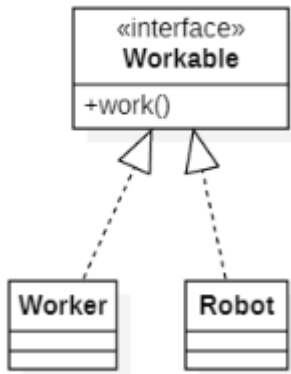
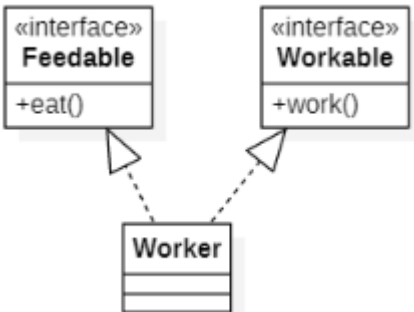


A la découverte des interfaces ...

Exercice 1 : Un petit bond vers la notion d'Interface

1. Qu'est-ce qu'une **interface** ?
2. De la conception à l'implémentation Java (à retenir ...)

Relation de réalisation << realize >> entre une interface et une classe (Conception)	Une classe réalise/implémente une interface avec le mot clé implements (Implémentation en Java)
	<pre>interface Workable { void work(); } public class Worker implements Workable { @Override public void work() { System.out.println("...working"); } } public class Robot implements Workable { @Override public void work() { System.out.println ("... working much more, 24 hours a day, 7 days a week"); } }</pre>

En java, l'héritage multiple de classes est interdit mais l'héritage multiple d'interfaces est autorisé 😊	
Héritage multiple d'interfaces (Conception)	Réalisation de plusieurs interfaces au sein d'une même classe mot clé implements et , (Implémentation en Java)
	<pre>interface Feedable { void eat(); } public class Worker implements Workable, Feedable { @Override public void work() { System.out.println("...working"); } @Override public void eat() { System.out.println("...eating in launch break"); } }</pre>

❑ **Quelques rappels UML** : Complétez le tableau suivant en vous aidant de la documentation précédente et de ce que vous avez appris lors des précédentes séances sur l'héritage 😊

En UML, comment s'appelle la relation d'héritage entre deux classes ?	En UML, comment s'appelle la relation qui va d'une classe vers une interface ?
Quelle est sa notation graphique ?	Quelle est sa notation graphique ?
Que pourrait-on écrire sur une telle relation ? (c-a-d que dit-on quand on <i>lit</i> une telle relation)	Que pourrait-on écrire sur une telle relation ? (c-a-d que dit-on quand on <i>lit</i> une telle relation)
Mot clé en java :	Mot clé en java :

3. Petit exemple pour bien rebondir sur la notion d'interface 😊

a. Analyse (présentation et identification des besoins) :

Nous souhaitons proposer un *contrat* qui permettra d'offrir à certains objets la capacité de rebondir. Une balle rebondira toujours, mais une balle de handball ne rebondira pas de la même manière qu'une balle de rugby. Toutes les balles ont une taille.

Un pneu sera également capable de rebondir, mais bien sûr beaucoup plus légèrement qu'une balle de handball. Il est bien sûr inutile de donner la capacité de rebondir à un terrain de sport, mais ce dernier a une longueur et une largeur.

Remarque : en anglais le terme **bounce** est utilisé pour **rebondir**. Comme nous sommes dans le contexte du sport, le terme **pitch** sera utilisé pour **terrain**.

En anglais, le terme **tyre** est utilisé pour pneu.

b. Conception :

Proposez une conception, au travers d'un **diagramme de classes**, qui permet de répondre aux besoins précédents. Votre diagramme de classes devra **forcément faire apparaître une interface**.

Bonne pratique : Une interface offre un(des) service(s) qui permette à l'objet d'**être capable de** faire quelque chose de nouveau (d'enrichir son comportement). C'est pourquoi, par convention, le nom des interfaces se terminent le plus souvent par le suffixe **-able** en anglais !

c. Implémentation

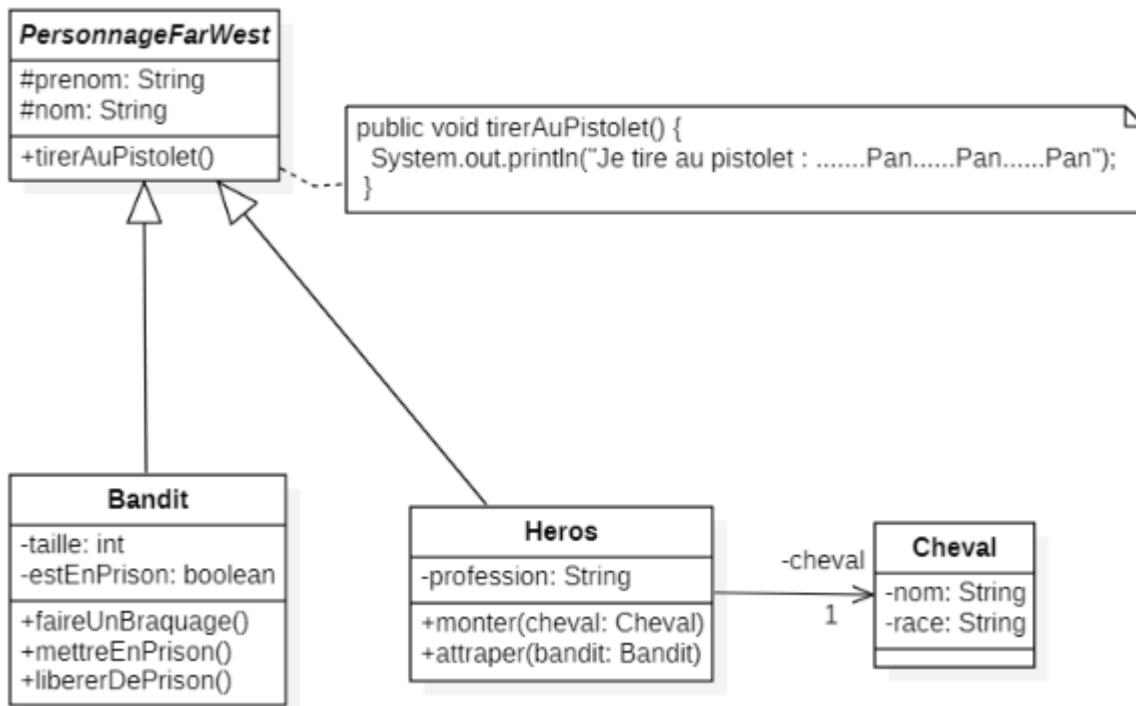
Implémentez en java le diagramme de classes précédent.

Exercice 2 : Le Far West à la recousse du polymorphisme et des interfaces ...

Pour vous familiariser avec la notion de polymorphisme et le concept d'Interface, vous allez retourner au Far West et vous focalisez sur la méthode **tirerAuPistolet** 😊

❑ Conception n°1 :

La conception actuelle de notre Far West est la suivante :



⇒ Sur ce diagramme de classes, on voit 3 relations entre classe.

Si vous deviez traduire ce diagramme de classes en 2 ou 3 phrases quelles seraient-elles ?

⇒ Pourquoi est-il pertinent de considérer la classe **PersonnageFarWest** comme une **classe abstraite** ? (car effectivement, remarquez bien que cette classe est écrite **en italique** dans le diagramme de classes ce qui indique qu'elle **est abstraite**) ?

Pour pouvoir tester cette conception, nous proposons le jeu d'essai que nous appellerons **jeu d'essai n°1** :

```
public static void main(String[] args) {  
  
    Cheval jollyJumper = new Cheval("Jolly Jumper", "appaloosa");  
    Heros luckyLuke = new Heros("Lucky", "Luke", "cow-boy", jollyJumper);  
    luckyLuke.tirerAuPistolet();  
  
    Bandit joeDalton = new Bandit("Joe", "Dalton", 150);  
    Bandit averellDalton = new Bandit("Averell", "Dalton", 190);  
    joeDalton.tirerAuPistolet();  
    averellDalton.tirerAuPistolet();  
}
```

⇒ Une fois la conception implémentée, quel affichage obtient-on sur la console à l'exécution du jeu d'essai n°1 ?

⇒ Tous les *objets* du Far West peuvent-ils tirer au pistolet ?

☐ Oui

☐ Non

Si non, quels objets ne sont pas capable de tirer au pistolet ?

⇒ Pensez-vous que :

☐ Tous les *objets* qui tirent au pistolet ont tous la même manière de tirer

☐ Chaque *objet* qui tire au pistolet a sa *propre* manière spécifique de tirer

☐ Un seul (ou plusieurs) objets a une manière spécifique de tirer, tous les autres tirent de la même manière

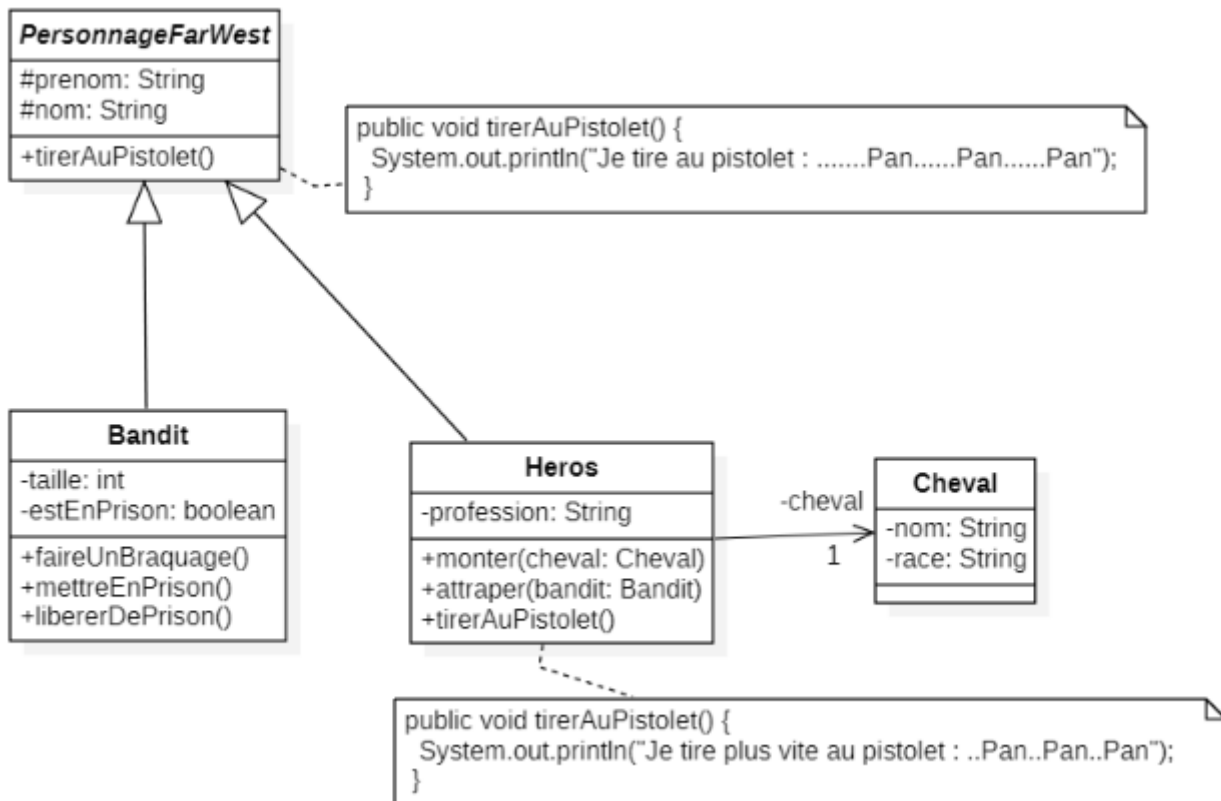
⇒ La conception proposée pourra-t-elle permettre un éventuel appel polymorphique à l'exécution, autrement dit le polymorphisme est-il mis en œuvre dans cette conception ?

☐ Oui

☐ Non

🔊 *Mais au fait, les héros ne tirent-t-ils pas plus vite que les bandits ?* 🔊
... Vous décidez donc de modifier la conception précédente ...

❑ Conception n°2 : Les héros tirent plus vite ...



⇒ Une fois cette conception implémentée, quel affichage obtient-on sur la console à l'exécution du jeu d'essai n°1 ?

⇒ Tous les *objets* du Far West peuvent-ils tirer au pistolet ?

☐ Oui

☐ Non

Si non, quels objets ne sont pas capable de tirer au pistolet ?

⇒ Pensez-vous que :

☐ Tous les *objets* qui tirent au pistolet ont tous la même manière de tirer

☐ Chaque *objet* qui tire au pistolet a sa *propre* manière spécifique de tirer

☐ Un seul (ou plusieurs) objets a une manière spécifique de tirer, tous les autres tirent de la même manière

⇒ La conception proposée pourra-t-elle permettre un éventuel appel polymorphe à l'exécution, autrement dit le polymorphisme est-il mis en œuvre dans cette conception ?

☐ Oui

Si oui, quelle annotation Java permet d'indiquer explicitement dans le code que le polymorphisme est mis en œuvre : _____

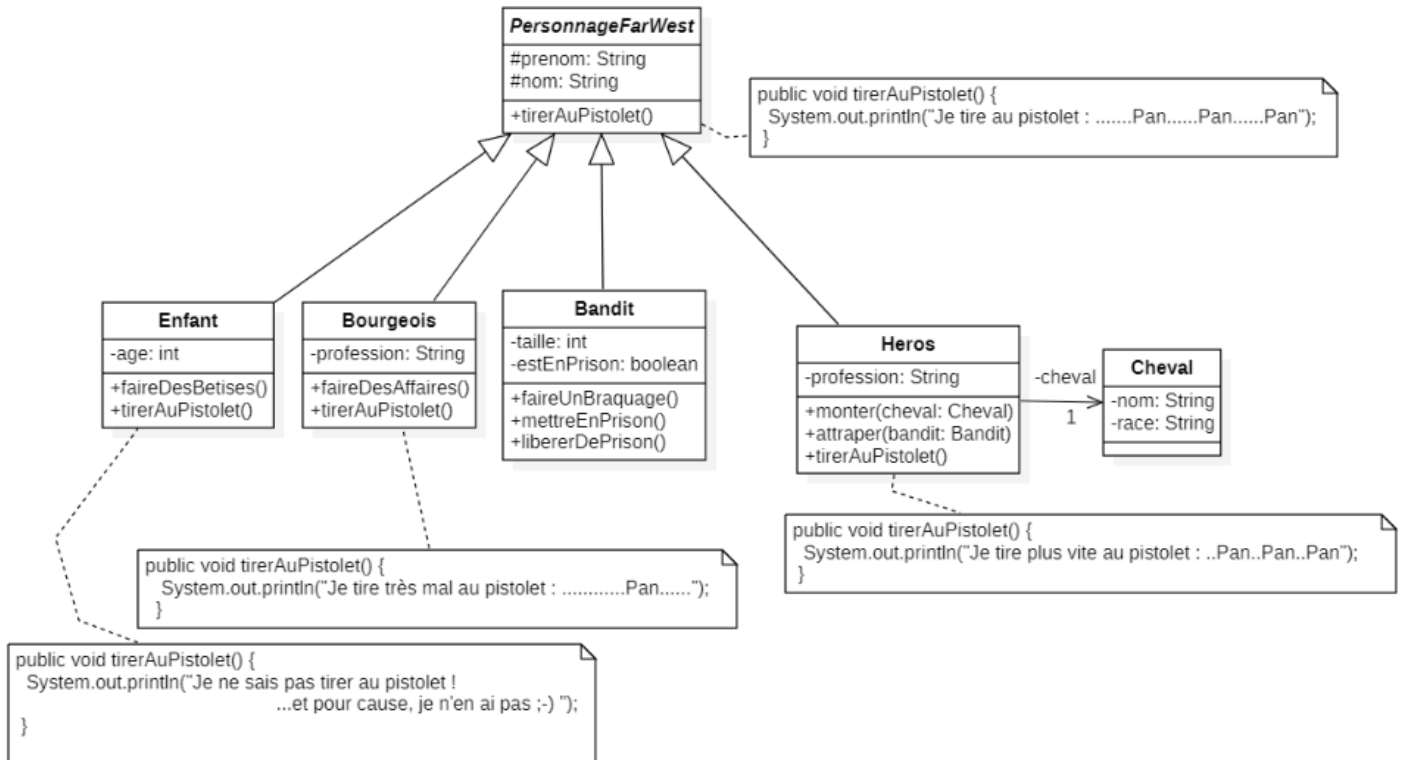
Quelle est la traduction française de cette annotation ? _____

Où cette annotation doit-elle être ajoutée ? _____

☐ Non

🔊 Mais au fait, n'y a -t-il pas aussi des enfants,
 mais aussi des bourgeoise(s) au Far West, victimes des bandit(s),
 et que les héros viennent sauver ? 🔊
 ... Vous décidez donc de modifier la conception précédente ...

❑ Conception n°3 : Des enfants et des bourgeois rejoignent le Far West 😊



⇒ Pour pouvoir tester tous les personnages du Far West, nous proposons un nouveau jeu d'essai que nous appellerons **jeu d'essai n°2** :

```

public static void main(String[] args) {
    Cheval jollyJumper = new Cheval("Jolly Jumper", "appaloosa");
    Heros luckyLuke = new Heros("Lucky", "Luke", "cow-boy", jollyJumper);
    Bandit joeDalton = new Bandit("Joe", "Dalton", 150);
    Bandit averellDalton = new Bandit("Averell", "Dalton", 190);
    Bourgeois zacharieMartins = new Bourgeois("Zacharie", "Martins", "inventeur");
    Enfant phineas = new Enfant("Phineas", 10);

    luckyLuke.tirerAuPistolet();
    joeDalton.tirerAuPistolet();
    averellDalton.tirerAuPistolet();
    zacharieMartins.tirerAuPistolet();
    phineas.tirerAuPistolet();
}

```

⇒ Une fois cette conception implémentée, quel affichage obtient-on sur la console à l'exécution du **jeu d'essai n°2** ?

⇒ Tous les *objets* du Far West peuvent-ils tirer au pistolet ?

☐ Oui

☐ Non

Si non, quels objets ne sont pas capable de tirer au pistolet ?

⇒ Pensez-vous que :

☐ Tous les *objets* qui tirent au pistolet ont tous la même manière de tirer

☐ Chaque *objet* qui tire au pistolet a sa *propre* manière spécifique de tirer

☐ Un seul (ou plusieurs) objets a une manière spécifique de tirer, tous les autres tirent de la même manière

⇒ La conception proposée pourra-t-elle permettre un éventuel appel polymorphique à l'exécution, autrement dit le polymorphisme est-il mis en œuvre dans cette conception ?

☐ Oui

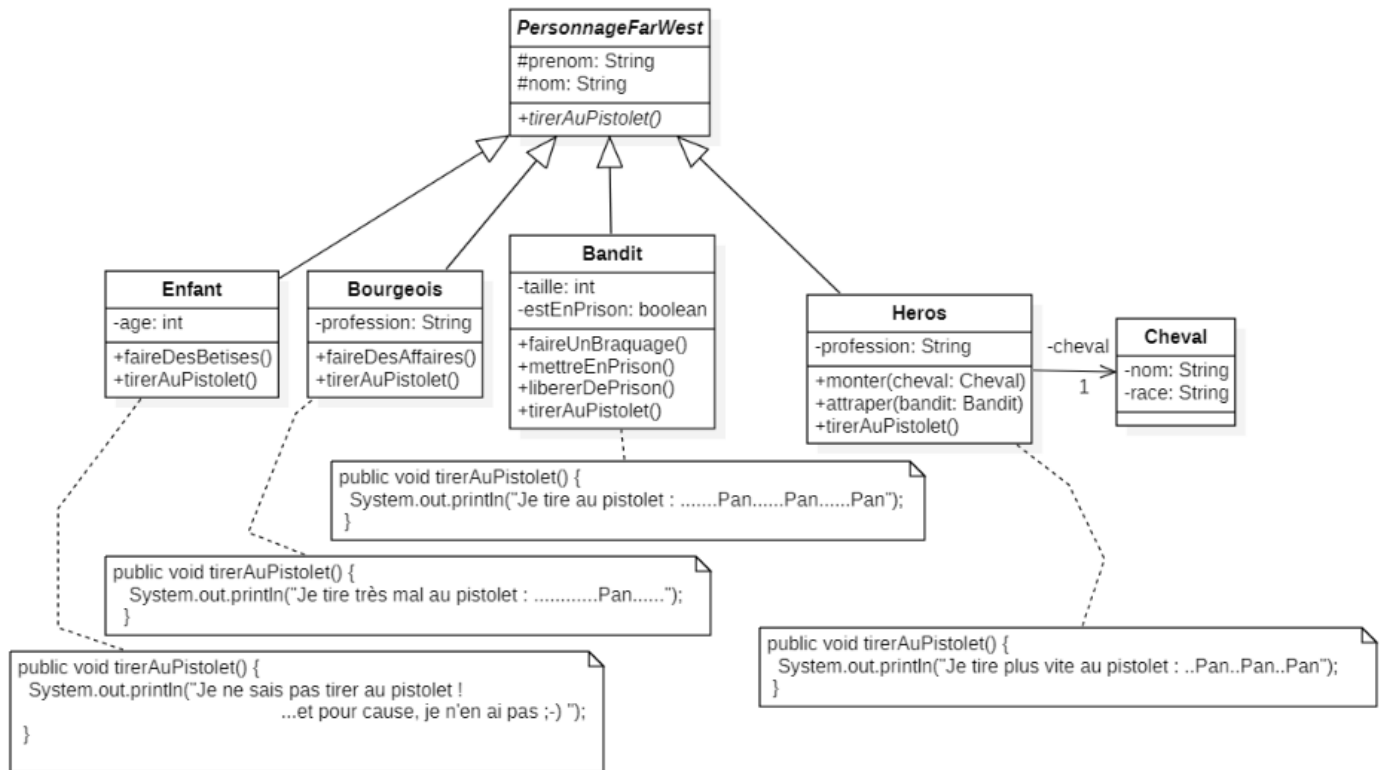
Si oui, indiquez où l'annotation Java : _____ doit être ajoutée :

☐ Non

*🔊 Puisque chacun a sa propre manière de tirer,
faisons en sorte que la nouvelle conception en tienne compte 🔊
... Vous décidez donc de modifier la conception précédente ...*

❑ Conception n°4 : A chacun sa manière de tirer au pistolet 😊

La nouvelle conception est plus *élégante* que la précédente. Elle permet de montrer de manière explicite que chaque *personnage du FarWest* se comporte différemment lorsqu'il tire au pistolet, ce qui se traduit par **une implémentation spécifique de la méthode tirerAuPistolet dans chaque classe fille.**



⇒ Quelles sont les différences avec la conception précédente ?

⇒ Quelle est désormais l'implémentation de la méthode **tirerAuPistolet** dans la classe **PersonnageFarWest** ?

⇒ La conception met-elle toujours en œuvre le polymorphisme ?

❑ Oui

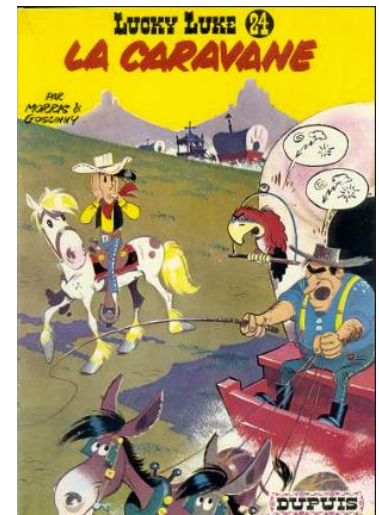
Si oui, ajoutez directement au(x) bon(s) endroit(s) sur le diagramme de classes précédent où durant la phase d'implémentation l'annotation Java `@Override` devra être ajoutée

❑ Non

⇒ La conception n°3 et la conception n°4 répondent de la même manière au jeu d'essai n°2.

Avec la nouvelle conception (n°4), nous pouvons proposer un nouveau jeu d'essai que nous appellerons **jeu d'essai n°3** :

```
public static void main(String[] args) {  
  
    Cheval jollyJumper = new Cheval("Jolly Jumper", "appaloosa");  
    Heros luckyLuke = new Heros("Lucky", "Luke", "cow-boy", jollyJumper);  
    Bandit joeDalton = new Bandit("Joe", "Dalton", 150);  
    Bandit averellDalton = new Bandit("Averell", "Dalton", 190);  
  
    Bourgeois zacharieMartins = new Bourgeois("Zacharie", "Martins", "inventeur");  
    Enfant phineas = new Enfant("Phineas", 10);  
  
    Collection <PersonnageFarWest> personnagesDansLaCaravane =  
        Arrays.asList(luckyLuke, joeDalton, averellDalton, zacharieMartins, phineas);  
  
    for (PersonnageFarWest personnage : personnagesDansLaCaravane) {  
        personnage.tirerAuPistolet();  
    }  
}
```



Remarque : Nous avons choisi dans ce jeu d'essai de regrouper les personnages du Far West qui apparaissent dans la Bande Dessinée **La Caravane** dans une objet de type Collection, mais un tableau auraient aussi très bien pu faire l'affaire 😊

❑ Une fois cette conception implémentée, quel affichage obtient-on sur la console à l'exécution du **jeu d'essai n°3** ?

❑ Au vu du résultat obtenu à l'issu de ce jeu d'essai, comprenez-vous un peu mieux l'intérêt du polymorphisme ?

🔊 Mais au fait, les enfants n'ont pas de pistolet
alors pourquoi leur imposer une méthode tirerAuPistolet ? 🔊

❑ Conception n°5 : Seuls les héros, les bandits et les bourgeois sont capable de tirer au pistolet 😊

Tirer avec une arme à feu est une capacité que de nombreux personnages du Far West possèdent. Mais tous les personnages du Far West n'ont pas de pistolet, comme les enfants par exemple.

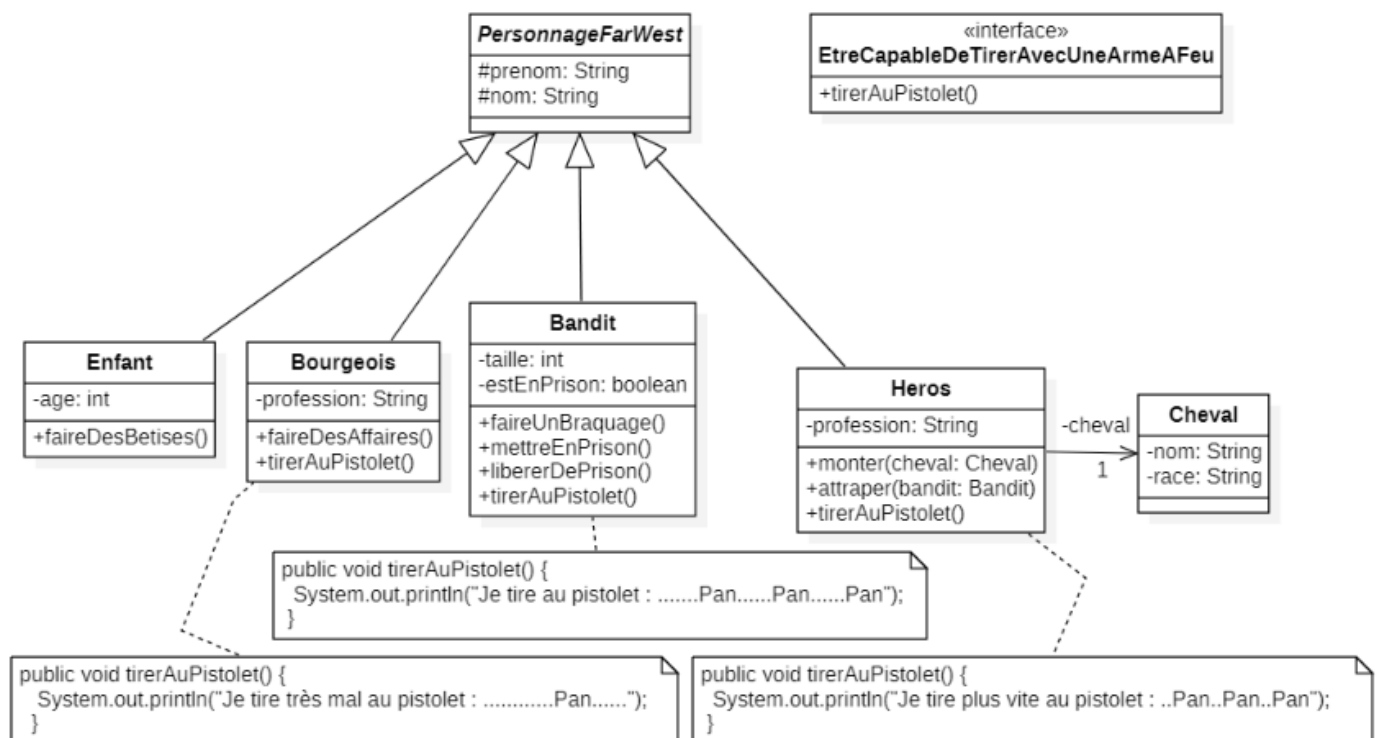
Nous décidons donc de proposer l'interface suivante EtreCapableDeTirerAvecUneArmeAFeu qui expose pour l'instant un seul service celui de pouvoir tirerAuPistolet



Remarque : Pour simplifier notre exemple, nous n'exposons dans notre interface qu'un seul service (une seule méthode), mais il faut savoir qu'une interface ne se limite pas forcément à un seul service et qu'elle peut exposer **un ensemble de services qui doivent être cohérents entre eux** (et surtout avec le nom de l'interface 😊)

Par exemple, le service tirerALaCarabine pourrait très bien être un service qui pourrait être ajouté plus tard à cet interface si le besoin évolue en ce sens ... 😊

Proposer une interface EtreCapableDeTirerAvecUneArmeAFeu avec une méthode tirerAuPistolet revient donc à retirer la méthode tirerAuPistolet de la classe PersonnageFarWest, ainsi que de la classe Enfant.



⇒ Compléter le diagramme de classes précédent en reliant correctement l'interface `EtreCapableDeTirerAvecUneArmeAFeu` aux classes qui sont capable de tirer avec une arme à feu.

⇒ Compléter l'entête de l'implémentation de la classe `Bourgeois` afin de respecter la conception proposée dans le diagramme de classes précédent :

```
public class Bourgeois _____ {  
    // ...  
}
```

⇒ Que se passe-t-il si on essaye de lancer le jeu d'essai n°3 avec le code implémenté pour cette conception ?

⇒ Compléter le code de la méthode `main` (déclaration de `Collection` et ajout d'une boucle `for`) pour qu'à l'exécution de ce jeu d'essai, l'affichage suivant soit obtenu :

```
Je tire plus vite au pistolet : ..Pan..Pan..Pan  
Je tire au pistolet : .....Pan.....Pan.....Pan  
Je tire au pistolet : .....Pan.....Pan.....Pan  
Je tire très mal au pistolet : .....Pan.....
```

```
public static void main(String[] args) {  
  
    Cheval jollyJumper = new Cheval("Jolly Jumper", "appaloosa");  
    Heros luckyLuke = new Heros("Lucky", "Luke", "cow-boy", jollyJumper);  
    Bandit joeDalton = new Bandit("Joe", "Dalton", 150);  
    Bandit averellDalton = new Bandit("Averell", "Dalton", 190);  
    Bourgeois zacharieMartins = new Bourgeois("Zacharie", "Martins", "inventeur");  
    Enfant phineas = new Enfant("Phineas", 10);  
  
    Collection<_____> personnagesCapableDeTirerAuPistolet =  
Arrays.asList(_____);  
  
    for (_____ ) {  
        personnage.tirerAuPistolet();  
    }  
}
```

⇒ Question subsidiaire :

Maintenant que la méthode `tirerAuPistolet` n'est plus un comportement par défaut de la hiérarchie **PersonnageFarWest**, serait-il pertinent d'introduire la classe `Cheval` dans cette hiérarchie ?


Plusieurs points de vue peuvent se défendre, justifiez votre choix, car rappelez-vous que :

Concevoir, c'est faire des choix 😊

Annexe : Interface vs Classe Abstraite

Interface	Classe abstraite
Une interface peut hériter un nombre illimité d'interfaces à la fois	Une classe abstraite ne peut hériter qu'une classe ou une classe abstraite à la fois.
<pre>interface Api extends i1, i2, i3 { // méthodes }</pre>	<pre>abstract class A { // méthodes } abstract class B extends A { // méthodes }</pre>
Les interfaces ne peuvent être héritées que par des interfaces. Les classes doivent les implémenter au lieu d'être héritées.	Une classe abstraite peut hériter une autre classe concrète ou abstraite
<pre>interface Api1{} interface Api2 extends Api1{} class MaClasse implements Api2{}</pre>	
Une interface ne peut avoir que des méthodes abstraites	Une classe abstraite peut avoir des méthodes abstraites et concrètes
<pre>interface Api{ public abstract void lire(); }</pre>	<pre>abstract class MaClasse { abstract void lire(); public void afficher(){ System.out.println("msg"); } }</pre>
Dans une interface, le mot clé «abstract» est facultatif pour déclarer une méthode en tant que abstract.	Dans une classe abstraite, le mot clé «abstract» est obligatoire pour déclarer une méthode en tant qu'abstrait.
Une interface ne peut avoir que des méthodes abstraites « public »	Une classe abstraite peut avoir des méthodes abstraites « protected » et « public »
L'interface n'a pas de modificateurs d'accès. Tout ce qui est défini à l'intérieur de l'interface est supposé « public ».	La classe abstraite peut avoir un modificateur d'accès.
Il est préférable d'utiliser l'interface lorsque plusieurs implémentations partagent uniquement la signature de méthode.	Il doit être utilisé lorsque différentes implémentations du même type partagent un comportement commun.
L'interface ne peut pas contenir des propriétés.	La classe peut avoir des propriétés.
Lorsque vous ne connaissez que les exigences et non son implémentation, vous utilisez une « Interface ».	Lorsque vous connaissez partiellement les implémentations, vous utilisez une « classe abstraite ».

Extrait : <https://waytolearnx.com/2019/04/difference-entre-une-interface-et-une-classe-abstraite-en-java.html>

	<p style="text-align: center;">A (re)voir</p>
	<p>→ dans la rubrique Classes, classes abstraites et interfaces Accès direct à la rubrique via : https://unil.im/butoo6 (URL complète : https://www.youtube.com/playlist?list=PLzzeuFUy_CniTo0Pm8Tdh7MVVYhF32fdx)</p> <p style="text-align: center;">❑ 03. Interface, exemple Comparable de User</p> <p>→ Dans la rubrique Programmation Objet : Encapsulation, Héritage et Polymorphisme en Programmation Objet Accès direct à la rubrique via : https://unil.im/butoo5 (URL complète : https://www.youtube.com/playlist?list=PLzzeuFUy_CniZpKCGfQ9HpJFxfkgGeGFO0)</p> <p style="text-align: center;">❑ 03. Héritage de type entre classes et interfaces</p>