Prérequis sur la programmation orientée objet illustrée avec JAVA

Christian Bac et Denis Conan

CSC4102

Télécom SudParis Janvier 2024

Grille d'auto-évaluation des prérequis sur le langage JAVA

Voici la grille d'auto-évaluation de compétences *a priori* acquises dans les modules CSC3101 et PRO3600 sur la programmation en JAVA. Si certaines notions de la grille ne vous semble pas acquises, prenez connaissance des pages qui suivent. Les pages qui suivent ne présentent pas les derniers concepts de la grille (à partir de « méthode polymorphique ») car nous revenons dessus dans le cours à venir.

Notions / concept de programmation	**0**	**1**	**2**	**3**	**4**	**5**
Machine virtuelle	X	X	X	X	X	X
JAVA *Standard Development Kit* : `javac`, JAVA, etc.	X	X	X	X	X	X
if, switch	X	X	X	X	X	X
for, while, do while	X	X	X	X	X	X
Types primitifs : short, int, long, float, double, boolean, byte	X	X	X	X	X	X
Attribut (déclaration, définition), d'instance *Versus* de classe	X	X	X	X	X	X
final	X	X	X	X	X	X
static	X	X	X	X	X	X
Classe (abstraction, encapsulation, class)	X	X	X	X	X	X
Constructeur	X	X	X	X	X	X
this.	X	X	X	X	X	X
this()	X	X	X	X	X	X
Objet / instance	X	X	X	X	X	X
Référence, passage d'arguments dans les méthodes	X	X	X	X	X	X
new	X	X	X	X	X	X
Ramasse-miettes	X	X	X	X	X	X
Tableau ([], new)	X	X	X	X	X	X
Méthode (prototype, définition, paramètre, argument)	X	X	X	X	X	X
void	X	X	X	X	X	X
main	X	X	X	X	X	X
package, chemin de recherche	X	X	X	X	X	X
Surcharge (en anglais *overloading*)	X	X	X	X	X	X
Héritage (classe de base, classe dérivée)	X	X	X	X	X	X
extends	X	X	X	X	X	X
protected	X	X	X	X		
Visibilité (public, private, protected, *package friendly*)	X	X	X	X		
Méthode polymorphique	X	X	X			
Redéfinition (en anglais *overriding*)	X	X	X	X		
super.	X	X	X	X		
super()	X	X	X	X		
Transtypage (en anglais *cast*)	X	X	X			
Upcast	X	X	X			
Downcast	X	X	X			
Liaison dynamique / tardive	X	X	X			
Classe abstraite	X	X	X			
Méthode abstraite	X	X	X			
Interface, implements	X	X	X			
Égalité (de références, d'objet), equals	X	X	X			

Table 1 : Grille d'auto-évaluation des prérequis sur la programmation JAVA

2

Sommaire

3

1 Introduction

4

1.1 JAVA un langage orienté objet

- Tout est classe sauf les types primitifs (`int`, `float`, `double`, etc.)
- Tout objet (de classe) est manipulé à travers une référence
- La généralisation spécialisation, appelée héritage dans les langages de programmation, est simple entre les classes
- Toutes les classes dérivent/héritent de `java.lang.Object`
- L'API (en anglais, *Application Programming Interface*) est un ensemble de classes

JAVA est un langage orienté objet. La notion de classe est centrale dans ce langage et tout le code est contenu dans des classes. Seules les variables de type primitif ne sont pas des objets. Les types primitifs servent à créer des variables locales dans les méthodes et des attributs pour les classes. Les objets ne sont pas manipulés directement mais à travers des références (pointeurs dé-référencés de manière automatique).

Comme tous les langages orientés objet, JAVA supporte la généralisation spécialisation, le plus communément appélée « héritage » dans les langages de programmation. Pour simplifier la mise en œuvre, JAVA ne permet que l'héritage simple, c'est-à-dire qu'une classe dérive d'une classe et d'une seule. En outre, JAVA introduit le concept d'interface ; une classe peut « implémenter » plusieurs interfaces. Nous présentons les concepts d'héritage et d'interface dans les pages qui suivent et y reviendrons dans le cours.

Toutes les classes dérivent d'une classe racine appelée `java.lang.Object`. Cette classe définit des comportements stéréotypés dont nous reparlerons dans le cours.

1.2 Machine Virtuelle JAVA

5

- Les compilateurs génèrent du code intermédiaire (en anglais, *bytecode*)
- Ce code intermédiaire est interprété par une « *JAVA Virtual Machine* » (JVM)
- Modèle « *compile once execute everywhere* »
- Les *JAVA Virtual Machines* (JVM) :
  - s'exécutent sur les systèmes d'exploitation (par exemple avec la commande `java`)
  - ou sont intégrées dans les navigateurs Web
- Taille et domaine de valeur des types primitifs identiques sur toutes les plate-formes
- Code source Unicode (accents et autres glyphes)
- Bibliothèques standards riches

La portabilité du code JAVA a toujours été un objectif pour ses concepteurs. Cette portabilité n'a pas toujours été parfaite mais elle est bien plus grande que celle des langages des générations précédentes.

Un compilateur JAVA génère du code intermédiaire appelé en anglais *bytecode*. Ce code intermédiaire est contenu dans un fichier dont le suffixe est `.class`. Un fichier `.class` peut être chargé par une machine virtuelle JAVA et interprété par celle-ci. Le code intermédiaire est totalement indépendant de la machine sur laquelle il a été généré. Il est aussi indépendant du compilateur qui l'a produit. Ce code permet de recréer le fichier source facilement. Le modèle de compilation est appelé en anglais « *compile once, execute everywhere* ».

Les tailles et les domaines de valeur des types primitifs sont identiques sur toutes les plates-formes. Ainsi, quelle que soit l'architecture matérielle sur laquelle le programme s'exécute, un entier utilisé par un programme JAVA possède une taille identique. Le principe du *bytecode* date des années 1980 avec son introduction dans certains compilateurs du langage Pascal. Il se retrouve aussi aujourd'hui dans l'ensemble de la chaîne de compilation de Microsoft.

Les JVM s'exécutent directement sur les systèmes d'exploitation. C'est par exemple la commande `java` sous Linux. Elles peuvent aussi être embarquées dans les navigateurs Web.

Le code source des classes peut être écrit en caractères Unicode. Cependant, la collaboration entre développeurs préconise l'utilisation de variables dont le nom est compréhensible par le plus grand nombre.

1.3 *JAVA Standard Development Kit*

6

- `javac` : compilateur
- `java` : JVM interpréteur de *bytecode*
- `javadoc` : générateur de documentation
- `javah` : générateur d'entêtes pour mélange avec code natif en `C` (`JNI`)
- `javap` : désassembleur de code intermédiaire pour obtenir du code JAVA
- `jdb` : dévermineur
- `javakey` : générateur de clés pour signer le code

Le JDK permet de matérialiser les différentes parties de la chaîne de production utilisée dans le développement d'un programme JAVA. Ainsi, il est nécessaire de traduire les fichiers contenant du langage JAVA en des fichiers contenant du *bytecode*. C'est le rôle du compilateur (`javac`). La commande `java` démarre la machine virtuelle dans laquelle le *bytecode* peut s'exécuter. La commande `javadoc` permet d'extraire la documentation du code pour réaliser des pages semblables à celles de la documentation des bibliothèques.

2 Tableaux

7

- Les tableaux sont des objets particuliers
- Leur taille est fixe
- Déclaration de la référence

```
int[] arrayOfInt; // ou  int arrayOfInt[];
```

- Création avec association de taille

```
arrayOfInt = new int[42];
```

- Taille

```
int t = arrayOfInt.length;
```

- Erreur d'accès en cas de dépassement de taille

Les tableaux sont des objets gérés de manière particulière par l'infrastructure du langage. Les variables de type tableau sont déclarées comme des références et ne sont pas associées à l'espace mémoire correspondant. La création d'un tableau est réalisée par l'appel du mot réservé `new`. C'est à ce moment que la taille du tableau est fixée et que la référence est associée à l'objet tableau. La taille d'un tableau est connue en accédant à l'attribut en lecture seule `length`. La syntaxe du langage JAVA pour accéder à un attribut ou une méthode associée à un objet correspond à celle du langage C pour accéder à un membre d'une structure : le point (« . ») permet de passer de la référence de l'objet à ses attributs ou méthodes.

Les tableaux sont traités de manière à empêcher les erreurs d'accès relativement au nombre d'entrées qu'ils contiennent.

Classe prerequis.tableaux.ExempleTableau

```
1 package eu.telecomsudparis.csc4102.prerequis.tableaux;
2 public class ExempleTableau {
3     public static void main(String[] args) {
4         int[] arrayOfInt; // or  int arrayOfInt[]; // declaration de la variable arrayOfInt
5         arrayOfInt = new int[42]; // creation du tableau et association a arrayOfInt
6         arrayOfInt[0] = 3; // affectation d'un element du tableau
7         System.out.println("Array length "+ arrayOfInt.length);// obtention de la taille du tableau (42)
8         System.out.println(arrayOfInt[42]);      // impossible levee d'un exception
9     }
10 }
```

Résultat de l'exécution :

```
Array length 42
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 42
 at ArrayExample.main(ExempleTableau.java:8)
```

Depuis la version 5 du langage, JAVA fournit une nouvelle construction, appelée « *for each* » ou « *enhanced for* », pour parcourir les éléments d'une collection, ici un tableau. Nous conseillons d'utiliser cette dernière forme. Voici un exemple incluant les deux formes, afin que vous les compariez.

Classe prerequis.tableaux.ExempledeParcoursDeTableau

```
1 package eu.telecomsudparis.csc4102.prerequis.tableaux;
2 public class ExempledeParcoursDeTableau {
3
4     public static void main(String[] args) {
5         // tableau
6         int i=0;
7         System.out.println("Parcours arguments de main par for avec indice");
8         for (i=0; i < args.length; i++) { System.out.println(args[i]); }
9         System.out.println("Parcours arguments de main par for type foreach");
10        for (String string : args) { System.out.println(string); }
11    }
12 }
```

8

3 Passage d'arguments dans les méthodes

- Par valeur : du type primitif ou de la référence (pour les objets et les tableaux)

Classe `prerequis.passageParametres.ExemplePassageParametres`

```
1 package eu.telecomsudparis.csc4102.prerequis.passageparametres;
2 public class ExemplePassageParametres {
3     private static void add(int c, final int[] as) {
4         c++;
5         as[0] ++;
6         System.out.println("add : c=" + c + ", as[0]=" + as[0]);
7     }
8     public static void main(final String argv[]) {
9         int i = 0;
10        int[] s = new int[10];
11        s[0] = 0;
12        add(i, s);
13        System.out.println("main : i=" + i + ", s[0]=" + s[0]);
14    }
15 }
```

Résultat de l'exécution :

```
add : c=1, as[0]=1
main : i=0, s[0]=1
```

Les variables de type primitif sont passées par copie de la valeur. Les variables de type tableau ou objet sont manipulées en JAVA à travers une référence. Lors du passage d'une référence à une méthode, cette référence permet de manipuler l'objet d'origine. Ainsi, toute modification réalisée à travers une référence dans une méthode appelée modifie l'objet qui est référencé de manière visible par la méthode appelante.

Dans l'exemple, la modification de la copie `c` de la variable `i` dans la méthode `add` n'a pas de répercussion sur la valeur de la variable `i`. La variable `c` est donc bien une variable différente de la variable `i`. Cette variable a été initialisée avec la valeur de la variable `i` lors de l'appel. Les variables `s` et `as` (`as` étant une copie de `s`) font référence au même tableau. Ainsi, lorsque l'entrée 0 du tableau est modifiée dans la méthode `add` en utilisant la variable `as`, l'entrée correspondante vue à travers la variable `s` est modifiée de manière identique.

9

4 Argument variadique

- Objectif = éliminer la nécessité de regrouper manuellement les listes d'arguments dans un tableau lors de l'invocation de méthodes acceptant des listes d'arguments de longueur variable

Classe `seance5.argumentvariadique.ExempleVarArg`

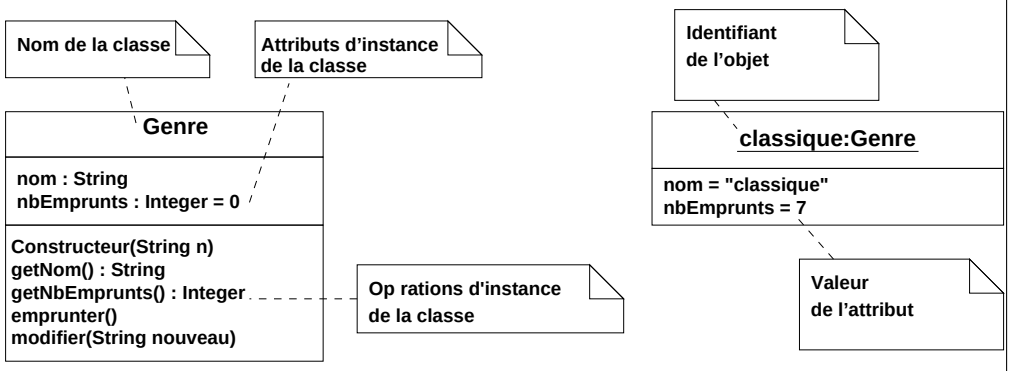
```
1   private static String format(final String... strings) {
2     Objects.requireNonNull(strings);
3     var result = new StringBuffer();
4     for (String string : strings) { result.append(" ").append(string); }
5     return result.toString(); }
```

10

5 Classes et objets en JAVA

11

5.1 Classe, attribut, méthode



Ce schéma décrit la notation UML d'une classe `Genre` de l'étude de cas exemple « Médiathèque ». La partie gauche contient la description des attributs et des opérations. C'est ce schéma que nous traduisons maintenant en JAVA.

Classe `prerequis.mediathequesimplifiee.classeobjet.Genre`

```
1 package eu.telecomsudparis.csc4102.prerequis.mediathequesimplifiee.classeobjet;
2
3 public final class Genre {
4         private String nom;
5         private int nbEmprunts;
6         public Genre(final String n) {
7                 nom = n;
8                 nbEmprunts = 0;
9         }
```

```
10          public String getNom() {
11                  return nom;
12          }
13          public int getNbEmprunts() {
14                  return nbEmprunts;
15          }
16          public void emprunter() {
17                  nbEmprunts++;
18          }
19          public void modifier(final String nouveau) {
20                  nom = nouveau;
21          }
22  }
```

La première ligne indique que la classe appartient au paquetage (mot réservé `package`) `prerequis.mediathequesimplifiee.classeobjet`. Ensuite, la description de classe commence par une ligne contenant le mot réservé `class`. Lorsque la classe est publique, le fichier qui contient celle-ci doit porter un nom identique à la classe : dans notre cas, le fichier doit s'appeler `Genre.java`. De manière traditionnelle, une classe commence par la description des attributs. Comme nous l'avons vu en UML, les attributs peuvent être spécifiques à chaque objet. Ce sont des attributs d'instance. Ils peuvent aussi être partagés entre tous les objets de la classe. Ce sont des attributs de classe ; en JAVA, ces attributs sont qualifiés de `static` (cf. plus loin dans cette présentation). Par ailleurs, les attributs peuvent correspondre à des types primitifs ou non. Dans notre cas, la variable `nbEmprunts` est du type primitif `int`. La variable `nom` est une référence sur un objet de la classe `java.lang.String`. Les attributs d'instance sont le plus souvent initialisés par le constructeur (méthode qui porte le même nom que la classe). Enfin, une classe définit les comportements de ses objets à travers des méthodes d'instance. Une méthode d'instance réalise une opération s'appliquant sur les attributs d'un objet. C'est le cas dans notre exemple des méthodes : `getNom`, `getNbEmprunts`, `emprunter` et `modifier`.

La classe suivante possède une méthode `main` qui montre la création et la manipulation d'une instance de la classe `Genre`.

Classe `prerequis.mediathequesimplifiee.classeobjet.ExempleInstanciationGenre`

```
1 package eu.telecomsudparis.csc4102.prerequis.mediathequesimplifiee.classeobjet;
2
3 public class ExempleInstanciationGenre {
4           public static void main(final String argv[]) {
5                   Genre g;
6                   g = new Genre("classique");
7                   System.out.println(g);
8           }
9 }
```

Résultat de l'exécution :

```
prerequis.mediathequesimplifiee.classeobjet.Genre@15db9742
```

Cet exemple montre la création d'un objet de la classe `Genre`. La première ligne du fichier contient une directive `import` qui permet de nommer plus facilement la classe `Genre`. Ce code est contenu dans une classe appelée `ExempleInstanciationGenre`. Cette classe est dotée d'une méthode de classe (`static`) publique appelée `main` et recevant comme argument un tableau de chaînes de caractères. Elle sert de point d'entrée à l'exécution du programme. À la ligne 7 est définie une variable locale appelée `g` qui permet de référencer un objet de la classe `Genre`. L'instance est créée à la ligne 8 par l'appel du mot réservé `new` suivi d'un appel au constructeur de la classe `Genre`. L'instance est ensuite manipulée à partir de la référence `g` comme dans l'appel de la méthode `println`, ligne 7.

Dans le prototype de la méthode `main`, le mot-clé `final` devant la déclaration du paramètre `argv` indique que, dans le corps de la méthode, `argv` ne peut pas changer de valeur. C'est une forme de programmation défensive : le principe d'un paramètre est de fournir des données entrée d'une méthode ; il n'y a pas de raison que sa valeur change ; pour plus de sûreté, nous ajoutons le mot-clé `final` pour que le compilateur vérifie cela. C'est une bonne pratique que nous conseillons et sur laquelle nous reviendrons en cours.

Non utilisé dans cet exemple, la spécification de la propriété « {readOnly} » attachée à un attribut dans le diagramme de classes ou la fiche d'une classe se traduit par la qualification `final` en JAVA. Par exemple, pour rendre l'attribut `nom` non modifiable après sa première affectation, la déclaration devient comme suit :

```
private final String nom;
```

Dans ce cas, il faut bien sûr aussi retirer la méthode `modifier`.

5.2 Constructeurs

- Début de la même classe `Genre` avec deux constructeurs

Classe `prerequis.mediathequesimplifiee.constructeur.Genre`

```
1  package eu.telecomsudparis.csc4102.prerequis.mediathequesimplifiee.constructeur;
2  public final class Genre {
3          private String nom;
4          private int nbEmprunts;
5          public Genre(final String nom, final int nbEmprunts) {
6                  this.nom = nom;
7                  this.nbEmprunts = nbEmprunts;
8          }
9          public Genre(final String nom) {
10                 this(nom, 0);
11         }
```

12

- `this` permet de référencer l'instance courante
- `this` est associé au concept d'auto-référence de l'objet
- `this` permet de décrire un attribut ou une méthode de l'instance courante sans ambiguïté
- `this(...)` en tant que méthode fait appel à un constructeur

Les constructeurs sont des méthodes particulières. Ils portent le même nom que la classe. Ils ne peuvent pas être appelés par un appel de méthode classique. De l'extérieur de la classe, ils sont appelés en utilisant le mot réservé `new`. De l'intérieur de la classe, les constructeurs peuvent s'appeler directement en utilisant `this`.

Les constructeurs n'ont pas de type de retour dans leur déclaration/prototype.

Le langage JAVA permet d'avoir plusieurs méthodes portant le même nom dans une classe à la condition que ces méthodes puissent être distinguées à l'aide de leurs paramètres. Cette propriété s'appelle la surcharge (en anglais *overloading*, aussi appelée le polymorphisme *ad hoc* [cf. le glossaire et le « pour aller plus loin » de la séance 3], et à ne pas confondre avec redéfinition [en anglais *overriding*]).

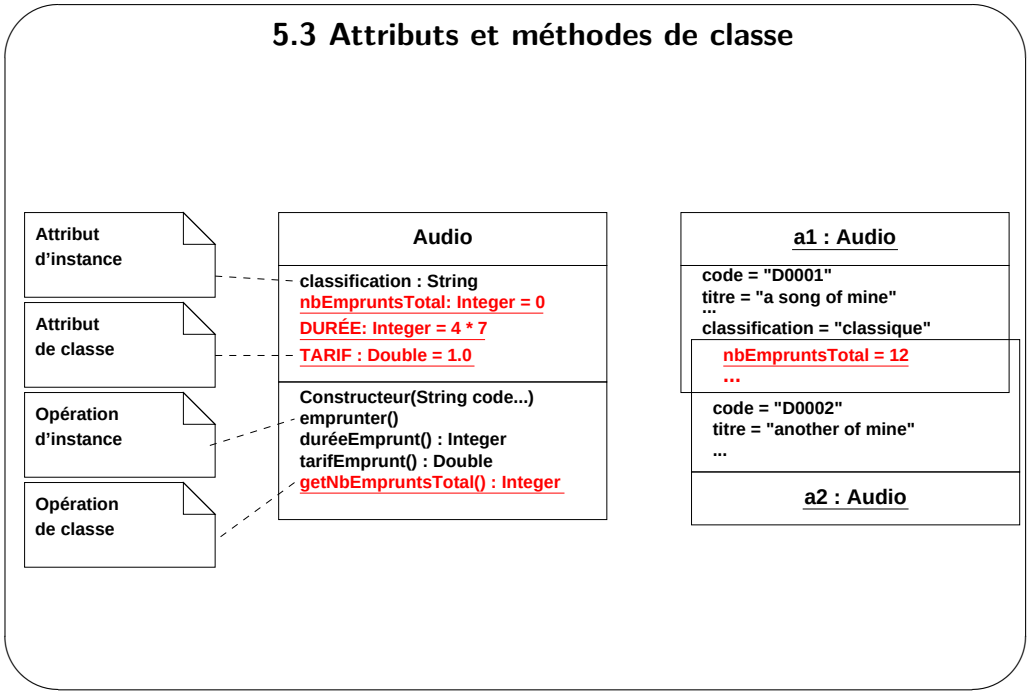
La propriété de surcharge peut être utilisée afin de proposer plusieurs façons de construire un objet d'une classe donnée. Ainsi, nous pouvons avoir plusieurs constructeurs avec des listes de paramètres différentes.

La classe de cette exemple contient deux constructeurs. Le premier constructeur initialise les attributs `nom` et `nbEmprunts` avec les arguments correspondants. Le second initialise l'attribut `nom` et laisse l'attribut `nbEmprunts` à la valeur par défaut 0.

L'utilisation de `this` aux ligne 6 et 7 permet de distinguer les attributs de l'objet courant `this.nom` et `this.nbEmprunts`, des paramètres `nom` et `nbEmprunts` du constructeur. Cette pratique est courante car les programmeurs utilisent souvent le même nom pour l'attribut et pour le paramètre qui permet de l'initialiser.

À la ligne 11, l'instruction `this(nom, 0)` correspond à l'appel du constructeur de la ligne 5. Cette pratique est courante afin de factoriser le code.

13



Ce schéma montre une classe qui contient des attributs et des opérations soulignés. Ces attributs et opérations sont partagés par l'ensemble des membres d'une classe. Ils sont appelés attributs ou méthodes de classes alors que les attributs et méthodes qui ne sont pas soulignés sont appelés attributs ou méthodes d'instance. En JAVA, le mot réservé `static` permet de qualifier les attributs ou méthodes de classe. Les méthodes de classe ne peuvent accéder qu'aux attributs de classe. Les instances peuvent accéder aux attributs de classe et peuvent invoquer les méthodes de classe. En outre, il n'est pas nécessaire d'instancier la classe pour accéder à ses membres « statiques ».

La traduction du schéma UML est donnée dans le code ci-dessous. Lorsque la visibilité qui leur est associée le permet, ils peuvent aussi être accédés directement en utilisant le nom de la classe suivi d'un point et du nom de l'attribut ou de la méthode. La classe `Audio` définit une méthode `toString` qui permet d'obtenir la représentation en chaîne de caractères d'un objet. Il est cependant nécessaire de comprendre que l'appel à la méthode `System.out.println` associée à un objet, fait appel à la méthode `toString()` de l'objet et affiche la chaîne de caractères ainsi produite. Par ailleurs, les méthodes de classe ne sont bien sûr jamais associées à l'auto-référence de l'objet (`this`).

Classe `prerequis.mediathequesimplifiee.attributsoperationsdeclasse.Audio`

```
1  package eu.telecomsudparis.csc4102.prerequis.mediathequesimplifiee.attributsoperationsdeclasse;
2
3  public final class Audio {
4          private String classification;
5          public static final int DUREE = 4 * 7;
6          public static final double TARIF = 1.0;
7          private static int nbEmpruntsTotal = 0;
8          public Audio(final String classif) {
9                  this.classification = classif;
10         }
11         public static int getNbEmpruntsTotal() {
12                 return nbEmpruntsTotal;
13         }
14         public String getClassification() {
15                 return classification;
16         }
17         public void emprunter() {
18                 nbEmpruntsTotal++;
19         }
20         @Override
21         public String toString() {
22                 return "Audio [classification=" + classification + ", nbEmpruntsTotal="
23                                 + nbEmpruntsTotal + "]";
24         }
25 }
```

Voici maintenant une utilisation :

Classe `prerequis.mediathequesimplifiee.attributsoperationsdeclasse.ExempleAttributsOperationDeClasse`

```java
package eu.telecomsudparis.csc4102.prerequis.mediathequesimplifiee.attributsoperationsdeclasse;

public class ExempleAttributsOperationDeClasse {

	public static void main(String[] args) {
		Audio a1 = new Audio("opera");
		Audio a2 = new Audio("hardcore");
		a1.emprunter();
		a2.emprunter();
		System.out.println(a1);
		System.out.println(a2);
	}

}
```

L'affichage d'une exécution montre que le nombre total de participations est identique dans les deux objets.

```
Audio [classification=opéra, nbEmpruntsTotal=2]
Audio [classification=hardcore, nbEmpruntsTotal=2]
```

Dans les lignes de l'exemple qui suit, la classe `java.lang.Math` est un bon exemple de l'usage qui peut être fait des attributs et des méthodes de classe. Il n'est pas nécessaire d'instancier la classe pour accéder à ses membres « statiques ». Comme son nom l'indique, la classe `Math` rassemble les constantes et les méthodes mathématiques les plus utilisées.

```
pi = Math.PI;      // static attribute 'PI'
b = Math.sqrt(2.0);// method call of static 'sqrt'
```

5.3.1 Destruction des objets

14

- Pas de technique particulière pour détruire un objet
- Objet détruit lorsqu'il n'est plus référencé
- Utilisation d'un ramasse miettes (en anglais, *garbage collector*)
- Le ramasse miettes détruit les objets non référencés
  - La destruction est asynchrone : on ne sait pas quand la destruction est effectuée
  - Il n'y a pas de garantie sur la destruction (le programme peut se terminer avant que l'objet ne soit détruit)
  - Si la classe décrit une méthode appelée `finalize`, cette méthode est appelée avant la libération de la mémoire de l'objet
    - Mais la spécification du langage ne garantit pas qu'un *finalizer* soit effectivement appelé (avant la fin ou à la fin de l'exécution)

Le langage JAVA a été pensé pour pallier les principales difficultés du langage `C++`, et en particulier, les erreurs associées à la gestion des références vers les objets. Il est en effet difficile de garantir qu'un objet est désalloué lorsqu'il n'est plus utilisé.

La machine virtuelle JAVA prend à sa charge la gestion des allocations et des libérations de la mémoire en tenant à jour la liste des références vers les blocs alloués. Pour ce faire, la machine virtuelle met en œuvre un ramasse miettes pour vérifier périodiquement que les objets alloués sont toujours référencés.

Lorsque les objets ne sont plus référencés, le ramasse miettes récupère l'espace mémoire qu'ils occupent. Puis, il met cet espace mémoire à la disposition de la création de nouveaux objets.

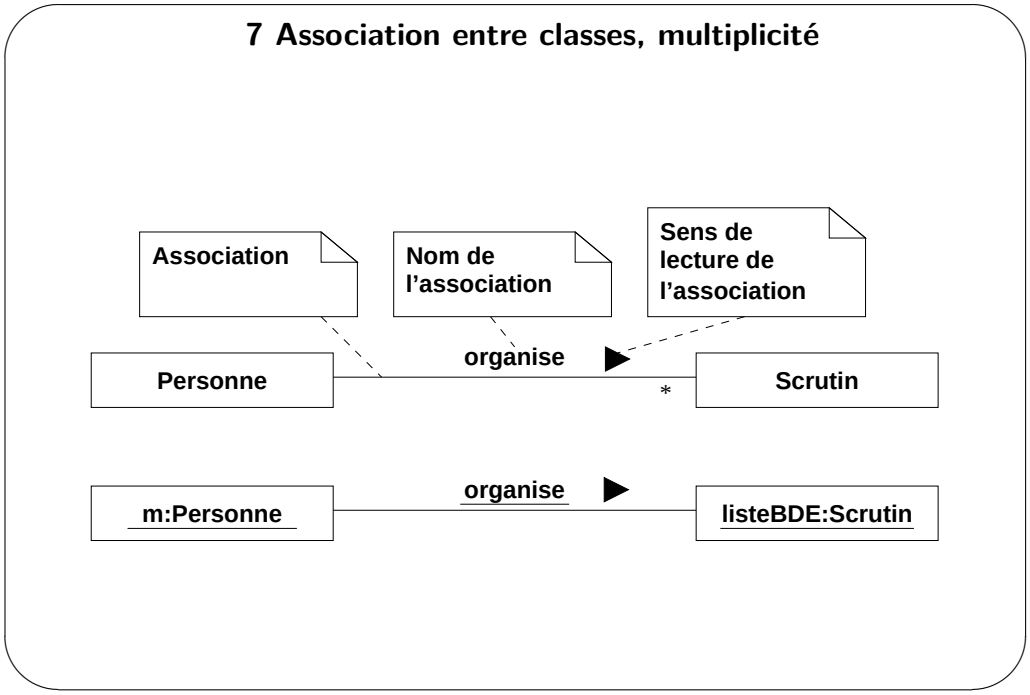
Ce ramasse miettes fonctionne de manière périodique et asynchrone à l'intérieur d'un fil d'exécution séparé de la JVM. Il n'y a pas de garantie qu'un objet soit détruit. Il est possible de déclencher le fonctionnement du ramasse miettes par l'appel à la méthode de classe `System.gc()`.

6 Annotation

15

- Annotation = sucre syntaxique associant des méta-données à un élément
  - Nom introduit par « `@` » avec possiblement des arguments (`nom=valeur`)
  - Annotations standards
    - `@Override` : utilisée par le compilateur pour vérifier l'invariance des prototypes lors de la redéfinition de méthode
    - `@Deprecated` : utilisée par `JavaDoc` pour déconseiller l'utilisation de la méthode
    - `@SuppressWarnings` : utilisée par le compilateur pour ne pas générer les avertissements concernant l'élément annoté
  - D'autres annotations par exemple pour la programmation des tests avec JUnit
    - Cf. cours de la séance 6

16



L'extrait de diagramme de classes montre une association entre des classes Personne et Scrutin. Cette association matérialise le fait qu'un participant peut organiser un ou plusieurs scrutins dans une application de vote électronique. L'association est bidirectionnelle. Elle matérialise le fait qu'un scrutin est organisé par un participant. La multiplicité indique qu'il n'y a qu'un participant.

Un exemple d'instanciation de ce diagramme de classes donne le diagramme d'objets en dessous. Dans ce diagramme d'objets, l'objet référencé par `m` de la classe Personne organise le scrutin référencé par listeBDE.

Pour modéliser la relation avec les scrutins organisés, nous ajoutons à la classe Personne un tableau de références vers des objets de la classe Scrutin. Le nombre d'entrées valides dans le tableau est contenu dans la variable `nborganisations`. La taille du tableau est initialisée à 10 dans le constructeur à la ligne 8 de la classe Personne. Les éléments du tableau sont affectés lorsque le participant organise un nouveau scrutin.

C'est la méthode organiserScrutin qui doit créer l'objet de type Scrutin. Pour que cet objet puisse initialiser la relation dans le sens opposé, il doit recevoir la référence sur l'objet qui a fait appel à cette méthode. C'est le rôle du second paramètre de la méthode organiserScrutin : à la ligne 11, c'est la référence de l'objet appelant (l'auto-référence `this`) qui est fournie lors de l'appel.

Classe `associationmultiplicite.Personne`

```
1  package eu.telecomsudparis.csc4102.prerequis.associationmultiplicite;
2  public class Personne {
3      private String nom;
4      private String prenom;
5      private int nbParticipations = 0;
6      private int nbOrganisations = 0;
7      private Scrutin[] scrutinsOrganises ;
8      public Personne(final String nom, final String prenom) {
9          this.nom = nom; this.prenom = prenom;
10         scrutinsOrganises = new Scrutin[10];
11     }
12     public Scrutin organiserScrutin(final String nom){
13         Scrutin s = new Scrutin(nom, this);
14         scrutinsOrganises[nbOrganisations] = s;
15         nbOrganisations ++;
16         return s;
17     }
18     @Override
19         public String toString() {
20                 return "Personne [nom=" + nom + ", prenom=" + prenom
21                                 + ", nbParticipations=" + nbParticipations
22                                 + ", nbOrganisations=" + nbOrganisations + "]";
23         }
24 }
```

Dans la classe `Scrutin`, le constructeur reçoit en argument la référence vers l'organisateur du scrutin et la mémorise dans l'attribut `organisateur` à la ligne 7.

Classe `prerequis.associationmultiplicite.Scrutin`

```java
package eu.telecomsudparis.csc4102.prerequis.associationmultiplicite;
public class Scrutin{
    private String nomScrutin;
    private Personne organisateur;
    public Scrutin(final String nom, final Personne personne) {
        nomScrutin = nom;
        organisateur = personne;
    }
    @Override
        public String toString() {
                return "Scrutin [nomScrutin=" + nomScrutin + ", organisateur="
                                + organisateur + "]";
        }
}
```

La classe qui suit montre un exemple d'utilisation des deux classes.

Classe `prerequis.associationmultiplicite.ExempleAssociationMultiplicite`

```java
package eu.telecomsudparis.csc4102.prerequis.associationmultiplicite;

public class ExempleAssociationMultiplicite {
  public static void main(final String[] args) {
        Personne p;                              // reference
        p = new Personne("Dupont", "Julien");  // instance creation
        Scrutin bde = p.organiserScrutin("Election bde 2010");
        System.out.println(bde);
  }
}
```

Une difficulté apparaît lorsque l'association est bidirectionnelle : par quoi commencer ? Comme dans cet exemple, nous préconisons l'utilisation d'une méthode qui permet d'affecter un attribut d'une classe en dehors du constructeur et le passage de la référence d'un des objets dans l'appel au constructeur de l'autre.

17

8 Généralisation spécialisation / Héritage

- Une classe ne peut hériter que d'une autre classe (héritage simple)
- Une classe hérite d'une autre par l'utilisation du mot réservé `extends`
- Une classe pour laquelle aucune spécialisation n'est explicitée spécialise implicitement la classe `java.lang.Object`
- L'opérateur `instanceof` permet de tester si une référence correspond à un objet d'une classe donnée
- Le mot réservé `final` utilisé devant le mot clé `class` interdit toute spécialisation de la classe sur laquelle il est utilisé

9 Visibilité des attributs et des opérations

18

```
package p1;
class C1 {
  public int a;
  protected int b;
  int c; // package
  private int d;
}
class C2 extends C1 {
    ...
}
class C3 {
    ...
}
```

```
package p2;
class C4 extends C1 {
   ...
}
class C5 {
   ...
}
```

- `private` = seule la classe courante
- *package* = `private` + classes du paquetage
  - *package* par défaut
- `protected` = *package* + classes enfants item
  `public` = toutes les classes

	a	b	c	d
Accessible de C2	oui	oui	oui	–
Accessible de C3	oui	oui	oui	–
Accessible de C4	oui	oui	–	–
Accessible de C5	oui	–	–	–

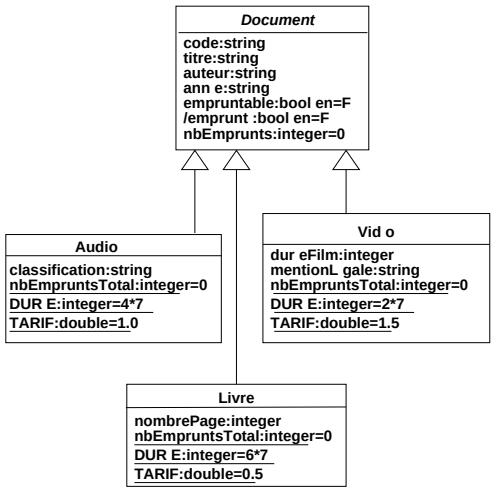
9.1 Héritage et constructeur

19

- Création d'un objet de classe dérivée $\Longrightarrow$ création de la partie de l'objet correspondant à la classe parente
- Appel dans le constructeur de la classe dérivée d'un des constructeurs de la classe parente par utilisation du mot réservé `super()`
  - En première ligne du constructeur de la classe enfant
  - Si aucun appel à `super()` alors appel au constructeur sans argument de la classe parente

Un objet d'une classe dérivée est un objet de la classe parente plus une partie qui correspond à la classe dérivée. Il est donc nécessaire d'initialiser la partie provenant de la classe parente lorsque l'objet est créé. Le constructeur de la classe dérivée doit donc faire appel au constructeur de la classe parente pour réaliser cette initialisation. Le plus souvent un constructeur de classe dérivée reçoit un ensemble de paramètres pour initialiser les attributs de la classe parente. Il utilise ces paramètres pour faire appel au constructeur de la classe parente.

Voici un exemple repris de l'étude de cas exemple « Médiathèque ».



La classe Document est notre classe parente. Elle contient des attributs privés : code, titre, etc. Ces attributs sont initialisés dans le constructeur. Les autres méthodes de la classe ne sont pas décrites. Seule la méthode `toString` dont nous parlerons avec le polymorphisme est définie.

Classe `prerequis.mediathequesimplifiee.heritage.Document`

```
1 package eu.telecomsudparis.csc4102.prerequis.mediathequesimplifiee.heritage;
2
3 public class Document {
4         private String code;
5         private String titre;
6         private String auteur;
7         private String annee;
8         private boolean empruntable;
```

```
9           private boolean emprunte;
10          private int nbEmprunts;
11          public Document(final String co, final String tit, final String aut,
12                          final String an) {
13                  this.code = co;
14                  this.titre = tit;
15                  this.auteur = aut;
16                  this.annee = an;
17                  this.empruntable = false;
18                  this.emprunte = false;
19                  nbEmprunts = 0;
20          }
21          @Override
22          public String toString() {
23                  return "Document [code=" + code + ", titre=" + titre + ", auteur="
24                                  + auteur + ", annee=" + annee + ", empruntable=" + empruntable
25                                  + ", emprunte=" + emprunte + ", nbEmprunts=" + nbEmprunts + "]";
26          }
27  }
```

La classe Audio est une spécialisation de la classe `Document`. Elle contient un attribut privé supplémentaire : classification. Cet attribut privé est initialisé dans le constructeur. Le constructeur de la classe Audio reçoit des paramètres pour initialiser ses attributs et des paramètres qu'il utilise pour faire appel au constructeur de la classe Document. Comme pour la classe parente, seule la méthode `toString` est définie.

Classe `prerequis.mediathequesimplifiee.heritage.Audio`

```
1  package eu.telecomsudparis.csc4102.prerequis.mediathequesimplifiee.heritage;
2
3  public final class Audio extends Document {
4          private String classification;
5          public static final int DUREE = 4 * 7;
6          public static final double TARIF = 1.0;
7          public Audio(final String code, final String titre, final String auteur,
8                          final String annee, final String classif) {
9                  super(code, titre, auteur, annee);
10                 this.classification = classif;
11         }
12         @Override
13         public String toString() {
14                 return "Audio [classification=" + classification + ", toString()="
15                                 + super.toString() + "]";
16         }
17 }
```

Voici une utilisation des classes parentes et dérivées.

Classe `prerequi.mediathequesimplifiee.heritage.ExempleHeritage`

```
1  package eu.telecomsudparis.csc4102.prerequis.mediathequesimplifiee.heritage;
2
3  public class ExempleHeritage {
4      public static void main(final String[] args) {
5          Document seigneur = new Document("C007",
6                  "Le seigneur des anneaux", "Tolkien", "1950");
7          Audio wyatt = new Audio("C003", "Rock bottom",
8                  "Rober Wyatt", "1973", "Progressif");
9          System.out.println("seigneur est un Document : "
10                 + (seigneur instanceof Document));
11         System.out.println("seigneur est un Audio : "
12                 + (seigneur instanceof Audio));
13         System.out.println("wyatt est un Document : "
14                 + (wyatt instanceof Document));
15         System.out.println("wyatt est un Audio : "
16                 + (wyatt instanceof Audio));
17     }
18 }
```

Résultat de l'exécution :

```
seigneur est un Document : true
seigneur est un Audio : false
wyatt est un Document : true
wyatt est un Audio : true
```

Dans cet exemple, la classe `Audio` étend (avec le mot-clé `extends`) la classe `Document`. Le constructeur de la classe `Audio` fait appel au constructeur de la classe `Document` à la ligne 9 par l'utilisation du mot réservé `super`. Le résultat de l'exécution permet de vérifier qu'un objet de la classe `Audio` est bien une instance de la classe `Document`.

20

10 Organisation des sources JAVA

- Unité de compilation
  - Un fichier source JAVA = une unité de compilation
  - Recommandation : une seule classe par fichier source
  - Obligation : nom du fichier source = nom de sa classe publique
- Paquetage
  - Paquetage = regroupement de classes dans un espace de nommage
  - Noms des classes : « `packagename.packagename.classname` »
  - Espace de nommage associé à la compilation et à l'exécution
    - Classe `Document` du paquetage `prerequis.mediathequesimplifiee.heritage` doit être dans un fichier correspondant au chemin `prerequis/mediathequesimplifiee/heritage/Document.java`
  - Ceci permet au compilateur et à la JVM de trouver les fichiers compilés
  - Mot réservé `package` : nom de paquetage des classes dans l'unité de compilation

Un fichier source en JAVA correspond à une unité de compilation. Une unité de compilation n'est compilable que si le compilateur dispose de l'ensemble des classes utilisées dans ce fichier. Ceci peut conduire un compilateur à compiler plusieurs classes lors d'une demande de compilation d'une seule classe. Il est fortement recommandé d'avoir une seule classe par fichier source. Un fichier source contenant une classe publique doit porter le même nom que cette classe.

Les paquetages permettent de regrouper un ensemble de classes dans un espace de nommage. Les noms des classes suivent le schéma « `packagename.packagename.classname` ». Cet espace de nommage est associé à la compilation et à l'exécution. La classe `Document` du paquetage « `prerequis.mediathequesimplifiee.heritage` » doit être dans un fichier correspondant au chemin « `prerequis/mediathequesimplifiee/heritage/Document.java` ». Ceci permet au compilateur et à la JVM de trouver les fichiers compilés. Le mot réservé `package` permet d'indiquer le nom de paquetage pour chaque unité de compilation.

10.1 Chemin de recherche et exécution d'un programme JAVA

21

- Chemin de recherche
  - Variable d'environnement `CLASSPATH` = liste des répertoires de recherche pour le compilateur et la JVM
  - Noms des classes complets contiennent le nom de paquetage
  - `import` : permet d'établir un alias
  - API JAVA : organisée en paquetages (`java.lang`, `java.util`, etc.)
- Exécution d'un programme JAVA
  - Point d'entrée `public static void main(String args[])` dans une classe
  - Classes chargées à la demande (en anglais, *dynamic loading*)

Les répertoires dans lesquels le compilateur et la JVM cherchent les paquetages sont décrits dans une variable d'environnement appelée `CLASSPATH`. À l'extérieur d'un paquetage, les noms des classes sont composés du nom de paquetage, d'un point et du nom de la classe. Un nom de paquetage peut être composé de plusieurs parties séparées elles aussi par des points. L'instruction `import` permet d'utiliser le nom de la classe importée sans le nom du paquetage en préfixe. L'API JAVA est organisée en paquetages (`java.lang`, `java.util`, etc.).

Les caractéristiques de la machine virtuelle JAVA font qu'il n'existe pas d'équivalent au programme en binaire exécutable obtenu dans une chaîne de compilation classique et en particulier en langage `C`. En effet, pour qu'un programme JAVA soit exécutable, il suffit d'un point d'entrée et d'un ensemble de classes. Le point d'entrée est matérialisé par la méthode `main`. Cette méthode contient les instructions de départ du programme. La machine virtuelle charge dynamiquement les classes qui sont référencées à partir de cette méthode. Le chargement des classes de l'API ne diffère pas de celui des autres classes.

Voici un exemple.

- CLASSPATH : `CLASSPATH=/src:/java`
- Fichier `eu/telecomsudparis/csc4102/prerequis/mediathequesimplifiee/organisationsources/Document.java` contenant le code suivant :

```
1  package eu.telecomsudparis.csc4102.prerequis.mediathequesimplifiee.organisationsources;
2
3  import eu.telecomsudparis.csc4102.prerequis.mediathequesimplifiee.classeobjet.Genre;
4
5  public abstract class Document {
6          private String code;
7          private String titre;
8          private String auteur;
9          private String annee;
10         private Genre genre;
11         private boolean empruntable;
12         private boolean emprunte;
13         private int nbEmprunts;
14         protected Document(final String co,
15                            final String tit, final String aut, final String an, final Genre g)
16                            {
17                 this.code = co;
18                 this.titre = tit;
19                 this.auteur = aut;
20                 this.annee = an;
21                 this.genre = g;
22                 this.empruntable = false;
23                 this.emprunte = false;
24                 nbEmprunts = 0;
```

```
25          }
26          public final String getCode() { return code; }
27          public final String getTitre() { return titre; }
28          public final String getAuteur() { return auteur; }
29          public final String getAnnee() { return annee; }
30          public final Genre getGenre() { return genre; }
31          public final int getNbEmprunts() { return nbEmprunts; }
32          public final void metEmpruntable() {     }
33          public final void metConsultable() {     }
34          public final boolean estEmpruntable() { return empruntable;      }
35          public void emprunter() {        }
36          public final boolean estEmprunte() { return emprunte; }
37          public void restituer() {        }
38  }
```

Pour compiler le fichier `Document.java`, le compilateur recherche le fichier source `Genre` du paquetage `eu.telecomsudparis.csc4102.prerequis.mediathequesimplifiee.organisationsources` à partir des chemins décrits dans le `CLASSPATH`. Il commence donc par chercher les sous-répertoires `eu`, `eu/telecomsudparis`, etc. dans le répertoire `/src`. Il ne trouve pas ce sous-répertoire et passe au chemin suivant. Il cherche dans `/java` et trouve un répertoire `eu/telecomsudparis/csc4102/prerequis/mediathequesimplifiee/organisationsources/Document.java`.
Dans ce répertoire il trouve le fichier class `Genre.class` ou il compile le fichier `Genre.java` pour obtenir le fichier `.class`. Avec ce fichier `.class` ainsi que les autres, il est à même de compiler le fichier `Document.java`.