Guide OptaPlanner - Concepts et Utilisation

Introduction

OptaPlanner est une librairie Java open-source spécialisée dans la résolution de problèmes d'optimisation combinatoire. Elle utilise des algorithmes métaheuristiques (recherche locale, tabou, recuit simulé, algorithmes génétiques) pour trouver la meilleure solution parmi un espace de solutions très vaste.

🎯 Concepts Fondamentaux

1. Les Objets à Planifier (Planning Entities)

Définition : Les objets que l'algorithme va organiser, positionner ou attribuer pour créer un planning optimal.

Caractéristiques :

- Ce sont les éléments **variables** du problème
- Leurs propriétés seront **ajustées automatiquement** par l'algorithme
- Ils possèdent des **variables de planification** (planning variables)

Exemple concret :

- **Emploi du temps** : Les `Cours` sont les objets à planifier
- **Affectation d'équipes** : Les `Employés` sont les objets à planifier
- **Livraisons** : Les `Commandes` sont les objets à planifier

2. Les Ressources (Resources)

Définition : Les éléments **fixes et limités** utilisés pour contraindre la planification des objets.

Caractéristiques :

- Quantité **restreinte** et **partagée**
- Définissent les **limites** du problème
- Ne changent pas pendant l'optimisation

Exemple concret pour un emploi du temps :

- **Professeurs** : nombre limité, ne peuvent enseigner qu'un cours à la fois

- **Salles** : capacité limitée, occupation exclusive
- **Créneaux horaires** : nombre fini de plages disponibles
- **Classes d'élèves** : ne peuvent suivre qu'un cours simultanément

3. Les Contraintes (Constraints)

Définition : Les règles métier que la solution doit respecter pour être considérée comme valide et optimale.

Types de contraintes :

- **Hard constraints** : **OBLIGATOIRES** (ex: un prof ne peut pas être à deux endroits)
- **Soft constraints** : **SOUHAITABLES** (ex: éviter les cours le vendredi après-midi)

🏗️ Architecture OptaPlanner

1. `@PlanningEntity` - L'Objet à Optimiser

Rôle : Classe représentant les objets que l'algorithme va manipuler.

Composants :

- **Planning Variables** : Propriétés que l'algorithme modifie (annotées `@PlanningVariable`)
- **Propriétés fixes** : Données immutables pendant l'optimisation

```
@PlanningEntity
public class Cours {
    private String matiere;           // Propriété fixe
    private int duree;                // Propriété fixe

    @PlanningVariable(valueRangeProviderRefs = "creneauxRange")
    private CreneauHoraire creneau;  // Variable de planification

    @PlanningVariable(valueRangeProviderRefs = "professeursRange")
    private Professeur professeur;   // Variable de planification

    @PlanningVariable(valueRangeProviderRefs = "sallesRange")
    private Salle salle;             // Variable de planification

    // Getters et setters...
}
```

2. `@PlanningSolution` - Le Problème Global

Rôle : Classe conteneur représentant l'état complet du problème à résoudre.

Composants :

- **Liste des entités** : Tous les objets à planifier
- **Listes des ressources** : Toutes les ressources disponibles
- **Score** : Évaluation de la qualité de la solution

```
@PlanningSolution
public class EmploiDuTemps {

    @ValueRangeProvider(id = "creneauxRange")
    private List<CreneauHoraire> creneaux;

    @ValueRangeProvider(id = "professeursRange")
    private List<Professeur> professeurs;

    @ValueRangeProvider(id = "sallesRange")
    private List<Salle> salles;

    @PlanningEntityCollectionProperty
    private List<Cours> cours;  // Les objets à planifier

    @PlanningScore
    private HardSoftScore score;  // Évaluation de la solution

    // Constructeurs, getters et setters...
}
```

3. `ConstraintProvider` - Les Règles Métier

Rôle : Définit toutes les contraintes que doit respecter la solution.

Fonctionnement : Utilise un `ConstraintFactory` pour créer des flux de données et appliquer des règles.

```
public class EmploiDuTempsConstraintProvider implements ConstraintProvider {

    @Override
    public Constraint[] defineConstraints(ConstraintFactory factory) {
        return new Constraint[] {
            // Contraintes obligatoires (HARD)
            unProfesseurParCreneau(factory),
            uneSalleParCreneau(factory),
```

```
            uneClasseParCreneau(factory),

            // Contraintes souhaitables (SOFT)
            eviterVendrediApresMidi(factory),
            equilibrerChargeProf(factory)
        };
    }

    // Implémentation des contraintes...
}
```

4. `SolverFactory` - Le Moteur d'Optimisation

Rôle : Configure et lance l'algorithme d'optimisation.

Configuration :

- Classe de solution
- Classe d'entité
- Fournisseur de contraintes
- Temps de calcul maximum
- Type d'algorithme

```
public class EmploiDuTempsSolver {

    public EmploiDuTemps resoudre(EmploiDuTemps problemeNonResolu) {
        SolverFactory<EmploiDuTemps> solverFactory = SolverFactory.create(
            new SolverConfigBuilder<EmploiDuTemps>()
                .withSolutionClass(EmploiDuTemps.class)
                .withEntityClasses(Cours.class)

.withConstraintProviderClass(EmploiDuTempsConstraintProvider.class)
                .withTerminationSpentLimit(Duration.ofMinutes(5))
                .buildConfig()
        );

        Solver<EmploiDuTemps> solver = solverFactory.buildSolver();
        return solver.solve(problemeNonResolu);
    }
}
```

🔧 Processus de Modélisation

Étape 1 : Identifier les Ressources

Créer les classes représentant les ressources fixes :

```
public class Professeur {
    private String nom;
    private List<String> specialites;
    private List<CreneauHoraire> disponibilites;
}

public class Salle {
    private String nom;
    private int capacite;
    private TypeSalle type; // AMPHITHEATRE, LABORATOIRE, etc.
}

public class CreneauHoraire {
    private DayOfWeek jour;
    private LocalTime heureDebut;
    private LocalTime heureFin;
}
```

Étape 2 : Définir l'Entité de Planification

```
@PlanningEntity
public class Cours {
    private String matiere;
    private Classe classe;
    private int duree;

    @PlanningVariable(valueRangeProviderRefs = "creneauxRange")
    private CreneauHoraire creneau;

    @PlanningVariable(valueRangeProviderRefs = "professeursRange")
    private Professeur professeur;

    @PlanningVariable(valueRangeProviderRefs = "sallesRange")
    private Salle salle;
}
```

Étape 3 : Créer la Solution

```java
@PlanningSolution
public class EmploiDuTemps {
    @ValueRangeProvider(id = "creneauxRange")
    private List<CreneauHoraire> creneaux;

    @ValueRangeProvider(id = "professeursRange")
    private List<Professeur> professeurs;

    @ValueRangeProvider(id = "sallesRange")
    private List<Salle> salles;

    @PlanningEntityCollectionProperty
    private List<Cours> cours;

    @PlanningScore
    private HardSoftScore score;
}
```

Étape 4 : Implémenter les Contraintes

Les contraintes fonctionnent en créant des **flux de données** à partir des entités de planification, puis en appliquant des **filtres**, **groupements** et **pénalités**.

📊 Constraint Factory - Fonctions Détaillées

`forEach()` - Point d'Entrée

Rôle : Crée un flux à partir de toutes les instances d'une classe.

```java
// Flux de tous les cours
factory.forEach(Cours.class)
```

`filter()` - Filtrage Conditionnel

Rôle : Ne garde que les éléments respectant une condition.

```java
// Seulement les cours du vendredi après-midi
factory.forEach(Cours.class)
    .filter(cours -> cours.getCreneau().getJour() == DayOfWeek.FRIDAY
                  &&
cours.getCreneau().getHeureDebut().isAfter(LocalTime.of(14, 0)))
```

`join()` - Combinaison de Flux

Rôle : Combine deux flux pour détecter des conflits ou relations.

```
// Détecter les cours qui se chevauchent pour un même professeur
factory.forEach(Cours.class)
    .join(Cours.class,
          Joiners.equal(Cours::getProfesseur),  // Même professeur
          Joiners.equal(Cours::getCreneau),     // Même créneau
          Joiners.lessThan(Cours::getId))       // Éviter les doublons
```

`groupBy()` - Regroupement et Agrégation

Rôle : Groupe les éléments et calcule des statistiques.

```
// Compter les cours par professeur
factory.forEach(Cours.class)
    .groupBy(Cours::getProfesseur, ConstraintCollectors.count())
```

`penalize()` / `reward()` - Attribution de Score

Rôle : Attribue des pénalités (score négatif) ou récompenses (score positif).

```
// Pénalité pour les conflits (HARD constraint)
.penalize("Conflit professeur", HardSoftScore.ONE_HARD)

// Récompense pour l'équilibrage (SOFT constraint)
.reward("Équilibrage charge", HardSoftScore.ONE_SOFT)
```

`penalizeBy()` / `rewardBy()` - Score Pondéré

Rôle : Applique un score proportionnel à une valeur.

```
// Pénalité proportionnelle au nombre de cours en excès
.penalizeBy("Surcharge professeur", HardSoftScore.ONE_SOFT,
            (prof, nbCours) -> Math.max(0, nbCours - 20)) // Max 20h/semaine
```

💡 Exemples de Contraintes Complètes

Contrainte Hard : Un Professeur par Créneau

```
private Constraint unProfesseurParCreneau(ConstraintFactory factory) {
    return factory.forEach(Cours.class)
        .join(Cours.class,
```

```
                Joiners.equal(Cours::getProfesseur),    // Même prof
                Joiners.equal(Cours::getCreneau),       // Même créneau
                Joiners.lessThan(Cours::getId))         // Éviter doublons
        .penalize("Un professeur par créneau", HardSoftScore.ONE_HARD);
}
```

Contrainte Hard : Capacité des Salles

```
private Constraint respecterCapaciteSalle(ConstraintFactory factory) {
    return factory.forEach(Cours.class)
        .filter(cours -> cours.getClasse().getNbEleves() >
cours.getSalle().getCapacite())
        .penalize("Capacité salle dépassée", HardSoftScore.ONE_HARD);
}
```

Contrainte Soft : Éviter Vendredi Après-midi

```
private Constraint eviterVendrediApresMidi(ConstraintFactory factory) {
    return factory.forEach(Cours.class)
        .filter(cours -> cours.getCreneau().getJour() == DayOfWeek.FRIDAY
                        &&
cours.getCreneau().getHeureDebut().isAfter(LocalTime.of(14, 0)))
        .penalize("Éviter vendredi après-midi", HardSoftScore.ONE_SOFT);
}
```

Contrainte Soft : Équilibrer la Charge des Professeurs

```
private Constraint equilibrerChargeProf(ConstraintFactory factory) {
    return factory.forEach(Cours.class)
        .groupBy(Cours::getProfesseur, ConstraintCollectors.count())
        .filter((prof, nbCours) -> nbCours > 20) // Plus de 20h/semaine
        .penalizeBy("Surcharge professeur", HardSoftScore.ONE_SOFT,
                    (prof, nbCours) -> nbCours - 20); // Pénalité = heures en
excès
}
```

🚀 Utilisation Complète

```
public class Main {
    public static void main(String[] args) {
        // 1. Créer le problème non résolu
        EmploiDuTemps probleme = creerProbleme();
```

```
        // 2. Configurer le solver
        SolverFactory<EmploiDuTemps> solverFactory = SolverFactory.create(
            new SolverConfigBuilder<EmploiDuTemps>()
                .withSolutionClass(EmploiDuTemps.class)
                .withEntityClasses(Cours.class)

.withConstraintProviderClass(EmploiDuTempsConstraintProvider.class)
                .withTerminationSpentLimit(Duration.ofMinutes(5))
                .buildConfig()
        );

        // 3. Résoudre
        Solver<EmploiDuTemps> solver = solverFactory.buildSolver();
        EmploiDuTemps solutionOptimale = solver.solve(probleme);

        // 4. Afficher le résultat
        System.out.println("Score final: " + solutionOptimale.getScore());
        afficherEmploiDuTemps(solutionOptimale);
    }

    private static EmploiDuTemps creerProbleme() {
        // Initialiser les ressources et les cours non planifiés
        // ...
    }
}
```

🎯 Points Clés à Retenir

1. **Planning Entity** = Ce qu'on optimise (les objets avec des variables)
2. **Planning Solution** = Le problème complet (entités + ressources + score)
3. **Constraints** = Les règles métier (hard = obligatoire, soft = souhaitable)
4. **ConstraintFactory** = Outil pour créer des flux et appliquer des règles
5. **Solver** = Le moteur qui trouve la solution optimale

OptaPlanner automatise la recherche de la meilleure combinaison en explorant intelligemment l'espace des solutions possibles selon vos contraintes métier.