

# Compte rendu : *Projet Kotlin*

## *Aventure*

### Tables des matières

<b>Contexte:</b> .....	3
Introduction : .....	3
Sprint 1 : Gestion des Items .....	4
1.1 Création du personnage .....	4
1.2 Modification de la méthode tourJoueur() .....	4
1.3 Modification de la méthode tourMonstre() .....	5
2.1 Classe Arme & TypeArme .....	6
2.2 Classe Armure & TypeArmure .....	6
2.3 Classe Potion & Bombe .....	7
3.0 : Comment faire un test unitaire .....	8
3.1 : Faire un test unitaire pour calculerDegats() .....	8
3.2 : Faire un test unitaire pour calculProtection() .....	8
3.3 : Faire un test unitaire pour la méthode utiliser() de la classe Bombe .....	9
Intermission (Creation de la classe Item) .....	9
4 : Faire hériter les classes Arme, Armure, Potion et Bombe de la classe Item .....	9
5.1 : méthodes attaque() et equipe() .....	10
5.2 : méthodes calculTotalDefense() et equipe() .....	11
5.3 : méthodes avoirPotion(), avoirBombe(), boirePotion() .....	12
6.1 : méthodes afficheInventaire() et loot() .....	13
6.2 : modification de la méthode tourJoueur() .....	13
6.3 : modification de la méthode tourMonstre() .....	13
Intermission 6 : Ajout de la méthode tourJoueur() dans la classe Combat .....	14
Sprint 2 : Gestion du choix de la classe (Guerrier, Voleur, Mage) .....	14
7.1 : Création de la classe Guerrier .....	14
Mission 7.2 : Création de la classe Voleur .....	15
Mission 7.3 : Création de la classe Mage et Sort .....	15
Intermission 7 : .....	16
8.1 : Méthodes toString() equipe() attaque() .....	16
8.2 : Méthode toString() et voler() .....	17
8.3 : Méthode toString(), afficherGrimoire() et choisirEtLancerSort() .....	18
Intermission 8 : .....	19
9.1 : Création de sorts : Boule de feu, Missile magique .....	19
9.2 : Création de sorts : Invocation d'une arme magique, Invocation d'une armure magique. ...	20
9.3 : Création de sort : Sort de soins .....	21

Intermission 9 : .....	22
10.1 : Faire des tests unitaires les sorts de boule de feu et Missile magique .....	22
10.2 : Faire des tests unitaires les sorts d'invocations .....	22
10.3 : Faire des tests unitaires pour le sort de soins .....	22

Projet réalisé par :  
**SONG** Steeven  
**TANDABANY** Adrien  
**THEVARANCHAN** Devamadushan

Du 20/09/2023 au 12/10/2023

## Contexte:

Dans le cadre de notre projet Kotlin Aventure, l'objectif était de nous permettre d'utiliser nos connaissances avec le langage Kotlin acquis en cours et de les développer tout en respectant le cahier des charges.

## Introduction :

Après avoir cerné les objectifs de ce projet, nous verrons comment instancier et modifier des classes, des objets et des méthodes. Puis nous expliquerons les difficultés rencontrées et enfin, nous terminerons par une conclusion personnelle.

**IMPORTANT**

Lien Github : <https://github.com/atandabany/KotlinAventure>

# Sprint 1 : Gestion des Items

## 1.1 Création du personnage

Pour saisir les caractéristiques des scores attaque, defense, endurance, et de vitesse de 40 point maximum, Nous avons utilisé une `do while` qui permet de repeter une boucle et de l'exécuter tant qu'elle est vrai.

```
println("Saisir les points de spécialité. 40 point au maximum")
do {
    println("Point d'attaque : ")
    ptsAttaque = readln().toInt();
    println("Point défense : ")
    ptsDefense = readln().toInt();
    println("Point d'endurance : ")
    ptsEndurance = readln().toInt();
    println("Point de vitesse : ")
    ptsVitesse = readln().toInt();
    val caractéristique: Int = ptsAttaque + ptsDefense + ptsEndurance +
ptsVitesse
} while (caractéristique < 40 || caractéristique > 40)
```

### NOTE

A noter que nous avons décidé que les points ne peuvent être inférieur ou supérieur à 40.

Ensuite nous voulions savoir que à chaque point d'endurance ajouter, les points de vie augmentent de 10. Pour cela nous avons utilisé l'instruction `if` qui permet de vérifier une condition.

```
if (ptsEndurance > 1) {
    ptsVieF = ptsEndurance * 10
}
val ptsVieMin = ptsVieF + ptsVieBase
val ptsVieFinalMax = ptsVieF + ptsVieMax
```

## 1.2 Modification de la méthode `tourJoueur()`

La méthode **`tourJoueur()`** permet au joueur de choisir une action lorsque son tour commence. Le joueur peut choisir une action parmi les suivantes : Attaquer, passer son tour, boire une potion, consulter son inventaire, lancer un sort et voler un item.

```
fun tourDeJoueur() {
    println("\u001B[34m ---Tour de ${this.jeu.joueur.nom} (pv:
${this.jeu.joueur.pointDeVie}) ---")
    //TODO Mission 1.2
    println("Choisir une action : 0 => Attaquer ; 1 => Passer son tour ; 2 =>
Boire une potion ; 3 => Inventaire ; 4 => Lancer un sort ; 5 => Voler un item")
}
```

```

val action = readln()

// Permet de choisir une action en fonction du chiffre indiqué
when (action) {
    "0" -> this.jeu.joueur.attaque(monstre)
    "1" -> println("${this.jeu.joueur.nom} passe son tour...")
    "2" -> this.jeu.joueur.boirePotion()
    "3" -> {
        val posObjet = this.jeu.joueur.afficheInventaire()
        val objet = this.jeu.joueur.inventaire[posObjet]

        if (objet is Bombe) {
            objet.utiliser(monstre)
        } else {
            objet.utiliser(this.jeu.joueur)
        }
    }
    "4" -> {
        val leMage = this.jeu.joueur as Mage
        leMage.choisirEtLancerSort(monstre)
    }
    "5" -> {
        val leVoleur = this.jeu.joueur as Voleur
        leVoleur.volerItem(monstre)
    }
}
println("\u001b[0m")
}

```

#### NOTE

Si le joueur décide d'attaquer le monstre, le joueur devra saisir le chiffre indiqué pour lancer l'action. Par exemple, pour attaque le joueur devra saisir 0 en appelant la méthode **attaque** de la classe **Personnage**. Les actions boire une potion, consulter son inventaire, lancer un sort et voler un item seront expliqués plus tard dans les prochaines missions.

### 1.3 Modification de la méthode tourMonstre()

La méthode choisit aléatoirement une action pour le monstre lors de son tour. Nous utilisons la méthode 'random()' pour choisir un nombre entre 1 et 100. Si le nombre est inférieur ou égal à 70, le monstre attaque. Si le nombre est compris entre 71 et 80, le monstre boit sa potion. Sinon, il passe son tour.

```

fun tourDeMonstre() {
    println("\u001b[31m---Tour de ${monstre.nom} (pv: ${monstre.pointDeVie}) ---")
    var potionMonstre = monstre.avoirPotion()
    var pv = monstre.pointDeVie < monstre.pointDeVieMax / 2

    val attaque = (1..100).random()
    if (attaque <= 70) {

```

```

        this.monstre.attaque(this.jeu.joueur)
    } else if (potionMonstre && pv && attaque <= 80) {
        monstre.boirePotion()

    } else {
        println("${monstre.nom} passe son tour... ")
    }
    println("\u001b[0m")
}

```

#### NOTE

Le monstre peut boire sa **potion** si le nombre est compris entre **71** et **81**, et aussi s'il en a une dans son inventaire. Pour cela, nous utilisons la méthode **avoirPotion()**, et également, le monstre doit avoir des points de vie inférieurs à la moitié de ses points de vie max pour pouvoir boire la potion

## 2.1 Classe Arme & TypeArme

Dans cette mission, nous avons créer les classes Arme et TypeArme et créer la méthode calculerDegats() pour connaître les dégats du personnage en fonction de l'arme équipé.

La méthode consiste à calculer les dégats en fonction, du nombre de tirageDes et d'effectuer un coup critique si le résultat obtenu lors du tirageDes est supérieur a l'activation critique.

```

fun calculerDegats(): Int {
    var resultat = TirageDes(this.type.nombreDes, this.type.valeurDeMax).lance()
    val desCritique = TirageDes(1, 20).lance()
    if (desCritique >= this.type.activationCritique) {
        println("Coup critique")
        resultat = type.activationCritique * this.type.multiplicateurCritique
    }
    return resultat + this.qualite.bonusRarete
}

```

#### NOTE

A savoir que le resultat obtenu peut être différent en fonction de l'arme choisi, car les armes n'ont pas les mêmes bonus de rareté.

## 2.2 Classe Armure & TypeArmure

Nous réalisons les classes et TypeArmure Armure.

La classe TypeArmure est représentée de la manière suivante ci-dessous. La classe a comme attributs nom de type String et bonusType de type Int.

```

class TypeArmure(
    val nom: String,
    val bonusType: Int
)

```

La classe Armure ci-dessous hérite de la classe mère Item. La classe Armure a comme propriétés nom et description de type String, qualité représente la classe Qualité et typeArmure représente la classe TypeArmure.

```
class Armure(  
    nom: String,  
    description: String,  
    val qualite: Qualite,  
    val typeArmure: TypeArmure  
) : Item(nom, description) {
```

La méthode calculProtection calcule la protection en additionnant la propriété bonusType de l'objet typeArmure et bonusRarete de l'objet qualite.

```
/**@author Adrien  
 * @return type + rareté  
 * Méthode "calculProtection" pour calculer la protection de l'armure  
 */  
fun calculProtection(): Int {  
    var additionProtection = this.typeArmure.bonusType + this.qualite.bonusRarete  
    return additionProtection  
}
```

## 2.3 Classe Potion & Bombe

Nous avons créé les classes **Bombe** et **Potion** et une méthode **utiliser** pour permettre aux personnages de provoquer des dégâts en utilisant des **bombes** et de restaurer des points de vie en consommant des **potions** dans le jeu."

```
class Bombe(  
    val nombreDeDes: Int,  
    val maxDes: Int,  
    nom: String,  
    description: String  
) : Item(nom, description) {  
}
```

```
class Potion(  
    val soins: Int,  
    nom: String,  
    description: String  
) : Item(nom, description) {  
}
```

Remarque : le nom et la description sont des paramètres hérités par la classe mère **Item** afin

d'éviter les répétitions."

La méthode **utiliser** de la classe **Bombe** récupère la cible en paramètre et inflige des dégâts à cette cible en simulant les dégâts par la somme des dés et des faces.

```
override fun utiliser(cible: Personnage) {
    var tirageDes = TirageDes(this.nombreDeDes, this.maxDes)
    var resultat = tirageDes.lance()
    resultat -= cible.calculDefense()
    if (resultat < 1) {
        resultat = 1
    }

    // utiliser la protection de la cible
    cible.pointDeVie = cible.pointDeVie - resultat
    print("$resultat")
}
```

#### NOTE

On crée un objet **TirageDes()**, puis on utilise la méthode **lance** de la classe pour effectuer la somme des dés et des faces, ce qui nous permet de calculer les dégâts infligés à la cible.

REMARQUE : Si le résultat final est inférieur à 1, nous le réglons à 1 pour garantir qu'il y ait au moins 1 point de dégâts ou plus à infliger à la cible.

### 3.0 : Comment faire un test unitaire

Les tests unitaires permettent de vérifier que le code d'une fonction fonctionne correctement. Pour réaliser un test unitaire il faut réaliser les étapes suivantes :

- Faire un clic droit sur la méthode choisie
- Puis cliquer sur "Generate"
- Et enfin, sur "Test..."

#### 3.1 : Faire un test unitaire pour calculerDegats()

#### 3.2 : Faire un test unitaire pour calculProtection()

Le test unitaire vérifie si la méthode calculProtection() de la classe Armure retourne bien la valeur attendue "1". Si le test est réussi alors la méthode fonctionne.

```
class ArmureTest {
    @Test
    fun calculProtection() {
        //creation d'un objet armure de type Armure
    }
}
```



```

    val armure = Armure("", "", qualiteCommun, typeArmure1)
    val result = armure.calculProtection()
    Assertions.assertEquals(1, result)
}
}

```

#### NOTE

Pour vérifier le résultat attendu, nous utilisons la méthode "assertEquals". La première propriété récupère la valeur attendue, et la deuxième propriété attend le résultat de la méthode **calculProtection()** de l'objet armure que nous venons de créer.

### 3.3 : Faire un test unitaire pour la méthode utiliser() de la classe Bombe

#### Intermission (Creation de la classe Item)

L'intermission 3 était de créer une classe **Item**, pour faire hériter les propriétés 'nom', 'description' afin d'éviter les répétitions

```

abstract class Item(val nom: String, val description: String) {

    /**
     * @author Adrien
     * @param Personnage
     * Méthode pour faire hériter la méthode 'utiliser' de la classe mère aux classes
     filles.
     */
    open fun utiliser(cible: Personnage) {
        println("$nom ne peut pas être utilisé")
    }

    /**
     * @author
     * @param
     * @return
     * Méthode pour ...
     */
    override fun toString(): String {
        return "${nom} (nom='$nom', description='$description')"
    }
}

```

#### NOTE

**abstract** permet d'éviter la création d'objets de la classe **Item**

### 4 : Faire hériter les classes Arme, Armure, Potion et Bombe de la classe Item

Les classes suivantes héritent des propriétés nom et description de la classe Item.

```

class Arme(

```

```

    nom: String,
    description: String,
    val type: TypeArme,
    val qualite: Qualite
) : Item(nom, description) {

```

```

class Armure(
    nom: String,
    description: String,
    val qualite: Qualite,
    val typeArmure: TypeArmure
) : Item(nom, description) {

```

```

class Potion(
    val soins: Int,
    nom: String,
    description: String
) : Item(nom, description) {

```

```

class Bombe(
    val nombreDeDes: Int,
    val maxDes: Int,
    nom: String,
    description: String
) : Item(nom, description) {

```

## IMPORTANT

Les propriétés nom et description des classes filles sont en parametre. Les classes filles héritent de la classe mère Item.

## 5.1 : méthodes attaque() et equipe()

La méthode attaque() consiste dans un premier temps à verifier qui si le personnage à une arme équipée. Si c'est le cas les dégats seront augmenté en fonction de l'arme. Ensuite de déduire les dégats en fonction de la défense adverse et de ses points de vie.

```

open fun attaque(adversaire: Personnage) {
    var degats = this.attaque / 2
    if (armePrincipale != null) {
        degats += this.armePrincipale!!.calculerDegats()
    }
    degats -= adversaire.calculerDefense()
    if (degats <= 1) {
        degats = 1
    }
    adversaire.pointDeVie -= degats
}

```

```
println("$nom attaque ${adversaire.nom} avec une attaque de base et inflige  
$degats points de dégâts.")  
}
```

#### NOTE

1. Les dégats de base sont toujours divisé par 2 car sinon les dégats du personnage serait trop fort.
2. Les dégats infligés en fonction de la défense adverse sont toujours égale à 1, pour éviter d'être en négatif.

Il existe plusieurs versions de la méthode équipé, la première méthode équipé consiste à parcourir l'inventaire et d'équiper une arme en arme principale si c'est vrai.

```
open fun equipe(uneArme: Arme) {  
    if (uneArme in inventaire) {  
        armePrincipale = uneArme  
        println("$nom équipe « ${uneArme.nom} ».")  
    }  
}
```

## 5.2 : méthodes calculTotalDefense() et equipe()

La méthode equipe() permet de vérifier si une armure est présente dans l'inventaire et si c'est le cas elle équipe cette armure et affiche le nom de l'armure équipée.

```
fun equipe(uneArmure: Armure) {  
    if (uneArmure in inventaire) {  
        this.armure = uneArmure  
        println("${this.nom} equipe ${uneArmure.nom}")  
    }  
}
```

La méthode calculeDefense calcule la défense d'un personnage en prenant la moitié de sa valeur de base. Si le personnage a une armure, on ajoute le bonus de l'armure à la défense et retourne le résultat de cette addition.

```
fun calculeDefense(): Int {  
    var result = this.defense / 2  
    if (this.armure != null) {  
        result = result + this.armure!!.calculProtection()  
    }  
    return result;  
}
```

### 5.3 : méthodes **avoirPotion()**, **avoirBombe()**, **boirePotion()**

Les méthodes **avoirPotion()** et **avoirBombe()** retournent **Vrai** seulement si la personne possède au moins un de ces items dans son inventaire.

```
fun avoirPotion(): Boolean {
    var result: Boolean = false
    for (item in inventaire) {
        if (item is Potion) {
            result = true
        }
    }
    return result
}
```

```
fun avoirBombe(): Boolean {

    var result: Boolean = false

    for (item in inventaire) {
        if (item is Bombe) {
            result = true
        }
    }
    return result
}
```

La méthode **boirePotion()** permet à un personnage de boire une potion pour restaurer ses points de vie. Elle accepte une potion en argument ou recherche une dans l'inventaire du personnage. Une fois la potion trouvée, elle la retire de l'inventaire du personnage et restaure ses points de vie.

```
fun boirePotion(unePotion: Potion? = null) {
    var soins: Int = 0
    var nomSoins: String? = null//="BLA"
    var pointDeVieMax = this.pointDeVieMax

    if (unePotion == null) {
        for (item in inventaire) {
            if (item is Potion) {
                soins = item.soins
                nomSoins = item.nom
                inventaire.remove(item)
                break
            }
        }
    } else {
        soins = unePotion.soins
        nomSoins = unePotion.nom
    }
}
```

```

        inventaire.remove(unePotion)
    }
    if (this.pointDeVie + soins >= pointDeVieMax) {
        soins = this.pointDeVieMax - this.pointDeVie
        this.pointDeVie = this.pointDeVieMax
    } else {
        this.pointDeVie += soins
    }
    println("$nomSoins a augmenté de $soins PV")
}

```

REMARQUE : Si le montant de soins que la personne va recevoir est supérieur à ses points de vie maximum, alors le montant de soins sera réglé sur les points de vie maximum du personnage.

## 6.1 : méthodes afficheInventaire() et loot()

Nous avons créer une méthode afficheInventaire qui permet d'afficher chaque item avec son index. Pour cela nous avons utilisé la boucle for pour parcourir l'inventaire et afficher l'index de chaque item. Et la condition do while pour choisir un item en fonction de la liste en index de l'inventaire.

```

fun afficheInventaire(): Int {
    println("Inventaire $nom")
    val size = inventaire.size
    for (i in 0..size - 1) {
        val item = inventaire[i]
        println("$i => ${item.nom}")
    }
    println("choisir un item : ")
    var option: Int
    do {
        option = readln().toInt()
    } while (option <= inventaire.size - 1 && option >= 0)
    return option
}

```

## 6.2 : modification de la méthode tourJoueur()

Nous modifions la méthode tourJoueur() en saisissant la code suivant : "2" → `this.jeu.joueur.boirePotion()`. La ligne de code exécute la méthode boirePotion() sur le joueur associé à un objet jeu.

## 6.3 : modification de la méthode tourMonstre()

## Intermission 6 : Ajout de la méthode `tourJoueur()` dans la classe `Combat`

Le code ci dessous affiche l'inventaire du joueur, et choisit sa position en fonction de sa clé. Si l'objet est une bombe, l'objet choisit est une bombe, la méthode `utiliser()` (méthode définie uniquement pour la bombe) permettra d'attaquer le monstre. Si l'objet est une potion, alors, l'objet sera utiliser sur le joueur pour regagner des points de vie.

```
"3" -> {  
    val posObjet = this.jeu.joueur.afficheInventaire()  
    val objet = this.jeu.joueur.inventaire[posObjet]  
  
    if (objet is Bombe) {  
        objet.utiliser(monstre)  
    } else {  
        objet.utiliser(this.jeu.joueur)  
    }  
}
```

## Sprint 2 : Gestion du choix de la classe (Guerrier, Voleur, Mage)

### 7.1 : Création de la classe `Guerrier`

Nous avons créer la classe `Guerrier` avec les attributs issu du diagrammes relié à l'héritage `Personnages`.

```
class Guerrier(  
    nom: String,  
    pointDeVie: Int,  
    pointDeVieMax: Int,  
    attaque: Int,  
    defense: Int,  
    endurance: Int,  
    vitesse: Int,  
    inventaire: MutableList<Item> = mutableListOf(),  
    armePrincipale: Arme?,  
    var armeSecondaire: Arme?,  
    armure: Armure?  
) : Personnage(  
    nom,  
    pointDeVie,  
    pointDeVieMax,  
    attaque,  
    defense,  
    endurance,  
    vitesse,  
    inventaire,  
    armePrincipale,
```

```
    armure  
)
```

**NOTE** La classe Guerrier possède une arme secondaire en plus de l'arme principale.

## Mission 7.2 : Création de la classe Voleur

Nous réalisons la classe Voleur. La classe Voleur est une classe fille qui hérite des propriétés de la classe Personnage. Nous pouvons affirmer l'héritage avec la notation suivante : `: Personnage(...)`

```
class Voleur(  
    nom: String,  
    pointDeVie: Int,  
    pointDeVieMax: Int,  
    attaque: Int,  
    defense: Int,  
    endurance: Int,  
    vitesse: Int,  
    inventaire: MutableList<Item> = mutableListOf(),  
    armePrincipale: Arme?,  
    armure: Armure?  
) : Personnage(  
    nom,  
    pointDeVie,  
    pointDeVieMax,  
    attaque,  
    defense,  
    endurance,  
    vitesse,  
    inventaire,  
    armePrincipale,  
    armure  
) {
```

## Mission 7.3 : Création de la classe Mage et Sort

La classe **Mage** est une sous-classe de **Personnage** qui hérite de toutes les propriétés de **Personnage**, à l'exception de la propriété **grimoire** qui contiendra une liste de sorts spécifiques au mage.

```
class Mage(  
    nom: String,  
    pointDeVie: Int,  
    pointDeVieMax: Int,  
    attaque: Int,  
    defense: Int,  
    endurance: Int,
```

```

    vitesse: Int,
    inventaire: MutableList<Item> = mutableListOf(),
    armePrincipale: Arme?,
    armure: Armure?,
    var grimoire: MutableList<Sort> = mutableListOf()

) : Personnage(
    nom, pointDeVie, pointDeVieMax, attaque,
    defense, endurance, vitesse, inventaire, armePrincipale, armure
) {

}

```

La classe "Sort" représente un sort magique avec un nom et un effet défini sous forme d'une lambda prenant deux personnages en tant qu'arguments.

```

class Sort(
    val nom: String,
    val effect: (Personnage, Personnage) -> Unit,
) {

}

```

#### NOTE

La propriété "effect" dans la classe "Sort" est une fonction anonyme qui définit comment le sort affecte les personnages. Cela permet de déterminer le comportement précis du sort lorsqu'il est utilisé dans le jeu.

## Intermission 7 :

### A COMPLETER

#### 8.1 : Méthodes toString() equipe() attaque()

Nous avons créer la dernière version de la méthode équipé pour définir l'emplacement de l'arme. Pour cela, nous allons utiliser when une instruction qui permet d'exécuter un programme lorsqu'une condition est remplie. Et l'instruction if pour parcourir l'inventaire et trouver une arme.

```

override fun equipe(uneArme: Arme) {
    println("Choisir l'emplacement de l'arme : 0 -> armePrincipale ; 1 ->
    armeSecondaire")
    val emplacementArme = readln().toString()
    when (emplacementArme) {
        "0" -> {
            super.equipe(uneArme)
            println("L'arme est en arme principale")
        }
        "1" -> {
            if (uneArme in inventaire) {

```



```

        armeSecondaire = uneArme
        println("L'arme est en arme secondaire")
    }
}
}

```

#### NOTE

Si le joueur décide d'équiper l'arme en arme principale c'est à dire 0, la méthode équipe de l'arme principale est réutiliser.

Nous avons redéfinir la méthode attaque dans le cas ou le joueur choisit la classe Guerrier et donc possède une arme secondaire. Si le personnage possède une arme secondaire, calcule les dégats du personnage en fonction de l'arme secondaire (même fonctionnement que la méthode attaque pour l'arme principale).

```

override fun attaque(adversaire: Personnage) {
    var degats = this.attaque / 2
    super.attaque(adversaire)
    if (armeSecondaire != null) {
        degats += this.armeSecondaire!!.calculerDegats()
    }
    degats = degats - adversaire.calculerDefense()
    if (degats <= 1) {
        degats = 1
    }
    println("$nom attaque ${adversaire.nom} avec une attaque de base et inflige $degats points de dégâts.")
}
}

```

#### NOTE

Identique à la méthode attaque de l'arme principale.

1. Les degats sont toujours divisé par 2.
2. Les degats sont toujours égale à 1 en fonction de la défense adverse.

## 8.2 : Méthode toString() et voler()

La méthode volerItem() permet de voler un objet dans l'inventaire d'un personnage.

```

fun volerItem(cible: Personnage) {
    if (cible.inventaire.isNotEmpty()) {
        var positionObjet = (1..cible.inventaire.size).random()
        var objet = cible.inventaire[positionObjet]

        if (objet == cible.armePrincipale) {
            cible.inventaire.remove(objet)
            cible.armePrincipale = null
            this.inventaire.add(cible.inventaire[positionObjet])
        }
    }
}

```

```

    }
    if (objet == cible.armure) {
        cible.inventaire.remove(objet)
        cible.armure = null
        this.inventaire.add(cible.inventaire[positionObjet])
    }
    if (objet is Bombe) {
        cible.inventaire.remove(objet)
        this.inventaire.add(cible.inventaire[positionObjet])
    }
    if (objet is Potion) {
        cible.inventaire.remove(objet)
        this.inventaire.add(cible.inventaire[positionObjet])
    }
    println("L'objet ${objet.nom} a été volé et a été ajouté dans
l'inventaire")
} else
    println("L'inventaire de la cible est vide")

```

#### NOTE

La méthode pourrait être améliorée et proposée de la manière suivant ci-dessous, or par soucis de compréhension et de logique et des difficultés rencontrées j'ai préféré m'en tenir à cette rédaction proposé ci dessus.

```

this.inventaire.add(objet)
cible.inventaire.remove(objet)

    if (objet==cible.armePrincipale ) {
        cible.armePrincipale = null
    }
    if (objet==cible.armure) {
        cible.armure=null
    }
    this.inventaire.add(cible.inventaire[positionObjet])
}

```

### 8.3 : Méthode toString(), afficherGrimoire() et choisirEtLancerSort()

La méthode 'afficheGrimoire' nous permet d'afficher les sorts et leur index appartenant au mage qui sont stockés dans la liste du grimoire.ore

```

fun afficheGrimoire() {

    for (i in 0..grimoire.size - 1) {
        println("$i => ${grimoire[i].nom}")
    }
}

```

La méthode 'choisirEtLancerSort', qui prend en paramètre l'adversaire, permet au joueur d'un personnage (Mage) de choisir un sort depuis le grimoire du personnage. Pour ce faire, elle utilise la méthode précédente 'afficheGrimoire' pour afficher la liste des sorts. Ensuite, elle demande aux joueurs de choisir un sort et la cible sur laquelle le sort sera utilisé.

```
fun choisirEtLancerSort(adversaire: Personnage) {
    val affiche = afficheGrimoire()

    println("Choisir un sort a lancer (entrez le numéro)")
    var index: Int

    do {
        index = readln().toInt()
    } while (index < 0 || index > grimoire.size - 1)

    println("choisir votre cible : [0] => vous-même ou [1] => Adversaire")
    var laCible: Personnage = this;
    var cible = readln().toInt()

    when (cible) {
        0 -> laCible = this
        1 -> laCible = adversaire
        else -> print("Erreur")
    }

    val utiliser = this.grimoire[index].effect(this, laCible)
}
```

REMARQUE : 'this' désigne le joueur

## Intermission 8 :

A COMPLETER

### 9.1 : Création de sorts : Boule de feu, Missile magique

Le sort boule de feu consiste à un lancement de des. Et en fonction du lancement de des, calcule le dégats du sort en fonction des degats du personnage, de la défense total adverse et déduire de ses points de vie.

```
val sortBouleDeFeu = Sort("Boule de feu") { caster, cible ->
    run {
        val degatCaster = caster.attaque / 3
        val tirageDes = TirageDes(1, 6)
```

```

        var degat = tirageDes.lance()
        degat += degatCaster
        degat -= cible.calculDefense()
        cible.pointDeVie -= degat
        println("${caster.nom} lance une « Boule de feu » et inflige $degat de
        dégat(s) à ${cible.nom}.")
    }
}

```

**NOTE** L'attaque est toujours divisé par 3 afin de ne pas rendre le sort trop fort.

Le sort missile magique à le même fonctionnement que le sort boule de feu, mais avec l'ajout d'un compteur qui indique que si les dégats du caster est supérieur au compteur, calcule les dégats du sort en fonction du lancement de dès et des dégats du personnage. Et ajouter +1 au compteur à chaque boucle.

```

val missileMagique = Sort("Missile magique") { caster, cible ->
    run {
        var compteur = 0
        var degatCaster = caster.attaque / 2
        val tirageDes = TirageDes(1, 6)
        if (compteur < degatCaster) {
            var degat = tirageDes.lance()
            degat -= cible.calculDefense()
            if (degat <= 1) {
                degat = 1
            }
            cible.pointDeVie -= degat
            println("Le « Projectile Magique » inflige $degat de dégat(s) à
            ${cible.nom}.")
            compteur + 1
        }
    }
}

```

**NOTE** Identique au sort boule de feu, l'attaque est tout le temps divisé par 2.

## 9.2 : Création de sorts : Invocation d'une arme magique, Invocation d'une armure magique

instanciation de l'Arme Magique, permet d'invoquer une arme avec une qualité qui est choisie aléatoirement en fonction du tirage dès

```

val invocationArmeMagique = Sort("Invocation Arme Magique") { caster, cible ->
    run {
        val tirageDes = TirageDes(1, 20)
        val rarete = tirageDes.lance()
    }
}

```

```

var qualite: Qualite? = null
when {
  rarete < 5 -> qualite = qualiteCommun
  rarete < 10 -> qualite = qualiteRare
  rarete < 15 -> qualite = qualiteEpic
  else -> qualite = qualiteLegendaire
}
val armeMagique = Arme("Arme Magique", "Blabla c'est trop bien", epeeLongue,
qualite!!)
caster.inventaire.add(armeMagique)
caster.equipe(armeMagique)
println("Une « Arme Magique » a été ajoutée à l'inventaire.")
}
}

```

instanciation l'Armure Magique, permet d'invoquer une armure avec une qualité qui est choisie aléatoirement en fonction du tirage dès

```

val invocationArmureMagique = Sort("Invocation Armure Magique") { caster, cible ->
run {
  val tirageDes = TirageDes(1, 20)
  val rarete = tirageDes.lance()
  var qualite: Qualite? = null
  when {
    rarete < 5 -> qualite = qualiteCommun
    rarete < 10 -> qualite = qualiteRare
    rarete < 15 -> qualite = qualiteEpic
    else -> qualite = qualiteLegendaire
  }
  val armureMagique = Armure("Armure magique", "BlablaBla...", qualite!!, cuir)
  caster.inventaire.add(armureMagique)
  caster.equipe(armureMagique)
  println("Une armure magique est ajoutée à l'inventaire")
}
}

```

### 9.3 : Création de sort : Sort de soins

On crée un objet 'sortDeSoins' de la classe 'Sort' pour restaurer les points de vie du joueur (Mage). Cela se fait en calculant les points de vie restaurés grâce à un lancer de dé à 6 faces et en ajoutant la moitié de ses points d'attaque.

```

val sortDeSoins = Sort("Sort de soins") { caster, cible ->
  run {
    val tirageDes = TirageDes(1, 6)
    var degat = tirageDes.lance() + (caster.attaque / 2)

    val pv = caster.pointDeVieMax - caster.pointDeVie

```

```
        if (degat > pv) {  
            degat = pv  
        }  
        caster.pointDeVie += degat  
        println("${caster.nom} a utilisé un « Sort de soins » et a récupéré $degat  
point(s) de vie !")  
    }  
}
```

REMARQUE : On vérifie toujours si les points de vie à restaurer dépassent les points de vie maximum. Si c'est le cas, on fixe les points de vie au maximum pour éviter les erreurs.

## **Intermission 9 :**

### **10.1 : Faire des tests unitaires les sorts de boule de feu et Missile magique**

### **10.2 : Faire des tests unitaires les sorts d'invocations**

### **10.3 : Faire des tests unitaires pour le sort de soins**