

La Gestion des Exceptions en Java



Architecture des Exceptions Java

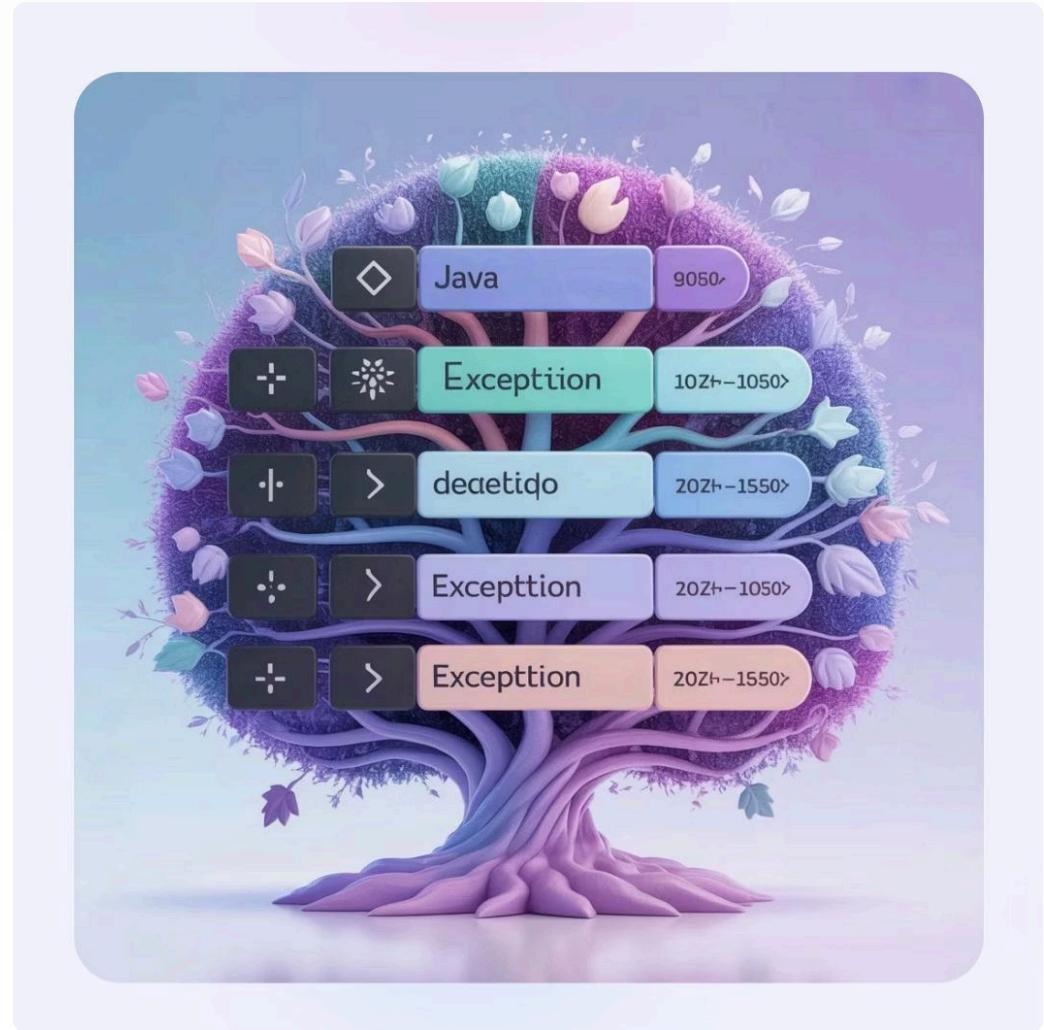
Hiérarchie Structurée

Throwable se divise en deux branches :

- **Error** : erreurs système graves
- **Exception** : conditions récupérables

Exceptions **checked** : déclarées ou capturées

Exceptions **unchecked** : erreurs de programmation



Deux Catégories Fondamentales

Exceptions Métiers

- Violations de règles de gestion
- Données invalides selon contexte
- États métiers incompatibles

Présentation claire à l'utilisateur

Exceptions Techniques

- Erreurs de connexion base de données
- Problèmes réseau ou I/O
- Timeouts et indisponibilités

Stratégies de retry et logging approfondi

Hiérarchie d'Exceptions Personnalisées

Structure bien définie pour améliorer la maintenabilité



```
public abstract class BusinessException extends Exception {  
    private final String errorCode;  
    public BusinessException(String message, String errorCode) {  
        super(message);  
        this.errorCode = errorCode;  
    }  
}
```

Le Pattern d'Exception Wrapping

Encapsuler pour Abstraire

Capturer une exception de bas niveau pour la réencapsuler dans une exception de plus haut niveau

Préserve la stack trace originale

Fournit un contexte sémantique approprié



```
try {  
    connection.executeQuery(sql);  
} catch (SQLException e) {  
    throw new DataAccessException(  
        "Échec de récupération du client", e);  
}
```

Quand Utiliser le Wrapping

01

Franchir une frontière architecturale

Passage entre couches pour maintenir l'abstraction

02

Ajouter du contexte métier

Transformer exceptions techniques en exceptions métier significatives

03

Unifier les APIs tierces

Encapsuler pour réduire le couplage

04

Simplifier la gestion d'erreurs

Consolider les erreurs similaires



Anti-Pattern : Exception Swallowing

Le Pire des Anti-Patterns

Avaler silencieusement une exception masque complètement les problèmes et rend le débogage extrêmement difficile

✗ MAUVAIS

```
try {  
    performCriticalSection();  
} catch (Exception e) {  
    // Silence total  
}
```

✓ CORRECT

```
try {  
    performCriticalSection();  
} catch (Exception e) {  
    logger.error("Échec", e);  
    throw e;  
}
```

Anti-Pattern : Catch Générique

Le Problème

Capturer Exception ou Throwable
attrape tout sans distinction

L'Impact

- Masque les erreurs inattendues
- Rend le débogage difficile
- Peut capturer des interruptions de threads

La Solution

Capturer uniquement les exceptions spécifiques que vous savez gérer

```
// BON
try {
    businessLogic();
} catch (BusinessException | ValidationException e) {
    handleExpectedErrors(e);
}
```

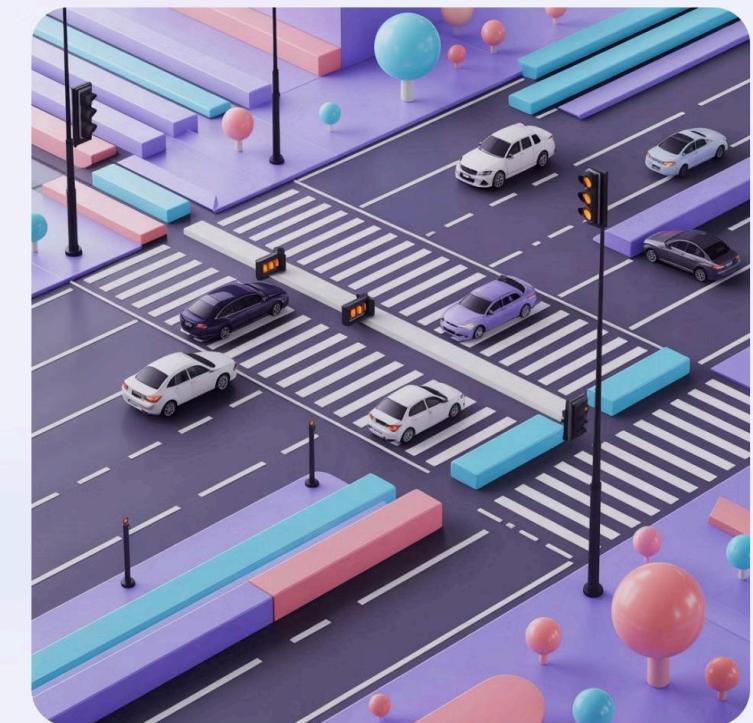
Anti-Pattern : Exception pour le Flux de Contrôle

Exceptions ≠ Logique Métier

Utiliser les exceptions pour contrôler le flux normal est coûteux en performance

~100x plus lent qu'un retour normal

Réservez les exceptions aux conditions véritablement exceptionnelles



```
// BON - Retour explicite
Optional<User> user = findUserById(id);
return user.map(User::getName).orElse("Anonyme");
```



Pattern : Try-With-Resources

Garantit la fermeture automatique des ressources depuis Java 7

Avant Java 7

Verbeux et sujet aux erreurs

```
BufferedReader reader = null;  
try {  
    reader = new  
    BufferedReader(...);  
    return reader.readLine();  
} finally {  
    if (reader != null) {  
        reader.close();  
    }  
}
```

Avec Try-With-Resources

Élégant et sûr

```
try (BufferedReader reader =  
     new BufferedReader(...)) {  
    return reader.readLine();  
} // Fermeture automatique
```

Pattern : Exception Translation

Transformer les exceptions d'une couche en exceptions appropriées à la couche suivante



Couche DAO

SQLException



Couche Service

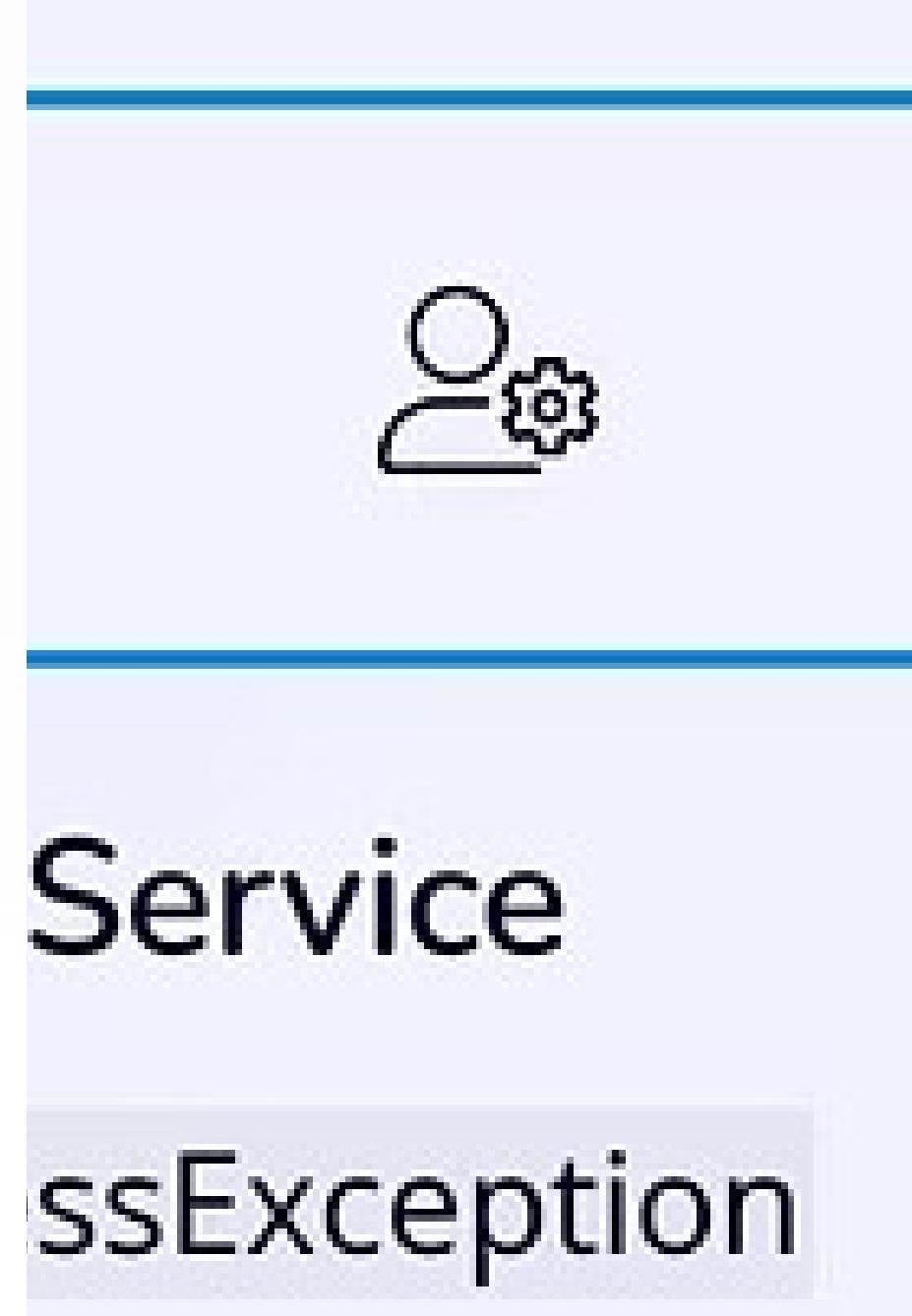
DataAccessException



Couche Controller

BusinessException

Maintient l'encapsulation et préserve la cause racine pour le diagnostic





Pattern : Fail Fast

Échouer Rapidement et Clairement

Déetecter et signaler les erreurs le plus tôt possible

1 Valider immédiatement

Paramètres d'entrée au début des méthodes

2 Vérifier les invariants

Utiliser des assertions pour les états attendus

3 Lancer dès détection

Ne pas continuer avec des données corrompues

```
if (amount.compareTo(BigDecimal.ZERO) <= 0) {  
    throw new IllegalArgumentException("Montant doit être positif");  
}
```

Checked vs Unchecked : Le Débat

Exceptions Checked



Exceptions Unchecked



✓ Avantages

- Contrat explicite
- Force la gestion
- Auto-documentation

✗ Inconvénients

- Verbosité excessive
- Pollution des signatures
- Tentation d'avaler

✓ Avantages

- Code plus concis
- Pas de pollution
- Flexibilité

✗ Inconvénients

- Erreurs non gérées
- Contrat implicite
- Documentation requise

Tendance moderne : **favoriser les unchecked** pour réduire la verbosité

Best Practice : Messages d'Erreur Clairs



Contexte Complet

Valeurs des paramètres, état de l'objet, identifiants de corrélation



Langage Métier

Termes compréhensibles, éviter le jargon technique



Actions Correctives

Suggérer des solutions possibles



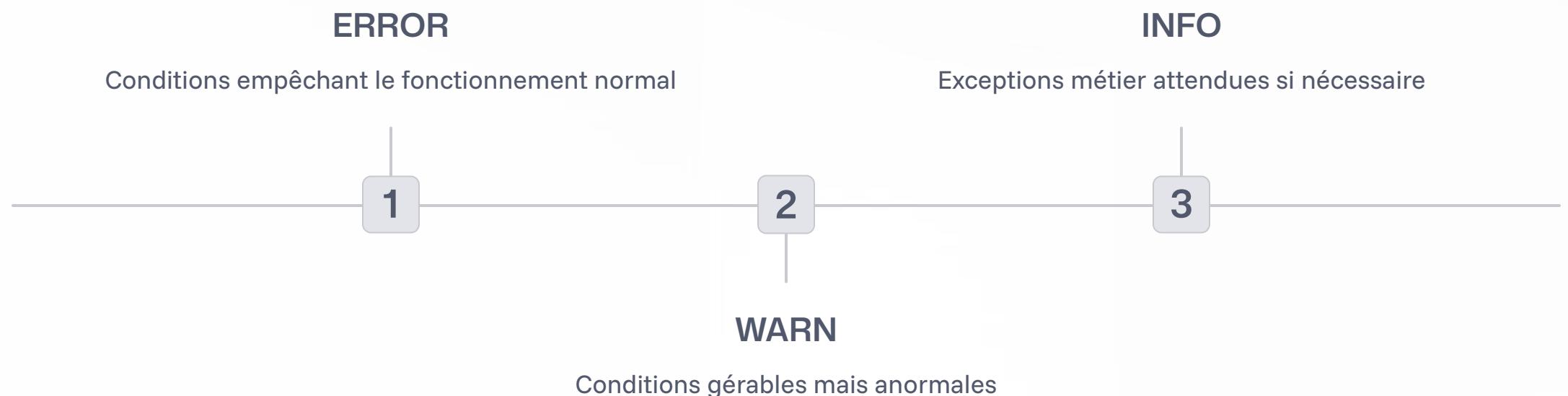
Sécurité

Ne jamais divulguer d'informations sensibles

```
throw new PaymentProcessingException(  
    String.format("Paiement de %s pour commande %s échoué. " +  
        "Vérifiez les informations de carte.", amount, orderId),  
    paymentGatewayException  
);
```



Best Practice : Logging Approprié



- ❑ **Règle d'or :** Logger une exception une seule fois, à la frontière de gestion

```
logger.error("Erreur traitement paiement commande {}: {}",  
orderId, e.getMessage(), e);
```

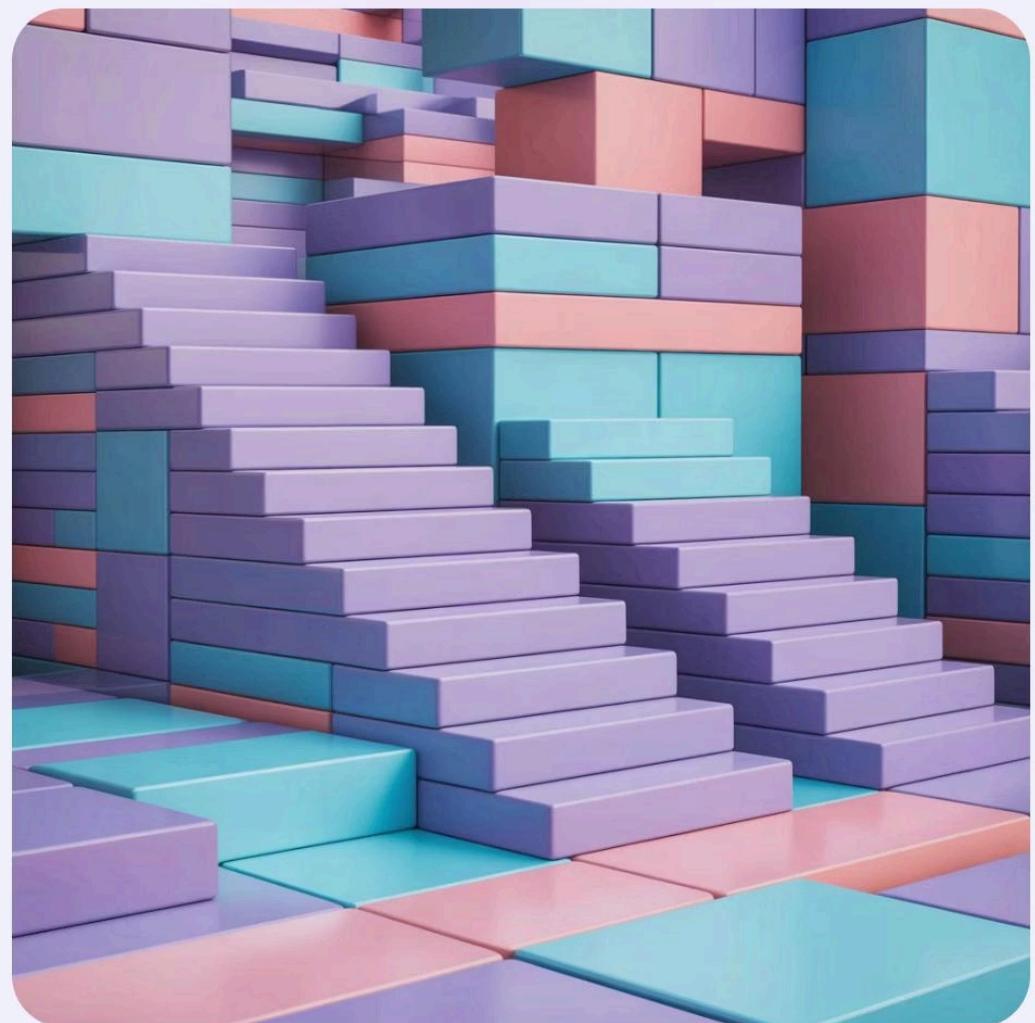
Pattern : Exception Enrichment

Enrichir le Contexte

Ajouter progressivement du contexte métier à mesure que l'exception remonte

Chaque couche ajoute ses informations spécifiques

Particulièrement utile dans les systèmes distribués



```
public class EnrichedBusinessException extends RuntimeException {  
    private final Map<String, Object> context = new HashMap<>();  
    public void addContext(String key, Object value) {  
        context.put(key, value);  
    }  
}
```

Pattern : Circuit Breaker

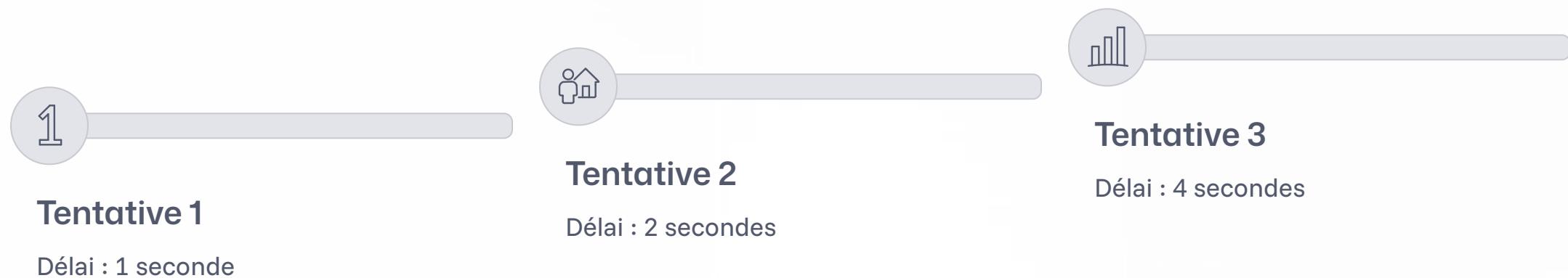
Protège contre les défaillances en cascade dans les architectures microservices



```
@CircuitBreaker(name = "paymentService",
fallbackMethod = "paymentFallback")
public Payment processPayment(PaymentRequest request) {
    return paymentServiceClient.process(request);
}
```

Pattern : Retry avec Backoff Exponentiel

Pour les exceptions transitoires : timeouts réseau, erreurs temporaires



```
@Retry(name = "database",
maxAttempts = 3,
retryExceptions = {TransientDataAccessException.class})
public User fetchUser(Long id) { ... }
```



Best Practice : Documentation des Exceptions

Documenter pour Communiquer

Documenter toutes les exceptions, même les unchecked, avec @throws dans la JavaDoc

```
/**  
 * Traite un paiement client.  
 *  
 * @param request détails du paiement  
 * @return le paiement traité  
 * @throws PaymentValidationException si données invalides  
 * @throws InsufficientFundsException si solde insuffisant  
 * @throws PaymentGatewayException si passerelle échoue  
 */  
public Payment process(PaymentRequest request) { ... }
```

Incluez les conditions de déclenchement et comment les gérer



Anti-Pattern : Exception Tunneling

Le Problème des Tunnels d'Exceptions

Wrapper une checked dans une unchecked pour éviter la déclaration, puis unwrapper plus haut

✗ Tunneling Artificiel

```
public void businessMethod()
{
    try {
        riskyOperation();
    } catch (CheckedException e)
    {
        throw new
        RuntimeException(e);
    }
}
```

✓ Exception Métier

```
public void businessMethod()
{
    try {
        riskyOperation();
    } catch (CheckedException e)
    {
        throw new
        BusinessException(
            "Échec opération", e);
    }
}
```

Pattern : Exception Chaining

Maintenir la Trace de Causalité

Chaque exception wrappée ajoute son contexte tout en préservant la cause racine



Couche Présentation

BusinessException : "Échec traitement commande"

Couche Service

DataAccessException : "Erreur accès données"

Couche DAO

SQLException : "Connection timeout"

Utilisez : new MyException("message", cause)

option Originale

ception: Connection timeout

opped Exception

ccessException causée par SQLI

ness Exception

nerServiceException causée par

Best Practice : Validation et Exceptions

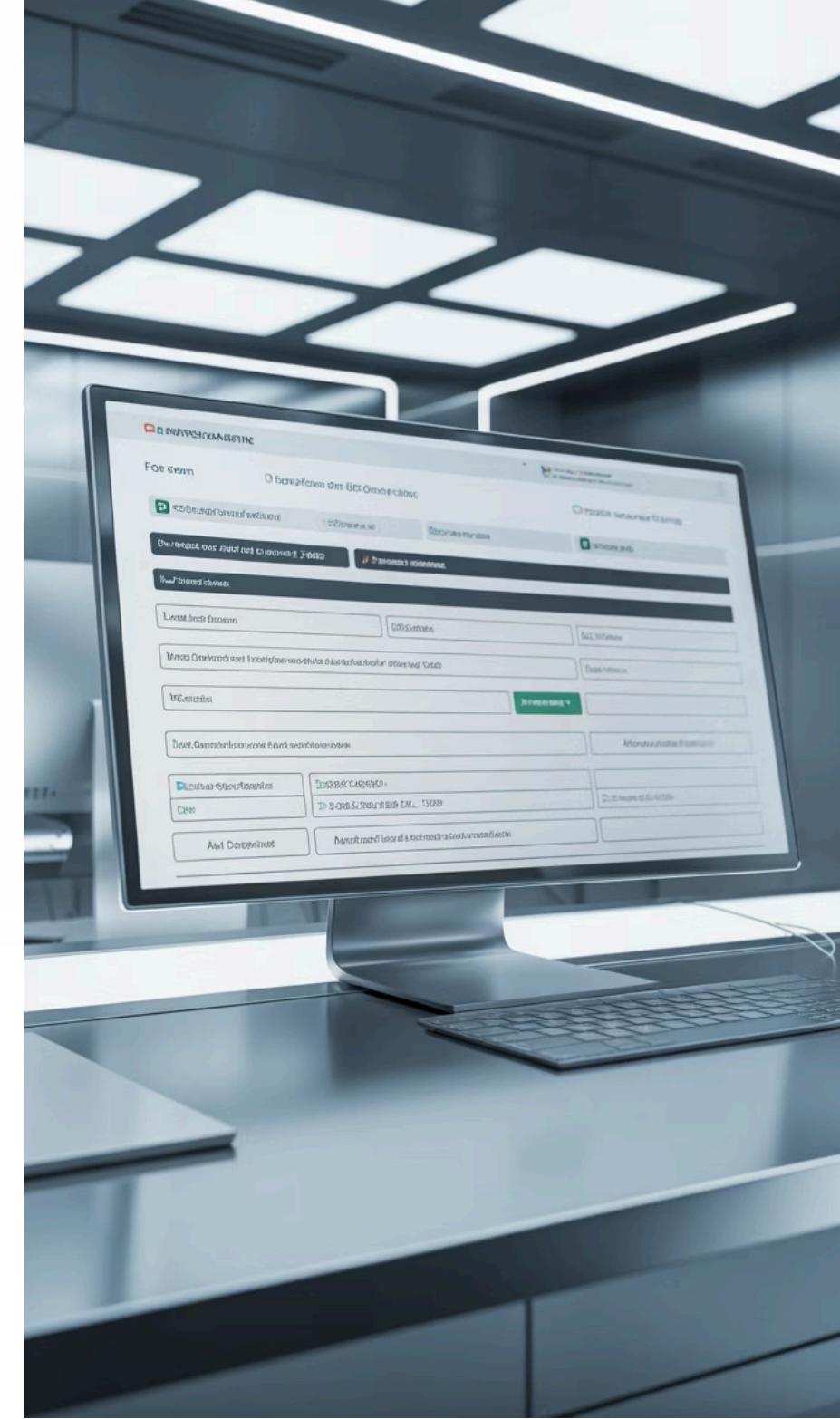
Validation systématique avec Bean Validation (JSR-380)

```
public class PaymentRequest {  
    @NotNull(message =  
            "Montant requis")  
    @Positive  
    private BigDecimal amount;  
  
    @NotBlank  
    @Size(min = 5, max = 20)  
    private String customerId;  
  
    @Email(message = "Email  
invalide")  
    private String email;  
}
```

Avantages

- Validation déclarative
- Messages cohérents
- Réutilisable
- Standardisé

Distinguez validation syntaxique
(annotations) de validation
sémantique (logique métier)



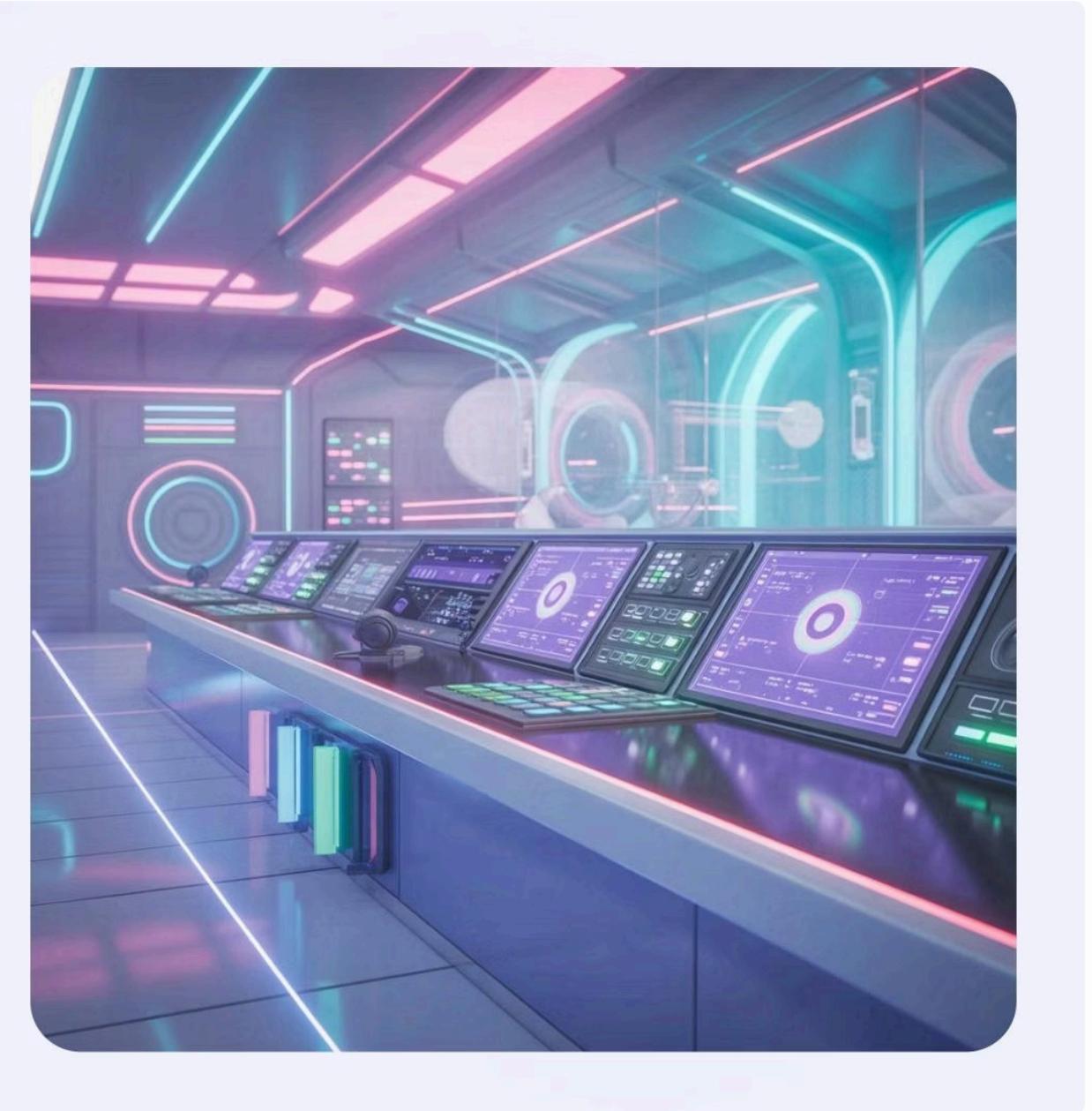
Pattern : Global Exception Handler

Centraliser la Gestion

Intercepte toutes les exceptions non gérées

Transforme en réponses appropriées

Utilise @ControllerAdvice et @ExceptionHandler



```
@ControllerAdvice  
public class GlobalExceptionHandler {  
    @ExceptionHandler(BusinessException.class)  
    public ResponseEntity<ErrorResponse> handleBusiness(  
        BusinessException ex) {  
        return ResponseEntity.status(BAD_REQUEST)  
            .body(new ErrorResponse(ex.getErrorCode(), ex.getMessage()));  
    }  
}
```



Best Practice : Exceptions et Transactions

Attention particulière dans un contexte transactionnel

1

Rollback par Défaut

Uniquement pour unchecked exceptions en Spring

2

Configuration Explicite

Utiliser rollbackFor pour checked exceptions

3

Transactions Imbriquées

Attention aux propagations et isolations

```
@Transactional(rollbackFor = {BusinessException.class})
public void processOrder(Order order) throws BusinessException {
    orderRepository.save(order);
    if (!inventoryService.reserve(order.getItems())) {
        throw new InsufficientStockException(order.getId());
    }
}
```

Pattern : Null Object vs Exception

Utiliser Null Object (Optional)

- L'absence est un cas normal et fréquent
- Un comportement par défaut fait sens
- Éviter les null checks répétitifs

Exemple : CustomerRepository.findById() retourne Optional

Lancer Exception

- L'absence est exceptionnelle et inattendue
- Le flux normal ne peut pas continuer
- Un diagnostic explicite est nécessaire

Exemple : CustomerService.getRequiredCustomer() lance
CustomerNotFoundException

Ces approches sont complémentaires, choisissez selon le contexte

Anti-Pattern : Ignorer les InterruptedException

Ne Jamais Ignorer les Interruptions

Signal qu'un thread doit s'arrêter proprement

✗ MAUVAIS

```
try {  
    Thread.sleep(1000);  
} catch (InterruptedException  
e) {  
    // Ignore l'interruption  
}
```

✓ BON

```
try {  
    Thread.sleep(1000);  
} catch (InterruptedException  
e) {  
  
    Thread.currentThread().interr  
upt();  
    throw new  
    RuntimeException(  
        "Opération interrompue", e);  
}
```

Toujours restaurer le flag d'interruption avec
Thread.currentThread().interrupt()



Best Practice : Exceptions dans les Streams

Gérer les Exceptions Checked

Les lambdas ne peuvent pas lancer d'exceptions checked directement

Approche 1: Wrapper

```
list.stream()
    .map(item -> {
        try {
            return process(item);
        } catch (CheckedException e) {
            throw new
                UncheckedIOException(e);
        }
    })
    .collect(toList());
```

Approche 2 : Méthode Utilitaire

```
list.stream()
    .map(unchecked(this::process))
    .collect(toList());

static <T, R> Function<T, R>
unchecked(CheckedFunction<
    T, R> f) {
    return t -> {
        try { return f.apply(t); }
        catch (Exception e) {
            throw new
                RuntimeException(e);
        }
    };
}
```



Pattern : Error Codes vs Exceptions

Error Codes

Pour : APIs REST, internationalisation, classification, métriques

```
public enum ErrorCode {  
    INSUFFICIENT_FUNDS("BUS_001"),  
    INVALID_ACCOUNT("BUS_002");  
}
```

Exceptions

Pour : Logique interne, propagation automatique, stack traces, typage fort

Approche Hybride

Combinez : Exceptions avec codes intégrés, conversion en bordure



Best Practice : Performance et Exceptions

100x

1000+

2KB

Coût Relatif

Lancer une exception est ~100x plus lent qu'un retour normal

Stack Frames

Une stack trace peut contenir 1000+ frames en environnement complexe

Mémoire

Chaque exception consomme ~2KB pour la stack trace

- Dans les chemins critiques haute performance, considérez Optional ou objets de résultat. Mais ne sacrifiez pas la clarté pour des micro-optimisations prématurées.

Excellence en Gestion d'Exceptions

Plus qu'un mécanisme technique : un outil de communication, une stratégie de résilience, un élément clé de qualité

