

# Programmation Orientée Objet (OBJET)

TP 3 : Début du projet « World of ECN »

Ajout de nouvelles classes et modifications des classes  
existantes

Jean-Marie Normand – Bureau A114

[jean-marie.normand@ec-nantes.fr](mailto:jean-marie.normand@ec-nantes.fr)



# Instructions

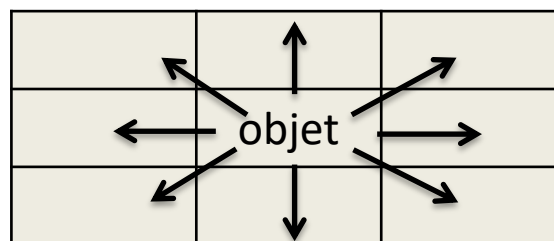
- Suivez les slides les uns après les autres
- À la fin de cette séance de TP, **vous devrez nous rendre un rapport par binôme**
- Ce rapport devra contenir :
  - Une introduction et une présentation rapide du sujet
  - Les réponses aux questions posées dans les slides repérés par une icône de panneau **STOP** de la séance 3
  - Une conclusion
- La notation tiendra compte du respect de ces consignes



# **1<sup>RE</sup> PARTIE : MISE À JOUR DE WOE !**

# Déplacement des Protagonistes

- Avant d'aller plus loin !
- Implémentez la méthode `deplace()` des classes `Personnage` et `Monstre`
- Ces méthodes doivent permettre de déplacer aléatoirement un objet d'une de ces classes (ou de leurs sous-classes) sur une case adjacente de là où il se trouve
- Vous n'avez pas à vérifier si la case est libre ou non !



→ déplacement autorisé



# Déplacement des Protagonistes

- **Illustrez** le fonctionnement de ces méthodes en déplaçant les attributs `robin`, `peon`, `bugs` de la classe `World`
- **Expliquez** dans votre rapport dans quel(s) cas il serait judicieux de surcharger la méthode `deplace()` dans une sous-classe de `Personnage` et/ou de `Monstre`
- **Expliquez** comment il serait possible d'interdire aux objets le déplacement sur une case déjà occupée (vous n'avez pas à implémenter cette fonctionnalité pour le moment)

# Javadoc

- La documentation en Java est primordiale pour pouvoir partager son code et son travail avec une équipe
- Java offre le mécanisme de la **Javadoc** qui permet de générer simplement la documentation d'un projet au format HTML
- La Javadoc utilise une syntaxe particulière et un ensemble de « **tags** » (débutant par un **@**) pour spécifier des informations spéciales (p. ex. l'auteur d'un fichier, les paramètres d'une méthode, le résultat d'une méthode, etc.)
- Certains de ces tags sont les suivants :
  - **@author** : identifie l'auteur du fichier
  - **@param** : permet de décrire un paramètre d'une méthode
  - **@return** : permet de décrire le résultat d'une méthode
  - Etc.
- Voir les exemples des classes présentées en cours et <https://fr.wikipedia.org/wiki/Javadoc>



# Écriture et génération Javadoc

- **Écrivez** la javadoc pour les classes : `Personnage`, `Monstre`, `World`
  - Tous les attributs
  - Toutes les méthodes publiques
  - Essayez avec une méthode privée si vous en avez une
  - NB : Vous pouvez utiliser la commande “Tools → Analyze Javadoc” pour vérifier ou prédéfinir la description des méthodes
- **Générez** la Javadoc (menu “Run → Generate Javadoc”) :
  - Attention aux erreurs éventuelles
  - Corrigez les !
- **Illustrez** dans votre rapport le résultat de la génération de la Javadoc (captures d’écran de la documentation générée par NetBeans)



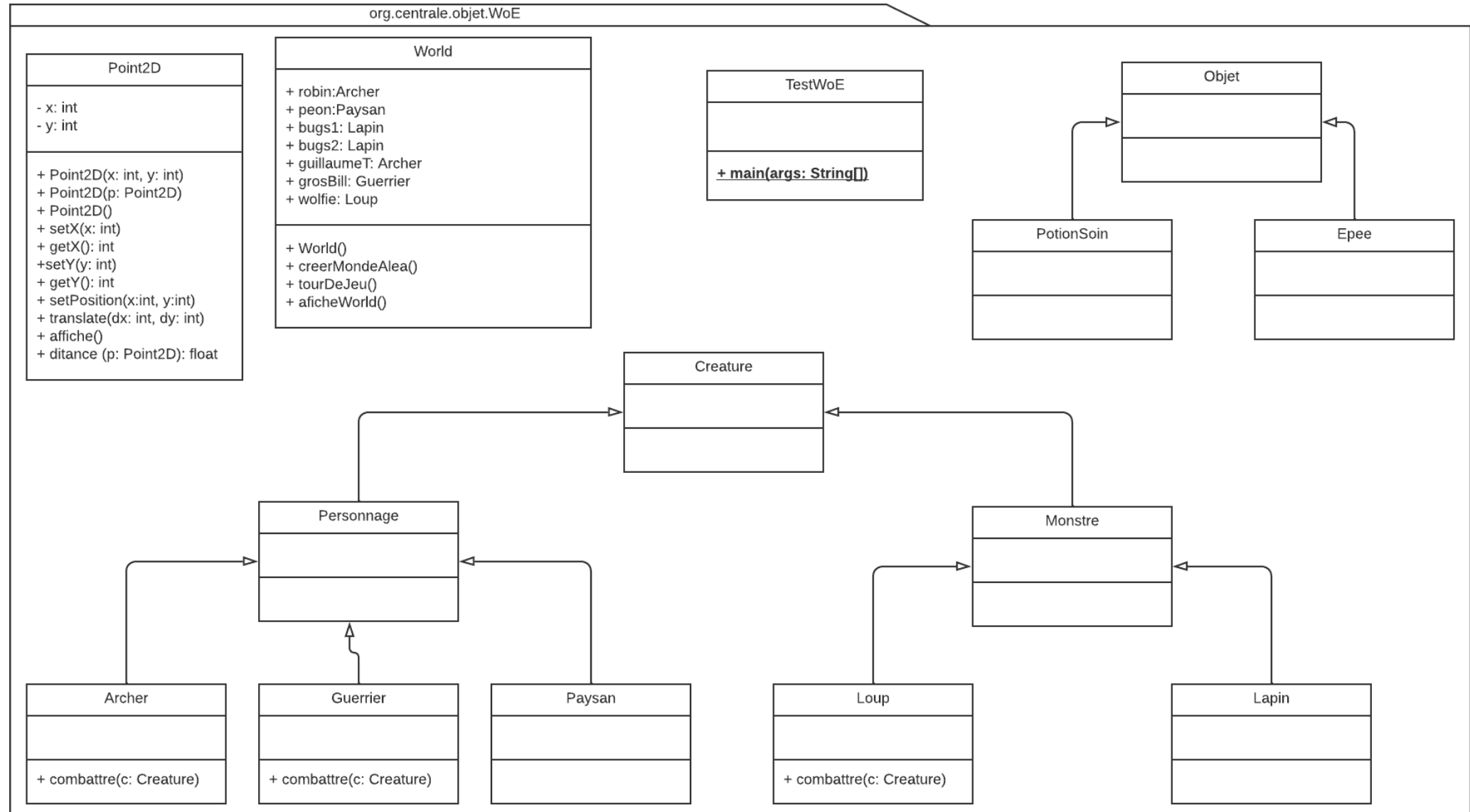
# Copie d'objet

- **Ajoutez** un nouvel attribut de type Archer dans la classe `World` dont le nom est « `guillaumeT` »
- **Copiez** `robin` dans `guillaumeT`
- **Déplacez** `robin` afin que sa position 2D soit modifiée
- **Affichez** les deux objets de la classe `Archer` : ont-ils bougé tous les 2 (i.e. les positions de `robin` et de `guillaumeT` sont-elles les mêmes) ?
  - Si oui :
    - Cherchez et expliquez pourquoi
    - Modifiez votre code de manière à corriger ce problème
  - Si non :
    - Expliquez pourquoi
- **Veillez** à bien répondre à ces questions et à inclure les explications et justifications (et modifications éventuelles) dans votre rapport



# Mise à jour du diagramme de classe UML

- En vous basant sur le diagramme UML suivant (disponible en version plus lisible dans le fichier [WoE-TP3.png](#)) :



# World of ECN – WoE

- Le diagramme de classes représente une **évolution de WoE** (néanmoins toujours incomplet pour l'instant)
- Nous avons **ajouté** de nouvelles classes (**Objet**, **PotionSoin**, **Epée**, **etc.**)
- Nous avons **modifié** les classes existantes (**Personnage**, **World**, **Archer**, **Point2D**)
- Les éléments UML sont maintenant **volontairement « vides ou presque » → à vous d'y mettre ce que vous y jugez bon !!! (et de noter ces choix dans votre rapport)**
- Toutefois si nous mettons une méthode ou un attribut dans une classe **veillez à le respecter**

# Détails de quelques nouveautés

- La plupart des classes sont explicites et ne nécessitent pas de détails particuliers (cf. Séance de TP 1)
- **Guerrier**, **Loup** représentent de nouveaux types de protagonistes possibles **dans cette nouvelle version de WoE**
- **Creature** est une **nouvelle classe**



# Distance entre les protagonistes

- Dans la séance précédente nous avons tenté de limiter la distance entre les objets de la classe World
- Pour tenter de simplifier le procédé, nous nous proposons d'ajouter une méthode **distance** à la classe Point2D :
  - **Proposez** l'entête de cette méthode
  - **Implémentez** cette méthode
  - **Détaillez** cette implémentation dans votre rapport
- Utilisation des méthodes mathématiques de Java :
  - Les méthodes mathématiques de Java font partie du paquetage `java.lang` et sont statiques (voir CM pour rappel sur les méthodes statiques) !
  - Ce paquetage est inclus par défaut en java, il est donc inutile d'écrire `import java.lang.Math;` en début de fichier !
  - Un exemple de calcul de racine carrée :
    - `double rc = Math.sqrt(5.8); // racine carrée de 5.8`
  - Plus d'informations : <http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html>



## Détails de quelques nouveautés (2)

- **World** : devra respecter les mêmes contraintes que lors de la séance précédente mais :
  - **Ajoutez** de nouveaux attributs (voir le diagramme) pour représenter les nouveaux types possibles de protagonistes de WoE
  - **Modifiez** la méthode **creerMondeAlea** pour prendre en compte ces modifications
  - **Explicitez** ces modifications dans votre rapport
- Mettez à jour la classe **TestWoE** par rapport aux nouvelles classes etc. ajoutées à cette séance



# Diagramme de classe UML

- A partir de ce diagramme UML (disponible en version lisible dans un fichier à part) :
  - **Expliquez et justifiez** les choix que vous avez faits pour les nouvelles classes :
    - Quels attributs ?
    - Quelles méthodes ? (Laissez de côté la méthode `combattre` pour le moment)
    - Pourquoi avez-vous fait ces choix ?
    - **NB : il est tout à fait possible que ces choix vous amènent à modifier les classes écrites précédemment !**
  - **Implémentez** les classes décrites dans le diagramme en respectant :
    - Les **noms** des classes
    - Les **noms**, **droits d'accès** et **types** des attributs
    - Les **noms**, **portées** et **arguments** des méthodes

# Gestion du système de combat

- Nous souhaitons maintenant implémenter le système de combat de notre jeu
- Ce système dépend du type de combat :
  - **Contact** (distance entre les protagonistes = 1)
  - **À distance** (distance entre les protagonistes  $> 1$  et  $< \text{distAttMax}$ , cf. classe **Personnage**)
- Chaque type de combat a une règle détaillée dans la suite qui est basée sur des **jets aléatoires** et sur les **pourcentages d'attaque** et de **parade** des objets de type **Personnage**

# Combat au corps à corps

- Règles du combat :
  - L'attaquant fait un jet de dé pour voir si il va réussir son attaque ou non (les détails vont suivre)
  - Si l'attaquant réussit son jet, alors le défenseur va tenter d'atténuer les dégâts reçus en faisant lui aussi un jet de dé
- Rappel : ces règles s'appliquent uniquement si un combat au corps à corps a lieu (les deux protagonistes sont sur des cases adjacentes)



# Combat au corps à corps (2)

- Règles du combat :

- Attaquant :

- Rand = tirage aléatoire (entre 1 et 100)
    - Si  $\text{Rand} > \text{pourcentageAtt}$  → attaque **ratée**
    - Si  $\text{Rand} \leq \text{pourcentageAtt}$  → attaque **réussie**

- Défenseur (seulement si l'attaque a réussi) :

- Rand = tirage aléatoire (entre 1 et 100)
    - Si  $\text{Rand} > \text{pourcentagePar}$  → dégâts subis =  $\text{degAtt}$  du Personnage attaquant
    - Si  $\text{Rand} \leq \text{pourcentagePar}$  → dégâts subis =  $\text{degAtt}$  (de l'attaquant) –  $\text{ptPar}$  (points de parade du défenseur)

# Combat au corps à corps (3)

- Précisions :
  - Dans un jeu de rôle les **caractéristiques** des **Personnage** représentent **leur capacité à réussir quelque chose** !
  - Ainsi **les jets de dés sont réussis** si ils sont **inférieurs au pourcentage** de la caractéristique du **Personnage**
  - Exemple pour un **Guerrier** possédant **80** en **ptAtt** et tentant une attaque :
    - un jet de dé **entre 1 et 80** est une réussite
    - un jet de dé **entre 81 et 100** est un échec

# Combat à distance

- Rappel : ces règles s'appliquent uniquement si un combat à distance à lieu (distance  $> 1$  et  $< \text{distAttMax}$  de l'attaquant)
- Règles du combat :
  - L'attaquant fait un jet de dé pour voir si il va réussir son attaque ou non (les détails vont suivre)
  - Quel que soit le résultat du jet d'attaque, un projectile est retiré de l'attaquant (p. ex. une flèche pour un archer)
  - Si l'attaquant réussit son jet, alors le défenseur va encaisser des dégâts

# Combat à distance (2)

- Règles du combat :
  - Attaquant :
    - Rand = tirage aléatoire (entre 1 et 100)
    - Si Rand > pourcentageAtt → attaque **ratée**
    - Si Rand <= pourcentageAtt → attaque **réussie**
  - Défenseur (seulement si l'attaque a réussi) :
    - dégâts subis = degAtt du Personnage attaquant



# Système de combat (1)

- **Implémentez** le système de combat tel que décrit dans les slides précédents (c'est-à-dire les méthodes `void combattre(Creature c){...}`)
- **Modifiez** votre classe `World` afin d'illustrez le bon fonctionnement de vos méthodes `combattre`
- **Veillez à bien illustrer dans votre rapport** le bon fonctionnement des différents types de combat (voir suite)



## Système de combat (2)

- **Créez** différents types de **Personnage** avec des caractéristiques différentes afin de vous permettre d'illustrer le système de combat
- Positionnez « **à la main** » les objets afin que les distances entre eux permettent d'illustrer les combats au corps à corps et à distance
- **Veillez à bien illustrer dans votre rapport** le bon fonctionnement des différents types de combat :
  - **Jets de dés** et **réussites/échecs** en fonction des caractéristiques des **Personnages**
  - **Mise à jour** des points de vie, nombre de flèches des **Personnages**



# Objets

- Nous voulons tester les classes `Objet` et ses sous-classes
- Dans la classe `World`, ajoutez quelques potions (ces dernières sont censées rendre des points de vie aux objets des classes/sous-classes de `Personnage`)
- **Illustrez** le déplacement d'un objet sur la case d'une `PotionSoin` et la **modification** associée à son nombre de points de vie
- Idéalement une fois la **potion consommée elle devrait disparaître**
- Avec ce que vous avez vu en cours du système de gestion de la mémoire en Java, **proposez un moyen de faire disparaître** les potions qui ont été consommées !



# Conclusion

- Ajoutez à votre rapport :
  - **L'illustration du bon fonctionnement** de votre fonction principale (sortie textuelle des tests effectués)
- Rendez une archive au format **.ZIP** dont le nom respectera la convention suivante **OBJET-TP3-NomBinome1-NomBinome2.zip** et contenant :
  - Votre rapport au format **.pdf** dont le nom respectera la convention suivante : **OBJET-TP3-NomBinome1-NomBinome2.pdf**
  - Avec **NomBinome1 < NomBinome2** dans l'ordre alphabétique !!!
  - Tous vos fichiers **.java**
  - **Veillez à bien avoir écrit la Javadoc** de tous les **attributs** de vos classes et des **principales méthodes** (déplacer, combattre, etc.)
  - **Faites générer la Javadoc** par NetBeans, **joignez** l'ensemble des fichiers résultats à **l'archive .zip dans un dossier documentation**
- Le respect de ces consignes est pris en compte dans la note !



