

Programmation Orientée Objet

Nicolas Hendrikx<n.hendrikx@helmo.be>

Table des matières

| | |
|--------------------------------------------------------------------------|----|
| Avant-propos | 1 |
| Conventions de l'ouvrage | 2 |
| 1. Les fonctions et leurs limites | 3 |
| 1.1. Les fonctions sont des blocs de code réutilisables | 3 |
| 1.2. Les fonctions se déclarent | 6 |
| 1.3. Les fonctions doivent être appelées | 8 |
| 1.4. Les fonctions ont des limites | 11 |
| 1.5. Les fonctions pures sont sans effets de bord | 17 |
| 1.6. Les fonctions peuvent recevoir un nombre variable d'arguments | 18 |
| 1.7. En résumé | 20 |
| 2. Les objets et les méthodes d'objet | 22 |
| 2.1. Les objets sont les valeurs d'un type | 23 |
| 2.2. Les objets ont des comportements | 25 |
| 2.3. Les objets se manipulent par référence | 29 |
| 2.4. Les objets doivent être créés | 35 |
| 2.5. Les objets possèdent un état | 37 |
| 2.6. Les méthodes de fabrique créent des objets | 39 |
| 2.7. Les objets outrepassent les limites des fonctions | 39 |
| 2.8. En résumé | 42 |
| 3. Les classes et les enums | 44 |
| 3.1. Les classes doivent être déclarées | 44 |
| 3.2. Les classes déclarent des méthodes d'objet | 46 |
| 3.3. Les classes déclarent des champs | 48 |
| 3.4. Les constructeurs initialisent les objets | 52 |
| 3.5. Les méthodes admettent des structures de contrôle | 54 |
| 3.6. Les redéfinitions masquent les méthodes héritées | 57 |
| 3.7. Les champs doivent être encapsulés | 63 |
| 3.8. Les enums définissent des objets par extension | 65 |
| 3.9. En résumé | 71 |
| 4. Le cycle de vie des objets | 73 |
| 4.1. Le constructeur initialise un objet | 73 |
| 4.2. this référence l'objet qui reçoit l'appel | 78 |
| 4.3. Les méthodes de fabrique statique masquent les constructeurs | 81 |
| 4.4. Les champs sont initialisés dans un ordre prédéfini | 82 |
| 4.5. Les objets peuvent être détruits | 85 |
| 4.6. En résumé | 88 |
| 5. Les relations de composition et d'héritage | 90 |
| 5.1. La composition assemble des objets | 90 |

| | |
|-----------------------------------------------------------------------------------------------|-----|
| 5.2. L'héritage définit des sous-classes | 101 |
| 5.3. La clause <code>extends</code> définit une relation d'héritage direct | 104 |
| 5.4. <code>super</code> fait référence à la partie héritée | 107 |
| 5.5. Une surcharge n'est pas une redéfinition | 110 |
| 5.6. Les affectations ascendantes sont implicites | 113 |
| 5.7. Le modificateur <code>final</code> empêche l'héritage | 116 |
| 5.8. La composition peut exploiter l'héritage | 119 |
| 5.9. En résumé | 122 |
| 6. Le polymorphisme | 124 |
| 6.1. Il existe plusieurs sortes de polymorphisme | 124 |
| 6.2. Le polymorphisme ad hoc choisit la bonne surcharge | 125 |
| 6.3. Le polymorphisme générique paramètre un type avec d'autres types | 128 |
| 6.4. Le polymorphisme par sous-typage choisit la version d'une méthode à exécuter | 133 |
| 6.5. Le polymorphisme par sous-typage demande un soin particulier pour l'initialisation | 136 |
| 6.6. Une relation d'héritage correcte respecte les contrats des ancêtres | 139 |
| 6.7. Les affectations descendantes sont dangereuses | 143 |
| 6.8. En résumé | 147 |
| 7. Les classes abstraites et les interfaces | 149 |
| 7.1. Les classes abstraites ne sont pas concrètes | 149 |
| 7.2. Les interfaces déclarent un contrat | 156 |
| 7.3. Vous devriez programmer avec des interfaces | 163 |
| 7.4. En résumé | 166 |

Avant-propos

Le présent syllabus accompagne l'activité d'apprentissage Programmation Orientée Objet (POO) de l'unité d'enseignement Programmation intermédiaire (PrI). Il s'adresse aux étudiants du Bloc 1 du cursus en Développement d'Application. Le syllabus étudie les concepts et les méthodes de la programmation orientée objet et vise à vous rendre capable de :

1. Développer une application écrite en Java 21 basée sur les principes de la POO.
2. Concevoir les classes et les objets qui composent votre application pour minimiser le couplage et maximiser la cohésion.
3. Valider des modules programmés à l'aide de tests unitaires écrits avec JUnit 5.

Il part du principe que vous programmez de façon procédurale en Java 21 dans un environnement de développement intégré comme Eclipse. Il suppose aussi les termes variable, fonction, paramètre, structure de contrôle et tableau n'ont aucun secret pour vous.

Deux parties composent le syllabus. Les chapitres 1 à 7 forment la première partie et sont consacrés aux briques fondamentales de la POO :

- Le **Chapitre 1** rappelle la notion de fonction et expose quelques limitations des fonctions Java. Il montre comment passer outre ces limitations.
- Le **Chapitre 2** introduit la notion d'objet et leurs caractéristiques fondamentales. À la fin de ce chapitre, les notions d'identité, d'état et de comportement d'un objet n'auront plus de secrets pour vous.
- Le **Chapitre 3** vous explique comment définir vos propres types d'objets. Nous verrons que les objets Java sont définis par des classes, et plus rarement par des énumérations.
- Le **Chapitre 4** approfondit le cycle de vie des objets. Il se consacre plus précisément aux mécanismes de constructions et de mise à mort des objets.
- Le **Chapitre 5** présente les principales méthodes de réutilisation du code de la POO : la composition et l'héritage de classe.
- Le **Chapitre 6** étudie les différentes sortes de polymorphisme et s'attarde sur le polymorphisme par sous-type. Ce dernier garantit qu'un objet réagit selon son type concret, peu importe la variable qui le manipule.
- Le **Chapitre 7**, introduit de nouveaux éléments d'abstraction. Il présente notamment les interfaces, indispensable pour créer du code flexible.

La deuxième partie utilise ces briques pour concevoir du code flexible et respectueux de la POO.

- Le **Chapitre 8** étudie la notion de responsabilités.
- Le **Chapitre 9** étudie des techniques pour réduire le couplage entre les types.
- Le **Chapitre 10** montre comment les tests unitaires aident à mesurer les progrès réalisés dans l'implémentation d'un système. Il explique également les implications que peuvent avoir

l'écriture d'un test avant l'écriture du code à tester.

Conventions de l'ouvrage

Nous mettrons en évidence plusieurs éléments. Selon leurs importances, nous les ferons précéder des icônes suivantes :



Signale un contenu à comprendre et à connaître pour poursuivre la lecture sereinement.



Signale un contenu intervenant dans l'évaluation du cours.



Signale un conseil ou un principe qui constitue une bonne pratique.



Signale un contenu facultatif, se rapportant le plus souvent à un élément avancé de Java ou à un autre langage de programmation.

Au cours d'un chapitre ou d'une section, nous ferons évoluer du code par petites modifications. Pour gagner quelques lignes, nous évitons autant que possible de présenter du code qui n'a pas changé entre les versions. Le code inchangé est remplacé dans les extraits par le commentaire `// Code inchangé`.

Chapitre 1. Les fonctions et leurs limites

Ce chapitre revoit les fonctions et présente certaines de leurs limites. Ces limites démontreront le besoin de nouveaux outils, tels que les objets.

1.1. Les fonctions sont des blocs de code réutilisables

Vous pouvez voir une fonction comme une recette de cuisine. Une recette décrit comment passer d'une liste d'ingrédients à un résultat, en principe mangeable. Une recette possède généralement un nom et est réutilisable : tant que vous lui fournissez des ingrédients similaires, vous obtiendrez le résultat attendu. Enfin, une recette peut entrer dans l'élaboration de recettes plus complexes.

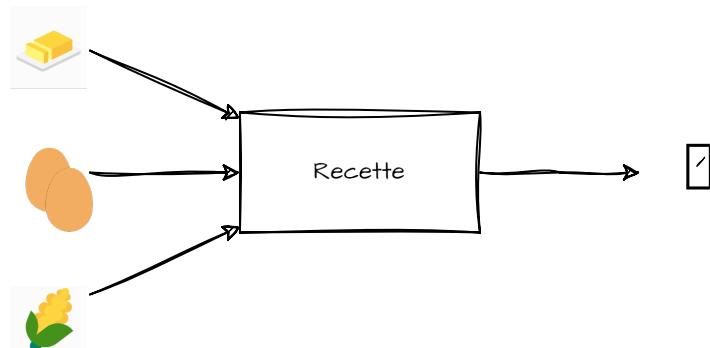


Figure 1. Une recette décrit comment produire un résultat

En programmation, une fonction décrit comment produire un résultat à partir d'une liste de paramètres à fournir. Elle possède aussi un nom et est réutilisable. Une fonction aide aussi à décomposer un problème en problèmes plus simples à résoudre.

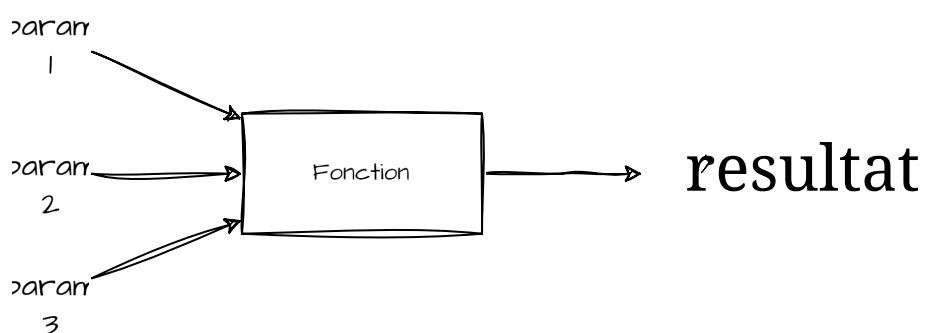


Figure 2. Une fonction décrit comment produire un résultat

Fonction

Une fonction est un bloc de code nommé et réutilisable qui effectue une tâche selon des paramètres et qui produit un résultat.

Pour exécuter une fonction, vous devez l'appeler. L'appel mentionne le nom de la fonction et fournit, entre parenthèses, une liste **d'arguments** compatibles avec les paramètres. Un argument est une expression à affecter à un paramètre. À la fin de l'appel, une fonction retourne souvent un résultat que vous pouvez affecter à une variable.

La Figure 3 appelle la fonction `pow` pour initialiser une variable.

```
1 package ch01;
2
3 public class FunctionCall {
4
5     public static void main(String[] args) {
6         double myDoubleVariable = Math.pow(2, 4);
7
8         System.out.printf("my double variable = %.2f\n", myDoubleVariable);
9     }
10}
11
```

The diagram shows the code for a `FunctionCall` class with a `main` method. An annotation "Nom de la fonction" points to the word `pow`. Another annotation "classe de la fonction" points to the word `Math`. A callout "Mémorise le résultat de l'appel" points to the assignment statement `double myDoubleVariable = Math.pow(2, 4);`. Annotations "argument pour le paramètre a" and "argument pour le paramètre b" point to the arguments `2` and `4` respectively in the `pow` call.

Figure 3. Un appel de fonction détaillé

➤ Sortie

```
my double variable = 16,00
```

Quand il retourne une valeur, un **appel de fonction est une expression** que vous pouvez utiliser dans des expressions plus complexes. Un appel peut même servir d'argument à un autre appel, comme dans l'extrait suivant.

```
1 public class ComplexFunctionCall {
2
3     public static void main(String[] args) {
4         long second = Math.min(42, Math.max(20, 75));
5         System.out.printf("second = %d\n", second);
6     }
7 }
```

➤ Sortie

```
second = 42
```

Pour la fonction appelante, Java exécute un appel de fonction en 3 étapes :

1. évaluer la liste des arguments, de gauche à droite ;
2. affecter chaque argument au paramètre de la fonction, selon sa position ;
3. exécuter les instructions de la fonction, ce qui produit un résultat utilisable par l'appelant.



En Java, les arguments sont uniquement transmis par **position** : un argument est associé au paramètre correspondant selon son ordre dans l'appel. Nous parlerons de mode d'appel positionnel.

D'autres langages, comme C#, permettent d'utiliser des arguments nommés : on associe directement un argument au nom du paramètre, sans se soucier de l'ordre.

Exemple d'appel nommé en C#

```
double _16 = Math.Pow(2, 4); // appel positionnel  
double _16Bis = Math.Pow(y: 4, x: 2); // appel nommé, l'ordre n'a plus  
d'importance
```

1.2. Les fonctions se déclarent

Une bibliothèque standard propose des fonctions qui répondent à des besoins généraux comme « arrondir un double » ou « calculer x à la puissance y ». En revanche, elle ne propose pas de fonctions adaptées à des problèmes spécifiques comme « calculer les dégâts d'une attaque » ou « retourner une carte ». Heureusement, vous pouvez déclarer de nouvelles fonctions.

Vous travaillez sur un jeu de rôle et vous devez coder un algorithme qui calcule le nombre de points d'expérience (XP) requis pour passer au niveau suivant, étant donné le niveau actuel du joueur. L'algorithme retenu est le suivant.

Soit `niveau`, un entier compris entre 1 et 99 inclus, pour calculer le nombre d'XP requis pour le niveau suivant, il faut :

- Si $1 \leq niveau \leq 16$, xp requis vaut $2 \times niveau + 7$
- Si $17 \leq niveau \leq 31$, xp requis vaut retourner $5 \times niveau - 38$
- Sinon, xp requis vaut $9 \times niveau - 158$

Ce code peut être réutilisé à plusieurs endroits : vous choisissez de l'implémenter avec une fonction, illustrée par l'extrait suivant.

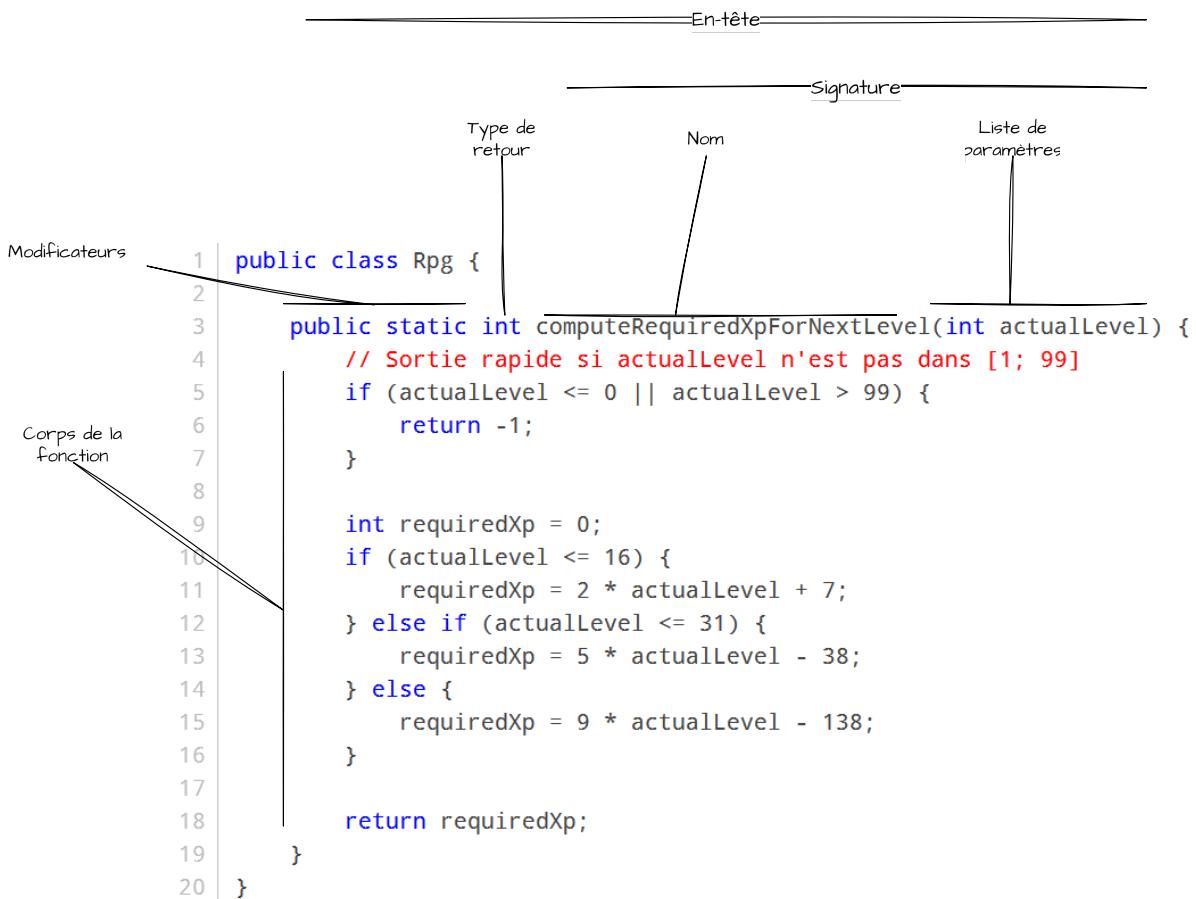


Figure 4. Déclaration d'une fonction

Notez la distinction entre l'en-tête et la signature d'une méthode. Le compilateur utilise cette signature pour distinguer des fonctions de même nom. Revenons sur chaque élément.

Conventions d'écriture



En Java, l'accolade ouvrante, `{`, est écrite sur la même ligne que l'en-tête. La première instruction du corps est écrite à la ligne suivante. Enfin, l'accolade fermante, `}`, est écrite à la ligne suivant la dernière instruction.

1.2.1. Les modificateurs

La déclaration d'une fonction commence par zéro, un ou plusieurs modificateurs.

Modificateur

Mot-clé modifiant les propriétés par défaut d'une déclaration.

Le tableau suivant présente quelques modificateurs utiles.

| Modificateur | Description | Exemple |
|----------------------|-------------------------------------------------------------------------|---------------------------------------------------------------------|
| <code>public</code> | Rend la fonction accessible depuis d'autres classes. | <code>public static int computeRequiredXp(int level) { ... }</code> |
| <code>static</code> | Associe la fonction à la classe plutôt qu'à un objet (voir chapitre 2). | <code>Math.pow(2, 4)</code> et pas <code>2.pow(4)</code> |
| <code>private</code> | Restreint l'accès à la fonction à la classe où elle est déclarée. | <code>private static boolean isValidLevel(int level) { ... }</code> |

1.2.2. Le type de retour

Le type du résultat, ou type de retour, suit les modificateurs. Il indique le domaine de la valeur renournée par la fonction.

Java autorise un et seul type de résultat par fonction. Dès lors, vous ne pouvez pas signaler qu'une fonction retourne un résultat de type `int` ou de type `string`.

Le compilateur utilise le type de retour pour vérifier plusieurs éléments pendant la traduction du programme. Par exemple, il s'assure que la valeur renournée est compatible avec le type de retour : si une fonction retourne un `int`, elle ne pourra pas retourner le string `"42"`. Il vérifie aussi que le résultat est utilisé correctement : si ma fonction retourne un `string`, je ne peux pas combiner l'appel aux opérateurs `*` ou `/`.

Typage statique



Java est un langage à typage statique : le type de chaque variable est vérifié à la compilation. Le compilateur garantit qu'une variable ne contiendra que des valeurs compatibles avec son type déclaré.

1.2.3. Le nom de la fonction

Le nom de la fonction sert à la désigner. Il doit respecter les règles de syntaxe relatives aux

identifiants.

Vous pouvez cependant déclarer plusieurs fonctions de même nom dans la même classe. Elles doivent alors se distinguer par leurs listes de paramètres : nous parlons de **surcharges**. Dans le reste cet ouvrage, nous prendrons l'habitude de désigner une fonction par son nom suivi de la liste des types de ses paramètres. Parfois, nous préfixerons le nom de sa classe au nom de la fonction. Plus rarement, nous préfixerons le nom du paquetage au nom de sa classe.

Ainsi, nous pourrons facilement distinguer `Math.max(double, double)` de `Math.max(int, int)`. En outre, `max(double, double)`, `Math.max(double, double)` et `java.lang.Math.max(double, double)` pourraient désigner la même fonction.

Conventions d'écriture



Le nom d'une fonction commence le plus souvent par un verbe, éventuellement suivi de compléments^[1]. Par exemple, les noms `computeNextDay(String[] players)` et `printMatch(String[] match)` ou `run()` respectent cette convention.

1.2.4. La liste des paramètres

Les paramètres sont des variables locales à la fonction, initialisées avec les arguments fournis par l'appelant. La déclaration d'un paramètre stipule au minimum son type et son nom. Une liste de paramètres prend la forme d'une paire de parenthèses entourant des déclarations de paramètres séparées par des virgules. La liste de paramètres d'une fonction peut être vide : elle correspond alors à une paire de parenthèses vide.

1.2.5. Le corps de la fonction

Le corps d'une fonction est le bloc d'instruction à exécuter quand la fonction est appelée. Comme tout bloc, il peut contenir des déclarations de variables locales et des structures de contrôle^[2].

Un corps peut également contenir des instructions `return`, utiles quand l'algorithme exécuté compte plusieurs chemins d'exécution. L'instruction `return` signale la fin de l'appel et, optionnellement, la production d'un résultat. En Java, tout chemin d'exécution doit se terminer par une instruction `return` si et seulement si le type de retour est autre que `void`.

1.3. Les fonctions doivent être appelées

Étudions le programme ci-dessous du point de vue des appels.

Exemple d'appels

```
1 public class FirstSample {  
2  
3     public static void main(String[] args) {  
4         int xpForLevel2 = Rpg.computeRequiredXpForNextLevel(1);  
5         int xpForLevel10 = Rpg.computeRequiredXpForNextLevel(9);
```

```

6     int xpForLevel99 = Rpg.computeRequiredXpForNextLevel(98);
7
8     System.out.printf("|%10s|%10s|%n", "N. suivant", "Xp requis");
9     System.out.printf("|%10s|%10s|%n", "-".repeat(10), "-".repeat(10));
10    System.out.printf("|%10d|%10d|%n", 2, xpForLevel12);
11    System.out.printf("|%10d|%10d|%n", 10, xpForLevel10);
12    System.out.printf("|%10d|%10d|%n", 99, xpForLevel99);
13 }
14 }
```

> Sortie

| N. suivant | Xp requis |
|------------|-----------|
| 2 | 9 |
| 10 | 25 |
| 99 | 744 |

Toute fonction définie peut être appelée. **Appeler** une fonction, c'est demander son exécution. L'appel mentionne le nom de la fonction et, entre parenthèses, une liste d'**arguments**. La JVM^[3] exécute un appel de fonction en plusieurs étapes ([Figure 5](#)).

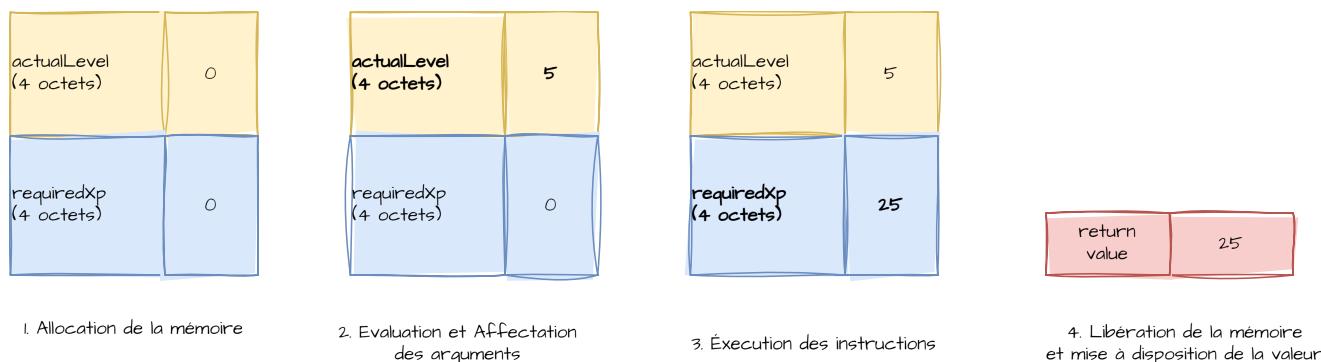


Figure 5. Exécution d'un appel à une fonction

Pour chaque appel à une fonction, la JVM lui alloue un espace mémoire, appelé **cadre d'exécution** (*execution frame*). La taille du cadre est calculée en fonction de la liste des paramètres et des variables locales à la fonction. À la fin de l'appel, la JVM détruit le cadre d'exécution, libérant ainsi la mémoire pour l'allouer à d'autres appels. La valeur de retour est mémorisée dans un registre du CPU.

La [Figure 6](#) illustre le principe d'empilement des cadres d'appels. La fonction principale et la fonction appelée occupent des espaces distincts en mémoire. Si la fonction appelée modifie la valeur d'un paramètre, cela est sans conséquence pour la fonction appelante.

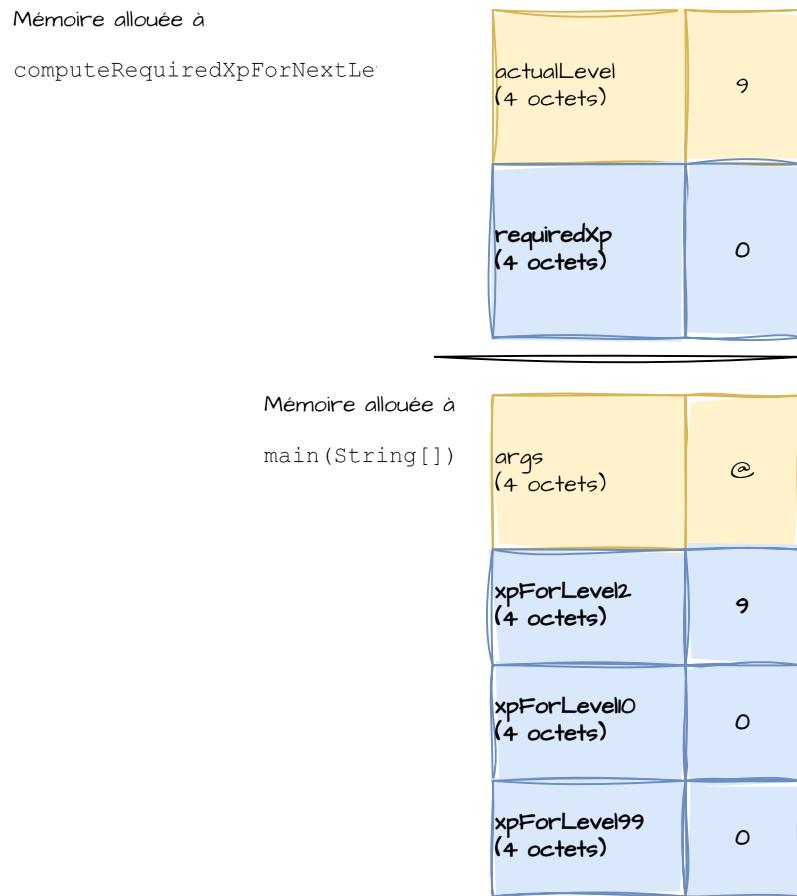


Figure 6. Les appels de fonction occupent des espaces distincts en mémoire

1.4. Les fonctions ont des limites



Bien qu'intéressantes à étudier, les pratiques présentées par cette section sont vivement déconseillées, car elles correspondent à des usages détournés des tableaux.

Comme nous le verrons dans les leçons suivantes, les objets offrent des solutions plus élégantes.

Supposez que vous programmiez un jeu utilisant des cartes à jouer classiques. Toutes les cartes d'un même jeu possèdent une face commune, appelée dos, et une face spécifique. Une carte est face cachée quand elle montre son dos aux joueurs. Dans la plupart des cartes à jouer, la face spécifique est composée d'un rang et d'une enseigne.

Vous devez créer un jeu de 52 cartes à jouer françaises où les enseignes sont ♠, ♥, ♦ et ♣ et où les rangs vont de l'as (1) au roi (13). Vous décidez de créer une fonction à cette fin : cette dernière retourne un jeu de cartes en respectant l'ordre standard (d'abord les piques, puis les coeurs, ensuite les carreaux et enfin les trèfles). Pour l'implémenter, vous devrez créer et manipuler des tableaux, mais des tableaux de quels types ?

1.4.1. Retourner plusieurs résultats

En Java, une fonction ne peut retourner qu'une seule valeur. Pour contourner cette limite, vous pouvez utiliser un tableau, mais cette solution a des inconvénients :

- elle est difficile à lire et à maintenir : vous devez vous souvenir de l'ordre des éléments dans le tableau.
- elle est fragile : si la structure du tableau change, tout le code qui l'utilise peut cesser de fonctionner.

Voici une première version de la fonction, accompagnée d'un programme d'exemple. Les éléments d'un tableau représentant une carte sont décrits par la [Table 1](#).

Table 1. Structure d'une carte

| Indice | 0 | 1 | 2 |
|---------|----------|----------|----------------|
| Rôle | Enseigne | Rang | Face visible ? |
| Exemple | '♥' | → 0x2665 | 1 |

Une fonction qui retourne plusieurs valeurs

```
1 public class CardsArray {  
2  
3     public static short ACE = 1;  
4     public static short KING = 13;  
5  
6     public static void main(String[] args) {  
7         short[][] cards = createCards();  
8     }  
9 }
```

```

8      for (short[] card : cards) {
9          System.out.printf("%d%c ", card[1], card[0]);
10     }
11 }
12
13 /**
14 * Retourne un tableau de 52 cartes dans l'ordre standard, faces cachées.
15 */
16 public static short[][] createCards() {
17     short[][] cards = new short[52][];
18     int nextPos = 0;
19
20     for (short suit = '♠'; suit <= '♣'; ++suit) {
21         for (short rank = ACE; rank <= KING; ++rank) {
22             cards[nextPos] = createCard(suit, rank);
23             ++nextPos;
24         }
25     }
26
27     return cards;
28 }
29
30 public static short[] createCard(short suit, short rank) {
31     return new short[] { suit, rank, 0 };
32 }
33 }
```

Le programme appelle cette fonction et mémorise son résultat dans la variable `cards` avant de l'afficher. Nous utilisons l'opérateur d'indexation, `[int indice]`, pour consulter les éléments d'une carte. Notez que, même si tableau est de type `short`, nous pouvons lui affecter des `char` et mettre en forme leurs valeurs comme des `char` : le type `char` est implicitement compatible avec le type `short`.

➤ Sortie

```
1♠ 2♠ 3♠ 4♠ 5♠ 6♠ 7♠ 8♠ 9♠ 10♠ 11♠ 12♠ 13♠ 1♥ 2♥ 3♥ 4♥ 5♥ 6♥ 7♥ 8♥ 9♥ 10♥ 11♥
12♥ 13♥ 1♦ 2♦ 3♦ 4♦ 5♦ 6♦ 7♦ 8♦ 9♦ 10♦ 11♦ 12♦ 13♦ 1♣ 2♣ 3♣ 4♣ 5♣ 6♣ 7♣ 8♣ 9♣
10♣ 11♣ 12♣ 13♣
```

Dans cette première version, toutes les données d'une carte sont de type `short`, ce qui paraît étrange. Vous souhaitez probablement utiliser des types proches du rôle joué par l'élément. Nous pouvons créer de tels tableaux en Java. Pour ce faire, Java nous impose alors de déclarer un tableau de type « universel » : un tableau d'objets. La version suivante retourne un tableau de cartes où chaque carte est un tableau d'objets dont la [Table 2](#) présente la structure.

Table 2. Structure d'une carte

| | | | |
|----------------|-------------------|------------------|----------------------|
| Indice | 0 | 1 | 2 |
| Type | <code>char</code> | <code>int</code> | <code>boolean</code> |
| Rôle | Enseigne | Rang | Face visible ? |
| Exemple | '♥' | 1 | <code>false</code> |

Retourner des valeurs de types hétérogènes

```
1 public class CardsArray {
2
3     public static short ACE = 1;
4     public static short KING = 13;
5
6     public static void main(String[] args) {
7         Object[][] cards = createCards();
8         for (Object[] card : cards) {
9             System.out.printf("%d%c ", card[1], card[0]);
10        }
11    }
12
13 /**
14 * Retourne un tableau de 52 cartes dans l'ordre standard, faces cachées.
15 */
16 public static Object[][] createCards() {
17     Object[][] cards = new Object[52][];
18     int nextPos = 0;
19
20     for (char suit = '\u2660'; suit <= '\u2663'; ++suit) {
21         for (short rank = ACE; rank <= KING; ++rank) {
22             cards[nextPos] = createCard(suit, rank);
23             ++nextPos;
24         }
25     }
26
27     return cards;
28 }
29
30 public static Object[] createCard(char suit, short rank) {
31     return new Object[] { suit, rank, 0 };
32 }
33 }
```

Les changements principaux concernent les types des variables et le type de retour de la fonction. Les tableaux d'objets présentent plusieurs inconvénients que l'extrait suivant illustre.

Utilisation avancée de tableaux hétérogènes

```
1 public class SumOfRanks {
2
3     public static void main(String[] args) {
4         Object[][] cards = CardsArray.createCards();
5         int sum = 0;
6
7         for (Object[] card : cards) {
8             sum += (int) card[0];
9         }
10
11         System.out.printf(
12             "La somme des rangs d'un jeu de carte vaut %d%n",
13             sum
14         );
15     }
}
```

1. Le rôle joué par chaque élément du tableau est masqué. Difficile sans contexte de savoir ce qui se cache derrière l'expression `(int)card[0]`...qui est, par ailleurs, incorrecte.
2. Les conversions `Object` → `int` coutent en performance. En effet, elles incluent des transformations intermédiaires.
3. Enfin, la fonction principale est sensible à la structure des éléments. Si `createCards` permute l'ordre des éléments du résultat, la fonction principale ne calculera plus la somme des rangs (au fait, avez-vous trouvé le souci ?).

Vous pouvez réduire ces inconvénients avec des fonctions utilitaires telles que `int rankOf(Object[])` ou `char suitOf(Object[])`. Cependant, elles n'amélioreront pas les performances.

Pour résumer, envisagez l'utilisation de tableaux comme type de retour si et seulement si :

- les éléments du tableau sont du même type ;
- chaque élément occupe une position naturelle, comme des coordonnées ou les noms d'une personne.

L'utilisation des tableaux d'objet est vivement déconseillée. Vous verrez au chapitre suivant que les objets répondent bien mieux à ce genre de problématique.

1.4.2. Modifier une variable déclarée dans un autre cadre

Comme nous l'avons vu précédemment, un appel de fonction provoque la copie des arguments dans les paramètres, ces paramètres se comportant ensuite comme des variables locales. Quand l'argument est une variable, **sa valeur** est recopiée dans le paramètre correspondant. Ce mode de transmission des paramètres par valeur explique le comportement du programme suivant.

```

1 class ImmutableNatural {
2     public static void increment(int val) {
3         ++val;
4     }
5 }
6
7 public class ValueParameter {
8     public static void main(String[] args) {
9         int a = 42;
10        ImmutableNatural.increment(a);
11
12        System.out.printf("a = %d", a);
13    } //==> a = 42
14 }
```

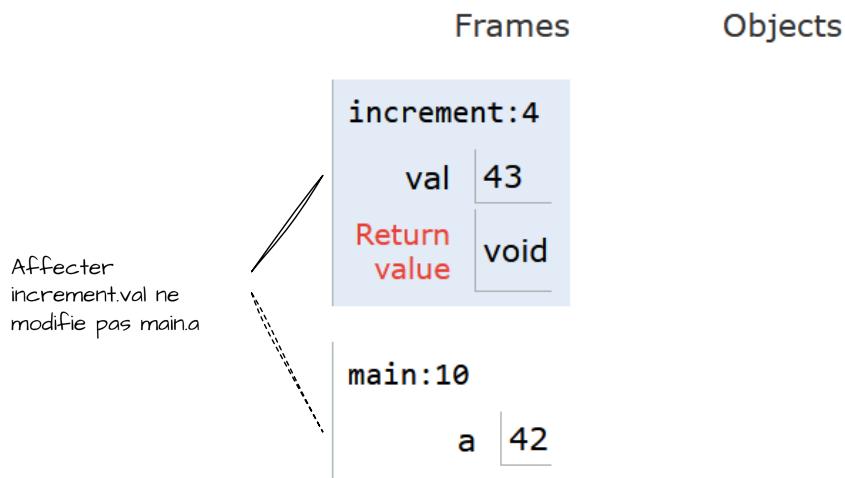


Figure 7. Situation de la mémoire illustrant la copie des arguments

Cependant, d'autres types acceptent des mutations de leurs valeurs. C'est le cas des tableaux. La fonction suivante fait passer une carte au rang suivant.

```

1 public class MuteArgs {
2
3     static void nextRank(Object[] card) {
4         if (card == null) {
5             return;
6         }
7         if (card.length < 3) {
8             return;
9         }
10
11         int currentRank = (int) card[1];
12         card[1] = 1 + (currentRank % 13);
13     }
14
15     public static void main(String[] args) {
16         Object[] trebolJack = { '\u2663', 11, false };
17         System.out.printf(
18             "Avant next rank : %d%c%n",
19             trebolJack[1],
20             trebolJack[0]
21         );
22
23         nextRank(trebolJack);
24
25         System.out.printf(
26             "Après next rank : %d%c%n",
27             trebolJack[1],
28             trebolJack[0]
29         );
30     }
31 }
```

> Sortie

```
Avant next rank : 11♣
```

Contrairement au premier extrait, les variables `trebolJack` et `card` référencent le même tableau, ce que montre la [Figure 8](#). Quand `nextRank(Object[])` modifie `card`, la modification est visible par la fonction principale.

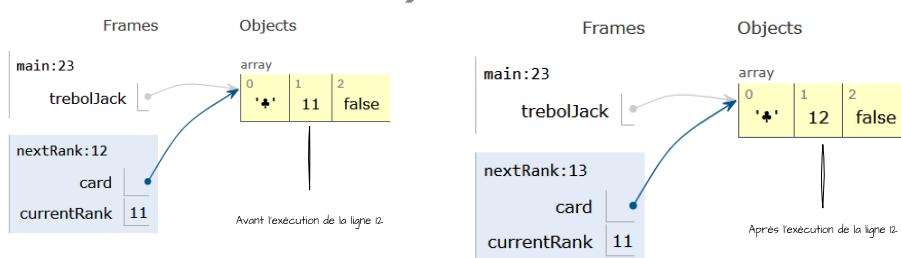


Figure 8. Situation en mémoire avant et après la modification d'un tableau

Les variables `trebolJack` et `card` ne mémorisent pas directement de tableaux. **Elles mémorisent plutôt l'adresse d'un tableau.** Ce sont des **références**, c'est-à-dire des variables qui contiennent l'adresse d'une donnée. Leur valeur est obtenue indirectement, en allant inspecter le contenu de l'adresse référencée. Cette connaissance indirecte de la valeur d'une donnée permet à une fonction de modifier une variable déclarée dans une autre fonction.

Référence

Variable mémorisant des adresses. Les références accèdent et modifient leur valeur par indirection. En Java, les tableaux sont mémorisés par des références

La fonction `nextRank(Object[])` produit un effet de bord : son appel modifie une variable déclarée à un autre endroit. Quand plusieurs fonctions produisent des effets de bord, des bugs tendent à apparaître. Ces bugs sont difficiles à repérer et à corriger. Il est en outre difficile de prédire le résultat d'une telle fonction. Pouvez-vous affirmer que les tableaux de l'extrait suivant seront égaux ? Pouvez-vous deviner l'ordre des cartes dans ces tableaux ? Vous faites face à une fonction qui produit des effets de bord : elle dépend d'un générateur de nombres aléatoires.

Cette fonction possède aussi des effets de bord

```

1 import java.util.Arrays;
2
3 public class CardsShuffling {
4
5     public static void main(String[] args) {
6         Object[][] firstSet = CardsArray.createCards();
7         Object[][] secondSet = CardsArray.createCards();
8
9         shuffle(firstSet, 100);
10        shuffle(secondSet, 100);
11
12        System.out.printf("First set%n");
13        System.out.printf("%s%n", Arrays.deepToString(firstSet));
14        System.out.printf("Second set%n");
15        System.out.printf("%s%n", Arrays.deepToString(secondSet));
16    }
}
```

```

17
18     private static void shuffle(Object[][] cards, int swapCount) {
19         int swapsLeft = Math.clamp(swapCount, 1, 100);
20
21         while (swapsLeft > 0) {
22             int firstIndex = (int) (Math.random() * cards.length);
23             int secondIndex = (int) (Math.random() * cards.length);
24
25             Object[] temp = cards[firstIndex];
26             cards[firstIndex] = cards[secondIndex];
27             cards[secondIndex] = temp;
28
29             --swapsLeft;
30         }
31     }
32 }
```

De nouveau, vous verrez au chapitre suivant que les objets répondent bien mieux à ce genre de problématique. Cependant, nous ne voulons pas terminer ce chapitre sur les fonctions sans discuter des fonctions pures, sans effets de bord.

1.5. Les fonctions pures sont sans effets de bord

De façon générale, il est préférable de concevoir des fonctions sans effets de bord. Ce n'est pas le cas des extraits précédents, car elles modifient des variables déclarées par d'autres fonctions.

Une fonction pure ne modifie aucune variable déclarée ailleurs et appelle uniquement des fonctions pures, dont les résultats sont déterministes. En conséquence, une telle fonction est aussi déterministe : les mêmes arguments produisent le même résultat. Vous pouvez aisément valider le comportement d'une fonction pure.

L'extrait suivant définit une version pure de la fonction `nextRank(Object[])`. Plutôt que de modifier le tableau référencé par `card`, la fonction crée et retourne un nouveau tableau, construit en consultant celui reçu en paramètre.

Une fonction pure

```

1 static Object[] nextRank(Object[] card) {
2     if (card == null) {
3         return;
4     }
5     if (card.length < 3) {
6         return;
7     }
8
9     int currentRank = (int)card[1];
10
11    return new Object[] {
12        card[0],
13        1 + ((currentRank + 1) % 13),
14        card[2]
15    };
}
```

Contrairement à sa version impure, cette fonction est composable : vous pouvez écrire l'expression `nextRank(nextRank(new Object[] { '♣', 2, false}))`. Vous pouvez par ailleurs aisément déterminer son résultat, car il dépend uniquement de l'argument initial.

Le principal inconvénient des fonctions pures est qu'elles consomment plus de ressources. Notre fonction pure alloue de la mémoire pour un tableau à chaque appel. Elle prend également plus de temps à s'exécuter à cause de cette allocation. Cette consommation accrue n'a pas, cependant, d'importance dans la plupart des contextes.

1.6. Les fonctions peuvent recevoir un nombre variable d'arguments

Java propose les fonctions `Math.min(int, int)` et `Math.max(int, int)` qui retournent respectivement la valeur minimum et maximum entre les deux arguments. Il serait intéressant de disposer de fonctions équivalentes, mais qui retourneraient le minimum ou le maximum entre **N** arguments, avec **N > 1**. Plutôt que d'écrire un nombre potentiellement grand de surcharges, nous souhaitons déclarer une fonction utilisée de façon semblable à la fonction `printf`. Cette dernière reçoit comme premier argument un modèle de format et un nombre quelconque de valeurs comme arguments supplémentaires. Comment déclarer une fonction recevant un nombre quelconque d'arguments ?

Java permet à une fonction de recevoir un nombre variable d'arguments du même type grâce à l'ellipse, appelée aussi **varargs**. Cette fonctionnalité évite de surcharger une fonction avec plusieurs versions qui diffèrent uniquement par le nombre de paramètres. Une ellipse est déclarée avec trois points (...) après le type du paramètre. Par exemple, `int... numbers` signifie que la méthode peut accepter zéro, un ou plusieurs arguments de type `int`. À l'intérieur de la méthode, l'ellipse est traitée comme un tableau du type spécifié.

L'extrait suivant déclare la fonction `max(int first, int... others)`. Elle attend au moins un argument (`first`) et un nombre variable d'autres arguments (`others`) de type entier.

Une fonction dotée d'une ellipse

```
public class MyMath {
    public static int max(int first, int... others) {
        int max = first;
        for (int num : others) {
            if (num > max) {
                max = num;
            }
        }
        return max;
    }
}
```

Un autre exemple est la méthode `toLine(char sep, String... words)`. Cette méthode prend un séparateur (`sep`) et une liste variable de chaînes de caractères (`words`). Elle concatène toutes les chaînes en les séparant par le caractère spécifié. Voici comment elle pourrait être implémentée :

```
public class MyArray {  
    public static String toLine(char sep, String... words) {  
        String result = "";  
        for (String word : words) {  
            result += sep + word;  
        }  
        result += sep;  
  
        return result;  
    }  
}
```

Ces fonctions sont appelables en spécifiant au moins un argument. Cet argument sera affecté au premier paramètre. Les arguments suivants seront tous affectés à l'ellipse. Vous pouvez aussi fournir un tableau à l'ellipse.

Exemples d'appels

```
1 public class VarArgsSample {  
2     public static void main(String[] args) {  
3         int max1 = MyMath.max(21, 42, 33, 5, 8);  
4         int max2 = MyMath.max(21, new int[] { 42, 33, 5, 8 });  
5         String line1 = MyArray.toLine('|', "PrB", "PrI", "PrA");  
6         String line2 = MyArray.toLine('%', "Web1", "Web2", "Web3");  
7  
8         System.out.printf("max 1 = %d, max 2 = %d%n", max1, max2);  
9         System.out.printf("%s%n", line1);  
10        System.out.printf("%s%n", line2);  
11    }  
12 }
```

Les ellipses sont soumises à quelques contraintes :

- Une fonction ne peut avoir qu'une ellipse dans sa liste de paramètre.
- L'ellipse correspond toujours au dernier paramètre déclaré.

Envisagez d'utiliser l'ellipse



Quand vous avez besoin de plusieurs versions de la même fonction et que ces versions ne changent que par le nombre de paramètres de même type, l'ellipse est une solution à envisager.

1.7. En résumé

Les fonctions sont des séquences d'instructions nommées et paramétrées. Elles servent à décomposer un problème et à éviter les répétitions. Demander l'exécution d'une fonction revient à l'appeler. Tout appel mentionne le nom de la fonction et fournit des arguments, qui sont des expressions affectées aux paramètres. Ces derniers se comportent comme des variables locales : ils vivent le temps de l'appel dans le cadre de l'appel. À la fin de l'appel, la fonction produit une valeur appelée résultat.

Déclarer une fonction consiste à déclarer un en-tête et un corps. L'en-tête est composé de modificateurs, du type du résultat, du nom de la fonction et de ses paramètres. Le corps correspond à la séquence d'instructions à exécuter. Cette séquence est composée de déclarations de variables, d'affectations, d'appels, de structures de contrôle et d'instructions `return` qui terminent l'exécution de la fonction.

En Java, une fonction ne peut retourner qu'un seul résultat. De même, le langage ne transmet pas les paramètres par référence pour la plupart des types primitifs, comme `int` et `double`. Vous pouvez contourner ces limites à l'aide des tableaux, mais les solutions produites sont rarement élégantes et peu performantes. Le chapitre suivant aborde la notion d'objet, beaucoup mieux intégrée au langage.

- [1] Un à trois compléments
- [2] Sélections, boucles et séquences
- [3] *Java Virtual Machine, JVM*

Chapitre 2. Les objets et les méthodes d'objet

Ce chapitre étudie les objets et explique comment ils facilitent la programmation par rapport aux tableaux utilisés au chapitre précédent.

Jusqu'à présent, vous avez défini et appelé des fonctions, c'est-à-dire des blocs nommés et paramétrés d'instructions. Les fonctions facilitent la décomposition d'un problème en sous-problèmes et aident à réutiliser du code.

En Java, toute fonction est définie dans une **classe** qui, jusqu'à présent, tient le rôle de **conteneur de fonctions**. Les appels aux fonctions que vous avez définies respectent l'écriture **NomDeLaClasse.nomDeLaFonction(<arguments>)**. Cependant, certains appels ne respectent pas cette écriture.

Imaginez un jeu dans lequel vous devez déterminer si deux cartes forment une paire. Deux cartes forment une paire si elles sont de même rang. Nous pourrions modéliser le problème avec des tableaux d'objets, comme au chapitre précédent, mais vous conviendrez que l'extrait suivant est plus parlant.

Déterminer si deux cartes forment une paire

```
1 public class CardsSample {  
2  
3     public static void main(String[] args) {  
4         PlayingCard firstCard = new PlayingCard(Rank.ACE, Suit.HEART);  
5         PlayingCard secondCard = new PlayingCard(Rank.ACE, Suit.SPADE);  
6  
7         if (firstCard.hasSameRankAs(secondCard)) {  
8             System.out.printf(  
9                 "%s et %s forment une paire%n",  
10                firstCard,  
11                secondCard  
12            );  
13        } else {  
14            System.out.printf(  
15                "%s et %s ne forment pas une paire%n",  
16                firstCard,  
17                secondCard  
18            );  
19        }  
20    }  
21 }
```

Cet extrait utilise des objets, que vous pouvez voir comme des entités auxquelles vous demandez des services. Quand ils sont bien conçus, les objets facilitent la rédaction des programmes, car ils permettent de l'écrire avec des termes spécifiques au domaine du problème.

Ce chapitre explique comment utiliser des objets, sans expliquer comment définir leurs types. Le chapitre suivant étudiera cet aspect.

2.1. Les objets sont les valeurs d'un type

Pour rappel, un type définit **un domaine de valeurs et des opérations** sur ces valeurs. En Java, les opérations prennent la forme d'opérateurs et de fonctions. Chaque type possède également une représentation en mémoire. Java prédéfinit les types repris par la [Table 3](#) : ce sont les types intégrés au langage^[1].

Table 3. Types intégrés à Java

| Type | Valeur par défaut | Taille en octets | Représentation |
|---------|-------------------|------------------|-------------------|
| byte | 0 | 1 | Entier signé |
| short | 0 | 2 | Entier signé |
| int | 0 | 4 | Entier signé |
| long | 0L | 8 | Entier signé |
| float | 0.0f | 4 | Standard IEEE 754 |
| double | 0.0d | 8 | Standard IEEE 754 |
| char | '\u0000' | 2 | Table UTF-16 |
| boolean | false | ? | Indéfini |

En plus des types intégrés, Java propose les types **String** pour les chaînes de caractères et le type **Array** pour les tableaux. Étudions la Javadoc^[2] du type **String**. Son premier paragraphe indique que « *la classe String représente des chaînes de caractères. Tous les strings littéraux... sont implémentés comme des occurrences de cette classe* ». Autrement dit, toutes les chaînes de caractères appartiennent à une classe. Attardons-nous sur ce concept.

Une classe désigne un regroupement d'éléments possédant des caractéristiques semblables. Par exemple, la classe des quadrilatères regroupe tous les polygones ayant quatre côtés. De même, la classe des cartes à jouer regroupe des cartes dotées d'un rang et d'une enseigne. Une classe définit un domaine de valeurs... comme un type.

Si vous continuez d'inspecter la Javadoc, vous constaterez que la classe **String** déclare plusieurs fonctions et un opérateur pour la concaténation. De même, nous pouvons définir des opérations sur les quadrilatères comme le calcul de leurs périmètres ou de leurs aires. Si elles étaient conscientes, nous pourrions aussi demander à une carte si elle est du même rang qu'une autre carte. Une classe définit des opérations sur les valeurs du domaine... comme un type.

En conclusion, la notion de classe va au-delà du conteneur de fonctions. **En POO, une classe définit un type dont les valeurs sont des objets.**

Classe en POO



Méthode de définition d'un type à l'aide d'un modèle. Les valeurs du type sont des objets.

Objet

Occurrence ou instance d'une classe. Si une classe définit un type, un objet représente une valeur de ce type.

Nous arrivons enfin à cette notion d'objet vu comme occurrence d'une classe. Dans les sections qui suivent, nous présentons leurs caractéristiques fondamentales qui les distinguent des valeurs des types intégrés.

2.2. Les objets ont des comportements

Parmi les fonctions de la classe **String**, certaines présentent la particularité **de ne pas être static**, comme **indexOf(int)** ou **length()**. Lisons la Javadoc de la première fonction :

```
public int indexOf(int ch)
```

*Returns the index within **this string** of the first occurrence of the specified character.*

— Javadoc de la méthode String.indexOf(int),

[`https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html#indexOf\(int\)'](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html#indexOf(int))

La première phrase doit vous interpeller : à quoi correspond ce « *this string* » qui n'apparaît pas dans l'en-tête de la fonction ? Si nous essayons d'appeler cette fonction en commençant **String**, nous obtenons l'erreur suivante. De nouveau, la notion de « *méthode non statique référencée depuis un contexte statique* » interpelle.

Erreur si vous préfixez **String** à **indexOf(int)** dans un appel

```
non-static method indexOf(int) cannot be referenced from a static context
String.indexOf('a');
^-----^
```

Une méthode non statique est une fonction qui nécessite un objet pour être appelée. En effet, pour s'exécuter, une méthode non statique utilise des caractéristiques propres à l'objet. Autrement dit, le cadre d'exécution d'une telle fonction est composé d'arguments, de variables locales et de l'objet associé à l'appel. En POO, une fonction non statique porte le nom de **méthode d'objet** tandis qu'une fonction statique porte le nom de **méthode de classe**.

Méthode d'objet

Fonction non statique qui a besoin d'un objet pour s'exécuter. L'objet et les arguments de la méthode définissent son cadre d'exécution.



Méthode de classe

Fonction statique qui est le plus souvent appelée avec une classe. Les arguments de la méthode définissent son cadre d'exécution.

Le mot-clé **this**

Comme nous le verrons dans les chapitres suivants, **this** est une référence à l'objet qui reçoit l'appel, utilisable dans le corps d'une méthode d'objet.



Pour faire une analogie, quand une personne dit "Je m'appelle Nicolas", le "Je" joue le même rôle que le **this** pour cette personne.

L'extrait suivant présente plusieurs appels corrects de méthodes d'objet. Si vous exécutez ce programme, vous constaterez que l'objet qui reçoit l'appel influence le résultat. Par exemple, les appels à la méthode d'objet **compareTo(String another)** retournent des résultats différents

selon l'objet qui reçoit l'appel.

Quelques exemples d'appels

```
1 public class OthersObjectMethodCall {
2
3     public static void main(String[] args) {
4         String hello = "hello";
5         String bigHello = hello.toUpperCase();
6
7         System.out.printf("Length of [%s] : %d%n", hello, hello.length());
8         System.out.printf("Length of [%s] : %d%n", bigHello, bigHello.length());
9         System.out.printf(
10             "4th letter of [%s] ? '%c'%n",
11             hello,
12             hello.charAt(3)
13         );
14         System.out.printf(
15             "4th letter [%s] ? '%c'%n",
16             bigHello,
17             bigHello.charAt(4)
18         );
19         System.out.printf(
20             "[%s] compared to [%s] ? %d%n",
21             hello,
22             bigHello,
23             hello.compareTo(bigHello)
24         );
25         System.out.printf(
26             "[%s] compared to [%s] ? %d%n",
27             bigHello,
28             hello,
29             bigHello.compareTo(hello)
30         );
31         System.out.printf(
32             "index of first [L] in [%s] ? %d%n",
33             hello,
34             hello.indexOf('L')
35         );
36         System.out.printf(
37             "index of first [L] in [%s] ? %d%n",
38             bigHello,
39             bigHello.indexOf('L')
40         );
41     }
42 }
```

➤ Sortie

```
Length of [hello] : 5
Length of [HELLO] : 5
4th letter of [hello] ? 'l'
4th letter [HELLO] ? 'O'
[hello] compared to [HELLO] ? 32
[HELLO] compared to [hello] ? -32
index of first [L] in [hello] ? -1
```

```
index of first [L] in [HELLO] ? 2
```

La figure suivante illustre la syntaxe d'un appel de méthode d'objet.

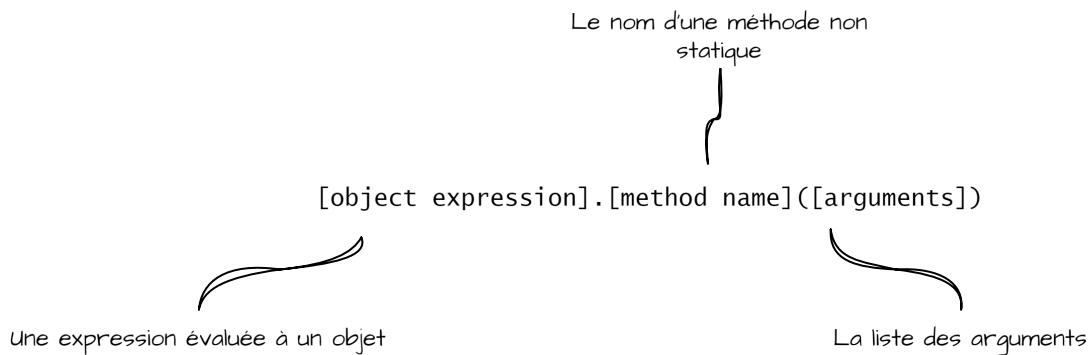


Figure 9. Syntaxe d'un appel de méthode d'objet en Java

Comme tout appel à une fonction, l'appel à une méthode d'objet est une expression. Il en est de même pour les arguments... et pour l'objet qui reçoit l'appel. Ces propriétés autorisent les constructions suivantes :

Appels de méthodes en cascade

```
String bigFullscreen = ("nicolas "+"Hendrikx").toUpperCase();
//==> "NICOLAS HENDRIKK"
String capitalized = "NICOLAS".toLowerCase().replace('n', 'N');
//==> "Nicolas"
```

La dernière expression se comprend plus facilement qu'une composition d'appels de fonction. En effet, sa lecture respecte l'ordre des transformations. À l'inverse, l'écriture alternative suivante, interdite en Java, masque cet ordre.

Ceci n'est pas du code Java

```
String capitalized = String.replace(String.toLowerCase("NICOLAS"), 'n', 'N');
// ==> "Nicolas"
```

Notez que chaque transformation s'effectue sur un objet différent. En effet, toutes les méthodes d'objet de la classe `String` retournent un nouveau `String`. La [Figure 10](#) illustre le nombre d'objets intermédiaires créés par l'expression précédente.



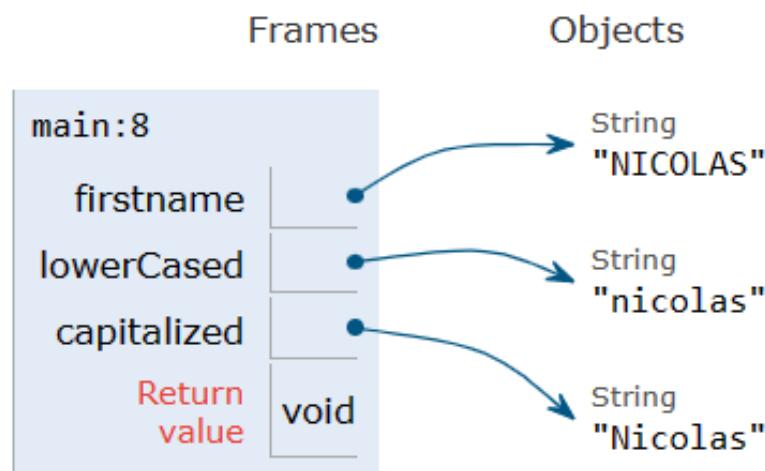


Figure 10. Un objet de départ, un objet par appel

Ces objets intermédiaires poseront un problème si la mémoire allouée au programme est limitée ou si cette transformation est à appliquer à un grand nombre d'objets, à tous les membres d'une école, par exemple.

Dans l'exemple précédent, l'expression crée trois strings. Le dernier est le seul utilisable. Les autres occupent de la mémoire, mais ne sont pas utilisables, faute de variables les mémorisant.

Pour conclure cette section, étudions les étapes suivies par la JVM pour traiter un appel de méthode d'objet. La JVM exécute un appel de méthode d'objet selon ces étapes :

1. Déterminer l'objet qui reçoit l'appel, c'est-à-dire évaluer l'expression à gauche de l'opérateur `".."`.
2. Déterminer les arguments, c'est-à-dire évaluer les expressions écrites entre parenthèses, de gauche à droite.
3. Créer un cadre d'exécution pour l'appel et affecter l'objet et les arguments à ce cadre.
4. Exécuter les instructions de la méthode jusqu'à rencontrer une instruction de sortie.
5. Libérer la mémoire et mettre à disposition le résultat.

En mémoire, un appel de méthode d'objet ressemble à un appel de fonction classique. La seule différence est la présence d'une variable `this`, faisant référence à l'objet qui reçoit l'appel. La [Figure 11](#) modélise un appel de méthode d'objet en mémoire.

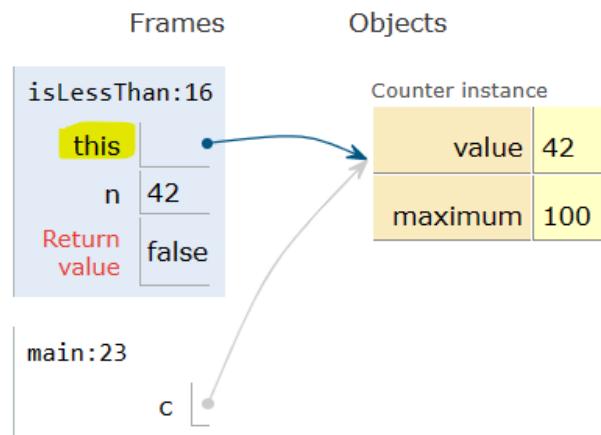


Figure 11. Appel à une méthode d'objet

Les variables `this` et `c` présentent la particularité de pointer sur la même structure de données, située dans une zone de la mémoire dédiée aux objets. Comme les tableaux, Java manipule les objets avec des références.

2.3. Les objets se manipulent par référence

Java manipule les objets avec des références, autrement dit avec des variables mémorisant des adresses. Ces adresses appartiennent à la zone de mémoire dédiée aux objets : **le tas^[3]**. Le tas mémorise également les tableaux et les strings qui sont créés pendant l'exécution du programme.

Voyez le tas comme un grand entrepôt où chaque objet est stocké dans une boîte étiquetée avec son adresse. Les références ressemblent à des tickets libellés avec ces adresses. Si deux références ont le même libellé, elles désignent la même boîte, autrement dit le même objet (Figure 12).

Deux références au même objet

```
1 String a = new String("Hello");
2 String b = a; // `b` pointe vers le même objet que `a`
```

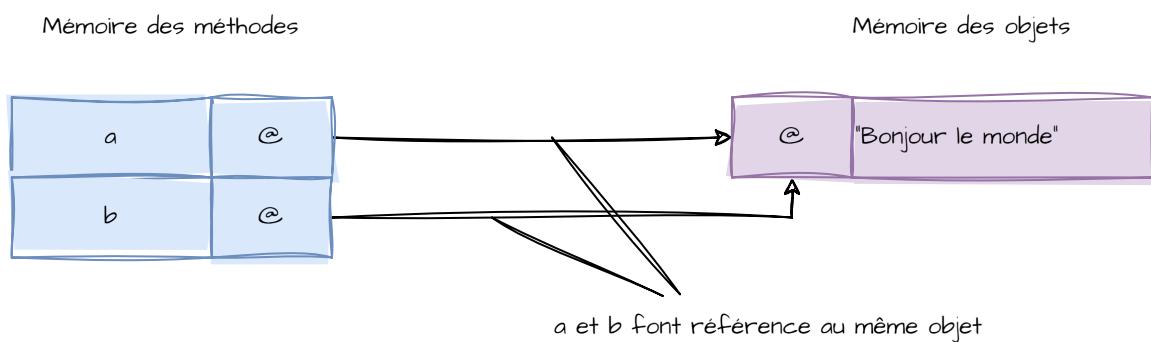


Figure 12. Exemple de références pointant sur le même objet

Référence d'objet

Variable mémorisant des adresses. En Java, les tableaux et les objets sont manipulés avec des références.



Tas (*Heap*)

Zone de la mémoire d'un programme dans laquelle les objets et les tableaux résident.

Deux cas méritent d'être étudiés en détail : les références à **null** et les références au même objet.

2.3.1. Une référence peut valoir null

Java impose d'affecter une valeur initiale à toute variable avant de l'utiliser. Pour les références, la valeur **null** peut être donnée. Cette valeur signale l'absence d'adresse. Si vous appelez une méthode depuis cette référence, vous obtiendrez une exception qui terminera brusquement l'exécution du programme.

Appel à une méthode depuis une référence à null

```
1 String test = null;
2
3 test.toLowerCase();
4 //==> java.lang.NullPointerException
```

La gestion de références à **null** demande de la discipline. Deux cas sont à considérer : celui des paramètres et celui du résultat. Si un paramètre d'une méthode peut valoir **null**, **vous devez vérifier ce paramètre avant d'exécuter vos instructions**. Si une méthode peut retourner **null**, **vous devez vérifier le résultat avant de poursuivre votre exécution**.

Prenons une méthode du JDK pour illustrer. La méthode de classe **System.getProperty(String)** retourne une valeur spécifique à l'environnement d'exécution sous forme de String. Cette méthode lance des exceptions si l'argument reçu vaut **null** ou est un texte vide, de longueur 0.

Table 4. Exemple d'informations retournées

| Argument | Résultat |
|----------------|---------------------------------|
| "os.name" | "Linux" |
| "java.version" | "17.0.1" |
| "" | IllegalArgumentException |
| null | NullPointerException |

Si l'argument est valide, **System.getProperty(String)** retourne la valeur de la propriété correspondante ou **null** quand l'argument ne correspond à aucune propriété. Nous voulons appeler cette méthode pour obtenir et afficher des propriétés de l'environnement, en mettant la valeur en lettres majuscules. Pour obtenir une fonction robuste, nous devons valider le paramètre et, le cas échéant, terminer immédiatement l'appel. Nous devons aussi valider le résultat avant d'appeler **toUpperCase()**.

Traitements de **null**

```
1 public static String getAndFormatProperty(String propertyName) {
```

```

2     if(propertyName == null || propertyName.isBlank()) {
3         return "Argument invalide";
4     }
5
6     String PropertyValue = System.getProperty(propertyName);
7
8     if(PropertyValue == null) {
9         PropertyValue = "?";
10    }
11
12    return String.format("%s= %s",
13                          propertyName, PropertyValue.toUpperCase());
14 }
```

Exemple d'appels

```

1 System.out.println(getAndFormatProperty("os.name"));
2 // Affiche "os.name= LINUX"
3 System.out.println(getAndFormatProperty(null));
4 // Affiche "argument invalide"
5 System.out.println(getAndFormatProperty("atuin"));
6 // Affiche "atuin= ?"
```

Vous constatez probablement que la gestion de la valeur `null` alourdit le code. Quand vous programmez une méthode qui retourne une référence, évitez de retourner la valeur `null`. Vous devrez aussi vous défendre contre elle, soit en validant les paramètres et les résultats d'appels, soit en interdisant explicitement de telles valeurs dans la documentation de vos méthodes.

2.3.2. Les objets ont une identité

Dans notre quotidien, nous avons l'habitude de désigner sous des noms différents le même objet. Par exemple, selon les personnes, la même maison pourra être désignée par les noms "*mon domicile*", "*la maison de ma mamie*" ou "*le repère aux chats*". Nous savons que nous parlons du même bâtiment, car ce dernier occupe un espace connu, identifié par une adresse postale.

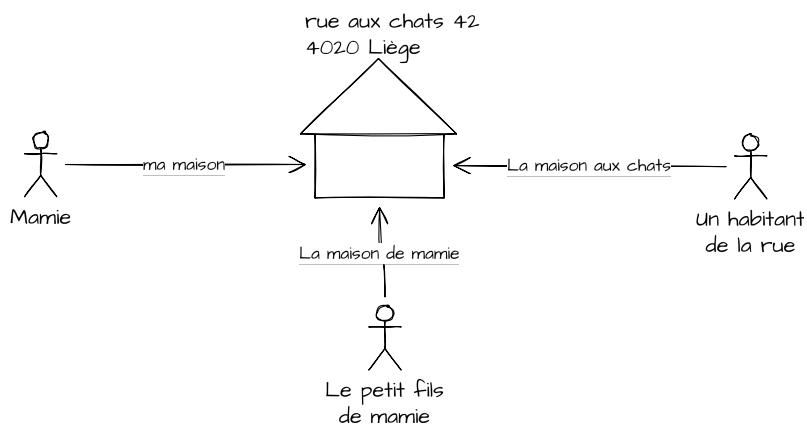


Figure 13. Le même bâtiment désigné de trois manières différentes

Au chapitre précédent, nous avons montré que cette situation vaut pour les tableaux. Deux références peuvent désigner le même tableau, ce qui autorise notamment une fonction à modifier le contenu d'un tableau pour le compte d'une autre fonction. Cette propriété s'explique par le fait

que le tableau occupe un espace en mémoire, qu'une adresse correspond à cet espace et que c'est cette adresse que mémorisent les variables désignant le tableau.

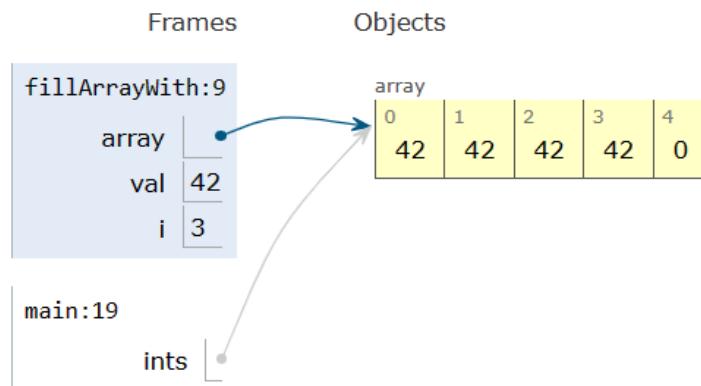


Figure 14. Deux références au même tableau

Comme pour les tableaux, différentes références peuvent désigner le même objet en mémoire. Un objet occupe également une zone de la mémoire identifiée par une adresse. Plus généralement, en Java, tout objet créé possède une identité correspondant à son adresse en mémoire.

Identité d'un objet



L'identité d'un objet est une caractéristique qui le rend unique par rapport aux autres. En Java, l'identité d'un objet peut correspondre à son adresse mémoire.

Java propose la méthode statique `System.toIdentityHashCode(Object)` pour connaître l'identité d'un objet. Elle retourne un entier que vous pouvez assimiler à l'adresse qu'occupe l'objet en mémoire. Appliqués à deux références, les opérateurs `==` et `!=` déterminent l'égalité ou la différence avec cette valeur : deux références sont égales si elles mémorisent la même adresse. L'extrait suivant illustre ces propriétés.

Égalité entre des références

```
1 public class ReferenceEquality {
2
3     public static void main(String[] args) {
4         String hello = "good morning";
5         String helloAgain = hello;
6         String helloCloned = hello.toUpperCase().toLowerCase();
7
8         System.out.printf(
9             "%16s|%16s|%16s|%16s%n",
10            "NOM",
11            "hello",
12            "helloAgain",
13            "helloCloned"
14        );
15        System.out.printf(
16            "%16s|%16s|%16s|%16s%n",
17            "VALEUR",
18            "hello",
19            "helloAgain",
```

```

20         "helloCloned"
21     );
22     System.out.printf(
23         "%16s|%16s|%16s|%16s%n",
24         "IDENTITE",
25         System.identityHashCode(hello),
26         System.identityHashCode(helloAgain),
27         System.identityHashCode(helloCloned)
28     );
29
30     System.out.printf("%n%n");
31
32     System.out.printf(
33         "%16s|%16s|%16s|%16s%n",
34         "",
35         "hello",
36         "helloAgain",
37         "helloCloned"
38     );
39     System.out.printf(
40         "%16s|%16s|%16s|%16s%n",
41         "hello ==",
42         hello == hello,
43         hello == helloAgain,
44         hello == helloCloned
45     );
46     System.out.printf(
47         "%16s|%16s|%16s|%16s%n",
48         "helloAgain ==",
49         helloAgain == hello,
50         helloAgain == helloAgain,
51         helloAgain == helloCloned
52     );
53     System.out.printf(
54         "%16s|%16s|%16s|%16s%n",
55         "helloCloned ==",
56         helloCloned == hello,
57         helloCloned == helloAgain,
58         helloCloned == helloCloned
59     );
60 }
61 }
```

➤ Sortie

| NOM | hello | helloAgain | helloCloned |
|----------------|-----------|------------|-------------|
| VALEUR | hello | helloAgain | helloCloned |
| IDENTITE | 951007336 | 951007336 | 2001049719 |
| | hello | helloAgain | helloCloned |
| hello == | true | true | false |
| helloAgain == | true | true | false |
| helloCloned == | false | false | true |

La Figure 15 présente la situation en mémoire. `helloCloned` référence un autre objet que `hello` et

`helloAgain`. Rappelez-vous que les méthodes d'objet de la classe `String` retournent d'autres objets que celui qui reçoit l'appel.

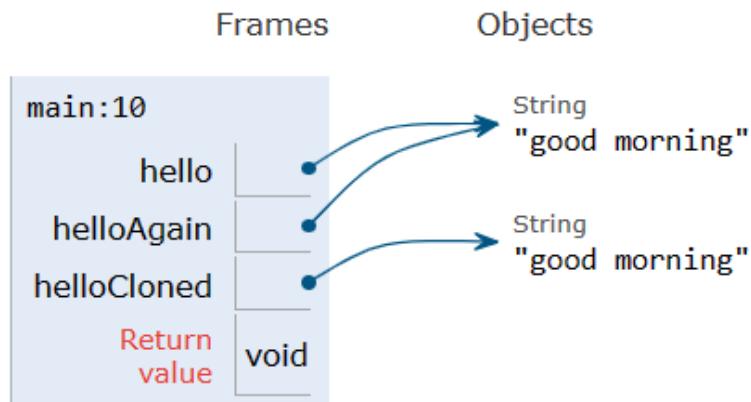


Figure 15. Trois références, dont deux sur le même objet

Égalité entre les références



Appliqués à des références, les opérateurs `==` et `!=` s'appuient sur les identités des objets, pas sur leurs contenus.

Deux références sont égales si elles pointent vers le même objet en mémoire ou si elles valent toutes les deux `null`. Dans les autres cas, elles sont différentes.

Un objet reçoit son adresse mémoire au moment de sa création. Comme la section suivante l'explique, cette notion de création est une étape indispensable, sans quoi une référence vaudra toujours `null` et sera inutilisable.

2.4. Les objets doivent être créés

Avant de manipuler un objet, vous devez le créer : nous parlerons aussi d'instanciation. L'expression de création d'un objet commence par l'opérateur `new` suivi du nom de sa classe et d'une liste, éventuellement vide, d'arguments. Le nom de la classe et la liste d'arguments jouent un rôle analogue à celui d'un appel de fonction. Cette fonction particulière a pour seul rôle d'initialiser l'objet créé : on l'appelle constructeur.

Création d'objet ou instantiation



La création d'un objet `o` de la classe `C` est une opération qui alloue de la mémoire pour `o`, puis initialise cette mémoire allouée à l'aide d'une méthode spéciale de `C` et retourne finalement l'adresse de l'objet.

Constructeur

Méthode spéciale, du même nom que la classe de l'objet créé, appelée pour initialiser un nouvel objet.

Une classe peut proposer plusieurs constructeurs, c'est-à-dire plusieurs méthodes pour créer ses objets. Prenons par exemple, la classe `StringJoiner` qui définit des objets concaténant des strings avec des séparateurs. Cette classe propose deux constructeurs. Le premier déclare un séparateur d'éléments comme seul paramètre tandis que le second constructeur en déclare trois : le séparateur d'éléments, un préfixe et un suffixe.



Pour utiliser la classe `StringJoiner`, vous devez l'importer ou utiliser son nom complet qualifié, à savoir `java.util.StringJoiner`.

L'extrait suivant crée deux objets `StringJoiner`. Nous avons défini un premier objet pour joindre les éléments d'un tableau à l'aide du constructeur à trois paramètres. Le second objet nous servira à écrire un texte de la forme `x >= y >= z`. Nous avons utilisé le constructeur à un paramètre pour créer cet objet.

Création de deux objets à l'aide de deux constructeurs

```
1 public class StringJoinerExample {  
2  
3     public static void main(String[] args) {  
4         StringJoiner arrayJoiner = new StringJoiner(", ", "[", "]");  
5         StringJoiner greqJoiner = new StringJoiner(" >= ");  
6  
7         joinChildren(arrayJoiner);  
8         joinChildren(greqJoiner);  
9  
10        System.out.printf("array joiner = %s%n", arrayJoiner);  
11        System.out.printf("greq joiner = %s%n", greqJoiner);  
12    }  
13  
14    private static void joinChildren(StringJoiner joiner) {  
15        joiner.add("Leo");  
16        joiner.add("Tom");  
17        joiner.add("Noa");  
18    }  
19}
```

```
18      }
19 }
```

➤ Sortie

```
array joiner = [Leo, Tom, Noa ]
greq joiner = Leo >= Tom >= Noa
```

`arrayJoiner` et `greqJoiner` réfèrent deux objets différents de la même classe. Bien qu'appartenant à la même classe, ces objets répondront différemment aux appels de méthodes. En effet, nous ne les avons pas créés avec les mêmes arguments. Autrement dit, ils ne sont pas tout à fait dans le même état. Étudions cette notion d'état.

2.5. Les objets possèdent un état

Les objets référencés par `arrayJoiner` et `greqJoiner` acceptent les mêmes appels de méthode, car ils appartiennent à la même classe. Cependant, comme ils ont été initialisés différemment, leurs résultats sont différents, comme le montre l'extrait suivant. Ainsi, **le traitement d'un appel de méthode par un objet est influencé par son état.**



État d'un objet

Valeurs de ses caractéristiques essentielles à un instant précis.

L'état d'un objet peut changer durant sa vie : on dit qu'il mute. Dans l'extrait précédent, `joinChildren(StringJoiner)` fait muter son paramètre en lui ajoutant de nouveaux éléments à joindre.



Mutation d'un objet

Changement d'état, c'est-à-dire des valeurs des caractéristiques essentielles d'un objet.

L'identité et l'état sont deux caractéristiques essentielles des objets. En effet, deux objets distincts peuvent se trouver dans le même état. Pensez par exemple à des jumeaux, à des clones ou aux produits sortant d'une usine.

En Java, les opérateurs `==` et `!=` permettent de savoir si deux références sont égales. Ces opérateurs comparent les identités des objets, par leur contenu. Pour déterminer si deux objets sont égaux, autrement dit dans le même état, utilisez la méthode `equals(Object)`.

Égalité de référence et d'objets

```
1 public class ObjectEquality {
2
3     public static void main(String[] args) {
4         String hello = "good morning";
5         String helloAgain = hello;
6         String helloCloned = hello.toUpperCase().toLowerCase();
7
8         System.out.printf(
9             "%20s|%12s|%12s|%12s%n",
10            "NOM",
11            "hello",
12            "helloAgain",
13            "helloCloned"
14        );
15        System.out.printf(
16            "%20s|%12s|%12s|%12s%n",
17            "IDENTITE",
18            System.identityHashCode(hello),
19            System.identityHashCode(helloAgain),
20            System.identityHashCode(helloCloned)
21        );
22
23        System.out.printf("%n%n");
```

```

24
25     System.out.printf(
26         "%20s|%12s|%12s|%12s%n",
27         "",
28         "hello",
29         "helloAgain",
30         "helloCloned"
31     );
32     System.out.printf(
33         "%20s|%12s|%12s|%12s%n",
34         "hello equals",
35         hello.equals(hello),
36         hello.equals(helloAgain),
37         hello.equals(helloCloned)
38     );
39     System.out.printf(
40         "%20s|%12s|%12s|%12s%n",
41         "helloAgain equals",
42         helloAgain.equals(hello),
43         helloAgain.equals(helloAgain),
44         helloAgain.equals(helloCloned)
45     );
46     System.out.printf(
47         "%20s|%12s|%12s|%12s%n",
48         "helloCloned equals",
49         helloCloned.equals(hello),
50         helloCloned.equals(helloAgain),
51         helloCloned.equals(helloCloned)
52     );
53 }
54 }
```

➤ Sortie

| NOM | hello | helloAgain | helloCloned |
|--------------------|-----------|------------|-------------|
| IDENTITE | 791452441 | 791452441 | 834600351 |
| <hr/> | | | |
| hello equals | true | true | true |
| helloAgain equals | true | true | true |
| helloCloned equals | true | true | true |



Si vous affectez une référence `hello` à une autre référence `helloSame`, elles désigneront le même objet. Fatalement, `hello.equals(helloSame)` et `helloSame.equals(hello)` retournent `true`.

Pour éviter de confondre l'égalité entre deux références et celle entre deux objets, nous parlerons plutôt d'équivalence entre deux objets pour le second cas.

Équivalence entre deux objets



Soient `ref1` et `ref2` deux références. Les objets référencés par `ref1` et `ref2` sont équivalents s'ils sont dans le même état. La définition d'un "même état"

dépend de la classe des objets.

2.6. Les méthodes de fabrique créent des objets

Plutôt qu'exposer des constructeurs, plusieurs classes du JDK proposent des méthodes de classe pour créer leurs objets. Par exemple, la classe `java.time.LocalDate` propose la fonction `now()` pour obtenir un objet correspondant à la date du jour locale^[4]. Cette méthode crée un nouvel objet à chaque appel, comme le prouve l'extrait suivant.

Différences des références, mais équivalence de deux dates du jour

```
1 import java.time.LocalDate;
2
3 public class LocalDatesNow {
4     public static void main(String[] args) {
5         LocalDate today = LocalDate.now();
6         LocalDate todayClone = LocalDate.now();
7
8         System.out.printf("today.equals(today clone) ? %b\n", today.equals(todayClone));
9         System.out.printf("today == todayClone? %b\n", today == todayClone);
10    }
11    //==> today.equals(today clone) ? true
12    //==> today == todayClone? false
13 }
```

Contrairement aux méthodes d'objet, ces méthodes n'ont pas besoin d'un objet pour être appelées. Elles s'appellent à partir du nom de la classe, d'où le terme de méthodes de classe. En Java, elles correspondent aux fonctions étudiées en programmation de base^[5].



Quand une méthode de classe a pour responsabilité de créer un objet, on l'appelle méthode de fabrique statique. Ces méthodes sont fréquentes parmi les classes du JDK.

Le tableau suivant récapitule les principales différences entre les méthodes d'objet et celles de classe. Notez que la méthode d'objet `formatted` existe depuis Java 15 : elle est récente.

| Critère | Méthode d'objet | Méthode de classe |
|-------------------------------|--------------------------------------|---------------------------------------------------------|
| Adressé à un objet | ✓ | ✗ |
| Présence de <code>this</code> | ✓ | ✗ |
| Exemple d'appel | "la réponse est %d".formatted(42) | <code>String.format("la réponse est %d", 42)</code> |

2.7. Les objets outrepassent les limites des fonctions

Nous avons conclu le chapitre précédent par la présentation de techniques qui contournent certaines limitations des fonctions, ou plutôt des méthodes de classe, en Java :

- pas de modifications des variables déclarées dans un autre cadre ;
- pas de possibilité de retourner plusieurs résultats de types différents.

Comme ils se manipulent à l'aide de références, les objets peuvent contourner ces limitations tout en facilitant l'exploitation des résultats.

2.7.1. Retourner plusieurs résultats

Avec les objets, nous pouvons simplifier les extraits de la [Section 1.4.1](#). Par exemple, nous pouvons calculer la somme des rangs sans conversions explicites. Le code utilise un vocabulaire proche de celui du problème. Sa compréhension est plus simple.

Les cartes vues comme des objets

```

1 public class SumOfRanks {
2
3     private static PlayingCard[] createCards() {
4         PlayingCard[] cards = new PlayingCard[52];
5         int nextIndex = 0;
6         for (Suit suit : Suit.values()) {
7             for (Rank rank : Rank.values()) {
8                 cards[nextIndex] = new PlayingCard(rank, suit);
9                 ++nextIndex;
10            }
11        }
12
13        return cards;
14    }
15
16    public static void main(String[] args) {
17        PlayingCard[] cards = SumOfRanks.createCards();
18        int sum = 0;
19
20        for (PlayingCard card : cards) {
21            sum += card.getRankValue();
22        }
23
24        System.out.printf(
25            "La somme des rangs d'un jeu de carte vaut %d%n",
26            sum
27        );
28    }
29 }
```

Les chapitres suivants étudient le code qui se cache derrière `PlayingCard`. À ce stade, ce code caché n'est pas nécessaire pour comprendre ce que fait le programme.

2.7.2. Modifier une variable déclarée dans un autre cadre

Les objets et les tableaux vivent sur le tas. Ce faisant, toute méthode qui reçoit une référence à un objet peut le faire muter. L'extrait de code suivant retourne une carte reçue en argument.

Mutation d'un entier encapsulé dans un tableau

```
1 import ch07.*;
2
3 public class FlipCards {
4
5     private static void doFlip(PlayingCard card) {
6         card.flip();
7     }
8
9     public static void main(String[] args) {
10        PlayingCard firstCard = new PlayingCard(Rank.ACE, Suit.HEART);
11
12        System.out.printf("Avant flip %s%n", firstCard);
13        doFlip(firstCard);
14        System.out.printf("Après flip %s%n", firstCard);
15    }
16 }
```

➤ Sortie

```
Avant flip A♥
Après flip
```

Notez qu'appeler la méthode **flip()** sur l'objet dans la méthode principale produirait le même effet.

Nous avons terminé d'étudier comment instancier et manipuler des objets. Dans le chapitre suivant, nous leverons un coin du voile qui cache l'implémentation d'une classe. Comme vous le verrez, une classe sera principalement composée de méthodes et de variables communes à ces méthodes.

2.8. En résumé

En Java, les classes définissent de nouveaux types dont les valeurs se nomment objets. Ces objets disposent de méthodes les dotant de comportements. Contrairement aux fonctions, que nous nommerons dorénavant méthodes de classe, le résultat d'un appel à une méthode d'objet dépend d'arguments, mais également de l'objet associé à l'appel. Outre leurs méthodes, tout objet possède un état et une identité.

Les objets se manipulent à l'aide de références qui sont des variables mémorisant des adresses. L'opérateur `==` indique si deux références désignent le même objet. La méthode `equals(Object)` détermine si deux objets sont dans le même état, nous parlerons d'équivalence.

Avant d'être utilisable, un objet doit avoir été créé. L'expression de création fait appel à un constructeur qui est une méthode spéciale, dont le nom correspond à celui de la classe de l'objet. En plus des constructeurs, plusieurs classes du JDK proposent des méthodes de classe pour fabriquer de nouveaux objets.

Ce chapitre porte essentiellement sur la manipulation des objets. Dans le chapitre suivant, nous étudions comment définir des classes, c'est-à-dire nos propres types d'objets.

[1] *built-in types*

[2] <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html>

[3] La pile est réservée aux cadres d'exécution des méthodes

[4] La localité d'une date dépend de la région du système.

[5] Selon les langages, on fera la distinction entre méthode d'objet, méthode de classe et fonction qui est une séquence nommée d'instructions de niveau global, c'est-à-dire n'appartenant à une classe

Chapitre 3. Les classes et les enums

Ce chapitre étudie les définitions de classe simple. Une classe simple est composée de champs de types primitifs et de méthodes d'objet.

Le [Chapitre 2](#) a étudié les objets, plus particulièrement comment les créer et appeler leurs méthodes. Les objets sont les valeurs d'un type qui correspondent à une classe.

Contrairement aux méthodes de classe, les méthodes d'objet doivent être associées à un objet pour être appelées. Le résultat d'un appel dépend des arguments et de l'état de l'objet.

Une classe ressemble à un moule à bougies. Les bougies qui en sortiront auront toutes la même forme et des comportements similaires. Cependant, une fois sorties du moule, les bougies connaîtront des vies différentes. Certaines seront consommées de suite, d'autres seront offertes, plusieurs seront stockées, etc.

Dans ce chapitre, nous étudions les premiers éléments de Java pour définir des classes d'objets. Nous étudierons une seconde méthode pour définir des types d'objets : les enums. Les chapitres suivants approfondissent ces éléments.

3.1. Les classes doivent être déclarées

En Java, un objet possède un type prenant le plus souvent la forme d'une classe. Comme pour les fonctions, les classes du JDK conviennent pour des problèmes génériques. Toutefois, elles répondent rarement à des problèmes spécifiques, comme la description d'une carte ou d'un joueur.

Heureusement, vous pouvez définir, nous écrirons plutôt **déclarer**, vos propres classes. La [Figure 16](#) illustre la structure complète d'une classe Java. Pour ce chapitre, nous nous contenterons de la structure simplifiée illustrée par la [Figure 17](#).

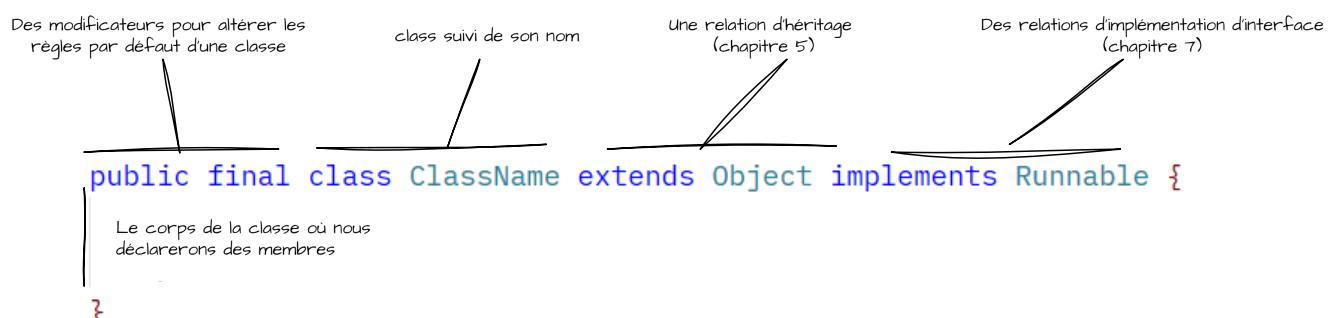


Figure 16. Structure complète d'une classe en Java

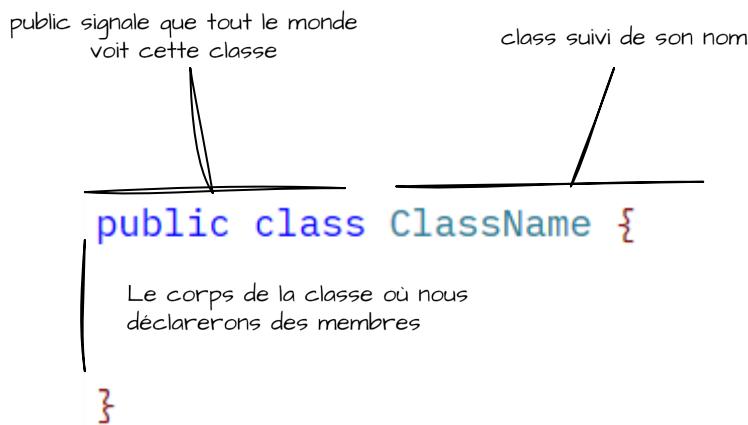


Figure 17. Structure simplifiée d'une classe en Java

Cette dernière structure suffit à créer des objets. Cependant, ils ne vous rendront pas de grands services.

Création d'un objet `ClassName`

```

1 public class ObjectCreation {
2
3     public static void main(String[] args) {
4         ClassName anObject = new ClassName();
5         System.out.printf("un objet = %s", anObject);
6     }
7 }
```

➤ Sortie

```
un objet = ch03.ClassName@77459877
```

Nous devons leur ajouter des méthodes et des données. Nous parlerons de déclarer des membres d'objet ou d'instance.

Membre d'objet



Données ou méthodes définies dans le corps d'une classe. Chaque objet possède sa propre copie des données qui influent sur l'exécution des méthodes.

Dans l'extrait précédent, l'objet référencé par `anObject` propose tout de même plusieurs méthodes d'objet telles que `equals(Object)`, `hashCode()` et `toString()`.



En vérité, toute classe hérite de méthodes définies dans la classe `java.lang.Object`. Plus généralement, en Java, toute classe est une sous-classe de `java.lang.Object`. Cette relation implique que tout objet est une valeur de la classe `java.lang.Object`.

Le texte affiché à droite du symbole `" = "` correspond au résultat retourné par l'appel `anObject.toString()`, fait par la méthode `printf(String, ...Object)`.

3.2. Les classes déclarent des méthodes d'objet

Inspirons-nous des cartes à jouer, introduites au chapitre précédent, pour déclarer une classe plus intéressante. Une carte à jouer possède un rang (*rank*), une enseigne (*suit*) et une face qui peut être visible ou cachée^[1]. Nous doterons une carte des comportements suivants :

- Retourner la carte.
- Déterminer si une carte est de même valeur qu'une autre.
- Déterminer si la carte est face visible.
- Obtenir le rang de la carte pour l'afficher.
- Obtenir l'enseigne de la carte pour l'afficher.

Chaque comportement correspond à une méthode. En POO, nous schématisons une classe et ses membres par une boite. Cette boite possède trois compartiments : le premier mentionne le nom de la classe, le second compartiment présentera les données et le troisième les méthodes. Ce type de schéma dépend moins du langage de programmation : il est utile pour réfléchir à une classe avant de la programmer.

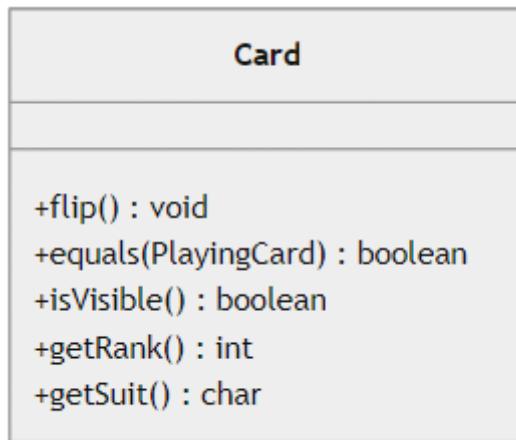


Figure 18. Schéma d'une classe

Nous déduisons un squelette de classe Java du schéma. Pour qu'il compile, nous avons ajouté des instructions `return` associées aux valeurs par défaut.

Squelette d'une classe Card

```
1 public class Card {  
2     public int getRank() {  
3         return 0;  
4     }  
5  
6     public char getSuit() {  
7         return '\0';  
8     }  
9  
10    public boolean isVisible() {  
11        return false;
```

```

12     }
13
14     public void flip() {
15     }
16
17     public boolean equals(Card o) {
18         return false;
19     }
20
21 }
```

Contrairement à une méthode de classe, l'en-tête d'une méthode d'objet est dépourvu du modificateur `static`. Faute de `static`, le compilateur rejette les appels `Card.getSuit()`, `Card.flip()`, `Card.isVisible()`, etc.

Message d'erreur retourné quand on appelle une méthode d'objet depuis une classe

```

1 Error:
2 | non-static method isVisible() cannot be
3 | referenced from a static context
4 | Card.isVisible()
5 | ^-----^
```

Déclarer une méthode d'objet



Une méthode d'objet est une méthode sans le modificateur `static`. Elle requiert un objet lors de tout appel.

En théorie, la méthode `flip()` produit un effet qui dépend de l'état de la carte qui reçoit l'appel. Plus précisément, `flip()` inverse la visibilité de cette carte(Table 5).

Table 5. Effet escompté du retournement

| Face visible avant l'appel | Face visible après l'appel |
|----------------------------|----------------------------|
| <code>false</code> | <code>true</code> |
| <code>true</code> | <code>false</code> |

Cependant, si nous créons deux cartes et que nous les retournons, aucune ne change d'état. Ces cartes restent face cachée, car la méthode `flip()` n'exécute aucune instruction et que la méthode `isVisible()` retourne toujours `false`.

Pas de changement d'état observé

```

1 public class CardFlip {
2
3     public static void main(String[] args) {
4         Card card1 = new Card();
5         Card card2 = new Card();
6
7         System.out.printf("|%12s|%12s|%12s|\n", "Moment", "card1", "card2");
8         System.out.printf(
9             "|%12s|%12b|%12b|\n",
```

```

10         "Avant flip()",  

11         card1.isVisible(),  

12         card2.isVisible()  

13     );  

14  

15     card1.flip(); //Attendu : card1.isVisible() == true  

16  

17     System.out.printf(  

18         "|%12s|%12b|%12b|\n",  

19         "Après flip()",  

20         card1.isVisible(),  

21         card2.isVisible()  

22     );  

23 }
24 }
```

➤ Sortie

| Moment | card1 | card2 |
|--------------|-------|-------|
| Avant flip() | false | false |
| Après flip() | false | false |

Ce qui nous manque, c'est un état que les méthodes `flip()` et `isVisible()` pourront consulter et modifier. L'état d'une carte à jouer correspond à son rang, à son enseigne et à un indicateur de retournement. Pour mémoriser cet état, nous pouvons déclarer une première variable de type `int`, une deuxième de type `char` et une troisième de type `boolean`.

Cependant, ces variables ne peuvent pas être locales à une méthode, sinon elles seraient créées et détruites à chaque appel. Nous devons les doter d'une portée plus grande, correspondant à la durée de vie de l'objet : nous devons déclarer des champs.

3.3. Les classes déclarent des champs

Un champ d'objet est une variable locale à un objet. Contrairement aux variables locales à une méthode, un champ se déclare dans le corps de la classe. Un champ présente deux caractéristiques intéressantes :

- sa durée de vie correspond à celle de son objet et va donc au-delà d'un seul appel de méthode ;
- il est à la portée des méthodes d'objet qui peuvent les consulter et les modifier.

Champ d'objet



Variable dont la valeur est propre à un objet. Contrairement aux variables locales à une méthode, un champ vit aussi longtemps que son objet.

Nous devons déclarer trois champs pour mémoriser l'état de nos cartes. L'extrait suivant déclare ces champs et les utilise dans les méthodes. Adaptons d'abord le schéma de la classe avant de présenter le code correspondant.

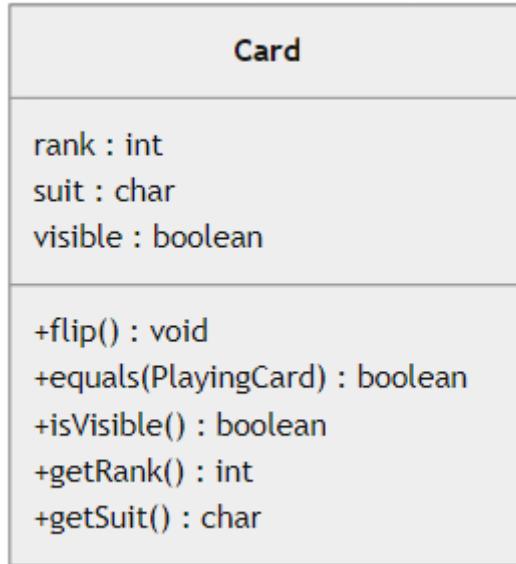


Figure 19. Schéma d'une classe avec ses champs

Une classe déclarant des champs d'objet

```

1 public class Card {
2
3     int rank = 1;
4     char suit = '♣';
5     boolean visible = false;
6
7     public int getRank() {
8         return rank;
9     }
10
11    public char getSuit() {
12        return suit;
13    }
14
15    public boolean isVisible() {
16        return visible;
17    }
18
19    public void flip() {
20        visible = !visible;
21    }
22
23    public boolean equals(Card other) {
24        return false;
25    }
26 }

```

Concrètement, les méthodes accèdent directement aux champs. Comme pour les variables locales, une méthode peut consulter ou modifier la valeur d'un champ. Contrairement aux variables locales, toute affectation d'un champ perdure après l'appel à une méthode.

Table 6. Comparaison entre champ, paramètre et variable locale

| Caractéristique | Champ | Paramètre | Variable locale |
|-----------------|-----------------------------|------------------------|------------------------|
| Déclaré dans | un corps de classe | un en-tête de méthode | un corps de méthode |
| Durée de vie | celle de l'objet | celle d'un appel | le bloc qui la déclare |
| Accessible par | toutes les méthodes d'objet | le corps de la méthode | le bloc qui la déclare |

Une fois créé, un objet **Card** possède sa propre version des champs **rank**, **suit** et **visible**. Si vous modifiez la valeur du champ **visible** pour une carte, cette modification est limitée à cette carte. Avec nos champs, le comportement souhaité pour les cartes est implémenté.

Dans l'extrait suivant, nous avons modifié les champs **rank** et **suit** de **card1** qui correspondent dorénavant au 10 de cœur. Une nouvelle carte représente l'as de pique.

Exemple d'accès aux champs

```

1 public class CardFlip {
2
3     public static void main(String[] args) {
4         Card card1 = new Card();
5         card1.rank = 10;
6         card1.suit = '♥';
7         Card card2 = new Card();
8
9         System.out.printf(
10             "|%12s|%12s|%12s|%n",
11             "Moment",
12             "%d%c".formatted(card1.rank, card1.suit),
13             "%d%c".formatted(card2.rank, card2.suit)
14         );
15         System.out.printf(
16             "|%12s|%12b|%12b|%n",
17             "Avant flip()", 
18             card1.isVisible(),
19             card2.isVisible()
20         );
21
22         card1.flip(); //Attendu : card1.isVisible() == true
23
24         System.out.printf(
25             "|%12s|%12b|%12b|%n",
26             "Après flip()", 
27             card1.isVisible(),
28             card2.isVisible()
29         );
30     }
31 }
```

➤ Sortie

| | | | |
|--|--------------|-------|-------|
| | Moment | 10♥ | 1♠ |
| | Avant flip() | false | false |

| | | |
|--------------|------|-------|
| Après flip() | true | false |
|--------------|------|-------|

La [Figure 20](#) présente une vue de la mémoire après l'appel à la méthode d'objet `flip()`.

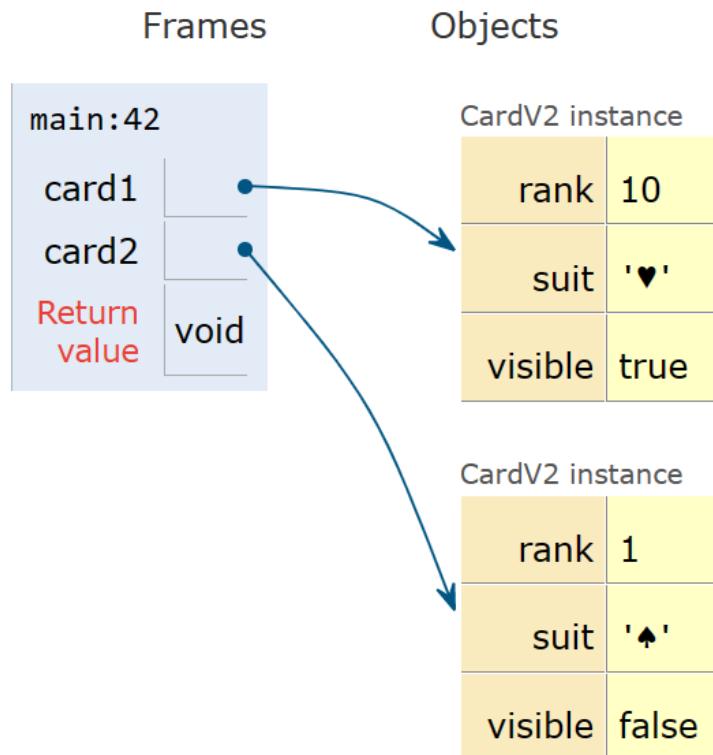


Figure 20. Situation des objets en mémoire après l'appel à `flip()`

Chaque objet :

1. occupe un espace mémoire différente ;
2. réserve de la mémoire pour ses champs ;
3. possède pour chaque champ une valeur spécifique.

Ajoutez le modificateur `static` à la déclaration d'un champ, et ce dernier devient un champ de classe. Contrairement à un champ d'objet, un champ est partagé par tous les objets de la classe. La [Figure 21](#) montre également que les champs de classe sont mémorisés dans une zone mémoire différente de celle associée aux objets.

 La valeur du champ étant commune, tout appel à `flip()` modifie « toutes » les cartes. Par ailleurs, un programme peut consulter ou modifier ce champ en passant par la classe.

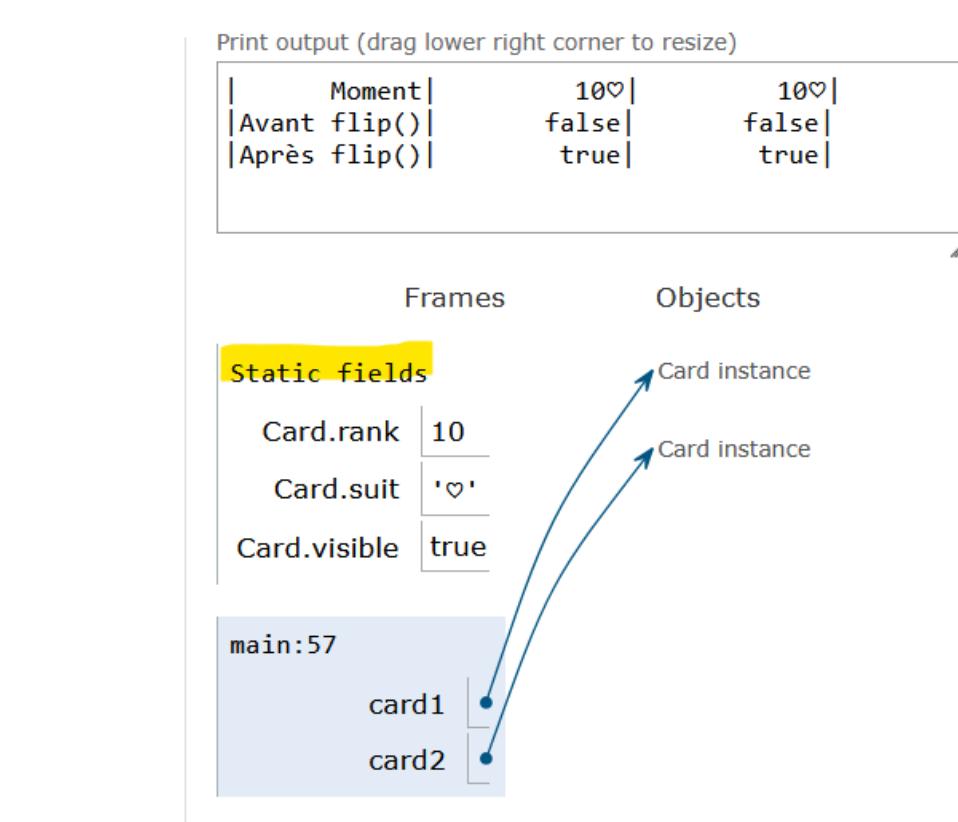


Figure 21. Deux objets partageant leurs champs

Les champs de classe publics **et** mutables sont vivement déconseillés. En effet, ils subissent fréquemment des modifications non désirées, difficiles à tracer et à corriger.

3.4. Les constructeurs initialisent les objets

Dans la [Section 2.4](#), nous avons évoqué le rôle joué par le nom d'une classe dans les expressions de création d'objet telles que `new Card()`. `Card()` y représente un appel au constructeur. Un constructeur est une méthode spéciale, chargée d'initialiser les champs d'un objet. Il présente la particularité de n'être utilisable qu'avec l'opérateur `new`, c'est-à-dire au moment de créer un objet.

Par défaut, Java génère un constructeur sans paramètres. Celui-ci est appelé quand nous écrivons `new Card()`. Si aucune valeur initiale n'a été donnée, le constructeur généré par défaut affecte à chaque champ la valeur par défaut de son type, le plus souvent une variation du `0` comme `0.0f`, `0.0`, `'\0'`, etc.

Constructeur

Méthode d'objet appelée uniquement pendant la création d'un objet pour l'initialiser.



Constructeur par défaut

Constructeur généré par le compilateur quand vous ne déclarez aucun constructeur dans une classe. Le constructeur par défaut est sans paramètres.

L'appel à un constructeur intervient pendant le processus de création d'un objet. En Java, la

création d'un objet compte 3 étapes :

1. Allouer de la mémoire nécessaire au stockage de l'objet ;
2. Initialiser cette zone de mémoire à l'aide du constructeur ;
3. Retourner l'adresse de l'objet pour l'affecter à une référence, par exemple.

Nous souhaitons affecter des valeurs précises aux cartes quand nous les créons. Actuellement, nous affectons le rang et l'enseigne désirés après la création de l'objet, ce qui n'est pas commode. Plus grave, nous prenons le risque de créer des cartes, mais d'oublier de les initialiser : nous pouvons obtenir un jeu de cartes composé uniquement d'as de pique.

Il est préférable d'initialiser l'état d'une carte dès sa création. Actuellement, écrire `new Card(10, '♥')` produit l'erreur suivante :

Tentative de création d'un objet sans constructeur spécifique déclaré

```
1 | Error:  
2 | constructor Card in class Card cannot be applied to given types;  
3 | required: no arguments  
4 | found: int, char  
5 | reason: actual and formal argument lists differ in length new Card(10, `♥`);
```

Java nous informe que le seul constructeur à sa disposition pour créer des cartes n'accepte aucun paramètre. Pour que l'expression `new Card(10, '♥')` soit acceptée, nous devons définir un constructeur compatible. L'extrait suivant déclare ce constructeur. Comme vous le constaterez, l'en-tête d'un constructeur présente deux particularités : il n'y a pas de type de retour et son nom est celui de la classe.

Déclaration d'un constructeur paramétré

```
1 public class Card {  
2  
3     int rank = 1;  
4     char suit = '♣';  
5     boolean visible = false;  
6  
7     public Card(int initialRank, char initialSuit) {  
8         rank = initialRank;  
9         suit = initialSuit;  
10    }  
11    // Reste du code inchangé  
12 }
```

Cet ajout rend acceptable la séquence suivante. Nous n'avons plus besoin d'accéder aux champs des cartes pour les initialiser : le constructeur s'en charge pour nous. Notre classe devient plus simple et sûre à utiliser. Contrairement au constructeur par défaut, notre constructeur est paramétré.

Création de cartes avec notre constructeur

```
1 public class CardFlip {
```

```

2
3     public static void main(String[] args) {
4         Card card1 = new Card(10, '\u2661');
5         Card card2 = new Card(1, '\u2660');
6
7         System.out.printf(
8             "|%12s|%12s|%12s|%n",
9             "Moment",
10            "%d%c".formatted(card1.rank, card1.suit),
11            "%d%c".formatted(card2.rank, card2.suit)
12        );
13        System.out.printf(
14            "|%12s|%12b|%12b|%n",
15            "Avant flip()", 
16            card1.isVisible(),
17            card2.isVisible()
18        );
19
20        card1.flip(); //Attendu : card1.isVisible() == true
21
22        System.out.printf(
23            "|%12s|%12b|%12b|%n",
24            "Après flip()", 
25            card1.isVisible(),
26            card2.isVisible()
27        );
28    }
29 }
```

Constructeur paramétré

Constructeur définissant un ou plusieurs paramètres utilisés pour initialiser un nouvel objet.



La déclaration d'un constructeur paramétré supprime le constructeur fourni par défaut. Nous étudierons les constructeurs et le cycle de vie d'un objet plus en détail au chapitre suivant.

Les objets devraient être cohérents après leur création



Avec ce principe, on devrait pouvoir appeler toutes les méthodes d'un objet après sa création, sans passer par une méthode intermédiaire telle que `init(args)` ou `setup(args)`.

3.5. Les méthodes admettent des structures de contrôle

Actuellement, nous pouvons créer de nouvelles cartes et les initialiser en une seule déclaration. Cependant, nous ne pouvons pas encore déterminer si une carte est égale à une autre. Autre problème, les paramètres du constructeur ne sont pas validés : rien n'empêche de créer une carte 42 de ♣.

Comme pour toute fonction, vous pouvez écrire des structures de contrôle dans le corps d'un constructeur ou d'une méthode d'objet. Le constructeur modifié valide les paramètres ou les

remplace, le cas échéant, par des valeurs par défaut.

Ajout de structures de contrôle

```
1 public class Card {  
2  
3     public static final int DEFAULT_RANK = 1;  
4     public static final char SPADE = '\u2660',  
5         HEART = '\u2661',  
6         DIAMOND = '\u2662',  
7         CLUB = '\u2663';  
8  
9     int rank;  
10    char suit;  
11    boolean visible;  
12  
13    public Card(int initialRank, char initialSuit) {  
14        if (initialRank < 1 || initialRank >= 14) {  
15            rank = DEFAULT_RANK;  
16        } else {  
17            rank = initialRank;  
18        }  
19  
20        if (initialSuit < SPADE || initialSuit > CLUB) {  
21            suit = SPADE;  
22        } else {  
23            suit = initialSuit;  
24        }  
25        visible = false;  
26    }  
27  
28    public boolean equals(Card other) {  
29        boolean notNull = other != null;  
30        boolean sameRank = rank == other.rank;  
31        boolean sameSuit = suit == other.suit;  
32  
33        return notNull && sameRank && sameSuit;  
34    }  
35    // Reste du code inchangé  
36 }
```

L'extrait suivant crée plusieurs cartes et teste leurs équivalences. Les cartes **underflow** et **overflow** sont créées avec des arguments invalides, le constructeur remplacera ces derniers par les valeurs par défaut. En guise d'exercice, nous vous laissons deviner la sortie du programme.

Création de plusieurs cartes

```
1 package ch03;  
2  
3 public class CardEquality {  
4  
5     public static void main(String[] args) {  
6         Card aceSpade = new Card(1, Card.SPADE);  
7         Card copy = new Card(1, Card.SPADE);  
8         Card underflow = new Card(0, ' ');
```

```

9      Card overflow = new Card(42, '🃑');
10
11     System.out.printf(
12         "%d%c equals %d%c ? %b\n",
13         aceSpade.getRank(),
14         aceSpade.getSuit(),
15         copy.getRank(),
16         copy.getSuit(),
17         aceSpade.equals(copy)
18     );
19     System.out.printf(
20         "%d%c equals %d%c ? %b\n",
21         aceSpade.getRank(),
22         aceSpade.getSuit(),
23         underflow.getRank(),
24         underflow.getSuit(),
25         aceSpade.equals(underflow)
26     );
27     System.out.printf(
28         "%d%c equals %d%c ? %b\n",
29         aceSpade.getRank(),
30         aceSpade.getSuit(),
31         overflow.getRank(),
32         overflow.getSuit(),
33         aceSpade.equals(overflow)
34     );
35 }
36 }
```

Comme pour les fonctions classiques, vous pouvez également appeler des méthodes dans d'autres méthodes d'objet. Nous pouvons appeler, par exemple, la méthode `Math.clamp(int val, int min, int max)` qui retourne l'argument `val` si `min≤val≤max` ou la valeur de la borne la plus proche.

Ajout d'une méthode qui appelle une autre méthode

```

1  public static final char SPADE = '♠',
2      HEART = '♥',
3      DIAMOND = '♦',
4      CLUB = '♣';
5
6  int rank;
7  char suit;
8  boolean visible;
9
10 public Card(int initialRank, char initialSuit) {
11     rank = Math.clamp(initialRank, 1, 13);
12     suit = (char) Math.clamp(initialSuit, SPADE, CLUB);
13     visible = false;
14 }
15 // Reste du code inchangé
16 }
```

Cette approche reste discutable. En effet, la création d'une carte invalide résulte plutôt d'une erreur

de programmation. Même si les données viennent de l'utilisateur, vous devriez les avoir validées avant de créer l'objet. Le dernier extrait remplace un argument incorrect par une valeur admise, masquant ainsi le problème.

Dans ce contexte, il est préférable de vérifier le paramètre et, le cas échéant, de lancer une exception. L'extrait suivant respecte ce principe. La méthode de classe `Contract.requireInRange(int val, int min, int max, String message)` vérifie que le premier argument est compris dans l'intervalle `[min; max]`. Si ce n'est pas le cas, elle lance une exception contenant le message fourni comme quatrième argument.

Échec rapide

```
1 import utils.Contract;
2
3 public class Card {
4
5     public static final char SPADE = '♠',
6         HEART = '♥',
7         DIAMOND = '♦',
8         CLUB = '♣';
9
10    int rank;
11    char suit;
12    boolean visible;
13
14    public Card(int initialRank, char initialSuit) {
15        rank = Contract.require(
16            initialRank,
17            1 <= initialRank && initialRank <= 13,
18            "Le rang doit appartenir à l'intervalle [1; 13]. Reçu " +
19            initialRank
20        );
21        suit = Contract.require(
22            initialSuit,
23            SPADE <= initialSuit && initialSuit <= CLUB,
24            "L'enseigne doit être ♠, ♥, ♦ ou ♣. Reçu " + initialSuit
25        );
26        visible = false;
27    }
28    // Reste du code inchangé
29 }
```



Échouer rapidement

Quand le paramètre d'un objet est invalide, il est recommandé de le signaler rapidement avec une exception plutôt que de la rendre silencieuse.

3.6. Les redéfinitions masquent les méthodes héritées

Revenons un instant aux premiers extraits de la [Section 3.1](#). Nous avions signalé que la méthode `printf` appelle la méthode `toString()` héritée d'une classe commune à toutes les autres : `java.lang.Object`. Attardons-nous sur la documentation de cette méthode et en particulier sur le

premier paragraphe.

Returns a string representation of the object. In general, the `toString` method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

La documentation officielle recommande aux héritiers de la classe de redéfinir la méthode `toString()` afin de fournir une représentation informative d'un objet, facile à lire pour une personne^[2].

La représentation par défaut n'est ni informative, ni facile à lire. Elle correspond à la concaténation du nom de la classe concrète de l'objet, c'est-à-dire celle utilisée pour le créer, et d'un code hexadécimal unique^[3] à chaque objet.

La Javadoc nous recommande de redéfinir `toString()` pour obtenir une meilleure représentation.

Redéfinition de méthode

Une classe **A** peut redéfinir une méthode **m** d'une classe **B** si et seulement si :



- **A** hérite de **B** (toutes les classes héritent de la classe **Object**) ;
- **m** est une méthode d'objet, accessible par **A** et non finale.

Redéfinir une méthode consiste à écrire dans **A** une méthode de signature identique à **m** et à lui associer un nouveau corps. Redéfinissons la méthode `toString()` pour la classe **Card**.

Redéfinition de la méthode `toString()`

```
1 public class Card {  
2     // Reste du code inchangé  
3  
4     public String toString() {  
5         return "%d%c".formatted(rank, suit);  
6     }  
7 }
```

Adaptions le programme de mise à jour des scores en transmettant directement des objets à `printf(String, Object...)`. Remarquez que le texte affiché pour une carte respecte le format spécifié par la redéfinition de `toString()`.

Affichage avec `toString()`

```
1 public class CardEquality {  
2  
3     public static void main(String[] args) {  
4         Card aceSpade = new Card(1, Card.SPADE);  
5         Card copy = new Card(1, Card.SPADE);  
6         Card kingHeart = new Card(13, Card.HEART);  
7  
8         System.out.printf(
```

```

9         "%s equals %s ? %b\n",
10        aceSpade,
11        copy,
12        aceSpade.equals(copy)
13    );
14
15    System.out.printf(
16        "%s equals %s ? %b\n",
17        aceSpade,
18        kingHeart,
19        aceSpade.equals(kingHeart)
20    );
21 }
22 }
```

➤ Sortie

```
1♠ equals 1♠ ? true
1♠ equals 13♥ ? false
```

Si aucune représentation naturelle n'existe pour votre classe, faites apparaître les éléments suivants quand vous redéfinissez `toString()` :



- le nom de la classe concrète ;
- pour chaque champ important, son nom et sa valeur.

Les environnements de développement vous proposent des aides pour générer une redéfinition de `toString()`. Elle constitue un bon point de départ, mais elle doit être adaptée.

Une seconde méthode intéressante à redéfinir est la méthode `equals(Object o)` qui détermine si deux objets sont équivalents. Par défaut, la relation d'équivalence définie par la méthode `equals(Object o)` s'établit sur les adresses qu'occupent les objets en mémoire. Pour certains types d'objets, cette relation par défaut donne des situations étonnantes.

Dans l'extrait suivant, nous avons déclaré deux cartes équivalentes. Nous voulons vérifier l'équivalence des deux cartes avec une assertion JUnit. Sans redéfinition de `equals(Object)`, l'assertion n'est pas vérifiée, ce qui provoque l'échec du test.

Équivalence incorrecte entre deux cartes

```

1 class CardTest {
2     @Test
3     void shouldBeEquivalent() {
4         Card aceOfSpade = new Card(1, Card.SPADE);
5         Card aceOfSpadeCopy = new Card(1, Card.SPADE);
6
7         Assertions.assertEquals(aceOfSpade, aceOfSpadeCopy);
8     }
9     //==> expected: ch03.Card@6c2c1385<Card(1♠)> but was: ch03.Card@5f354bcf<Card(1♠)>
10 }
```

Pour réussir ce test, nous devons redéfinir `equals(Object o)`. Attention au respect de la signature : le nom de la méthode et la liste de paramètres^[4] doivent être identiques. Sinon, vous ne redéfinissez pas une méthode : vous la surchargez. Ainsi, la méthode `equals(Card)` est une surcharge : elle n'est pas prise en charge par JUnit.

Redéfinition de la méthode `equals(Object)`

```
1 public class Card {
2     // Code inchangé
3
4     public boolean equals(Object other) {
5         if (this == other) {
6             return true;
7         }
8
9         if (!(other instanceof Card)) {
10            return false;
11        }
12
13        Card toCard = (Card) other;
14
15        return equals(toCard);
16    }
17
18    public boolean equals(Card other) {
19        boolean notNull = other != null;
20        boolean sameRank = rank == other.rank;
21        boolean sameSuit = suit == other.suit;
22
23        return notNull && sameRank && sameSuit;
24    }
25 }
```

Cette redéfinition mérite quelques explications. Premièrement, le paramètre est de type `Object`, mais nous souhaitons vérifier l'égalité entre deux objets `Card`. Pour procéder à la vérification en toute sécurité, nous devons :

- Tester l'égalité entre les références : si les deux références sont égales, c'est qu'elles désignent le même objet et cet objet est équivalent à lui-même.
- Nous assurer que l'objet référencé par `other` est bien un objet de la classe `Card`, ce que nous effectuons à l'aide de l'opérateur booléen `instanceof`. Notez que `instanceof` retourne `false` si `other` vaut `null`.
- Convertir explicitement `other` en une référence à un `Card` pour accéder aux champs pertinents. Sans cette conversion, notre méthode ne voit à travers `other` que les membres définis par la classe `Object`.

Avec cette redéfinition, nous réussissons le test précédent.

L'expression `other instanceof Card` est indispensable pour éviter de convertir des objets qui ne sont pas des cartes à jouer. Sans elle, vous pourriez tenter de convertir, par exemple, des dates en

cartes. Concrètement, une conversion explicite peut lancer une `ClassCastException` qui terminera l'exécution du programme.

La vérification est nécessaire

```
1 @Override
2 public boolean equals(Object other) {
3     // if(!(other instanceof Card)) {
4     //     return false;
5     // }
6     // ✗ Risque de ClassCastException
7     Card toCard = (Card) other;
8
9     return equals(toCard);
10}
11
12 Card c = new Card(1, Card.SPADE);
13 // ✗ Lance ClassCastException
14 c.equals("As de pique");
```

Depuis Java 17, vous pouvez combiner le test du type et la conversion de type en une seule opération `instanceof`. Pour ce faire, vous remplacez le second opérande de `instanceof` par un patron de type (*type pattern*). Ce patron est composé du type ciblé et d'un nom de variable. Cette dernière est initialisée si le premier opérande est un objet du type du second opérande. La portée de la variable correspond aux instructions pour lesquelles le patron de type est vrai.

L'extrait suivant illustre `equals(Object)` avec l'utilisation du patron de type. Le code obtenu est plus concis.

Utilisation d'un patron de type

```
1 public class Card {
2     // Code inchangé
3
4     public boolean equals(Object other) {
5         if(this == other) {
6             return true;
7         }
8
9         return other instanceof Card that
10            && equals(that); // Appelle equals(Card)
11    }
12
13    public boolean equals(Card otherCard) {
14        boolean notNull = otherCard != null;
15        boolean sameRank = rank == otherCard.rank;
16        boolean sameSuit = suit == otherCard.suit;
17
18        return notNull && sameRank && sameSuit;
19    }
20 }
```



Redéfinissez toujours la méthode `toString()` et envisagez de redéfinir la

méthode `equals(Object)`.

En réalité, redéfinir `equals(Object)` nous impose d'autres obligations qui garantissent le fonctionnement de nos objets avec d'autres classes du JDK et les classes qui en hériteront. Nous y reviendrons au [Chapitre 6](#).

3.7. Les champs doivent être encapsulés

Nous avons énoncé un premier principe stipulant qu'un objet doit être dans un état cohérent dès sa création. Si vous ne respectez pas ce principe, vous prenez le risque de créer des objets aux comportements aberrants. Ce principe seul ne garantit pas que vos objets resteront cohérents toute leur vie.

L'extrait suivant affecte un rang négatif à une carte. Nous avons créé une carte correcte, puis nous avons modifié un de ses champs pour la rendre anormale. Nous souhaitons interdire l'accès direct aux champs d'un objet et fournir une protection en cas de modifications. Pour y arriver, nous devons consulter et modifier nos objets uniquement à travers des méthodes.

L'accès direct aux champs est dangereux pour nos objets

```
1 public class WrongCardUsage {  
2  
3     public static void main(String[] args) {  
4         Card aceSpade = new Card(1, Card.SPADE);  
5         aceSpade.rank = 42;  
6         aceSpade.suit = '🃑';  
7  
8         System.out.printf("%s, WTF are you ?%n", aceSpade);  
9     }  
10 }
```

En Java, vous pouvez interdire l'accès aux champs d'un objet depuis l'extérieur en leur ajoutant le modificateur **private**. Vous pouvez aussi ajouter ce modificateur aux méthodes et aux constructeurs. Typiquement, les méthodes utiles à des objets d'un autre type seront associées au modificateur **public**.



Masquez vos champs d'objet en les qualifiant de **private**. Ce faisant, vous gardez le contrôle sur la cohérence de vos objets.

La classe **Card** protège ses champs d'objet en les rendant **private**. Vous ne pouvez plus modifier la visibilité de la face d'une carte qu'avec la méthode **flip()**. De plus, vous pouvez consulter le rang d'une carte avec la méthode **getRank()**, son enseigne avec la méthode **getSuit()** et connaître sa visibilité avec la méthode **isVisible()**.

L'accès aux champs est interdit aux objets qui ne sont pas des cartes

```
1 public class Card {  
2  
3     public static final char SPADE = '♠',  
4         HEART = '♥',  
5         DIAMOND = '♦',  
6         CLUB = '♣';  
7  
8     private int rank;  
9     private char suit;  
10    private boolean visible;  
11}
```

```
12 // Code inchangé  
13 }
```

Les autres classes ne peuvent plus accéder aux champs

```
1 public class WrongCardUsage {  
2  
3     public static void main(String[] args) {  
4         Card aceSpade = new Card(1, Card.SPADE);  
5         // aceSpade.rank = 42; // ✗ les champs sont privés  
6         // aceSpade.suit = '†'; // ✗ les champs sont privés  
7  
8         System.out.printf("%s, WTF are you?%n", aceSpade);  
9     }  
10 }
```

En Java, une méthode retournant une valeur caractéristique d'un objet se nomme accesseur. Elle prend typiquement la forme `<type> getXXX()` où XXX correspond au nom de la propriété. Pour les caractéristiques booléennes, la forme est `boolean isXXX()` où XXX correspond au nom de la propriété.

Si vous souhaitiez changer le rang d'une carte, vous devriez ajouter une méthode à cette fin. En Java, une méthode modifiant une valeur caractéristique d'un objet se nomme mutateur. Elle prend typiquement la forme `void setXXX(<type> newValue)` où XXX correspond au nom de la caractéristique. Grâce aux mutateurs, vous vérifiez facilement que la nouvelle valeur respecte plusieurs règles avant de l'affecter à le champ. Par exemple, vous pourrez refuser la mise à jour du rang d'une carte si l'argument est négatif ou plus grand que 14.

Accesseur

Méthode d'objet pour consulter la valeur d'une propriété de l'objet. En Java, l'accesseur commence par le verbe *get*, ou *is* pour les caractéristiques booléennes.



Mutateur

Méthode d'objet pour modifier la valeur d'une propriété de l'objet. En Java, le mutateur commence par le verbe *set*.

Quand une classe interdit l'accès à ses champs d'objet et qu'elle propose des méthodes pour consulter l'état d'un objet et, éventuellement modifier cet état, nous dirons de cette classe qu'elle encapsule ses champs. L'encapsulation est un principe fondamental de la POO.

Encapsulation



Principe qui consiste à accéder aux données d'un objet, que ce soit en lecture ou en écriture, uniquement par l'intermédiaire de méthodes. L'encapsulation favorise l'intégrité des objets et facilite les futurs changements internes.

Le schéma suivant résume notre déclaration de la classe **Card** avec ses champs, son constructeur et ses méthodes d'objet, en particulier ses accesseurs. Les symboles + et - montrent respectivement

les membres publics et ceux qui sont privés.



Figure 22. Schéma final de la classe

3.8. Les enums définissent des objets par extension

Nous avons étudié les classes lors des sections précédentes. Déclarer une classe, c'est définir un type dont les valeurs sont des objets. Une classe ressemble à un moule à objets : vous ne connaissez pas à l'avance tous les objets qui vont être créés avec elle. On dit d'une classe qu'elle définit ses objets en intention.

Java propose d'autres méthodes pour définir des types d'objets. Si vous connaissez à l'avance le nombre d'objets d'un type et que ce nombre est petit, de l'ordre d'une dizaine d'objets, vous déclarez plutôt une enum.

En programmation, les enums sont des types de données qui associent des codes à un petit ensemble de valeurs. Elles regroupent des valeurs en un type. Leur rôle principal est de faciliter la lecture du code. Contrairement à une classe, une enum définit ses valeurs en extension.

L'extrait suivant déclare une enum en C#. Dans ce langage, une enum se présente comme une liste de codes associés à des `int`. Vous pouvez omettre l'entier associé : chaque code reçoit alors un entier selon sa position dans la liste.

Une enum en C#

```
1 public enum Suit {
2     Spade=0, Heart=1, Diamond=2, CLUB=3
3 }
4
5 //Cette déclaration est équivalente à la précédente
```

```
6 public enum Suit {  
7     Spade, Heart, Diamond, CLUB  
8 }
```

Contrairement à C#, **les enum de Java définissent des types d'objets**. Comme ce sont des objets, Java interdit l'initialisation d'un code avec un entier. L'extrait suivant définit l'enum **Suit** en Java. Si la syntaxe est identique à l'écriture raccourcie en C#, les ressemblances s'arrêtent là : les enums proposés par Java sont en réalité des classes^[5] dont on connaît tous les objets.

Enum pour les enseignes en Java

```
1 public enum Suit {  
2     SPADE, HEART, DIAMOND, CLUB  
3 }
```

Cette section étudie les enum de Java. Nous les comparerons notamment aux classes afin de mettre en évidence les cas où les enums sont préférables aux classes, et inversement.

3.8.1. Le nombre d'objets d'une enum est fixé à sa déclaration

Contrairement aux classes, Java interdit la création de nouveaux objets avec les enum. Le langage vous limite aux objets énumérés. Si vous tentez malgré tout de créer un nouvel objet depuis une enum, vous obtiendrez l'erreur suivante.

Utilisation incorrecte d'une enum pour créer un objet

```
1 ./Playground/SuitPlayground.java:7: error: enum types may not be instantiated  
2 Suit mySuit = new Suit();  
3                                     ^  
4 1 error
```

L'extrait suivant déclare une référence à une enum et l'initialise correctement. Comme vous le voyez, vous accédez aux objets à travers le nom de l'enum. Cette manière est semblable à celle utilisée pour accéder aux membres **static** d'une classe.

Utilisation correcte d'une enum

```
1 public class SuitPlayground {  
2     public static void main(String[ ] args) {  
3         Suit mySuit = Suit.SPADE;  
4         System.out.printf("my suit is %s", mySuit);  
5     }  
6 }  
7  
8 // ==> My suit is SPADE
```

Autre élément notable, par défaut, la méthode **toString()** d'une enum retourne le nom de l'objet, **SPADE** dans l'extrait précédent, plutôt qu'un string étrange comme **"Suit@1234567"**. Le compilateur garnit notre enum de redéfinitions et de nouvelles méthodes.

Parmi ces nouvelles méthodes, citons `values()` que vous utiliserez fréquemment dans une boucle `foreach`. Plus généralement, les enums s'intègrent aux structures de contrôle.

3.8.2. Les enums s'intègrent aux structures de contrôle

Contrairement aux classes, Java intègre naturellement les enums aux structures `switch` et `for(:)`, comme les extraits suivants le présentent. La méthode de classe `values()` retourne un tableau des objets de l'enum, dans l'ordre de leur déclaration. Cette meilleure intégration s'explique par la connaissance des objets de l'enum dès la compilation, connaissance impossible à avoir avec des classes, car leurs objets sont connus seulement pendant l'exécution du programme.

Valeurs d'une enum et structures de contrôle

```
1 public static boolean isBlack(Suit suit) {
2     return switch(suit) {
3         case SPADE, CLUB -> true;
4         case HEART, HEART -> false;
5     };
6 }
7
8 public static void printColors() {
9     for(Suit s : Suit.values()) {
10         System.out.printf("%s --> %s%n", s, isBlack(s) ? "B" : "R");
11     }
12 }
```

3.8.3. Les enums sont convertibles en string

Comme les extraits précédents le montrent, vous pouvez obtenir une représentation textuelle des valeurs d'une enum avec la méthode `toString()`. À l'inverse, vous pouvez obtenir la constante correspondant à un string avec la fonction `valueOf(String)`. **Attention**, si aucune correspondance n'est trouvée, Java lance une exception `IllegalArgumentException`.

Valeur énumérée d'un string

```
1 public class SuitPlayground {
2     public static void main(String[ ] args) {
3         Suit fromString = Suit.valueOf("SPADE");
4         System.out.printf("From string is %s\n", fromString);
5     }
6 }
```

➤ Sortie

```
From string is SPADE
```

3.8.4. Les enums sont comparables

Les objets d'une enum supportent naturellement l'égalité, car il est impossible de créer de nouveaux objets avec une enum. Comme vous ne pouvez pas créer plusieurs exemplaires de la

même valeur, les relations d'égalité entre références et entre objets se confondent.

Les objets d'une enum sont également comparables selon l'ordre de leurs déclarations. Pour comparer deux variables de types enums, appelez la méthode `compareTo(...)`. Elle retourne un entier négatif, valant 0 ou positif selon que l'objet qui reçoit l'appel vient avant, est égal à, ou vient après l'objet reçu en argument. Par ailleurs, vous pouvez connaître la position d'un objet d'une enum avec la méthode `ordinal()`.

```
1 public class SuitPlayground {
2
3     public static void main(String[] args) {
4         Suit[] suits = Suit.values();
5         for (int i = 0; i < suits.length; ++i) {
6             System.out.printf(" |%7s <=> ", suits[i]);
7             for (int j = 0; j < suits.length; ++j) {
8                 System.out.print(
9                     " | %7s : %2d",
10                     suits[j],
11                     suits[i].compareTo(suits[j])
12                 );
13             }
14             System.out.printf(" |%n");
15         }
16     }
17 }
```

➤ Sortie

| | | | | |
|-------------|-----------|------------|--------------|-----------|
| SPADE <=> | SPADE : 0 | HEART : -1 | DIAMOND : -2 | CLUB : -3 |
| HEART <=> | SPADE : 1 | HEART : 0 | DIAMOND : -1 | CLUB : -2 |
| DIAMOND <=> | SPADE : 2 | HEART : 1 | DIAMOND : 0 | CLUB : -1 |
| CLUB <=> | SPADE : 3 | HEART : 2 | DIAMOND : 1 | CLUB : 0 |

Java recommande de ne pas compter sur l'ordre des déclarations des objets pour implémenter les comportements, cet ordre pouvant changer pour d'autres raisons. La bonne pratique consiste à déclarer un champ caractéristique de cet ordre et à l'initialiser avec un constructeur. Voyons comment faire.

3.8.5. Une enum peut avoir des champs et des méthodes

Les enums sont traduites en classes par le compilateur. Comme pour une classe, vous pouvez y déclarer des champs, des constructeurs et des méthodes.

Dans l'extrait suivant, chaque valeur reçoit un argument utilisé par le constructeur déclaré après le champ. Nous définissons une méthode pour connaître la couleur de l'enseigne. Nous redéfinissons également `toString()`.

Déclaration de champs et de méthodes

```
1 enum Suit {
2     SPADE('♠'),
```

```

3     HEART('♥'),
4     DIAMOND('♦'),
5     CLUB('♣');
6
7     private final char symbol;
8
9     private Suit(char givenSymbol) {
10         this.symbol = givenSymbol;
11     }
12
13    public boolean isBlack() {
14        return switch (this) {
15            case SPADE, CLUB -> true;
16            case HEART, DIAMOND -> false;
17        };
18    }
19
20    public String toString() {
21        return "%c".formatted(symbol);
22    }
23 }
```

La redéfinition de `toString()` modifie la sortie du programme précédent, sans autres adaptations de notre part.

➤ Sortie modifiée

| |
|--------------------------------------|
| ♠ <=> ♠ : 0 ♥ : -1 ♦ : -2 ♣ : -3 |
| ♥ <=> ♠ : 1 ♥ : 0 ♦ : -1 ♣ : -2 |
| ♦ <=> ♠ : 2 ♥ : 1 ♦ : 0 ♣ : -1 |
| ♣ <=> ♠ : 3 ♥ : 2 ♦ : 1 ♣ : 0 |

Notez le modificateur `final` associé aux déclarations des champs. Il interdit la modification du champ après son initialisation. Java recommande d'appliquer systématiquement `final` aux champs d'une enum. Grâce à lui, vous êtes certains que vos objets ne muteront plus une fois initialisés : ils seront immuables.



Utilisez le modificateur `final` au moment de déclarer vos champs pour rendre vos objets immuables.

Les classes et les enums sont les deux constructions historiques pour implémenter des types d'objet en Java. Le tableau ci-dessous les compare. Une troisième construction existe, les enregistrements. Elle sera étudiée dans le cours de programmation avancée.

| Critère | Classe | Enum |
|----------------------------|-----------|-----------|
| Approche | Intention | Extension |
| Nombre d'objets | ∞ | < 20 |
| Opérateur <code>new</code> | ✓ | ✗ |
| Membres | ✓ | ✓ |

| Critère | Classe | Enum |
|----------------------|---------------|-------------|
| Égalité des objets | À redéfinir | Généré |
| Conversion en string | À redéfinir | Généré |
| Comparaison | À implémenter | Généré |

3.9. En résumé

Dans ce chapitre, nous avons défini de nouveaux types d'objets à l'aide des classes. Plus qu'un conteneur de fonctions, le rôle principal d'une classe est de définir des types d'objets. Déclarer une classe revient à lui donner un nom et à déclarer des membres. Parmi les membres, nous avons vu :

1. les champs qui sont des variables décrivant l'état de l'objet ;
2. les méthodes qui spécifient les services qu'un objet peut rendre ;
3. les constructeurs qui initialisent un objet créé.

Les méthodes et les constructeurs sont des fonctions. Quand l'en-tête d'une fonction est dépourvue du modificateur **static**, nous parlerons de méthode d'objet : elle a besoin d'un objet pour être appelée. Un constructeur est appellable uniquement à la création d'un objet, en combinaison avec l'opérateur **new**. Les champs sont des variables accessibles aux méthodes et aux constructeurs déclarés dans la classe.

Toutes les classes héritent des méthodes de la classe **Object**. Ces méthodes peuvent être redéfinies pour leur donner un sens plus proche de la classe déclarée. La redéfinition de **equals(Object)** en particulier vous permet d'implémenter la relation d'égalité entre les objets d'une même classe.

En Java, les enums sont une seconde manière de définir des types d'objets. Contrairement aux classes, vous devez connaître tous les objets qui composent votre enum au moment de la déclarer. Les enums sont à privilégier quand vous connaissez tous les objets et qu'ils sont peu nombreux (de l'ordre d'une dizaine). Comme vous ne pouvez pas créer de nouveaux objets avec une enum, l'association d'égalité par défaut suffit. De plus, la méthode **toString()** est redéfinie adéquatement. Enfin, les enums s'intègrent naturellement à certaines structures de contrôle.

Nous avons étudié comment déclarer de nouveaux types en Java avec les classes et les enums. Dans le chapitre suivant, nous nous intéresserons au cycle de vie des objets.

- [1] Quand une carte est face cachée, on dit aussi qu'elle montre son dos.
- [2] Personne capable de programmer bien entendu
- [3] Il s'agit d'une représentation hexadécimale du haché de l'objet.
- [4] À l'exception de leurs noms.
- [5] Le compilateur Java traduit l'enum en une classe.

Chapitre 4. Le cycle de vie des objets

Ce chapitre étudie le cycle de vie des objets, en particulier leur création et leur destruction.

Tout objet devrait respecter deux principes :

- Être utilisable dès sa création, ce qui suppose qu'il soit dans un état adéquat ;
- Libérer les ressources qui lui ont été allouées quand il n'en a plus besoin.

Garantir qu'un objet créé soit directement utilisable prévient les bugs. S'assurer de la libération des ressources évite les blocages du système, faute de disponibilité. Nous pensons à la mémoire vive, mais ces blocages peuvent également concerter des fichiers, des connexions à une BD, du matériel spécifique comme un GPU^[1], etc.

Contrairement à des langages comme C, Java propose plusieurs mécanismes pour régir ces problèmes. Vous initialisez les objets à l'aide de méthodes spéciales appelées au moment de créer un objet : ce sont les constructeurs. Java encadre aussi la libération de la mémoire allouée aux objets à l'aide du ramasse-miettes^[2].

Ce chapitre étudie ces aspects et met l'accent sur quelques bonnes pratiques.

4.1. Le constructeur initialise un objet

Java propose un concept dédié à l'initialisation d'un objet : les constructeurs. Grâce aux constructeurs, vous définissez la logique d'initialisation d'un objet à un seul endroit. Les utilisateurs de vos objets ne pourront plus oublier de les initialiser, sous peine d'erreurs à la compilation. Pour peu que les préconditions des paramètres soient correctement documentées, le risque de créer des objets inutilisables diminue.

Constructeur

Méthode appelée uniquement pendant la création d'un objet pour l'initialiser.

Un constructeur possède un en-tête particulier qui le distingue des méthodes classiques. En Java, et pour la majorité des descendants de C++, le constructeur porte le même nom que celui de sa classe et ne possède pas de type de retour. Comme il ne retourne pas de valeur, l'instruction `return <expression>` y est interdite.

Reprendons la définition de classe `Card` de la fin du chapitre précédent. Nous y avons défini un constructeur acceptant deux paramètres : le rang et la suite de la carte. Concrètement, son nom est celui de sa classe, il ne retourne aucun résultat et son corps ne compte pas d'instructions `return <expression>`.

Définition d'un constructeur

```
1 public class Card {  
2  
3     private int rank;
```

```

4   private Suit suit;
5   private boolean visible;
6
7   public Card(int initialRank, Suit initialSuit) {
8       rank = Contract.require(
9           initialRank,
10          1 <= initialRank && initialRank <= 13,
11          "Le rang doit appartenir à l'intervalle [1; 13]. Reçu " +
12          initialRank
13      );
14      suit = Contract.require(
15          initialSuit,
16          initialSuit != null,
17          "L'enseigne doit être ♠, ♥, ♦ ou ♣. Reçu " + initialSuit
18      );
19      visible = false;
20  }
21 }
```

Le constructeur doit respecter les invariants de l'objet, c'est-à-dire des conditions à remplir pendant toute sa vie. Dans l'exemple d'une carte à jouer, son enseigne est définie et prend une des valeurs de `Suit` et son rang est dans l'intervalle `[1; 14]`. Pour garantir cet invariant d'objet, le constructeur valide les paramètres et, le cas échéant, lève une exception.

Spécifiez les invariants d'un objet et les préconditions des paramètres



Spécifiez les invariants, c'est-à-dire les conditions respectées pendant toute la vie d'un objet ou d'une méthode. Spécifiez aussi les préconditions que les paramètres du constructeur devraient respecter pour respecter les invariants.

Avec le constructeur déclaré, nous pouvons créer des `Card` avec l'opérateur `new` en transmettant au constructeur des arguments.

Création de deux cartes

```
Card spadeAce = new Card(1, Suit.SPADE);
Card twoTrebol = new Card(2, Suit.CLUB);
```

Comme nous l'avons vu à la fin de la [Section 3.4](#), définir notre constructeur fait que nous n'avons plus de constructeurs par défaut, sans paramètres. Avec notre dernière version, l'instruction `Card defaultCard = new Card();` est illégale. Vous obtiendrez le message suivant.

Tentative de création d'une carte avec le constructeur par défaut

```
Error:
constructor Card in class Card cannot be applied to given types;
required: int, Suit
found: no arguments
reason: actual and formal argument lists differ in length
Card defaultCard = new Card();
^-----^
```

Revenons un instant sur le constructeur par défaut et sur son comportement.

4.1.1. Le constructeur par défaut

Les premières sections du chapitre précédent utilisent le constructeur par défaut généré par le langage, faute de constructeur écrit par nos soins. Ce constructeur ne prend aucun paramètre : on l'appelle parfois constructeur sans paramètres.

Si aucune valeur initiale n'a été donnée, le constructeur par défaut initialise un champ avec la valeur par défaut de son type. Le [Table 7](#) liste les valeurs par défaut de chaque type. Notez que la valeur par défaut d'une référence est `null` !

Table 7. Valeur par défaut par type

| Type | Valeur par défaut | Taille en octets | Représentation |
|----------------------|-----------------------|------------------|-------------------|
| <code>byte</code> | <code>0</code> | 1 | Entier signé |
| <code>short</code> | <code>0</code> | 2 | Entier signé |
| <code>int</code> | <code>0</code> | 4 | Entier signé |
| <code>long</code> | <code>0L</code> | 8 | Entier signé |
| <code>float</code> | <code>0.0f</code> | 4 | Standard IEEE 754 |
| <code>double</code> | <code>0.0d</code> | 8 | Standard IEEE 754 |
| <code>char</code> | <code>'\u0000'</code> | 2 | Table UTF-16 |
| <code>boolean</code> | <code>false</code> | ? | Indéfini |
| Référence | <code>null</code> | 4 | À définir |

Constructeur par défaut



Java génère un constructeur par défaut, sans paramètres, quand vous n'en déclarez aucun dans une classe. Ce constructeur initialise chaque champ de l'objet avec la valeur par défaut de son type, sauf si vous avez fourni une valeur initiale.

Disposer d'un constructeur sans paramètres est utile dans de nombreuses occasions, pour peu que l'état initial de l'objet construit soit valide. Il est cependant tout aussi utile de disposer de plusieurs constructeurs pour initialiser nos objets selon nos besoins. Dans ce cas, nous déclarerons des surcharges de constructeurs.

4.1.2. Surcharges de constructeurs

Comme toute méthode, vous pouvez déclarer des surcharges de constructeur. Chaque surcharge se distingue des autres par sa liste de paramètres. Les surcharges offrent alors plusieurs possibilités de créer des objets. L'extrait de code suivant présente deux surcharges de constructeur. Dans cet extrait, les listes des paramètres des constructeurs se distinguent par leurs longueurs.

La classe Player possède deux constructeurs

```
1 public class Player {
2
3     private String name;
4     private int highestScore;
5
6     public Player(String initialName, int initialScore) {
7         name = Contract.require(
8             initialName,
9             initialName != null && !initialName.isBlank(),
10            "Le nom doit être non vide"
11        );
12        highestScore = Contract.require(
13            initialScore,
14            0 <= initialScore && initialScore <= 100,
15            "Le score doit être dans [0; 100]. Reçu " + initialScore
16        );
17    }
18
19    public Player(String initialName) {
20        name = Contract.require(
21            initialName,
22            initialName != null && !initialName.isBlank(),
23            "Le nom doit être non vide"
24        );
25        highestScore = 0;
26    }
27
28    public void setHighestScore(int newScore) {
29        if (newScore > highestScore) {
30            highestScore = newScore;
31        }
32    }
33
34    public int getHighestScore() {
35        return highestScore;
36    }
37
38    public String getName() {
39        return name;
40    }
41
42    public String toString() {
43        return String.format("Player(name: %s, score: %d)", name, highestScore);
44    }
45 }
```

Surcharges de méthodes

N-uplet ($N \geq 2$) de méthodes de même nom, se distinguant par leurs listes de paramètres. Deux listes de paramètres sont distinctes si elles varient en nombre ou si deux paramètres de même position n'ont pas le même type.

Rappelons que, puisque nous avons défini nos propres constructeurs, Java ne génère pas de constructeur par défaut. L'expression `new Player()` reste illégale... sauf si nous la déclarons

explicitement. Parmi les surcharges possibles d'un constructeur, le constructeur de copie occupe une place particulière.

4.1.3. Cloner les objets : le constructeur par copie

Un constructeur par copie prend comme seul paramètre un autre objet de la même classe et s'en sert pour initialiser les champs de l'objet construit.

Ajoutons un constructeur par copie à la classe `Player`. Ce dernier valide l'argument de `other`. Si ce dernier ne vaut pas `null`, nous pouvons affecter les valeurs de ses champs. Sinon, nous affectons les valeurs par défaut.

La classe Player possède dorénavant trois constructeurs

```
1 public class Player {  
2  
3     private String name;  
4     private int highestScore;  
5  
6     public Player(Player original) {  
7         Contract.require(original != null, "L'objet original doit être défini");  
8         name = original.getName();  
9         highestScore = original.getHighestScore();  
10    }  
11    //Reste du code inchangé  
12 }
```

L'extrait suivant et la Figure 23 illustrent la copie du joueur "Simon". Remarquez que le nom de `simon` et celui de sa copie `simonCloned` référencent le même objet en mémoire. Notre constructeur affecte une référence à une autre : elles référencent le même objet. Si nous souhaitions éviter cette situation, nous devrions utiliser le constructeur par copie prévu par `String`. Cependant, comme les strings ne mutent pas en Java, nous ne prenons aucun risque de modifier par accident l'état d'un objet.

Création de joueurs par copie

```
1 public class PlayerCopy {  
2  
3     public static void main(String[] args) {  
4         Player simon = new Player("Simon", 42);  
5         Player henry = new Player("Henry");  
6         Player simonCloned = new Player(simon);  
7  
8         System.out.printf("simon = %s\n", simon);  
9         System.out.printf("simon = %s\n", henry);  
10        System.out.printf("simon cloned = %s\n", simonCloned);  
11    }  
12 }
```

➤ Sortie

```
1 simon = Player(name: Simon, score: 42)
```

```

2 simon = Player(name: Henry, score: 0)
3 simon cloned = Player(name: Simon, score: 42)

```

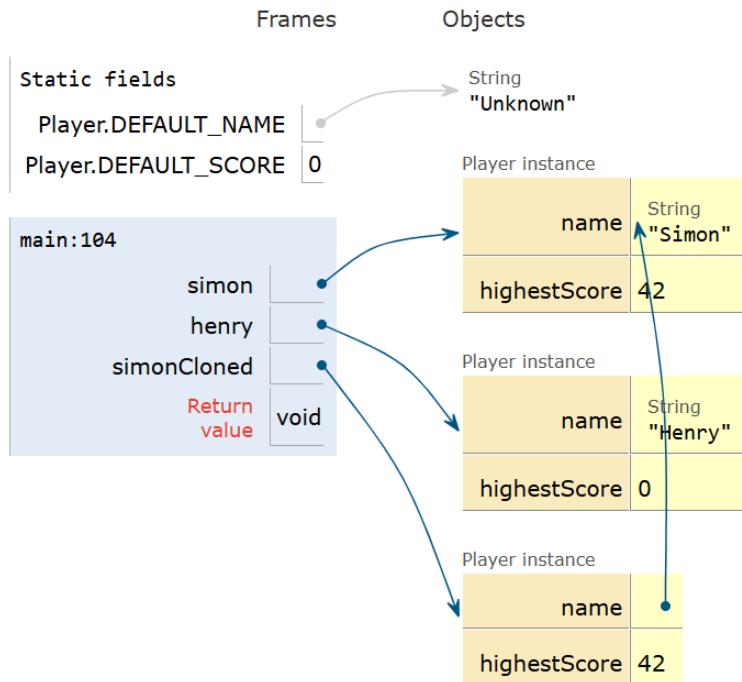


Figure 23. Attention aux affectations de références avec un constructeur par copie

Nos constructeurs ont des instructions similaires. Il est tentant d'essayer de factoriser le code répété en appelant un constructeur dans un autre constructeur. Appeler un constructeur dans un autre doit se faire avec précaution. Débutons par la découverte du mot-clé `this`, qui nous permettra de procéder à de tels appels.

4.2. `this` référence l'objet qui reçoit l'appel

Nous avons vu au chapitre 2 qu'appeler une méthode d'objet nécessite un objet. Par exemple, avec la classe `Player`, l'appel `Player.getName()` est illégal.

Appel de méthode d'objet illégal

```

| Error:
| non-static method getName() cannot be referenced
| from a static context
| Player.getName();
| ^-----^

```

Supposons une méthode d'objet `hasABetterScoreThan(Player other)` qui retourne `true` si le score du joueur qui reçoit l'appel est meilleur que celui de l'autre joueur. Pour ne pas dupliquer du code, nous appellerons la méthode `getScore()` dans le corps de la méthode `hasABetterScoreThan(Player other)`. Cette méthode est une méthode d'objet : un objet reçoit donc l'appel, mais comment identifier cet objet ?

Qui reçoit l'appel `getScore()`

```
1 public boolean hasABetterScoreThan(Player other) {  
2     return getScore() > other.getScore();  
3 }
```

La réponse la plus évidente est la bonne : l'objet qui a reçu l'appel à `hasABetterScoreThan(Player)` est celui qui reçoit l'appel à `getScore()`. Sans mention du contraire, l'objet qui reçoit l'appel à une méthode interne est celui qui a reçu l'appel initial. Nous pouvons expliciter cet objet destinataire grâce au mot-clé `this` qui joue alors le rôle de **référence prédefinie** sur l'objet qui exécute la méthode.

Explicitation du destinataire de l'appel

```
1 public boolean hasABetterScoreThan(Player other) {  
2     return this.getScore() > other.getScore();  
3     //Est équivalent à :  
4     //return getScore() > other.getScore();  
5 }
```

`this` fait référence à l'objet qui reçoit l'appel. Nous pouvons l'utiliser dans les corps des méthodes d'objet. Nous approfondissons les conséquences de cette propriété dans les sous-sections suivantes.

4.2.1. `this` est une référence à un objet

`this` se révèle utile pour lever d'éventuelles ambiguïtés entre deux variables. Par exemple, quand un paramètre et un champ portent le même nom. En effet, sans mention à `this`, Java choisit la variable la plus locale, le paramètre dans notre exemple.

Quand il est utilisé comme résultat d'un appel, `this` facilite également l'enchaînement des appels sur un même objet, ce qui peut améliorer la lecture du code. Avec ce mécanisme, vous pouvez, par exemple, chaîner les appels à un `StringJoiner`.

Méthodes retournant la référence à cet objet

```
1 import java.util.StringJoiner;  
2  
3 public class ThisSample {  
4     public static void main(String[] args) {  
5         StringJoiner arrayJoiner = new StringJoiner(", ", "[", "]");  
6  
7         arrayJoiner.add("A").add("B").add("C");  
8  
9         System.out.printf("array = %s\n", arrayJoiner.toString());  
10    }  
11    //==> array = [A, B, C]  
12 }
```

Le corps de la méthode `StringJoiner.add(String)` se termine par une instruction `return this`. Cette instruction fait que tous les appels à `add(String)` sont adressés au même objet, celui

référencé par **arrayJoiner** dans notre exemple.

Plusieurs classes proposent la possibilité de chaîner les appels. Cependant, ces chaînes peuvent ne pas travailler sur le même objet. C'est le cas avec les strings. Pour rappel, les objets de la classe **String** sont immuables. Cette propriété d'immuabilité est désirable pour des objets représentant des valeurs sans identité propre ou des quantités.

4.2.2. **this** n'existe pas dans une méthode de classe

Par définition, une méthode de classe n'est exécutée par aucun objet. **this** n'a dès lors aucun sens dans ce contexte : Java interdit l'utilisation de **this** dans une méthode **static**. De plus, comme une méthode n'est exécutée par aucun objet, Java vous oblige à appeler une méthode d'objet avec une référence explicite, autre que **this**.



Java interdit l'accès à **this** dans le corps d'une méthode statique ainsi que l'appel à une méthode d'objet sans objet récepteur, cet objet n'existant pas dans un contexte statique.



Comme nous l'avons vu au chapitre précédent, l'inverse est cependant possible : une méthode d'objet peut appeler des méthodes **static**.

4.2.3. **this** peut appeler un constructeur

Reprenons les constructeurs de notre classe **Player**. Leurs instructions se ressemblent beaucoup : elles diffèrent d'une valeur. Nous souhaitons supprimer ces répétitions. À cette fin, Java propose d'utiliser **this** comme nom d'appel à un second constructeur.

L'extrait suivant montre un exemple d'un tel appel dans le second constructeur. Java déduit la surcharge appelée de la liste des arguments.

Appels à un constructeur dans un autre constructeur

```
1 public class Player {  
2  
3     private String name;  
4     private int highestScore;  
5  
6     public Player(String initialName, int initialScore) {  
7         name = Contract.require(  
8             initialName,  
9             initialName != null && !initialName.isBlank(),  
10            "Le nom doit être non vide"  
11        );  
12        highestScore = Contract.require(  
13            initialScore,  
14            0 <= initialScore && initialScore <= 100,  
15            "Le score doit être dans [0; 100]. Reçu " + initialScore  
16        );  
17    }  
18  
19    public Player(String initialName) {
```

```

20         this(initialName, 0);
21     }
22
23     public Player(Player original) {
24         this(
25             Contract.require(
26                 original,
27                 original != null,
28                 "L'objet original doit être défini"
29             ).getName(),
30             original.getHighestScore()
31         );
32     }
33
34     // Code inchangé
35 }
```

L'appel à un constructeur dans un autre constructeur est soumis à quelques règles :

- Cet appel doit être la première instruction du constructeur. Cette raison explique la forme du constructeur par copie, qui appelle `Objects.requireNonNull(original)` au moment de fournir le premier argument.
- Il doit exister au moins un constructeur qui n'appelle pas un autre constructeur pour éviter les appels infinis.

Don't Repeat Yourself



Une bonne pratique consiste à concentrer le code d'initialisation dans le constructeur ayant la liste de paramètres la plus longue. Les autres surcharges appellent des constructeurs aux listes de plus en plus longues en fournissant des valeurs par défaut pour les arguments manquants.

4.3. Les méthodes de fabrique statique masquent les constructeurs

Un débat existe au sein des programmeurs pour savoir quel outil privilégier entre les constructeurs et les méthodes de fabrique statiques, telles que `LocalDate.now()`. Nous ne donnerons pas une réponse ferme dans cette section. Nous pointons cependant quelques indicateurs :

1. Une méthode de fabrique peut valider les arguments avant la création effective de l'objet. Cette validation avant la création évite l'allocation inutile de mémoire.
2. Une méthode de fabrique peut avoir un nom autre que celui de la classe. Ce faisant, elle peut être plus expressive. `LocalDate.now()` est plus clair que `new LocalDate()`.
3. Plusieurs méthodes de fabrique peuvent avoir la même liste de paramètres tant qu'elles se distinguent par leurs noms. Par exemple, `LocalDate.fromEpoch(int days)` et `LocalDate.fromToday(int days)` sont deux méthodes légales.
4. Les constructeurs sont intégrés au langage et demandent moins de code pour assurer

l'encapsulation.

5. Comme nous le découvrirons au chapitre suivant, les constructeurs restent indispensables pour initialiser les champs hérités d'un objet.

Un principe général consiste à privilégier les constructeurs quand l'objet sera toujours créé, même en cas d'arguments invalides, ou lorsqu'un objet doit initialiser les parties dont il hérite. Dans les autres cas, les méthodes de fabrique sont recommandées. D'ailleurs, chaque nouvelle version du JDK apporte son lot de nouvelles méthodes de fabrique.

4.4. Les champs sont initialisés dans un ordre prédéfini

Quand vous déclarez une variable dans une méthode, Java vous impose de lui donner une valeur initiale sous peine d'erreurs de compilation. Le problème ne se pose pas lorsque vous déclarez des champs. Généralement, vous les initialisez dans un constructeur. Une alternative consiste à initialiser le champ dès sa déclaration, comme dans l'exemple suivant :

Le champ visible est initialisé au moment de sa déclaration

```
1  private int rank;
2  private Suit suit;
3  private boolean visible;
4
5  public Card(int initialRank, Suit initialSuit) {
6      rank = Contract.require(
7          initialRank,
8          1 <= initialRank && initialRank <= 13,
9          "Le rang doit appartenir à l'intervalle [1; 13]. Reçu " +
10         initialRank
11     );
12     suit = Contract.require(
13         initialSuit,
14         initialSuit != null,
15         "L'enseigne doit être ♠, ♥, ♦ ou ♣. Reçu " + initialSuit
16     //Reste du code inchangé
17 }
```

Vous pouvez vous poser alors plusieurs questions :

- Existe-t-il un ordre d'initialisation ?
- Les constructeurs sont-ils appelés avant ou après l'initialisation des champs ?
- Initialisations et appels aux constructeurs sont-ils exécutés aléatoirement ?
- Etc.

Voici une version simplifiée de la règle suivie par Java. Elle est simplifiée, car nous n'avons pas encore discuté des champs statiques.

Ordre d'initialisation (version simple)

Java commence par initialiser les champs d'une classe dans l'ordre de leur déclaration. Java

appelle ensuite le constructeur associé à `new`.

Prenons l'extrait suivant. Si vous inspectez les messages affichés par le programme, vous constaterez que Java initialise tout d'abord les champs, même s'ils sont mélangés aux méthodes. Vient ensuite le constructeur et, une fois l'objet initialisé, l'appel à la méthode `f`.

Démonstration de l'ordre d'initialisation (B. Eckel; Thinking in Java)

```
1 class Tag {
2     Tag(int marker) {
3         System.out.println("Tag(" + marker + ")");
4     }
5 }
6
7 class TagTriple {
8     Tag t1 = new Tag(1);
9     TagTriple() {
10         System.out.println("Card()");
11         t3 = new Tag(33);
12     }
13     Tag t2 = new Tag(2);
14     void f() {
15         System.out.println("f()");
16     }
17     Tag t3 = new Tag(3);
18 }
19
20 public class OrderOfInitialization {
21     public static void main(String[] args) {
22         TagTriple t = new TagTriple();
23         t.f(); // Shows that construction is done
24     }
25     //==>Tag(1)
26     //==>Tag(2)
27     //==>Tag(3)
28     //==>TagTriple()
29     //==>Tag(33)
30     //==>f()
31 }
```

4.4.1. champs statiques

Lorsque votre classe possède un champ statique, dont la valeur est commune à chaque objet, vous procéderez de la même façon. Notons que l'initialisation du champ statique a lieu une et une seule fois, avant de créer le premier objet ou d'accéder à l'un de ses membres statiques pour la première fois.

En plus de l'initialisation d'un champ statique au moment de sa déclaration, Java prévoit la possibilité d'écrire des séquences d'instructions statiques. Ces séquences seront également exécutées une unique fois, après l'initialisation des champs statiques.

Initialisation d'un champ statique

```
1 class Game {  
2     public static int gamesCount = 0;  
3  
4     static {  
5         gamesCount = 42;  
6     }  
7  
8     private int id = Game.gamesCount++;  
9     private int playersCount;  
10  
11    public Game(int playersCount) {  
12        this.playersCount = playersCount;  
13    }  
14  
15    public int getId() {  
16        return id;  
17    }  
18 }  
19  
20 public class OrderOfInitializationFull {  
21     public static void main(String[] args) {  
22         Game game1 = new Game(2);  
23         Game game2 = new Game(3);  
24  
25         System.out.printf("game 1 id = %d\n", game1.getId());  
26         System.out.printf("game 2 id = %d\n", game2.getId());  
27     }  
28     //==> game 1 id = 42  
29     //==> game 2 id = 43  
30 }
```

Initialiser la valeur d'un champ statique dans un constructeur peut poser un problème. En effet, chaque objet créé provoque l'appel d'un constructeur. La valeur du champ statique sera écrasée à chaque appel.

Pour conclure, l'ordre d'initialisation des champs respecte la règle suivante :

Ordre d'initialisation (version complète)

- Java commence par initialiser les champs de classe dans l'ordre de leur déclaration.
- Java exécute ensuite les séquences d'instructions statiques.
- Java initialise les champs d'objets dans l'ordre de leur déclaration.
- Java appelle ensuite le constructeur associé à `new`.

Fréquence d'initialisation des champs

La fréquence d'initialisation est fonction de la catégorie du champ :

- Les champs de classe sont initialisés une et une seule fois lors du premier accès à la classe ;
- Les champs d'objets sont initialisés à chaque création d'objet.

4.5. Les objets peuvent être détruits

Les paramètres et les variables locales d'une méthode font partie de la mémoire allouée à cette méthode, c'est le fameux cadre d'exécution. Java détruit ce cadre dès la fin de l'exécution de l'appel de méthode. Java gère les cadres d'exécution dans une zone appelée pile.

Comme les variables locales et les paramètres, Java alloue également de la mémoire à chaque objet créé. Dans la foulée, il exécute un constructeur pour initialiser tout nouvel objet. Cet objet vit sur le tas. Comme il vit dans un espace mémoire différent de celui des méthodes, lorsque la méthode qui a créé un objet prend fin, l'objet continue d'exister en mémoire.



Les objets vivent sur **le tas**. Cette zone mémoire est indépendante de la **pile**.

Pendant l'exécution d'un programme, des objets deviennent inutiles. Le programme a tout intérêt à récupérer la mémoire allouée à ces objets pour l'utiliser de nouveau. Java ne vous donne pas cette responsabilité. Un mécanisme de ramasse-miettes (*garbage collection*) assure cette tâche.

L'application appelle le ramasse-miettes lorsqu'elle manque de mémoire. Quand il est invoqué, le ramasse-miette détermine les objets devenus inutilisables par le programme. Le ramasse-miettes appelle la méthode `finalize` des objets à détruire. Cette méthode, à redéfinir, doit libérer la mémoire allouée de façon spéciale, notamment via l'appel à des méthodes natives. Après cette série d'appels, le ramasse-miettes libère effectivement la mémoire allouée aux objets inutiles qui sont alors détruits.

Une exécution peut ne jamais appeler le ramasse-miettes. Dans ce cas, toute la mémoire allouée aux objets est libérée d'un coup à la fin du programme. Cette approche est préférable du point de vue des performances : appeler le ramasse-miettes consomme des ressources CPU.



Cette méthode a une conséquence importante, à ne pas sous-estimer si vous avez alloué des ressources externes au programme : un objet peut ne jamais recevoir l'appel `finalize` et, par conséquent, ne jamais libérer les ressources. Comment garantir que ces ressources soient libérées ? Tout simplement en déterminant vous-mêmes le moment où l'objet devient inutile.

Depuis la version 7 du langage, Java propose le bloc de ressources (*try with resources*) pour encadrer l'allocation et la libération des ressources. L'allocation et la libération des ressources seront abordées par les cours du bloc 2.

La [Figure 24](#) résume le cycle de vie d'un objet Java.

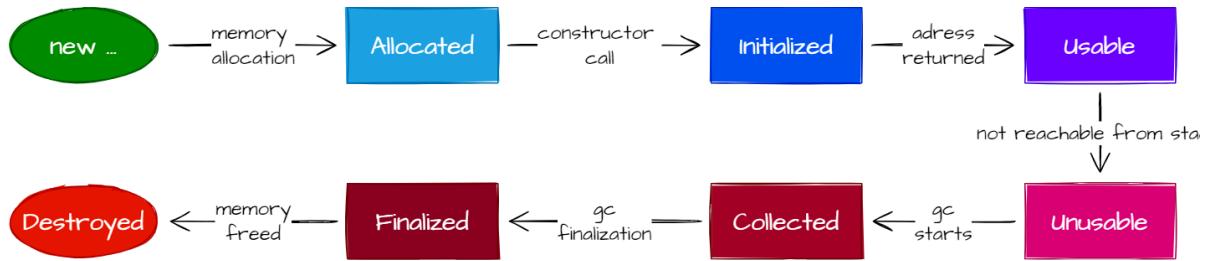


Figure 24. Cycle de vie d'un objet Java

4.5.1. Comment le ramasse-miettes identifie-t-il un objet inutilisable ?



Cette section est là à titre d'information et ne fait pas partie de la matière à maîtriser.

Un objet devient inutilisable quand aucune référence accessible ne le désigne. Cela signifie qu'aucun objet ne peut le manipuler. Une technique consiste alors à mémoriser pour chaque objet créé **le nombre de références qui le désignent**. Lorsque cette valeur tombe à 0, l'objet est déclaré inutilisable et pourra être détruit à la prochaine collecte.

Cette technique présente cependant un inconvénient important lorsque deux objets se réfèrent mutuellement.

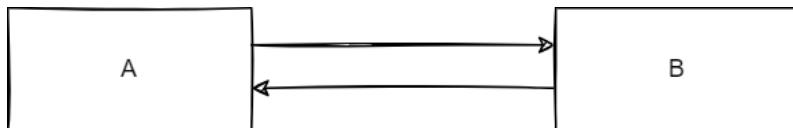


Figure 25. Deux objets se référant mutuellement

Dans ce cas, les compteurs de référence n'atteindront jamais 0. Pourtant, les deux objets seront inutilisables dès que les compteurs de A et de B valent 1 ! Pour résoudre le problème, des concepts supplémentaires doivent être introduits tels que la notion de référence faible (*weak reference*).

L'approche précédente, qui est utilisée par d'autres langages de plus bas niveau comme le C++ et le Swift, n'est pas implémentée par les JVM. Sans entrer dans les détails, elles partent du principe qu'un objet est vivant si et seulement s'il existe un chemin dont :

- le point de départ est une référence vivant dans la pile ou dans la zone de mémoire statique (celle dédiée aux constantes);
- les nœuds sont formés d'objets accessibles à partir du point de départ ;
- le point d'arrivée est l'objet considéré.

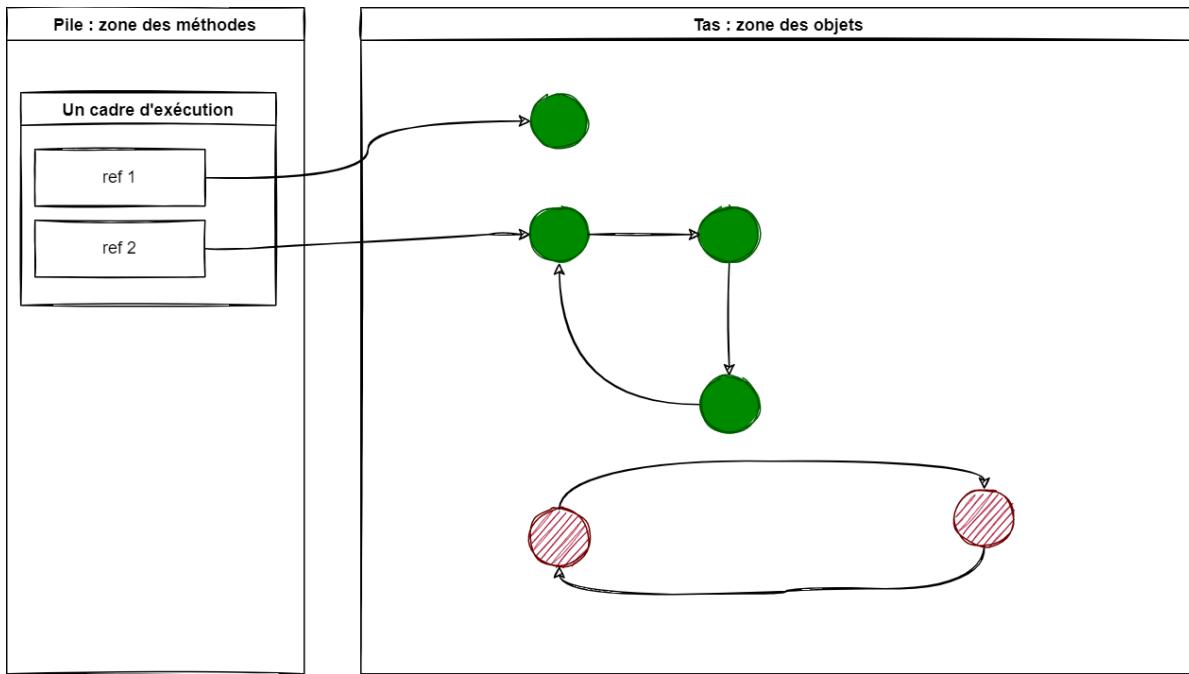


Figure 26. Objets vivants (en verts) et morts (en orange)

Les objets inutilisables ne possèdent pas de tels chemins. Partant de ce principe, le ramasse-miettes génère ces chemins, déplace les objets trouvés dans une zone de sauvegarde et détruit dans la foulée les objets restants. Notons que dans ce cas de figure, les références cycliques ne posent plus de problèmes ! Par contre, la collecte prendra plus de temps puisqu'il faut découvrir les objets vivants et les déplacer dans une zone sûre...

4.6. En résumé

Ce chapitre parle des mécanismes de création et de destruction des objets. En Java, vous vous intéressez tout particulièrement à l'initialisation des objets.

Mettre un objet dans un état initial adéquat dès sa création est une bonne pratique de programmation. Java propose à cette fin la notion de constructeur, méthode spéciale appelée uniquement au moment de créer l'objet. Vous pouvez définir plusieurs constructeurs se distinguant par leur liste de paramètres. Les constructeurs peuvent s'appeler, ce qui évite la duplication du code d'initialisation. Si le programmeur n'en code pas, Java génère un constructeur par défaut qui affecte à chaque champ la valeur par défaut de son type.

Outre les constructeurs, vous pouvez initialiser les champs au moment de les déclarer. L'initialisation a toujours lieu avant l'appel aux constructeurs et se fait toujours dans l'ordre de déclaration des champs. Dans le cas des champs de classe, ces derniers sont initialisés une et une seule fois, avant la première utilisation de la classe.

Les objets vivent et peuvent mourir lorsqu'ils deviennent inutilisables. Lorsqu'elle manque de mémoire, une JVM lance le processus de ramasse-miettes consommateur de ressources. Comme le processus n'est pas exécuté systématiquement, vous devrez libérer les ressources externes à la JVM explicitement et en temps utile.

- [1] *Graphical Processing Unit*, c'est-à-dire une carte graphique.
[2] *Garbage collector* en anglais

Chapitre 5. Les relations de composition et d'héritage

Ce chapitre explique comment réutiliser du code en combinant des objets, c'est la composition, ou en créant des hiérarchies de classes, c'est l'héritage, tout en évitant les pièges courants.

Jusqu'à présent, nous avons défini des classes d'objets simples dont les champs étaient de types primitifs, de type **string** ou énumérés. Elles rendent déjà des services intéressants et améliorent la lecture du code.

Ce chapitre explore davantage les types de relations entre les objets. Ces relations représentent des techniques de réutilisation du code.

Nous commencerons par étudier la composition, qui consiste à créer un objet par assemblage d'autres objets qui forment ses composants. Ce type de relation convient pour exprimer des phrases telles que « *Tout objet X possède un ou plusieurs objets Y* ».

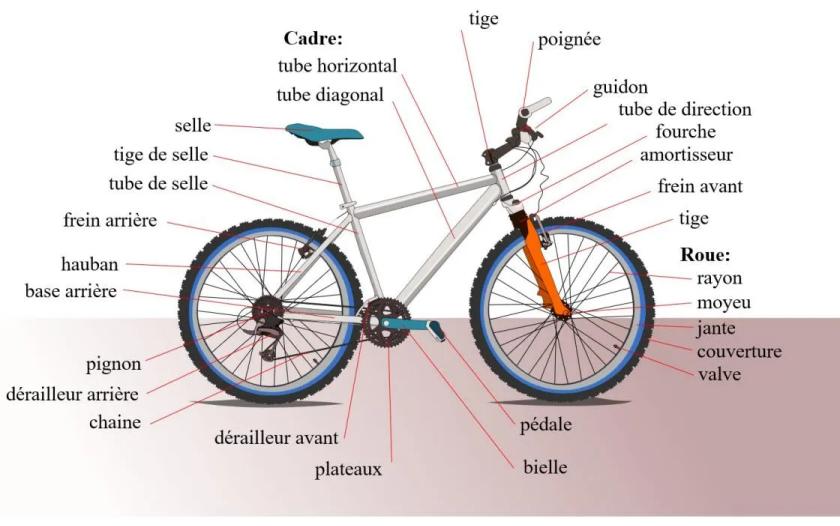
Nous poursuivrons notre étude par l'héritage, qui consiste à déclarer qu'un objet appartient à une sous-classe d'une classe plus générale. L'héritage convient pour exprimer des relations telles que « *X est une sorte de Y qui ...* ».

Nous terminerons par quelques recommandations sur les cas d'utilisation de ces relations.

5.1. La composition assemble des objets

Composer consiste à assembler divers éléments pour en former un nouveau, plus intéressant. Par exemple, un vélo ([Figure 27](#)) résulte de l'assemblage d'éléments tels que le cadre, la fourche, des roues, un système de frein, etc. Ces éléments se composent eux-mêmes de sous-éléments. Une voiture, quant à elle, assemble en moyenne 30 000 pièces. Pour l'utilisateur occasionnel, la voiture et le vélo sont plus intéressants que leurs parties : ils vous véhiculent d'un endroit à un autre. Vous percevez probablement moins de valeur dans un moteur ou dans une roue.

Les différentes parties du vélo :



Creative Commons licence by AI2

Figure 27. Une machine résulte de l'assemblage d'éléments plus simples

En programmation, vous utilisez cette technique quand vous définissez le comportement d'une méthode en appelant d'autres méthodes. Vous l'utilisez également quand les arguments d'un appel de méthode sont les résultats d'autres appels de méthodes. Cette forme de composition vous aide à :

- Résoudre un problème en combinant les solutions de sous-problèmes plus simples.
- Réutiliser du code déjà écrit.

En POO, la composition porte sur les objets plutôt que sur les méthodes. Un objet est créé en assemblant d'autres objets. L'objet créé porte le nom de composite, tandis que les objets assemblés portent le nom de composants. Comme l'assemblage dure toute la vie du composite, les composants prennent la forme de champ.

La composition



Relation où un objet résulte de l'assemblage d'autres objets qui forment ses composants. En POO, les compositions prennent la forme de champs correspondant aux composants.

Une relation de composition exprime une phrase du type «**Tout objet [Composite] possède [Cardinalité] [Composant] faisant office de [Rôle]**». Le composite est l'objet qui résulte de l'assemblage. La cardinalité définit les nombres minimum et maximum de composants pouvant intervenir pour chaque objet composite. Le rôle correspond typiquement au nom porté par les composants dans le composite, il peut être omis.

La [Figure 28](#) présente la représentation graphique type d'une relation de composition. Le losange est placé du côté du composite, mais est souvent omis. La flèche du côté du composant. Les cardinalités et le rôle du composant sont placés du côté du composant.

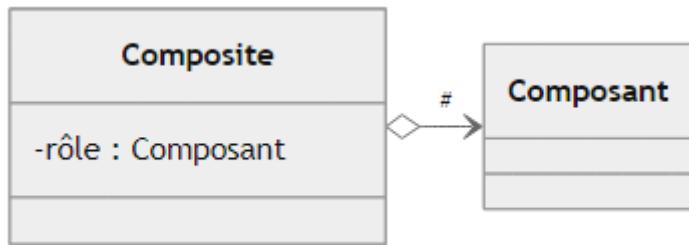


Figure 28. Représentation graphique d'une composition

Prenons quelques exemples :

- « Toute [Carte] possède [une] [Enseigne] »,
- « Toute [Carte] possède [un] [Rank] »,
- « Toute [Main de cartes] possède de [0 à N] [Carte] jouant le rôle de [cartes] ».

Graphiquement, nous représenterons la dernière relation comme sur la [Figure 29](#). La cardinalité **[0 à N]** y est notée **0..n** ou **0:n**, parfois *****.

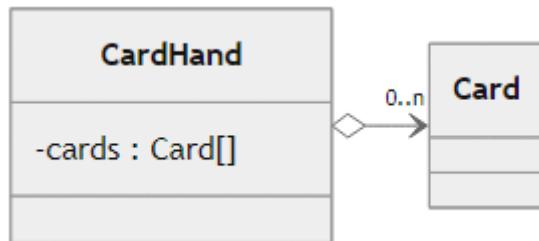


Figure 29. Représentation complète d'une composition

Intéressons-nous aux cardinalités maximales. Deux cas sont intéressants :

- si la cardinalité vaut au plus 1, le composant est monovalué ;
- si la cardinalité vaut plus que 1, le composant est multivalué.

5.1.1. Composants monovalués

Un composant est monovalué si sa cardinalité maximale vaut 1. **Il prend la forme d'un champ correspondant à une référence à un objet.**

Illustrons ce concept avec une carte à jouer. *Toute carte à jouer possède un rang et une enseigne.* Nous avons déjà défini les enseignes par une énumération ([Section 3.8.5](#)). Nous pouvons faire de même avec les rangs. En effet, il y a treize rangs connus. La [Figure 30](#) illustre ces relations.

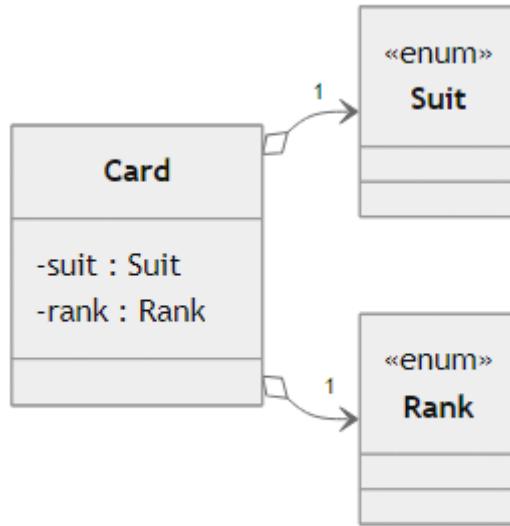


Figure 30. Composants monovalués

L'extrait suivant présente une version remaniée de la classe **Card**. Tout objet de cette classe sera composé d'une enseigne et d'une référence à un rang. Nous souhaitons que ces composants soient définis pour toute nouvelle carte. En conséquence, le constructeur vérifie l'absence de valeur **null** pour les deux paramètres correspondants.

Une première composition

```

1 public class Card {
2
3     private Rank rank;
4     private Suit suit;
5     private boolean visible;
6
7     public Card(Rank rank, Suit suit) {
8         this.rank = Contract.require(
9             rank,
10            rank != null,
11            "Le rang doit être défini."
12        );
13        this.suit = Contract.require(
14            suit,
15            suit != null,
16            "L'enseigne doit être défini. "
17        );
18        visible = false;
19    }
20
21    public Rank getRank() {
22        return rank;
23    }
24
25    public Suit getSuit() {
26        return suit;
27    }
28
29    public boolean isVisible() {
30        return visible;
31    }
  
```

```

32
33     public void flip() {
34         visible = !visible;
35     }
36
37     public String toString() {
38         if (this.isVisible()) {
39             return "%s%s".formatted(rank, suit);
40         } else {
41             return "??";
42         }
43     }
44
45     public boolean equals(Object other) {
46         if (this == other) {
47             return true;
48         }
49         return other instanceof Card that && this.equals(that);
50     }
51
52     public boolean equals(Card other) {
53         boolean notNull = other != null;
54         boolean sameRank = rank == other.rank;
55         boolean sameSuit = suit == other.suit;
56
57         return notNull && sameRank && sameSuit;
58     }
59
60     public int compareTo(Card that) {
61         Objects.requireNonNull(that);
62         int rankComparison = this.rank.compareTo(that.rank);
63
64         return rankComparison == 0
65             ? this.suit.compareTo(that.suit)
66             : rankComparison;
67     }
68 }
```

Nous avons apporté quelques modifications aux redéfinitions de `toString()` et de `equals(Objects)`:

- La méthode `Card.toString()` appelle implicitement les méthodes `Rank.toString()` et `Suit.toString()` de ses composants. Plus précisément, la méthode `String.formatted(Object...)` appelle ces méthodes dans son corps.
- La méthode `equals(Object)` condense en une instruction `return <expression>` le test d'appartenance à un type et l'appel à la surcharge (voir [Section 3.6](#)).

Nous avons ajouté une méthode pour comparer deux cartes. La méthode `Card.compareTo(Card)` définit un ordre entre les objets d'une carte. Elle retourne

- un entier négatif si la carte qui reçoit l'appel précède celle reçue en argument;
- un entier positif si la carte qui reçoit l'appel suit celle reçue en argument;

- l'entier **0** si les deux cartes occupent la même position dans la relation d'ordre.

Cette méthode compare d'abord les cartes sur base de leurs rangs. En cas d'égalité, elle utilise leurs enseignes pour déterminer l'ordre. La composition d'objets encourage la déclaration de méthodes qui délèguent une partie de leurs opérations aux méthodes des composants. Cette forme de délégation représente l'un des attraits majeurs de la POO, car elle correspond à l'organisation de nos machines.

L'extrait suivant construit un jeu de cartes à partir des rangs et des enseignes, les mélange et affiche le jeu mélangé. Remarquez qu'un tableau d'objets se manipule comme un tableau d'entiers.

Création d'objets composés

```

1 import java.util.Arrays;
2 import java.util.Random;
3 import java.util.random.RandomGenerator;
4
5 public class CardsPackShuffling {
6
7     public static void main(String[] args) {
8         Card[] cardsPack = new Card[Rank.values().length *
9             Suit.values().length];
10        // Créer le jeu
11        int index = 0;
12        for (Rank rank : Rank.values()) {
13            for (Suit suit : Suit.values()) {
14                cardsPack[index] = new Card(rank, suit);
15                cardsPack[index].flip();
16                ++index;
17            }
18        }
19        //Après les boucles imbriquées, index == Rank.values().length *
20        Suit.values().length();
21
22        // Mélanger le jeu
23        var randomSequence = Random.from(RandomGenerator.getDefault());
24        for (int swapCount = 0; swapCount < 100; ++swapCount) {
25            int firstIndex = randomSequence.nextInt(cardsPack.length);
26            int secondIndex = randomSequence.nextInt(cardsPack.length);
27
28            Card temp = cardsPack[firstIndex];
29            cardsPack[firstIndex] = cardsPack[secondIndex];
30            cardsPack[secondIndex] = temp;
31        }
32        // Afficher le jeu
33        System.out.println(Arrays.toString(cardsPack));
34    }
35 }
```

➤ Sortie

```
[6♥, 3♠, 12♠, 9♠, 11♣, 4♣, 11♠, 8♣, 6♦, 5♣, 4♥, 10♠, 12♥, 2♦, 6♠, 5♦, 7♠, 1♣, 5♣,
1♣, 5♥, 13♥, 7♦, 7♥, 3♦, 3♥, 10♣, 11♦, 10♥, 13♣, 13♦, 4♦, 8♠, 1♦, 11♥, 8♥, 4♣, 9♦,
```

```
[13♠, 7♣, 8♦, 9♥, 2♣, 12♣, 12♦, 6♣, 2♥, 10♦, 9♣, 1♥, 3♣, 2♠]
```

Allons plus loin et définissons des relations de composition multivaluées.

5.1.2. Composants multivalués

Un composant est multivalué si sa cardinalité maximale vaut N, avec $N > 1$. Il prend la forme d'une référence à une collection d'objets du type du composant.

Définissons un type pour les mains de cartes (*cards hand*) au blackjack. Ses objets devront calculer leurs valeurs selon les règles du blackjack. L'algorithme à implémenter tient compte des deux valeurs que peut prendre un as : un ou onze. Pour y arriver, l'algorithme explore les scores possibles et retient le score le plus proche de 21, et idéalement inférieur.

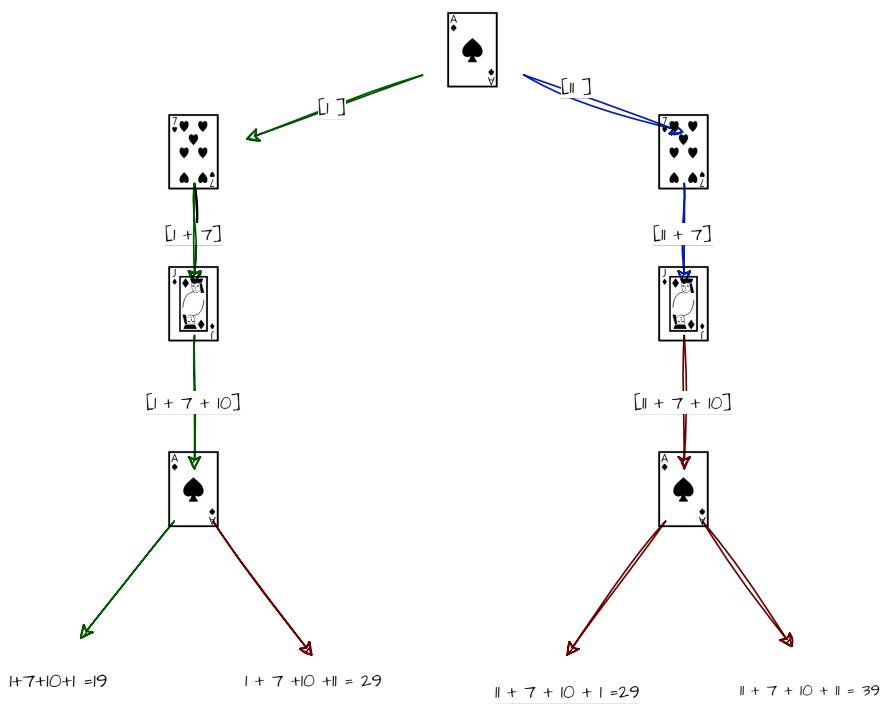


Figure 31. Exploration des combinaisons possibles

L'extrait suivant déclare la classe `CardHand`. Il déclare un champ dont le type est un tableau de cartes. Ce champ est initialisé par le constructeur et manipulé par la méthode `getValue(int, int)` pour calculer le score.

Une composition de plusieurs valeurs

```

1 import java.util.Arrays;
2 import utils.Contract;
3
4 public class CardHand {
5
6     private Card[] cards;
7
8     public CardHand(Card... cards) {
9         Contract.require(cards != null, "cards doit être défini");
10        this.cards = new Card[cards.length];
11        int nextIndex = 0;

```

```

12     for (Card card : cards) {
13         this.cards[nextIndex] = Contract.require(
14             card,
15             card != null,
16             "card[" + nextIndex + "] est indéfini"
17         );
18         ++nextIndex;
19     }
20 }
21
22 public int getValue() {
23     return getValue(0, 0);
24 }
25
26 private int getValue(int branchValue, int index) {
27     if (index >= cards.length) {
28         // Quand cette condition est vérifiée,
29         // la valeur de la main est connue pour cette branche
30         return branchValue;
31     }
32
33     if (Rank.ACE.equals(cards[index].getRank())) {
34         // La carte inspectée est un as.
35         // En conséquence, deux valeurs sont à explorer
36         int valueWithOne = getValue(branchValue + 1, index + 1);
37         int valueWithEleven = getValue(branchValue + 11, index + 1);
38         int deltaWithOne = valueWithOne - 21;
39         int deltaWithEleven = valueWithEleven - 21;
40
41         if (valueWithOne == 21 || valueWithEleven == 21) {
42             return 21;
43         }
44
45         return Math.min(deltaWithOne, deltaWithEleven) == deltaWithOne
46             ? valueWithOne
47             : valueWithEleven;
48     } else if (cards[index].getRank().isFigure()) {
49         return getValue(branchValue + 10, index + 1);
50     } else {
51         return getValue(
52             branchValue + cards[index].getRank().getValue(),
53             index + 1
54         );
55     }
56 }
57
58 public String toString() {
59     return String.format("(hand: %s)", Arrays.toString(cards));
60 }
61 }
```

`CardHand.getValue(int, int)` implémente un algorithme récursif, une famille d'algorithmes étudiée en algorithmique. Toutefois, retenez que :

- La méthode est récursive, car elle s'appelle dans son corps ;

- La méthode se termine, car l'argument correspondant à `index` transmis aux appels récursifs tend vers la condition d'arrêt (il augmente de 1 à chaque appel) ;
- Le premier paramètre accumule des résultats intermédiaires, ce qui évite de nombreux calculs.

L'extrait suivant construit deux mains et affiche leurs composants et leurs valeurs.

Une composition de plusieurs valeurs

```

1 public class CardHandSample {
2
3     public static void main(String[] args) {
4         Card fourthSpade = new Card(Rank.FOUR, Suit.SPADE);
5         Card secondHeart = new Card(Rank.TWO, Suit.HEART);
6         Card sixthDiamond = new Card(Rank.SIX, Suit.DIAMOND);
7         Card aceClub = new Card(Rank.ACE, Suit.TREBOL);
8         Card tenDiamond = new Card(Rank.TEN, Suit.DIAMOND);
9
10        CardHand hand1 = new CardHand(
11            fourthSpade,
12            secondHeart,
13            sixthDiamond,
14            aceClub
15        );
16
17        CardHand hand2 = new CardHand(aceClub, tenDiamond);
18
19        System.out.printf("%s, value = %2d%n", hand1, hand1.getValue());
20        System.out.printf("%s, value = %2d%n", hand2, hand2.getValue());
21    }
22 }
```

➤ Sortie

```
(hand: [??, ??, ??, ??]), value = 13
(hand: [??, ??]), value = 21
```

La classe `CardHand` est un exemple de réutilisation de code. En effet, pour connaître la valeur d'une main, il suffit de la lui demander. Sans ce type, nous devrions programmer l'algorithme nous-mêmes. Manipuler une main est également plus simple que de manipuler le tableau : le programme se contente d'appeler quelques méthodes. Ces dernières masquent les boucles et les manipulations du tableau. Dit autrement, le composite s'utilise plus simplement que ses composants.

Le composite est plus simple que ses composants



Le composite devrait être plus simple à utiliser que la somme de ses composants. Ce conseil est lié au principe *Tell Don't Ask* présenté dans la seconde partie.

En revanche, vous devez prendre garde aux données que vos objets exposeront. Si vous ne prenez aucune précaution, vous risquez de rendre vos objets invalides à cause de modifications extérieures.

5.1.3. Protéger ses composants de l'extérieur : les copies défensives

Plutôt que d'affecter directement la référence au tableau de cartes reçu, le constructeur en fait une copie. Cette pratique évite à nos mains de subir des modifications externes à son insu. Cependant, cette protection seule ne suffit pas.

Si vous regardez la sortie du programme précédent, vous constaterez que les cartes montrent leur dos. Nous voulons qu'une main montre la face spécifique d'une carte. Pour ce faire, nous appelons la méthode `flip()` sur les cartes reçues.

Une mutation naïve

```
1 public class CardHand {
2
3     private Card[] cards;
4
5     public CardHand(Card... cards) {
6         Contract.require(cards != null, "cards doit être défini");
7         this.cards = new Card[cards.length];
8         int nextIndex = 0;
9         for (Card card : cards) {
10             this.cards[nextIndex] = Contract.require(
11                 card,
12                 card != null,
13                 "card[" + nextIndex + "] est indéfini"
14             );
15             this.cards[nextIndex].flip();
16             ++nextIndex;
17         }
18     }
19     // Code inchangé
20 }
```

L'exécution du programme précédent affiche une sortie inattendue : certaines cartes restent masquées.

➤ Sortie

```
(hand: [4♠, 2♡, 6♦, ??]), value = 13
(hand: [??, 10♦]), value = 21
```

Si vous étudiez la fonction principale, vous constaterez que nos deux mains sont composées de cartes communes. Quand une carte commune reçoit deux appels à `flip()`, elle retourne à l'état « face cachée » après le second appel. Ceci explique « l'absence » de changement d'état pour ces cartes. Notez qu'un nombre impair de mains masquerait le problème...

Trouvez-vous normal que deux mains partagent des cartes ? Dans notre monde réel, cela ne peut pas arriver : si plusieurs exemplaires sont autorisés, elles correspondent à des copies physiques indépendantes. Dans notre monde virtuel, en revanche, nous avons la preuve que cela est possible. Pour prévenir ce phénomène, nous pouvons créer des copies des cartes reçues. Ce faisant, nous réalisons une copie profonde du tableau reçu. Cette copie profonde résout le problème d'affichage.

Une mutation sur une copie

```
1 public class CardHand {  
2  
3     private Card[] cards;  
4  
5     public CardHand(Card... cards) {  
6         Contract.require(cards != null, "cards doit être défini");  
7         this.cards = new Card[cards.length];  
8         int nextIndex = 0;  
9         for (Card card : cards) {  
10             this.cards[nextIndex] = new Card(card);  
11             this.cards[nextIndex].flip();  
12             ++nextIndex;  
13         }  
14     }  
15     // Code inchangé  
16 }
```

➤ Sortie

```
(hand: [4♠, 2♡, 6♦, 1♣]), value = 13  
(hand: [1♣, 10♦]), value = 21
```

Créer une copie profonde du tableau reçu corrige notre problème. Elle n'est cependant pas satisfaisante. En effet, plus nous avons de mains à créer, plus le nombre de cartes copiées augmente : nous occupons plus de mémoire. Pour réduire l'empreinte sur la mémoire, deux alternatives sont possibles :

- Tester l'état des cartes reçues et retourner seulement celles qui montrent leur dos,
- S'assurer, dans la méthode principale, que chaque main reçoit des cartes non partagées avec les autres.

Cette dernière approche nous semble la plus pertinente. En effet, elle modélise mieux la réalité : une même carte ne peut pas se trouver à deux endroits en même temps.

La problématique des modifications externes non prévues se pose aussi quand vous déclarez des accesseurs. La méthode suivante, ajoutée à la classe **CardHand**, nous permet de modifier une main à son insu.

Cet accesseur expose directement le champ

```
1 public class CardHand {  
2     // Code inchangé  
3  
4     public PlayingCard[] getAllCards() {  
5         return cards;  
6     }  
7 }
```

Avec cet accesseur, les manipulations suivantes enfreignent les invariants de l'objet et provoquent

la levée d'une exception.

Destruction externe de l'état d'un objet

```
1 CardHand hand1 = new CardHand(fourthSpade, secondHeart, sixthDiamond, aceClub);
2
3 System.out.printf("%s, value = %2d%n", hand1, hand1.getValue());
4
5 Card[] tab = hand1.getAllCards();
6 tab[1] = null;
7 System.out.printf("%s, value = %2d%n", hand1, hand1.getValue());
8 // ✗ NullPointerException
9 // Cannot invoke "ch05.Card.getRank()" because "this.cards[index]" is null
```

Pour prévenir ce problème, retournez systématiquement une copie, éventuellement profonde, du tableau.

Cet accesseur n'expose pas le champ

```
1 public class CardHand {
2     // Code inchangé
3
4     public PlayingCard[] getAllCards() {
5         return Arrays.copyOf(cards, cards.length);
6     }
7 }
```

Copies défensives



Créez des copies des tableaux que vous affectez à vos champs. Faites de même avec les tableaux que vous retournez.

Créer des copies défensives n'est pas sans inconvénients : recopier un tableau à chaque appel pénalise les performances. Cependant, les performances importent souvent moins que l'intégrité des objets. Dans tous les cas, soignez la documentation de vos méthodes et indiquez quand ces dernières retournent des copies.

Nous avons fini notre étude de la composition. Cette relation est probablement la plus utilisée en POO. Son idée est simple : définir le comportement des objets d'une classe en combinant ceux d'autres objets. Dans la section suivante, nous introduisons un second type de relation : l'héritage de classe.

5.2. L'héritage définit des sous-classes

En Java, toute classe déclarée possède des méthodes prédéfinies telles que `toString()` ou `equals(Object)`. Ces méthodes sont déclarées dans la classe `java.lang.Object`. En Java, toute classe déclarée hérite des membres de `java.lang.Object`. Plus généralement, une classe peut hériter des membres d'une autre classe. Grâce à ce mécanisme, la classe qui hérite possède un comportement de base que vous pouvez enrichir et adapter (voir Section 5.5).

Héritage de classe



Relation où une classe reçoit les comportements définis par une autre classe.
La classe qui hérite peut enrichir et modifier ces comportements.

Une relation d'héritage exprime des relations du type « [Classe fille] est une sorte de [Classe mère] qui [comportements spéciaux] ». La classe mère correspond à une définition générale que la classe fille précise :

- Un [Avion] est une sorte de [Véhicule] qui se déplace dans les airs
- Un [Ordinateur portable] est une sorte de [Ordinateur] qui peut fonctionner sans alimentation
- Une [Carte à jouer] est une sorte de [Carte] qui est comparable sur base de son rang et de son enseigne.

Nous pouvons représenter ces relations avec des ensembles (Figure 32). Toutes les classes définies correspondent à des sous-ensembles propres de la classe `java.lang.Object`. Un sous-ensemble peut, à son tour, admettre d'autres sous-ensembles.

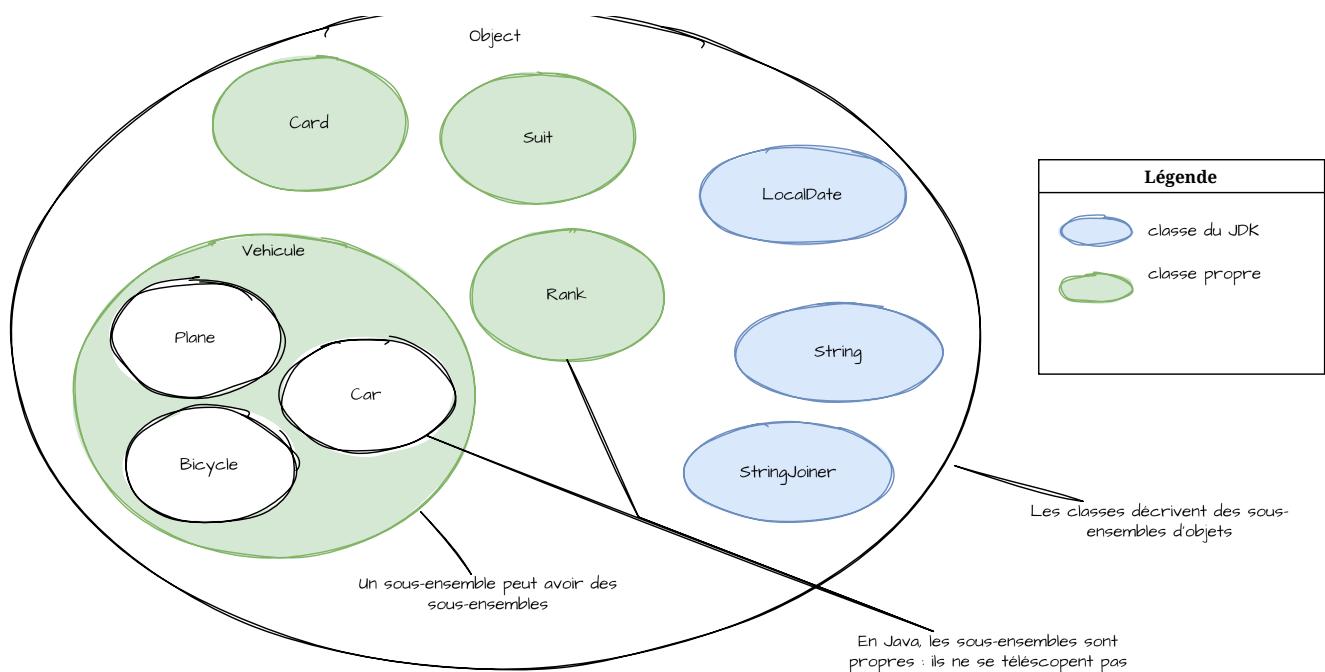


Figure 32. Les relations d'héritage vues comme des ensembles

Un autre point de vue fréquent est de modéliser les relations d'héritage comme un arbre. Un arbre est un graphe acyclique, c'est-à-dire sans cycles, et connexe, il est d'un seul tenant. Les sommets d'un arbre sans parents sont appelés racines, tandis que les sommets sans enfants sont appelés feuilles.

Les classes forment un arbre dont l'unique racine est la classe `Object`. Toute classe qui hérite directement de `Object` est directement connectée à cette dernière. Plus généralement, toute classe `X` qui hérite directement de `Y` sera connectée à cette dernière. Prenez une classe et remontez de parent en parent, et vous tomberez sur `java.lang.Object`. Cette dernière remarque implique que l'héritage est une relation transitive.

Transitivité de l'héritage



La relation d'héritage est transitive : si **X** hérite de **Y** et que **Y** hérite de **Z**, alors **X** hérite de **Z**. **Z** est un ancêtre de **X** et **X** est un descendant de **Z**.

En pratique, la hiérarchie des classes Java est tellement grande que seuls des fragments sont utilisés. La [Figure 33](#) présente la hiérarchie des classes pour Swing, une boîte à outils graphique.

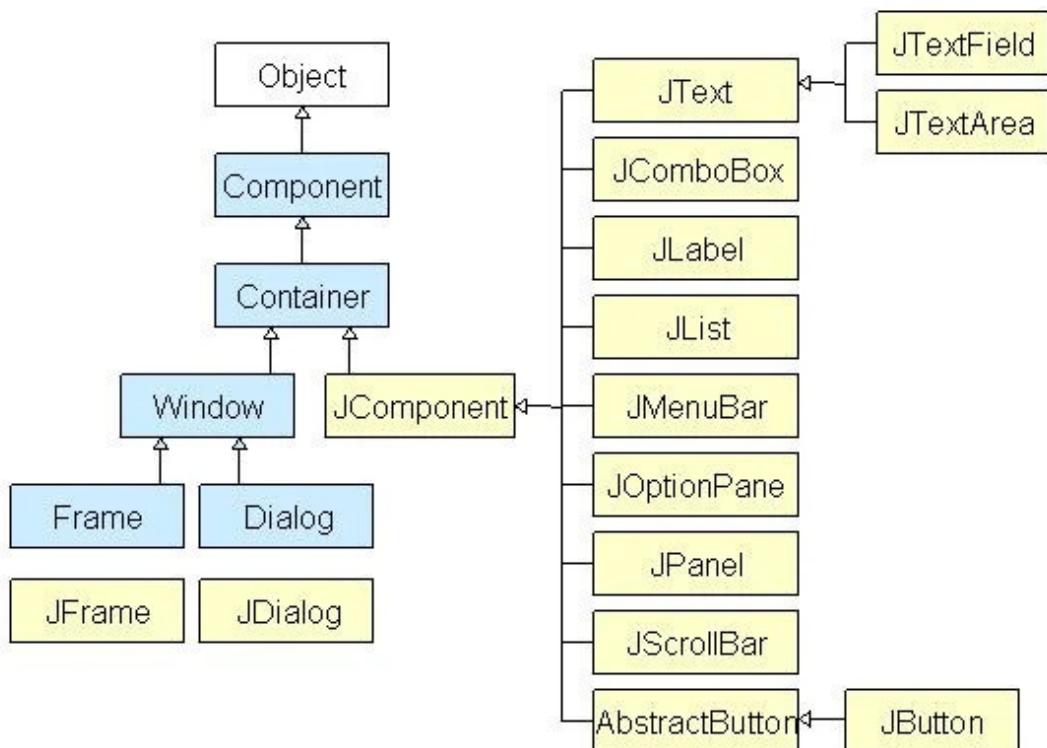


Figure 33. Les classes vues comme une hiérarchie

Plusieurs termes sont utilisés pour désigner la classe qui hérite et celle qui lègue. Les tableaux suivants présentent les termes fréquents pour une relation d'héritage directe et indirecte.

Table 8. Termes utilisés pour qualifier les membres d'une relation d'héritage directe

| Classe qui lègue | Classe qui hérite |
|------------------|-------------------|
| Classe mère | Classe fille |
| Classe parente | Classe enfant |
| Superclasse | Sous-classe |
| Classe de base | Classe dérivée |

Table 9. Termes utilisés pour qualifier les membres d'une relation d'héritage indirecte

| Classe qui lègue | Classe qui hérite |
|------------------|--------------------|
| Classe ancêtre | Classe descendante |
| Superclasse | Sous-classe |
| Classe de base | Classe dérivée |

Après ces éléments théoriques, étudions comment déclarer une relation d'héritage directe en Java.

5.3. La clause `extends` définit une relation d'héritage direct

Nous pouvons définir des relations d'héritage pour des classes que nous avons déclarées dans les chapitres précédents. Par exemple, il existe des cartes moins complexes que des cartes à jouer. Une carte élémentaire (« *base card* ») possède un dos visible ou caché. L'extrait suivant présente sa déclaration.

Définition d'un ancêtre commun à toutes les cartes

```
1 public class BaseCard {  
2  
3     private boolean visible = true;  
4  
5     public void flipIfNotVisible() {  
6         if (!isVisible()) {  
7             flip();  
8         }  
9     }  
10  
11    public boolean isVisible() {  
12        return visible;  
13    }  
14  
15    public void flip() {  
16        visible = !visible;  
17    }  
18 }
```

En Java, ajouter une clause `extends [Classe mère]` à l'en-tête de la classe fille suffit pour déclarer une relation d'héritage. Dans l'extrait suivant, la classe `Card` des chapitres précédents a été renommée `PlayingCard` et hérite de la classe `BaseCard`. Remarquez que les méthodes `flip()` et `isVisible()` ont disparu de `PlayingCard`, de même que le champ `visible`. Ces membres sont dorénavant reçus en héritage.

Toutes les cartes à jouer sont des cartes élémentaires

```
1 public class PlayingCard extends BaseCard {  
2  
3     private Rank rank;  
4     private Suit suit;  
5  
6     public PlayingCard(Rank rank, Suit suit) {  
7         this.rank = Contract.require(  
8             rank,  
9             rank != null,  
10            "Le rang doit être défini."  
11        );  
12        this.suit = Contract.require(  
13            suit,  
14            suit != null,  
15            "L'enseigne doit être défini. "
```

```

16         );
17     }
18
19     public PlayingCard(PlayingCard original) {
20         this(
21             Contract.require(
22                 original,
23                 original != null,
24                 "L'original doit être défini."
25             ).rank,
26             original.suit
27         );
28     }
29
30     public Rank getRank() {
31         return rank;
32     }
33
34     public Suit getSuit() {
35         return suit;
36     }
37
38     public String toString() {
39         if (this.isVisible()) {
40             return "%s%s".formatted(rank, suit);
41         } else {
42             return "?";
43         }
44     }
45
46     public boolean equals(Object other) {
47         if (this == other) {
48             return true;
49         }
50         return other instanceof PlayingCard that && this.equals(that);
51     }
52
53     public boolean equals(PlayingCard other) {
54         boolean notNull = other != null;
55         boolean sameRank = rank == other.rank;
56         boolean sameSuit = suit == other.suit;
57
58         return notNull && sameRank && sameSuit;
59     }
60
61     public int compareTo(PlayingCard that) {
62         Objects.requireNonNull(that);
63         int rankComparison = this.rank.compareTo(that.rank);
64
65         return rankComparison == 0
66             ? this.suit.compareTo(that.suit)
67             : rankComparison;
68     }
69 }
```

En Java, l'héritage est simple : une classe possède au plus une classe mère. En revanche, une classe

peut avoir plusieurs ancêtres. C'est le cas de la classe **PlayingCard** qui a pour ancêtres **BaseCard** et **Object**. La Figure 34 et la Figure 35 présentent les relations à l'aide d'ensembles et d'un arbre.

Héritage de classe simple



Stratégie d'héritage où une classe ne peut avoir qu'une seule classe mère. Par défaut, cette classe mère est la classe **java.lang.Object**.

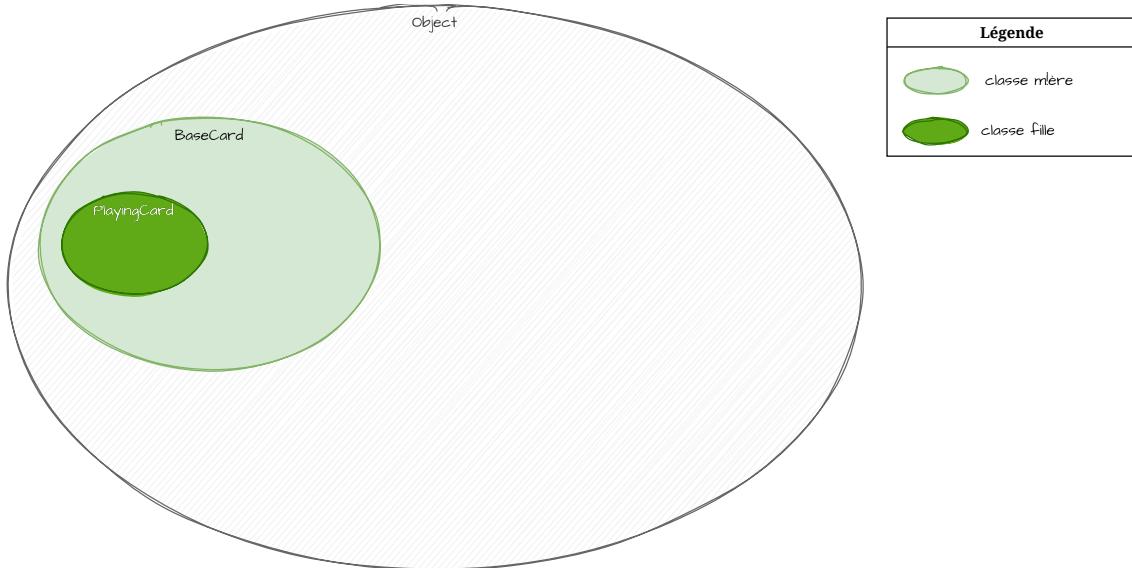


Figure 34. Relation d'héritage vue comme des ensembles

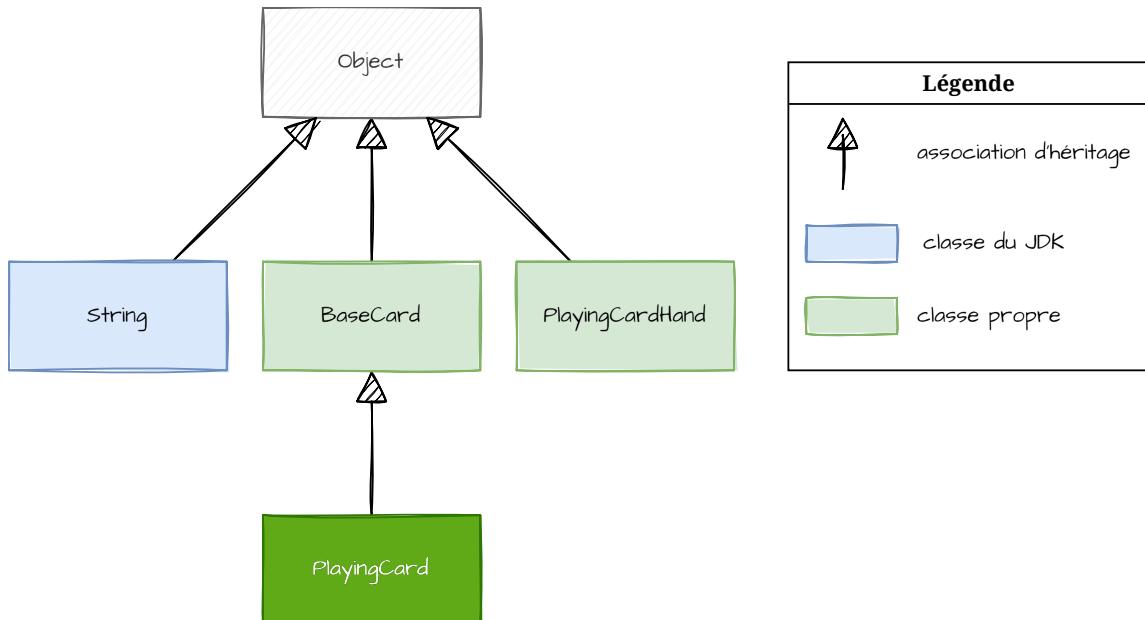


Figure 35. Relation d'héritage vue comme une hiérarchie

L'extrait suivant présente la classe **CardHand**. Son constructeur appelle la méthode **flipIfNotVisible()** d'une carte à jouer. Cette méthode est déclarée dans la classe **BaseCard** dont **PlayingCard** hérite. Le reste du code ne change pas : les méthodes spécifiques à **PlayingCard** sont toujours utilisées.

Appels aux méthodes héritées et redéfinies

```
1 public class CardHand {  
2
```

```

3   private PlayingCard[] cards;
4
5   public CardHand(PlayingCard... cards) {
6       Contract.require(cards != null, "cards doit être défini");
7       this.cards = new PlayingCard[cards.length];
8       int nextIndex = 0;
9       for (PlayingCard card : cards) {
10           this.cards[nextIndex] = card;
11           this.cards[nextIndex].flipIfNotVisible();
12           ++nextIndex;
13       }
14   }
15   //Code inchangé
16 }
```

Puis-je renoncer à une partie de mon héritage ?



Un descendant doit assumer l'héritage de ses ancêtres. En particulier, un objet doit traiter les appels de méthodes dont il hérite. Une classe fille peut ajouter de nouveaux membres ou redéfinir des méthodes héritées. En revanche, elle ne peut pas supprimer des membres hérités.

Nous souhaitons déclarer un constructeur dans `BaseCard` qui initialise une carte élémentaire à l'aide d'un booléen indiquant si la face est visible ou cachée. Quand nous déclarons ce constructeur, l'extrait précédent ne fonctionne plus. Nous obtenons ce message d'erreur :

```

error: constructor BaseCard in class BaseCard cannot be applied to given types;
public PlayingCard(int initialRank, Suit initialSuit)
                           ^
required: boolean
found: no arguments
reason: actual and formal argument lists differ in length
```

La raison principale de l'erreur est l'absence d'un constructeur sans paramètres dans `BaseCard`. Ce constructeur est implicitement appelé par `PlayingCard` pour initialiser la partie héritée. Puisque ce constructeur n'existe plus, nous devons trouver une autre solution pour initialiser la partie héritée. La section suivante étudie cet aspect.

5.4. `super` fait référence à la partie héritée

Toutes les cartes possèdent une face visible ou cachée. Quand vous écrivez `class PlayingCard extends BaseCard`, vous déclarez une relation d'héritage : tout objet `PlayingCard` est un objet `BaseCard` qui possède en plus un rang et une enseigne.

En Java, un objet `PlayingCard` possède une référence à un objet fictif `BaseCard`, donnant accès à une partie de son héritage. Plus généralement, tout objet d'une classe fille possède une référence à un objet fictif de la classe mère. Cette référence spéciale se nomme `super`.

Mot-clé super

Référence à la partie héritée de la classe mère.

Comme `this` (voir [Section 4.2.1](#)), `super` présente deux utilités : initialiser l'état hérité d'un objet et accéder aux membres hérités.

5.4.1. `super` initialise la partie héritée

Tout objet créé doit être dans un état initial valide. Ce principe implique que la partie héritée soit initialisée avant la partie spécifique. En effet, un constructeur pourrait utiliser la partie héritée pour initialiser ses champs spécifiques.

Deux solutions se présentent alors à vous :

- La classe mère contient un constructeur sans paramètres. Dans ce cas, Java appelle implicitement ce dernier avant d'exécuter les instructions d'initialisation spécifiques à la classe fille.
- La classe mère ne contient pas de constructeur sans paramètres. Dans ce cas, vous devez explicitement appeler un constructeur de la classe mère dans les constructeurs de la classe fille avant toute autre instruction. Pour ce faire, `super` s'utilise comme `this`, avec les mêmes contraintes.

Notre classe `PlayingCard` ne compile plus, car nous n'avons mis en œuvre aucune des solutions. Nous choisissons d'appeler le constructeur de la classe `BaseCard` dans le constructeur de `PlayingCard`. L'extrait suivant présente cet appel. Comme pour `this`, l'appel au constructeur de la classe mère doit être la première instruction du constructeur de la classe fille.

Initialisation de la partie héritée grâce à `super`

```
1 public class PlayingCard extends BaseCard {  
2     private Rank rank;  
3     private Suit suit;  
4  
5     public PlayingCard(Rank rank, Suit suit) {  
6         super(true);  
7         this.rank = Objects.requireNonNull(rank, "No rank provided");  
8         this.suit = Objects.requireNonNull(suit, "No suit provided");  
9     }  
10    // Reste du code inchangé  
11 }
```



En Java, l'initialisation de la partie héritée doit être la première instruction d'un constructeur. Sans l'instruction `super(args);`, Java tente d'appeler le constructeur sans paramètres. Si ce constructeur n'existe pas, vous obtiendrez une erreur de compilation.

5.4.2. `super` donne accès aux membres hérités

`super` permet également d'accéder aux membres hérités... selon les modificateurs d'accès associés à ces membres. Dans notre exemple, `PlayingCard` ne peut pas accéder au champ `visible`, car ce dernier a le niveau d'accès `private`. Heureusement, nous pouvons consulter l'état du champ avec `isVisible()`.

Un cas d'utilisation fréquent consiste à appeler la méthode héritée dans sa redéfinition. Si nous redéfinissons `toString()` dans `BaseCard`, nous pouvons appeler cette version dans la redéfinition de `toString()` de la classe `PlayingCard` avec `super`.

Redéfinition de `toString()` dans la classe mère

```
1 public class BaseCard {
2     private boolean visible;
3
4     public BaseCard(boolean visible) {
5         this.visible = visible;
6     }
7
8     public void flip() {
9         visible = !visible;
10    }
11
12    public boolean isVisible() {
13        return visible;
14    }
15
16    public String toString() {
17        if(isVisible()) {
18            return "\u1F0A0";
19        } else {
20            return "\u1F0BF";
21        }
22    }
23 }
```

Appel à la méthode héritée dans sa redéfinition

```
1 public class PlayingCard extends BaseCard {
2     // ...
3     public String toString() {
4         if(isVisible()) {
5             return String.format("%2s%s", rank, suit);
6         } else {
7             return super.toString();
8         }
9     }
10    // Reste du code inchangé
11 }
```

L'utilisation du `super` dans ce contexte illustre comment l'héritage permet de réutiliser du code déjà écrit. Cependant, avant la version 5 de Java, les redéfinitions pouvaient être délicates. En effet, il n'était pas rare d'ajouter de nouvelles méthodes dans une classe en pensant redéfinir une méthode. Toutefois, les méthodes ajoutées n'étaient pas des redéfinitions, mais des surcharges...

5.5. Une surcharge n'est pas une redéfinition

Rappelons qu'une surcharge est une méthode du même nom que d'autres méthodes définies dans la même classe, mais qui se distingue par sa liste de paramètres.

Rappels de la notion de surcharge

```
1 // Surcharge : mêmes noms, signatures différentes
2 int compareTo(Card card) { ... }
3 int compareTo(Rank rank) { ... }
4 int compareTo(Suit suit) { ... }
```

Le bénéfice d'une redéfinition est qu'un même appel de méthode entraîne des exécutions spécifiques à la classe concrète de l'objet recevant l'appel. Redéfinir, c'est donner un nouveau corps à une méthode héritée sans changer son en-tête, en particulier son nom et sa liste de paramètres.



Classe concrète d'un objet

Classe utilisée pour créer l'objet avec l'opérateur `new`.

Ajoutons une méthode publique `BaseCard.compareTo(BaseCard)` qui fixe l'ordre entre deux cartes basiques sur base de leur visibilité. `PlayingCard` en hérite et dispose en outre d'une méthode `PlayingCard.compareTo(PlayingCard)`. Quelle relation lie ces deux méthodes ? Nous pouvons affirmer que `PlayingCard.compareTo(PlayingCard)` surcharge `BaseCard.compareTo(BaseCard)`, car elles diffèrent par leurs listes de paramètres.

Déclaration d'une méthode surchargée par la classe fille

```
1 public class BaseCard {
2     // Reste du code inchangé
3     public int compareTo(BaseCard that) {
4         Objects.requireNonNull(that);
5
6         int thisVisible = this.isVisible() ? 1 : 0;
7         int thatVisible = that.isVisible() ? 1 : 0;
8
9         return thisVisible - thatVisible;
10    }
11 }
```

Dans l'extrait suivant, seuls le premier et le quatrième appels sont des appels à `PlayingCard.compareTo(PlayingCard)`. En effet, l'objet qui reçoit l'appel et l'argument déterminent la méthode appelée.

```
1 public class PlayingCardSample {
2     public static void main(String[] args) {
3         PlayingCard aceSpade = new PlayingCard(Rank.ACE, Suit.SPADE);
4         PlayingCard fourthDiamond = new PlayingCard(Rank.FOUR, Suit.DIAMOND);
5         BaseCard simpleCard = new BaseCard(false);
6
7         System.out.printf("%s compareTo %s ? %d%n",
8             aceSpade, fourthDiamond, aceSpade.compareTo(fourthDiamond));
9     }
10 }
```

```

8         aceSpade, fourthDiamond, aceSpade.compareTo(fourthDiamond));
9     System.out.printf("%s compareTo %s ? %d%n",
10            simpleCard, aceSpade, simpleCard.compareTo(aceSpade));
11     System.out.printf("%s compareTo %s ? %d%n",
12            aceSpade, simpleCard, aceSpade.compareTo(simpleCard));
13 //Appel des méthodes héritées
14     aceSpade.flip();
15
16     System.out.printf("%s compareTo %s ? %d%n",
17            aceSpade, fourthDiamond, aceSpade.compareTo(fourthDiamond));
18     System.out.printf("%s compareTo %s ? %d%n",
19            simpleCard, aceSpade, simpleCard.compareTo(aceSpade));
20     System.out.printf("%s compareTo %s ? %d%n",
21            aceSpade, simpleCard, aceSpade.compareTo(simpleCard));
22 }
23 //==> A♠ compareTo 4♦ ? -3
24 //==> compareTo A♠ ? -1
25 //==> A♠ compareTo ? 1
26 //Après flip()
27 //==> compareTo 4♦ ? -3
28 //==> compareTo ? 0
29 //==> compareTo ? 0
30 }

```

Déclarons à présent une méthode `PlayingCard.compareTo(BaseCard)`. Cette dernière redéfinit `BaseCard.compareTo(BaseCard)` : sa signature correspond à une méthode dont `PlayingCard` hérite. Avec cette déclaration, le troisième et le sixième appels seront des appels à `PlayingCard.compareTo(BaseCard)`.

```

1 public class PlayingCard extends BaseCard {
2     // Reste du code inchangé
3     // Cette méthode surcharge BaseCard.compareTo(BaseCard)
4     public int compareTo(PlayingCard that) {
5         Objects.requireNonNull(that);
6         int rankComparison = this.rank.compareTo(that.rank);
7
8         return rankComparison == 0
9             ? this.suit.compareTo(that.suit) : rankComparison;
10    }
11
12    // Cette méthode redéfinit BaseCard.compareTo(BaseCard)
13    public int compareTo(BaseCard that) {
14        if(that instanceof PlayingCard thatPlayingCard) {
15            return this.compareTo(thatPlayingCard);
16        } else {
17            return super.compareTo(that);
18        }
19    }
20    // Reste du code inchangé
21 }

```

JUnit exploite ce mécanisme de surcharge dans ses assertions. Par exemple, l'assertion `Assertions.assertEquals(Object, Object)` appelle la méthode `equals(Object)` du premier

argument, méthode que vous pouvez redéfinir. En revanche, JUnit ignore toute surcharge du type `equals(ClasseConcrète)`. Dans le même ordre d'idées, nous ne parlerons pas de redéfinition si la méthode que vous cherchez à redéfinir dans la classe fille est `private`. Cette méthode ne fait pas partie des membres accessibles par la classe fille.

Pour parler de redéfinition entre deux méthodes `d'objet X.m(params)` et `Y.m(params)`, il faut que :



- `X` hérite de `Y` ;
- les listes de paramètres soient équivalentes ;
- la classe fille a accès à la méthode de sa mère ;
- le modificateur d'accès de la redéfinition soit au moins aussi ouvert que celui de la méthode dans la classe mère.

Depuis Java 5, l'annotation `@Override` peut précéder une redéfinition de méthode. Grâce à elle, le compilateur vérifie l'existence d'une méthode correspondante dans la classe mère, faute de quoi il signale une erreur. Cette vérification évite de confondre une redéfinition avec une surcharge. Si vous ajoutez cette annotation devant la méthode `PlayingCard.compareTo(PlayingCard)`, vous obtiendrez une erreur de compilation signalant le problème.

Le compilateur détecte une redéfinition erronée

```
error: method does not override or implement a method from a supertype  
@Override
```

Dans le prochain chapitre, nous verrons comment le polymorphisme permet d'utiliser des objets de sous-classes de manière interchangeable, sans connaître leur type exact.

5.6. Les affectations ascendantes sont implicites

Une pile de cartes est une collection de cartes empilées les unes sur les autres. Seule la carte au sommet de la pile est face visible. On peut retirer la carte au sommet de la pile tant que cette dernière n'est pas vide. La nouvelle carte au sommet sera alors retournée.

L'extrait suivant déclare la classe **CardStack**. Concrètement, cette classe est composée d'un tableau de **BaseCard**. Son initialisation ressemble à la classe **CardHand**. Notez que cette classe ne dépend pas de la classe **PlayingCard**.

Déclaration d'une classe composée d'un tableau d'objets de la classe mère

```
1 public class CardsStack {
2     private static final BaseCard[] DEFAULT_CARDS = new BaseCard[0];
3     private BaseCard[] cards;
4     private int topPosition;
5
6     public CardsStack(BaseCard...cards) {
7         this.cards = cards != null ?
8             Arrays.copyOf(cards, cards.length) :
9             DEFAULT_CARDS;
10
11         //Rendre la carte au sommet visible, s'il y a des cartes
12         if(this.cards.length > 0) {
13             topPosition = this.cards.length - 1;
14             if(!this.cards[topPosition].isVisible()) {
15                 this.cards[topPosition].flip();
16             }
17         } else {
18             topPosition = -1;
19         }
20
21         //Cacher les autres cartes
22         for(int i=0; i < this.topPosition; ++i) {
23             if(this.cards[i].isVisible()) {
24                 this.cards[i].flip();
25             }
26         }
27     }
28
29     public BaseCard getTop() {
30         if(topPosition >= 0) {
31             return this.cards[topPosition];
32         } else {
33             return null;
34         }
35     }
36
37     public void take() {
38         if(topPosition >= 0) {
39             --topPosition;
40             if(topPosition >= 0) {
41                 cards[topPosition].flip();
42             }
43         }
44     }
45 }
```

```
44     }
45 }
```

Il paraît cependant naturel de créer des piles de cartes à jouer. Autrement dit, nous devrions pouvoir écrire le code ci-dessous.

Utilisation souhaitée d'une pile de cartes à jouer

```
1 public class CardsStackSample {
2     public static void main(String[] args) {
3         PlayingCard aceOfSpade = new PlayingCard(Rank.ACE, Suit.SPADE);
4         PlayingCard aceOfHeart = new PlayingCard(Rank.ACE, Suit.HEART);
5         PlayingCard aceOfDiamond = new PlayingCard(Rank.ACE, Suit.DIAMOND);
6         PlayingCard aceOfClub = new PlayingCard(Rank.ACE, Suit.CLUB);
7
8         CardsStack acesStack = new CardsStack(aceOfSpade, aceOfHeart, aceOfDiamond,
9             aceOfClub);
10        System.out.printf("top = %s\n", acesStack.getTop());
11        acesStack.take();
12        System.out.printf("top = %s\n", acesStack.getTop());
13    }
14    //==> top = 1♣
15    //==> top = 1◊
16 }
```

Deux options sont envisageables :

1. Créer une surcharge de constructeur acceptant un nombre quelconque de **PlayingCard** ;
2. Espérer une adaptation naturelle.

Bonne nouvelle, la seconde option est celle implémentée par Java. Notre code compile et s'exécute sans avoir à modifier quoi que ce soit. En effet, comme l'affectation d'un **int** à une variable de type **long** ne demande pas d'interventions particulières, l'affectation d'un objet de classe **X** à une référence de classe **Y** se fait naturellement... si **X** hérite de **Y**.

Affectation ascendante implicite

Affectation d'un objet de classe **X** à une référence de type **Y**, où **X** hérite de **Y**. Java gère ce type d'affectation implicitement.

L'affectation est qualifiée d'ascendante en référence aux conventions utilisées pour représenter les hiérarchies d'héritage. Selon ces conventions, les ancêtres sont placés au-dessus des descendants.

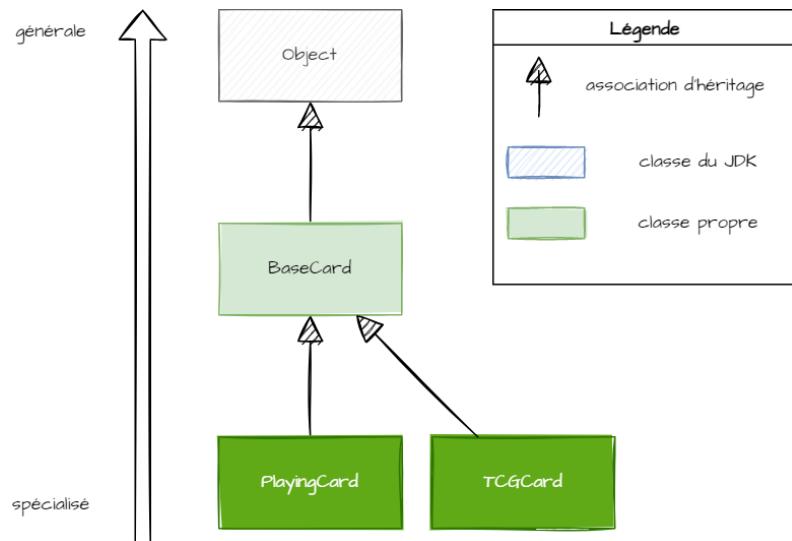


Figure 36. Représentation classique d'une relation d'héritage

Si vous observez la [Figure 36](#), les objets des classes `PlayingCard` et `TCGCard` ^[1] héritent de `BaseCard`. Ils peuvent entrer dans la composition de piles de cartes. Nous pouvons même créer une pile de cartes mélangeant des cartes à jouer et des cartes à collectionner !

L'affectation ascendante produit d'autres effets, notamment sur les tableaux. Vous pourrez affecter un tableau de type `X[]` à une référence de type `Y[]` et accéder à ses éléments. L'extrait suivant déclare une référence à un tableau de `BaseCard` initialisée avec un tableau de `PlayingCard`.

Utilisation souhaitée d'une pile de cartes à jouer

```

1 public class CardsStackSample {
2     public static void main(String[] args) {
3         BaseCard[] aces = new PlayingCard[] {
4             new PlayingCard(Rank.ACE, Suit.SPADE),
5             new PlayingCard(Rank.ACE, Suit.HEART),
6             new PlayingCard(Rank.ACE, Suit.DIAMOND),
7             new PlayingCard(Rank.ACE, Suit.CLUB)
8         }
9
10        CardsStack acesStack = new CardsStack(aces);
11
12        System.out.printf("top = %s\n", acesStack.getTop());
13        acesStack.take();
14        System.out.printf("top = %s\n", acesStack.getTop());
15    }
16    //==> top = 1♣
17    //==> top = 1♦
18 }
```

En Java, un tableau d'une classe fille est automatiquement un tableau de sa classe mère. Autrement dit, le type `F[]` hérite de `M[]` quand la classe `F` hérite de `M`. La relation d'héritage des tableaux **varie avec** celle de leurs éléments : c'est la **covariance des tableaux**.

Covariance des tableaux

Propriété d'un tableau de `X` à être également un tableau de `Y` quand `X` hérite de `Y`.

5.7. Le modificateur `final` empêche l'héritage

Le sens général de ce modificateur est « empêcher la modification ». Vous pouvez utiliser `final` dans l'en-tête d'une classe, d'une méthode ou dans la déclaration d'une variable. La signification précise du modificateur dépend de l'élément qualifié. Son utilisation permet au compilateur d'appliquer plusieurs optimisations.

5.7.1. final pour les variables

Appliqué à une variable, `final` signifie que cette dernière conserve sa valeur initiale pendant toute sa vie : le compilateur Java interdira toute affectation ultérieure. Précisons que Java impose qu'une telle variable soit initialisée avant sa première utilisation. La séquence suivante est donc légale.

Affectation légale d'une variable `final`

```
1 public double area() {  
2     final double PI;  
3     PI = 3.14159;  
4     return radius*radius*PI;  
5 }
```

Le type d'une variable détermine son domaine de valeur. Le type affecte également le comportement du compilateur. En effet, lorsque la variable `final` est de type primitif, Java la voit comme une constante. Lorsqu'il peut déduire sa valeur au moment de la compilation, Java remplace la constante par sa valeur dans le reste du programme. Ce comportement accélère imperceptiblement l'exécution du programme.

Différences entre constantes calculables à la compilation ou à l'exécution

```
1 public class FinalData {  
2     // Constantes calculables dès la compilation  
3     final int perfectNumber = 42;  
4     public static final float GOLDEN_NUMBER = 1.618033f;  
5  
6     // Constantes calculables pendant l'exécution  
7     final int i4 = (int)(Math.random()*20);  
8     static final int i5 = (int)(Math.random()*20);  
9 }
```

Le cas d'une référence finale est plus subtil. Une telle référence mémorise l'adresse d'un objet et cette adresse ne changera plus. Cependant, cet objet peut muter, c'est-à-dire changer d'état.

Une référence finale à un objet qui change d'état

```
1 final BaseCard myCard = new PlayingCard(1, '\u2661');  
2  
3 myCard.flip(); //myCard change d'état  
4 // du code  
5 myCard.flip(); //myCard change d'état
```



Une référence `final` désigne toujours le même objet. Cependant, l'état de cet objet peut évoluer.

Le niveau de déclaration de la variable influence les possibilités d'affectation :

- Par exemple, vous pouvez affecter la valeur d'un champ `final` soit au moment de sa déclaration, soit dans les constructeurs. Cette règle est cependant à nuancer lorsque le champ est `static`. Dans ce cas, il ne peut recevoir sa valeur qu'au moment de sa déclaration.
- En revanche, une méthode ne peut pas affecter un paramètre `final`. En effet, cette valeur a été fixée par l'appelant au moment de l'appel.

5.7.2. final pour les méthodes

Appliqué à une méthode, `final` interdit sa redéfinition par une classe qui en hérite. Autrement dit, vous ne pouvez plus changer son comportement dans les classes descendantes. En revanche, vous pouvez toujours l'appeler.

Rendre une méthode `final` autorise des optimisations. En effet, dans certains cas, le compilateur peut alors recopier les instructions d'une méthode `final` plutôt que de manipuler la pile, ce qui rend l'appel plus rapide. Une telle approche, efficace pour de courtes méthodes, montre rapidement ses limites si la méthode est longue. Le compilateur peut optimiser davantage le code grâce à ce modificateur, mais ce sujet sort du cadre de ce cours.

Dans l'extrait ci-dessous, nous rendons les méthodes `flip()` et `isVisible()` finales. Autrement dit, les classes peuvent en hériter, mais elles ne peuvent plus les redéfinir. Ces méthodes sont de bons candidats à une optimisation de la pile. En revanche, nous avons laissé `toString()` ouverte à la redéfinition, redéfinition que nous faisons dans la classe `PlayingCard`.

Déclaration de méthodes finales

```
1 public class BaseCard {  
2     private boolean visible;  
3  
4     public BaseCard(boolean visible) {  
5         this.visible = visible;  
6     }  
7  
8     public final void flip() {  
9         visible = !visible;  
10    }  
11  
12    public final boolean isVisible() {  
13        return visible;  
14    }  
15  
16    public String toString() {  
17        if(isVisible()) {  
18            return "\uD83C\uDCBF";  
19        } else {  
20            return "\uD83C\uDCA0";  
21        }  
22    }  
23}
```

```

22     }
23
24     public int compareTo(BaseCard that) {
25         Objects.requireNonNull(that);
26
27         int thisVisible = this.isVisible() ? 1 : 0;
28         int thatVisible = that.isVisible() ? 1 : 0;
29
30         return thisVisible - thatVisible;
31     }
32 }
```

5.7.3. final pour les classes

Appliqué à une classe, `final` empêche d'en hériter. Puisqu'il est impossible d'en hériter, Java considère que toutes les méthodes d'une classe `final` sont `final` ce qui permet d'appliquer davantage d'optimisations. Dans l'extrait suivant, nous rendons la classe `PlayingCard` finale. Ce faisant, nous considérons qu'il n'existe pas de sous-ensemble de cartes à jouer avec des comportements ou des champs différents.

Déclaration d'une classe finale

```

1 public final class PlayingCard extends BaseCard {
2     private Rank rank;
3     private Suit suit;
4
5     public PlayingCard(Rank rank, Suit suit) {
6         super(true);
7         this.rank = Objects.requireNonNull(rank, "No rank provided");
8         this.suit = Objects.requireNonNull(suit, "No suit provided");
9     }
10
11    public boolean isAce() {
12        return this.rank == Rank.ACE;
13    }
14
15    public boolean isFigure() {
16        return rank.isFigure();
17    }
18
19    public int getRankValue() {
20        return rank.getValue();
21    }
22
23    // Cette méthode surcharge BaseCard.compareTo(BaseCard)
24    public int compareTo(PlayingCard that) {
25        Objects.requireNonNull(that);
26        int rankComparison = this.rank.compareTo(that.rank);
27
28        return rankComparison == 0
29            ? this.suit.compareTo(that.suit) : rankComparison;
30    }
31
32    // Cette méthode redéfinit BaseCard.compareTo(BaseCard)
33    public int compareTo(BaseCard that) {
```

```

34         if(that instanceof PlayingCard thatPlayingCard) {
35             return this.compareTo(thatPlayingCard);
36         } else {
37             return super.compareTo(that);
38         }
39     }
40
41     public String toString() {
42         if(isVisible()) {
43             return String.format("%2s%s", rank, suit);
44         } else {
45             return super.toString();
46         }
47     }
48
49     public boolean equals(Object o) {
50         if(this == o) {
51             return true;
52         }
53
54         return o instanceof PlayingCard that &&
55             this.rank.equals(that.rank) &&
56             this.suit.equals(that.suit);
57     }
58 }
```

Attention, rendre une classe finale ne signifie pas que ses objets soient immuables. En effet, un objet est immuable si et seulement si son état n'évolue pas, autrement dit si les valeurs de ses champs ne changent pas. Dans notre exemple, nous pouvons toujours retourner une carte à jouer.

5.8. La composition peut exploiter l'héritage

Les relations de composition et d'héritage prennent tout leur sens quand vous les combinez. Bien utilisées, elle produisent du code facile à changer.

Reprendons nos cartes à jouer. L'algorithme de comparaison définit un ordre par défaut pour les cartes : elles sont ordonnées par enseigne et, pour chaque enseigne, par rang. Cependant, cet ordre peut varier d'un jeu à l'autre. Par exemple, à la bataille, les As sont plus grands que les rois et les enseignes n'ont aucune importance. Au baccara, les rois, les dames, les valets et les dix valent 0, l'as vaut 1 et les autres cartes valent la valeur de leur rang. Si on souhaite que la méthode `compareTo(PlayingCard)` respectent ces ordres, nous devons la redéfinir. Nous étudions d'abord une solution naïve avant de présenter une solution plus flexible.

Nous pourrions étendre `PlayingCard` avec une nouvelle classe. Cette approche peut convenir si l'algorithme de comparaison est le seul critère utilisé pour étendre la hiérarchie. Toutefois, les choses dégénèrent dès qu'une seconde raison d'extension apparaît, par exemple, si la mise en forme d'une carte change. La Figure 37 montre que le nombre de classes filles augmente exponentiellement avec le nombre de critères d'extension.

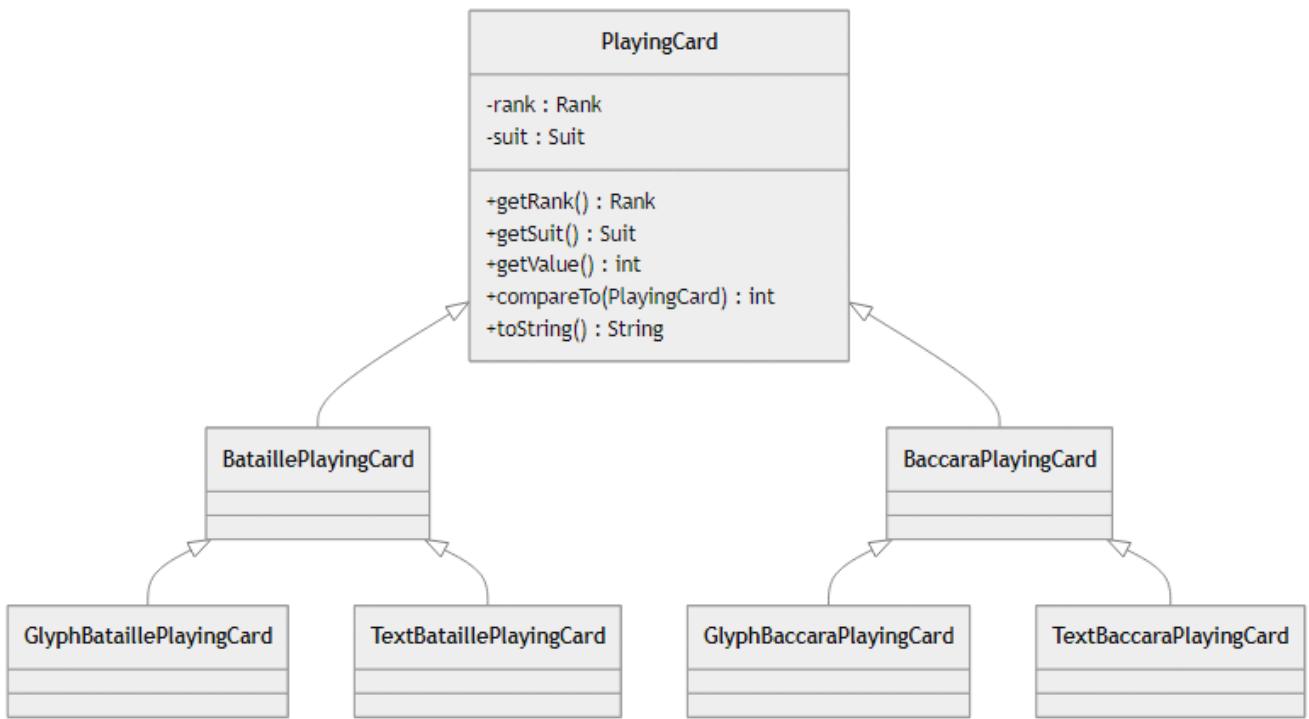


Figure 37. Explosion combinatoire d'une hiérarchie d'héritage

L'approche recommandée pour ce cas (Figure 38) consiste à maintenir deux hiérarchies indépendantes : la première pour la mise en forme, la seconde pour les comparaisons. **PlayingCard** déclarera deux références à des objets des classes racines de ces hiérarchies, ainsi que des mutateurs pour changer de comportements. Cette solution combine les relations de composition et d'héritage.

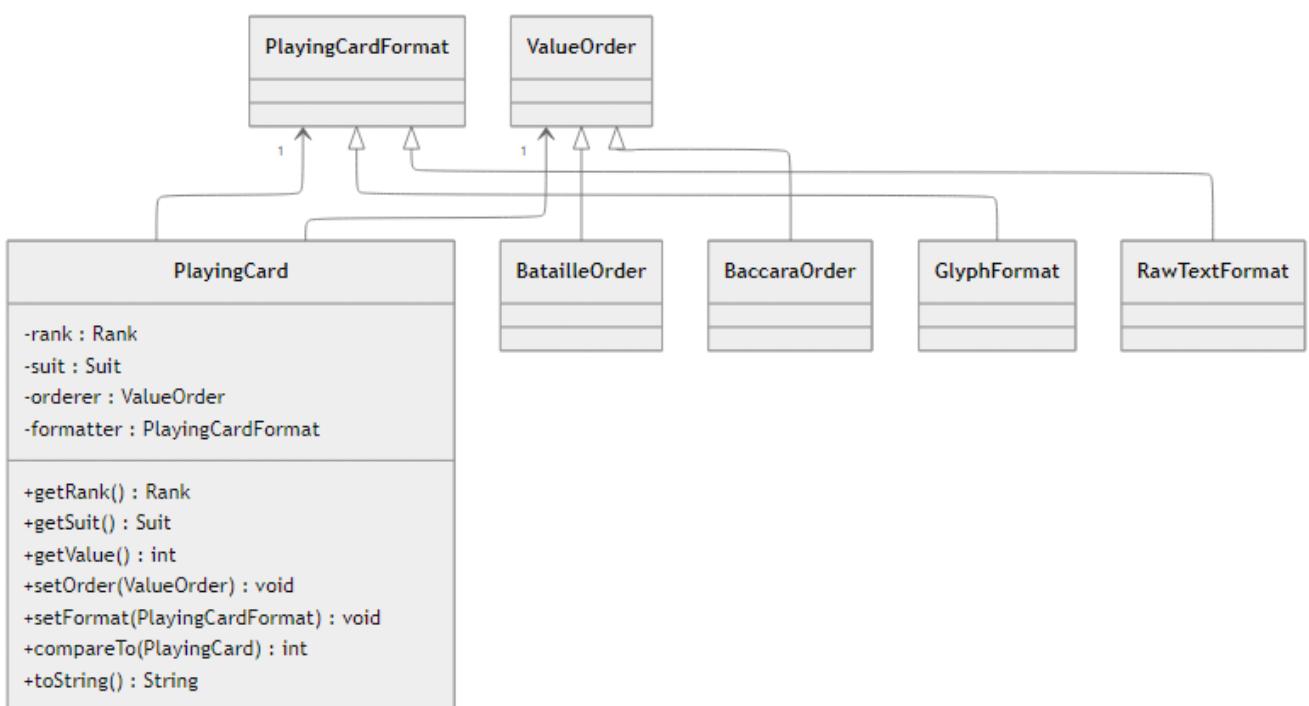


Figure 38. Approche recommandée

Cette approche présente l'avantage principal que les deux préoccupations, mettre en forme et comparer des cartes à jouer, sont séparées. Ainsi, implémenter un nouvel ordre n'exige aucune adaptation de la hiérarchie pour la mise en forme et inversement. Autre avantage important, nous

pouvons également changer d'algorithme de mise en forme ou de comparaison pendant l'exécution du programme.

La composition est votre outil principal. Contrairement à une relation d'héritage, vous pouvez changer les composants d'un objet pendant l'exécution.

L'héritage s'utilise plus rarement. Contrairement à la composition, vous ne pouvez pas changer la classe mère d'un objet pendant l'exécution. Les relations d'héritage mal définies peuvent également causer des problèmes d'exécution difficile à trouver et à résoudre. Le chapitre suivant en étudiera plusieurs. En conclusion, mal utilisée, l'héritage rend le code difficile à changer.



Favoriser la composition à l'héritage

La POO encourage la composition par rapport à l'héritage.

5.9. En résumé

La composition et l'héritage constituent deux moyens de réutiliser le code déjà écrit. Elles représentent cependant deux relations différentes.

La composition définit le comportement d'une classe en combinant des objets d'autres classes. Elle est associée à la phrase type « *Tout [nom de la classe composite] possède un [nom de la classe composante] qui tient le rôle de [nom du champ]* ». C'est la relation à privilégier.

L'héritage dote une classe des comportements définis dans d'autres classes. La classe fille peut enrichir ces comportements en ajoutant ses propres méthodes. Elle peut également redéfinir des méthodes reçues en héritage. Propriété intéressante, vous pouvez affecter à une référence de type X tout objet de type Y si Y hérite de X : c'est l'affectation ascendante implicite. L'héritage correspond à la phrase type « *Toute [classe fille] est une sorte de [classe mère] qui [comportement spécifique]* ».

Votre responsabilité de programmeur vous oblige à réfléchir aux classes héritables. Dans la majorité des cas, vous interdirez à une classe d'être l'ancêtre d'autres classes : elle sera alors qualifiée de **final**. Parfois, vous empêcherez la redéfinition d'une méthode qui sera, elle aussi, qualifiée de **final**. N'oubliez pas que l'héritage introduit un lien fort entre deux types, voire plus. Si vous cassez ou changez le fonctionnement de la classe ancêtre, vous risquez ensuite de devoir adapter tous les descendants.

[1] *Trading Card Game Card*, ou TGC. Ce sont des cartes à collectionner.

Chapitre 6. Le polymorphisme

Ce chapitre étudie les types de polymorphismes. Il s'attarde sur celui qui est appliqué pendant l'exécution du programme.

Au cours des chapitres précédents, nous avons observé des comportements plutôt étonnants.

Par exemple, redéfinir la méthode `toString()` dans une classe modifie le texte imprimé par la méthode `printf(String, Object...args)`. Cette dernière appelle la version redéfinie de `toString()`, bien qu'elle voie nos objets à travers des références à un `Object`. Ce comportement vaut aussi pour la méthode `equals(Object)`, appelée par les assertions de JUnit. Nous pouvons affirmer que `toString()` et `equals(Object)` sont polymorphes, car le même appel de méthode engendre des exécutions dépendantes de la classe concrète de l'objet qui reçoit l'appel.

En programmation, le polymorphisme fait référence à un ou plusieurs mécanismes proposés par un langage vous permettant d'adapter du code sans avoir à le modifier. Appliquée à la POO, le polymorphisme consiste à adapter le comportement d'objets ou de types sans les modifier.

Polymorphisme



Littéralement, caractère d'une entité qui peut prendre plusieurs formes. En programmation, le polymorphisme fait référence à la capacité d'une méthode ou d'un opérateur à s'adapter aux types de ses arguments ou de l'objet qui reçoit l'appel.

Ce chapitre étudie plusieurs types de polymorphisme. Il distingue le polymorphisme qui prend place pendant la compilation de celui qui survient durant l'exécution du programme. Commençons par les sortes de polymorphisme appliquées à la compilation.

6.1. Il existe plusieurs sortes de polymorphisme

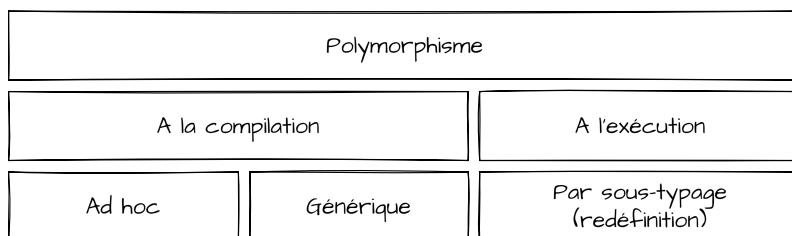


Figure 39. Différentes variétés de polymorphismes

On peut classer les types de polymorphismes en fonction du moment où ils interviennent :

- Le polymorphisme à la compilation est mis en œuvre par le compilateur. En Java, il consiste notamment à choisir la bonne surcharge de méthode ou d'opérateur selon les types des opérandes : c'est le polymorphisme ad hoc. Une seconde sorte de polymorphisme à la compilation est le polymorphisme générique où un type ou une méthode est paramétrable par des types.
- Le polymorphisme à l'exécution est mis en œuvre par la JVM. Il consiste à choisir la méthode à

exécuter selon la classe concrète de l'objet recevant l'appel. Nous parlerons de polymorphisme par sous-typage.

Les sections suivantes étudient chaque sorte de polymorphismes.

6.2. Le polymorphisme ad hoc choisit la bonne surcharge

Le polymorphisme ad hoc détermine la méthode ou l'opérateur à exécuter selon le type des arguments ou des opérandes. Java est un langage fortement typé où les types des arguments sont connus par le compilateur : il est responsable d'appliquer le polymorphisme ad hoc^[1].

Polymorphisme ad hoc

Application de la surcharge d'une méthode ou d'un opérateur selon les types des arguments. En Java, le compilateur applique la surcharge adéquate.

L'extrait suivant déclare deux versions d'une méthode `printCentered(param)`. Comme l'exemple le montre, le compilateur identifie la version à appeler sur base de la liste d'arguments. Si une liste de paramètres compatible est trouvée, l'appel est exécuté. En revanche, si le compilateur ne trouve aucune correspondance, il lance un message d'erreur.

Utilisation du polymorphisme ad hoc

```
1 import java.time.LocalDate;
2
3 class PrettyPrinter {
4
5     static void printCentered(int val, int space) {
6         String toString = Integer.toString(val);
7         var startPos = (space - toString.length()) / 2;
8
9         System.out.print(" ".repeat(startPos));
10        System.out.print(toString);
11        System.out.println(" ".repeat(startPos));
12    }
13
14    static void printCentered(String s, int space) {
15        var startPos = (space - s.length()) / 2;
16
17        System.out.print(" ".repeat(startPos));
18        System.out.print(s);
19        System.out.println(" ".repeat(startPos));
20    }
21
22    public static void main(String[] args) {
23        PrettyPrinter.printCentered(42, 12);
24        PrettyPrinter.printCentered("BONJOUR", 12);
25
26        // PrettyPrinter.printCentered(LocalDate.of(1970, 1, 1), 12); // ✗ pas de surcharge
27    }
28 }
```

Le programme suivant déclare trois variables de type `int` et calcule le maximum entre les deux premières. Le compilateur appelle la surcharge `printMax` adéquate selon les arguments. Le premier appel transmet trois entiers tandis que le second appel convertit le premier argument en flottant. Vous venez d'expérimenter un cas de polymorphisme ad hoc. Le compilateur choisit d'appeler `printMax(float, float, float)`, car c'est la seule version compatible avec les arguments.

Une application exploitant le polymorphisme ad hoc

```
1 public class AdHocPolymorphism {
2
3     public static void main(String[] args) {
4         int a = 6,
5             b = 10,
6             max = 0;
7         max = Math.max(a, b);
8
9         printMax(a, b, max);
10        printMax((float) a, b, max);
11    }
12
13    private static void printMax(int a, int b, int max) {
14        System.out.printf("Max(%d, %d) == %d\n", a, b, max);
15    }
16
17    private static void printMax(float a, float b, float max) {
18        System.out.printf("Max(%3.02f, %3.02f) == %3.02f\n", a, b, max);
19    }
20 }
```

Notez que transmettre des arguments de type `double` provoque une erreur de compilation : aucune signature n'est compatible avec ce type. En effet, Java n'autorise les affectations implicites que d'un type spécifique vers un type plus général. Si nous voulons transmettre un double, nous devrons le convertir explicitement.

Erreur obtenue si max est de type double

```
error: no suitable method found for printMax(int,int,double)
      printMax(a, b, max);
      ^
method AdHocPolymorphism.printMax(int,int,int) is not applicable
  (argument mismatch; possible lossy conversion from double to int)
method AdHocPolymorphism.printMax(float,float,float) is not applicable
  (argument mismatch; possible lossy conversion from double to float)
```

6.2.1. Le mot-clé `var` déduit le type d'une variable locale

Depuis Java 10, nous pouvons laisser le compilateur inférer le type d'une variable locale sur base de sa valeur initiale : c'est l'inférence de type. Pour l'utiliser, remplacez le type d'une variable par `var`. Attention, vous devez initialiser la variable dès sa déclaration. Avec l'inférence de type, le compilateur décide de la surcharge à appliquer après avoir déduit le type des variables.

Une application exploitant le polymorphisme ad hoc et l'inférence de type

```
1 public class AdHocPolymorphism {
2     public static void main(String[] args) {
3         var a=6;
4         var b= 10;
5         var max=10.0f;
6
7         max = Math.max(a,b);
8
9         printMax(a, b, max);
10    }
11    //==> Max(6,00, 10,00) == 10,00
12    private static void printMax(int a, int b, int max) {
13 }
```

L'inférence de type possède quelques limitations. Il ne fonctionne qu'avec les variables locales à une méthode : les champs et les paramètres ne peuvent pas en bénéficier. En outre, le type déduit correspond à celui de l'expression d'initialisation. Quand cette dernière est une expression de création d'objet, son type est la classe concrète de cet objet. En conséquence, vous serez limité à affecter des objets de cette classe ou de ses descendants.

L'inférence de type est un mécanisme intéressant quand la portée de la variable est courte. N'abusez cependant pas de ce mécanisme, sans quoi votre code perdra en expressivité.

6.2.2. Les classes sont substituables avant la compilation

Une autre forme de polymorphisme ad hoc consiste à remplacer une classe par une autre avant la compilation. Cette approche est intéressante quand on dispose de plusieurs stratégies pour résoudre un problème et que la nature du problème est connue avant de compiler le code. En Java, cette technique utilise les paquetages.

Reprenons le problème discuté à la [Section 5.8](#). Nous voulons adapter l'ordre des cartes à jouer selon un jeu sans avoir à changer le code appelant la méthode. Pour ce faire, nous définissons chaque méthode dans son propre paquetage comme le montre l'arbre suivant.

Chaque algorithme de tri est défini dans son propre paquetage

```
► ch06
  └► bataille
    |   └► ValueOrder.java (nom complet : ch06.bataille.ValueOrder)
  └► baccara
    |   └► ValueOrder.java (nom complet : ch06.baccara.ValueOrder)
  └► standard
    └► ValueOrder.java (nom complet : ch06.standard.ValueOrder)
```

Les classes portent le même nom, mais elles diffèrent par leurs noms complets. Ce nom complet est unique pour chaque classe et est utilisé pour les instructions d'import. Si ces classes exposent les mêmes en-têtes de méthode, passer d'un ordre à l'autre revient à changer l'instruction d'import : le reste du code client ne change pas.

```

1 package ch06;
2
3 // import ch06.baccara.ValueOrder;
4 // import ch06.bataille.ValueOrder;
5 import ch06.natural.ValueOrder;
6 import utils.Contract;
7
8 public final class PlayingCard extends BaseCard {
9     public int getValue() {
10         return ValueOrder.getValue(this);
11     }
12
13     public int compareTo(PlayingCard that) {
14         return ValueOrder.compare(this, that);
15     }
16     // Code inchangé
17 }
```

La signature commune des méthodes est la clé de la substitution. Comme les classes portent le même nom, les appels ne doivent pas être adaptés. Quand la signature est commune et que les implémentations respectent le même contrat, les appelants ne peuvent pas deviner la stratégie réellement appliquée.

Cet exemple de substitution avant la compilation convient si vous devez choisir une seule stratégie parmi plusieurs et si ce choix ne change plus une fois le programme compilé. Elle ne convient pas si vous devez changer votre stratégie d'une exécution à l'autre, voire pendant la même exécution.

Pour ces cas d'utilisation, le polymorphisme par sous-typage convient davantage. Avant de l'aborder, étudions une dernière forme de polymorphisme à la compilation : le polymorphisme générique.

6.3. Le polymorphisme générique paramètre un type avec d'autres types



Cette section est facultative.

Java propose plusieurs sortes de collections pour grouper des valeurs de même type^[2]. Vous les trouverez dans le paquetage `java.util`. Contrairement aux tableaux, ces collections sont dynamiques : vous pouvez y ajouter et supprimer des éléments et, ce faisant, modifier leurs tailles.

Parmi les collections, une liste chaînée (linked list) mémorise ses éléments comme les maillons d'une chaîne. Chaque maillon possède une référence au maillon suivant. L'extrait suivant déclare deux listes chaînées : la première mémorise des rangs et la seconde des enseignes.

Utilisation de listes chainées

```

1 import java.util.LinkedList;
2
```

```

3 public class LinkedListSample {
4     public static void main(String[] args) {
5         LinkedList<Rank> figures = new LinkedList<>();
6         figures.add(Rank.JACK);
7         figures.add(Rank.QUEEN);
8         figures.add(Rank.KING);
9
10        Rank first = figures.getFirst();
11        Rank last = figures.getLast();
12
13        System.out.printf("figures[0] = %s\n", first);
14        System.out.printf("figures[-1] = %s\n", last);
15
16        LinkedList<Suit> blacks = new LinkedList<>();
17        blacks.add(Suit.SPADE);
18        blacks.add(Suit.CLUB);
19        blacks.add(Suit.DIAMOND);
20
21        System.out.printf("blacks[1] = %s\n", blacks.get(1));
22
23        blacks.remove(Suit.DIAMOND);
24
25        // blacks.add(Rank.ACE); // ✗ refusé par le compilateur
26    }
27 }
```

Comme vous pouvez le remarquer, les déclarations des variables correspondantes sont paramétrées par un type placé entre «<>». Nous parlerons **d'argument de type**. Il indique au compilateur le type des éléments de la liste, comme le type des éléments dans un tableau. Le compilateur affectera cet argument de type à **un paramètre de type**. Grâce à lui, vous pouvez créer des listes chaînées de n'importe quel type d'éléments tout en garantissant la sécurité des types à la compilation.

Les génériques

Mécanisme paramétrant une déclaration de classe ou de méthode avec des types. La forme de la classe ou de la méthode dépendra des arguments de type.

Utilisons les génériques pour définir la notion de valeur datée. Une valeur datée est un type dont les objets mémorisent l'instant de leur création. Ce concept peut être utile pour représenter des séries temporelles, ou pour la mise en cache. Elle s'implémente bien avec les génériques.

Déclaration d'une classe générique

```

1 /**
2 * Une valeur immuable associée à un timestamp (moment de sa création).
3 * @param <T> Le type de la valeur encapsulée.
4 */
5 public final class TimestampedValue<T> {
6     private final T value;           // La valeur stockée
7     private final Instant timestamp; // Le moment où la valeur a été encapsulée
8
9
10    public TimestampedValue(T value) {
```

```

11         this(value, Instant.now());
12     }
13
14     public TimestampedValue(T value, Instant timestamp) {
15         this.value = Objects.requireNonNull(value);
16         this.timestamp = Objects.requireNonNull(timestamp);
17     }
18
19     public T getValue() {
20         return value;
21     }
22
23     public Instant getTimestamp() {
24         return timestamp;
25     }
26
27     public int compareInstant(TimestampedValue<T> other) {
28         return this.timestamp.compareTo(other.timestamp);
29     }
30
31     @Override
32     public String toString() {
33         return String.format("%s at %s", value, timestamp);
34     }
35 }
```

Au moment d'utiliser la classe, les paramètres de type `T` sont remplacés par des types spécifiques. L'extrait ci-dessous utilise des valeurs datées avec différents arguments de type. Il montre que la classe `TimestampedValue<T>` est polymorphe, car elle s'adapte naturellement à son argument de type.

Utilisation d'une classe générique

```

1 @Test
2 void should_equals_timestamps() {
3     TimestampedValue<Float> temperature1 = new TimestampedValue(14.2f, Instant.MAX);
4     TimestampedValue<Float> temperature2 = new TimestampedValue(21.1f, Instant.MAX);
5
6     assertEquals(0, temperature1.compareInstant(temperature2));
7     assertEquals(0, temperature2.compareInstant(temperature1));
8 }
9
10 @Test
11 void should_compare_timestamps() {
12     TimestampedValue<String> previous = new TimestampedValue("test", Instant.EPOCH);
13     TimestampedValue<String> now = new TimestampedValue("test");
14
15     assertTrue(previous.compareInstant(now) < 0);
16     assertTrue(now.compareInstant(previous) > 0);
17 }
```

Le premier test utilise des `TimestampedValue<Float>` plutôt que des `TimestampedValue<float>`: `Float` est un type distinct du type `float`. Au moment d'écrire ces lignes, Java interdit d'utiliser un type comme `float` ou `int`, comme argument de type. Plus généralement, Java impose une

condition sur les arguments de type : ceux-ci doivent correspondre à des types référence, dont les valeurs se manipulent avec des références, comme les classes et les enums. Par opposition, les types `int` et `float` sont des types valeur, dont les valeurs sont directement mémorisées dans leurs variables.

Java interdit d'utiliser un type valeur comme argument de type. Cependant, le langage propose une classe enveloppe pour chaque type valeur. Son nom correspond à celui du type valeur avec la première lettre en majuscule. Seules exceptions à cette règle, la classe enveloppe de `int` se nomme `Integer` et celle de `char` se nomme `Character`.



Java convertit implicitement une valeur primitive vers un objet de la classe enveloppe correspondante, et inversement. Nous parlerons d'emballage (*boxing*) et de déballage (*unboxing*). L'emballage et le déballage coutent en mémoire et en temps d'exécution, coute à considérer si ces ressources importent.

Au chapitre précédent, nous avons programmé une classe `CardsStack` qui décrit une pile de cartes. Cette classe mémorisait les cartes avec des références à la classe `BaseCard`. Grâce au mécanisme d'affectation ascendante, la classe `CardsStack` peut mémoriser des instances de la classe `PlayingCard`, car cette dernière hérite de la classe `BaseCard`.

La version suivante de `CardStack` est devenue générique : elle accepte comme argument de type tout type qui hérite de `BaseCard`. Sans cette contrainte, le compilateur ne peut pas garantir la présence des méthodes `flipIfVisible()` et `flipIfNotVisible()`.

Définition d'une pile de cartes générique

```
1 public class CardsStack<T extends BaseCard> {
2
3     private LinkedList<T> cards;
4
5     public CardsStack(T... cards) {
6         Contract.require(cards != null, "Paramètre cards indéfini");
7         this.cards = new LinkedList<>();
8
9         for (int i = 0; i < cards.length; ++i) {
10             Contract.check(cards[i] != null, "cards[i] == null");
11             cards[i].flipIfNotVisible();
12             this.cards.add(cards[i]);
13         }
14
15         if (this.cards.size() > 0) {
16             var last = this.cards.getLast();
17             last.flipIfVisible();
18         }
19     }
20
21     public T getTop() {
22         Contract.check(
23             cards,
24             !cards.isEmpty(),
25             "État incohérent : cette pile est vide"
26         );
27     }
28 }
```

```

27         return this.cards.getLast();
28     }
29
30     public T take() {
31         Contract.check(
32             cards,
33             !cards.isEmpty(),
34             "État incohérent : cette pile est vide"
35         );
36
37         T last = this.cards.removeLast();
38         if (!this.cards.isEmpty()) {
39             this.cards.getLast().flip();
40         }
41         return last;
42     }
43
44     public void push(T card) {
45         Contract.require(card, card != null, "Paramètre card indéfini");
46         this.cards.addLast(card);
47     }
48
49     public boolean hasCards() {
50         return this.cards.size() > 0;
51     }
52 }
```

Le programme suivant montre que, dorénavant, une pile peut être typée avec des classes qui héritent de **BaseCard**. Les méthodes s'adaptent naturellement à l'argument de type utilisé.

Utilisation d'une pile de cartes générique

```

1 public class CardsStackSample {
2
3     public static void main(String[] args) {
4         var aceOfSpade = new PlayingCard(Rank.ACE, Suit.SPADE);
5         var twoOfHeart = new PlayingCard(Rank.TWO, Suit.HEART);
6         var threeOfDiamond = new PlayingCard(Rank.THREE, Suit.DIAMOND);
7         var fourOfClub = new PlayingCard(Rank.FOUR, Suit.CLUB);
8
9         var aStack = new CardsStack<PlayingCard>(
10             aceOfSpade,
11             twoOfHeart,
12             threeOfDiamond,
13             fourOfClub
14         );
15
16         while (aStack.hasCards()) {
17             PlayingCard top = aStack.getTop();
18             System.out.printf(
19                 "top.rank = %s, top.suit= %s%n",
20                 top.getRank(),
21                 top.getSuit()
22             );
23             aStack.take();
24         }
25     }
26 }
```

```
25     }
26 }
```

Ceci termine notre introduction aux génériques. Ce concept sera approfondi dans le cours de programmation avancée de deuxième année. Après avoir étudié les types de polymorphismes qui prennent place à la compilation, étudions le polymorphisme qui a lieu pendant l'exécution.

6.4. Le polymorphisme par sous-typage choisit la version d'une méthode à exécuter

Pour introduire le polymorphisme par sous-typage, partons d'un exemple. Vous déclarez une référence à une **BaseCard** et demandez à l'utilisateur s'il doit lui affecter une carte à jouer ou une carte **TCG**. Le mécanisme d'affectation ascendante autorise ce genre d'affectation tant qu'une relation d'héritage existe entre le type de la référence et celui de l'objet créé. Vous souhaitez afficher la carte en appelant **toString()**. Qu'affiche le programme ?

Un exemple de polymorphisme par sous-typage

```
1 public class SubTypePolymorphism {
2
3     public static void main(String[] args) {
4         String cardType = Console.readLine(
5             "What type do you wish ? (P)laying card or (T)cg card : "
6         );
7         BaseCard myCard = switch (cardType.toUpperCase()) {
8             case String p when p.startsWith("P") -> new PlayingCard(
9                 Rank.ACE,
10                Suit.SPADE
11            );
12            case String p when p.startsWith("T") -> new TCGCard(
13                "Great hall",
14                "Set a mission's health and mana to 3",
15                2,
16                3
17            );
18            default -> new BaseCard(true);
19        };
20
21         System.out.printf("Votre carte : %n%s", myCard.toString());
22     }
23 }
```

➤ Sortie avec P encodé

```
What type do you wish ? (P)laying card or (T)cg card : P
Votre carte :
A♠
```

➤ Sortie avec T encodé

```
What type do you wish ? (P)laying card or (T)cg card : : T
```

```
Votre carte :  
Great hall  
-----  
HP: 2  
MP: 3  
Set a mission's health and mana to 3
```

La réponse dépend du choix de l'utilisateur. Si ce dernier répond par P, le programme affichera une carte à jouer. S'il répond par T, il affichera une carte à échanger. Dans les autres cas, le programme affiche une simple carte.

C'est ici qu'intervient le polymorphisme par sous-typage. Contrairement aux formes précédentes, la version de la méthode `toString()` à exécuter est décidée à l'exécution, en fonction du type concret de l'objet. Ce type de polymorphisme convient particulièrement à la rédaction de code devant être flexible pendant l'exécution.

En Java, nous pouvons expliquer ce comportement par le fait que la liaison des méthodes se fait de façon tardive, c'est-à-dire pendant l'exécution du programme. Cette liaison tardive met en œuvre le polymorphisme par sous-typage.

6.4.1. Liaison précoce et tardive des méthodes

La liaison de méthode consiste à choisir les instructions à exécuter en réponse à un appel. Deux types de liaisons existent, déterminant le moment où la liaison est faite :

- La liaison précoce (*early binding*) qui prend place pendant la compilation ;
- La liaison tardive (*late binding*) qui a lieu durant l'exécution.

Cette dernière met en œuvre le **polymorphisme par sous-typage**, celui que la méthode `printf(String, Object...)` utilise quand elle appelle `toString()` sur ses paramètres.

Prouvons par l'absurde que la liaison à la méthode `toString()` est tardive. Si elle était précoce, le compilateur choisirait les instructions à exécuter. Par définition, le compilateur ne peut pas prédire le choix de l'utilisateur. En conséquence, son choix se limite au type de la référence. Le programme appellera toujours `BaseCard.toString()`. Les sorties de l'extrait précédent prouvent que cela n'est pas le cas : la liaison doit donc intervenir au moment de l'exécution.

Pour comprendre comment Java implémente la liaison tardive, inspectons la structure d'une référence à un objet (reference-structure). Une référence mémorise l'adresse d'un objet qui vit sur le tas. À cette adresse se trouve une structure composée des champs de l'objet et de métadonnées. Les métadonnées reprennent, entre autres choses, le nom de la classe concrète de l'objet et une table qui associe à chaque signature de méthode l'adresse de la première instruction à exécuter. La table ([Figure 41](#)) responsable d'associer une signature aux instructions à exécuter se nomme table virtuelle des méthodes.

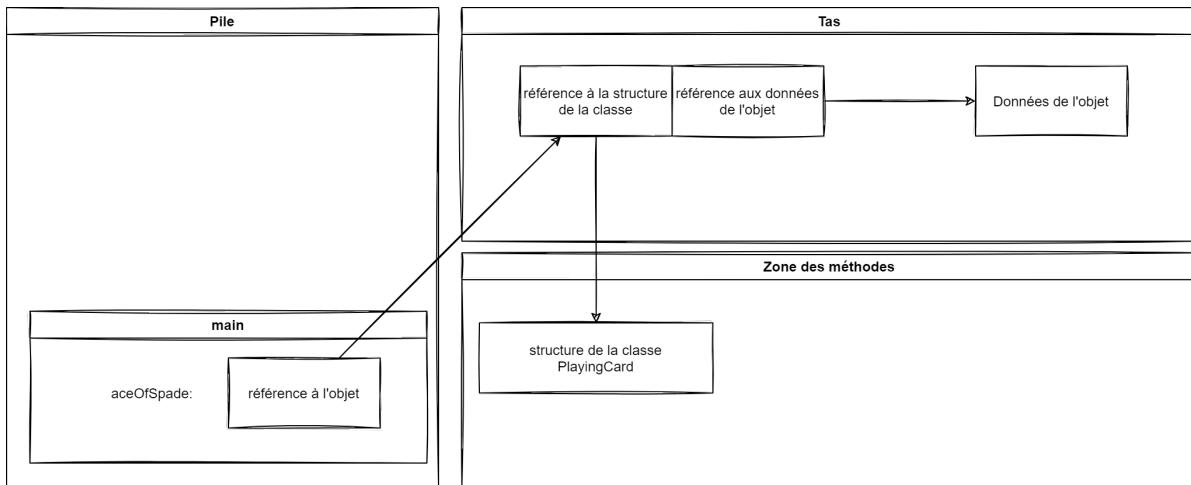


Figure 40. Détails d'une référence à un objet

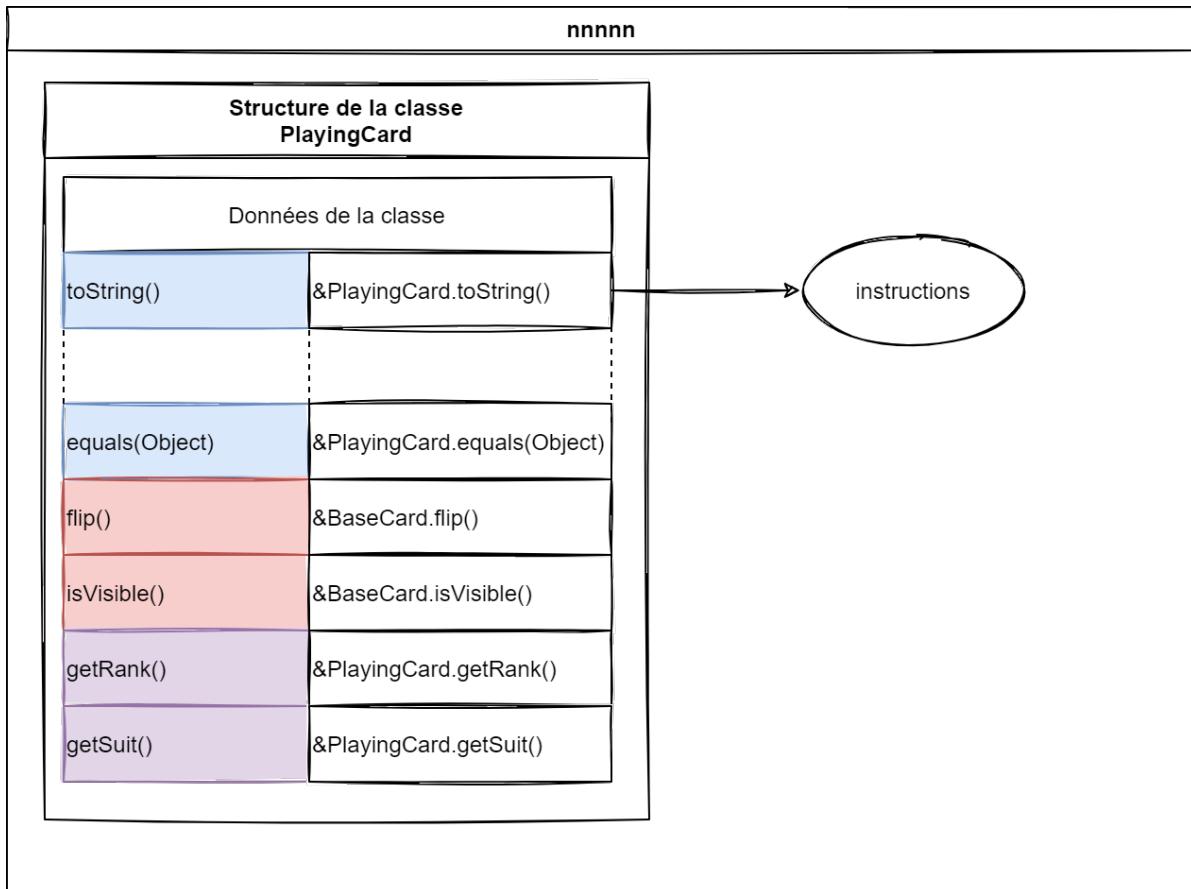


Figure 41. Détails de la structure d'une classe

Le compilateur crée une table virtuelle des méthodes pour chaque classe. Pour chaque méthode accessible, il détermine si une définition existe dans la classe concrète. Si oui, il l'affecte à l'entrée correspondante. Sinon, il inspecte la définition de la classe mère. Il remonte ainsi la hiérarchie jusqu'à tomber sur une version de la méthode. En outre, la table est organisée pour que les méthodes respectent l'ordre de leur première déclaration.

Les méthodes liées de façon précoce n'utilisent pas de tables virtuelles. Les adresses des instructions à exécuter sont directement fournies par le compilateur. La liaison précoce concerne les méthodes privées et les méthodes finales qui ne sont pas des redéfinitions. Le tableau suivant compare les liaisons tardives et les liaisons précoces.

| Critère | Liaison précoce | Liaison tardive |
|----------------------------------|-------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| Détermine à la méthode à appeler | À la compilation, selon le type de la référence. | À l'exécution, en fonction du type concret de l'objet. |
| Types de méthodes concernées | Méthodes statiques, finales, privées ou constructeurs | Méthodes d'objet non finales et redéfinitions |
| Performance | Plus rapide, car résolu à la compilation. | Plus lente, car nécessite une recherche dans la table des méthodes virtuelles (v-table) à l'exécution. |
| Polymorphisme par sous-typage | Non | Oui |



Contrairement à Java, en C++ et en C#, les liaisons sont précoces par défaut. Pour lier tardivement une méthode, vous devrez ajouter le modificateur **virtual** à son en-tête.

Avec de grands pouvoirs viennent de grandes responsabilités. Le polymorphisme par sous-typage est puissant, mais pas sans risque sur le code dont vous héritez. Si vous n'y prenez pas garde, un descendant peut détruire l'état défini par ses ancêtres.

6.5. Le polymorphisme par sous-typage demande un soin particulier pour l'initialisation

Si vous n'y prenez pas garde, les redéfinitions de méthodes et le polymorphisme par sous-typage peuvent casser l'initialisation d'objets.

Rappelons que, pour garantir une initialisation correcte de tous les champs d'un objet, Java appelle d'abord le constructeur de la classe mère avant d'exécuter les instructions spécifiques au constructeur de la classe concrète. L'appel peut être implicite, Java appelle le constructeur sans paramètres, ou explicite grâce au mot-clé **super**. Cette règle garantit que les membres dont votre classe hérite sont initialisés avant d'exécuter les instructions spécifiques à votre classe.

Un ancêtre peut cependant perdre le contrôle de son initialisation lorsque son constructeur appelle des méthodes redéfinissables. En effet, Java exécutera la version correspondant au type concret de l'objet. Cette dernière peut adopter un comportement qui ne correspond pas à celui attendu par l'ancêtre.

La déclaration de classe suivante est a priori anodine, mais elle possède une faille. En effet, **add(T)** est redéfinissable et le constructeur appelle cette méthode.

Le constructeur appelle une méthode redéfinissable

```
1 public class CardsCollection<T extends BaseCard> {
```

```

2
3     private ArrayList<T> cards = new ArrayList<T>();
4
5     public CardsCollection(T... cards) {
6         Contract.require(cards != null, "Paramètre cards indéfini");
7         this.cards = new ArrayList<T>();
8
9         for (int i = 0; i < cards.length; ++i) {
10             add(cards[i]);
11         }
12     }
13
14     public void add(T card) {
15         Contract.require(card, card != null, "Paramètre card indéfini");
16         this.cards.add(card);
17     }
18
19     public int getCount() {
20         return cards.size();
21     }
22
23     public T getCardsAt(int index) {
24         checkInRange(index, "index");
25         return this.cards.get(index);
26     }
27
28     public boolean isVisible(int posStart, int posEnd) {
29         int checkedPosStart = checkInRange(posStart, "posStart");
30         int checkedPosEnd = checkInRange(posEnd, "posEnd");
31         boolean allVisible = true;
32
33         for (
34             int cardPos = checkedPosStart;
35             cardPos <= checkedPosEnd;
36             ++cardPos
37         ) {
38             allVisible = allVisible && getCardsAt(cardPos).isVisible();
39         }
40
41         return allVisible;
42     }
43
44     public void flip(int posStart, int posEnd) {
45         int checkedPosStart = checkInRange(posStart, "posStart");
46         int checkedPosEnd = checkInRange(posEnd, "posEnd");
47
48         for (
49             int cardPos = checkedPosStart;
50             cardPos <= checkedPosEnd;
51             ++cardPos
52         ) {
53             getCardsAt(cardPos).flip();
54         }
55     }
56
57     private int checkInRange(int value, String name) {
58         return Contract.require(

```

```

59         value,
60         0 <= value && value < getCount(),
61         "%s devrait être dans [0; %d]. Reçu %d.".formatted(
62             name,
63             getCount(),
64             value
65         )
66     );
67 }
68 }
```

La déclaration suivante hérite de notre classe et redéfinit la méthode `add(T)` pour empêcher les ajouts. Elle casse le comportement défini par la classe mère. En effet, le constructeur de `CardsCollection<T>` s'exécute en premier et appelle la redéfinition de `ImmutableCardsCollection<T>.add(T)`. Cette redéfinition est vide d'instructions. En conséquence, la collection déclarée dans `CardsCollection<T>` demeure vide.

La redéfinition casse le comportement de la classe mère

```

1 public class ImmutableCardsCollection<T extends BaseCard> extends CardsCollection<T> {
2     public ImmutableCardsCollection(T...cards) {
3         super(cards);
4     }
5
6     @Override
7     public void add(T card) {
8         //Ne fait rien
9         return;
10    }
11 }
```

Ce programme ne se comporte pas comme prévu

```

1 public class ImmutableCardsCollectionSample {
2
3     public static void main(String[] args) {
4         var aceOfSpade = new PlayingCard(Rank.ACE, Suit.SPADE);
5         var twoOfHeart = new PlayingCard(Rank.TWO, Suit.HEART);
6         var threeDiamond = new PlayingCard(Rank.THREE, Suit.DIAMOND);
7         var fourOfClub = new PlayingCard(Rank.FOUR, Suit.CLUB);
8
9         var cardsCollection = new ImmutableCardsCollection<>(
10             aceOfSpade,
11             twoOfHeart,
12             threeDiamond,
13             fourOfClub
14         );
15
16         System.out.printf("count = %s\n", cardsCollection.getCount());
17         System.out.printf("cards at 2 = %b\n", cardsCollection.getCardAt(2));
18     }
19 }
```

```

count = 0
Exception in thread "main" java.lang.IllegalArgumentException: index devrait être dans [0; 0].
Reçu 2.
at utils.Contract.require(Contract.java:58)
at ch06.CardsCollection.checkInRange(CardsCollection.java:63)
at ch06.CardsCollection.getCardsAt(CardsCollection.java:29)
at ch06.ImmutableCardsCollectionSample.main(ImmutableCardsCollectionSample.java:19)

```

Pour éviter ces fuites potentielles, évitez qu'un constructeur n'appelle des méthodes redéfinissables. Toute méthode appelée par un constructeur devrait être privée, finale ou statique. La solution la plus sûre reste de rendre la classe finale, autrement dit d'interdire l'héritage.



Interdire l'héritage par défaut

Les relations d'héritage introduisent des dépendances fortes entre une classe mère et ses filles. Sans documentation soignée, les classes filles peuvent casser le code de la classe mère. Inversement, tout changement apporté à la classe mère peut casser ses filles. Pour réduire ces risques, il est conseillé d'interdire l'héritage par défaut.

Observez les classes du JDK, vous constaterez qu'elles adoptent majoritairement cette approche.

Le problème que nous avons étudié trouve son origine dans la relation d'héritage. En effet, nous avons indiqué qu'une collection immuable de cartes est une collection de cartes mutables, ce qui n'a aucun sens ! À cause de cette relation erronée, nous avons modifié le comportement d'une méthode pourtant pertinente pour la classe mère. Nous devrions nous assurer que tout objet d'une classe se comporte comme tous les objets de ses classes ancêtres pour les échanger facilement. C'est ce que préconise le principe de substitution de Liskov.

6.6. Une relation d'héritage correcte respecte les contrats des ancêtres

Un contrat bien connu des programmeurs Java est celui de `Object.hashCode()` dont vous retrouvez l'extrait ci-dessous. Le deuxième item du contrat lie le résultat retourné par `hashCode()` au résultat de `equals(Object)`. Si vous ne respectez pas ce contrat, les clients de vos objets ne se comporteront pas comme attendus. C'est notamment le cas avec les collections standards de Java dépendantes du résultat de `hashCode()`.

The general contract of hashCode is:

- *Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.*

- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

— Javadoc de la classe Object,

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#hashCode>

L'extrait suivant montre le comportement d'une `HashSet<PlayingCard>` quand vous l'utilisez avec une classe qui ne respecte pas le contrat de `hashCode()`. Dans `PlayingCard`, nous avons redéfini `equals(Object)` sans avoir redéfini `hashCode()`. À cause de ce manquement, nous ne pouvons pas retrouver des copies de cartes dans notre collection, même si elles sont équivalentes.

Comportement étrange d'une collection Java, car notre définition ne respecte pas le contrat de `hashCode()`

```

1 public class HashSetSample {
2
3     public static void main(String[] args) {
4         var aceOfSpade = new PlayingCard(Rank.ACE, Suit.SPADE);
5         var twoOfHeart = new PlayingCard(Rank.TWO, Suit.HEART);
6         var threeDiamond = new PlayingCard(Rank.THREE, Suit.DIAMOND);
7         var fourOfClub = new PlayingCard(Rank.FOUR, Suit.CLUB);
8
9         var cardsSet = new HashSet<>();
10        Collections.addAll(
11            cardsSet,
12            aceOfSpade,
13            twoOfHeart,
14            threeDiamond,
15            fourOfClub
16        );
17
18        var clone = new PlayingCard(Rank.ACE, Suit.SPADE);
19        System.out.printf(
20            "|%10s|%10s|%10s|%10s|%n",
21            "Critère",
22            "aceOfSpace",
23            "clone",
24            ""
25        );
26        System.out.printf(
27            "|%10s|%10s|%10s|%10b|%n",
28            "equals ?",
29            aceOfSpade,
30            clone,
31            aceOfSpade.equals(clone)
32        );
33        System.out.printf(
34            "|%10s|%10d|%10d|%10b|%n",
35            "hash ?",
36            aceOfSpade.hashCode(),
37            clone.hashCode(),

```

```

38         aceOfSpade.hashCode() == clone.hashCode()
39     );
40     System.out.printf(
41         "|%10s|%10b|%10b|%10s|",
42         "in set ?",
43         cardsSet.contains(aceOfSpade),
44         cardsSet.contains(clone),
45         ""
46     );
47 }
48 }
```

➤ Sortie

| | | | |
|----------|------------|------------|-------|
| Critère | aceOfSpace | clone | |
| equals ? | A♠ | A♠ | true |
| hash ? | 1450495309 | 1324119927 | false |
| in set ? | true | false | |

Quand nous cassons le contrat d'une classe héritée, nous trompons ses utilisateurs. Dans cet exemple, nous faisons moins que ce que l'ancêtre promet. Autrement dit, notre relation d'héritage est incorrecte. Ce raisonnement vaut aussi pour [ImmutableCardsCollection](#).

L'extrait suivant redéfinit `hashCode()` dans la classe `PlayingCard` en respectant son contrat. La redéfinition suffit à rendre notre programme correct.

Redéfinition de `hashCode()` respectueuse du contrat

```

1 public final class PlayingCard extends BaseCard {
2     @Override
3     public boolean equals(Object other) {
4         if (this == other) {
5             return true;
6         }
7         return other instanceof PlayingCard that && this.equals(that);
8     }
9
10    @Override
11    public int hashCode() {
12        return Objects.hash(rank, suit);
13    }
14    // Code inchangé
15 }
```

➤ Sortie avec les modifications

| | | | |
|----------|------------|------------|------|
| Critère | aceOfSpace | clone | |
| equals ? | A♠ | A♠ | true |
| hash ? | -608502698 | -608502698 | true |
| in set ? | true | true | |

Respecter le contrat posé par une classe importe. Mais qu'entendons-nous précisément par contrat ? En POO, un contrat spécifie le comportement des objets d'une classe en stipulant :

- Des préconditions qui sont les règles que les utilisateurs de l'objet doivent respecter ;
- Des postconditions qui sont les règles que tout objet de la classe doit respecter quand les préconditions sont respectées ;
- Des invariants qui sont des règles que tout objet de la classe doit respecter pendant sa vie ;
- Les mutations qui expliquent comment l'état d'un objet est modifié suite à un appel de méthode ;
- Les exceptions produites lorsqu'une règle est enfreinte.

Contrat d'une classe



Description formalisée des comportements des objets d'une classe à l'aide de préconditions, invariants, postconditions, mutations et exceptions.

Java ne propose pas de mécanismes dédiés à la déclaration des contrats^[3]. Nous pouvons cependant décrire le contrat dans la Javadoc et l'implémenter avec des classes utilitaires comme la classe **Contract**.

Interdire l'héritage ou le documenter soigneusement



Concevoir une bonne Javadoc pour une classe revient à décrire les éléments d'un contrat de classe. Ce contrat devrait être respecté par tous les clients de classe, en particulier par ses descendants.

Quand une classe **X** hérite d'une classe **Y**, nous devons nous attendre à ce que ses objets soient manipulés via des références à des **Y**. Pour garantir une bonne utilisation des objets **X** dans ce contexte, nous devons respecter le contrat défini par **Y**. Autrement dit, tout objet **X** doit en faire autant que tout objet **Y**. Nous pouvons cependant :

- Affaiblir les préconditions, c'est-à-dire accepter un domaine de valeur plus grand que celui de la superclasse ;
- Renforcer les postconditions, c'est-à-dire restreindre le domaine de valeur produit par la superclasse.

Prenons une méthode **m** définie dans **Y** qui n'accepte pas la valeur **null** comme argument pour un paramètre **p**. La redéfinition de **m** dans **X** peut accepter **null** comme argument de **p**. La redéfinition accepte tous les appels à **m** qu'un objet **Y** peut recevoir, et elle accepte également de nouveaux appels spécifiques à **X**. Nous pouvons substituer un objet **X** partout où un objet **Y** est attendu.

Si nous respectons ces contraintes, nous nous conformons au **Principe de Substitution de Liskov (Liskov Substitution Principle)**. Nous parlons de substitution, car le respect de ce principe permet de remplacer un objet d'une classe ancêtre par un objet d'une classe descendante en toute transparence pour les utilisateurs de l'objet ancêtre.

Principe de substitution de Liskov



Si **p(y)** est une propriété démontrable pour tout objet **y** de type **Y**, alors **p(x)** est vraie pour tout objet **x** de type **X** quand **X** est un sous-type de **Y**. Autrement

dit, les objets d'une classe doivent respecter les contrats imposés par ses ancêtres.

6.7. Les affectations descendantes sont dangereuses

Une affectation est ascendante quand le type de la référence qui reçoit correspond à un ancêtre de l'objet affecté. Java effectue une affectation ascendante de façon implicite. En effet, le langage garantit que l'objet référencé peut répondre aux appels reçus par la référence.

Un descendant peut exposer plus de méthodes que ses ancêtres. Elles sont cependant invisibles si le descendant est mémorisé par une référence à un type ancêtre. Pour appeler les méthodes spécifiques, vous devez explicitement convertir la référence dans le type du descendant. Est-il toujours possible de faire cette conversion ? Autrement dit, serait-il possible de réaliser une affectation descendante entre des objets ?

Affectation descendante d'objet



Affectation d'une référence de type ancêtre à une référence de type descendant. Elle s'oppose à l'affectation ascendante.

Répondre à cette question demande d'étudier deux possibilités :

1. L'objet à affecter appartient au type de la référence affectée, l'affectation est alors légale.
2. L'objet à affecter n'appartient pas au type de la référence affectée, la JVM lance une **ClassCastException**, car l'objet peut ne pas être en mesure de répondre à tous les appels de méthode.

Exemple d'affectation descendante incorrecte

```
1 // Affectation ascendante car BaseCard <|-- TCGCard
2 BaseCard card = new TCGCard(
3     "Great hall",
4     "Set a mission's health and mana to 3",
5     2,
6     3
7 );
8 // ✗ ClassCastException lancée
9 PlayingCard toPlayCard = (PlayingCard) animal;
```

Considérons la [Figure 42](#) qui présente les méthodes appelables sur une collection de cartes et de deux de ses classes filles. Comme vous pouvez le constater, les filles possèdent des méthodes spécifiques. Par exemple, **CardsHand<T>** possède une méthode **takeAt(int)** que ne possède pas **CardsStack<T>**. Inversement, **CardsStack<T>** déclare une méthode **getTop()** qui n'existe pas dans **CardsHand<T>**. En conséquence, un objet **CardsStack<T>** ne peut pas répondre à tous les appels qu'un objet **CardsHand<T>** peut recevoir et inversement.

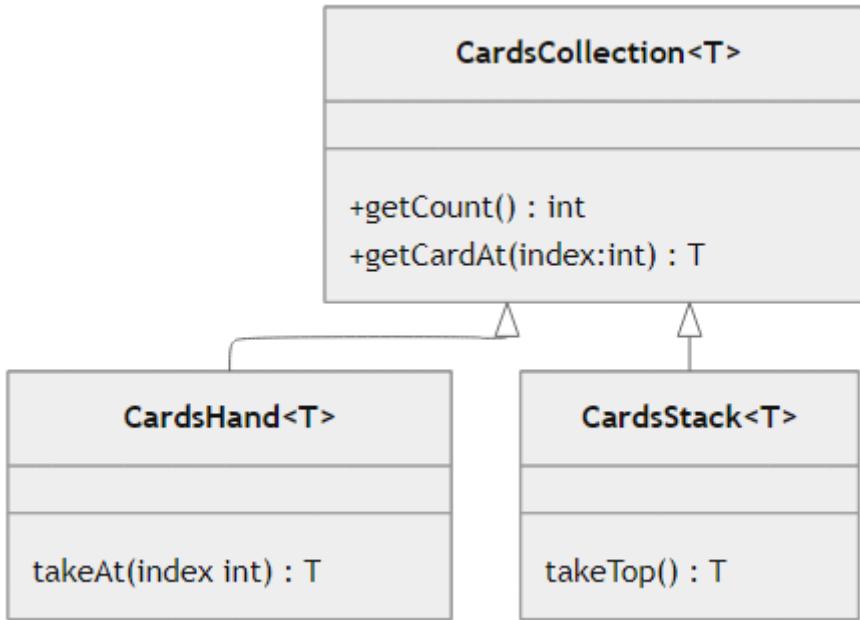


Figure 42. Une hiérarchie de classes où les filles exposent leurs propres méthodes

La méthode ci-dessous essaie de supprimer la première carte d'une collection de cartes. La méthode de suppression à appeler dépend de la classe concrète de l'objet : nous devons opérer une affectation descendante pour appeler la méthode spécifique. Pour éviter les soucis de conversion, nous devons tester l'appartenance de l'objet à affecter au type de la référence.

Tentative de suppression

```

1 public void removeFirst(CardsCollection<T> collection) {
2     Objects.requireNonNull(collection);
3
4     if(collection instanceof CardsStack<T>) {
5         CardsStack<T> asStack = (CardsStack<T>)collection;
6         asStack.take();
7     } else if(collection instanceof CardsHand<T>) {
8         CardsHand<T> asHand = (CardsHand<T>)collection;
9         asHand.takeAt(0);
10    }
11    // Ajouter les collections spécifiques à la suite
12 }

```

Les tests d'appartenance à un type sont indispensables. Sans eux, vous risquez d'obtenir une exception `ClassCastException`. La JVM lance cette exception quand vous tentez d'affecter un objet à une référence incompatible. Elle pourrait survenir si vous affectez un objet `CardsHand<T>` à une référence à un `CardsStack<T>`.

Depuis Java 16, nous pouvons simplifier ce code avec les patrons de type pour `instanceof`. Cet opérateur teste et, le cas échéant, convertit une référence en une référence d'un type descendant.

Tentative de suppression depuis Java 16

```

1 public void removeFirst(CardsCollection<T> collection) {
2     Objects.requireNonNull(collection);
3

```

```

4  if(collection instanceof CardsStack<T> asStack) {
5      asStack.take();
6  } else if(collection instanceof CardsHand<T> asHand) {
7      asHand.takeAt(0);
8  }
9 // Ajouter les nouvelles collections spécifiques à la suite
10 }

```

La méthode `removeFirst(CardsCollection)` n'exploite pas le polymorphisme par sous-typage et est fragile aux changements. En effet, nous devrons la modifier pour gérer de nouvelles sortes de collections de cartes. Elle présente un indicateur de code qui sent mauvais : la présence d'une instruction de branchement (*Switch statement smell*). En effet, elle peut facilement être réécrite avec une instruction `switch` et les patrons de type.

Tentative de suppression depuis Java 16

```

1 public void removeFirst(CardsCollection<T> collection) {
2     Objects.requireNonNull(collection);
3
4     switch(collection) {
5         case CardsStack<T> asStack -> asStack.take();
6         case CardsHands<T> asHand -> asHand.takeAt(0);
7         // Ajouter les nouvelles collections spécifiques à la suite
8         default -> {}
9     }
10 }

```

En POO, de telles instructions devraient être remplacées par du polymorphisme par sous-typage^[4]. Citons à ce propos un célèbre développeur.

Anytime you find yourself writing code of the form "if the object is of type T1, then do something, but if it's of type T2, then do something else," slap yourself.

— Scott Meyer, Effective C++

La solution consiste à uniformiser les méthodes de la hiérarchie, comme le montre la [Figure 43](#). Quand vous concevez une hiérarchie d'héritage uniforme, vous manipulez tous les objets de la hiérarchie de la même façon. Vous pouvez alors collaborer avec ces objets à travers des références à l'ancêtre commun. Quand votre hiérarchie d'héritage est uniforme, vous n'êtes plus tentés par les conversions descendantes.

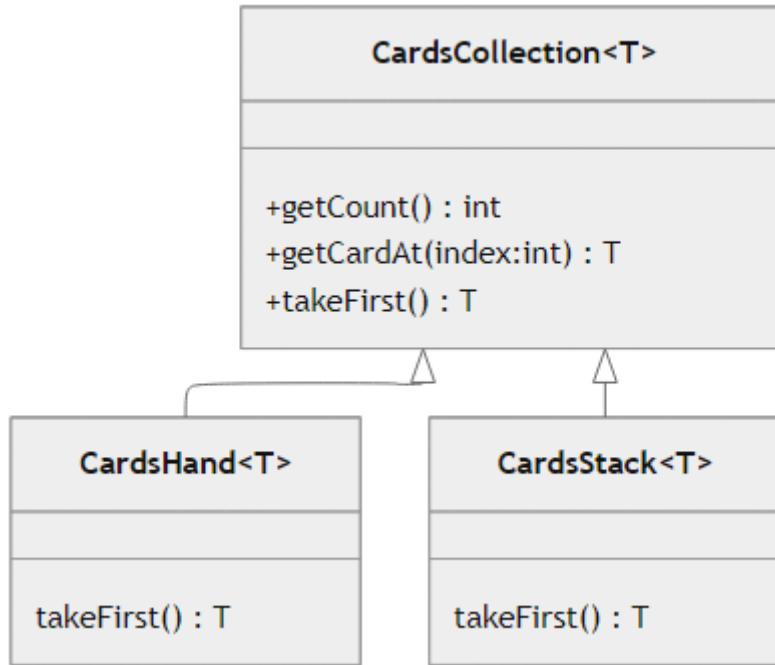


Figure 43. Une hiérarchie de classe où les filles exposent la même interface que leur mère

Notez qu'une hiérarchie d'héritage uniforme ne supprime pas les problèmes liés aux affectations descendantes. En effet, Java ne possède pas de mécanismes de vérification de l'uniformité d'une hiérarchie. Sans lui, seules les affectations ascendantes sont implicites. Cependant, les dernières évolutions du langage, telles que les classes scellées, ouvrent la voie à ce support.

6.8. En résumé

Java exploite trois formes de polymorphisme. Le tableau suivant les résume.

Table 10. Résumé des trois types de polymorphisme en Java

| Type de polymorphisme | Résolution | Principes |
|------------------------|-------------------------|-------------------------------------------------------------------------------|
| Ad hoc | À la compilation | Identifie la surcharge à utiliser sur base des arguments |
| Générique | À la compilation | Paramètre une déclaration par des types qui seront remplacés à l'utilisation. |
| Par sous-typage | À l' exécution | Identifie la redéfinition à appliquer selon le type concret de l'objet |

Grâce au polymorphisme par sous-typage, Java garantit que tout appel de méthode d'un objet entraîne l'exécution des instructions propres à la classe concrète de cet objet. À cette fin, Java réalise une liaison tardive, pendant l'exécution du programme. C'est un apport majeur de la POO par rapport à la programmation procédurale, car il permet d'étendre du code sans avoir à le modifier.

Restez cependant vigilant à certains pièges :

- Appeler des méthodes redéfinissables dans un constructeur peut entraîner des erreurs difficiles à identifier ;
- Ajouter des méthodes spécifiques à des classes descendantes peut nous exposer aux problèmes liés à l'affectation descendante ;
- Casser les contrats des classes ancêtres peut affecter le comportement de leurs clients.

Bien utilisé, le polymorphisme facilite l'évolution du code. Les principes ouvert-fermé et de substitution de Liskov constituent les règles à suivre. À l'inverse, l'utilisation d'affectation descendante est un signal d'alarme à écouter pour éviter d'avoir du code difficilement maintenable.

Ce chapitre et celui qui le précède nous ont expliqué comment réutiliser du code grâce au concept d'héritage de classe. Définir une relation d'héritage consiste à spécialiser la définition d'une classe mère. Ainsi, plus nous remontons vers la classe **Object**, plus nos définitions sont générales, dépourvues de détails. Dans le chapitre suivant, nous poussons cette notion d'abstraction plus loin.

- [1] Ce n'est pas possible pour des langages aux typages faibles comme PHP ou Python.
- [2] Voir le cours d'algorithme
- [3] D'autres langages implémentent les contrats, comme le langage Eiffel, inventé par Bertrand Meyer.
- [4] Une exception à cet indicateur est le cas où vous souhaitez créer un objet parmi plusieurs classes candidates héritant toutes d'un même classe.

Chapitre 7. Les classes abstraites et les interfaces

Ce chapitre étudie les classes et les méthodes abstraites ainsi que les interfaces. Grâce à elles, vous pouvez déclarer des méthodes sans instructions, et forcer d'autres types à les implémenter.

Au [Chapitre 6](#), nous avons vu l'importance des hiérarchies uniformes pour éviter les affectations descendantes. Mais comment garantir qu'une hiérarchie reste uniforme ? Comment forcer les classes filles à implémenter certaines méthodes ?

Si vous suivez le principe des hiérarchies uniformes, vous constaterez que l'ancêtre déclarera des méthodes pour lesquelles il est difficile, voire impossible, d'écrire des instructions. En effet, les implémentations seront spécifiques à chaque descendant. Imaginez que vous développez un jeu avec différents types d'acteurs : des joueurs et un croupier. Tous doivent pouvoir « jouer leur tour », mais vous ignorez encore comment chacun le fera. Vous avez besoin d'un « contrat » que toutes les classes devront respecter pour les utiliser de la même façon. En parallèle, vous ne souhaitez pas instancier un objet acteur trop général, car il sera incapable de jouer son tour.

Les classes abstraites et les interfaces répondent à ces besoins.

7.1. Les classes abstraites ne sont pas concrètes

Au blackjack, le croupier et les joueurs ont des comportements communs : ils sont notamment responsables de déterminer leur prochaine action. Pour simplifier, nous dirons que deux actions sont possibles : tirer (« *hit* ») ou rester (« *stand* »). L'algorithme qui détermine la prochaine action est spécifique à chaque type d'acteur. Partant de ce constat, nous pouvons définir une hiérarchie semblable à la [Figure 44](#)

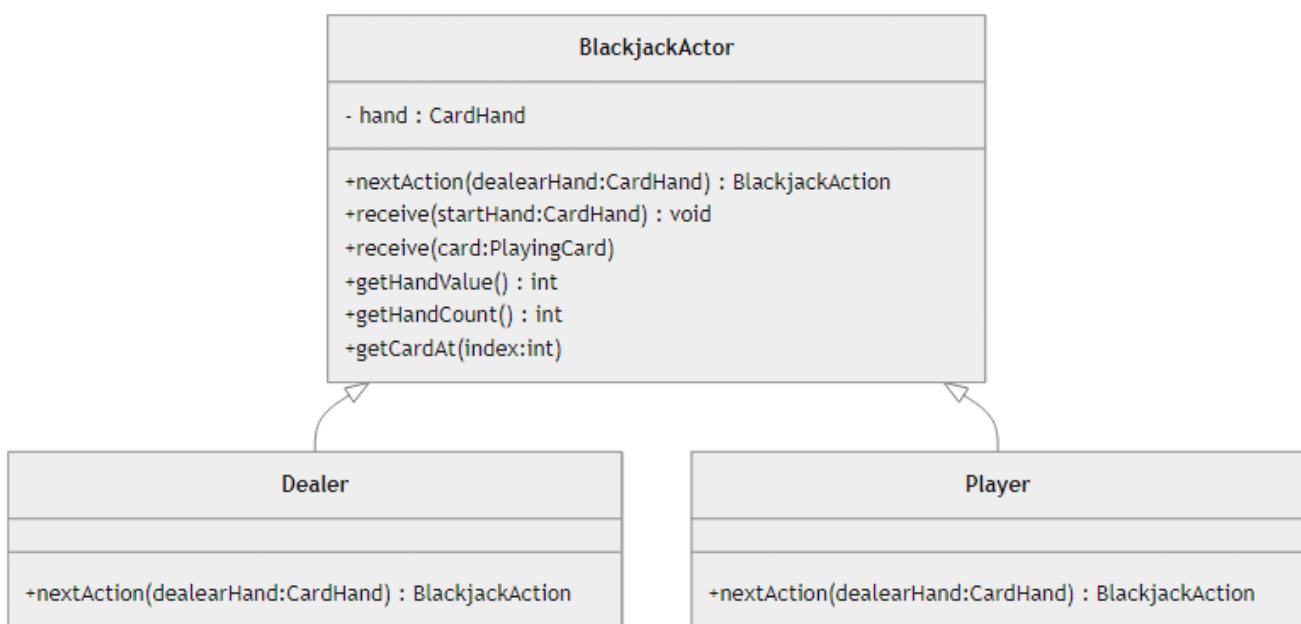


Figure 44. Hiérarchie d'héritage des acteurs de blackjack

L'extrait suivant déclare la classe `BlackjackActor`.

Une classe composée de méthodes inutiles

```
1 public class BlackjackActor {  
2  
3     private CardHand hand = null;  
4  
5     public BlackjackAction nextAction(CardHand dealerHand) {  
6         // Cette méthode doit être redéfinie  
7         return BlackjackAction.NONE;  
8     }  
9  
10    public void receive(CardHand hand) {  
11        this.hand = Objects.requireNonNull(hand, "Argument non défini");  
12    }  
13  
14    public void receive(PlayingCard card) {  
15        hand.append(card);  
16    }  
17  
18    public int getHandValue() {  
19        checkHasHand();  
20        return hand.getValue();  
21    }  
22  
23    public int getHandCount() {  
24        checkHasHand();  
25        return this.hand.getCount();  
26    }  
27  
28    public PlayingCard getCardAt(int index) {  
29        checkHasHand();  
30        return this.hand.getCardAt(index);  
31    }  
32  
33    private void checkHasHand() {  
34        Contract.check(this.hand != null, "Pas de main défini");  
35    }  
36 }
```

Cette déclaration nous pose un problème : sa méthode déterminant l'action suivante ne fait rien. Elle *doit* être redéfinie par ses descendants. Actuellement, vous pouvez instancier cette classe. Vous pouvez en outre en hériter et oublier de redéfinir la méthode. Dans les deux cas, vous obtenez des joueurs incapables de jouer.



Nous pourrions supprimer la méthode de la classe mère, mais cela nous expose au problème des affectations descendantes ([Section 6.7](#)).

Nous souhaitons interdire l'instanciation de `BlackjackActor` et forcer ses descendants à implémenter `nextAction(CardHand)`. Ceci reviendrait à déclarer l'en-tête de la méthode `BlackjackActor.nextAction(CardHand)` sans lui associer de corps. Bonne nouvelle, Java implémente cette approche.

Java autorise la déclaration d'une méthode sans corps avec le modificateur **abstract** : nous parlerons **d'une méthode abstraite**. Déclarer une méthode abstraite n'est cependant pas sans conséquences. Un objet d'une classe déclarant des méthodes abstraites ne peut pas répondre aux appels à ces méthodes abstraites, car elles sont sans corps : il est inutilisable.

Instancier une telle classe est dénué de sens et doit être interdit. Pour indiquer au compilateur une classe qui ne peut pas être instanciée, utilisez de nouveau le mot clé **abstract** dans son en-tête : votre classe est devenue **abstraite**.

Classe abstraite



Classe que vous ne pouvez pas instancier, car elle *peut* déclarer des méthodes abstraites.

Méthode abstraite

Méthode d'objet sans corps. Dès que vous qualifiez une méthode d'abstraite, vous devez qualifier sa classe d'abstraite.

Rendons la classe **BlackjackActor** abstraite. Remarquez la présence du modificateur **abstract** dans l'en-tête de la classe et dans celui de la méthode `nextAction(CardHand)`. Faute de corps, Java nous impose de terminer la déclaration de la méthode par un `« ; ».

Déclaration d'une classe et d'une méthode abstraites

```
1 public abstract class BlackjackActor {  
2  
3     private CardHand hand = null;  
4  
5     public abstract BlackjackAction nextAction(CardHand dealerHand);  
6  
7     public void receive(CardHand hand) {  
8         this.hand = Objects.requireNonNull(hand, "Argument non défini");  
9     }  
10  
11    public void receive(PlayingCard card) {  
12        hand.append(card);  
13    }  
14  
15    public int getHandValue() {  
16        checkHasHand();  
17        return hand.getValue();  
18    }  
19  
20    public int getHandCount() {  
21        checkHasHand();  
22        return this.hand.getCount();  
23    }  
24  
25    public PlayingCard getCardAt(int index) {  
26        checkHasHand();  
27        return this.hand.getCardAt(index);  
28    }  
29}
```

```

30     private void checkHasHand() {
31         Contract.check(this.hand != null, "Pas de main défini");
32     }
33 }
```

`BlackjackActor` étant abstraite, le compilateur refuse dorénavant l'expression `new BlackjackActor()`. Notez que vous pouvez déclarer une classe abstraite sans méthodes abstraites. Cette déclaration vous empêchera d'instancier cette classe.

Vous ne pouvez pas créer d'objets d'une classe abstraite

```

1 error: BlackjackActor is abstract; cannot be instantiated
2     var a0 = new BlackjackActor();
3             ^-----^
```

Qu'est-ce qu'une abstraction ?



En programmation, une abstraction est une représentation simplifiée d'un concept, qui met en évidence ses caractéristiques essentielles et ignore les détails inutiles à la résolution d'un problème.

Une classe abstraite est une représentation tellement générale d'un concept qu'elle devient impossible à instancier. En effet, elle décrit des comportements communs, mais est incapable de fournir les instructions spécifiques aux sous-classes.

Instancier la classe `BlackjackActor` est dorénavant illégal. En revanche, vous pouvez déclarer des références à des `BlackjackActor` et leur affecter des objets issus de ses descendants... si ces derniers sont des classes concrètes.

Déclaration de références à la classe abstraite

```

1     BlackjackActor player1 = new ArtificialPlayer("P1 (COM)");
2
3     switch(player1.nextAction(dealerHand)) {
4         case HIT -> { /* Tirer une carte */}
5         case STAND -> { /* Passer au joueur suivant */}
6         case NONE -> { /* Situation inattendue */}
7     }
```

La sous-section suivante précise les règles à respecter pour rendre une classe concrète.

7.1.1. Une classe concrète implémente toutes ses méthodes

Nous ne pouvons plus instancier la classe `BlackjackActor`, car elle est devenue abstraite. Toutefois, nous souhaitons instancier les classes `Player` et `Dealer`. En effet, nous sommes capables à leur niveau d'écrire les instructions à exécuter quand elles reçoivent un appel à `nextAction(CardHand)`.

Pour instancier ces classes, nous devons donner un corps aux méthodes abstraites dont elles héritent. Autrement dit, nous devons implémenter la méthode `nextAction(CardHand)`.

Implémenter une méthode abstraite

Une classe fille implémente une méthode abstraite, héritée de sa mère, quand elle lui fournit un corps.



Classe concrète ou instanciable

Classe dont toutes les méthodes, en particulier les méthodes abstraites héritées, sont implémentées. Cette classe est instanciable et dépourvue du modificateur `abstract`.

L'extrait suivant présente la classe `Dealer`. Le modificateur `abstract` a disparu de son en-tête, car elle implémente la méthode `nextAction(CardHand)`. Nous pouvons donc l'instancier. Remarquez aussi l'absence du modificateur `abstract` dans l'en-tête de la méthode `nextAction(CardHand)`.

Cette classe est concrète, car elle implémente la méthode abstraite

```
1 public class Dealer extends BlackjackActor {  
2     @Override  
3     public BlackjackAction nextAction(CardHand dealerHand) {  
4         if(this.getHandValue() > 17) {  
5             return BlackjackAction.STAND;  
6         } else {  
7             return BlackjackAction.HIT;  
8         }  
9     }  
10 }
```



Bien entendu, par défaut, une classe est concrète en Java. Sinon, les codes des chapitres précédents ne serviraient à rien. Cependant aucun mot-clé n'existe cependant pour indiquer qu'une classe est concrète.

Notez enfin qu'une classe peut hériter d'une classe abstraite et décider de ne pas implémenter les méthodes abstraites dont elle hérite. Dans ce cas, Java vous oblige à déclarer cette classe abstraite. Heureusement, Java ne vous impose pas de déclarer une seconde fois la méthode abstraite dans le corps de votre nouvelle classe.

7.1.2. Les patrons de méthodes définissent un cadre à compléter

Parmi les techniques classiques implémentées avec des classes abstraites, les patrons de méthodes occupent une place de choix. Un patron de méthode est une classe abstraite déclarant une méthode définissant un squelette d'opération que ses descendants complèteront.

Par exemple, nous pouvons modéliser la plupart des jeux de cartes avec une classe abstraite et laisser l'implémentation des règles de jeu concrètes à ses descendants. La [Figure 45](#) présente les opérations principales d'une partie de cartes.

cadre pour une partie de cartes

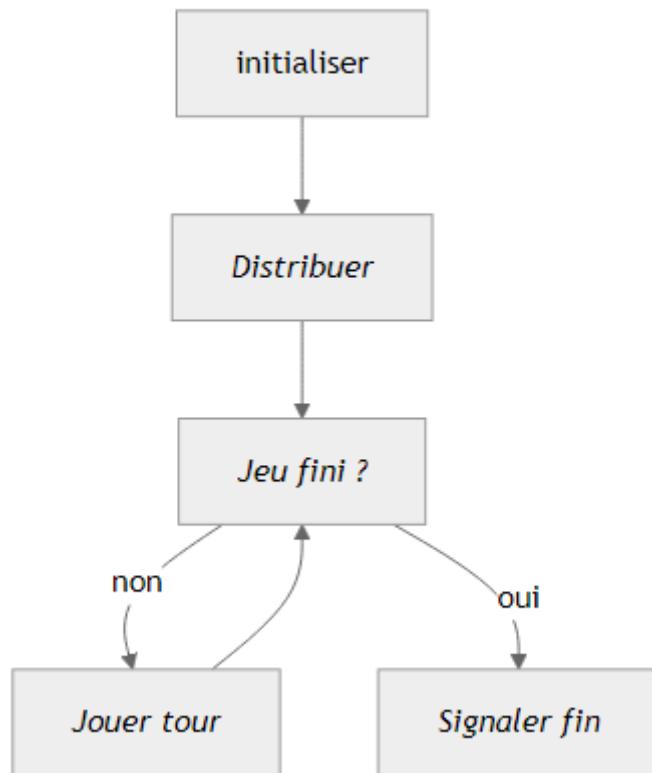


Figure 45. Cadre pour une partie de cartes

L'extrait suivant implémente ce cadre dans une méthode `play()`. Cette dernière appelle des méthodes abstraites dont on connaît, à ce stade, l'existence tout en étant incapable d'écrire les instructions exécutées derrière ces appels. À charge des descendants de fournir ces détails.

```
1 public abstract class AbstractCardGame {  
2  
3     private CardsCollection<PlayingCard> deck;  
4  
5     public AbstractCardGame() {  
6         var cards = new ArrayList<PlayingCard>(52);  
7         for (var suit : Suit.values()) {  
8             for (var rank : Rank.values()) {  
9                 cards.add(new PlayingCard(rank, suit));  
10            }  
11        }  
12        deck = new CardsCollection<PlayingCard>(  
13            cards.toArray(new PlayingCard[0])  
14        );  
15    }  
16  
17    protected final CardsCollection<PlayingCard> getDeck() {  
18        return deck;  
19    }  
20  
21    public final void play() {  
22        init();  
23        dealCards();  
24    }  
25}
```

```

24
25     do {
26         playTurn();
27     } while (!isGameOver());
28
29     end();
30 }
31
32 // Ces méthodes abstraites sont à implémenter par les descendants
33 protected abstract void init();
34
35 protected abstract void dealCards();
36
37 protected abstract void playTurn();
38
39 protected abstract boolean isGameOver();
40
41 protected abstract void end();
42 }

```

Le niveau d'accès des membres à implémenter est **protected**. Tout type appartenant au même paquetage ainsi que les descendants de la classe **AbstractCardGame** peuvent appeler ou redéfinir ces méthodes. Ce niveau se situe entre le niveau privé et le niveau public.



Ne rendez pas vos champs protégé

Les champs protégés sont directement consultables et modifiables par les descendants. Ceci constitue une infraction au principe d'encapsulation.

7.1.3. Les classes abstraites ont des défauts

Le concept de classe abstraite est intéressant pour :

- Déclarer une hiérarchie d'héritage uniforme et bénéficier ainsi du polymorphisme par sous-typage ;
- Implémenter les parties communes dans les ancêtres ;
- Implémenter les parties spécifiques dans les descendants ;
- Interdire l'instanciation de classes jugées trop générales pour être utiles.

Elles souffrent cependant des inconvénients liés à l'héritage de classes :

- L'héritage de classe est simple en Java. Elle définit un arbre de classes et une classe ne peut appartenir qu'à une seule branche de l'arbre. Cependant, certaines classes pourraient appartenir à plusieurs branches. Par exemple, un smartphone est simultanément un téléphone et un appareil photo. L'héritage simple exige alors du programmeur de faire des choix qui ne reflètent pas la réalité.
- L'implémentation d'une classe dépend fortement de celle de ses ancêtres. Modifier un ancêtre peut casser le fonctionnement de ses descendants. Pire, un descendant peut involontairement casser les parties dont il hérite (voir [Section 6.5](#)).

- La tentation est grande de créer des hiérarchies d'héritage techniques qui répondent à un besoin, mais qui ne se justifient pas sur le plan des comportements. Le principe de substitution de Liskov doit vous guider dans vos choix (voir [Section 6.6](#)).

Pour répondre à ces critiques, Java propose un second concept : celui des interfaces.

7.2. Les interfaces déclarent un contrat

En POO, une interface définit un comportement à implémenter sans préciser comment l'implémenter. Ce comportement prend la forme d'une liste de signatures de méthodes.

Les interfaces Java définissent une liste de méthodes abstraites à implémenter par une classe ou une enum. Cette liste définit des appels de méthodes recevables par tous les objets dont la classe implémente l'interface.

Interface en Java



Construction listant des méthodes publiques et abstraites à implémenter. En Java, une classe ou une enum implémente une interface.

Comme les classes et les enums, la déclaration d'une interface en Java compte plusieurs clauses :

- La déclaration commence par un modificateur d'accès. Si vous indiquez le modificateur **public**, l'interface est visible de partout. Sans modificateur, l'interface est visible par les types définis dans le même paquetage.
- Le nom de l'interface précédé du mot clé **interface**.
- Une liste d'interfaces héritées, précédée par le mot clé **extends**.
- Une liste de signatures de méthodes à implémenter.

Une interface prend une forme semblable à celle présentée par l'extrait suivant. Par défaut, les méthodes d'une interface sont abstraites : vous pouvez omettre le modificateur **abstract**. De plus, les méthodes, et plus généralement les membres, d'une interface sont publiques : impossible d'y déclarer des membres **private** ou **protected**. Enfin, contrairement aux classes, les interfaces ne possèdent pas d'état : seules les constantes y sont autorisées. Tout champ déclaré dans une interface sera implicitement **static** et **final**.

Exemple d'une interface Java

```

1 public interface HasRank {
2     int getRankValue();
3
4     boolean isFigure();
5     boolean isAce();
6     boolean isNumber();
7 }
```

Toute classe souhaitant implémenter une interface doit la citer dans la clause **implements** de son

en-tête. Le compilateur s'assure alors que la classe définit chaque méthode de l'interface, définition qui peut compter sur l'état de la classe. Si une des méthodes de l'interface n'est pas définie, vous devrez rendre la classe abstraite, sous peine d'erreurs de compilation.



Implémentation d'une interface

Définition de toutes les méthodes d'une interface par une classe ou une enum l'implémentant.

Étudions l'extrait suivant. La classe `PlayingCard` implémente l'interface `HasRank`, comme sa clause `implements` l'indique. Pour ce faire, elle doit définir chaque méthode déclarée par `HasRank`. Les méthodes définies sont annotées par `@Override` pour demander au compilateur des vérifications supplémentaires.

Exemple d'une implémentation d'interface en Java

```
1 public final class PlayingCard extends BaseCard implements HasRank {
2     private Rank rank;
3     private Suit suit;
4
5     public PlayingCard(Rank rank, Suit suit) {
6         super(true);
7         this.rank = Objects.requireNonNull(rank, "No rank provided");
8         this.suit = Objects.requireNonNull(suit, "No suit provided");
9     }
10
11    @Override
12    public boolean isAce() {
13        return this.rank == Rank.ACE;
14    }
15
16    @Override
17    public boolean isNumber() {
18        return Rank.ACE.compareTo(this.rank) >= 0
19            && this.rank.compareTo(Rank.NINE) <= 0;
20    }
21
22    @Override
23    public boolean isFigure() {
24        return rank.isFigure();
25    }
26
27    @Override
28    public int getRankValue() {
29        return rank.getValue();
30    }
31    // Code inchangé
32 }
```

Nous avons vu les fondamentaux des interfaces Java. Étudions maintenant leurs propriétés les plus importantes.

7.2.1. Une classe peut implémenter plusieurs interfaces

Une interface expose une liste de méthodes abstraites. Idéalement, les méthodes d'une interface forment un ensemble cohérent : elles décrivent un rôle à jouer. Que se passe-t-il quand une classe doit jouer plusieurs rôles ? Autrement dit, une classe peut-elle implémenter plusieurs interfaces ?

Contrairement à l'héritage de classe, l'implémentation des interfaces est multiple : une classe peut directement implémenter plusieurs interfaces. Implémenter plusieurs interfaces revient à ajouter une liste de noms après `implements`.

Java définit l'interface générique `Comparable<T>` qui se lit « comparable avec des objets T ». Sa méthode `compareTo(T o)` est utilisée par plusieurs collections Java pour garder leurs éléments ordonnés. Si nous voulons qu'une classe implémente `Comparable<T>`, il suffit de l'ajouter après `HasRank` dans la clause `implements`.

Désormais, la classe `PlayingCard` implémente deux interfaces : `HasRank` et `Comparable<PlayingCard>`. Elle implémente la méthode `compareTo(PlayingCard)`. Grâce à cette implémentation, nous pouvons trier un groupe de cartes à jouer selon leur rang en nous aidant des collections du JDK.

Implémentation de plusieurs interfaces

```
1 public final class PlayingCard extends BaseCard implements HasRank, Comparable<PlayingCard>
{
2
3     @Override
4     public int compareTo(PlayingCard that) {
5         Objects.requireNonNull(that);
6         int rankComparison = this.rank.compareTo(that.rank);
7
8         return rankComparison == 0
9             ? this.suit.compareTo(that.suit) : rankComparison;
10    }
11    // Reste du code inchangé
12 }
```

Envisagez d'implémenter Comparable



L'interface générique `java.util.Comparable<T>` définit un ordre naturel entre les objets de type `T`. Cette interface est utilisée par plusieurs collections du JDK, celles qui implémentent `SortedSet<T>` et `SortedMap<T, V>` notamment.

L'extrait suivant trie des cartes à jouer à l'aide d'un objet `TreeSet<PlayingCard>`. Cette classe organise ses éléments en un arbre pour que leur parcours respecte l'ordre défini par la méthode `compareTo(PlayingCard)`.

Utilisation de notre implémentation avec une collection triée

```
1 public class PlayingCardSample {
2
3     public static void main(String[] args) {
4         var threeOfDiamond = new PlayingCard(Rank.THREE, Suit.DIAMOND);
```

```

5      var aceOfSpade = new PlayingCard(Rank.ACE, Suit.SPADE);
6      var fourOfClub = new PlayingCard(Rank.FOUR, Suit.CLUB);
7      var twoOfHeart = new PlayingCard(Rank.TWO, Suit.HEART);
8      var kingOfSpade = new PlayingCard(Rank.KING, Suit.SPADE);
9
10     var sortedCards = new TreeSet<PlayingCard>();
11     sortedCards.add(threeOfDiamond);
12     sortedCards.add(aceOfSpade);
13     sortedCards.add(fourOfClub);
14     sortedCards.add(twoOfHeart);
15     sortedCards.add(kingOfSpade);
16
17     for (var card : sortedCards) {
18         System.out.println(card);
19     }
20 }
21 }
```

➤ Sortie

```

A♠
2♡
3♢
4♣
K♠
```

Une classe peut implémenter plusieurs interfaces. Outre cette relation qui lie les interfaces aux classes, une seconde relation lie les interfaces entre elles : une interface peut hériter d'autres interfaces.

7.2.2. Une interface peut hériter d'interfaces

L'interface **Comparable<T>** définit un ordre naturel sur les objets de type **T**. Sur base de cet ordre, nous pouvons déduire plusieurs opérations de comparaisons telles que « o1 est plus petit que o2 », « o2 est plus grand ou égal à o1 », etc. Ces opérations peuvent prendre la forme d'une interface **ComparisonOperators<T>**. Pour que ces opérations aient du sens, il paraît souhaitable d'imposer à la classe qui implémente **ComparisonOperators<T>** d'implémenter aussi **Comparable<T>**.

Pour ce faire, Java permet à une interface d'hériter d'une autre interface. Utilisez à cette fin la clause **extends** de l'interface. Comme avec l'héritage de classe, l'interface fille reçoit alors les membres de l'interface mère en héritage. Avec cette solution, Java force toute classe implementant **ComparisonOperators** à implémenter **Comparable<T>**.

Définition d'un héritage d'interface

```

1 public interface ComparisonOperators<T> extends Comparable<T> {
2     boolean lessThan(T other);
3     boolean greaterThan(T other);
4     boolean lessOrEqualThan(T other);
5     boolean greaterOrEqualThan(T other);
6 }
```

Voici notre définition d'une carte à jouer remaniée. Elle ne mentionne pas l'interface Comparable<PlayingCard> dans sa clause implements. Cependant, la méthode compareTo(PlayingCard) est toujours annotée par @Override. Cette méthode doit toujours être définie à cause de la relation d'héritage entre ComparisonOperators<T> et Comparable<T>.

Définition d'une méthode qu'une interface a reçue en héritage

```
1 public final class PlayingCard extends BaseCard implements HasRank, ComparisonOperators
2     <PlayingCard> {
3
4     @Override
5     public int compareTo(PlayingCard that) {
6         Objects.requireNonNull(that);
7         int rankComparison = this.rank.compareTo(that.rank);
8
9         return rankComparison == 0
10            ? this.suit.compareTo(that.suit) : rankComparison;
11     }
12
13     @Override
14     public boolean lessThan(PlayingCard other) {
15         return this.compareTo(other) < 0;
16     }
17
18     @Override
19     public boolean greaterThan(PlayingCard other) {
20         return this.compareTo(other) > 0;
21     }
22
23     @Override
24     public boolean lessOrEqualThan(PlayingCard other) {
25         return this.compareTo(other) <= 0;
26     }
27
28     @Override
29     public boolean greaterOrEqualThan(PlayingCard other) {
30         return this.compareTo(other) >= 0;
31     }
32 // Reste du code inchangé
33 }
```

Les interfaces sont des types au même titre que les classes et les enums. Nous avons jusqu'à présent étudié les relations entre les interfaces et les classes. Intéressons-nous maintenant aux liens qui existent entre les interfaces et les variables.

7.2.3. Les interfaces sont polymorphes

Vous pouvez déclarer des variables dont le type est une interface. Ces variables seront des références à des objets dont la classe implémente l'interface. Vous ne pouvez pas appeler les méthodes spécifiques à la classe concrète de l'objet : vous êtes limités aux membres déclarés par l'interface. En réponse à un appel, Java exécute la version spécifique à la classe concrète de l'objet : encore un cas de polymorphisme par sous-typage.

L'extrait suivant affecte une carte à jouer à plusieurs variables de types différents. Chaque variable donne accès à une liste de méthodes selon son type. Notez que toutes les affectations sont ascendantes. Si nous avions affecté la variable `asObject` à `asComparable`, le compilateur refuserait cet extrait sans transtypage explicite.

Affectation d'un objet à des variables de différents types

```
1 PlayingCard aceOfSpade = new PlayingCard(Rank.ACE, Suit.Spade);
2
3 BaseCard asBaseCard = aceOfSpade;
4 Object asObject = aceOfSpade;
5 HasRank asHasRank = aceOfSpade;
6 Comparable<PlayingCard> asComparable = aceOfSpade;
```

De nouveau, la variable `asObject` limite les méthodes appelables à celles déclarées par la classe `Object`. De même, la variable `asBaseCard` nous limite aux méthodes définies par la classe `BaseCard`. Il en va de même pour les méthodes appelables depuis `asHasSuit` et `asComparable`.

Le caractère polymorphe des interfaces permet d'exposer différentes facettes d'un même objet. L'utilisateur d'un objet ne voit alors que ce dont il a besoin et ne dépend pas de détails inutiles : nous avons fait abstraction de la classe concrète d'un objet pour n'exposer que son interface. Cette propriété ouvre le code utilisateur à tout objet qui implémentera l'interface.

7.2.4. Corps de méthode par défaut

Si vous étudiez plusieurs implémentations des méthodes de l'interface `ComparisonOperators<T>`, vous penserez probablement que ces dernières seront dupliquées : elles font uniquement appel à la méthode `compareTo(T)`. Il serait intéressant d'éviter de dupliquer cette implémentation par défaut.

Avant Java 8, une solution consistait à définir une classe de base commune qui implémente l'interface et d'en faire hériter toutes les autres classes. Cette solution nous expose aux inconvénients de l'héritage de classe : ces classes ne peuvent pas hériter d'une seconde classe.

Depuis Java 8, vous pouvez déclarer un corps par défaut pour les méthodes d'une interface. La JVM utilise un corps par défaut faute d'une implémentation correspondante dans la classe qui implémente l'interface. Pour déclarer un corps par défaut à une méthode, ajoutez le modificateur `default` à son en-tête. L'extrait suivant déclare un corps par défaut pour les méthodes de l'interface `ComparisonOperators<T>`.

Interface déclarant des corps de méthode par défaut

```
1 public interface ComparisonOperators<T> extends Comparable<T> {
2     default boolean lessThan(T other) {
3         return this.compareTo(other) < 0;
4     }
5
6     default boolean greaterThan(T other) {
7         return this.compareTo(other) > 0;
8     }
}
```

```

9
10     default boolean lessOrEqualThan(T other) {
11         return this.compareTo(other) <= 0;
12     }
13
14     default boolean greaterOrEqualThan(T other) {
15         return this.compareTo(other) >= 0;
16     }
17 }
```

Avec ces déclarations par défaut, nous pouvons supprimer celles définies par `PlayingCard`. De plus, toute classe qui implémente l'interface `ComparisonOperators<T>` bénéficiera de ces déclarations si elle le souhaite.

Les corps par défaut sont soumis à une contrainte : ils ne peuvent pas dépendre de champs, autrement dit de détails d'implémentation. En revanche, vous pouvez y déclarer des variables locales.

Un corps par défaut ne peut pas déclarer de champs d'objet

```

1 public interface InRange<T> extends Comparable<T> {
2
3     // private int compareCount; // ✗ Erreur de compilation
4
5     default boolean isInRange(T min, T max) {
6         // ✓ Variables locales à isInRange(T,T)
7         boolean aboveMin = this.compareTo(min) >= 0;
8         boolean belowMax = this.compareTo(max) <= 0;
9         //++compareCount; // ✗ Erreur de compilation
10        return aboveMin && belowMax;
11    }
12 }
```



Grâce aux corps par défaut, les interfaces Java se rapprochent des traits qui sont des constructions proposées par d'autres langages orientés objet comme Scala et Groovy, deux autres langages tournant sur la JVM.

Avec l'arrivée des corps par défaut, les concepts de classe et d'interface en Java sont complémentaires. Par défaut, les méthodes d'une classe sont concrètes, tandis que celles d'une interface sont abstraites. Lorsque vous souhaitez passer outre ces comportements par défaut, vous devez utiliser le modificateur adéquat.

7.2.5. Collisions de méthodes

Supposons qu'une classe `C` implémente deux interfaces `I1` et `I2`. Si `I1` et `I2` déclarent toutes les deux la méthode `m()` de même signature, il y a une collision entre `I1` et `I2` dans `C`.



Collision d'interfaces dans une classe

Des interfaces entrent en collision dans une classe qui les implémente lorsqu'elles déclarent des signatures de méthode identiques.

Les collisions posent un premier problème quand un langage autorise l'héritage multiple de classe. Dans ce cas, une classe peut hériter de plusieurs implémentations distinctes pour la même signature : laquelle choisir en cas d'appel ?

Un second problème posé par l'héritage multiple est celui de l'héritage en diamant^[1]. Dans ce problème, une classe hérite de deux classes qui héritent elles-mêmes d'une quatrième. Si l'ancêtre déclare une méthode abstraite `vm()` définie par ses deux descendants directs, quelle version choisir pour le descendant de ces deux classes ?

Avant Java 8, les problèmes liés aux collisions n'existaient pas. En effet, les corps de méthode n'étaient définis que dans des classes où seul l'héritage simple est autorisé : aucune collision n'est possible. Depuis Java 8 et l'apparition des corps de méthode par défaut, les collisions peuvent apparaître si deux interfaces proposent deux implementations par défaut pour une même signature de méthode : quels corps de méthode par défaut choisir ?

L'approche par défaut de Java est radicale : en cas de collision de méthodes, votre classe doit définir elle-même les méthodes concernées : les corps par défaut sont ignorés. Cependant, vous pouvez choisir explicitement la version exécutée avec l'expression `Interface.super`.

Collision de méthodes et choix du corps par défaut

```
1 interface Printable {
2     default String format() {
3         return "Printable format";
4     }
5 }
6
7 interface Displayable {
8     default String format() {
9         return "Displayable format";
10    }
11 }
12
13 // * Collision : quelle version de format() utiliser ?
14 class Document implements Printable, Displayable {
15     // OBLIGATOIRE : résoudre la collision
16     @Override
17     public String format() {
18         // Option 1 : choisir une version
19         return Printable.super.format();
20
21         // Option 2 : nouvelle implémentation
22         // return "Document format";
23     }
24 }
```

7.3. Vous devriez programmer avec des interfaces

Le tableau suivant compare les concepts de classe abstraite et d'interface.

Table 11. Comparatif Classes abstraites vs Interfaces

| Critère | Classe abstraite | Interface |
|---------------------|---------------------------------------|-------------------------------|
| Héritage | Simple (1 seule) | Multiple |
| Champs d'objet | ✓ Oui | ✗ Non (constantes uniquement) |
| Méthodes abstraites | ✓ Oui | ✓ Oui (par défaut) |
| Méthodes concrètes | ✓ Oui (par défaut) | ✓ Oui (avec default) |
| Constructeur | ✓ Oui | ✗ Non |
| Quand l'utiliser ? | Relation « est un » avec état partagé | Contrat de comportement, rôle |

Ce tableau montre que les interfaces n'ont pas été pensées pour implémenter des objets, mais bien pour spécifier leurs comportements. Les classes abstraites quant à elles restent des implémentations, bien que partielles, d'un type d'objet.

Vous devez considérer les classes abstraites comme des implémentations à compléter. Idéalement, le nom de la classe abstraite devrait mettre en évidence ce caractère partiel. Le nom d'une classe abstraite est préfixé par **Base** ou **Abstract**... Comme notre carte de base que nous pourrions rendre abstraite.

Voici quelques questions à vous poser pour décider de l'outil à utiliser pour représenter vos abstractions. Comme vous le constatez, les interfaces reviennent fréquemment.

Choisir entre les classes abstraites ou les interfaces

Besoin d'état (champs d'objet) ?

- └ Oui → Classe abstraite
- └ Non → Interface

Relation "est un" avec implémentation commune ?

- └ Oui → Interface + Classe abstraite
- └ Non → Interface

Besoin d'héritage multiple ?

- └ Oui → Interface
- └ Non → Interface + Classe abstraite

Grâce aux interfaces, vous définissez les messages qu'un objet peut recevoir sans préciser son implémentation. Idéalement, les utilisateurs d'un objet devraient interagir avec lui à travers une référence à une interface.

L'extrait suivant ne respecte pas ce principe. Non seulement **BlackjackGame** connaît les joueurs et le croupier, mais il est également responsable de leur création. Si je veux augmenter le nombre de joueurs ou jouer avec des joueurs humains, je devrai adapter cette classe. Ce faisant, elle enfreint le principe ouvert-fermé (voir [Section 6.4](#)).

Un exemple de classe avec des dépendances trop concrètes

```
1 // ✗ Dépendances trop fortes
2 public class BlackjackGame {
3     private Dealer dealer = new Dealer("Croupier");
4     private List<ComputerPlayer> players = List.of(
5         new ComputerPlayer("P1 (COM)"),
6         new ComputerPlayer("P2 (COM)"),
7     );
8     private CardsStack deck = CardsStack.create();
9 }
```

Pour résoudre partiellement ce problème, nous pouvons retirer à cette classe la responsabilité de créer ses dépendances. Nous les lui passerons au moment de sa création. Ce faisant, nous appliquons le principe **d'injection des dépendances**.

Les dépendances sont injectés

```
1 // ⚡ Dépendances réduites
2 public class BlackjackGame {
3     private Dealer dealer;
4     private List<ComputerPlayer> players;
5     private CardsStack deck = CardsStack.create();
6
7     public BlackjackGame(Dealer dealer, CardsStack deck, ComputerPlayer...players) {
8         this.dealer = dealer;
9         this.players = Arrays.asList(players);
10        this.deck = deck;
11    }
12 }
```

L'injection des dépendances consiste à fournir à un objet ses dépendances plutôt que de lui laisser la responsabilité de les créer. Elles seront le plus souvent transmises comme argument d'un constructeur. Plus rarement, des mutateurs seront aussi prévus.

Injection des dépendances



Méthode de création des dépendances d'un objet dans laquelle ce dernier reçoit ses dépendances, le plus souvent au moment de sa construction. L'objet n'est plus responsable de créer les objets dont il dépend.

La classe a gagné en flexibilité : je peux jouer avec un nombre quelconque de joueurs simulés par l'ordinateur. En revanche, je ne peux pas jouer avec des acteurs humains. Pour arriver à une solution convenable, **BlackjackGame** doit dépendre de types abstraits.

Les dépendances sont inversés

```
1 // ✓ Dépendances à des abstractions
2 public class BlackjackGame {
3     private BlackjackActor dealer;
4     private List<BlackjackActor> players;
5     private CardsStack deck = CardsStack.create();
6 }
```

```
7     public BlackjackGame(BlackjackActor dealer, Lifo<PlayCard> deck, BlackjackActor
8         ...players) {
9         this.dealer = dealer;
10        this.players = Arrays.asList(players);
11    }
12 }
```

Lorsqu'une classe dépend uniquement d'abstractions telles que les interfaces, elle se conforme à un principe appelé **principe d'inversion des dépendances**.

Principe d'inversion des dépendances



Le principe d'inversion des dépendances (*Dependency Inversion Principle - DIP*) stipule qu'une classe ne devrait dépendre que d'abstractions. De même, toute abstraction ne devrait dépendre que d'abstractions.

Ce principe est notamment utile dans la rédaction de tests unitaires (voir [\[test-driven-development\]](#)). En effet, grâce à lui, vous pouvez simuler les dépendances de l'objet testé avec des objets factices. Vos tests valident alors une portion limitée et clairement identifiée de code.

Quand une classe ne dépend que d'abstractions, elle tend à être adaptable et réutilisable. Comme avec les Lego, vous pouvez commencer à combiner les objets de plusieurs façons selon vos besoins.

7.4. En résumé

À force de construire des abstractions décrivant des concepts de plus en plus généraux, nous nous trouvons dans la situation où nous prévoyons l'existence de méthodes que nous sommes incapables de coder, car le concept décrit est devenu trop vague, trop abstrait. Java propose deux mécanismes pour nous aider dans ce cas : les classes abstraites et les interfaces.

Une classe abstraite est une classe que le programmeur ne peut pas instancier, parce qu'elle est susceptible de contenir des méthodes abstraites, sans corps à exécuter. Une classe peut hériter d'une classe abstraite. Elle doit alors définir un corps pour toutes les méthodes abstraites héritées afin de devenir concrète. Les classes abstraites souffrent des mêmes problèmes que l'héritage de classe : le descendant est fortement couplé à ses ancêtres et un descendant ne peut avoir qu'un parent direct.

Les interfaces résolvent ces problèmes. Une interface Java est une liste de méthodes abstraites, sans état. Comme il n'y a pas d'état, Java autorise l'héritage multiple pour les interfaces : une classe peut implémenter plusieurs interfaces et une interface peut hériter de plusieurs interfaces. Contrairement aux classes, une interface ne peut pas déclarer de champs d'objet.

Depuis Java 8, les interfaces peuvent déclarer des corps de méthode par défaut. Ces derniers ne peuvent cependant pas s'appuyer sur des champs. En cas de collision entre deux corps par défaut, le programmeur de la classe doit proposer sa propre implémentation.

Pour obtenir des systèmes souples et adaptables, vos classes et vos interfaces doivent dépendre

d'abstractions. C'est le principe d'inversion des dépendances. Pour que ce principe fonctionne, un objet ne peut plus créer ses dépendances, il doit les recevoir, le plus souvent au moment de sa construction. Nous parlerons d'injection des dépendances.

Les interfaces forment le socle sur lequel une bonne conception s'appuie. Dans le chapitre suivant, nous étudions une méthode de conception adaptée à la POO. Celle-ci devrait vous permettre d'imaginer la structure de vos applications à l'aide d'objets s'échangeant des messages.

[1] https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem