Classes abstraites

Classes abstraites

- Une classe abstraite est une classe qui **ne peut pas être instanciée** directement ;
- Il est impossible de créer des objets d'une classe abstraite ;
- Elle sert de classe de base pour une dérivation.

Une classe abstraite **se déclare** ainsi : $abstract\ class\ A\{\}$

Méthodes abstraites

Dans une classe abstraite, on peut trouver des champs et méthodes classiques, dont héritera toute sous-classe dérivée.
Mais on peut aussi trouver des méthodes, dites abstraites :

- Les méthodes abstraites sont des méthodes sans corps, seule la signature et le type de la valeur de retour sont spécifiés ;
- Leur implémentation doit être faite dans les sous-classes ;
- Ces méthodes définissent des comportements qui doivent être implémentés par les classes dérivées.

Une classe qui comporte une méthode abstraite **est considérée comme abstraite**.

- **Quelques règles**
  - Une classe qui comporte une ou plusieurs méthodes abstraites est abstraite, même si l'on n'indique pas le mot clé $abstract$:

    Il est correct d'écrire : $class\ A\{\ public\ abstract\ void\ f(\ )\}$

    Il est cependant vivement conseillé d'utiliser le mot-clé $abstract$ pour ces classes.
  - Une méthode abstraite doit obligatoirement être déclarée $public$ (ou $protected$) car sa vocation est d'être redéfinie dans les classes dérivées ;
  - Dans l'en-tête d'une méthode déclarée abstraite, les noms d'arguments doivent figurer (bien qu'ils ne servent à rien) :

    $abstract\ class\ A\{$

    $public\ abstract\ void\ g(int\ k)$ A écrire à la place d'en haut $\}$

Classes abstraites et constructeurs

- Les classes abstraites peuvent avoir des constructeurs :
  - Bien qu'une classe ne puisse pas être instanciée directement, ses constructeurs sont utilisés par les sous-classes.
- Objectif du constructeur dans une classe abstraites :
  - Initialiser les variables d'instance communes aux sous-classes ;
  - Permettre aux sous-classes de réutiliser l'initialisation.

Classes abstraites et instanciation :

Soit une classe abstraite A :

- Il est possible de déclarer une variable de type A ;
- Cependant, toute instanciation d'un objet de type A sera rejetée par le compilateur ;

- Supposons que B dérive de A et définit la méthode abstraite g :
  Il est possible d'instancier un objet de type B et même affecter sa référence à une variable de type A : $A\ a = new\ B(\dots);$

Dérivation de classes abstraites

- Une classe dérivée d'une classe abstraite n'est pas obligée de (re)définir toutes les méthodes abstraites de sa classe de base ;
- Une telle classe dérivée **reste simplement abstraite** (et il est conseillé de mentionner $abstract$ dans sa déclaration).

Remarque :

- Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et/ou contenir des méthodes abstraites ;
- Toutes les classes abstraites définies ici dérivent implicitement de la classe $Obejct$ qui n'est pas abstraite.

Conclusion : Caractéristiques et intérêt des 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

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.

Code générique avec des objets comparables

- On peut écrire du code générique qui fonctionne avec des objets comparables, sans avoir besoin de savoir à quelle classe ces objets appartiennent ;
- Cela permet de manipuler des objets de différentes classes, tant qu'ils implémentent l'interface $Comparable$.

L'interface $Cloneable$

- Pour pouvoir redéfinir la méthode $clone()$ de la classe $Object$, il est nécessaire d'implémenter l'interface $Cloneable$;
- L'interface $Cloneable$ ne contient aucune méthode : C'est une interface marquante utilisée uniquement pour indiquer qu'une classe permet le clonage de ses objets via la méthode $clone()$.

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.

Polymorphisme

Implémenter une interface fournit le polymorphisme par le type de l'interface : on peut passer un objet partout où ce type d'interface est attendu.

Méthodes publiques

Visibilité des méthodes :

- Les méthodes d'une interface sont automatiquement publiques, il n'est donc pas nécessaire de spécifier le modificateur $public$;
- En revanche, les classes qui implémentent l'interface doivent utiliser le modificateur $public$ pour les méthodes redéfinies afin de respecter la visibilité de l'interface.

Méthodes statiques

- Depuis Java 8, les interfaces peuvent également contenir des méthodes statiques ;
- Les méthodes statiques **doivent toujours avoir une implémentation**, même dans une interface.

- Intérêt :
  - Cohésion : Regrouper les fonctions liées au contrat **dans l'interface** elle-même, **sans créer d'objet** ;
  - Moins de « classes utilitaire » : Eviter des classes artificielles qui ne servent qu'à héberger des méthodes statiques ;

- **API plus claire :** Appel explicite via $NomInterface.methode(\dots)$, rendant l'usage **visible** et **organisé**.

Méthodes privées

- A partir de Java 9, il est possible de définir des méthodes privées dans les interfaces ;
- 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).

Utilité :

- Les méthodes privées agissent comme des méthodes « helper » (ou utilitaires) pour éviter de dupliquer du code dans l'interface.
- Elles doivent obligatoirement être définies (avoir un corps) ;
- Elles peuvent être statiques ou non statiques.

Méthodes par défaut

- Depuis Java 8, les interfaces peuvent fournir une **définition par défaut** pour certaines méthodes ;
- Les méthodes avec une définition par défaut doivent utiliser le modificateur $default$ (sauf si elles sont $private$ ou $static$).

- **Motivation derrière l'introduction des méthodes par défaut :**

  Extension des interfaces sans casser le code :

  - Avant les méthodes par défaut, ajouter une nouvelle méthode à une interface largement utilisée aurait cassé le code existant, car il aurait manqué une implémentation pour la nouvelle méthode ;
  - Les méthodes par défaut fournissent une implémentation par défaut si aucune autre implémentation n'est spécifiée, évitant ainsi les problèmes de compatibilité.

  Méthodes optionnelles :

  - Les méthodes par défaut permettent de spécifier des méthodes optionnelles dans une interface, selon son utilisation.

- **Héritage :**
  - Les classes qui implémentent l'interface héritent des méthodes par défaut, ce qui signifie qu'elles ne sont pas obligées de les redéfinir ;
  - **Redéfinition possible** : Les classes peuvent toujours surcharger (Override) ces méthodes pour fournir une implémentation spécifique.
- **Utilisation d'autres méthodes**
  - Une méthode par défaut peut appeler d'autres méthodes ayant une définition, y compris :
    - Méthodes privées ;
    - Autres méthodes par défaut ;
    - Méthodes statiques, mais uniquement via le nom de l'interface.
- **Utilisation de méthodes abstraites dans une méthodes par défaut**
  - Les méthodes par défaut peuvent également appeler des méthodes abstraites !
  - Les méthodes abstraites doivent être implémentées par les classes ; la méthode par défaut s'appuie dessus et utilisera automatiquement leur implémentation à l'exécution.

- **Champs dans une interface**
  - Les champs d'une interface sont automatiquement $public\ static\ final$, même si ces modificateurs ne sont pas spécifiés ;
  - Une interface ne peut pas avoir de champs d'instance.
- **Accessibilité des champs**
  - Les champs définis dans une interface sont accessibles par les classes qui l'implémentent, mais ils sont constants (ne peuvent pas être modifiés) ;

$class\ D\ implements\ I\{\}$

  - $D.c$ Est champ $public\ static\ final$ de la clase D qui implémente I ;
  - $I.c$ Fait référence au même champ.
- **Utilité des constantes dans les interfaces**
  - Les grandes applications utilisent souvent de nombreuses valeurs constantes (ex. tailles de tableaux, limites, valeurs spéciales) ;
  - Les variables d'interface offrent un moyen pratique de rendre ces constantes disponibles dans plusieurs fichiers source ;

Comment définir des constantes partagées

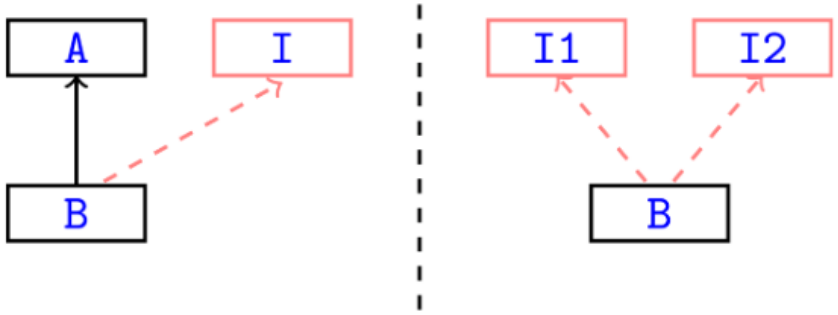
  - Pour définir un ensemble de constantes partagées, on peut créer une interface qui ne contient que ces constantes, sans méthodes ;
  - Les classes qui ont besoin d'accéder aux constantes peuvent simplement implémenter l'interface, ce qui rend les constantes accessibles.
- **Redéfinition (Overriding) des méthodes**
  - Une classe qui implémente une interface doit retenir toutes les méthodes abstraites (c'est-à-dire celles sans corps) ;
  - Si une classe ne redéfinit pas toutes les méthodes abstraites, elle doit être déclarée **abstraites**.

Résumé

  - Les champs d'une interface sont des constantes accessibles via l'interface ou les classes qui l'implémentent ($I.c$ ou $D.c$) ;
  - Les méthodes abstraites **doivent être redéfinies** par les classes concrètes, sinon la classe doit être déclarée abstraite ;
  - Les méthodes par défaut peuvent être redéfinies mais ce n'est pas obligatoire.

Héritage multiple

  - Des conflits de définitions peuvent être générés par l'héritage multiple.



  - Qu'est ce qui se passe si A et I ou bien $I1$ et $I2$ **définissent la même constante**, ou la **même méthode** (même signature) ?

Conflits des constantes

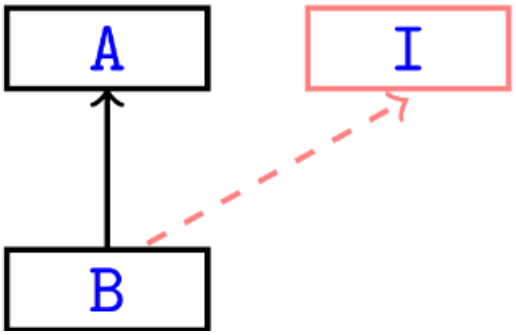
Problème :

- Lorsqu'une sous-classe hérite de plusieurs interfaces ou classes qui définissent des constantes identiques, il peut y avoir un conflit de nom ;
- Toutes les constantes conflictuelles sont **héritées dans la sous-classe**, ce qui peut entraîner des ambiguïtés.

Solution :

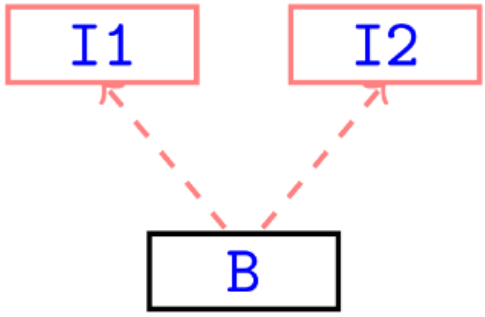
- Pour résoudre l'ambiguïté, il faut utiliser la notation $NomClasse.nomConstante$ pour accéder à la constante souhaitée ;
- Cette notation est similaire à celle utilisée pour accéder aux **champs statiques**.

Conflit de méthodes



Premier cas : Conflit entre classe et interface

- Si une classe et une interface héritée possèdent une méthode avec la même signature, la classe prend toujours la priorité ;
- La sous-classe hérite de la méthode définie dans la classe parente, et toute définition de la même méthode dans l'interface est **ignorée**.



Deuxième cas : Conflit entre interfaces

- Si deux interfaces héritées définissent des méthodes avec la même signature et qu'au moins l'une d'elles a une implémentation par défaut, la classe qui les implémente par défaut, la classe qui les implémente doit redéfinir la méthode pour résoudre le conflit ;
- Cela est vrai même si m'une des méthodes est abstraite (sans corps).

Résoudre les conflits de méthodes entre interfaces

- La classe peut choisir d'appeler l'implémentation par défaut d'une des interfaces à l'aide de la syntaxe $NomInterface.super.methode()$.

Conflits entre méthodes abstraites (même nom/paramètres)

Règle générale : Si deux interfaces héritées déclarent une méthode $f(\dots)$ **avec la même signature** (même nom et mêmes paramètres) mais **des types de retour différents** :

- Pas de conflit si l'un des types de retour est un **sous-type** de l'autre ;
- Conflit si les types de retour sont **incomparables** (aucune relation de sous-typages).

- Cas 1 – Pas de conflits :
  - Ici, $Number \subset Object$. La méthode la plus spécifique gagne : dans B, $f()$ est de type $Number$;
  - Remarque : B peut redéfinir $f()$ (optionnel) ; si elle ne le fait pas, elle reste abstraite.
- Cas 2 – Retours incompatibles (conflits) :
  - Aucun des deux types n'est sous type de l'autre → Conflit irrésoluble.

Héritage d'interface

- Tout comme les classes, une interface peut étendre une ou plusieurs autres interfaces ;
- Cela permet de **combiner les comportements** et de les regrouper dans une seule interface.

Héritage multiple d'interface

- Une interface peut étendre plusieurs interfaces, ce qui permet de combiner différents comportements.
- Les règles de **résolution des conflits** pour les constantes ou les méthodes sont les mêmes que pour l'implémentation de plusieurs interfaces dans une classe.

Classes internes

Introduction aux classes internes en Java

Une classe interne est une classe définie à l'intérieur d'une autre classe :

- Elles peuvent accéder aux membres (y compris privés) de la classe externe qui les encapsule ;
- Les classes internes améliorent l'organisation du code en regroupant les classes qui ne sont utilisées que par une autre classe, ce qui permet de les garder ensemble et de clarifier leur relation.

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

- Une classe membres non-statique est une classe définie à l'intérieur d'une autre classe ;
- Elle est liée à une instance de la classe englobante et peut accéder à ses membres y compris les membres privés.

Instance de la classe interne

- Une classe interne est liée à une instance de la classe englobante, mais elle ne crée pas automatiquement un objet de cette classe interne à chaque fois que la classe englobante est instanciée ;
- Pour manipuler un objet de la classe interne ($Cursor$), il faut **explicitement instancier cette classe**.

Utilité des classes membres internes

1. Relation logique avec la classe englobante :
   - Une classe membre interne a du sens uniquement dans le contexte de la classe qui l'englobe.
2. Encapsulation et limitation de la visibilité :
   - Lorsque la classe membre interne est déclarée $private$, elle limite la visibilité de la classe membre uniquement à la classe englobante ;
   - Cela renforce l'encapsulation et cache la complexité interne aux autres classes.

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.

Accès à l'objet englobant depuis l'objet de la classe interne

- 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 : $NomClasseEnglobante.this$.

Masquage de variables et accès à la classe externe

- Masquage de variable : La classe interne possède un champ $message$ qui masque celui de la classe externe. Une référence à $message$ dans la classe interne, fait accéder au champ de $ClasseInterne$;
- Accès explicite à la classe externe : Pour accéder au champ $message$ de $ClasseExterne$ à l'intérieur de la classe interne, on doit utiliser $ClasseExterne.this.message$.

Classe membre non-statique :

- **Membres :** Les membres statiques ne sont permis que dans des classes membres statiques.
- **Contrôle d'accès**

  La **classe membre** a accès aux autres membres de la classe englobante indépendamment de leur modificateur d'accès ;

  La **classe englobante** a accès à ses classes membres, ainsi qu'à leurs champs et méthodes, indépendamment du modificateur d'accès.

Héritage et visibilité dans les classes membres internes

Visibilité dans le cadre de l'héritage classique

- Une sous-classe n'a pas accès aux membres privés de sa classe mère.

Visibilité et héritage dans le cadre des classes internes

- Cela semble être pareil dans le cas d'héritage des classes internes.
- Le **comportement spécial des classes internes** permet à une classe interne d'accéder aux membres privés d'une autre classe interne parente
- Cette exception s'applique uniquement aux classes internes et est due au fait que Java autorise un accès plus flexible entre les membres privés des classes imbriquées.

Visibilité en dehors de la classe englobante

Le modificateur d'accès d'une classe membre et de ses champs et méthodes règle uniquement leur visibilité en dehors de la classe englobante.

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.

Interface membres

- Une interface peut aussi être membre d'une classe ;
- Une interface membre est implicitement considérée comme $static$, même si le mot clé $static$ n'est pas spécifié ;
- Cela signifie qu'une interface membre n'est pas liée à une instance spécifique de la classe englobante.

Utilité :

- Organiser des interfaces liées à une classe spécifique.

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.

Création d'un objet de la classe membre statique

Dans une méthode d'une autre classe : $A.B\ f = new\ A.B()$ // Création d'une instance de B sans avoir besoin d'une instance de A.

Une classe membre statique peut être instanciée sans avoir besoin d'une instance de la classe englobante A.

- Cela est possible car une classe membre statique est indépendante de toute instance de la classe englobante.

Pourquoi une classe interne statique ?

- Indépendance : La classe $Noeud$ n'a pas besoin d'accéder aux membres de la classe $Pile$, donc elle est déclarée pour refléter cette indépendance ;
- Efficacité : L'absence de lien avec une instance de $Pile$ signifie que chaque instance de $Noeud$ occupe moins de mémoire, car elle n'a pas de référence implicite à l'instance de Stack ;
- Organisation : En regroupant $Noeud$ à l'intérieur de $Pile$, on garde un e structure de code logique, mais en utilisant le mot clé $static$, on évite une dépendance inutile.

Hiérarchie des noms

- On utilise les noms complets pour accéder aux classes internes :

$$Pile.Noeud\ n = new\ Pile.Noeud();$$

Classe membre statique : Accès

- **Explications :**
  - Les classes membres statiques peuvent être instanciées directement sans avoir besoin d'une instance de la classe englobante. Elles sont indépendantes de toute instance.
  - Les classes membres non-statiques sont liées à une instance de la classe englobante.

Classe membre et héritage

Il existe différentes façons d'étendre une classe membre (statique ou non) :

- **Première façon : Directement dans la classe**
  - $Interne$ est une classe membre non-statique de la classe $Externe$;
  - Cela signifie que pour instancier $Interne$, une instance de $Externe$ est nécessaire ;
  - La syntaxe pour étendre une classe membre est **la même** que pour étendre toute autre classe en Java ;
  - $InterneEntendu$ Hérite de $Interne$ et peut utiliser ou redéfinir les membres et méthode de $Interne$.
- **Deuxième façon** : **Dans une autre classe**
  - Tous les membres de $Externe$, y compris $Interne$, sont hérités par $ExterneEtendue$;
  - Il est possible d'étendre la classe membre interne héritée $Interne$ comme vous le feriez pour n'importe quel autre membre. Cela permet de créer une version étendue de $Interne$, appelée $InterneEtendue$, qui hérite des fonctionnalités de $Interne$.
- **Troisième façon :** Une classe interne peut également être étendue en dehors de sa classe englobante.
  - Cas de classe interne statique :
    Pas besoin d'un objet $Externe$ pour créer un $InterneStatique$.
  - Cas de classe interne non-statique : Une classe interne non-statique peut être étendue en dehors de sa classe englobante, mais elle doit être associée à un objet de la classe englobante.
    Raison : Un objet $Interne$ (ou d'une de ses extensions) n'a de sens qu'attaché à un objet $Externe$.

Classe membre et occultation des champs

- Les **champs** des classes internes et les paramètres de méthodes peuvent occulter (cacher) les champs de la classe externe s'ils portent le même nom ;

- Cependant, il est toujours possible d'accéder aux champs occultés de la classe externe.

Classe membre et occultation des méthodes

- Les classes internes peuvent également occulter les **méthodes** de la classe externe si elles utilisent des méthodes du même nom.

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.

Usage principal :

- Les classes locales sont utiles lorsqu'il faut créer des objets dont l'existence n'a de sens que localement dans un bloc de code ;
- Cela inclut des situations spécifiques comme les interfaces graphiques, où des objets temporaires ou des extensions locales de classes sont nécessaires.

- **Accès aux variables locales**
  - La classe locale $Iter$ est définie dans la méthode $parcourir( )$ pour implémenter l'interface $Iterator < Object >$;
  - Cette classe locale n'est visible que dans la méthode $parcourir( )$ et n'est accessible nulle part ailleurs.

Une classe locale peut accéder :

- Aux membres de la classe englobante (par exemple, le tableau $data[ ]$) ;
- Aux variables locales de la méthode dans laquelle elle est définie (ici, up).

Classes anonymes :

- **Motivation**
  - Parfois, il n'est pas nécessaire de donner un nom à une clase qui ne sert qu'à créer un seul objet ;
  - Par exemple, dans une méthode comme $parcourir( )$, on peut renvoyer une classe anonyme qui implémente l'interface $Iterator$.
- **Définition**

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

- Créer un objet d'une classe anonyme qui hérite d'une classe/interface $TypeDeBase$:

$$new\ TypeDeBase(paramètres\ de\ constructeur)\{$$
$$//champs\ et\ méthodes\ additionnels\ de\ la\ classe\ anonyme\};$$

- **Limites**
  - Une classe anonyme ne peut pas avoir de constructeur car elle n'a pas de nom ;
  - Pour initialiser la classe mère, on passe des paramètres au constructeur de la classe de base via new ;
  - Les champs additionnels de la classe anonyme sont initialisés via des initialiseurs ou dans des blocs d'initialisation.

Expression lambda

Expressions lambda en Java

- Introduites dans *Java 8*, les expressions lambda permettent de définir un bloc de code qui peut être affecté à une variable ou passé en paramètre, pour être exécuté plus tard.
- On construit un objet « bouton » et on lui passe la fonction qui sera exécutée quand l'utilisateur cliquera sur le bouton.

Introduction aux expressions lambdas

- L'expression lambda introduit une nouvelle syntaxe et un nouvel opérateur dans le langage Java ;
- 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.

Exemples de lambda ayant un corps

- Exemple 1 : Expression lambda qui renvoie une constante : $()->3$
  Méthode associé : $int\ maMethode()\{\ return\ 3;\}$
- Exemple 2 : Expression lambda avec une expression mathématique :
  $()->Math.random()*50$
- Exemple 3 : Expression lambda avec paramètre : $(x)->1.0/x$
- Exemple 4 : Expression lambda avec retour booléen : $(x)->(x\%2)==0$
  - Ce lambda renvoie $true$ si x est pair, et $false$ sinon.

Si ce lambda a un seul paramètre, il n'est pas nécessaire de l'entourer de parenthèses.

Lambdas simples : Conclusion

- Un lambda simple a un corps d'expression : une seule expression après $->$, sans accolades et sans $return$ (le résultat est implicite) ;
- Les types des paramètres sont le plus souvent inférés ; avec un seul paramètre, les parenthèses sont facultatives.

Limitation des lambdas avec corps d'expression

- Les lambdas avec corps d'expression sont pratiques, mais parfois insuffisants ;
- Il peut être nécessaire d'effectuer plusieurs opérations au sein de lambda.

Par exemple :

- Déclarer des variables locales ;
- Utiliser des boules ou des conditions ;
- Gérer plusieurs étapes avant de retourner un résultat.

Expressions lambda : Corps de bloc

- Les lambdas avec corps de bloc permettent d'avoir plusieurs instructions dans le corps de lambda ;
- Contrairement aux lambdas simples, le corps de bloc est défini entre accolades { } ;
- Cela permet d'intégrer des structures de contrôle comme des **boucles**, des **conditions** $\boldsymbol{if}$, etc.

Utilité des expressions lambdas

- Les expressions lambdas permettent de passer des fonctions en paramètre, une approche similaire aux langages fonctionnels ;
- Avant *Java 8*, il était possible de passer du code en paramètre en créant un objet contenant ce code dans une méthode. Cependant, cette approche était lourde et artificielle (même avec des classes anonymes) ;
- Les expressions lambdas ont été introduites pour simplifier et rendre cette tâche plus directe.

Interfaces fonctionnelles

Une interface fonctionnelle est une interface qui ne possède qu'une seule méthode abstraite.

Référence de méthode :

- Parfois, la fonction que l'on souhaite passer en paramètre existe déjà ;
- Au lieu de définir une nouvelle expression lambda, on peut passer une référence à la méthode existante.

Syntaxe :

- $Classe :: méthode$: Référence à une méthode statique ou d'instance d'une classe ;
- $objet :: méthode$: Référence à une méthode d'instance de l'objet.

Exemple :

$Math :: pow$	Equivalent à $(x, y) -> Math.pow(x, y)$
$System.out :: prinln$	Equivalent à $s -> System.out.println(s)$
$String :: concat$	Equivalent à $(x, y) -> x.concat(y)$

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 $NomClasse$.

Capture des variables locales

Une expression lambda accès :

- Aux champs et méthodes de la classe englobante, indépendamment de leur modificateur d'accès ;
- Aux variables locales visibles dans le bloc englobant, si elles sont « effectively final » (non modifiées après leur initialisation).

Ce phénomène est appelé captures des variables locales.

Différence entre lambdas et classes anonymes

- Classes anonymes :
  - Plus verbeuses, permettent d'implémenter plusieurs méthodes ;
  - Utilisables pour étendre une classe ou implémenter plusieurs méthodes (si héritage) ;
  - Créent une **nouvelle instance** de classe avec un accès explicite à $this$.
- Expressions lambda :
  - Syntaxe plus **concise** et lisible ;
  - Valable uniquement pour les interfaces fonctionnelles (une seule méthode abstraite) ;
  - Ne créent pas Ne créent pas de nouvelle instance de classe, capturent automatiquement les variables locales.

Règle pratique :

- Utiliser les lambda pour simplifier le code lorsqu'on travaille avec des interfaces fonctionnelles ;
- Utiliser les classes anonymes lorsqu'on a besoin d'une implémentation plus complète ou d'accéder à plusieurs méthodes.