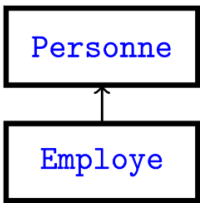
Héritage

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).

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.

Type déclaré et type effectif

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

Type déclaré : $B\ b;$

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

Type effectif : $B\ b = new\ B();$

- 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.

Liaisons :

- **Type effectif :** Méthode dynamique ;
- **Type déclaré :**
  - Méthode statique ;
  - Variable (y compris les paramètres de fonctions).
- 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 :

- **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.
  - Toute instance d'une sous-classe est également une instance de sa super-classe ;
- **Downcasting :**
  - Le downcasting consiste à convertir un objet d'un type de niveau supérieur vers un type dérivé (sous-classe) dans la hiérarchie des classes.
  - 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 ;

Classes abstraites

- Une classe abstraite peut :
  - Peut contenir des méthodes abstraites ou non abstraites ;
  - Peut contenir des attributs.
- Les sous-classes doivent implémenter toutes les méthodes abstraites, sauf si elles sont elles-mêmes abstraites ;
- Utilisée pour le polymorphisme : Une classe abstraite peut être utilisée comme type.

Intérêt :

- Définit des méthodes communes qui peuvent être partagées par plusieurs sous-classes ;
- Imposer l'implémentation de certaines méthodes dans les sous-classes ?
- Avoir un mécanisme de **réutilisation du code** tout en laissant de la **flexibilité** sur certaines méthodes.

Une classe ne peut hériter que **d'une seule classe abstraite**.

Interfaces

- Une interface est une collection de méthodes abstraites (signatures de méthodes) ;
- Une interface peut contenir également des constantes et des classes internes mais **pas de variables d'instance** ;
- A partir de Java 8, les interfaces peuvent contenir des méthodes par défaut et des méthodes statiques ;
- Une classe peut **implémenter** plusieurs interfaces, ce qui permet d'obtenir une forme de **multi-héritage**.

- Différence fondamentale : Une classe peut **hériter** d'une seule autre classe, mais peut **implémenter** plusieurs interfaces, ce qui permet d'obtenir une forme de **multi-héritage**.
- **Utilité des interfaces :**

    - Une interface est une sorte de « contrat » ou « étiquette » qu'une classe peut implémenter. Elle garantit que la classe implémente certaines méthodes définies par cette interface.
    - Une même classe peut implémenter **plusieurs interfaces**, ce qui permet d'ajouter différents comportements ou fonctionnalités.
- **Interfaces et polymorphisme**
    - Une interface définit un type, tout comme une classe ;
    - On peut déclarer des variables ou des paramètres de type interface, mais on ne peut pas instancier un objet de ce type.
- **Méthodes statiques**
    - Les méthodes statiques **doivent toujours avoir une implémentation**, même dans une interface.
- **Méthodes privées**
    - Les méthodes privées sont visibles uniquement au sein de l'interface, ce qui signifie qu'elles ne peuvent être appelées que par d'autres méthodes de l'interface ayant une définition (ex. méthodes par défaut ou statiques).
    - Elles doivent obligatoirement être définies (avoir un corps) ;
    - Elles peuvent être statiques ou non statiques.
- **Méthodes par défaut**
    - Les méthodes avec une définition par défaut doivent utiliser le modificateur $default$ (sauf si elles sont $private$ ou $static$).
    - Les classes qui implémentent l'interface héritent des méthodes par défaut, ce qui signifie qu'elles ne sont pas obligées de les redéfinir ;

Classes internes

Les différents types de classes internes :

1. Classes membres internes :
    - Classes internes définies comme membres d'une autre classe ;
    - Elles peuvent être déclarées comme **statiques** ou non.
2. Classes locales :
    - Classes définies à l'intérieur d'un boc de code, comme dans une méthode, un constructeur ou un bloc de code spécifique ;
    - Elles sont visibles uniquement dans le bloc où elles sont définies.
3. Classes anonymes :
    - Classes internes locales sans nom, souvent utilisées pour fournir une implémentation immédiate d'une interface ou d'une classe abstraite.
        - Elles sont définies et instanciées en une seule expression ;
        - Utilisées fréquemment dans les gestionnaires d'évènements.

Classe membre non statique : L'objet englobant

- Un objet d'une classe membre non-statique ne peut être créé qu'à partir d'une instance de la classe englobante ;
- L'objet de la classe membre possède automatiquement une référence implicite vers l'objet de la classe englobante.

- L'objet de la classe englobante est accessible depuis l'objet de la classe interne ;
- Dans la classe interne, on peut faire référence à l'objet de la classe englobante en utilisant : $NomClasseEnglobante.this$.

Classe membre non-statique :

Membres : Les membres statiques ne sont permis que dans des classes membres statiques.

Création d'objets d'une classe membre interne

- Si une classe membre est visible en dehors de sa classe englobante, elle est dénotée comme suit : $NomClasseEnglobante.NomClasseMembre$;
- Cela signifie qu'il est possible de créer une instance de la classe membre interne à partir de n'importe quel objet de la classe englobante.

Classes membres statiques

- Une classe membre statique est définie à l'intérieur d'une autre classe avec le mot-clé $static$;
- Comme les champs et méthodes statiques, une classe membre statique n'est pas associée à une instance de la classe englobante ;
- Cela signifie que la classe membre statique n'a pas accès aux membres d'instance de la classe englobante, mais uniquement aux membres statiques de cette dernière.

- **Accès aux membres d'instance et aux membres statiques**

  Les classes statiques **ne sont pas liées à une instance de la classe englobante**, donc elles ne peuvent pas accéder aux membres d'instance comme a.

  Mais elles ne peuvent accéder aux champs statiques de la classe englobante.

Classes locales

- Les classes locales sont des classes définies à l'intérieur d'un bloc de code (comme une méthode ou un constructeur) ;
- Elles sont similaires aux variables locales : Elles n'existent que dans le bloc de code où elles sont définies.

Caractéristiques :

- Non-membre de la classe englobante : Une classe locale n'est pas un membre de la classe qui l'entoure, contrairement aux classes internes classiques ;
- Visibilité limitée : Une classe locale est visible uniquement dans le bloc de code dans lequel elle est définie. Elle ne peut pas être utilisée en dehors de ce bloc.

Classes anonymes :

Une classe anonyme est une classe locale sans nom qui permet de créer un objet à la volée.

Expression lambda

- L'opérateur lambda ou **opérateur flèche** est $->$;
- Il divise une expression lambda en deux parties :
  1. A gauche : Spécifie les paramètre requis ;
  2. A droite : Spécifie les actions de l'expression lambda.

Java définit **deux types de corps** pour un lambda :

1. Un corps constitué d'une expression unique ;
2. Un corps constitué d'un bloc de code.

Références de constructeur

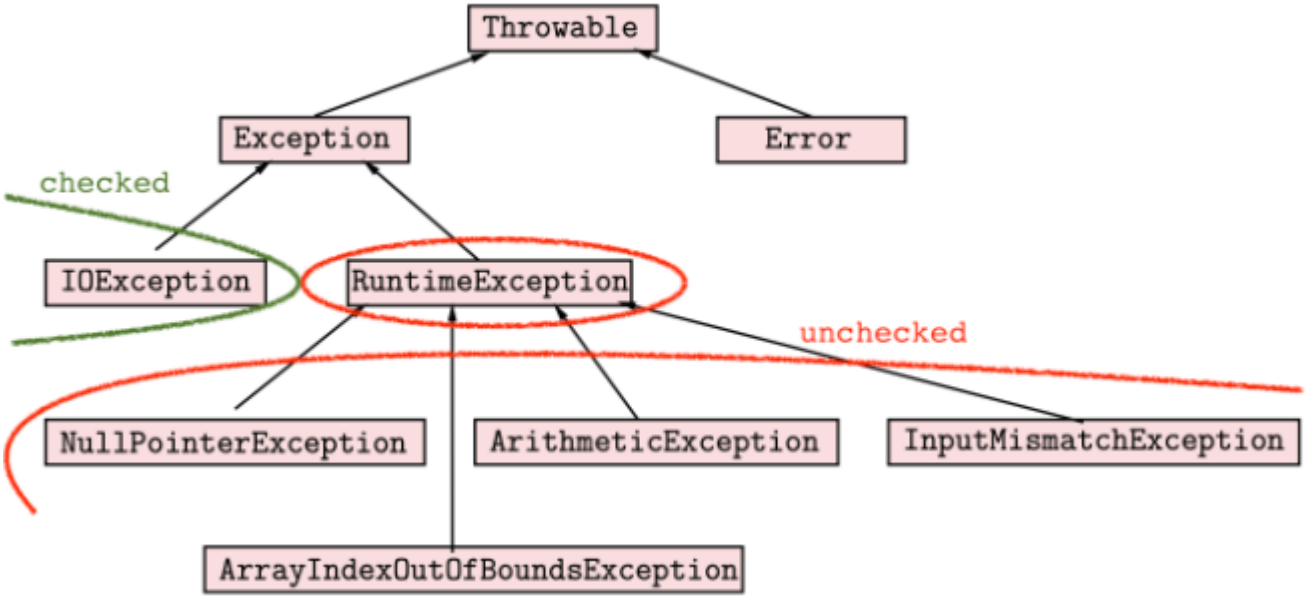
- En Java, $NomClasse :: new$ est une référence à un constructeur compatible avec la signature d'une **interface fonctionnelle** ;
- Cela permet d'appeler un constructeur existant sans écrire explicitement une expression lambda ;
- Syntaxe : $NomClasse :: new$ fait référence à un constructeur de $NomClasse$.

Exceptions

Mécanisme $try - catch$

Le bloc $try - catch$ est un mécanisme qui permet de gérer les exceptions qui peuvent se produire pendant l'exécution d'un programme, sans que ce dernier ne se termine brutalement.

- Bloc try : Contient le code qui peut potentiellement lever une exception ;
- Bloc $catch$: Capture et gère l'exception si elle est levée dans le bloc try.



Le mot-clé $throws$

- Le mot-clé $throws$ est utilisé dans la signature d'une méthode pour indiquer que cette méthode peut potentiellement lancer des exceptions ;
  Il est utilisé pour propager des exceptions à la méthode appelante au lieu de les gérer localement avec un bloc $try - catch$.
- Si une méthode déclare une exception dans $throws$, elle ne gère pas l'exception directement ;
- L'appelant de cette méthode doit gérer l'exception ou déclarer qu'il la lance à son tour.

Le mot-clé $throw$

- Il est possible de lever une exception manuellement (qu'elle soit **checked** ou unchecked) lorsqu'on détecte une situation d'erreur spécifique) ;

- Le mot-clé $throw$ permet de lancer une exception manuellement dans le programme ;
- Utilisé lorsqu'une condition anormale ou une erreur est rencontrée et qu'on souhaite signaler cette erreur pour qu'elle soit gérée ailleurs.

$throw\ new\ ExceptionType("Message\ d'erreur");$

La clause $finally$

- La clause $finally$ peut être ajoutée à un bloc $try - catch$;
- Elle contient du code qui sera exécuté dans tous les cas après le bloc $try - catch$, quelle que soit la manière dont on sort du bloc.

Généricité

Généricité et covariance

Les types génériques ne sont pas covariants.
Si A est un **sous-type** de B, alors $C < A >$ **n'est pas un sous type** de $C < B >$!

Type anonyme ($Wildcard$)

- Le wildcard ? représente un type inconnu ;
- Il permet d'utiliser une classe générique sans préciser le type exact de ses éléments ;
- Utilisé lorsque le type concret n'a pas d'importance dans le contexte.

Compatibilité : N'importe quel type (sous-type de $Object$) peut être utilisé avec ?.

Wildcards avec bornes

Les wildscards (?) peuvent également être bornés, mais avec des **restrictions** :

- Un seul type de borne est autorisé : Une borne supérieure ($extends$) ou une borne inférieure ($super$) mais **pas les deux à la fois** ;
- Pas de combinaison avec plusieurs bornes (par exemple, pas de & comme pour les types génériques classiques).

- **Wildcard avec borne supérieur ($extends$)** : $List <? extends\ T >$
- **Wildcard avec borne inférieure ($super$)** : $List <? super\ T >$

Enumérations

Enumérations

- Une énumération est une liste de constantes nommées qui définit un nouveau type de données ;
- Un objet d'un type énumération ne peut prendre qu'une des valeurs prédéfinies dans cette énumération ;
- Les énumérations permettent de représenter un ensemble fixe de valeurs valides.