpaRepository<Professeur,
Long> { }

// Permet des requêtes spécifiques ET polymorphiques
```

5. Utiliser @MappedSuperclass pour les champs techniques

```
// ✅ BON pour l'audit, timestamps, champs communs non métier
@MappedSuperclass
public abstract class BaseEntity {
```

```
    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;
}
```

6. Implémenter des méthodes helper pour le polymorphisme

```
@Entity
public abstract class Paiement {
    // Méthode abstraite pour forcer l'implémentation
    public abstract String getMethodeDescription();
    // Méthode commune
    public boolean estValide() {
        return montant != null && montant > 0;
    }
}
```

□ À ÉVITER

1. Mélanger les stratégies dans une même hiérarchie

```
// □ IMPOSSIBLE
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Personne { }

@Entity
@Inheritance(strategy = InheritanceType.JOINED)  // □ Conflit !
public class Etudiant extends Personne { }
```

2. Oublier @DiscriminatorValue avec SINGLE_TABLE

```
// ⚠□ Fonctionne mais utilise le nom de la classe
@Entity
public class Etudiant extends Personne { }  // Discriminateur =
"Etudiant"

// □ Meilleur
```

```
@Entity
@DiscriminatorValue("ETUDIANT")
public class Etudiant extends Personne { }
```

3. Utiliser TABLE_PER_CLASS avec requêtes polymorphiques fréquentes

```
// ❌ MAUVAIS : génère des UNION très lentes
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Document { }

// Service avec requêtes polymorphiques fréquentes
documentRepository.findAll();  // ❌ UNION sur toutes les tables !
```

4. Créer des hiérarchies trop profondes

```
// ❌ MAUVAIS : trop de niveaux
Personne
  → Employe
    → Cadre
      → Directeur
        → DirecteurGeneral

// ✅ BON : maximum 2-3 niveaux
Personne
  → Etudiant
  → Professeur
  → Administrateur
```

5. Utiliser @Inheritance pour du code partagé non métier

```
// ❌ MAUVAIS : utiliser @Inheritance
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class EntiteAvecTimestamps { }

// ✅ BON : utiliser @MappedSuperclass
@MappedSuperclass
```

```
public abstract class EntiteAvecTimestamps { }
```

Cas d'Usage Réels

E-commerce

```
// SINGLE_TABLE pour Produit (peu de différences)
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "type_produit")
public abstract class Produit {
    @Id
    private Long id;
    private String nom;
    private Double prix;
}

@Entity
@DiscriminatorValue("PHYSIQUE")
public class ProduitPhysique extends Produit {
    private Double poids;
    private String dimensionsEmballage;
}

@Entity
@DiscriminatorValue("NUMERIQUE")
public class ProduitNumerique extends Produit {
    private String urlTelechargement;
    private Integer nombreTelechargements;
}

// JOINED pour Utilisateur (différences importantes)
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Utilisateur {
    @Id
    private Long id;
    private String email;
    private String motDePasse;
}

@Entity
public class Client extends Utilisateur {
    private String adresseLivraison;
    @OneToMany(mappedBy = "client")
    private List<Commande> commandes;
}

@Entity
```

```
public class Vendeur extends Utilisateur {
    private String nomBoutique;
    @OneToMany(mappedBy = "vendeur")
    private List<Produit> produits;
}

@Entity
public class Administrateur extends Utilisateur {
    private Set<String> permissions;
}

// @MappedSuperclass pour l'audit
@MappedSuperclass
public abstract class EntiteAuditee {
    private LocalDateTime createdAt;
    private LocalDateTime updatedAt;
}
```

Système bancaire

```
// JOINED pour Compte (intégrité critique)
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Compte {
    @Id
    private Long id;
    private String numero;
    private Double solde;
    @ManyToOne
    private Client proprietaire;
}

@Entity
public class CompteCourant extends Compte {
    private Double decouvertAutorise;
    private Double fraisTenus;
    private Boolean carteAssociee;
}

@Entity
public class CompteEpargne extends Compte {
    private Double tauxInteret;
    private LocalDate dateOuverture;
    private Integer nombreRetraitsAnnuels;
}

@Entity
public class CompteJeune extends Compte {
    private LocalDate dateNaissance;
    private String nomResponsable;
    private Double plafondOperations;
}
```

```
// SINGLE_TABLE pour Transaction (performances)
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "type_transaction")
public abstract class Transaction {
    @Id
    private Long id;
    private Double montant;
    private LocalDateTime date;
    @ManyToOne
    private Compte compte;
}

@Entity
@DiscriminatorValue("DEPOT")
public class Depot extends Transaction {
    private String origine;
}

@Entity
@DiscriminatorValue("RETRAIT")
public class Retrait extends Transaction {
    private String destination;
    private String autorisationCode;
}

@Entity
@DiscriminatorValue("VIREMENT")
public class Virement extends Transaction {
    @ManyToOne
    private Compte compteDestination;
    private String reference;
}
```

Plateforme éducative

```
// JOINED pour Personne (structure complexe)
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Personne {
    @Id
    private Long id;
    private String nom;
    private String prenom;
    private String email;
}

@Entity
public class Etudiant extends Personne {
    private String numeroEtudiant;
    private Integer promotion;
    @ManyToMany
    @JoinTable(
```

```
        name = "inscription",
        joinColumns = @JoinColumn(name = "etudiant_id"),
        inverseJoinColumns = @JoinColumn(name = "cours_id")
    )
    private Set<Cours> cours = new HashSet<>();
}

@Entity
public class Professeur extends Personne {
    private String specialite;
    private String bureau;
    @OneToMany(mappedBy = "professeur")
    private Set<Cours> coursEnseignes = new HashSet<>();
}

@Entity
public class Administrateur extends Personne {
    private String departement;
    private Set<String> permissions;
}

// SINGLE_TABLE pour Contenu (types variés, requêtes fréquentes)
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "type_contenu")
public abstract class Contenu {
    @Id
    private Long id;
    private String titre;
    @ManyToOne
    private Cours cours;
}

@Entity
@DiscriminatorValue("VIDEO")
public class Video extends Contenu {
    private String url;
    private Integer dureeSecondes;
}

@Entity
@DiscriminatorValue("PDF")
public class DocumentPdf extends Contenu {
    private String fichier;
    private Integer nombrePages;
}

@Entity
@DiscriminatorValue("QUIZ")
public class Quiz extends Contenu {
    private Integer dureeMinutes;
    @OneToMany(mappedBy = "quiz")
    private List<Question> questions;
}
```