UNIVERSITE BOURGOGNE EUROPE

Licence 1 Informatique-Electronique

Année universitaire 2024-2025

Présenté par : **DIALLO Aissatou Bobo**

Rapport de projet

Jeu – Des Personnages

Date de remise :07 /05/2025

Chargé du cours : Mr SAVELLI Joel

Table des matières

I. Introduction

Le projet présenté dans ce rapport s'inscrit dans un enseignement de programmation orientée objet en Java. Il consiste à concevoir un jeu de plateau interactif dans lequel un joueur, contrôlé par l'utilisateur, doit évoluer dans un environnement peuplé d'adversaires et de réservoirs d'énergie. Le joueur doit ainsi faire preuve de stratégie pour se déplacer, combattre, gérer ses ressources énergétiques, et tenter de survivre le plus longtemps possible tout en accumulant de l'énergie.

Ce jeu, intégralement textuel, repose sur une modélisation rigoureuse des entités du jeu (joueur, adversaires, salles, bidons, etc.) à l'aide des principes de la POO : encapsulation, héritage, polymorphisme, délégation. L'architecture logicielle adoptée met en avant une séparation claire des responsabilités et une interaction cohérente entre les classes.

Ce rapport a pour objectif de détailler les différentes étapes de la conception, les fonctionnalités effectivement développées, les choix de modélisation adoptés ainsi que les résultats observés à travers plusieurs jeux d'essais commentés. Il s'appuie également sur un diagramme de classes illustrant la structure globale du programme.

II. Description des fonctionnalités de la réalisation

Le projet consiste en la réalisation d'un jeu en Java dans lequel des personnages évolue sur un plateau rectangulaire composé de différentes salles. Dans ces personnages on a un joueur et différents types d'adversaires. L'objectif du joueur est de **neutraliser les adversaires présents** sur le plateau tout en **gérant son énergie** grâce à des bidons d'énergie et un collecteur.

Fonctionnalités principales :

- **Création du plateau de jeu** :
  Le plateau est composé de salles contenant des bidons (SalleBidon), de salles internes (SalleDedans) et de bords (Bordure) empêchant les personnages de sortir du cadre du jeu. Les salles de bord agissent comme des murs et bloquent tout déplacement vers l'extérieur et aucun déplacement vers Bordure n'est autorisé.

- **Déplacement du joueur** :
  À chaque tour, le joueur peut **effectuer deux déplacements** dans une direction choisie parmi les huit directions (haut, bas, gauche, droite, et diagonales).
  Il peut interagir avec la salle : récupérer de l'énergie depuis un bidon ou **stocker de l'énergie** dans un collecteur.
  Mais il peut y arriver des cas ou le joueur ne peut pas effectuer de déplacement dans la direction voulue (parce que c'est vers les bords, ou il sort du plateau etc..) dans ce cas il passe son tour, dans le cas contraire il perd une unité d'énergie.

- **Gestion de l'énergie** :
  Le joueur possède une énergie limitée(comme pour les adversaires), c'est-à-dire que y a un maximum de quantité d'énergie fixé qu'il ne peut pas dépasser.
  Il peut :
  - prendre de l'énergie dans un bidon qu'il trouve dans une salle au cours de ses déplacement,
  - recharger son énergie actuelle,
  - ou stocker l'énergie excédentaire dans le collecteur(les adversaires n'ont pas accès au collecteur).

- **Adversaires de différents types** :
  Le joueur affronte *un seul type d'adversaire par partie*, à choisir au lancement du jeu :
  - **Adversaires Déterminés** : ils fuient ou poursuivent le joueur de manière systématique.
  - **Adversaires Velléitaires** : ils ont un comportement aléatoire biaisé (souvent rationnel, mais parfois imprévisible).
  - **Adversaires Intelligents** : ils tentent de choisir les meilleures directions en contournant les obstacles.

- **Combat** :
  Lorsqu'un joueur ou un adversaire entre dans une salle déjà occupée par un adversaire ou le joueur, un **combat automatique** a lieu. Le personnage le plus fort remporte le combat et peut éventuellement **absorber de l'énergie** de l'autre, selon ce qu'il a déjà en réserve, ou si il veut mettre dans le collecteur(dans le cas du joueur).

- **Victoire et défaite** :
  - lorsque **tous les adversaires sont neutralisés** (le joueur peut alors décider d'arrêter),
  - ou lorsque le joueur **n'a plus d'énergie** (défaite).

- **Affichage visuel du plateau** à chaque tour :
  Le plateau est imprimé en console à chaque étape avec les symboles :
  - X : bord du plateau
  - . : salle vide
  - 0 : bidon
  - J : joueur
  - A : adversaire (pour tout type)

- **Interactions utilisateur claires** :
  Le joueur est guidé par des messages pour choisir les directions et les actions possibles. S'il des informations invalides il a des messages qui lui redemande les bonnes informations.

Autres fonctionnalités :

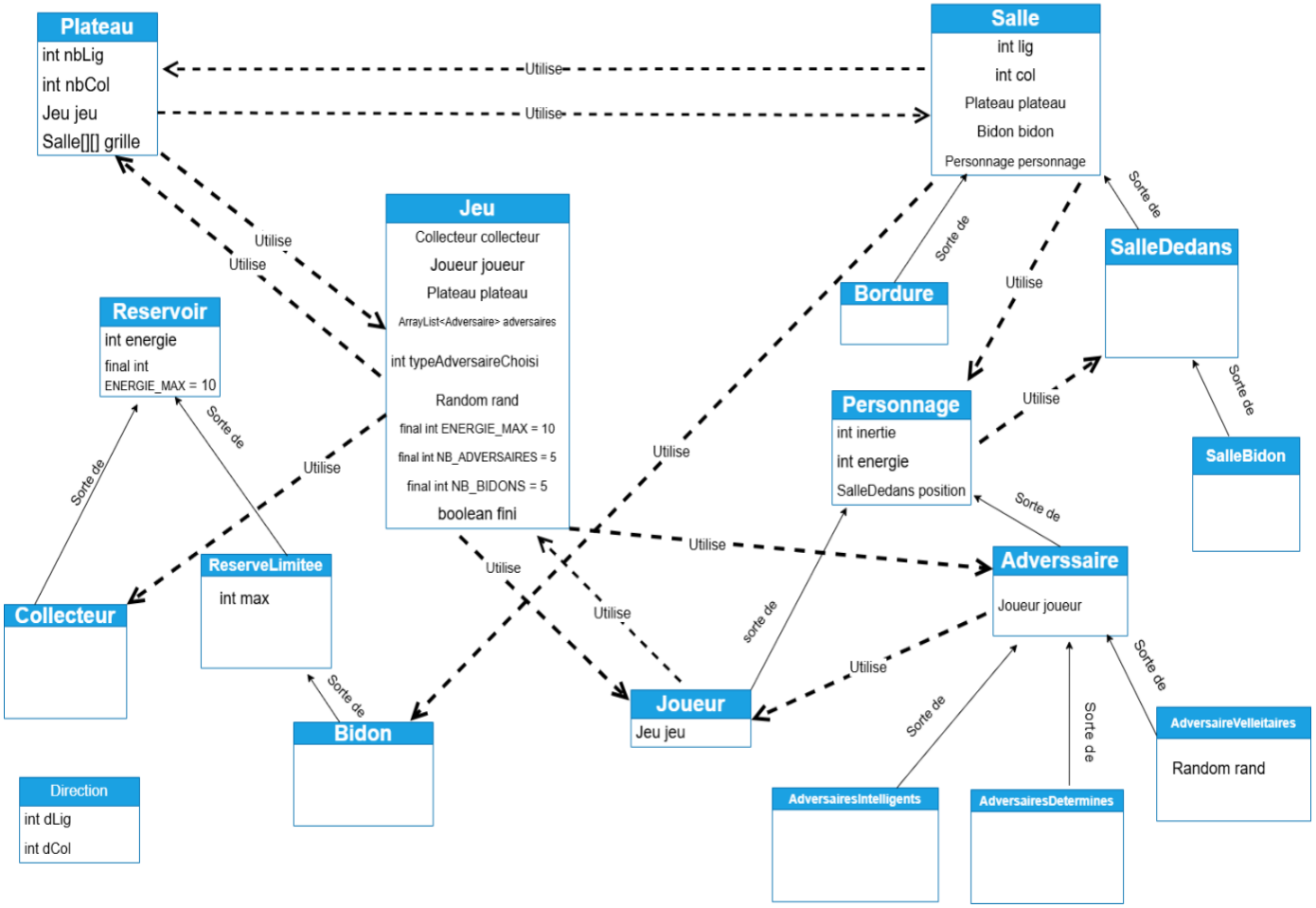
- **Choix du nom du joueur** :
  Au début du jeu, le joueur l'utilisateur a la chance d'entrer don nom au clavier.

- **Choix du type d'adversaire au démarrage :**
  Après le choix du nom, le joueur a aussi la possibilité de choisir le type d'adversaire qu'il souhaite affronter, ce qui rend le jeu plus modulable.

- **Affichage des événements** :
  Les actions importantes (déplacements, fuites, poursuites, blocages, combats, etc.) sont affichées en temps réel avec des messages explicatifs.

III. Diagramme de classes

Le diagramme de classes présenté ci-dessous décrit l'architecture principale de notre application. Il met en évidence les différentes entités du jeu ainsi que leurs relations :



- **Héritage** : 

  - La classe **Personnage** est une classe mère dont héritent les classes **Joueur** et **Adversaire**.
  - **Adversaire** est une classe abstraite à partir de laquelle sont dérivées trois sous-classes concrètes : **AdversaireVelleitaires**, **AdversairesDetermines** et **AdversairesIntelligents**.
  - **Salle** est une classe, dont dérivent les classes **SalleDedans**, **SalleBidon** et **Bordure.**
  - **Reservoir** est la classe dont dérivent les classe **Collecteur** et **RserveLimitee** qui, aussi est une classe dont dérive la classe **Bidon**.

- **Relations d'utilisation** : 

    - **Jeu** utilise fortement **Plateau**, **Collecteur**, **Joueur** et la liste des **Adversaires** pour orchestrer la partie.
    - **Plateau** utilise le **Jeu** et **Salle**
    - **Salle** utilise **Plateau**, **Personnage** et **Bidon**
    - **Personnage** utilise **SalleDedans**
    - **Adversaire** utilise **Joueur**
    - **Joueur** utilise **Jeu**

    - Les **personnages** utilisent des objets de type `SalleDedans` pour connaître leur position et se déplacer.
    - `SalleDedans` peut contenir un `Personnage` et un `Bidon`.
    - Les classes `Bidon` et `Collecteur` héritent toutes deux de `Reservoir` ou `ReserveLimitee`, pour mutualiser la gestion d'énergie.
    - La classe `AdversaireVelléitaires` introduit une dépendance à `Random` pour les déplacements aléatoires.

IV. Descriptif des classes

❖ Jeu

- **Rôle** : C'est la classe centrale qui orchestre la logique du jeu (initialisation, tours de jeu, interactions, conditions de victoire/défaite).
- **Attributs principaux** :

```
private Plateau plateau;
private Joueur joueur;
private int typeAdversaireChoisi;
private ArrayList<Adversaire> adversaires;
private Collecteur collecteur;
private boolean fini = false;
```

Plateau plateau : la grille de jeu avec les salles.

Joueur joueur : le joueur principal contrôlé par l'utilisateur.

int typeAdversaireChoisi : type d'adversaire utilisé (déterminé, velléitaire ou intelligent).

ArrayList<Adversaire> adversaires : liste des ennemis.

Collecteur collecteur : stocke l'énergie extraite.

boolean fini : pour déterminer la fin du jeu.

- **Méthodes** :
  Constructeur :

```
public Jeu(int nbLig, int nbCol) {
    this.plateau = new Plateau(nbLig, nbCol, this);
    this.adversaires = new ArrayList<>();
    this.collecteur = new Collecteur();
    this.fini = false;

    String nomJoueur = Lire.S("Entrez votre nom");

    // Choix du type d'adversaire
    System.out.println("Choisissez le type d'adversaires à affronter");
    System.out.println("1 - Adversaires Déterminés (ils vous poursuivent/fuient sans réfléchir)");
    System.out.println("2 - Adversaires Velléitaires (se déplacent avec un peu de hasard)");
    System.out.println("3 - Adversaires Intelligents (analysent les alentours)");

    int choixType = Lire.i("Votre choix : ");
    while (choixType < 1 || choixType > 3) {
        choixType = Lire.i("Veuillez entrer 1, 2 ou 3");
    }

    this.typeAdversaireChoisi = choixType;

    initJeu();
    joue();
}
```

Ce constructeur initialise :

le **plateau** de jeu avec les dimensions données (nbLig, nbCol) ;

une **liste vide** pour stocker les adversaires ;

un **collecteur d'énergie** (réserve spéciale) ;

le booléen fini est mis à false pour indiquer que le jeu n'est pas terminé.

String nomJoueur = Lire.S("Entrez votre nom");

L'utilisateur saisit son nom.

Affiche un menu décrivant les **3 types de comportements** possibles des adversaires :

poursuivent ou fuient sans réfléchir, déplacés aléatoirement, analysent la position du joueur pour décider.

int choixType = Lire.i("Votre choix : ");

while (choixType < 1 || choixType > 3) {

choixType = Lire.i("Veuillez entrer 1, 2 ou 3");

}

this.typeAdversaireChoisi = choixType;

L'utilisateur entre un chiffre correspondant à un type d'adversaire. La **boucle while vérifie** que l'entrée est correcte (entre 1 et 3), Puis le choix est enregistré dans typeAdversaireChoisi.

Puis appelle, **initJeu()** : initialise les salles, place le joueur, les bidons et les adversaires sur le plateau. **joue()** : démarre la **boucle de jeu** où joueur et adversaires jouent à tour de rôle.

- **initJeu()** : installe le joueur, les bidons et les adversaires sur le plateau.

```
public void initJeu() {
    int ligCentre = plateau.getNbLig() / 2 + 1;
    int colCentre = plateau.getNbCol() / 2 + 1;
    SalleDedans salleCentre = (SalleDedans) plateau.getSalle(ligCentre, colCentre);

    // Création du joueur
    joueur = new Joueur(plateau, ENERGIE_MAX, this);
    joueur.setPosition(salleCentre);
    salleCentre.setPersonnage(joueur);
```

Dans ce fragment de code, avec les deux premières lignes du corps de la méthode, on récupère le centre du plateau (avec la ligne et la colonne).

Et avec cette ligne ***SalleDedans salleCentre = (SalleDedans) plateau.getSalle(ligCentre, colCentre);*** on récupère l'objet Salle se trouvant au centre du plateau en faisant un cast de SalleDedans sur lui.

Par consequent, *salleCentre* représente donc la salle vide au centre du plateau**,** prête à accueillir le joueur.

Puis on crée un en lui donnant tout ce qu'il a besoin c'est-à-dire le *Plateau* ou il va jouer, un *maximum énergétique* et la classe courante *Jeu* elle-même d'où le (this).

Ensuite, setPosition(salleCentre) : le joueur **mémorise la salle** dans laquelle il se trouve.

salleCentre.setPersonnage(joueur) : la salle **mémorise le joueur** qui y est actuellement.

```
ArrayList<SalleDedans> sallesDispo = new ArrayList<>();
for (int i = 1; i <= plateau.getNbLig(); i++) {
    for (int j = 1; j <= plateau.getNbCol(); j++) {
        Salle s = plateau.getSalle(i, j);
        if (s instanceof SalleDedans sd && s != salleCentre) {
            sallesDispo.add(sd);
        }
    }
}

// Mélange aléatoire de toutes les positions disponibles
Collections.shuffle(sallesDispo);
```

On crée une **liste vide** pour stocker toutes les salles de type ***SalleDedans*** (les salles internes jouables), pour stocker les salles disponibles, ***ArrayList<SalleDedans> sallesDispo = new ArrayList<>();***

Ensuite, on parcourt les lignes et les colonnes du plateau(on commence à 1 parce que 0 et nbLig +1 sont les bords), en récupérant chaque salle à chaque indice i, j des boucles, on vérifie si cette salle est une instance de SalleDedans et que ce n'est pas la salle au centre c'est-à-dire où se trouve le joueur, si cette condition est validée on cast cette salle en SalleDedans avec ***instanceof SalleDedans sd*** ou avec ***SalleDedans sd = (SalleDedans) s***, après on ajoute cette salle dans la liste de salles disponibles qu'on a créé.

Collections.shuffle(sallesDispo); On mélange aléatoirement les éléments de la liste.

```
// Placement des bidons
int bidonsPlaces = 0;
int index = 0;
while (bidonsPlaces < NB_BIDONS && index < sallesDispo.size()) {
    SalleDedans sd = sallesDispo.get(index);
    int lig = sd.getLig();
    int col = sd.getCol();
    SalleBidon sb = new SalleBidon(lig, col, plateau);
    plateau.setSalle(sb, lig, col); // remplace la salle
    bidonsPlaces++;
    index++;
}
```

On initialise deux compteurs à 0, un pour le nombre de bidons déjà placé et l'autre pour la position actuelle dans la liste déjà mélangée.

Après on a une boucle while qui tourne tant que Le **nombre de bidons placés** est inférieur à la constante NB_BIDONS (souvent 5), et qu'on n'a pas **épuisé la liste des salles disponibles** (index < sallesDispo.size()).

SalleDedans sd = sallesDispo.get(index);

int lig = sd.getLig();

int col = sd.getCol();

On récupère la salle à la position index dans la liste mélangée. Puis on extrait ses coordonnées (lig, col).

On crée une nouvelle instance de type `SalleBidon`, une salle contenant un bidon **plein**. On met dans la grille du plateau cette nouvelle `SalleBidon` à l'emplacement `(lig, col)`.

On compte un bidon de plus placé. On passe à l'index suivant dans sallesDispo.

```java
int advPlaces = 0;
while (advPlaces < NB_ADVERSAIRES && index < sallesDispo.size()) {
    Salle s = plateau.getSalle(sallesDispo.get(index).getLig(), sallesDispo.get(index).getCol());
    if (s instanceof SalleDedans sd && !(s instanceof SalleBidon)) {

        Adversaire a;
        int inertie = 2 + rand.nextInt(8); // aléatoire entre 2 et 9
        int energie = 10;                  // énergie fixée à 10

        switch (typeAdversaireChoisi) {
            case 1 ->
                a = new AdversairesDetermines(inertie, energie, joueur);          // Déterminés
            case 2 ->
                a = new AdversaireVelleitaires(inertie, energie, joueur);         // Velléitaires
            case 3 ->
                a = new AdversairesIntelligents(inertie, energie, joueur);        // Intelligents
            default -> {
                System.out.println("Type inconnu, on choisit des Déterminés par défaut.");
                a = new AdversairesDetermines(inertie, energie, joueur);
            }
        }

        a.setPosition(sd);
        sd.setPersonnage(a);
        adversaires.add(a);
        advPlaces++;
    }
    index++;
```

De même pour les adversaires on initialise à 0 le nombre d'adversaires à placer, pour l'index on continue là où il s'est arrêté pour les bidons.

La boucle tourne tant qu'on n'a pas placé le nombre requis d'adversaires, et qu'on n'a pas parcouru toute la liste sallesDispo

On récupère une salle à partir de l'index courant de la liste mélangée.

if (s instanceof SalleDedans sd && !(s instanceof SalleBidon))

On vérifie que c'est bien une SalleDedans (jouable), et que ce **n'est pas une SalleBidon** (interdiction de placer un adversaire sur une case contenant un bidon dès le début du jeu).

L'**inertie** est générée aléatoirement entre 2 et 9, comme demandé dans le sujet (pas de 0 ni 1 pour éviter un adversaire trop faible au début).

L'**énergie** est toujours initialisée à 10 (valeur max)

Selon ce que l'utilisateur a choisi en début de partie (1, 2 ou 3), on crée un adversaire :

Déterminé, Velléitaire, Intelligent.

En cas d'erreur, on retombe par défaut sur un adversaire déterminé (choix de sécurité).

a.setPosition(sd); // Donne sa position à l'adversaire

sd.setPersonnage(a); // Place le personnage dans la salle

adversaires.add(a); // Ajoute dans la liste des adversaires

advPlaces++; // Incrémente le compteur

index++; Puis On passe à l'index suivant dans **sallesDispo** pour tester une nouvelle salle si besoin.

- Joue() : boucle principale du jeu, gère les déplacements et interactions.

```
public void joue() {
    while (!fini && !joueur.estNeutralise()) {
        System.out.println(plateau);
        System.out.println("Énergie du joueur : " + joueur.getEnergie());
        System.out.println("Collecteur : " + collecteur.getContenu());
        System.out.println("Adversaires restants : " + adversaires.size());
        System.out.println();

        // Le joueur joue 2 fois
        joueur.avance();
        if (!joueur.estNeutralise()) {
            joueur.avance();
        }
```

On a une boucle qui tourne tant que le jeu n'est pas fini et que le joueur n'est pas neutralisé.

Affichage de l'état actuel du jeu :

- Le plateau est affiché via System.out.println(plateau), en appelant la méthode toString() du plateau qui concatène les toString() des salles.
- L'énergie restante du joueur, le contenu du collecteur et le nombre d'adversaires encore en jeu sont affichés.

Le joueur joue deux fois :

- Il appelle joueur.avance(), qui demande une direction et tente de se déplacer.
- Si après le premier déplacement le joueur **n'est pas neutralisé**, il rejoue une deuxième fois immédiatement.

Les adversaires jouent à leur tour :

```
// Les adversaires jouent
ArrayList<Adversaire> aSupprimer = new ArrayList<>();
// for (Adversaire a : adversaires) {
for (int i = 0; i < adversaires.size(); i++) {
    Adversaire a = adversaires.get(i);
    if (!a.estNeutralise()) {
        a.avance();
    } else {
        if (a.getPosition() != null) {
            a.getPosition().setPersonnage(null);
        }
        aSupprimer.add(a);
    }
}
adversaires.removeAll(aSupprimer);
```

- Une **liste temporaire** aSupprimer est utilisée pour **marquer les adversaires à retirer**.
- Pour chaque adversaire de la liste adversaires :
    - S'il est neutralisé (énergie nulle), on libère sa salle et on l'ajoute à aSupprimer.
    - Sinon, il appelle sa propre méthode avance() — dont le comportement dépend de sa sous-classe :
        - **Déterminés** : poursuivent/fuient directement le joueur.
        - **Velléitaires** : choisissent une direction avec du hasard.
        - **Intelligents** : testent les chemins autour d'eux pour décider au mieux.
- Après la boucle, tous les adversaires marqués dans aSupprimer sont supprimés de la liste principale avec adversaires.removeAll(...).

Fin de partie possible :

```
// Vérifie victoire
if (adversaires.isEmpty()) {
    System.out.println("Tous les adversaires sont neutralisés !");
    int choix = Lire.i("Voulez-vous terminer la partie ? (1=oui, 0=non) ");
    if (choix == 1) {
        fini = true;
        System.out.println("Score final (énergie dans collecteur) : " + collecteur.getContenu());
    }
```

- Si **tous les adversaires sont éliminés** (adversaires.isEmpty()), on demande au joueur s'il veut finir.
- Si oui (1), on met fini = true et on affiche le **score final** (énergie dans le collecteur).

Fin automatique si le joueur est neutralisé :

```
if (joueur.estNeutralise()) {
    System.out.println("Le joueur a été neutralisé. GAME OVER.");
}
```

- En dehors de la boucle, on teste à nouveau joueur.estNeutralise() pour afficher "GAME OVER" si c'est le cas.

❖ Plateau

- **Rôle :**
- **Attributs principaux :**

```
private int nbLig;
private int nbCol;
private Salle[][] grille;
private Jeu jeu;
```

Int nbLig : pour les lignes du plateau.

Int nbCol : Pour les colonnes du plateau.

Salle[][] grille : un tableau des salles qui se trouvent sur le plateau.

Jeu jeu : pour relier au contexte global du jeu.

- **Getters et Setters :**

Restitue le nombre effectif de lignes occupables (sans compter les bords).

```
public int getNbLig() {
    return nbLig;
}
```

Restitue le nombre effectif de colonnes occupables (sans compter les bords).

```
public int getNbCol() {
    return nbCol;
}
```

- **Méthodes :**
  Constructeur :

Créer un objet Plateau en définissant :

son **nombre de lignes** jouables (nbLig) ; son **nombre de colonnes** jouables (nbCol) ;

le **jeu auquel il appartient** (jeu) — ce lien permet au plateau de connaître le contexte global (ex : pour accéder au collecteur ou au joueur si besoin).

On **enregistre les dimensions** du plateau dans les attributs nbLig et nbCol.
Le plateau garde une **référence vers le jeu** pour pouvoir accéder à des informations ou objets globaux (comme les adversaires, le collecteur, etc.).
Puis on appelle la méthode qui remplie le plateau.

```
public Plateau(int nbLig, int nbCol, Jeu jeu) {
    this.nbLig = nbLig;
    this.nbCol = nbCol;
    this.jeu = jeu;
    initContenu(); // création des salles
}
```

public Salle getSalle(int lig, int col) → Restitue la salle d'indices lig et col. Si lig et col ne sont pas valide, la méthode restitue null.
public void setSalle(Salle s, int lig, int col) → Affecte une salle dans le plateau à la position lig-col si cette position est valide.
private boolean isValide(int lig, int col) → Restitue true si et seulement si les numéros de ligne lig et de colonne col sont valides(c'est-à-dire que si c'est une position qui n'est pas sur les bords et qui ne dépasse pas les bords).

public Salle getVoisine(SalleDedans s, Direction d) → Elle sert à trouver la salle voisine d'une salle donnée (`s`) dans une direction précise (`d`). C'est essentiel pour permettre le déplacement des personnages sur le plateau.

```
public Salle getVoisine(SalleDedans s, Direction d) {

    int lig = s.getLig() + d.getdLig();
    int col = s.getCol() + d.getdCol();

    if (lig >= 1 && lig <= nbLig && col >= 1 && col <= nbCol) {
        return grille[lig][col];
    }

    return null; // Si c'est une bordure, retourne null
}
```

On récupère la position actuelle de la salle s (lig pour ligne, col pour colonne).

Puis on ajoute les décalages fournis par la direction d :

d.getdLig() : variation verticale (haut ou bas),

d.getdCol() : variation horizontale (gauche ou droite).

Cela nous donne les coordonnées de la salle voisine dans cette direction.

if (lig >= 1 && lig <= nbLig && col >= 1 && col <= nbCol) {

return grille[lig][col];

}

On vérifie que la salle calculée est dans la zone jouable (intérieure du plateau).

lig >= 1 et lig <= nbLig → on reste entre les vraies lignes jouables,

col >= 1 et col <= nbCol → pareil pour les colonnes.

Les lignes/colonnes valides commencent à 1, car les bordures sont à 0 et nbLig+1 / nbCol+1.

Si les coordonnées sont valides, on renvoie la salle située à cet emplacement (grille[lig][col]).

Si la position est **hors de la zone jouable**, on retourne null.

private void initContenu()

```
private void initContenu() {
    grille = new Salle[nbLig + 2][nbCol + 2];

    for (int i = 0; i < grille.length; i++) {
        for (int j = 0; j < grille[0].length; j++) {
            if (i == 0 || i == nbLig + 1 || j == 0 || j == nbCol + 1) {
                grille[i][j] = new Bordure(i, j, this);
            } else {
                grille[i][j] = new SalleDedans(i, j, this);
            }
        }
    }
}
```

grille = new Salle[nbLig + 2][nbCol + 2];

On crée un tableau 2D de `Salle`, avec **2 lignes et 2 colonnes en plus** que la taille du plateau jouable.
Cela permet de réserver automatiquement les **bords du plateau** (haut, bas, gauche, droite), sans traitement spécial ailleurs dans le code.

for (int i = 0; i < grille.length; i++) {

for (int j = 0; j < grille[0].length; j++) {

On parcourt chaque cellule du tableau : i = ligne, j = colonne
Cette double boucle couvre toute la surface de la grille, y compris les futures bordures.

if (i == 0 || i == nbLig + 1 || j == 0 || j == nbCol + 1) {

grille[i][j] = new Bordure(i, j, this);

} else {

grille[i][j] = new SalleDedans(i, j, this);

}

On test les bordures ou les salles jouable, si la position (i, j) est **sur une bordure** (ligne 0, dernière ligne, colonne 0, dernière colonne), alors on y place une **salle de type Bordure**. Sinon, on crée une **salle classique SalleDedans**, dans laquelle des personnages ou bidons pourront être placés.

public String toString()

```
@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < grille.length; i++) {
        for (int j = 0; j < grille[0].length; j++) {
            sb.append(grille[i][j].toString());
        }
        sb.append("\n");
    }
    return sb.toString();
}
```

StringBuilder sb = new StringBuilder();

On crée un objet StringBuilder, qui permet de construire du texte progressivement de manière très efficace (mieux que String += ... dans une boucle).
Il sert ici à assembler le plateau ligne par ligne sous forme de chaîne de caractères.

for (int i = 0; i < grille.length; i++) {

for (int j = 0; j < grille[0].length; j++) {

sb.append(grille[i][j].toString());

}

sb.append("\n");

}

Cette double boucle :

parcourt chaque case de la grille (i = ligne, j = colonne) pour chaque salle, elle appelle toString() (chaque type de salle a son propre symbole : ".", "A", "J", etc.) après chaque ligne complète, on ajoute un retour à la ligne "\n" pour afficher le plateau de manière lisible.

return sb.toString();
Une fois toutes les lignes ajoutées, on transforme le StringBuilder en String.
Cette chaîne représente **l'état visuel complet du plateau**, prêt à être affiché dans le terminal.

❖ Salle

- **Rôle :** case du plateau, pouvant contenir un personnage ou un bidon.
- **Attributs principaux :**

```
private int lig;
private int col;
private Plateau plateau;
private Bidon bidon;
private Personnage personnage;
```

Int lig : La ligne de la salle dans la grille du plateau (coordonnée Y).
Int col :La colonne de la salle dans la grille du plateau (coordonnée X).
Plateauplateau : Référence au plateau auquel cette salle appartient. Utile pour accéder aux autres salles, vérifier les voisins, etc.
Bidon bidon : Contient un objet d'énergie que le joueur peut récupérer s'il est présent dans cette salle.
Personnage personnage : Contient un joueur ou un adversaire qui occupe actuellement la salle. Peut être null si la salle est vide.

- **Méthodes :**
  Constructeur :
  Créer une salle seulement avec sa ligne et sa colonne.

```
public Salle(int lig, int col) {
    this.setLig(lig);
    this.setCol(col);
}
```

C'est un **constructeur complet** qui sert à **instancier une salle avec tous ses éléments** directement au moment de sa création, pour éviter de devoir les ajouter un par un ensuite.

```
public Salle(int lig, int col, Plateau plateau, Bidon bidon, Personnage personnage) {
    this.setLig(lig);
    this.setCol(col);
    this.setPersonnage(personnage);
    this.setBidon(bidon);
    this.setPlateau(plateau);

}
```

public boolean hasBidon() ; verification si ya un bidon
public boolean hasPersonnage() ; verification si ya un personnage
public abstract void entre(Personnage p) ; Elle permet d'abstraire le comportement d'entrée dans une salle, en laissant chaque sous-classe définir sa propre logique**.**

❖ SalleDedans

- **Rôle :** Représente une salle interne au plateau, qui contient potentiellement un personnage
- **Attributs principaux :** Elle hérite les attributs de sa classe mère qui est Salle.

- **Méthodes :**
  Constructeur :

```
public SalleDedans(int lig, int col, Plateau p) {
    super(lig, col, p, null, null);
}
```

Il appelle le **constructeur de la classe parente `Salle`** via super(…) en ne mettant aucun bidon dans la salle et aucun personnage.
public Salle getVoisine(Direction d) : Restitue la salle voisine dans une direction donnée

public void entre(Personnage p)

```
@Override
public void entre(Personnage p) {
    Personnage occupant = this.getPersonnage();

    if (occupant != null) {
        p.rencontre(occupant);
        if (this.getPersonnage() == null) {
            p.migre(this);
        }
    } else {
        p.migre(this); // Salle vide
    }
}
```

Personnage occupant = this.getPersonnage();
On récupère le personnage **actuellement présent dans la salle** (ou null si la salle est vide).

if (occupant != null) {
p.rencontre(occupant);
if (this.getPersonnage() == null) {
p.migre(this);
}
}

p.rencontre(occupant) : Le personnage p déclenche un **combat** avec le personnage déjà présent (occupant).

Ensuite, on vérifie si **la salle a été libérée** (c'est-à-dire que le personnage occupant a été neutralisé et retiré de la salle).

Si oui : p.migre(this) permet **à p d'entrer dans la salle** (migration effective).

else {

p.migre(this); // Salle vide

}

Si personne n'occupait la salle dès le départ, p y **entre directement** (migration sans combat).

```java
@Override
public String toString() {
    if (getPersonnage() instanceof Joueur) {
        return "J";
    }
    if (getPersonnage() instanceof Adversaire) {
        return "A";
    }
    if (getBidon() != null) {
        return "0";
    }
    return ".";
}
```

Si la salle contient un objet de type Joueur, on retourne "J".
Sinon, si le personnage est un Adversaire, on retourne "A".
Si la salle contient un bidon d'énergie, elle est représentée par "0".
Si aucun personnage ni bidon n'est présent, la salle est vide, on retourne ".".

❖ SalleBidon

- **Rôle :** Représente une salle interne au plateau qui contient un bidon.
- **Attributs principaux :** Elle hérite les mêmes attributs que hérite sa classe mère qui est SalleDedans.

- **Méthodes :**
  Constructeur : Appelle le constructeur de la classe parente SalleDedans pour initialiser la position (ligne et colonne) ainsi que le plateau (Plateau p) auquel elle appartient. Ajoute dans cette salle un **nouveau bidon plein** en créant une instance de la classe Bidon.

```java
public SalleBidon(int lig, int col, Plateau p) {
    super(lig, col, p);
    this.setBidon(new Bidon());
}
```

Public String toString()

```
@Override
public String toString() {
    return super.toString();
}
```

Même principe que le toString de sa classe mère en l'appelant à travers **super.toString().**

public void entre(Personnage p)

```
@Override
public void entre(Personnage p) {
    Personnage occupant = this.getPersonnage();

    if (occupant != null) {
        p.rencontre(occupant);

        // Si le combat libère la salle, migration du gagnant
        if (this.getPersonnage() == null) {
            p.migre(this);

            if (this.getBidon() != null) {
                p.prendEnergie(this.getBidon());
            }
        }
    } else {
        // Salle vide -> migration directe
        p.migre(this);

        if (this.getBidon() != null) {
            p.prendEnergie(this.getBidon());
        }
    }
```

Personnage occupant = this.getPersonnage();
On récupère le personnage actuellement dans la salle (s'il y en a un). Cela permet de savoir si la salle est occupée ou non.

if (occupant != null) {

p.rencontre(occupant);

S'il y a déjà quelqu'un, on déclenche la **rencontre**, qui lancera un **combat** via rencontre().

```
if (this.getPersonnage() == null) {
   p.migre(this);
   if (this.getBidon() != null) {
      p.prendEnergie(this.getBidon());
   }
}
```

Si le personnage déjà présent est neutralisé (donc supprimé de la salle), le nouvel arrivant peut migrer dans la salle, puis **prendre de l'énergie du bidon**, si présent.

```
} else {
   p.migre(this);
   if (this.getBidon() != null) {
      p.prendEnergie(this.getBidon());
   }
}
```

Si la salle était vide dès le départ, le personnage migre immédiatement, et il peut aussi se servir dans le bidon s'il y en a un.

❖ Bordure

- **Rôle :** Représente les salles qui matérialisent le bord du plateau, auxquelles les personnages ne peuvent pas accéder
- **Attributs principaux :** Elle hérite les attributs de sa classe mère qui est Salle.
- **Méthode :**

  Constructeur : Il appelle le **constructeur de la classe parente** `Salle` via super(…) en ne mettant aucun bidon dans la salle et aucun personnage.

```
public Bordure(int lig, int col, Plateau p) {
    super(lig, col, p, null, null);

}
```

Public String toString() Restitue l'aspect du bord du plateau.

```
@Override
public String toString() {
    return "X";
}
```

Public void entre(Personnage p) : Si un personnage essaye de se diriger vers les bordures il a automatiquement un message à l'écran qui le lui interdit.

```java
@Override
public void entre(Personnage p) {
    System.out.println("Le personnage ne peut pas entrer dans une salle de bordure.");
}
```

❖ Reservoir

- **Rôle :** Représente une réserve d'énergie, qui peut être remplie ou vidée.
- **Attributs principaux :**

```java
// Attribut contenant la quantité d'énergie actuelle
private int energie;

// Valeur maximale par défaut, peut être utilisée par les classes filles
protected static final int ENERGIE_MAX = 10;
```

- **Méthode :**
  Constructeur : Ce constructeur rend possible la création d'un **réservoir d'énergie personnalisé.**

```java
public Reservoir(int energie) {
    setEnergie(energie);
}
```

public int getContenu() Restitue la quantité d'énergie disponible
private void setEnergie(int energie) Affecte une quantité d'énergie initiale, en vérifiant si l'énergie entré en paramètre est positive sinon on l'a met à 0.
public boolean estVide() Restitue true si la réserve est vide
public abstract int ajustementAjout(int montant); Ajustement du montant positif que l'on veut ajouter à un réservoir d'énergie. Le code dépend du type de réservoir.
public int ajustementRetrait(int montant) Ajustement du montant positif que l'on veut retirer au réservoir. Le montant que l'on peut retirer ne peut pas dépasser la quantité disponible.
public void modifEnergie(int montant) Modifie la quantité d'énergie disponible d'un montant spécifié positif ou négatif.

Public void transfert(int montant, Reservoir autre) Effectuer un transfert d'énergie d'un réservoir source (`autre`) vers le réservoir courant (`this`)

```java
public void transfereDe(int montant, Reservoir autre) {
    if (montant <= 0 || autre == null) {
        return;
    }

    int possibleADonner = autre.ajustementRetrait(montant);
    int possibleARecevoir = this.ajustementAjout(possibleADonner);
    int montantEffectif = Math.min(possibleADonner, possibleARecevoir);

    autre.modifEnergie(-montantEffectif); // retire de la source
    this.modifEnergie(montantEffectif);   // ajoute à la destination
}
```

Si le montant est nul ou négatif, ou si la source est null, on annule l'opération immédiatement.

possibleADonner : quantité que la source **peut donner** sans dépasser son contenu.

possibleARecevoir : quantité que le réservoir courant **peut accepter**.

montantEffectif : on **prend le minimum** des deux pour ne jamais dépasser l'un ou l'autre.

On applique le transfert :

On **retire** à la source (autre) l'énergie effectivement transférée,

On **ajoute** à this cette même quantité.

❖ Collecteur

- **Rôle :** Ou le joueur peut stocker de l'énergie, et à la fin du jeu c'est le contenu du collecteur qui sera le score du joueur si il gagne évidemment.
- **Attributs principaux :** Sont hérités de sa classe mère.
- **Méthode :**
  Constructeur : création d'un collecteur vide

```java
public Collecteur() {
    super(0);
}
```

- **public int ajustementAjout(int montant)** Si montant est **positif**, la méthode retourne ce montant tel quel. Si montant est **négatif**, elle retourne 0.

public int ajoustementRetrait(int montant)

Si on essaie de retirer une quantité **strictement positive**, un message d'erreur s'affiche. Dans **tous les cas**, la méthode retourne 0. C'est pas possible de retirer de l'énergie dans le collecteur.

```java
@Override
public int ajustementRetrait(int montant) {
    if (montant > 0) {
        System.out.println("Erreur : impossible de retirer de l'énergie au collecteur.");
    }
    return 0;

}
```

❖ ReserveLimitee

- **Rôle :** sert à modéliser une réserve d'énergie qui a une capacité énergétique maximale, ici spécialement pour les personnages et les bidons.
- **Attributs principaux :** Sont hérités de sa classe mère.
- **Méthode :**
  Constructeur : Ce constructeur permet de créer rapidement une réserve limitée à 10 d'énergie **sans devoir spécifier le max manuellement** à chaque fois.

```java
public ReserveLimitee(int energie) {
    this(energie, 10);
}
```

public ReserveLimitee(int energie) Création d'une réserve avec une capacité maximale par défaut.
public int getMax() obtenir la capacité max

public int ajustementAjout(int montant)

Si le montant est négatif ou nul, aucun ajout n'est permis. On retourne donc 0.

Si le montant est négatif ou nul, aucun ajout n'est permis. On retourne donc 0.

```
@Override
public int ajustementAjout(int montant) {
    if (montant <= 0) {
        return 0;
    }

    int manque = max - getContenu(); //ce
    return Math.min(montant, manque);
}
```

Si le montant est négatif ou nul, aucun ajout n'est permis. On retourne donc 0.

On calcule combien d'énergie **il manque** pour atteindre le maximum (max).
On ne peut pas ajouter plus que ce qui manque.

❖ Bidon

- **Rôle :** Ce sont des réserves limitées d'énergie (avec un maximum d'énergie) auxquelles on ne peut pas ajouter d'énergie ; peut seulement en prendre.
- **Attributs principaux :** Sont hérités de sa classe mère.
- **Méthode :**
  Constructeur :création d'un bidon plein, avec un maximum

```
public Bidon() {
    super(10, 10);
}
```

Public int ajustementAjout(int montant)

Si on essaie d'ajouter une énergie **strictement positive**, un message d'erreur est affiché.

La méthode retourne toujours 0, donc **aucune énergie n'est ajoutée**, quelle que soit la demande.

```
@Override
public int ajustementAjout(int montant) {
    if (montant > 0) {
        System.out.println("Impossible d'ajouter de l'énergie dans un bidon !");
    }
    return 0;
}
```

public String toString() Restitue le symbole d'un bidon. Utile pour afficher l'état du plateau.

```
@Override
public String toString() {
    return "0";
}
```

❖ Personnage

- **Rôle :** Représente les personnages du jeu qui, dans cette version, peuvent être un Joueur ou un Adversaire.
- **Attributs principaux :**

```
private int inertie;
private int energie;
private SalleDedans position;
```

int inertie : l'inertie d'un personnage
int energie : son energie
SalleDedans position : la position de la salle ou il se trouve.

- **Méthode :**

Constructeur : crée un personnage avec inertie et une energie.

```
public Personnage(int inertie, int energie) {
    this.setInertie(inertie);
    this.setEnergie(energie);
}
```

public int getForce() Restitue la force du personnage (sont inertie fois son énergie).
public int getEnergie() Restitue la quantité d'énergie restante pour le personnage.
public void decEnergie() Diminue l'énergie du personnage d'une unité (exécuté à chaque changement de salle effectif).
public boolean estNeutralise() Restitue true si la réserve d'énergie du personnage est vide.
public abstract void prendEnergie(ReserveLimitee r); action d'un personnage pour prendre de l'énergie dans une réserve limitée**.**
public abstract void rencontre(Personnage p); rencontre d'un personnage(selon le type de personnage).

public abstract void perd(); Action à effectuer après la perte d'un combat. Le code dépend du type de Personnage(qui est implémentée selon le personnage).
public void migre(SalleDedans destination)

```
public void migre(SalleDedans destination) {
    if (destination.getPersonnage() == null) {
        // Libère la salle actuelle
        if (this.position != null) {
            this.position.setPersonnage(null);
        }

        // Affecte la nouvelle salle
        destination.setPersonnage(this);
        this.setPosition(destination);

        // Perte d'énergie
        this.decEnergie();
    } else {
        // La salle est occupée : on ne migre pas
        System.out.println("Migration échouée : salle occupée.");
    }
}
```

Le personnage ne peut migrer **que si la salle destination est vide** (aucun autre personnage dedans).
Si le personnage est déjà dans une salle (non null), alors cette salle est **vidée de lui** (on enlève sa référence).
destination.setPersonnage(this);
this.setPosition(destination);

Le personnage est ajouté à la nouvelle salle (setPersonnage(this)). Sa nouvelle position est mise à jour (setPosition(destination)).

System.out.println("Migration échouée : salle occupée."); Si un autre personnage est déjà présent, le déplacement est **refusé** et un message est affiché.

public void prendEnergie(Personnage autre)

```
public void prendEnergie(Personnage autre) {
    int aPrendre = Math.min(10 - this.getEnergie(), autre.getEnergie());
    autre.setEnergie(autre.getEnergie() - aPrendre); // on vide l'autre
    this.ajouteEnergie(aPrendre); // on remplit ce qu'on peut
}
```

int aPrendre = Math.min(10 - this.getEnergie(), autre.getEnergie());

On prend **le minimum** entre : Ce qui manque à ce personnage pour atteindre 10 d'énergie (10 - this.getEnergie()) et ce que l'autre possède réellement (autre.getEnergie()).

Cela **évite les dépassements** : ni dépasser 10, ni prendre plus que ce que l'autre a.

On retire aPrendre à l'autre personnage. On **ajoute l'énergie récupérée** à soi-même (dans la limite imposée par ajouteEnergie qui utilise un Math.min()).

public static void combat(Personnage p1, Personnage p2)

```
public static void combat(Personnage p1, Personnage p2) {

    if (p1.getForce() >= p2.getForce()) {
        p1.prendEnergie(p2);
        p2.perd();
    } else {
        p2.prendEnergie(p1);
        p1.perd();
    }
```

Si p1 a une **force supérieure ou égale** à p2, alors p1 **gagne** le combat.

p1 prend de l'énergie à p2 (dans la limite de ce qu'il peut recevoir et de ce que p2 possède).

p2 réagit à la défaite (selon son type : joueur ou adversaire).

Si p2 est plus fort, alors **même logique inversée** : c'est p2 qui gagne.

❖ Joueur

- **Rôle :** Sorte de personnage, personnage principale contrôlé par l'utilisateur.
- **Attributs principaux :**

```
private Jeu jeu;
```

sert à **référencer l'instance du jeu en cours**. Cela permet au joueur d'interagir directement avec certains éléments globaux du jeu.

- **Méthode :**

Constructeur : Ce constructeur crée un joueur avec : une **inertie constante**, une **énergie initiale personnalisée**, et une **connexion directe avec le jeu** pour gérer les règles ou actions plus complexes.

```
public Joueur(Plateau plateau, int energieMax, Jeu jeu) {
    super(5, energieMax);
    this.jeu = jeu;
```

public void rencontre(Personnage p) Action à effectuer quand un joueur (le joueur) rencontre un autre personnage. A la rencontre d'un personnage un combat est déclenché entre les deux**.**

public String toString() pour reprensenter le symbole du joueur.

Public void perd()

```
@Override
public void perd() {
    if (this.estNeutralise()) {
        System.out.println("Le joueur a été neutralisé. Fin du jeu !");
        if (this.getPosition() != null) {
            this.getPosition().setPersonnage(null); // libère la salle
            this.setPosition(null);                 // sécurité anti-bug
        }
        jeu.setFini(true); //on termine la partie
    }
}
```

if (this.estNeutralise()) { On vérifie si le joueur a **0 énergie**. Si ce n'est pas le cas, on ne fait rien.

Affiche un message de fin au joueur.

if (this.getPosition() != null) {

this.getPosition().setPersonnage(null); // supprime le joueur de la salle

this.setPosition(null); // évite des bugs liés à une position fantôme

}

On **efface la trace du joueur du plateau**, pour éviter qu'il soit encore visible ou manipulable.

jeu.setFini(true); On signale au système de jeu que la partie est **terminée**. Cela provoque l'arrêt de la boucle principale dans la méthode joue ( ) .

Public void avance()

```
@Override
public void avance() {
    Direction d = Direction.getDirectionQuelconque(); //
    Salle destination = this.getPosition().getVoisine(d);
    if (destination != null) {
        destination.entre(this);
    } else {
        System.out.println("Déplacement impossible.");
    }
}
```

Récupère une **direction quelconque** (nord, sud, est, ouest, diagonales…), soit saisie par l'utilisateur (pour un Joueur), soit calculée ou tirée aléatoirement (pour un Adversaire).

À partir de la **position actuelle**, on regarde **quelle salle se trouve dans cette direction**. Si la direction mène vers une bordure ou une case invalide, on obtient null.

if (destination != null) {

destination.entre(this);

}

Si la salle voisine **existe** (elle n'est pas une bordure ou hors limite), on tente d'y **entrer** via entre(this).

Si la salle n'existe pas, on affiche un message et **aucune action n'est faite**.

Public void prendEnergie(ReserveLimitee r)

```
@Override
public void prendEnergie(ReserveLimitee r) {
    System.out.println("Énergie disponible : " + r.getContenu());

    int demande = Lire.i("Combien d'énergie voulez-vous prendre ?");
    if (demande < 0) {
        demande = 0;
    }
    if (demande > r.getContenu()) {
        demande = r.getContenu();
    }

    // Retirer l'énergie du bidon ou adversaire
    r.modifEnergie(-demande);

    int choix = Lire.i("1 - Recharger ma propre énergie\n2 - Stocker dans le collecteur\n Veuillez choisir 1 ou 2");
    if (choix == 1) {
        this.ajouteEnergie(demande); // méthode à ajouter dans Personnage
    } else {
        int accepte = jeu.getCollecteur().ajustementAjout(demande);//demande au collecteur si on peut ajouter demande
        jeu.getCollecteur().modifEnergie(accepte);//et on ajoute cette energie (demande) au collecteur
    }
}
```

Affiche la quantité actuelle d'énergie dans la réserve concernée.
Le joueur choisit une quantité avec le **Lire.i()**. Ensuite on vérifie qu'elle est valide.
if (demande < 0) demande = 0;
if (demande > r.getContenu()) demande = r.getContenu();
On s'assure que le joueur **ne demande pas plus que ce qui est disponible**, et pas de valeur négative.
L'énergie est **retirée du bidon ou adversaire** (réserve d'origine).
Le joueur choisit quoi faire avec l'énergie prise.
this.ajouteEnergie(demande);
On appelle une méthode qui ajoute l'énergie au joueur (en respectant la limite de 10).
int accepte = jeu.getCollecteur().ajustementAjout(demande);
jeu.getCollecteur().modifEnergie(accepte);
On demande au collecteur combien il peut accepter (ajustementAjout), puis on lui ajoute cette énergie via modifEnergie.

❖ Adversaire

- **Rôle :** Sorte de personnage, qui joue contre le joueur principal.
- **Attributs principaux :**

```
private Joueur joueur;
```

Pour suivre ou fuir le joueur.

- **Méthode :**

Constructeur : Ce constructeur initialise un **adversaire** dans le jeu, en lui donnant ses caractéristiques de base **(inertie, énergie)** et une **référence directe vers le joueur**.

```
public Adversaire(int inertie, int energie, Joueur joueur) {
    super(inertie, energie);
    this.joueur = joueur;
}
```

public void prendEnergie(ReserveLimitee r) Action spécifique à un adversaire pour prendre de l'énergie dans une réserve limitée. Par principe, un adversaire prend toute l'énergie qu'il peut dans une réserve (bidon ou l'énergie du joueur).
public void rencontre(Personnage p) un adversaire ne peut pas combattre un autre adversaire donc on regarde si le personnage est une instance de joueur, si oui on déclenche un combat entre eux.
public void perd() Action à effectuer après la perte d'un combat contre le joueur. Si l'adversaire est neutralisé (Ce n'est pas obligatoire ; le joueur n'a pas forcément pu prendre toute son énergie), il libère la place dans la salle**.**
public String toString() restitue le symbole de l'adversaire.

public Direction getDirectionVersJoueur() pour traquer le joueur.

```
public Direction getDirectionVersJoueur() {

    if (joueur == null || joueur.estNeutralise() || joueur.getPosition() == null) {
        return new Direction(0, 0);
    }
    //recuperation de la position du joueur
    int ligJ = joueur.getPosition().getLig();
    int colJ = joueur.getPosition().getCol();

    //recuperation de la position de l'adversaire lui meme
    int ligThis = this.getPosition().getLig();
    int colThis = this.getPosition().getCol();

    //on compare les differences de position entre le joueur et l'adversaire

    // calcule de la direction verticale à prendre
    int dLig = Integer.compare(ligJ - ligThis, 0);

    // calcule de la direction horizontale à prendre
    int dCol = Integer.compare(colJ - colThis, 0);


    return new Direction(dLig, dCol);
}
```

Si le joueur **n'existe pas**, est **neutralisé** ou **n'a pas de position**, on retourne une **direction nulle** (0,0), qui signifie : *« ne pas bouger »*.

On récupère les **coordonnées ligne/colonne** du joueur (ligJ, colJ) et de **l'adversaire** lui-même (ligThis, colThis).

int dLig = Integer.compare(ligJ - ligThis, 0);

int dCol = Integer.compare(colJ - colThis, 0);

Ces lignes **comparent les coordonnées** :

Si le joueur est **plus bas** que l'adversaire → dLig = 1 (descendre)

Si plus haut → dLig = -1 (monter)

Même ligne → dLig = 0
Idem pour dCol : -1, 0 ou 1 selon si le joueur est à gauche, même colonne, ou à droite.

return new Direction(dLig, dCol);

Crée un nouvel objet Direction avec les décalages verticaux et horizontaux calculés

❖ AdversaireVelleitaires

- **Rôle :** Sorte d'adversaire, comportement aléatoire biaisé.
- **Attributs principaux :**

```
private Random rand = new Random();
```

Pour leur déplacement aléatoire.

- **Méthodes** :
Constructeur : **créer un adversaire velléitaire** avec des valeurs spécifiques d'inertie, d'énergie et un joueur cible en appelant le constructeur de sa classe mère.

```
public AdversaireVelleitaires(int inertie, int energie, Joueur joueur) {
    super(inertie, energie, joueur);
}
```

Public void avance()

```
public void avance() {
    if (this.estNeutralise()) {
        System.out.println("AdversaireVelleitaire : neutralisé, il ne bouge pas.");
        return;
    }
```

Si l'adversaire est **neutralisé** (énergie à 0), il ne peut plus se déplacer.

```
Direction versJoueur = getDirectionVersJoueur();
Direction d;
int chance = rand.nextInt(100);
```

On calcule la **direction vers le joueur**, puis on tire **un nombre aléatoire entre 0 et 99** pour déterminer le comportement.

70 % de chances que l'adversaire agisse « intelligemment » :
il **poursuit** si plus fort que le joueur, **fuit** sinon.

```
// 70% de chances d'aller dans le bon sens
if (chance < 70) {

    if (this.getForce() >= getJoueur().getForce()) {
        d = versJoueur;
    } else {
        d = versJoueur.getInverse();
```

```
else if (chance >= 95) {
    System.out.println("AdversaireVelleitaire : ne bouge pas (comportement aléatoire).");
    return;
```

5 % de chances de ne pas bouger du tout.

```
else {
    // Tire aléatoirement une direction parmi les directions valides
    d = Direction.getDirection(new Random().nextInt(8)); // 0 à 7
    System.out.println("AdversaireVelleitaire : se déplace de manière aléatoire.");
}
```

25 % de chances de choisir une direction complètement aléatoire parmi les 8 directions possibles.

```
if (destination instanceof SalleDedans sd && sd.getPersonnage() == null) {
    System.out.println("AdversaireVelleitaire : se déplace vers (" + d.getdLig() + "," + d.getdCol() + ").");
    destination.entre(this);
```

On cherche la salle voisine dans la direction choisie.
S'il s'agit bien d'une salle interne (`SalleDedans`) **et qu'elle est vide**, alors l'adversaire y entre.

```
else {
    System.out.println("AdversaireVelleitaire : déplacement impossible (salle occupée ou invalide), il passe son tour.");
```

Sinon (salle occupée ou hors plateau), il ne fait rien.

❖ AdversaireDetermines

- **Rôle :** Sorte d'adversaire, suit ou fuit systématiquement.
- **Attributs principaux :** hérite de sa classe mère.
- **Méthodes :**
  Constructeur : créer un adversaire déterminé avec des valeurs spécifiques d'inertie, d'énergie et un joueur cible en appelant le constructeur de sa classe mère.

```
public AdversairesDetermines(int inertie, int energie, Joueur joueur) {
    super(inertie, energie, joueur);
}
```

Public void avance()

```
public void avance() {
    if (this.estNeutralise()) {
        System.out.println("AdversaireDéterminé : neutralisé, il ne bouge pas.");
        return;
    }
```

Si l'adversaire est **neutralisé** (énergie à 0), il ne peut plus se déplacer.

```
Direction d = getDirectionVersJoueur();
```

On calcule la **direction à prendre pour aller vers le joueur** (haut, bas, diagonale, etc.).

```
if (d.getdLig() == 0 && d.getdCol() == 0) {
    System.out.println("AdversaireDéterminé : déjà sur la case du joueur, il ne bouge pas.");
    return;
}
```

Si la direction est (0,0), cela signifie que l'adversaire est **déjà sur la même case que le joueur** donc pas de mouvement.

```
if (this.getForce() < getJoueur().getForce()) {
    d = d.getInverse();
    System.out.println("AdversaireDéterminé : fuit le joueur.");
} else {
    System.out.println("AdversaireDéterminé : poursuit le joueur.");
}
```

L'adversaire regarde sa force :

S'il est **moins fort** que le joueur, il **fuit** (inverse de la direction). Sinon, il **le poursuit**.

```
Salle destination = getPosition().getVoisine(d);
```

On demande la salle voisine dans la direction choisie.

```
if (destination instanceof SalleDedans sd && sd.getPersonnage() == null) {
    System.out.println("AdversaireDéterminé : se déplace vers (" + d.getdLig() + "," + d.getdCol() + ").");
    destination.entre(this);
```

Si la **salle est une SalleDedans vide**, l'adversaire s'y déplace.

```
else {
  System.out.println("AdversaireDéterminé : déplacement impossible (salle occupée ou invalide), il passe son tour.");
```

Si la salle est occupée ou invalide (bordure par exemple), il ne bouge pas.

❖ AdversaireIntelligents

- **Rôle :** Sorte d'adversaire, évalue les directions autour.
- **Attributs principaux :**hérite de sa classe mère.
- **Méthode :**

Constructeur : créer un adversaire intelligent avec des valeurs spécifiques d'inertie, d'énergie et un joueur cible.

```
public AdversairesIntelligents(int inertie, int energie, Joueur joueur) {
    super(inertie, energie, joueur);
}
```

public void avance()

```
public void avance() {
    if (this.estNeutralise()) return;
```

Si l'adversaire est **neutralisé** (énergie à 0), il ne peut plus se déplacer.

```
Direction versJoueur = this.getDirectionVersJoueur();
Direction directionCible;
```

On récupère la direction à suivre pour aller vers le joueur, grâce à la méthode getDirectionVersJoueur().

```
if (this.getForce() >= getJoueur().getForce()) {
    System.out.println("L'adversaire intelligent poursuit le joueur.");
    directionCible = versJoueur;
} else {
    System.out.println("L'adversaire intelligent fuit le joueur.");
    directionCible = versJoueur.getInverse();
}
```

L'adversaire **compare sa force avec celle du joueur** :

Il **poursuit** s'il est plus fort ou égal. Il **fuit** dans la direction inverse sinon.

```
ArrayList<Direction> directions = getDirectionsParProximite(directionCible);
```

On génère une **liste de directions triées par pertinence**, à partir de la cible :

D'abord la direction principale, Puis les voisines (trigonométriquement), Puis les autres directions (en complément).

```
for (Direction d : directions) {
    Salle destination = this.getPosition().getVoisine(d);
    if (destination instanceof SalleDedans sd && sd.getPersonnage() == null) {
        System.out.println("AI avance vers : " + d);
        destination.entre(this);
        return;
    }
}
```

L'Adversaire intelligent essaie **chaque direction dans l'ordre**, et entre dans la première salle vide trouvée.

```
System.out.println("AI ne peut pas bouger (toutes les directions sont bloquées). Il passe son tour.");
```

Si **aucune direction n'est praticable**, l'adversaire ne bouge pas.

private ArrayList<Direction> getDirectionsParProximite(Direction cible) Elle retourne une liste de directions, classées **par proximité** de la direction principale (`cible`), pour tenter un déplacement intelligent.

```
ArrayList<Direction> directions = new ArrayList<>();
```

On crée une liste vide qui contiendra toutes les directions possibles que l'adversaire pourra tenter.

```
directions.add(cible);
```

On commence par ajouter **la direction cible** (la plus logique), qui est soit vers le joueur, soit dans la direction opposée (s'il fuit).

```
Direction gauche = cible.getPred();
Direction droite = cible.getSucc();

if (gauche != null) directions.add(gauche);
if (droite != null) directions.add(droite);
```

On récupère les **voisines trigonométriques** de la direction cible :

getPred() : la direction qui précède dans le sens trigonométrique (ex. haut → haut-gauche),

getSucc() : celle qui suit (ex. haut → haut-droite).

```
for (Direction d : new Direction[] {
        Direction.HAUT, Direction.BAS, Direction.GAUCHE,
        Direction.DROITE, Direction.HAUT_DROITE,
        Direction.HAUT_GAUCHE, Direction.BAS_DROITE, Direction.BAS_GAUCHE
}) {
    if (!directions.contains(d)) {
        directions.add(d);
    }
}
```

Ce bloc :

Parcourt **toutes les 8 directions possibles** (diagonales comprises), ajoute uniquement celles **qui ne sont pas déjà dans la liste**, cela garantit que l'adversaire aura toujours une **liste complète et ordonnée**, avec la direction préférée en tête.

```
return directions;
```

On retourne la liste priorisée, que l'adversaire utilisera pour tester **chaque direction dans l'ordre**, jusqu'à trouver une salle libre.

❖ Main

```
public static void main(String[] args) {

    // Création d'un jeu avec une petite taille de plateau
    Jeu jeu = new Jeu(5, 7); // 5 lignes, 7 colonnes (hors bordures)

    // Affichage du plateau à l'initialisation
    System.out.println("Affichage initial du plateau :");
    System.out.println(jeu);

}
```

Rôle : Cette méthode **lance tout le jeu** automatiquement via l'appel au constructeur de Jeu. Elle sert aussi à **tester** rapidement l'affichage initial du plateau. Très utile pour vérifier manuellement que la génération aléatoire a bien fonctionné.

V. Conclusion

La réalisation de ce projet m'a offert une occasion précieuse de mettre en pratique les principes fondamentaux de la programmation orientée objet en Java. À travers la conception de ce jeu de plateau complet, j'ai pu développer ma capacité à structurer un programme en plusieurs classes collaboratives, à gérer l'héritage, le polymorphisme, ainsi que l'interaction entre objets.

Le projet m'a également permis de mieux comprendre l'importance de la modularité et de la clarté du code dans un système comportant de nombreuses entités. Les notions de conception, telles que la délégation, la réutilisation de code et l'organisation en couches logiques, ont été appliquées de façon concrète.

Enfin, ce travail m'a apporté une plus grande autonomie dans la résolution de problèmes, dans l'analyse d'un énoncé complexe, et dans le choix de solutions adaptées aux contraintes du jeu. Il s'agit d'une étape formatrice qui renforce ma compréhension du développement logiciel.