Héritage

L'héritage = Un mécanisme de réutilisation de code

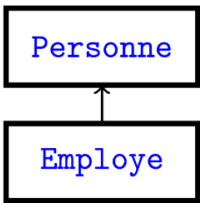
- L'héritage est un principe fondamental de la programmation orientée objet (POO) qui permet à une classe d'hériter des **propriétés et des comportements (méthodes et attributs)** d'une autre classe ;
- Certaines classes sont, par nature, des spécialisations d'autres classes : elles partagent les mêmes propriétés et méthodes, tout en ajoutant d'autres.

Le mot clé ***extends*** précise au compilateur que *Employe* est une classe dérivée de la classe *Personne*.

Terminologie

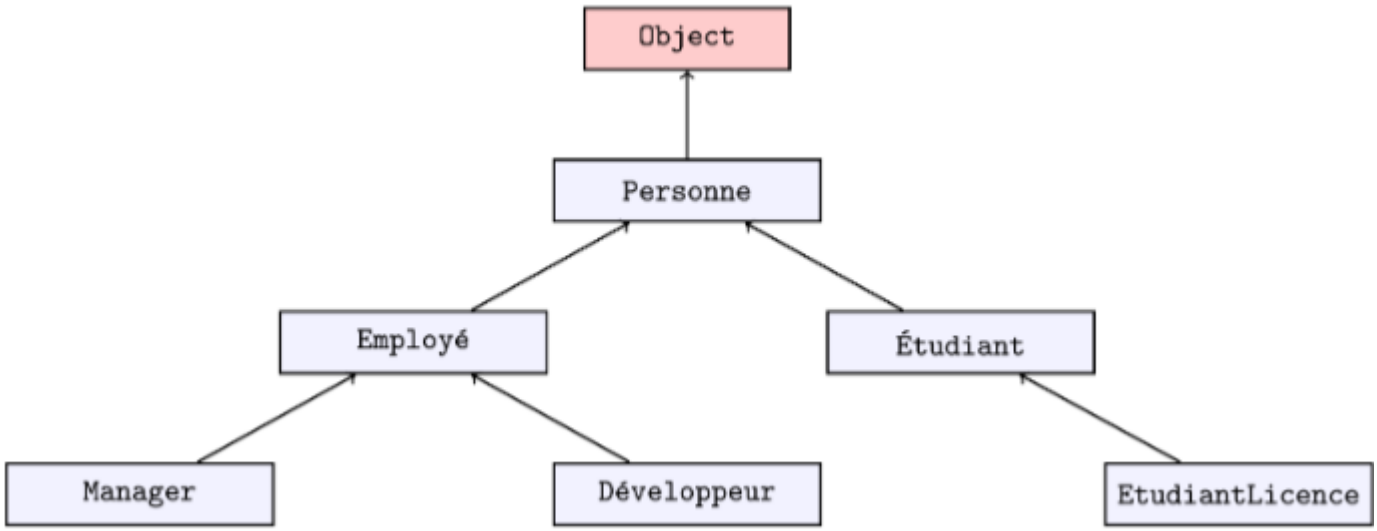
- *Personne* : Super-classe (aussi : **classe de base**, classe **parente**) ;
- *Employe* : Sous-classe (aussi : classe **dérivée**, ou classe **enfant**).

Tous les membres de la classe parent sont **automatiquement** membres de la classe enfant (hérités).



Structure arborescente

- En Java, une classe peut avoir plusieurs sous-classes ;
- Mais une classe peut avoir une seule classe parent (contrairement à certaines autres langages) ;
- La hiérarchie est en général arborescente.
- La classe *Object* (dans *java.lang*) est la racine de cette hiérarchie ;
- Une classe qui n'étend aucune classe, étend implicitement *Object*.



Avantages de l'héritage

1. Réutilisation du code : L'héritage permet aux classes dérivées de réutiliser le code des classes de base, sans savoir à le réécrire ;
2. Maintenance du code : Les modifications apportées à la super-classe se répercutent automatiquement sur toutes les sous-classes, ce qui simplifie la maintenance et la mise à jour du code.

Accès aux méthodes

Important : Un objet d'une classe dérivées accède aux membres publics de sa super-classe, comme s'ils étaient définis dans la classe dérivée elle-même.

Accès d'une classe dérivée aux membres de sa classe de base

Important : Une méthode de la classe dérivée :

- Peut manipuler les champs hérités visibles ;
- N'a pas accès direct aux champs privés de sa classe de base.

Constructeurs

Constructeur de la classe dérivée

Note : Le constructeur de la classe dérivée doit prendre en charge **l'intégralité de la construction** de l'objet.

Deux options :

- Initialiser les trois champs dans le constructeur de la classe dérivée ;
  Attention : Comme le champ nom est privé, il faut disposer de méthodes d'altération ($setter$) dans la classe $Personne$ pour pouvoir le modifier.
- Appeler le constructeur de la super-classe ($Personne$) :
  - Initialiser le champ $employeur$ (accessible)
  - Appeler le constructeur de $Personne$ pour initialiser les champs nom et age.

Si un constructeur d'une classe dérivée appelle un constructeur d'une classe de base, il utilise le mot-clé super et cette instruction doit obligatoirement se faire en premier.

Attention : Une classe peut dériver d'une classe qui dérive elle-même d'une autre classe. Dans ce cas, l'appel par super ne concerne que le constructeur de la classe de base du niveau immédiatement supérieur.

Absence de constructeur

Deux cas à traiter :

1. La super-classe ne possède aucun constructeur :
   - Quand **aucun constructeur** n'est déclaré dans A, Java génère « $A( )$ » (sans argument) ;
   - La construction d'un B se fait par étages : d'abord $A( )$, puis $B( )$.

   A retenir :
   - $super( )$ est toujours la première instruction d'un constructeur (implicite si non écrit) ;
   - On peut créer et utiliser B sans connaître les détails d'initialisation de A.
2. La classe dérivée ne possède aucun constructeur.
   - Une seule façon de construire un objet de type B : $B\ b = new\ B( );$
   - Appel du constructeur sans arguments de A (constructeur sans arguments).

   Erreur de compilation : Le constructeur par défaut B cherche à appeler le constructeur sans arguments de A. Comme A dispose d'au moins un constructeur. Il n'est plus possible d'appeler le constructeur par défaut A.

   S'il n'y a aucun constructeur dans A et dans B : Le constructeur par défaut de B est appelé, qui appelle à son tour le constructeur par défaut de A. Aucun problème à la compilation.

Redéfinition et surcharge de méthodes

Redéfinition

Redéfinition de méthode (overriding) : Fonctionnalité en Java qui permet à une sous-classe de fournir une implémentation spécifique d'une méthode qui est déjà définies dans sa super-classe.

- **Intérêt de la redéfinition**
  - Personnalisation du comportement : Adapter le comportement hérité pour mieux correspondre aux besoins spécifiques de la sous-classe ;
  - Polymorphisme : Permettre aux objets de sous-classes différentes d'être traités de manière uniforme tout en exécutant des comportements spécifiques à leur type.

  Une règle importante :
  - Pour redéfinir une méthode il faut respecter sa signature (nom et liste des arguments).
  - Une méthode redéfinie cache la méthode de la classe parent ;
  - Il est important de préciser qu'on souhaite appeler la fonction de sa super classe : Il suffit d'utiliser le mot-clé $super$.
- **Variables vs méthodes**

  Pour les variables : Prise en compte du nom de la variable (pas le type)
  - Dans une classe, à chaque nom de variable ne correspond qu'une seule déclaration ;
  - Même nom dans une sous-classe ⇒ Redéfinition.

  Pour les méthodes : Prise en compte de la signature (nom + type des paramètres)
  - Dans une classe, à une signature correspond une seule définition ;
  - Même signature dans une sous-classe ⇒ Redéfinition.
- **Type de retour**
  - Le type de retour d'une fonction **ne fait pas partie de la signature** ;
  - Il peut changer dans une redéfinition.

  Restriction : Doit être un sous-type de l'original.
- **Modificateur d'accès**

  Le modificateur d'accès peut changer dans une redéfinition.

  La redéfinition d'une méthode **ne doit pas diminuer les droits d'accès** à cette méthode.

  Il est cependant possible **d'augmenter** les droits d'accès.
- **Interdire la redéfinition**
  - Le modificateur $final$ interdit la redéfinition pour une méthode ;
  - Remarque : En revanche une variable avec modificateur $final$ (c'est-à-dire une **constante**) peut être occultée.

Le mot clé @Override

- Le mot-clé $@Override$ est annotation en Java utilisée pour indiquer **explicitement** qu'une méthode est destinée à redéfinir une méthode dans une super-classe ;

L'annotation $@Override$ se place directement au-dessus de la méthode redéfinie dans la sous-classe.

- **Intérêt :**

  Vérification par le compilateur :

- Le compilateur Java vérifie que la méthode annotée redéfinit bien une méthode dans la super-classe ;
- Si ce n'est pas le cas, une erreur de compilation est générée ;
- Cela permet de détecter les erreurs comme les fautes de frappe dans le nom de la méthode ou des signatures de méthodes incorrectes.

Lisibilité et Maintenance du Code :

Indiquer **explicitement** qu'une méthode redéfinit une méthode de la super-classe améliore la lisibilité et la maintenance du code ainsi que la compréhension des intentions du développeur.

Surcharge des méthodes (overloading)

Si le nom d'une méthode est le même, mais pas la signature, on ne parle pas de redéfinition (**overriding**), mais de surcharge (**overloading**).

Une classe dérivée peut surcharger une méthode d'une classe de base (ou plus généralement d'une classe ascendante).

- **Valeur de retour :**

  Remarque : Lorsqu'une méthode est surchargée, il n'est p**as obligatoire de respecter le type de la valeur de retour**.

Polymorphisme

- Un objet d'une sous-classe est également un objet de sa super-classe ;
- Il hérite de tous le champs et méthode de la super-classe.

Un objet d'une sous-classe peut être affecté à une variable de type super-classe.

Note : L'inverse n'est pas possible.

Polymorphisme :

- Une même variable de type super-classe peut référencer des objets de plusieurs sous-classes différentes au cours de sa vie ;
- Ce phénomène est appelé polymorphisme.

- **Règle générale**
  - Un objet avec un type effectif est aussi considéré comme un objet de tous les types supérieurs dans la hiérarchie ;
  - Cela signifie qu'il peut être référencé par des variables de tout type supérieur ou égal à C.
  - Le type déclaré de c est C ; Le type effectif de c est également C ;
  - Cependant, *c* est aussi implicitement considéré comme un objet de type B et A, car C hérite de B et A.

Attribution polymorphique :

- Un objet de type effectif C peut être référencé par une variable de type déclaré D si D est une classe située au-dessus de C dans la hiérarchie d'héritage ;
- Le polymorphisme permet à un objet d'être manipulé par des variables de ses classes parentes.

Type déclaré et type effectif

Une conséquence du polymorphisme est que lorsqu'un objet est créé, il possède deux types distincts :

Type déclaré :

- C'est le type utilisé lors de la déclaration de variable ;
- Détermine à la compilation et ne change jamais.

Type effectif :

- C'est la classe avec laquelle l'objet référencé a été réellement instancié ;
- Ce type peut changer dynamiquement, c'est-à-dire au moment de l'exécution.

Différence clé :

- Type déclaré : Fixe et ne peut pas être modifié après la modification ;
- Type effectif : Dynamique, il correspond à la classe réelle de l'objet créé et eut varier au cours de l'exécution.

La méthode $getClass()$

- La méthode $getClass()$ retourne le type effectif (classe réelle) de l'objet sur lequel elle est appelé.

Utilisation de $instanceof$ pour tester le type d'un objet

- L'objet $instanceof$ permet de tester si un objet est une instance d'une classe donnée ou d'une de ses classes parentes ;
- Il retourne $true$ si l'objet est du type spécifié, ou d'un type qui hérite de ce dernier dans la hiérarchie d'héritage.

Résumé : $obj\ instanceof\ A$ retourne $true$ si :

- A est le type effectif de obj ;
- A est un ancêtre (classe mère ou au-dessus) du type effectif de obj.

Sinon, $instanceof$ retourne $false$.

Polymorphisme et type déclaré

Règle de vérification

- Un appel de méthode $a.f()$ est valide si le type déclaré de a possède une méthode $f()$ applicable ;
- Un accès à un champ $a.m$ est valide si le type déclaré de a possède un champ m.

Important : Le compilateur ne connaît que le type déclaré d'une variable, il ne vérifie donc que les méthodes et champs disponible pour ce type.

- Le type déclaré d'une variable définit ce que l'on peut faire avec l'objet ;
- Seuls les champs et méthodes visibles de la classe correspondant au type déclaré sont accessibles, même si l'objet a un type effectif plus spécifique.

Intérêt du polymorphisme

- Grâce au polymorphisme, la version spécifique de la méthode $getRole$ propre à chaque sous-classe est appelée.

Un intérêt du polymorphisme : Traiter des objets **uniformément**, avec les méthodes qu'ils ont en commun, même s'ils sont de type effectif différent.

Dynamic binding (liaison dynamique)

Dynamic binding

- Processus par lequel la JVM détermine au moment de l'exécution, quelle méthode doit être invoquée lorsqu'une méthode est appelée sur un objet en fonction de son type ;
- C'est le type **effectif** d'un objet qui détermine la définition de la méthode à utiliser ;
- Le type effectif est connu seulement à l'exécution.

Grâce au polymorphisme, la boucle peut appeler la même méthode (par exemple : $getRole()$) sur chaque élément typé $Personne$ sans if ni $cast$; c'est l'objet réel ($Personne$, $Employe$, $Developpeur$) qui fournit automatiquement le bon comportement.

- **Remarque**
  - Le type **déclaré** détermine quelles méthodes on peut invoquer ;
    Fait statiquement (en phase de compilation).
  - Le type **effectif** détermine quelle définition à utiliser pour ces méthodes ;
    Fait dynamiquement (en phase d'exécution).

Liaison dans le cas des variables

Les variables ne bénéficient pas de liaison dynamique en Java !

Liaison statique (static binding) :

- La liaison entre le nom d'une variable et sa définition se fait à la compilation, un mécanisme appelé liaison statique (static binding) ;
- Ce lien **ne dépend pas du type effectif** ;
- Lorsqu'une variable est variable est accédée, c'est la version correspondant au type déclaré de la référence qui est utilisée, pas celle du type effectif de l'objet auquel la référence pointe ;
- Redéfinir un champ n'a donc pas le même effet que redéfinir une fonction.

Conséquence de la liaison statique

- Une autre conséquence de la liaison statique : Une méthode de classe A peut uniquement accéder aux champs de la classe A (ou de ces ancêtres) ;
- Pour les champs, le nom écrit dans une méthode est résolu à la compilation selon la classe où la méthode est définie ;
- Donc, dans une méthode de A, le nom c désigne un champ de A (ou d'un ancêtre de A), **jamais** un champ ajouté dans une sous-classe.

Cas des méthodes $static$

- Une méthode de classe ($static$) peut être redéfinie ou surchargée comme toute autre méthode ;
- Mais la liaison des méthodes statiques est **statique** ;
- Le comportement d'une méthode statique est déterminé par le type déclaré.

Casting :

- $((B)ab).f();$: Même si la méthode $f()$ est statique, l'instruction $((B)ab).f();$ force Java à traiter la référence comme étant de type B, ce qui fait que la méthode statique $f()$ de la classe B sera appelée ;
- $((A)b).f();$: Le cast transforme la référence en type A, donc la méthode statique $f()$ de la classe A sera appelée.

Casting et héritage

Casting : règles

Sur les types références, le **casting est autorisé** uniquement si les types déclarés appartiennent à une même hiérarchie d'héritage.

Upcasting : Un mécanisme toujours sur

- Le upcasting consiste à convertir un objet d'un type dérivé (sous-classe) vers un type de **niveau supérieur** dans la hiérarchie des classes.

L'upcasting est toujours sur :

- Toute instance d'une sous-classe est également une instance de sa super-classe ;
- Cela masque des fonctionnalités spécifiques à la sous-classe pour utiliser uniquement celles définies dans la super-classe.

Downcasting :

- **Manipuler des types spécifiques**
  - Le downcasting consiste à convertir un objet d'un type de niveau supérieur vers un type dérivé (sous-classe) dans la hiérarchie des classes.
- **Précaution à prendre :**
  - Contrairement à l'upcasting, le downcasting n'est pas toujours sûr et nécessite des vérifications préalables, car un objet de super-classe ne peut pas être automatiquement traité comme une sous-classe sans validation ;
  - Le downcasting doit être effectué uniquement si l'objet référencé est réellement une instance de la sous-classe cible, sinon une exception de type $ClassCastException$ sera levée ;
  - Le downcasting est sûr uniquement s'il ne dépasse pas le type effectif ;
- **« pattern matching » avec $instanceof$**
  - Permet de simplifier la vérification du type d'une variable et son transtypage (casting) en un seul et même endroit dans le code.

Ordre d'initialisation

Ordre d'initialisation (sans héritage)

Lorsqu'un constructeur est invoqué, l'objet est déjà créé et créé et ses champs sont initialisés avec des valeurs par défaut :

- Champs numériques : $\emptyset$;
- Champs booléens : $false$;
- Champs référence : $null$.

En effet, les initialisations d'instance s'exécutent avant le corps du constructeur :

1. Champs initialisés dans **l'ordre d'écriture** ;
2. Blocs d'initialisation $\{\dots\}$;
3. Puis le corps du constructeur.

Ordre d'initialisation – avec chaînage $this(\dots)$

Un constructeur peut appeler un autre constructeur de la même classe via $this(\dots)$

- $this(\dots)$ Doit être la première instruction d'un constructeur ;
- Le chaînage se termine dans le **constructeur final** – le dernier de la chaîne, celui qui ne réappelle pas $this(\dots)$;
- Les initialisations d'instance s'exécutent une seule fois, avant le corps du constructeur final ; puis on exécute, en remontant, les corps des constructeurs appelants.

Ordre d'initialisation – avec héritage

- Un objet C (avec $C\ extends\ B\ extends\ A$) contient une partie **A**, une partie **B**, puis **C** ;
- A la construction, on initialise du parent vers l'enfant :
  1. **Allocation** de l'objet → Champ à leurs valeurs par défaut $(0, false, null)$;
  2. On démarre par le constructeur le plus dérivé ($C(\dots)$) :
     - S'il commence par $this(\dots)$, on suit la **chaîne interne de C** jusqu'au constructeur de C qui n'appelle pas $this(\dots)$;
     - Ce « dernier » constructeur de C appelle alors $super(\dots)$ (implicite ou explicite).
  3. On monte niveau par niveau : Appel d'un constructeur de **B**, puis de **A**, etc.
  4. Pour chaque niveau, après le $super(\dots)$ du niveau inférieur :
     - Exécuter les **initialisations d'instances** de ce niveau (champs initialisés+ blocs $\{\dots\}$ **dans l'ordre du code**) ;
     - Puis exécuter le **corps du constructeur** de ce niveau.
  5. On redescend jusqu'à C et on termine par ses initialisations d'instance puis le corps de son constructeur.

  A retenir : $this(\dots)$ ou $super(\dots)$ est **toujours la 1e instruction** d'un constructeur ; les initialisations d'instance d'une classe s'exécutent une seule fois, **après** le $super(\dots)$ qui la précède, avant le corps du constructeur de cette classe.

Classes scellées (sealed classes)

- Permettent de contrôler quelles classes peuvent hériter d'une classe.

Syntaxes (exemple) : $public\ sealed\ class\ Shape\ permits\ Circle, Square, Rectangle\{\}$.

- $sealed$: Mot-clé pour sceller la classe ;
- $permits$: Spécifie les sous-classes autorisées

Même si la classe $Shape$ est $public$, les classes qui peuvent l'étendre sont les classes $Circle$, $Rectangle$ et $Square$.

Intérêt : Garantir que les extensions imprévues ne sont pas possibles.

La super-classe $Object$

La classe $Object$

- La classe $Object$ est la racine de la hiérarchie des classes en Java ;
- Chaque classe en Java hérite implicitement de $Object$;
- Fournit des méthodes de base pour toutes les classes (certaines de ces méthodes doivent être redéfinies dans les classes dérivées pour fonctionner correctement).

Méthodes importantes de la classe $Object$

$public\ boolean\ equals(Object\ obj)$
$public\ String\ toString()$
$public\ int\ hashCode()$
$protected\ Object\ clone()\ throwns\ CloneNotSupportedException$
$public\ final\ Class<?>\ \ getClass()$

Méthodes

- **$equals()$:** Cette méthode permet de savoir si deux objets sont égaux
  - L'implémentation par défaut de $equals()$ fournie par la classe Object compare juste l'adresse (référence) de deux objets.

  Quand $equals()$ est redéfinie dans une sous-classe B de A, la classe A devrait tester l'égalité des types effectifs (au lieu d'utiliser $instanceof$).

  Règles pour l'implémentation de la méthode $equals$:
  - Réflexivité : Pour tout objet non $null$, $x.equls(x)$ doit renvoyer $true$;
  - Symétrie : Pour deux objets non $null$, si $x.equals(y)$ renvoie $true$, alors $y.equals(x)$ doit renvoyer $true$;
  - Transitivité : Pour trois objets non $null$, si $x.equals(y)$ et $y.equals(z)$ renvoient $true$, alors $x.equals(z)$ doit renvoyer $true$;
  - Cohérence : Si $x.equals(y)$ renvoie $true$, il doir rester $true$ tant que ni x ni y ne sont modifiés ;
  - Nullité : Pour tout objet non $null$, $x.equals(null)$ doit renvoyer $false$.
- **$toString()$:** Fournit une chaîne de caractères contenant
  - Le nom de la classe concernée ;
  - L'adresse de l'objet en hexadécimal (précédée par @).

  Cast implicite : $System.out.println(p);$ ⇔ $System.out.println(p.toString());$

  Implémentation par défaut dans la classe $Object$:

  $$getClass().getName() + '@' + Integer.toHexString(hashCode())$$
- **$hashCode()$:**
  - $hashCode()$ Est une méthode qui renvoie un code entier dérivé de l'objet. Ce code est principalement utilisé pour le stockage et la recherche dans des structures de données comme les tables de hachage (exemple : $HashMap$, $HashSet$) ;
  - La méthode $hashCode()$ permet de déterminer l'emplacement d'un objet dans une table de hachage. Un bon code de hachage réduit le nombre de collisions, ce qui améliore les performances de recherches et de récupération des données.

  Implémentation par défaut :
  - Dans la classe $Object$, la méthode $hashCode()$ est implémentée pour renvoyer un entier dérivé de l'adresse mémoire de l'objet ;
  - Par défaut, **deux objets distincts auront généralement des codes de hachage différents**.

  Redéfinition de $hashCode()$:
  - Si $equals()$ est redéfinie, il faut également redéfinir $hashCode()$;
  - Si deux objets sont considérés comme **égaux** (selon $equals()$), **leur $hashCode()$ doit être le même**.

- $Objects.hash()$:
  - $Objects.hash()$ est une méthode utilitaire de la classe $java.util.Objects$ qui génère un code de hachage pour plusieurs champs d'un objet ;
  - La méthode prend en argument un nombre variable d'objets et calcule un code de hachage combiné pour ces objets ;
  - Elle simplifie la redéfinition de la méthode $hashCode()$ en calculant le hachage en fonction de plusieurs attributs d'un objet.
    Signature : $public\ static\ int\ hash(Object\ ...values)$
    Paramètres : $Object\ ...values$ – Un tableau d'objets variadiques ($varargs$) : Nombre quelconque d'arguments.

  Redéfinition : Pour hacher un seul champ a :
  - Si a est un objet $a.hashCode()$;
  - Si a est un tableau : $Arrays.hashCode(a)$;
  - Si a est de type primitif int : $Integer.hashCode(a)$ (similaires pour les autres types primitifs).

  Remarque : Sur un tableau a, la méthode $a.hashCode()$ ne renvoie pas un code de hachage basé sur le contenu du tableau, mais plutôt une référence basée sur l'adresse mémoire du tableau.
  - $Arrays.hashCode()$ prend bien en compte les éléments du tableau.
- $\boldsymbol{clone}()$: La méthode $clone()$ est sensée renvoyer une copie d'un objet.
  Implémentation par défaut dans la classe $Object$:
  - Effectue une copie superficielle (ou shallow copy) : elle copie les valeurs de chaque champ (attribut) de l'objet d'origine dans le clone ;
  - Pour les champs primitifs (comme int, $boolean$), les valeurs sont simplement dupliquées ;
  - Pour les objets référencés dans les champs (tableaux, listes, ou autres objets), seule la référence est copiée.

  Accessibilité :
  - La méthode est déclarée $protected$ dans la classe $Object$.
    $$protected\ Object\ clone()\ throws\ CloneNotSupportedException$$
  - Bien que $clone()$ soit présente dans toutes les classes dérivées de $Object$, elle n'est pas accessible directement à l'extérieur de la classe ;
  - Conséquence : Une classe ne peut appeler $clone()$ sur un objet que si elle redéfinit cette méthode.
  - L'implémentation par défaut de $clone()$ dans $Object$ est considérée comme insuffisante pour la majorité des classes personnalisées ;
  - Si la classe possède des champs qui sont des références à des objets, la copie superficielle peut provoquer des problèmes, car le clone et l'objet original partageront ces références, ce qui peut entraîner des comportements inattendus (modifications partagées) ;
  - Le fait que $clone()$ soit $protected$ encourage les développeurs à redéfinir cette méthode dans leurs sous-classes pour s'assurer qu'une copie adaptée (par exemple, une copie profonde dans certains cas) soit effectuée.