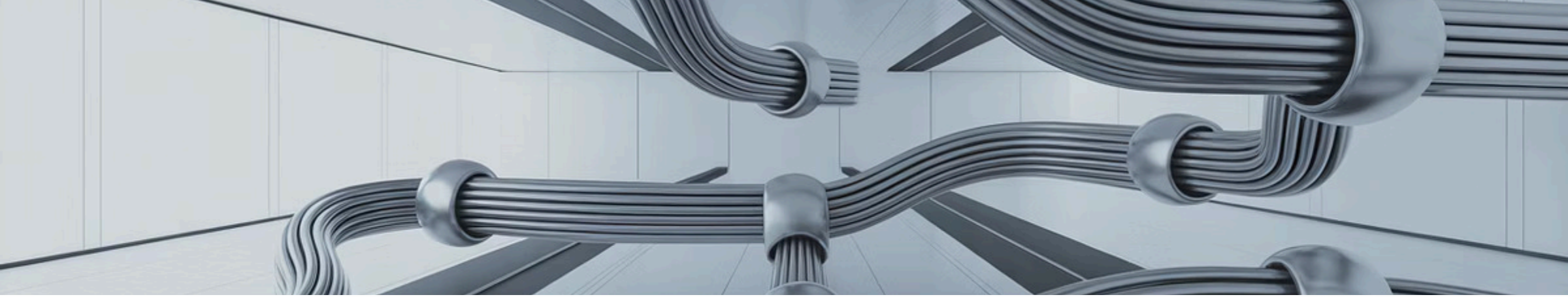
# Maîtriser CompletableFuture en Java



# Pourquoi CompletableFuture ?

### Composition fluide

Construire des pipelines asynchrones expressifs

### Gestion élégante

Enchaîner traitements et gérer erreurs simplement

### Contrôle précis

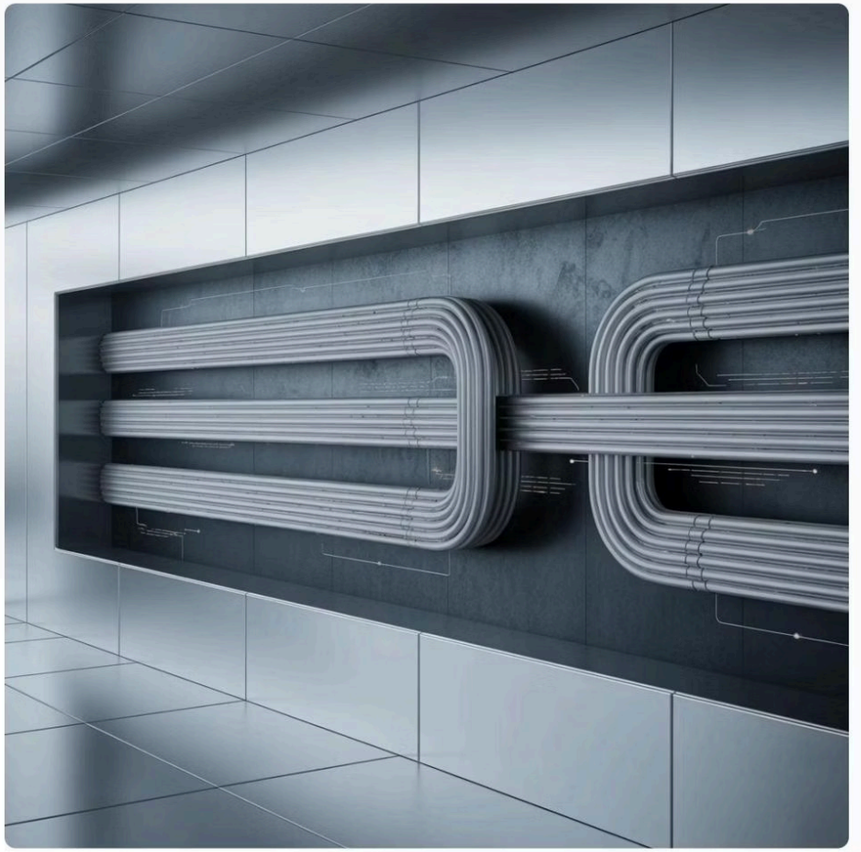
Maîtriser threads et exécution asynchrone

# Architecture et fondements

## Pattern Promise/Future

État interne évoluant de "non complété" vers "complété avec succès" ou "complété avec exception"

Séparation claire entre création et résolution



01

### Élimination du blocage

Plus besoin de get() explicite

02

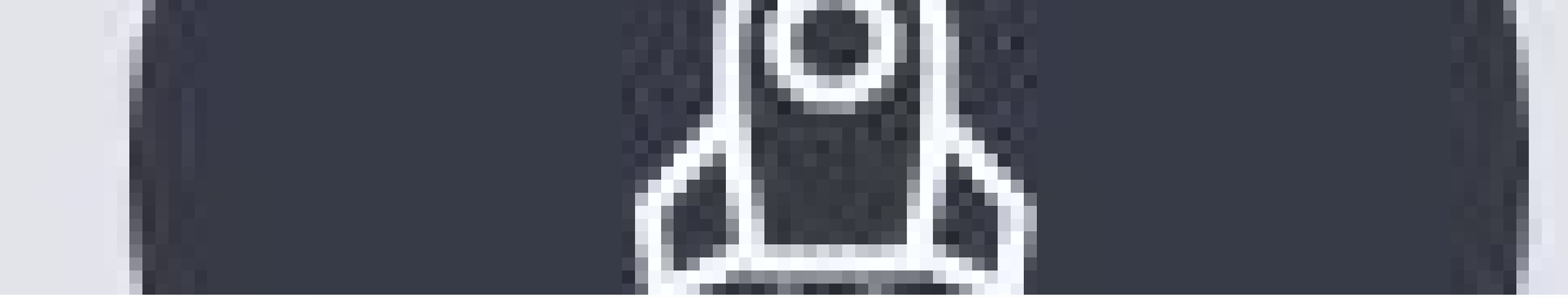
### Composition multiple

Enchaîner opérations asynchrones

03

### Gestion centralisée

Exceptions traitées élégamment



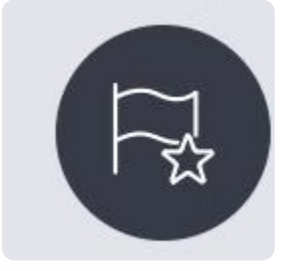
# Méthodes de création

### supplyAsync()

Exécute fonction retournant valeur dans thread séparé du ForkJoinPool

### runAsync()

Tâche asynchrone sans retour, idéal pour effets de bord



### completedFuture()

CompletableFuture déjà complété avec valeur donnée

# Transformation avec thenApply

Transformer résultat d'un CompletableFuture une fois complété

Similaire à map() des streams, mais asynchrone

```
CompletableFuture.supplyAsync(() -> "42")
  .thenApply(Integer::parseInt)
  .thenApply(n -> n * 2)
  .thenApply(n -> n + 10);
```

- **Composition fonctionnelle élégante**
- **Pas de blocage intermédiaire**
- **Type-safe à compilation**

Pour transformations asynchrones imbriquées, utilisez **thenCompose()** au lieu de thenApply()

# Effets de bord : thenAccept & thenRun

### thenAccept()

Reçoit résultat, effectue action avec valeur

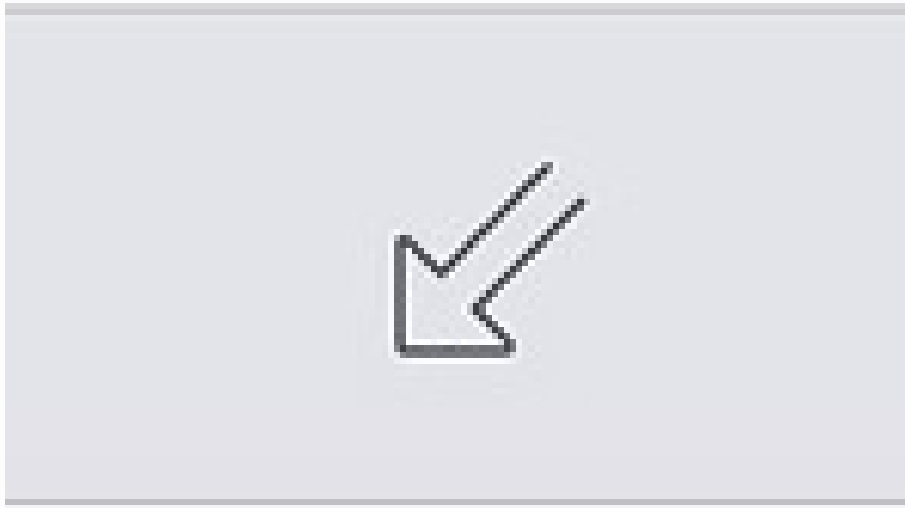
Logger, enregistrer, afficher

### thenRun()

N'accède pas au résultat

Nettoyage, notification, finalisation

```
CompletableFuture.supplyAsync(() -> fetchUserData())
 .thenAccept(user -> logger.info("Utilisateur: " + user))
 .thenRun(() -> logger.info("Terminé"));
```



cept

ne la valeur sans

at

# Combiner plusieurs CompletableFuture

### thenCombine()

Combine deux futures indépendants, exécution parallèle

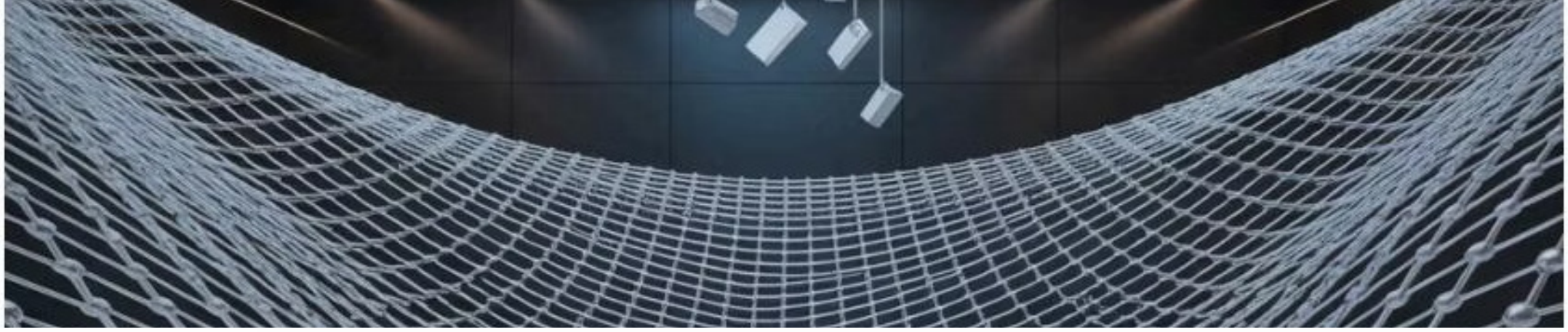
### allOf()

Attend complétion de tous les futures fournis

### anyOf()

Se complète dès le premier future terminé

```
userFuture.thenCombine(ordersFuture,
 (user, orders) -> new UserProfile(user, orders));
```



# Gestion robuste des erreurs

### exceptionally()

Valeur de fallback en cas d'exception

### handle()

Reçoit résultat ET exception, traite les deux cas

### whenComplete()

Action avec résultat et exception sans modifier valeur

Toujours terminer chaînes avec **exceptionally()** ou **handle()** pour garantir capture des exceptions

# Contrôle des threads d'exécution

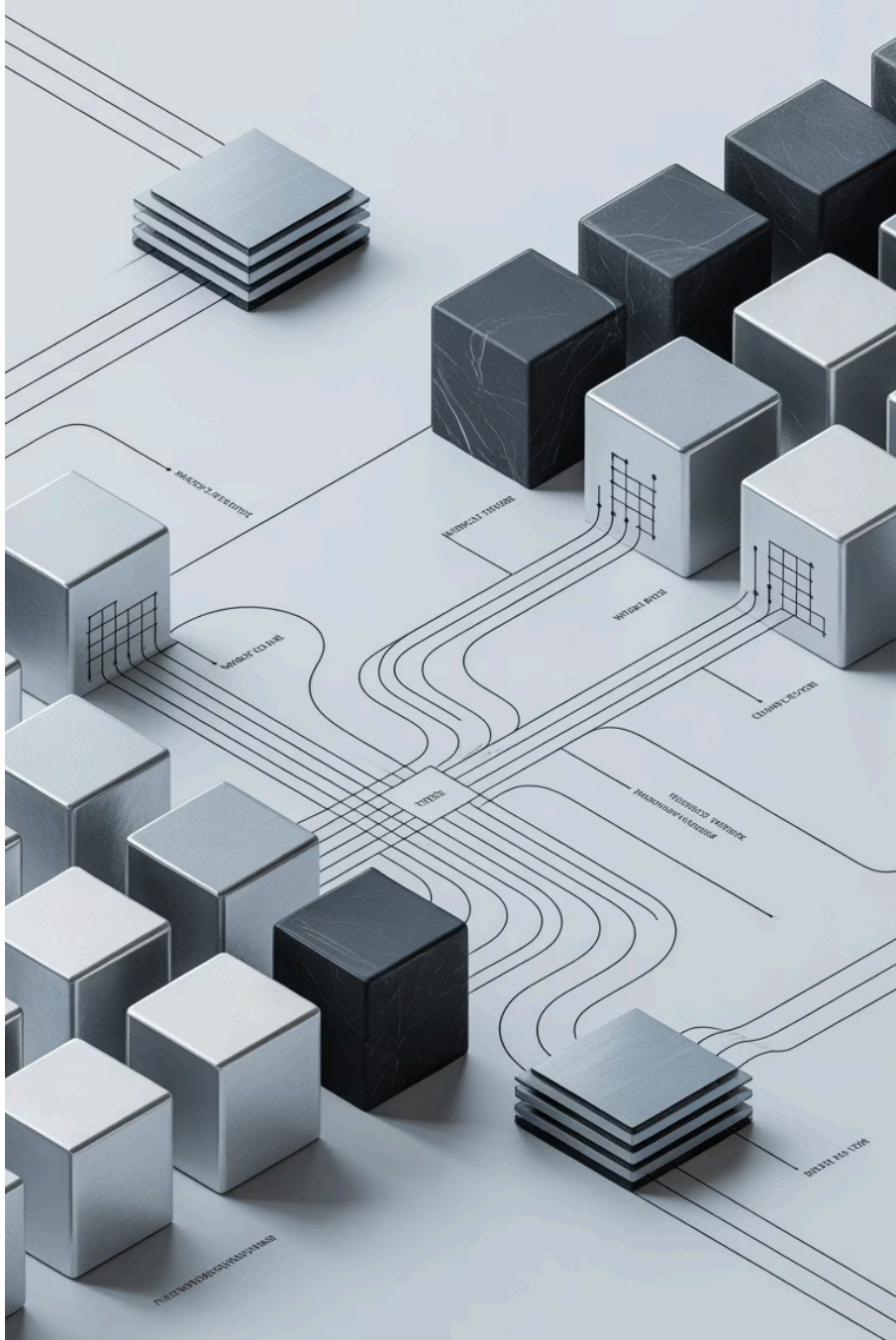
## Pool commun (défaut)

- Threads = processeurs - 1
- Partagé entre toutes opérations
- Simple, aucune configuration
- Risque de saturation

## ExecutorService personnalisé

- Contrôle taille pool et priorités
- Isolation des workloads
- Configuration adaptée
- Recommandé charges importantes

```
ExecutorService executor = Executors.newFixedThreadPool(10);
CompletableFuture.supplyAsync(() -> heavyComputation(), executor);
executor.shutdown();
```



# Timeout et fallback

1. **0s - Démarrage**

   Lancement opération asynchrone

2. **3s - Timeout**

   Si non complété, applique fallback

3. **Résultat**

   Valeur ou fallback retourné

### orTimeout()

```
future.orTimeout(5,
TimeUnit.SECONDS)
  .exceptionally(ex ->
"Timeout");
```

### completeOnTimeout()

```
future.completeOnTimeout(
 "Fallback", 3,
TimeUnit.SECONDS);
```

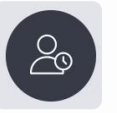
# Testing des CompletableFuture

### completedFuture()

Futures déjà complétés pour résultats instantanés sans exécution réelle

### failedFuture()

Java 9+ pour créer futures en échec et tester chemins d'erreur



### join() pour assertions

Bloquer et récupérer valeur dans tests, attente contrôlée

```
@Test
public void testAsyncOperation() {
 CompletableFuture future = service.fetchUserAsync(123);
 User result = future.join();
 assertEquals("John", result.getName());
}
```

# Performance et optimisations

**3x**

### Amélioration moyenne

Gain performance typique avec parallélisation correcte

**70%**

### Utilisation CPU

Meilleure exploitation ressources multi-cœurs

**200...**

### Latence réduite

Réduction moyenne temps de réponse

## Bonnes pratiques

- Éviter sur-parallélisation
- Dimensionner pools correctement
- Réutiliser ExecutorService
- Opérations bulk avec allOf()

## Anti-patterns

- Bloquer avec get()
- Futures pour opérations triviales
- Ne pas gérer exceptions
- Fuites de threads

# Intégration frameworks modernes

### Spring WebFlux

Approche réactive avec Reactor, interopère avec CompletableFuture

### @Async annotation

Méthodes retournent automatiquement CompletableFuture

### Repository asynchrone

Spring Data supporte requêtes non-bloquantes

```
@Service
public class UserService {
 @Async
 public CompletableFuture findUserAsync(Long id) {
 return CompletableFuture.completedFuture(user);
 }
}
```

# Évolution future

## Programmation Réactive

Project Reactor et RxJava pour flux asynchrones complexes

## Virtual Threads (Loom)

Java 21 révolutionne concurrence, code bloquant simple qui performe

## Coroutines Kotlin

Syntaxe élégante avec suspend functions, meilleure ergonomie

> "La maîtrise de CompletableFuture reste essentielle pour tout développeur Java moderne. Les principes fondamentaux de la programmation asynchrone demeurent constants."