

Maîtriser Optional en Java

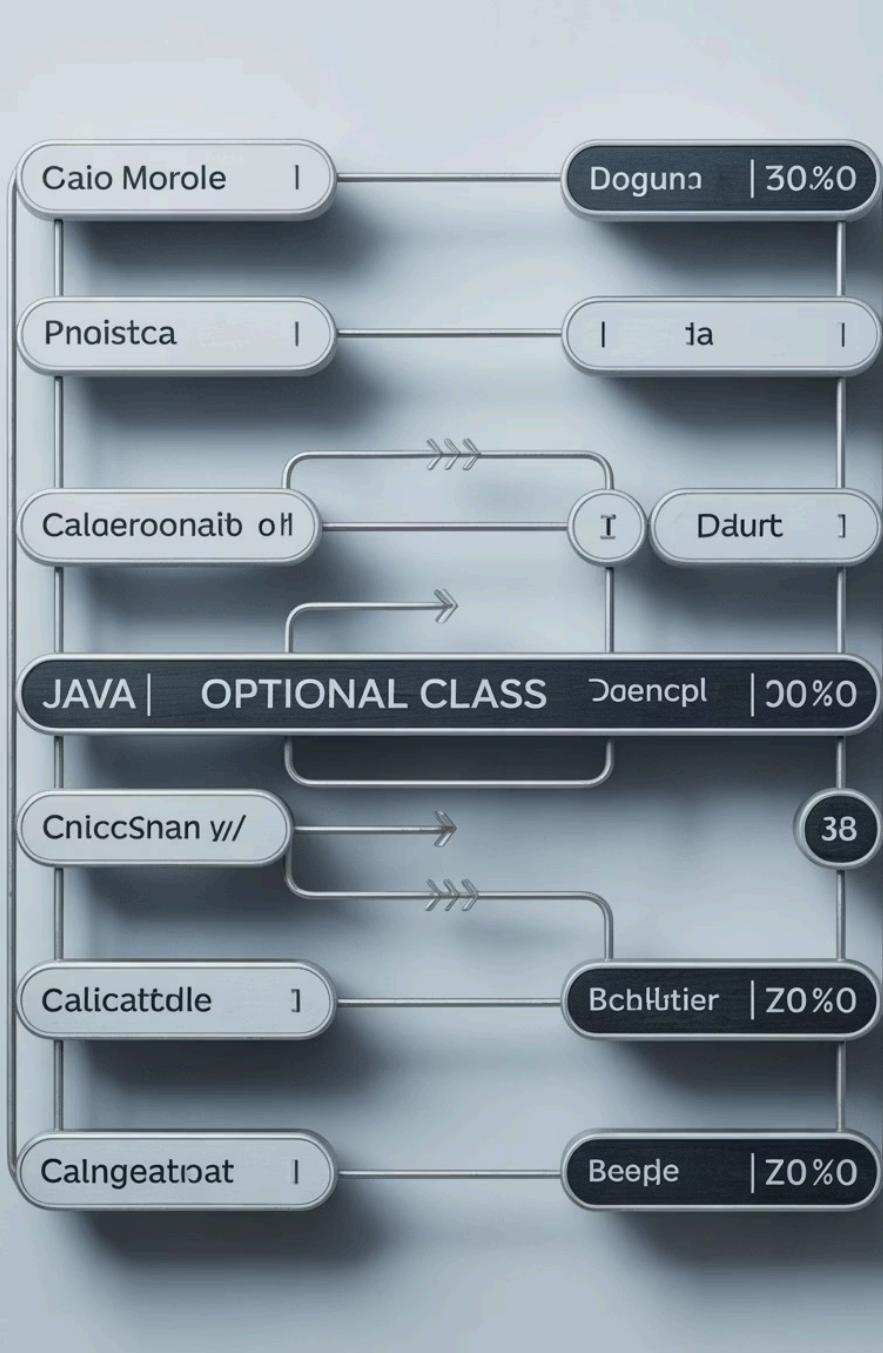
Le problème du null en Java

Un fléau historique

Tony Hoare : "erreur à un milliard de dollars"

- Vérifications défensives omniprésentes
- Code pollué et moins lisible
- Aucune garantie contre les NPE





Qu'est-ce qu'Optional ?



Conteneur typé

Peut contenir ou non une valeur non-nulle



Protection intégrée

Empêche l'accès direct sans vérification



Expressivité accrue

Rend explicite l'absence potentielle

Créer un Optional

01

Optional.of(value)

Valeur non-nulle garantie. Lance NPE si null.

02

Optional.ofNullable(value)

Tolère null, crée Optional vide si nécessaire.

03

Optional.empty()

Crée explicitement un Optional vide.

```
Optional<String> opt1 = Optional.of("Hello");
Optional<String> opt2 = Optional.ofNullable(getValue());
Optional<String> opt3 = Optional.empty();
```

Pattern : Retourner Optional

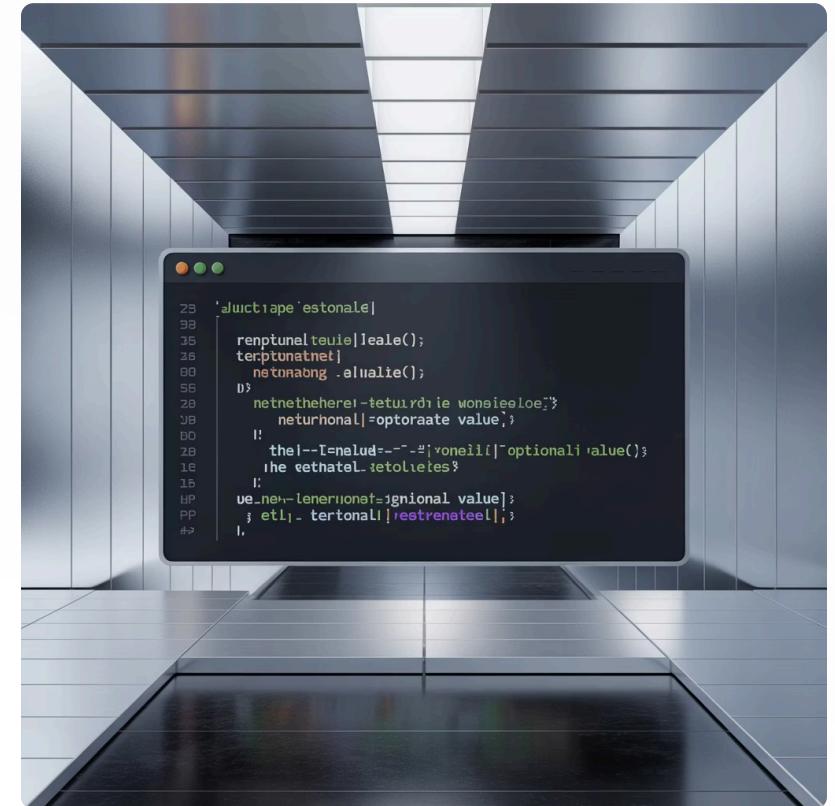
Rendre le contrat de méthode explicite

```
public Optional<User> findUserById(int id) {  
    for (User user : users) {  
        if (user.getId() == id) {  
            return Optional.of(user);  
        }  
    }  
    return Optional.empty();  
}
```

Utilisation

```
userOpt.ifPresent(u ->  
    System.out.println(u.getName()));
```

```
String name = userOpt  
.map(User::getName)  
.orElse("Utilisateur inconnu");
```



Antipattern : Optional en paramètre

Pourquoi éviter

Complexité inutile, force l'appelant à créer un Optional même avec valeur concrète.

Mauvaise pratique

```
public void processValue(  
    Optional<String> optionalValue) {  
    if (optionalValue.isPresent()) {  
        // traitement  
    }  
}
```

Bonne pratique

```
public void processValue(String value) {  
    if (value != null) {  
        // traitement  
    }  
}  
  
public void processValue() {  
    // valeur par défaut  
}
```

Pattern : ifPresent et ifPresentOrElse

ifPresent(Consumer)

Action si valeur présente



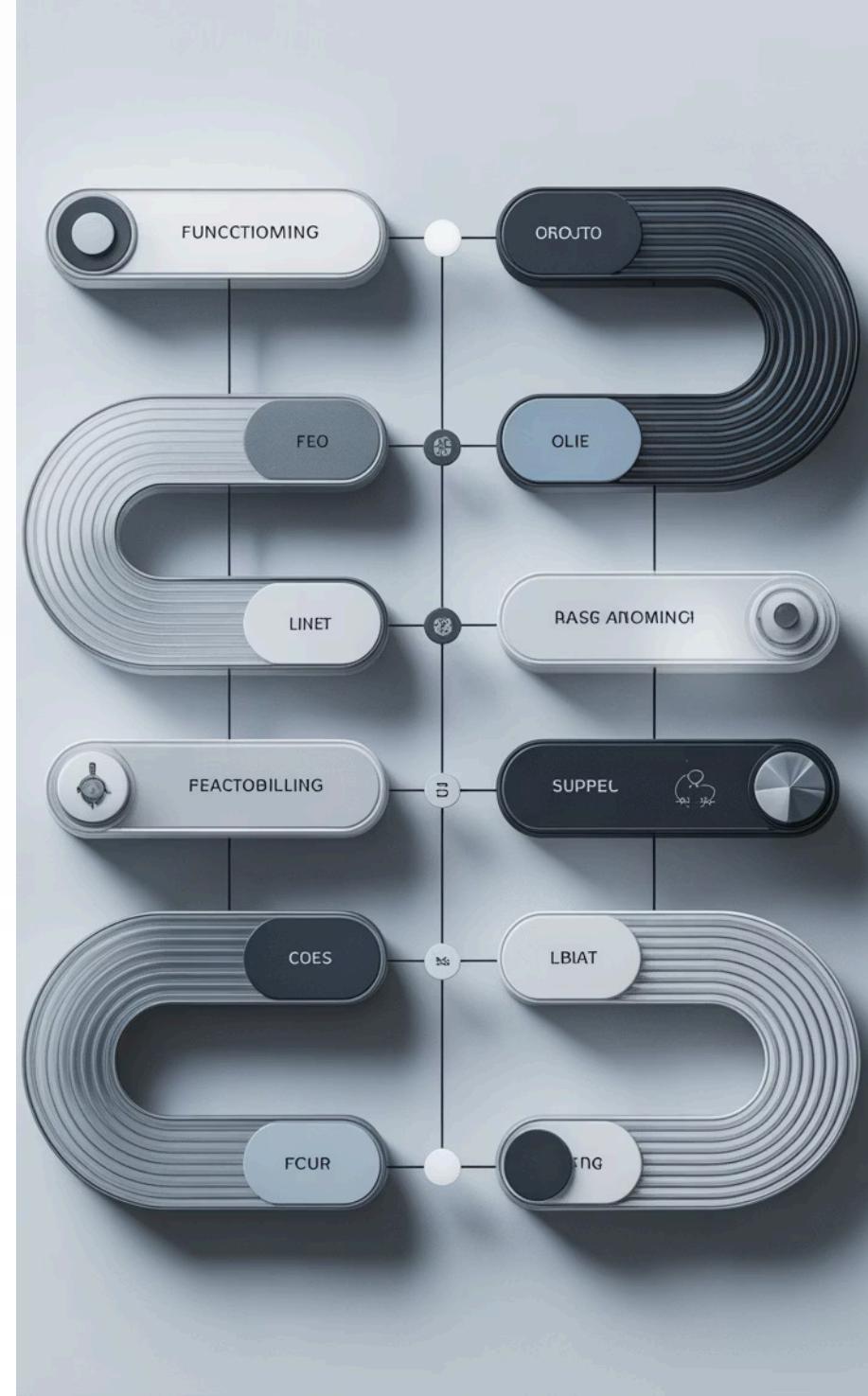
```
userOpt.ifPresent(user ->  
    sendWelcomeEmail(user));
```

ifPresentOrElse

Gère présence ET absence



```
userOpt.ifPresentOrElse(  
    user -> sendEmail(user),  
    () -> logNotFound()  
);
```



Pattern : orElse et orElseGet

orElse(T)

Valeur par défaut toujours évaluée

```
String name = nameOpt  
.orElse("Inconnu");
```

orElseGet(Supplier)

Supplier appelé uniquement si nécessaire

```
String name = nameOpt  
.orElseGet(() ->  
computeDefaultName());
```





Antipattern : orElse avec calculs coûteux

X Problème avec orElse()

```
String result = value.orElse(computeDefault());  
// computeDefault() exécuté MÊME SI value présente
```

Opération coûteuse toujours exécutée !

✓ Solution avec orElseGet()

```
String result = value.orElseGet(() -> computeDefault());  
// computeDefault() appelé SEULEMENT si nécessaire
```

Évaluation paresseuse optimise les performances

Pattern : map et flatMap

1

map(Function)

Transforme la valeur si présente

```
Optional<String> upperName =  
    nameOpt.map(String::toUpperCase);
```

2

flatMap(Function)

Pour fonctions retournant Optional

```
Optional<Address> address =  
    userOpt.flatMap(User::getAddress);
```





Pattern : Chaînage d'Optional

Pipeline élégant éliminant la pyramide de vérifications null

```
String cityName = userRepository.findById(userId)
    .flatMap(User::getAddress)
    .flatMap(Address::getCity)
    .map(City::getName)
    .orElse("Ville inconnue");
```



Antipattern : get() sans vérification

“

La méthode la plus dangereuse

get() lance NoSuchElementException si Optional vide. Utiliser sans vérification = ne pas utiliser Optional !

”

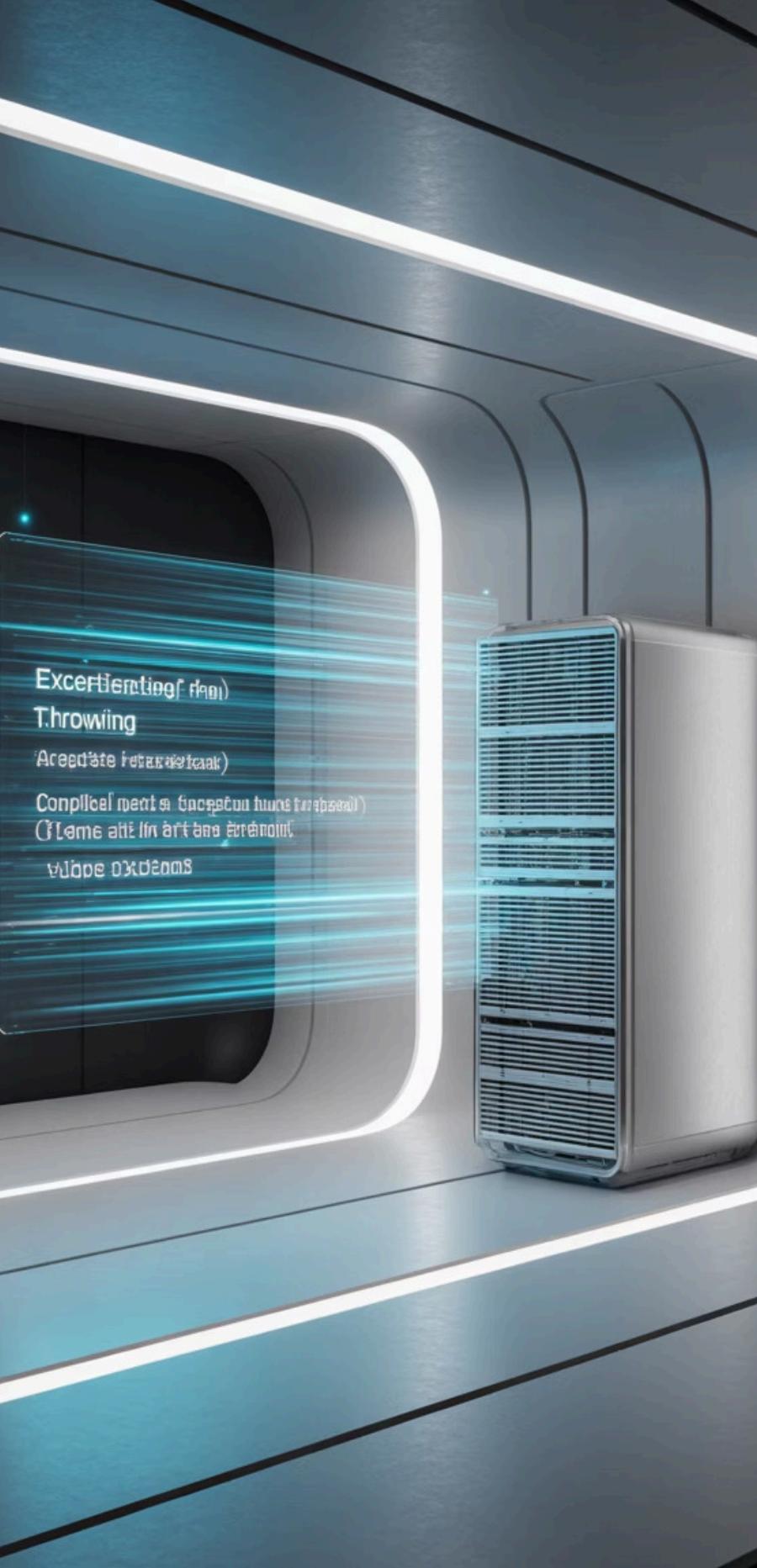
X Dangereux

```
String value = emptyOptional.get();
// NoSuchElementException !
```

✓ Alternatives sûres

```
// orElse
String safe = opt.orElse("défaut");

// orElseThrow
String safe = opt.orElseThrow(
    () -> new MyException());
```



Pattern : orElseThrow pour erreurs explicites

1

Java 10+ simple

```
User user =  
userOpt.orElseThrow();
```

2

Exception personnalisée

```
User user =  
userOpt.orElseThrow(  
() -> new  
UserNotFoundException(  
"User not found: " +  
userId));
```

3

Message contextuel

```
Order order = orderOpt.orElseThrow(  
() -> new IllegalStateException(  
"Order " + orderId +  
" should exist"));
```

Pattern : filter pour validation



Recherche utilisateur

Récupération Optional depuis base



Application filtre

Vérification utilisateur actif



Utilisation ou défaut

Traitement si actif, sinon gestion absence

```
Optional<User> activeUser = userRepository.findById(userId)
    .filter(User::isActive)
    .filter(user -> user.getAge() >= 18);
```

```
activeUser.ifPresentOrElse(
    user -> grantAccess(user),
    () -> logAccessDenied(userId));
```



Antipattern : Optional dans collections

✗ List<Optional<T>>

Structure complexe et difficile à manipuler

```
List<Optional<String>>  
optionalNames =  
    rawNames.stream()  
        .map(Optional::ofNullable)  
        .collect(Collectors.toList());  
  
// Complexé à traiter  
optionalNames.forEach(optName -> {  
    optName.ifPresent(name ->  
        System.out.println(name));  
});
```

✓ Filtrer les nulls en amont

Direct et sans Optional superflus

```
List<String> filteredNames =  
    rawNames.stream()  
        .filter(Objects::nonNull)  
        .collect(Collectors.toList());  
  
// Traitement direct  
filteredNames.forEach(name ->  
    System.out.println(name));
```

Pattern : Optional avec Streams



Filtrage et extraction

```
List<Address> addresses =  
users.stream()  
.map(User::getAddress)  
.flatMap(Optional::stream)  
.collect(Collectors.toList());
```



Recherche conditionnelle

```
Optional<User> firstActive =  
users.stream()  
.filter(User::isActive)  
.findFirst();
```



Transformation

```
List<String> emails =  
users.stream()  
.flatMap(u ->  
u.getEmail().stream())  
.collect(Collectors.toList());
```

Antipattern : Optional comme champ

Problèmes

- Sérialisation : Optional non Serializable
- Surcharge mémoire inutile
- Violation intention première d'Optional

✗ Antipattern

```
public class User {  
    private final String username;  
    private final Optional<String> email;  
  
    public Optional<String> getEmail() {  
        return email;  
    }  
}
```

✓ Meilleure pratique

```
public class User {  
    private final String username;  
    private final String email; // null OK  
  
    public Optional<String> getEmail() {  
        return Optional.ofNullable(email);  
    }  
}
```

Pattern : or() pour alternatives

Java 9+ : chaîner plusieurs sources avec évaluation paresseuse

1

Cache local

Tentative rapide

2

Base de données

Si absent du cache

3

API externe

Dernier recours

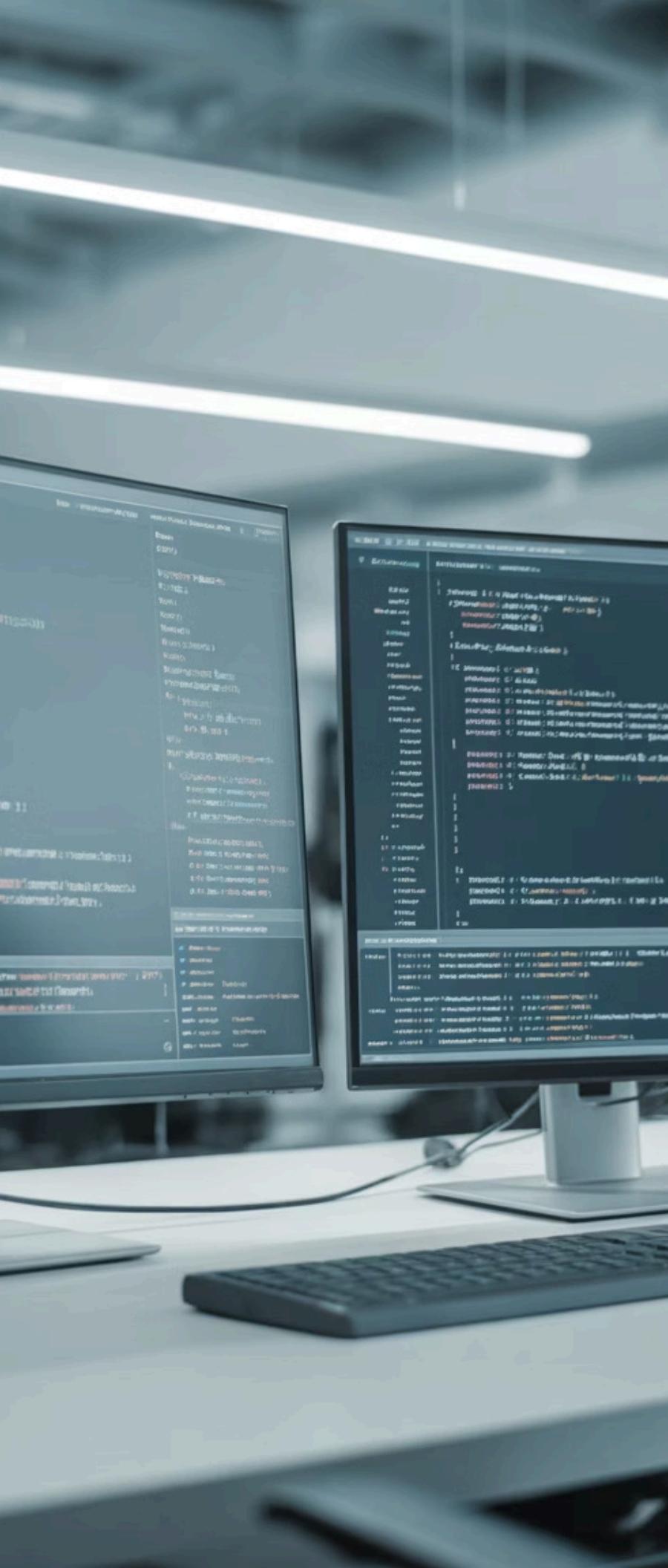
4

Valeur défaut

Si tout échoue

```
Optional<User> user = findInCache(userId)
.or(() -> findInDatabase(userId))
.or(() -> findInExternalAPI(userId))
.or(() -> Optional.of(createGuestUser()));
```

```
User finalUser = user.orElseThrow();
```



Comparaison : Avant et après

✗ Approche traditionnelle

```
public String  
getUserDisplayName(  
    Long userId) {  
    User user =  
        repo.findById(userId);  
    if (user == null) {  
        return "Utilisateur inconnu";  
    }  
    Profile profile =  
        user.getProfile();  
    if (profile == null) {  
        return user.getUsername();  
    }  
    String name =  
        profile.getDisplayName();  
    if (name == null ||  
        name.isEmpty()) {  
        return user.getUsername();  
    }  
    return name;  
}
```

Code verbeux, vérifications imbriquées

✓ Approche moderne

```
public String  
getUserDisplayName(  
    Long userId) {  
    return repo.findById(userId)  
        .flatMap(User::getProfile)  
        .map(Profile::getDisplayName)  
        .filter(name ->  
            !name.isEmpty())  
        .or(() ->  
            repo.findById(userId)  
                .map(User::getUsername))  
        .orElse("Utilisateur  
inconnu");  
}
```

Code concis, linéaire, expressif

Conclusion : Optional comme outil qualité

62%

Réduction NPE

Diminution moyenne en production

35%

Code en moins

Réduction code défensif

45%

Lisibilité

Amélioration score analyse statique

28%

Debugging

Temps réduit erreurs null

Optional force l'explicitation de l'absence, rend le code fonctionnel et réduit les NPE. **Adoption progressive** : changement culturel demandant pratique et discipline.

