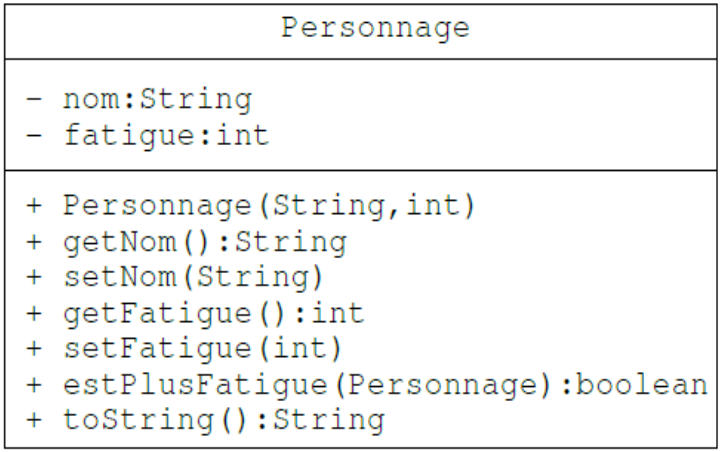
Programmation Orientée Objet (POO)
Java

Accès aux attributs privés d'un objet de la même classe

Contexte : *Le niveau de fatigue d'un personnage peut-être comparé à celui d'un autre personnage.*

Un objet peut accéder aux attributs privés d'un autre objet de la même classe. En effet, en Java, l'unité d'encapsulation est la classe et non l'objet.



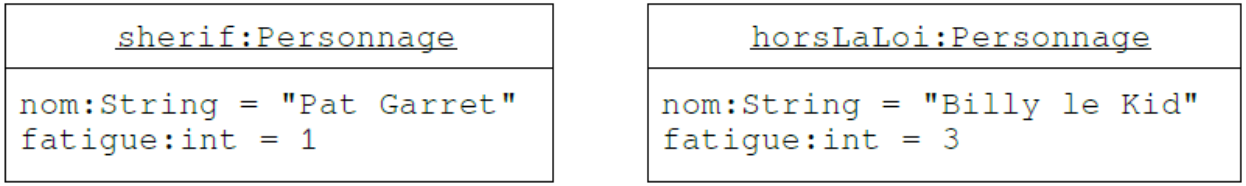
Représentation UML de la classe **`Personnage`**

A cette classe, nous venons d'ajouter la méthode `estPlusFatigue()` qui permet de savoir si le personnage est plus fatigué que le personnage passé en paramètre.

Instanciations et appel à la méthode `estPlusFatigue()` :

```
Personnage sherif = new Personnage( "Pat Garret" , 1 ) ;
Personnage horsLaLoi = new Personnage( "Billy le Kid" , 3 ) ;

String nomSherif = sherif.getNom() ;
String nomHLL = horsLaLoi.getNom() ;

if( sherif.estPlusFatigue( horsLaLoi ) == true ){
     System.out.println( nomSherif + " plus fatigué que " + nomHLL ) ;
}
else {
     System.out.println( nomSherif + " pas plus fatigué que " + nomHLL ) ;
}
```

Diagramme d'objets :



Première version de la méthode `estPlusFatigue()` de la classe `Personnage` :

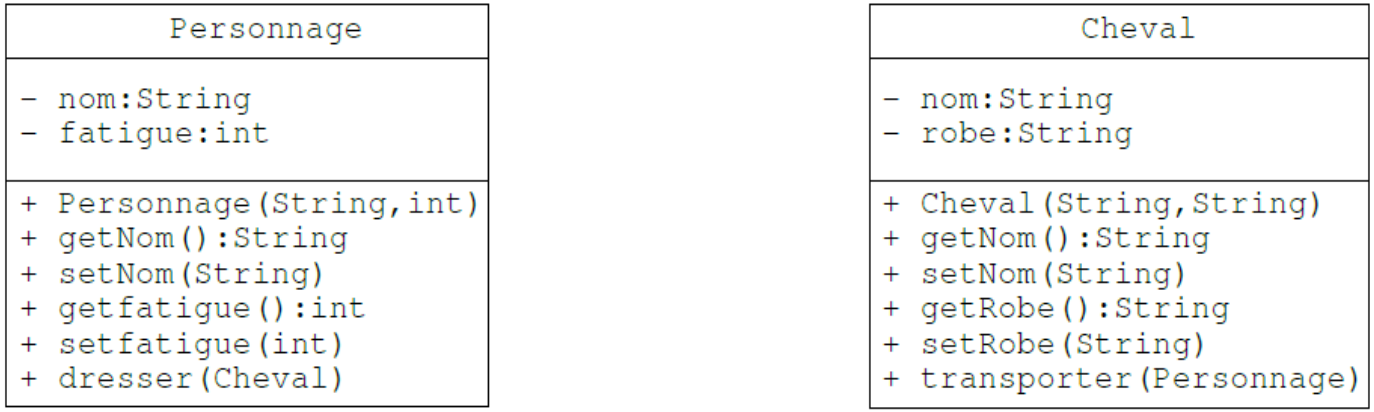
```
public boolean estPlusFatigue( Personnage autrePerso ){

      if( this.fatigue > autrePerso.getFatigue() )
            return true ;
      }
      else {
            return false ;
      }
}
```

Seconde version de la méthode `estPlusFatigue()` de la classe `Personnage` :

```
public boolean estPlusFatigue( Personnage autrePerso ){

      if( this.fatigue > autrePerso.fatigue )
            return true ;
      }
      else {
            return false ;
      }
}
```

Droits d'accès

Contexte : *Un cheval est caractérisé par son nom et par sa robe (couleur). Un personnage peut dresser un cheval et un cheval peut transporter un personnage.*



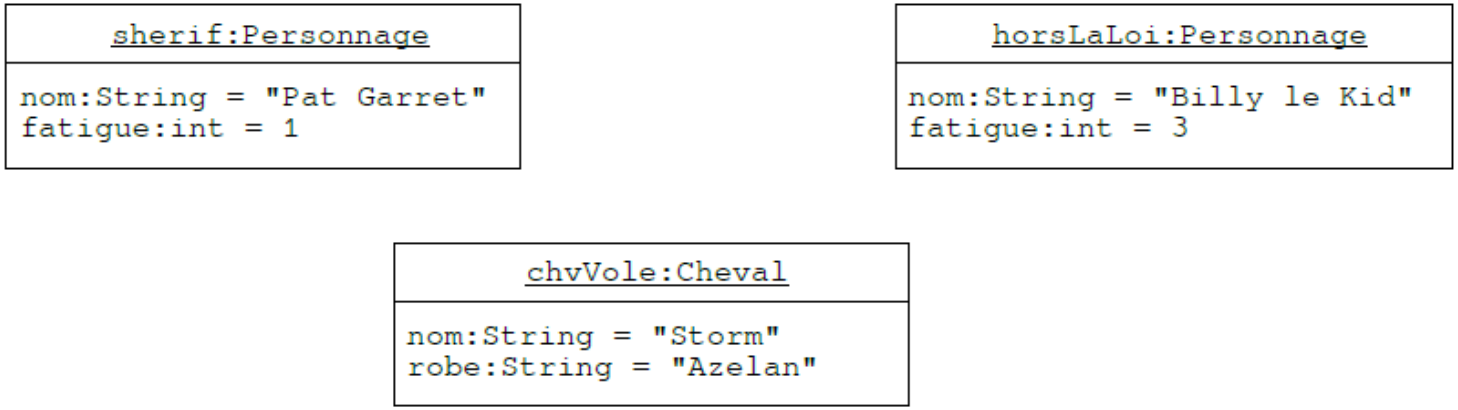
Représentation UML des classes ***Personnage*** *et* ***Cheval***

Les objets - Opérations

Instanciations :

```
Personnage sherif = new Personnage( "Pat Garret" , 1 ) ;
Personnage horsLaLoi = new Personnage( "Billy le Kid" , 3 ) ;
Cheval chvVole = new Cheval( "Storm" , "Azelan" ) ;
```

Diagramme d'objets :



Accessibilité (sans passer par les altérateurs/accesseurs) :

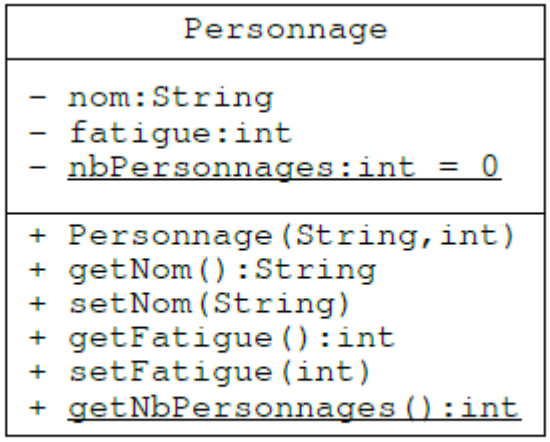
	sherif		horsLaLoi		chvVole	
	nom	fatigue	nom	fatigue	nom	robe
sherif						
horsLaLoi						
chvVole						

Les objets - Opérations

- Il est possible de définir des attributs, qui au lieu d'exister dans chacune des instances de la classe, n'existent **qu'en un seul exemplaire partagé** entre tous les objets de la même classe. On parle alors d'***attributs de classe*** ou encore d'***attributs statiques*** (déclarés avec le mot-clef `static` en Java).

- Il est aussi possible de créer des ***méthodes de classe*** (***méthodes statiques***) indépendantes de tout objet.

- Ces méthodes ne pourront accéder qu'aux attributs de classe alors qu'une méthode d'instance peut accéder aux attributs d'instance et aux attributs de classe.



Représentation UML de la classe `Personnage`

Les objets - Opérations

Implémentation Java de la classe `Personnage` :

```
public class Personnage {

      private String nom ;
      private int fatigue ;
      private static int nbPersonnages = 0 ;

      public Personnage( String nom , int fatigue ){
            this.nom = nom ;
            this.fatigue = fatigue ;
            nbPersonnages += 1 ;
      }

      ...

      public static int getNbPersonnages(){
            return nbPersonnages ;
      }
}
```

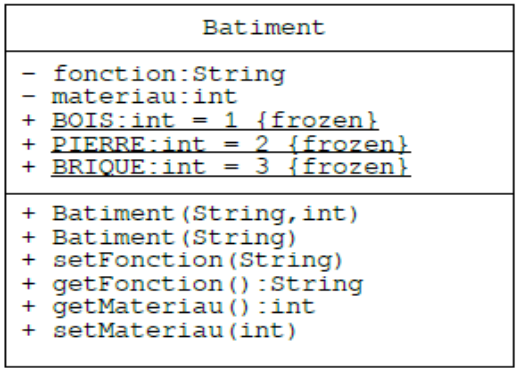
Instanciations et manipulations :

Code Java	***Trace d'exécution***
`Personnage outlaw1 , outlaw2 , outlaw3 ;`	
`System.out.println( Personnage.getNbPersonnages() ) ;`	0
`outlaw1 = new Personnage( "Bob" , 2 ) ;` `System.out.println( Personnage.getNbPersonnages() ) ;`	1
`outlaw2 = new Personnage( "Grat" , 0 ) ;` `System.out.println( Personnage.getNbPersonnages() ) ;`	2
`outlaw3 = new Personnage( "Bill" , 0 ) ;` `System.out.println( Personnage.getNbPersonnages() ) ;`	3

Constantes

Un attribut de classe dont la valeur est fixée de façon définitive à l'initialisation, peut-être vu comme une constante symbolique (*exemple :* La classe `Calendar` propose les constantes `DAY_OF_MONTH`, `DAY_OF_WEEK`, `MONTH`...).

Contexte : *Un bâtiment du jeu West peut-être construit en pierres, en bois ou en briques.*



Représentation UML de la classe `Batiment`

Implémentation Java de la classe `Batiment` :

```
class Batiment {

      private String fonction ;
      private int materiau ;

      public static final int BOIS = 1 ;
      public static final int PIERRE = 2 ;
      public static final int BRIQUE = 3 ;

      public Batiment( String fonction , int materiau ){
            this.fonction = fonction ;
            this.materiau = materiau ;
      }

      public Batiment( String fonction ){
            this.fonction = fonction ;
            this.materiau = BOIS ;
      }
```

Collections

- Une collection est une structure de données ***dynamique*** utilisée pour mémoriser des éléments.
- L'interface *générique* **`Collection<E>`** définit le comportement commun à toutes les collections.

- Cette interface est paramétrée sur le type des éléments à mémoriser (`E`).

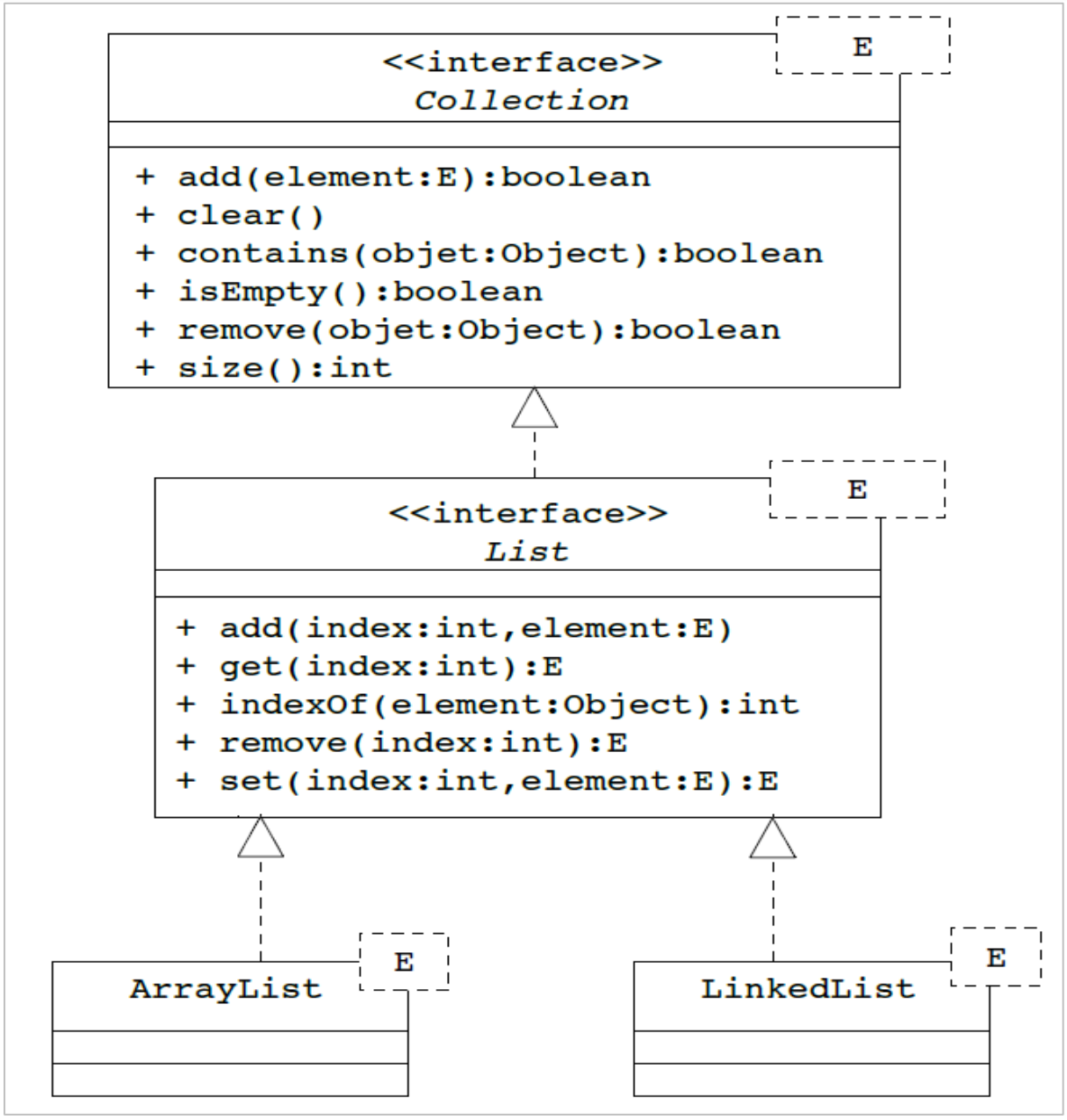
- Toutes les classes qui implémentent et qui complètent l'interface **`Collection<E>`** sont des classes dites ***génériques*** (depuis la version 5 de *Java*) :

- Tous les éléments d'une même collection sont du même type (`E`) ou d'un type dérivé de `E`.

- Les méthodes de base d'une collection sont l'***ajout*** et la ***suppression*** d'éléments.

- Java propose plusieurs types de collection : les ***listes*** (**`LinkedList<E>`**), les ***vecteurs*** dynamiques
- (**`ArrayList<E>`**), les ***ensembles*** (**`HashSet<E>`**)... Chaque type de collection a ses avantages et ses inconvénients.

- Le choix du type de la collection se fait en fonction de critères d'organisation et de performance.


<<interface>>
Collection
E
+ add(element:E):boolean
+ clear()
+ contains(objet:Object):boolean
+ isEmpty():boolean
+ remove(objet:Object):boolean
+ size():int
<<interface>>
List
E
+ add(index:int,element:E)
+ get(index:int):E
+ indexOf(element:Object):int
+ remove(index:int):E
+ set(index:int,element:E):E
ArrayList
E
LinkedList
E

- Les vecteurs dynamiques (classe **`ArrayList<E>`**) sont des collections dans lesquelles les éléments sont ordonnés.
- Les vecteurs dynamiques implémentent l'interface `List<E>`. En mémoire, ces listes utilisent des tableaux.
- Tout comme pour les tableaux d'objets, la classe **`ArrayList<E>`** propose des méthodes d'accès efficaces (fonctionnalités rapides) aux éléments (accès direct à un élément).
- La taille (le nombre d'éléments) d'un vecteur dynamique peut varier au cours de l'exécution de l'application, contrairement à celle d'un tableau.
- Par contre, de par son organisation mémoire, cette classe ne permet pas une insertion ou une suppression efficace (en terme de temps d'exécution) d'un élément à une position donnée.
- La classe ArrayList<E> est définie dans le package : ***`java.util.ArrayList`***.

Exemple :

La classe `Personnage` représente un personnage. Un extrait de l'implémentation Java de cette classe est donnée ci-dessous.

```
class Personnage {
      private String nom ;
      private int age ;

      public Personnage( String nom , int age ){
            this.nom = nom ;
            this.age = age ;
      }

      ...

      public String toString(){
            return this.nom + " [ " + this.age + " ans ]" ;
      }
}
```

Déclarer et initialiser la liste des personnages :

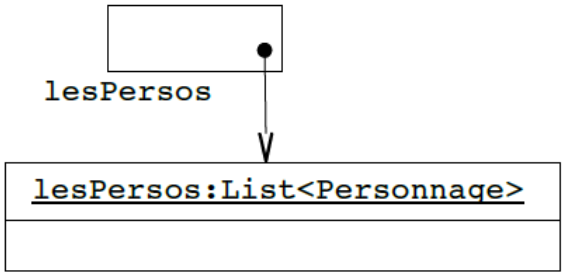
La variable **`lesPersos`** référence une liste de personnages (objets de la classe **`Personnage`**).

Cette variable est déclarée et initialisée avec une liste vide de la façon suivante :

```
ArrayList<Personnage> lesPersos = new ArrayList<Personnage>();
```

Peupler la liste des personnages :

```
lesPersos.add( new Personnage( "Joe" , 28 ) ) ;
lesPersos.add( new Personnage( "Willy" , 32)  ) ;
lesPersos.add( 0 , new Personnage( "Laura" , 27 ) ) ;
lesPersos.add( 1 , new Personnage( "Charles" , 43 ) ) ;
```



:Personnage
nom:String = "Joe" age:int = 28

:Personnage
nom:String = "Willy" age:int = 32

:Personnage
nom:String = "Laura" age:int = 27

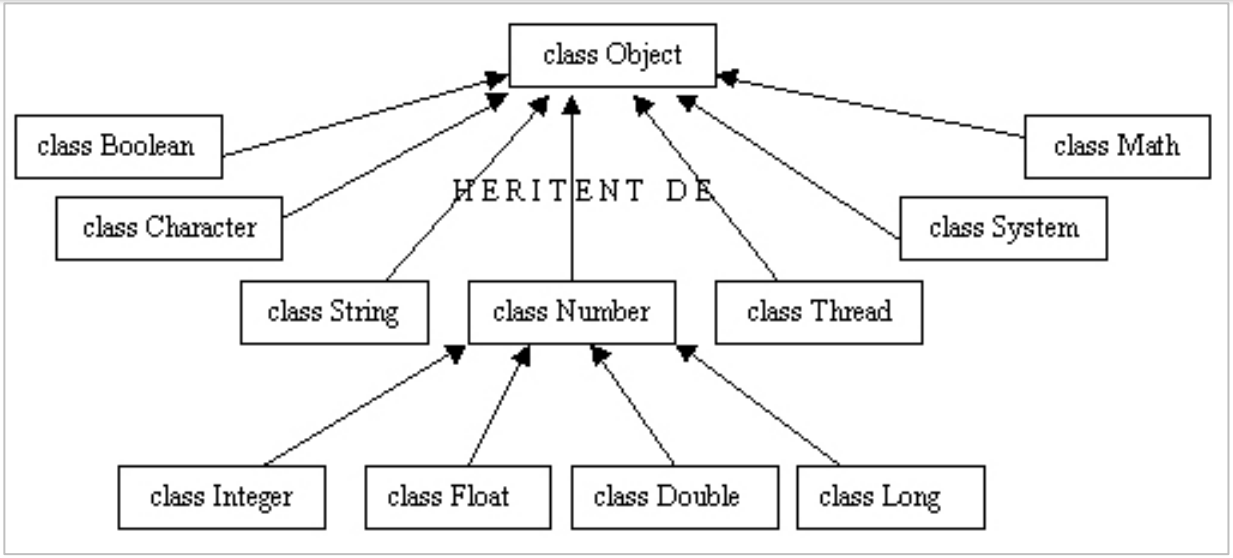
:Personnage
nom:String = "Charles" age:int = 43

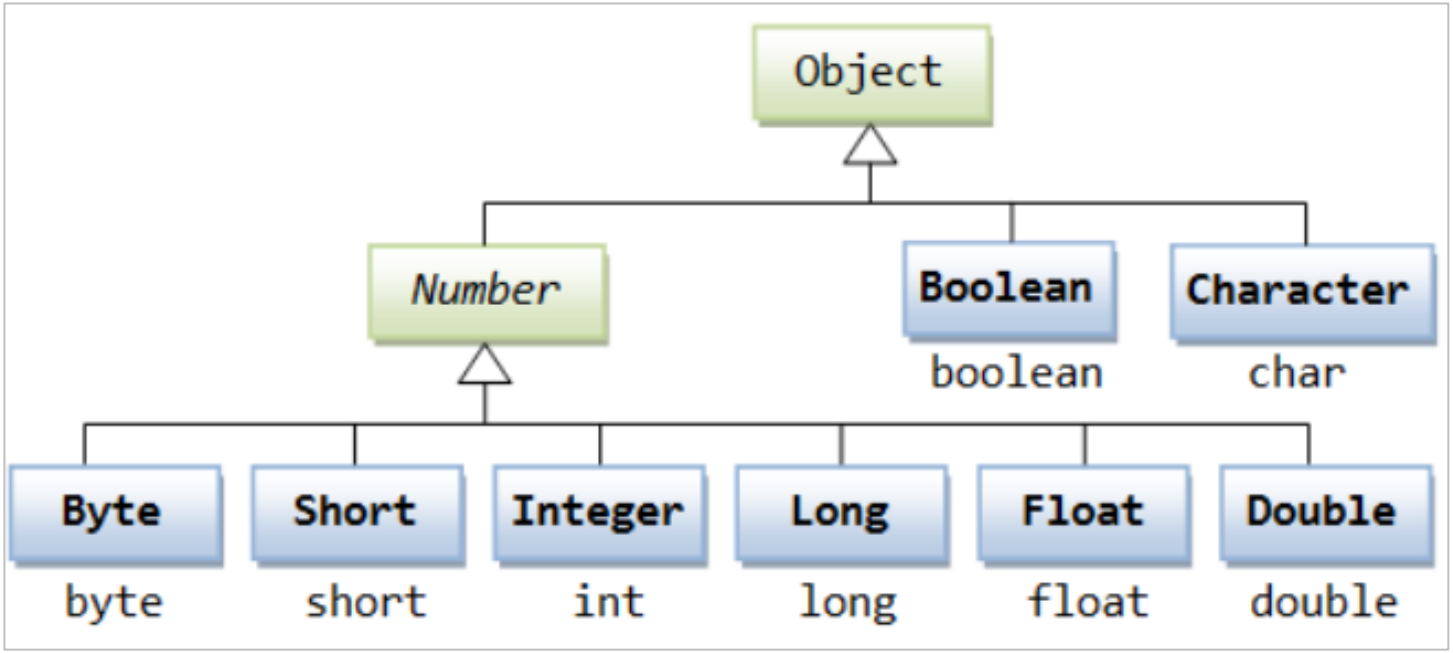
boolean add(Object)	Ajouter un élément à la fin du tableau
void clear()	Supprimer tous les éléments du tableau
Object get(index)	Renvoyer l'élément du tableau dont la position est précisée
int indexOf(Object)	Renvoyer la position de la première occurrence de l'élément fourni en paramètre
boolean isEmpty()	Indiquer si le tableau est vide
Object set(int, Object)	Remplacer l'élément à la position indiquée par celui fourni en paramètre
int size()	Renvoyer le nombre d'éléments du tableau
Object remove(int)	Supprimer dans le tableau l'élément fourni en paramètre

Java – Héritage

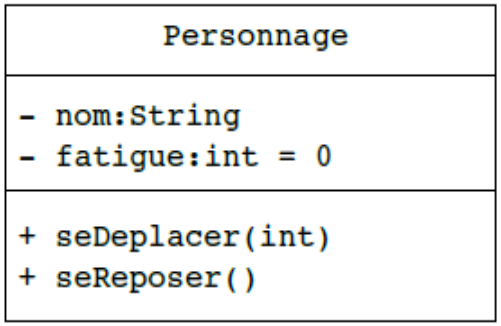
- La *Programmation Objet (POO)* s'appuie entre autre sur le concept d'***héritage***.
- Ce concept permet la réutilisation de classes existantes. On peut créer, à partir d'une classe dite "**classe de base**" (ou "super classe"), de nouvelles classes dites "**classes dérivées**" (ou "sous-classes").
- L'héritage permet de spécialiser les classes existantes : La classe ***Etudiant*** spécialise la classe ***Personne***.
- Une classe dérivée hérite des champs et des méthodes de sa classe de base. Une classe dérivée peut devenir une classe de base pour d'autres classes dérivées. On parle alors de dérivations successives.
- Toute classe hérite d'une façon ou d'une autre de la super classe **Object**.
- Le langage java n'autorise pas l'héritage multiple (Une Classe dérivée appartenant à plusieurs classes mère). Java utilise une autre technique très puissante pour palier à ce problème : L'interfaçage.
- Le langage C++ permet l'héritage multiple.

Java – Héritage





```
class Personnage {

      private String nom ;
      private int fatigue = 0 ;

      public Personnage( String nom , int fatigue ){
            this.nom = nom ;
            this.fatigue = fatigue ;
      }

      public Personnage( String nom ){
            this.nom = nom ;
      }

      public String getNom(){
            return this.nom ;
      }

      public void setNom( String nom ){
            this.nom = nom ;
      }

      public int getFatigue(){
            return this.fatigue ;
      }
```



Représentation UML de la classe Personnage

```
        public void setFatigue( int fatigue ){
                this.fatigue = fatigue ;
        }

        public void seDeplacer( int nbKm ){
                if( nbKm > 3 && fatigue < 10 ){
                        fatigue = fatigue + 1 ;
                }
        }

        public void seReposer(){
                this.fatigue = 0 ;
        }
}
```

```
class Indien extends Personnage {

      private String tribu ;

      public Indien( String nom , int fatigue , String tribu ){
            super( nom , fatigue ) ;
            this.tribu = tribu ;
      }

      public Indien( String nom , String tribu ){
            super( nom ) ;
            this.tribu = tribu ;
      }

      public String getTribu(){
            return this.tribu ;
      }

      public void setTribu( String tribu ){
            this.tribu = tribu ;
      }

}
```

Personnage
- nom:String - fatigue:int = 0
+ Personnage(String,int) + Personnage(String) + getNom():String + setNom(String) + getFatigue():int + setFatigue(int) + seDeplacer(int) + seReposer()

Un indien est avant tout un Personnage

Indien
- tribu:String
+ Indien(String,int,String) + Indien(String,String) + getTribu():String + setTribu(String)

Diagramme de classes

Héritage et Constructeur

- Lors de la création d'un objet de la classe **Indien**, les attributs hérités de la classe **Personnage** (***nom*** et ***Fatigue***) doivent-être initialisés au même titre que l'attribut ***tribu***.

- Pour se faire, les constructeurs de la classe **Indien** font appel aux constructeurs de la classe **Personnage** au moyen de la méthode **super()**.

- Cette instruction d'appel au constructeur de la classe de base doit-être la première instruction du constructeur de la classe dérivée.

Remarques

- **super()** ne permet d'appeler que le constructeur de la classe de base de niveau immédiatement supérieur.

- Si aucun constructeur n'a été défini dans cette classe de base, c'est son pseudo-constructeur par défaut qui est appelé.

- L'instruction super() est alors le seul appel autorisé dans le constructeur de la classe dérivée.

Héritage et Constructeur

```
class Oeuvre {
   //les attributs
   protected String titre, auteur, proprietaire ;
   protected double prix ;
   //le constructeur avec comme paramètre le titre,
l'auteur et le prix
   //l'attribut proprietaire est initialisé
   public Oeuvre(String t, String a, double p){
      titre = t ; auteur = a ; prix = p ; proprietaire = "";
   }
   //une oeuvre est neuve si elle n'a pas encore de
propriétaire
   //d'où l'intérêt de l'initialisation dans le
constructeur
   public boolean Est_neuf() {
      if ( proprietaire =="" ) return true ;
      else return false ;
   }
   //affiche les informations sur l'oeuvre
   public void Afficher() {
      System.out.println("Titre : " + titre) ;
      System.out.println("Auteur : " + auteur) ;
      System.out.println("Prix : " + prix) ;
      if (this.Est_neuf())
         System.out.println("Aucun proprietaire" );
      else
         System.out.println("Proprietaire : "+proprietaire);
      System.out.println() ;
   }
```

```
//La classe Film hérite de la classe Oeuvre
class Film extends Oeuvre {
   //un seul attribut supplémentaire:la durée du film en mn
   protected int durée ;
   //le constructeur de cette classe appelle celui de la
   //classe supérieure c'est-à-dire de la classe Oeuvre
   public Film(String t, String a, double p, int d){
      super(t,a,p) ;
      durée = d ;
   }
   //la méthode revente modifie simplement le propriétaire
et le prix du livre
   public void Revendre(String acheteur,double prix_occas){
      proprietaire = acheteur ;
      prix = prix_occas ;
   }
   public String Proprietaire() {
      return (proprietaire) ;
   }
}
```

Redéfinition des membres

- Pour illustrer cette partie, on utilisera la méthode toString() définie dans la classe Object.
- Pour rappel, toutes les classes dérivent, d'une façon ou d'une autre, de cette super classe.
- La classe Object propose une méthode nommée toString() qui est appelée dans un contexte de chaîne (expression basée sur l'opérateur de concaténation par exemple).
- Cette méthode retourne une représentation textuelle de l'objet dans laquelle on retrouve le nom de la classe de l'objet ainsi que sa référence.

Méthode `toString()` de la classe `Personnage` redéfinie

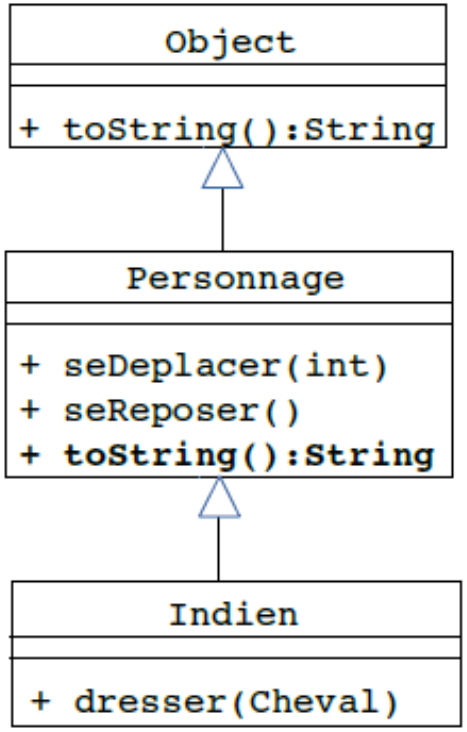


Diagramme de classes

Dans la définition de la classe `Personnage`, redéfinissons la méthode héritée `toString()`.

```
public String toString(){
    return "nom = " + nom + ", fatigue = " + fatigue ;
}
```

Appel à la méthode de la super classe

Dans la définition de la classe **`Indien`**, on redéfinit la méthode héritée **`toString()`** pour répondre au besoin de la nouvelle classe :

```
public String toString(){
return super.toString() + ", tribu = " +
tribu ;
}
```

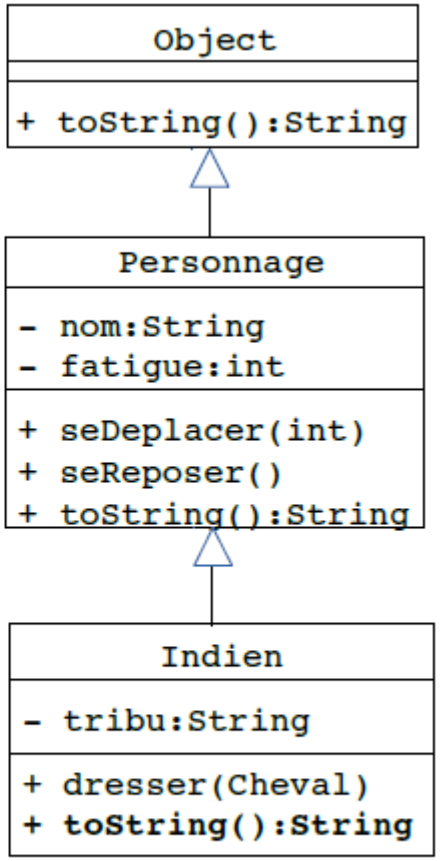


Diagramme de classes