Institut Paul Lambin

BAC 2 Informatique de gestion

Atelier Java

Synthèse Atelier Java

Auteurs :
Christopher Sacré

Professeur :
A. Legrand
L. Leleux
S. Ferneeuw
E. Leconte

12 février 2017

Table des matières

1 Semaine 1 **3**

1.1 Héritage . . . 3
1.1.1 Introduction . . . 3
1.1.2 Protected . . . 3
1.1.3 Constructeurs . . . 3
1.1.4 Substitution . . . 3
1.1.5 Overriding et principe de polymorphisme . . . 4
1.1.6 Méthodes et classe final . . . 4
1.1.7 Classe abstraite . . . 4
1.1.8 Redéfinition et modificateurs . . . 4
1.1.9 Méthodes et attributs static . . . 4
1.1.10 Masquage . . . 5
1.1.11 Principes . . . 5

1.2 Interfaces . . . 5
1.2.1 Notion d'interface . . . 5
1.2.2 Implémentation d'interface . . . 5
1.2.3 Héritage d'interfaces . . . 5
1.2.4 La collision de nom . . . 5
1.2.5 Interfaces vides . . . 6
1.2.6 Interfaces fonctionnelles . . . 6
1.2.7 Implémentation par défaut . . . 6
1.2.8 Méthodes statiques implémentées dans une interface . . . 6

1.3 Packages . . . 6
1.3.1 Usage . . . 6
1.3.2 Sous-package . . . 6
1.3.3 Utilité . . . 6
1.3.4 La clause import . . . 7

1.4 Les dates . . . 7
1.4.1 Introduction . . . 7
1.4.2 Créer une date/heure . . . 7
1.4.3 La classe Instant . . . 7
1.4.4 La classe LocalDate . . . 7
1.4.5 La classe LocalTime . . . 7
1.4.6 La classe LocalDateTime . . . 8
1.4.7 La classe ZonedDateTime . . . 8
1.4.8 La classe OffsetDateTime . . . 8
1.4.9 Formatage des dates . . . 8

1.5 Exceptions . . . 8
1.5.1 Définition . . . 8
1.5.2 Traiter les exceptions - Quand Traiter ? . . . 8
1.5.3 Traiter les exceptions - Comment traiter ? . . . 8
1.5.4 Postposer le traitement d'une exception . . . 9
1.5.5 Exceptions et héritage . . . 9
1.5.6 L'instruction throw . . . 9

1.5.7 Définir soi-même ses propres exceptions 9
1.6 Égalité entre les objets 9
1.6.1 objet1 == objet2 9
1.6.2 Égalité structurelle 9

1 Semaine 1

1.1 Héritage

1.1.1 Introduction

Une classe (classe mère) pourrait être étendue (à l'aide du mot clé "extends") en une sous-classe (classe enfant). Cette sous classe pourra être utilisée partout ou la classe mère pouvait être utilisée, contiendra tout ce qui se trouve dans la classe d'origine (méthodes et attributs).

Certains attributs private ou package ne sont pas accessible / visible mais existent néanmoins.

Toute classe hérite de la classe Objet (implicitement ou explicitement).

On ne peut hériter que d'une sous-classe.

1.1.2 Protected

Le mot clé "protected" permet de rendre un attribut ou une méthode d'une classe, il est disponible pour tous les enfants de la classe que ceux-ci se trouvent dans le même package ou non.

1.1.3 Constructeurs

Les constructeurs d'une classe peuvent s'appeler entre eux à l'aide de l'instruction "this (...)".

Une sous classe peut appeler le constructeur de la classe parent à l'aide de l'instruction "super(...)".

Ces deux instructions doivent obligatoirement être la première ligne du constructeur.

Si il n'y a pas d'appel explicite à super, Java effectuera d'office un appel implicite à super sans paramètres, si la classe parent ne possède pas de constructeurs vide, il y aura une erreur de compilation.

Lors de la construction d'un objet, les attributs sont en premier lieu initialisé aux valeurs par défaut, ensuite par celles indiquées dans leur initialisateur ("init") et enfin par les valeurs du constructeur.

1.1.4 Substitution

Un objet de la classe enfant peut être utilisé partout où on utilise un objet de la classe parent.

1.1.5 Overriding et principe de polymorphisme

On peut redéfinir une méthode héritée au sein d'une classe enfant. Une méthode redéfinie aura la même signature et le même type de retour que la méthode héritée (à ne pas confondre avec l'overloading (surcharge) qui permet d'avoir des méthodes de même nom mais de signature différente).

Si en utilisant la substitution on remplace un objet de la classe parent par un objet de la classe enfant, alors ce sera la méthode de la classe enfant qui sera appelée (dynamic binding).

1.1.6 Méthodes et classe final

- *Attribut final :* Attribut immuable au sein de l'objet.
- *Variable locale / paramètre final :* même signification d'immuabilité.
- *Référence final :* la référence et non modifiable mais pas le contenu de l'objet.
- *Méthode final :* la méthode ne peut être redéfinie dans une sous classe.
- *Classe final :* la classe ne pourra être redéfinie au sein d'une sous classe.

1.1.7 Classe abstraite

Une méthode abstraite est une méthode pour laquelle on désire indiquer l'existence mais pour laquelle on ne désire pas préciser l'implémentation. On le signifiera à l'aide du mot clé : "abstract".

Si une méthode d'une classe est abstract alors la classe elle même devra être déclarée abstract. Toutefois une classe pourrait être abstraite sans qu'aucune méthode ne le soit.

Il est impossible d'instancier des objets d'une classe abstraite. Elle peut néanmoins avoir des constructeurs.

1.1.8 Redéfinition et modificateurs

On peut redéfinir les modificateurs d'une méthode héritée. (cf tableau redéfinition modificateur).

1.1.9 Méthodes et attributs static

Une méthode ou un attribut static n'est jamais hérité (On ne peut pas redéfinir une méthode static). Ces attributs et méthodes restent tout de même accessible (ils sont appelés en étant précédés du nom de la classe). Une méthode static ne peut donc pas être abstraite.

1.1.10 Masquage

Seules les méthodes que l'on hérite (accessible dans la sous classe) sont modifiables. Si on définit une méthode avec la même signature qu'une méthode non accessible d'une classe parent on parle de masquage.

Pour accéder à un attribut, c'est toujours la classe de déclaration qui est utilisée.

Un attribut ou une méthode masquée peut être accédée via super.

1.1.11 Principes

Sur un objet on ne peut appeler que les méthodes déclarées dans sa classe de déclaration mais les méthodes qui seront effectivement appelées seront celles de sa classe de définition.

Sur un objet on ne peut accéder qu'aux attributs déclarés dans se classe de déclaration et les attributs effectivement accédés seront ceux de la classe de déclaration.

1.2 Interfaces

1.2.1 Notion d'interface

Une interface ne peut contenir que des méthodes abstraites et ne peux pas avoir d'attributs (à part des constantes). On les définira comme suit :

Les attributs seront d'office public static final. Les méthodes quant à elle seront toujours public abstract.

Une interface peut ne pas être visible en dehors du package.

1.2.2 Implémentation d'interface

Une classe n'étand pas une interface elle l'implémente (à l'aide du mot clé "implements"), une classe pourrait implémenter plusieurs interfaces.

1.2.3 Héritage d'interfaces

Les interfaces peuvent hériter les unes des autres et on peut hériter de plusieurs interfaces. Par contre une interface ne peut pas hériter d'une classe.

1.2.4 La collision de nom

Si une méthode a le même nom dans plusieurs interface implémentées dans une même classe il arrive deux cas :

— *Les méthodes ont des signatures différentes :* la classe devra implémenter les différentes versions.
— *Les méthodes ont la même signature :* Elles doivent obligatoirement avoir le même type de retour si l'on souhaite empêcher les erreurs de compilation. L'implémentation de la méthode vaudra pour les deux interfaces.

1.2.5 Interfaces vides

Certaines interfaces ne définissent aucune méthodes / aucune constantes. Mais cela est utilisé en java pour marquer les classes qui les implémentent.

1.2.6 Interfaces fonctionnelles

Interface possédant une unique méthode abstraite. (cf : interfaces fonctionnelles)

1.2.7 Implémentation par défaut

On peut fournir une implémentation par défaut pour certaines méthodes à l'aide du mot clé "default". Les classes filles ne doivent donc plus obligatoirement fournir une implémentation pour ces méthodes (en cas d'absence d'une implémentation spécifique, c'est celle par défaut qui sera utilisée).

1.2.8 Méthodes statiques implémentées dans une interface

Les méthodes statiques peuvent êtres définies et implémentées dans des interfaces.

1.3 Packages

1.3.1 Usage

Un package est un regroupement de classe ayant un même thème. Une classe faisant partie d'un package aura comme première ligne : "package monPackage ;".

1.3.2 Sous-package

Les packages peuvent eux-mêmes contenir des sous packages. La première ligne d'une classe de ce package sera : "package monPackage.monSousPackage ;"

1.3.3 Utilité

Les packages permettent de définir un espace de noms (On peut ainsi donner à plusieurs classes le même nom).

Les packages sont également un moyen de protection. Ils limitent les accès aux classes, méthodes et attributs. Ils permettent de rajouter un niveau d'encapsulation.

1.3.4 La clause import

Si l'on désire inclure à notre programme des classes situées dans un autre package (autre que java.lang ou que celui ou est placé la classe), on doit importer ce package (ou au moins les classes que l'on désire utiliser). Pour cela on utiliser l'instruction "import".

Si l'on ne spécifie pas de package pour notre classe elle sera dans le package "default". Il est impossible d'importer le package "default".

Il y a deux sortes d'import :
— *Les imports simples :* "import package.class ;".
— *Les imports à la demande :* "import package.* ;", le compilateur n'importera que les classes nécessaire.

1.4 Les dates

1.4.1 Introduction

L'API java.time contient un ensemble de classes permettant de gérer les dates et les heures.
Il existe deux types de temps :
— *Temps absolu :* définis de façon unique pour tous les utilisateurs du monde.
— *Temps humain :* correspond à la manière dont on utilise le temps de façon usuelle et plus ou moins locale.

1.4.2 Créer une date/heure

L'API ne fournis pas de constructeurs, ainsi pour créer un de ces objets il faut passer par une méthode (ex : now()).

1.4.3 La classe Instant

Elle représente le temps absolu. (La classe Duration représente une durée entre deux instants).

1.4.4 La classe LocalDate

Elle permet de manipuler les dates usuelles.

1.4.5 La classe LocalTime

Permet de manipuler des heures de la journée.

1.4.6 La classe LocalDateTime

Permet de manipuler un objet contenant une date et une heure (sans préciser le fuseau horaire). Cette classe ne tient pas compte du changement d'heure.

1.4.7 La classe ZonedDateTime

Permet de manipuler un objet contenant une date et une heure en précisant le fuseau horaire. Elle tient compte des particularités locales ainsi que du changement d'heure.

1.4.8 La classe OffsetDateTime

Permet de manipuler un objet contenant une date et une heure en précisant le décalage horaire par rapport au méridien de Greenwich.

1.4.9 Formatage des dates

La classe DateTimeFormatter fournit 3 types de formateurs afin d'afficher des dates/heures.

- *Des formatteurs standards prédéfinis :* utilisées en général pour les échanges entre logiciels.
- *Des formatteurs locaux prédéfinis :* permet d'avoir des présentations plus usuelles des dates en tenant compte des spécificités locales.
- *Des formatteurs suivant un pattern à définir :* créés à l'aide de la méthode statique ofPattern(String,pattern).

Une fois qu'on possède un objet de la classe DateTimeFormatter, on peut utiliser la méthode format afin d'avoir une chaîne de caractères au format voulus.

1.5 Exceptions

1.5.1 Définition

Les exceptions sont des classes comme les autres héritant de la classe "Exception".

1.5.2 Traiter les exceptions - Quand Traiter ?

Avant toute chose : se poser la question : "Suis je au bon endroite et en mesure de le faire ? ". Si ce n'est pas le cas on laissera l'appelant le faire.

1.5.3 Traiter les exceptions - Comment traiter ?

Pour traiter les exceptions on utilise les instructions "try...catch...finally".

- *try :* on y place ce qu'on essaie de faire et qui pourrait provoquer une exception.
- *catch :* le traitement à effectuer en cas d'erreur.

— *finally :* Utilisé lorsqu'un traitement doit être effectué dans tous les cas (Si un system.exit() a lieu, le bloc finally ne sera pas exécuté).

Plusieurs blocs catchs sont permis mais on doit respecter la hiérarchie des classes d'exceptions (d'abord la classe enfant, puis la classe parent).

1.5.4 Postposer le traitement d'une exception

Si une exception est checked, il est obligatoire d'annoncer dans l'en-tête de la méthode l'existence de cette exception. Cela se fait au moyen du mot clé : "throws". Il est onc obligatoire de traiter les checked exceptions, ce qui n'est pas le cas des unchecked.

1.5.5 Exceptions et héritage

Si une exception est postposée, toutes les exceptions ayant héritée de celle-ci sont également postposée.

Une méthode redéfinie n'a pas le droit de postposer des checked exceptions qui ne le serait pas par la méthode qu'elle redéfinit dans la classe parent.

1.5.6 L'instruction throw

Si l'on désire lancer une exception nous même on utilise l'instruction "throw".

1.5.7 Définir soi-même ses propres exceptions

Il faut avant toute chose définir une sous classe d'Exception. A partir de là on peut distinguer deux cas :

— On ne désire pas faire passer de message :

— On désire faire passer un message :

Les méthodes héritées de classe Exception :

— *String getMessage() :* affiche le message d'erreur

— *String toString() :* renvoie une String formée du nom de la classe + du message d'erreur.

— *void printStackTrace() :* affiche le contenu de le pile des appels lors du lancement d'une exception.

1.6 Égalité entre les objets

1.6.1 objet1 == objet2

Objet 1 et objet2 sont égaux si ils ont la même référence en mémoire.

1.6.2 Égalité structurelle

On va regarder la structure de l'objet pour définir une égalité. Pour cela on va utiliser deux méthodes : equals et hascode.

Quand on écrit la méthode equals, on doit obligatoirement écrire la méthode equals (Deux objets égaux doivent avoir le même hashcode).

Lorsque l'on ne redéfinit pas les méthodes equals et hashcode elles ne feront que comparer les références des objets.