



Java : La représentation du temps avec java.time



L'évolution de la gestion du temps en Java

✗ L'ancienne approche

Date et Calendar : design contre-intuitif, objets mutables, gestion complexe des fuseaux horaires

- Erreurs fréquentes dans les calculs
- Fragmentation avec Joda-Time
- Complexité accrue des projets

✓ La révolution java.time

Java 8 : package inspiré de Joda-Time, conforme ISO 8601

- Immutabilité et thread-safety
- Clarté sémantique
- Gestion robuste des calculs

Architecture du package java.time

Classes principales

LocalDate, LocalTime, LocalDateTime pour dates et heures sans fuseau horaire

Durées et périodes

Duration, Period pour intervalles de temps précis

Gestion des zones

ZonedDateTime, OffsetDateTime pour timestamps avec contexte géographique

Instant et précision

Instant pour points dans le temps avec précision nanoseconde

Architecture modulaire permettant de choisir exactement la classe appropriée selon le contexte. Chaque classe a une responsabilité claire et bien définie.



LocalDate : Travailler avec les dates

Représente une date sans heure ni fuseau horaire. Idéale pour anniversaires, dates d'échéance ou événements indépendants du moment de la journée.

01

Création

`LocalDate.now()`, `LocalDate.of(2024, 1, 15)`, `LocalDate.parse("2024-01-15")`

02

Manipulation

`plusDays()`, `minusMonths()`, `withYear()`
- API fluent intuitive

03

Comparaison

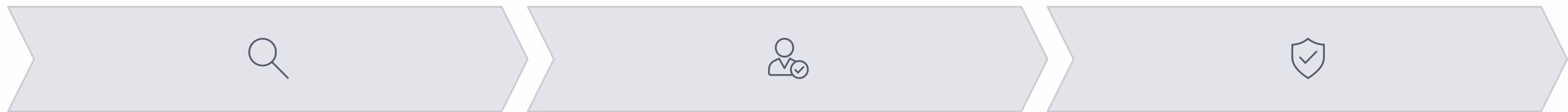
`isBefore()`, `isAfter()`, `isEqual()` - booléens clairs et expressifs



Bonne pratique : Validez toujours les dates entrées avec try-catch autour de `parse()`. Évitez TIMESTAMP en base - préférez DATE.



Pattern : Choisir la bonne classe temporelle



Analysez votre besoin

Date seule ? Heure seule ? Date et heure ? Besoin de fuseau horaire ?

Sélectionnez la classe

LocalDate pour dates, LocalTime pour heures, ZonedDateTime pour timestamps complets

Validez la cohérence

Assurez-vous que la classe correspond au domaine métier

Le choix approprié évite les bugs subtils. Un anniversaire avec `ZonedDateTime` introduit une complexité inutile. Un rendez-vous international avec `LocalDateTime` ignore le contexte géographique essentiel.

LocalTime : Gérer les heures sans date

Représente une heure de la journée sans date ni fuseau horaire. Parfaite pour horaires d'ouverture, durées de films ou heures de cours.

- Précision jusqu'à la nanoseconde
- LocalTime.of(14, 30) ou LocalTime.now()
- plusHours(), minusMinutes() pour calculs
- Duration.between() pour durées

Attention : LocalTime ne gère pas les débordements de minuit automatiquement



Antipattern : N'utilisez pas LocalTime pour mesurer des durées d'exécution. Préférez Instant et Duration pour cette tâche.



LocalDateTime : Combiner date et heure

Fusionne LocalDate et LocalTime en une seule classe. Appropriée pour événements locaux : rendez-vous médicaux, réservations, cours universitaires.

Construction

LocalDateTime.of(2024, 1, 15, 14, 30) ou combinaison LocalDate + LocalTime

Manipulation fluente

withYear(2025).plusMonths(2).withHour(16) - opérations chainables

Extraction

toLocalDate() et toLocalTime() pour composants individuels

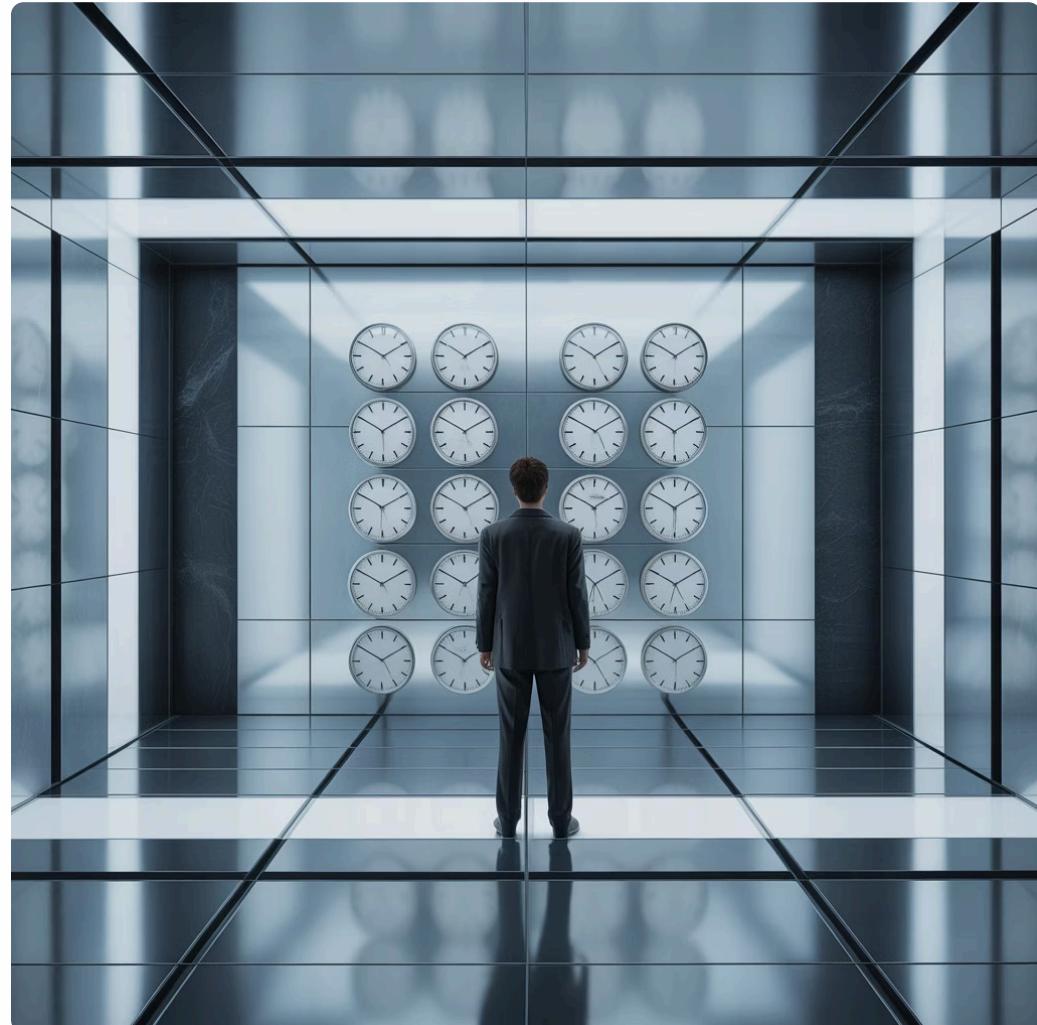
Erreur classique : Utiliser LocalDateTime pour événements internationaux. Sans fuseau horaire, "14:30 le 15 janvier" est ambigu dès qu'on traverse des frontières.

Antipattern : Ignorer les fuseaux horaires

✗ Le problème

LocalDateTime pour événements internationaux crée des ambiguïtés impossibles à résoudre

- Confusion entre pays
- Bugs lors des changements d'heure d'été
- Conversions impossibles



✓ La solution

Utilisez ZonedDateTime dès que le contexte géographique compte

- Clarté sémantique du moment exact
- Gestion automatique des DST
- Conversions fiables entre zones



Règle d'or : Si votre application a des utilisateurs dans plus d'un fuseau horaire, utilisez systématiquement ZonedDateTime.

ZonedDateTime : Le timestamp complet

La classe la plus complète de java.time : point précis dans le temps avec date, heure et fuseau horaire. Gère automatiquement les changements d'heure d'été et conversions entre zones.



Création avec ZoneId

```
ZonedDateTime.now(ZoneId.of("Europe/Paris"))
```



Conversion automatique

withZoneSameInstant() convertit vers autre fuseau



Stockage robuste

TIMESTAMP WITH TIME ZONE en base de données

- Important :** withZoneSameInstant() garde le même moment absolu. withZoneSameLocal() garde l'heure locale mais change le moment - utilisez avec précaution.





Pattern : Stratégie de stockage des timestamps

01

Capturer en UTC

Convertissez immédiatement les entrées utilisateur en UTC avec Instant ou ZonedDateTime en UTC

02

Stocker avec contexte

Enregistrez le fuseau horaire original si nécessaire pour l'affichage ultérieur

03

Calculer en UTC

Effectuez tous les calculs de durée et comparaisons en UTC ou Instant

04

Afficher en local

Convertissez vers le fuseau horaire de l'utilisateur uniquement au moment de l'affichage

Cette stratégie élimine la majorité des bugs liés aux fuseaux horaires en créant une source de vérité unique, indépendante des changements d'heure d'été.

Instant : Le point temporel absolu



Point précis sur la ligne du temps, mesuré en nanosecondes depuis l'époque Unix (1er janvier 1970, 00:00:00 UTC)

- Idéal pour timestamps système et logs
- Mesures de performance précises
- Travaille uniquement avec UTC
- Extrêmement efficace pour comparaisons

Création

```
Instant.now(), Instant.parse("2024-01-15T13:30:00Z")
```

Conversion

```
instant.atZone(Zoneld) → ZonedDateTime, zdt.toInstant() → Instant
```

Usage recommandé

Événements système, logs d'audit, création/modification d'entités

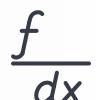


Duration : Mesurer le temps écoulé



Précision temporelle

Duration mesure des durées en secondes et nanosecondes, idéal pour intervalles courts et mesures précises



Calculs de différence

Entre deux Instant, LocalTime ou LocalDateTime, Duration calcule automatiquement l'écart temporel



Manipulation fluente

Ajoutez, soustrayez, multipliez des durées avec une API intuitive et chainable

Distinction fondamentale : Duration.ofDays(1) = exactement 24 heures.
Period.ofDays(1) = "un jour" qui peut faire 23, 24 ou 25 heures selon les changements d'heure.

Period : Travailler avec les périodes calendaires

Représente une quantité de temps en années, mois et jours calendaires.

Respecte les variations : mois de différentes longueurs, années bissextiles, changements d'heure.

Création intuitive

`Period.ofYears(1),
Period.ofMonths(3), Period.of(1, 2,
15)`

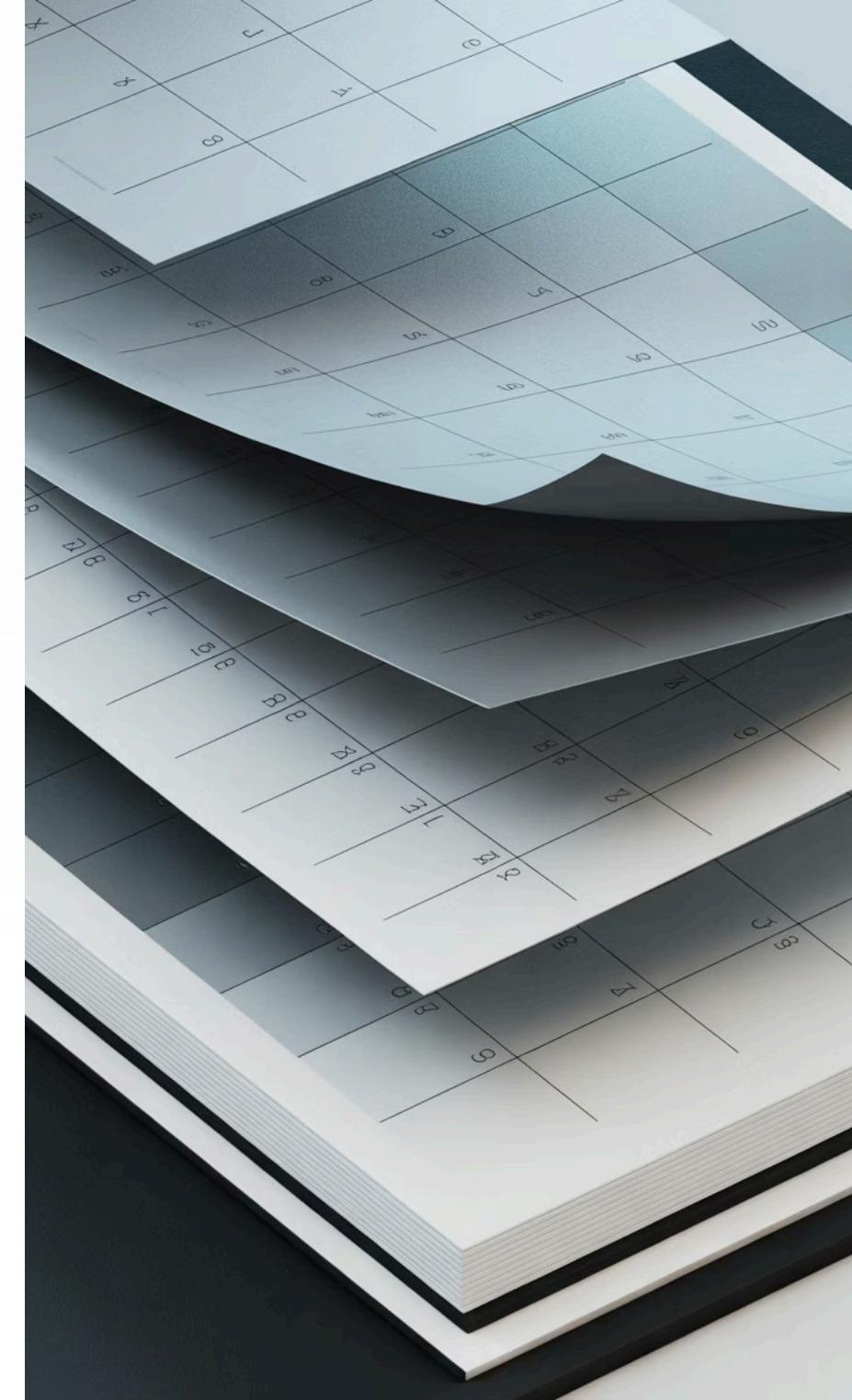
Calcul entre dates

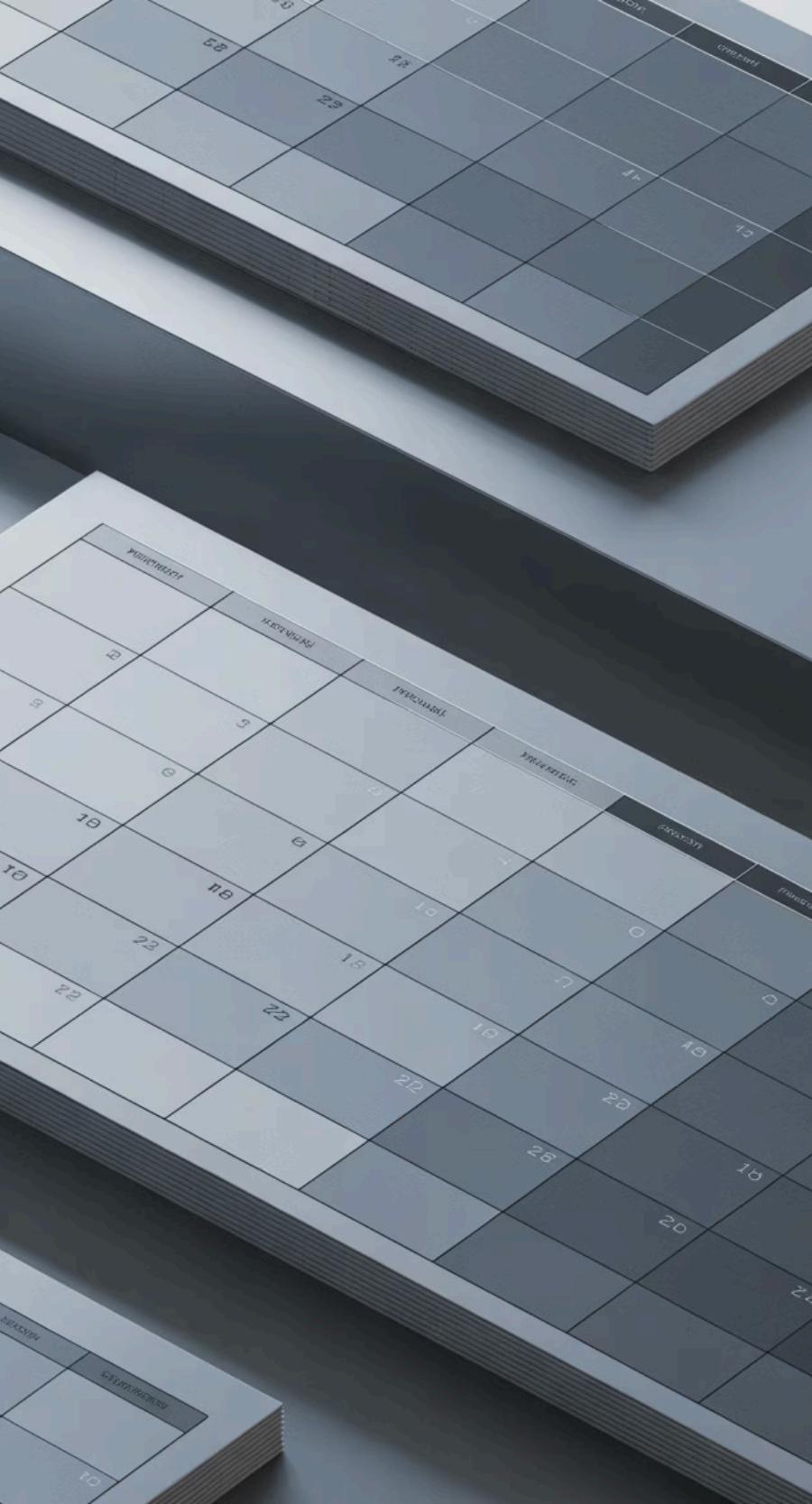
`Period.between(date1, date2)`
retourne objet normalisé

Respect du calendrier

$31 \text{ janvier} + 1 \text{ mois} = 28 \text{ février} (\text{ou } 29 \text{ en année bissextile})$

- ❑ **Évitez :** Ne convertissez jamais Period en Duration ou vice-versa - ce sont des concepts fondamentalement différents.





Antipattern : Confusion entre Duration et Period

✗ Erreur courante

```
// MAUVAIS  
Duration unMois =  
Duration.ofDays(30);  
LocalDateTime expiration =  
maintenant.plus(unMois);
```

```
// Problème : 30 jours != 1  
mois  
// Février fait 28/29 jours
```

Period ne convertit pas en temps
absolu !

✓ Approche correcte

```
// BON  
Period unMois =  
Period.ofMonths(1);  
LocalDate expiration =  
aujourdHui.plus(unMois);
```

```
// Respecte les limites des  
mois  
// 31 janvier + 1 mois = 28  
février
```

Duration pour temps absolu précis

Règle mnémotechnique : Duration pour chronomètre, Period pour calendrier.



Pattern : Parsing robuste des dates

1 Validation stricte

Utilisez toujours try-catch autour de parse() pour gérer DateTimeParseException proprement

2 Formatters explicites

Définissez des DateTimeFormatter personnalisés pour les formats non-ISO

3 Feedback utilisateur

Retournez des messages d'erreur clairs indiquant le format attendu

Ce pattern permet une flexibilité contrôlée : l'utilisateur peut entrer plusieurs formats, mais tous sont validés strictement. Documentez clairement les formats acceptés.



DateTimeFormatter : Personnaliser l'affichage

Classe centrale pour convertir les objets temporels en chaînes lisibles et vice-versa. Supporte formats ISO, patterns personnalisés et localisation.



Formatters prédéfinis

ISO_LOCAL_DATE,
ISO_LOCAL_DATE_TIME,
ISO_ZONED_DATE_TIME - thread-safe
et réutilisables



Patterns personnalisés

```
ofPattern("dd/MM/yyyy  
HH:mm").withLocale(Locale.FRENCH)
```



Localisation

'EEEE dd MMMM yyyy' adapte noms de mois et jours selon la locale

- ❑ **Bonne pratique :** Définissez vos formatters comme constantes static final réutilisables. La création est coûteuse.

ChronoUnit : Calculs et mesures flexibles

Calcul de différence

ChronoUnit.DAYS.between(date1, date2)
sans objet intermédiaire



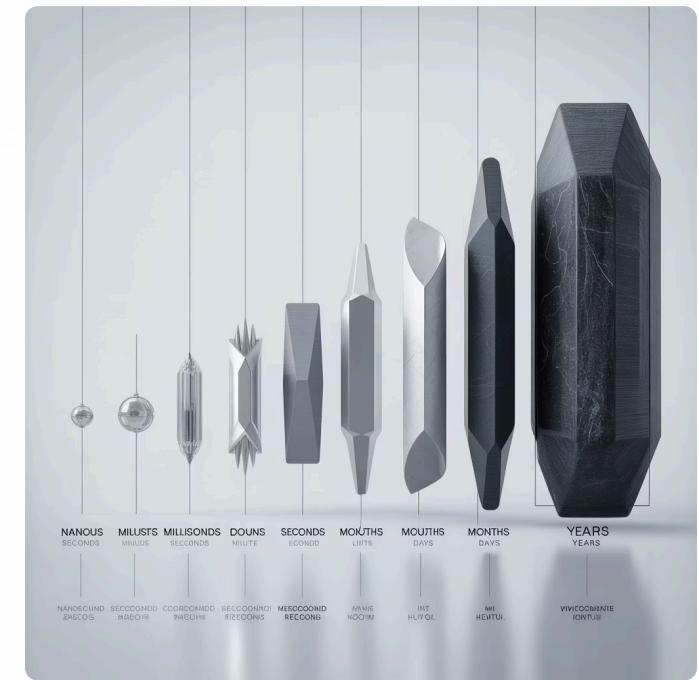
Addition temporelle

date.plus(5, ChronoUnit.WEEKS) -
expressif et lisible



Unités variées

De NANOS à ERAS, toutes les unités
temporelles imaginables



ChronoUnit offre une alternative plus directe pour certains calculs. La méthode truncatedTo() est particulièrement utile pour normaliser les timestamps avant comparaison.

TemporalAdjusters : Manipulations avancées

Méthodes statiques pour ajustements complexes : premier lundi du mois, dernier jour de l'année, prochain jour ouvré.

Début/fin de période

`firstDayOfMonth()`, `lastDayOfMonth()`, `firstDayOfYear()`

Navigation jours

`next(DayOfWeek)`, `previous(DayOfWeek)`, `nextOrSame()`

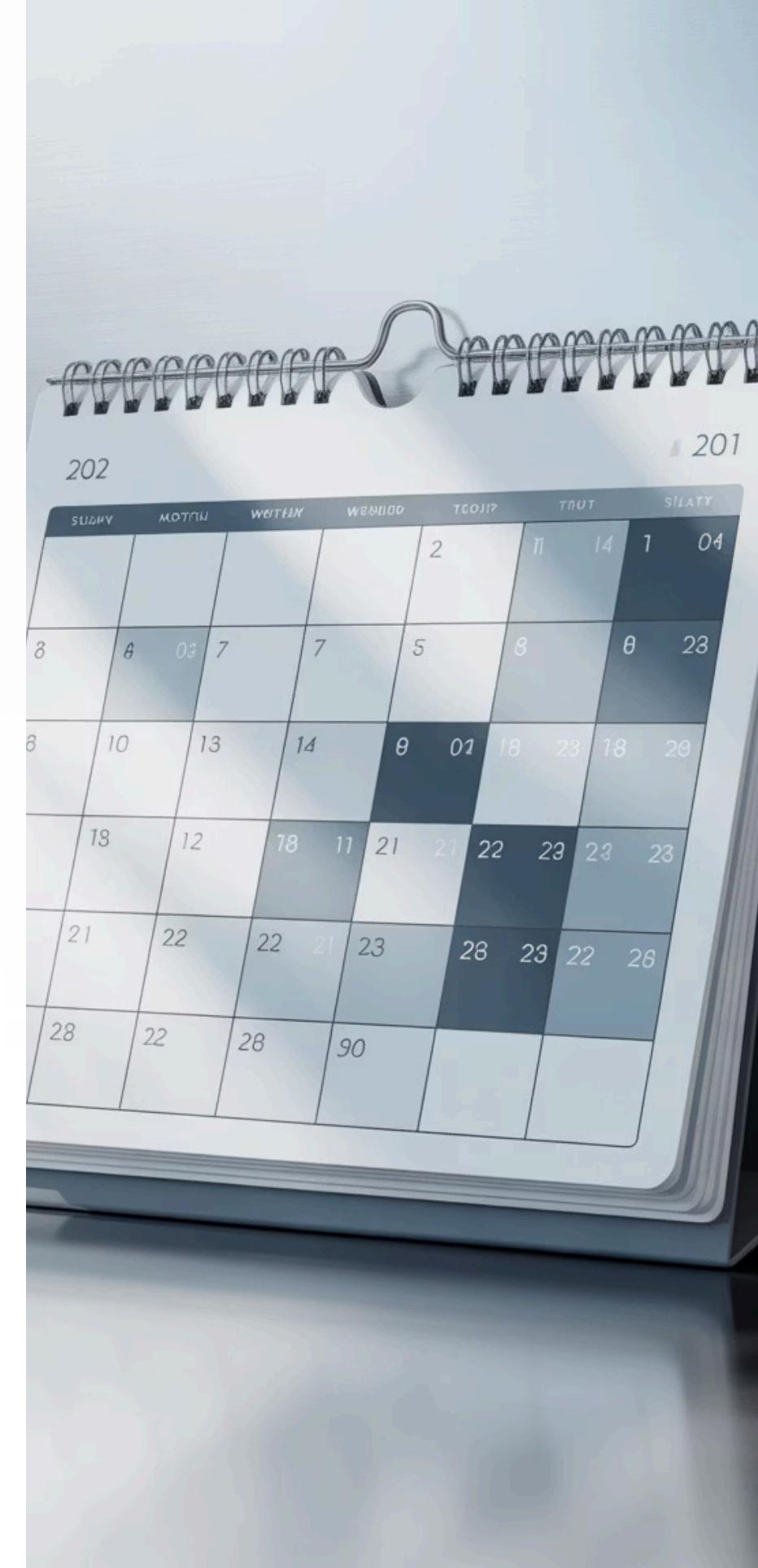
Recherche dans mois

`firstInMonth(DayOfWeek)`, `lastInMonth(DayOfWeek)`

Adjusters personnalisés

Créez vos propres logiques métier : jours ouvrés, règles comptables

Les adjusters rendent le code métier expressif. Au lieu d'une boucle complexe, écrivez
`date.with(TemporalAdjusters.next(DayOfWeek.MONDAY))`.



Antipattern : Arithmétique manuelle des dates

✗ Code fragile

```
// MAUVAIS : Logique manuelle  
int mois =  
date.getMonthValue();  
mois++;  
if (mois > 12) {  
    mois = 1;  
    annee++;  
}  
  
// Problème : ignore jours invalides  
// 31 janvier + 1 mois = 31 février ???  
  
LocalDate resultat =  
    LocalDate.of(annee, mois,  
jour);  
  
// CRASH : DateTimeException
```

✓ Code robuste

```
// BON : API java.time  
LocalDate date =  
LocalDate.now();  
LocalDate resultat =  
date.plusMonths(1);  
  
// Gère automatiquement :  
// - Changement d'année  
// - Jours invalides (31 jan → 28 fév)  
// - Années bissextiles  
// - Tous les cas limites
```



Ne réinventez jamais la roue temporelle. java.time est testé exhaustivement et gère tous les cas limites.



Pattern : Gestion des jours fériés



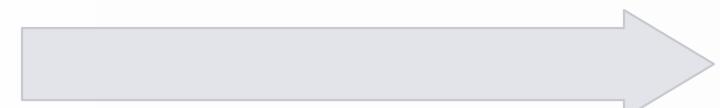
Définir les jours fériés

Set de LocalDate pour fériés fixes +
calcul des fériés mobiles (Pâques, etc.)



Vérification

Méthode estFerie(LocalDate) vérifie
fériés fixes et mobiles



Calcul jours ouvrés

ajouterJoursOuvres() exclut weekends
et jours fériés dans les calculs

Essentiel pour applications métier : délais légaux, dates de livraison, plannings de projet. Considérez des bibliothèques comme jollyday pour gestion multi-pays.

Testing des comportements temporels

Tester du code dépendant du temps est difficile : tests qui passent à une heure et échouent à une autre. `java.time` offre des mécanismes pour contrôler le temps.



Injecter Clock

Utilisez `Clock` au lieu d'appeler directement `now()`



Figter le temps

`Clock.fixed()` pour tests déterministes



Simuler le passage

`Clock.offset()` pour avancer dans le temps

- ❑ **Pattern avancé :** Créez un `MockClock` qui peut avancer le temps programmatiquement avec `tick(Duration)` pour tester timeouts et expirations.





Performance et optimisation

2x

Parsing vs construction

LocalDate.of() est 2x plus rapide que parse()

100K

Cache des Zoneld

Zoneld.of() cache les instances - réutilisation sans allocation

1M

Operations/seconde

Plus d'un million d'opérations simples par seconde

Pour la majorité des applications, la performance de java.time est largement suffisante. Mesurez d'abord avec un profiler avant d'optimiser prématulement.

Les vrais problèmes de performance viennent généralement des I/O, requêtes base de données ou algorithmes inefficaces, pas des opérations temporelles.

Persistance et sérialisation

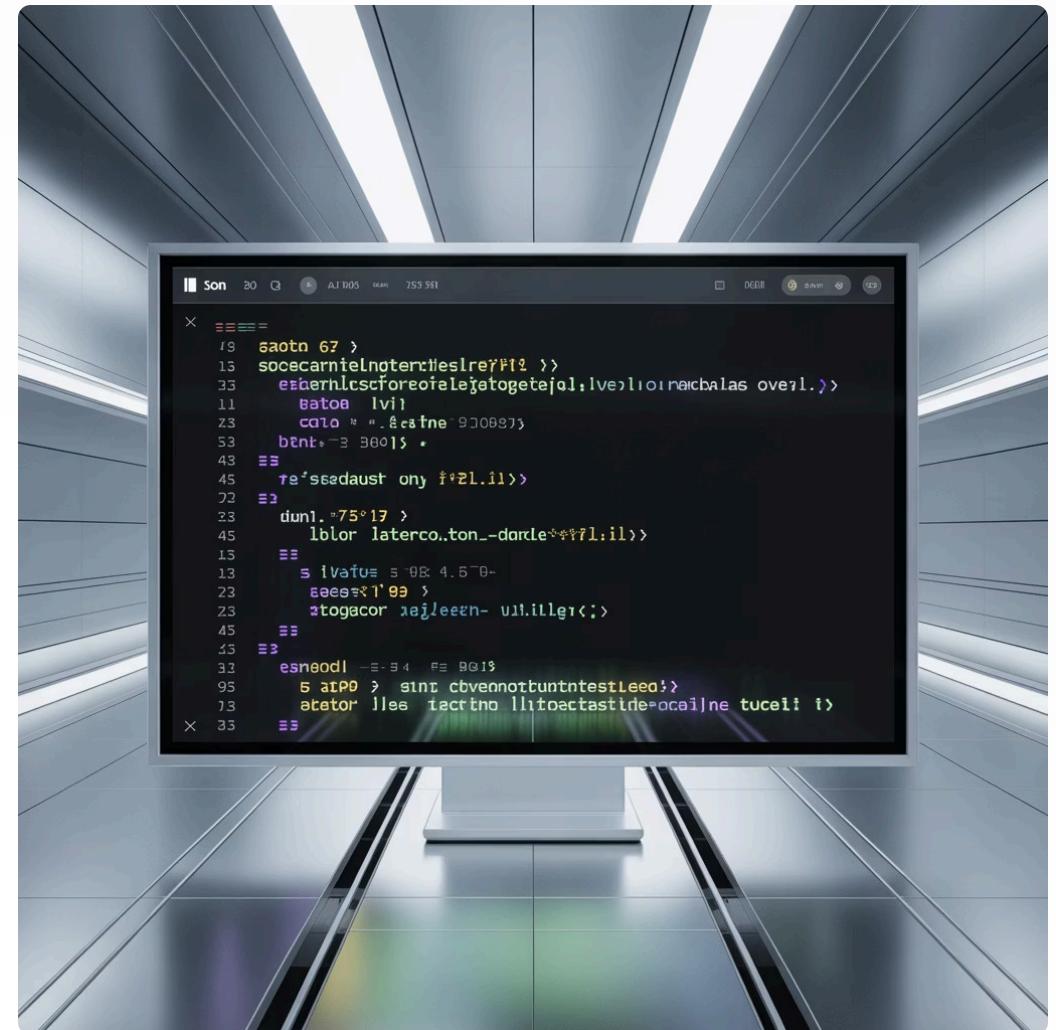
Base de données

- Stockez en UTC avec TIMESTAMP WITH TIME ZONE
- JDBC 4.2+ : setObject() et getObject() supportent java.time
- JPA/Hibernate 5+ : mapping automatique
- Vérifiez dialecte SQL pour support correct



JSON avec Jackson

- Module jackson-datatype-jsr310 requis
- Désactivez WRITE_DATES_AS_TIMESTAMPS
- Format ISO-8601 pour interopérabilité
- @JsonFormat pour personnalisation



Normalisez systématiquement en UTC au point d'entrée pour éliminer les ambiguïtés et simplifier les requêtes.

Checklist des bonnes pratiques

- **Choix de la classe**

LocalDate pour dates, ZonedDateTime pour international, Instant pour système, Duration pour chronomètre, Period pour calendrier

- **Immutabilité**

Assignez toujours le résultat : date = date.plusDays(1). Partagez librement entre threads

- **Fuseaux horaires**

Stockez en UTC, affichez en local. Utilisez ZoneId.of("Europe/Paris") pas des offsets

- **Parsing et formatage**

Try-catch autour de parse(). Privilégiez ISO-8601. Spécifiez la Locale pour textes

- **Tests**

Injectez Clock pour testabilité. Clock.fixed() pour déterminisme. Testez cas limites

Vers la maîtrise de java.time

La gestion du temps en programmation est intrinsèquement complexe. `java.time` transforme cette complexité en une API élégante, expressive et fiable.

Approfondir

Documentation officielle Java, JSR 310,
cas d'usage avancés

Évoluer

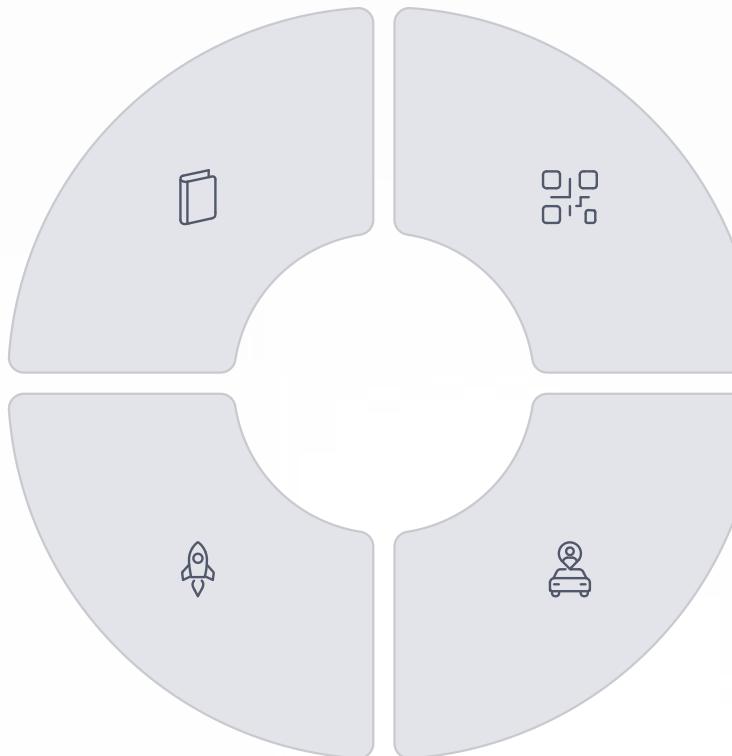
Restez informé des évolutions, explorez
bibliothèques complémentaires

Pratiquer

Refactorisez code legacy, écrivez tests
exhaustifs, créez utilitaires réutilisables

Partager

Formez votre équipe, établissez
standards, effectuez code reviews
focalisées



En maîtrisant `java.time`, vous ne devenez pas seulement meilleur avec les dates - vous devenez un meilleur développeur Java.