

PROGRAMMER EN JAVA



1

Objectifs de la formation

- Comprendre la syntaxe et la sémantique de Java
- Comprendre et appliquer les concepts objet avec Java
- Connaître et utiliser les classes de base de Java

Sommaire

- Présentation de Java
- Java en ligne de commande
- Présentation d'Eclipse
- Les bases du langage
- Les concepts objet avec Java
- Les exceptions
- Les classes de base
- Les collections
- Les entrées / sorties (IO et NIO)
- L'accès aux bases de données
- Les Lambda Expressions
- Les streams

PRÉSENTATION JAVA

4

Historique de Java

Caractéristiques de Java

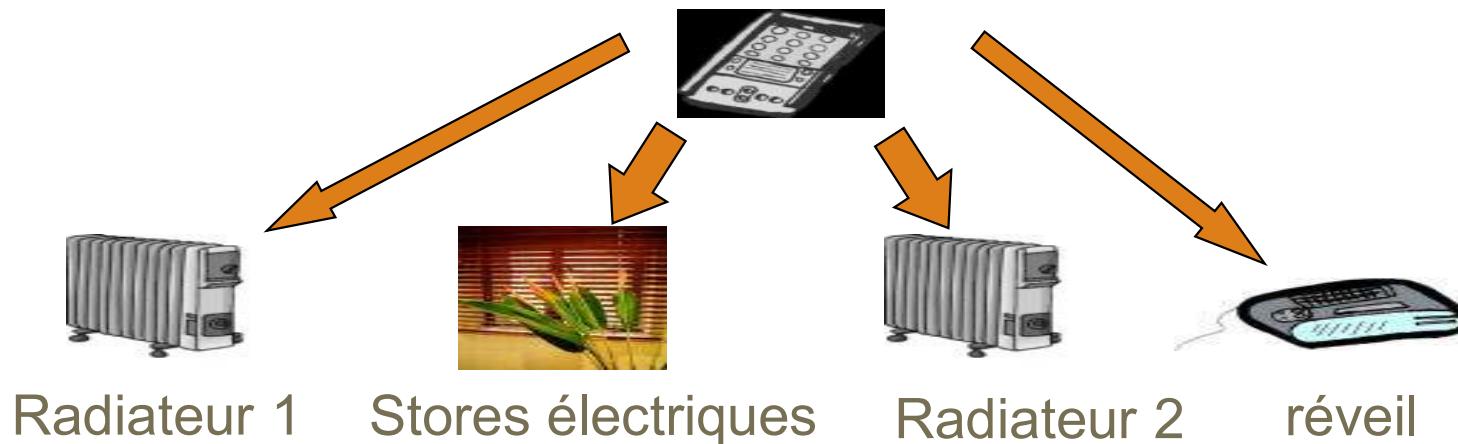
Positionnement de Java par rapport aux autres langages

Historique



➤ 1990 : Le ‘Green Project’

- Une équipe d'ingénieurs de Sun conçoit un nouveau langage pour l'électronique grand public. Le langage doit être portable, simple et compact.
- Ce langage devait permettre à des appareils ménagers hétérogènes de communiquer entre eux via une plate-forme commune.



Historique

- 1992 : Un début prometteur
 - ▶ Création du langage Oak par Grosling
 - ▶ Première démonstration (PDA Star 7)
 - ▶ Duke, la mascotte de Java
 - ▶ Crédit de la filiale FirstPerson, Inc.
- 1994 : Le marché de la domotique ne décolle pas
 - ▶ Sun doit faire un choix :
 - ❑ Stopper le projet, et attendre que le marché se développe.
 - ❑ Trouver un autre domaine dans lequel un langage portable pourrait être utile.
 - ▶ Recentrage du projet sur le Web



Historique



- Début 1995 : Le marché de l'Internet est en pleine explosion
 - ▶ Les sites Internet statiques fleurissent sur le Web
 - ▶ Sun travaille sur son propre navigateur Web et cherche le moyen de télécharger facilement des applications pour les exécuter dans ce navigateur
 - ▶ La 'Green team' réalise que l'Internet aura besoin d'un langage multi-plates-formes.

« WebRunner était juste une démo, mais une démo impressionnante : Il a porté à l'écran, pour la première fois, des objets animés et un contenu dynamique exécutable à l'intérieur d'un navigateur Web. »()*



Historique



- Mai 1995 : Sun met à disposition le code source de Java sur Internet
 - ▶ En quelques mois, Java devient célèbre. Plusieurs milliers d'utilisateurs le téléchargent chaque mois.
- Fin 1995 : Netscape intègre Java dans son navigateur
 - ▶ Java devient potentiellement utilisable par des millions d'Internautes.
- 1996 : Microsoft intègre Java dans Internet Explorer
 - ▶ Sun rallie de nombreux partenaires (IBM, Oracle, Netscape...) autour de la technologie Java

Historique



- 1996 : Version 1.0 du Java Development Kit (JDK)
 - ▶ Lancement officiel de Java par Sun et de nombreux autres partenaires (IBM, Oracle, Netscape...)
 - ▶ Première conférence JavaOne
- 1997: JDK 1.1
 - ▶ Téléchargé plus de 220'000 fois en 3 semaines
 - ▶ 1998:
 - Plus de 2 millions de téléchargements
 - Formalisation du programme Java Community Process (JCP)
 - Cartes visas basées sur Java Card
- 1998 : Java 2
 - ▶ Sortie de Java 2 (JDK 1.2)
 - ▶ Séparation des plateformes : J2SE, J2EE et J2ME
 - ▶ Première beta de la plateforme J2EE (Java 2 Enterprise Edition)

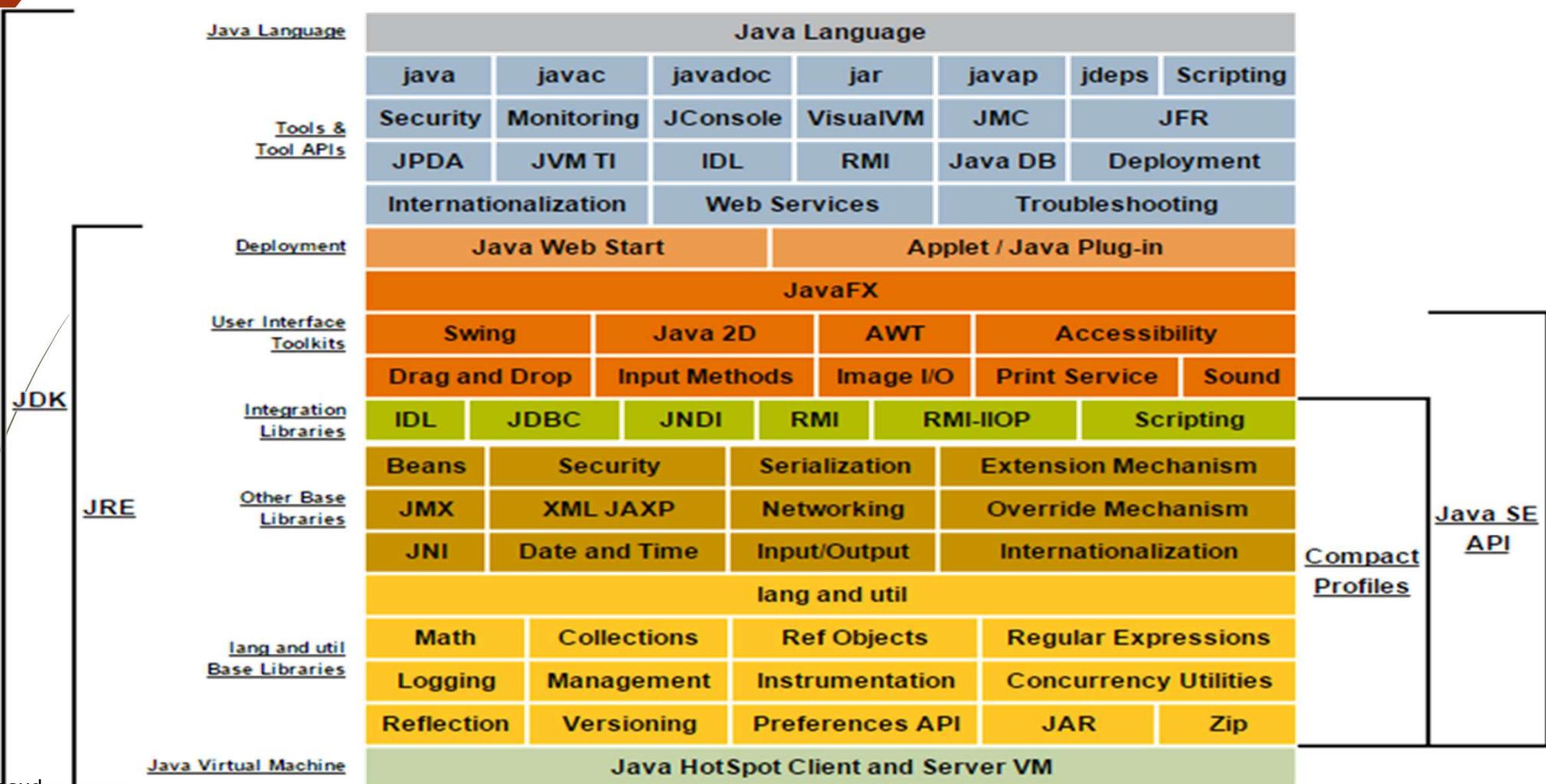
Versions

➤ Les versions

► JDK 1.0	- 1996	- 0211	classes
► JDK 1.1	- 1997	- 0477	classes
► JSE 1.2	- 1998	- 1524	classes
► JSE 1.3	- 2000	- 1840	classes
► JSE 1.4	- 2002	- 2723	classes
► JSE 5.0	- 2004	- 3279	classes – 166 packages
► JSE 6.0	- 2006	- 3793	classes – 203 packages
► JSE 7.0	- 2011	- 4025	classes – 209 packages
► JSE 8.0	- 2014	- 4241	classes – 217 packages
► JSE 9.0	- 2017	- 5987	classes – ~300 packages
► JSE 10 / 11	- 2018	- 4580	classes – ~223 packages
► JSE 12 / 13	- 2019	- 4555	classes – ~223 packages
► JSE 14/15/16	- 2020	- 4420	classes – ~223 packages
► JSE 17	- 2021	- 4422	classes – ~223 packages
► JSE 18/19	- 2022		

L'architecture Java

11



Java 9 / 10 / 12 / .. / 19 ?

12

- Oracle ne fournit un support que sur les versions *Long Term Support* (LTS)
- Cela implique que les versions qui ne sont pas LTS disparaissent aussi rapidement qu'elles sont apparues
- Il est très fortement conseillé de rester sur des versions LTS



A los Spazier'

Java versions LTS

13

- 7 (LTS) Sortie en Juillet 2011
 - ➡ Support jusqu'en Juillet 2022
- 8 (LTS) Sortie en Mars 2014
 - ➡ Support jusqu'en Décembre 2030
- 11 (LTS) Sortie en Septembre 2018
 - ➡ Support jusqu'en Septembre 2026
- 17 (LTS)) Sortie en Septembre 2021
 - ➡ Support jusqu'en Septembre 2029
- 21 (LTS) Sortie en Septembre 2023
 - ➡ Support jusqu'en Septembre 2031

Java 11 et les versions suivantes

- Depuis septembre 2018
- Ne contient plus (du tout) les API :
 - ▶ pour les applets
 - ▶ le Java Web Start
 - ▶ dépréciées dans les Thread (stop, destroy)
 - ▶ JAX, JAX-WS, JAF, JAXB, CORBA, JTA
 - ▶ Java FX
- Ne contient plus l'outil
 - ▶ Java Mission Control
 - ▶ La mise à jour automatique



Java 17 et les versions suivantes

15

- Est de nouveau gratuit en usage production, mais ...
 - ▶ Était payant depuis la version 8
- Vous avez une deadline de gratuité
 - ▶ Par exemple le JDK 17 est gratuit jusqu'en septembre 2024
 - ▶ Attention : bien consulté les règles sur
<https://www.oracle.com/java/technologies/javase/jdk-faqs.html>
- Objectifs :
 - ▶ Limiter les productions qui font usage d'un OpenJDK ou d'une 'vieille version'
 - ▶ Forcer les entreprises à migrer à chaque rapidement (et pas toutes les 10 ans ...)



Les caractéristiques de Java

16

Objet

Portable

Compact

Robuste

API

Multitâches

Les caractéristiques de Java : Objet

17

➤ Java est un langage objet

- Des objets constituent une abstraction du monde réel.



```
public class Ampoule {           Conception
    private int puissanceW;
    private int etat;

    public void allumer() { ... }
}
```

```
Ampoule a = new Ampoule();   Utilisation
a.allumer();
a.ajusterLuminosite(10);
```

Les caractéristiques de Java : Portable

18

- Avec d'autres langages de programmation, un programme compilé sous Windows ne s'exécute pas dans un environnement Unix.
- Java est un langage **portable** : ***WRITE ONCE, RUN ANYWHERE***
 - ▶ Le même code Java peut s'exécuter sur toutes les plates-formes qui supportent Java, sans recompilation.
 - ▶ Les programmeurs Java n'ont pas à gérer les conversions de types primitifs d'une plate-forme à l'autre (integer, float...).
- La portabilité Java est basée sur le concept de machine virtuelle.

Principe de fonctionnement (1)

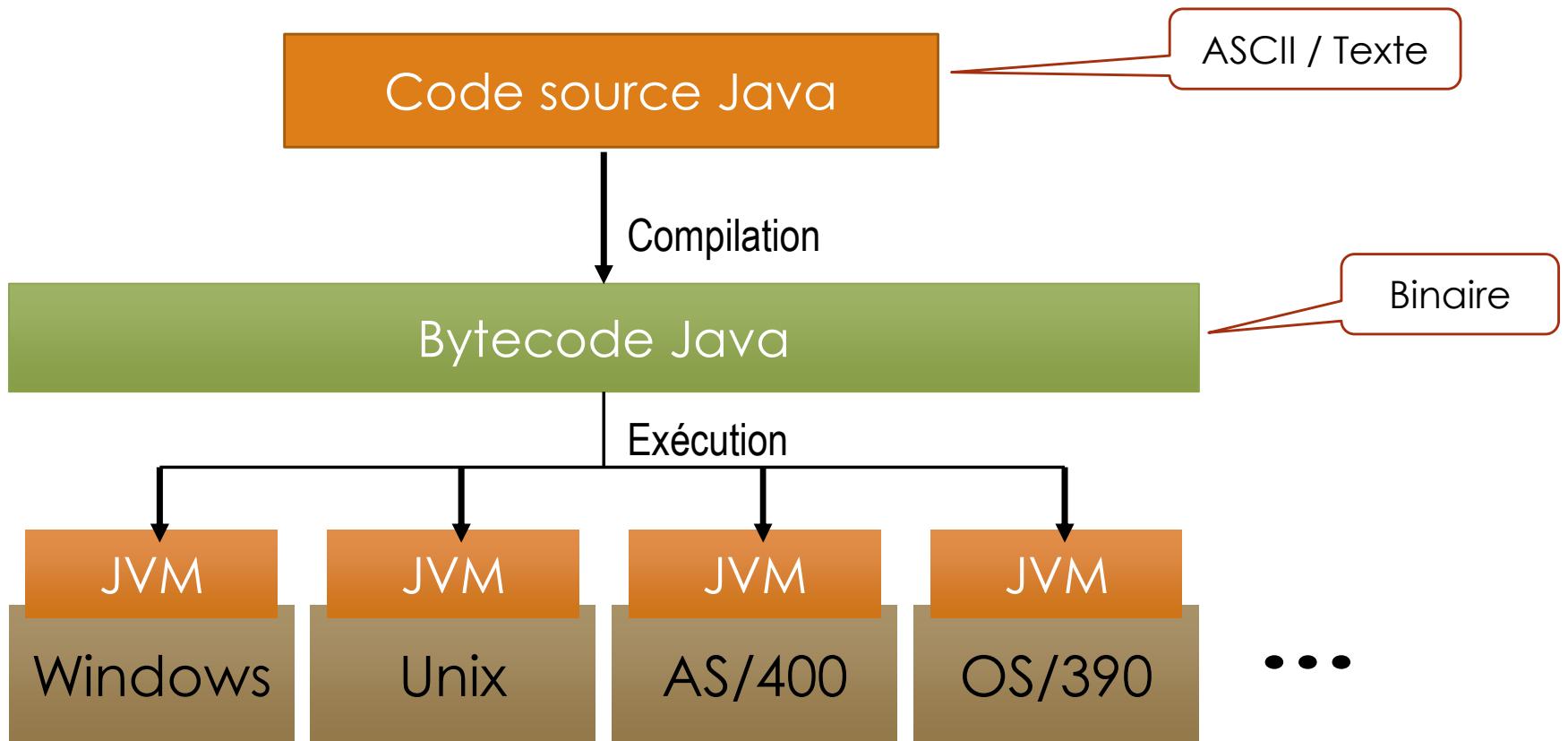
➤ Principe de la machine virtuelle

- ▶ le compilateur ne génère pas un exécutable pour un processeur particulier, il génère du code binaire (bytecode) pour un processeur virtuel (la machine virtuelle)

➤ Portabilité

- ▶ la machine virtuelle est un programme disponible sur de nombreuses plates-formes
- ▶ le bytecode est traité par la machine virtuelle (tout comme les instructions d'un programme classique sont traitées par le processeur)
- ▶ ce traitement peut être une interprétation ou bien compilation dynamique.

Principe de fonctionnement (2)



Les machines virtuelles Java (JVM)

- Les fonctionnalités d'une JVM sont clairement spécifiées
 - ➡ Plusieurs éditeurs peuvent implémenter la machine virtuelle
 - ➡ Les principaux éditeurs sont Sun/Oracle et IBM
 - ➡ La machine virtuelle est disponible pour de nombreux OS

Les machines virtuelles Java (JVM)

- Il existe différentes techniques d'implémentation de machines virtuelles
 - ▶ Interpréteur de bytecode
 - Première génération de compilateur
 - ▶ Compilateur natif : transforme les fichiers sources ou bytecode en fichiers exécutables spécifiques au système d'exploitation.
 - Bonnes performances au détriment de la portabilité.

Les machines virtuelles Java (JVM)

- Il existe différentes techniques d'implémentation de machines virtuelles (suite)
 - Compilateur juste à temps (just-in time / JIT) : compilation du code à la 1ère demande et mise en cache, puis exécution du code compilé plus rapide
 - Performant et portable
 - Recompilateur dynamique : la machine virtuelle analyse le comportement du programme et en recompile sélectivement les parties.
 - Technique plus sophistiquée pouvant offrir de meilleures performances que les compilateurs JIT.

Les caractéristiques de Java : Compact

➤ Java est **compact**

- ▶ Cet aspect a été privilégié depuis l'origine de Java
- ▶ Le bytecode dispose d'instructions de haut niveau
- ▶ La machine virtuelle fait une grande part du travail

Les caractéristiques de Java : Robuste 1/2

25

➤ Java est **robuste**

- ▶ Le développeur ne manipule pas directement la **mémoire**
 - Pas de notion de pointeur ⇔ Tout est référence sauf les types primitifs
- ▶ La machine virtuelle **gère la libération de la mémoire**
 - Les objets sont détruits par le ‘garbage collector’ (ramasse miettes)
 - Le travail du développeur est simplifié
 - Pas de risque de libérer deux fois la même zone ou d’oublier de libérer

Les caractéristiques de Java : Robuste 2/2

26

➤ Java est **robuste**

- ▶ Contrôle de **type très strict**
 - Typage fort
- ▶ Tous les **attributs sont initialisés** (pas de valeurs résiduelles)
 - Attributs (mais pas les variables locales)
- ▶ Les **bornes des tableaux** sont toujours contrôlées
- ▶ Les **erreurs potentielles doivent être traitées** par le développeur : notion d'exceptions
 - Gestion coercitive des exceptions

Les caractéristiques de Java : API

27

- Java est composé de plusieurs **API**
 - ▶ Réseaux
 - ▶ Interface graphique (IHM / GUI)
 - Lourde : Java FX / Swing / AWT
 - Web : JSTL / JSF
 - ▶ Bases de données
 - ▶ Annuaires (LDAP)
 - ▶ ...
- En Java, avant de coder, on regarde si cela n'existe pas déjà
- Tout existe sauf votre code métier/applicatif

Les caractéristiques de Java : multi-tâches

28

➤ Java est multi-tâches

- ▶ Java intègre la notion de threads : la machine virtuelle peut gérer l'exécution de plusieurs « parties du programme » simultanément
- ▶ La machine virtuelle pourra profiter de plusieurs processeurs s'ils sont disponibles (vrai parallélisme)
- ▶ Les programmes Java qui utilisent le multitâches restent portables
- ▶ Cette fonctionnalité permet à Java d'être très bien adapté à l'écriture d'applications s'exécutant sur un serveur (nécessité de traiter simultanément des requêtes de plusieurs utilisateurs)

Langage ou plate-forme ?

29

- Java est plus qu'un langage
- Le JRE (**J**ava **R**untime **E**nvironnement) contient
 - ▶ une machine virtuelle (Java Virtual Machine)
 - ▶ un ensemble de librairies (Java Application Programming Interface)
- Exemple de librairies (regroupées dans différents packages)
 - ▶ JDBC : Java **D**a**t**a**B**ase **C**onnectivity
 - ▶ AWT : **A**bstract **W**indow **T**oolkit
 - ▶ RMI : **R**emote **M**ethod **I**nvocation
 - ▶ JNI : Java **N**ative **I**nterface
- Java est utilisable sur différentes plates-formes
 - ▶ Linux, Windows, iOS mais aussi Java Card, serveur z/Os...

Java et les autres langages

- Java est une synthèse de nombreux autres langages
- C'est un langage objet qui peut être directement comparé au mariage entre C++ et Smalltalk

Java et les autres langages (2)

➤ Java et C++

- ▶ un point commun : la syntaxe
- ▶ plus simple : élimination de certains aspects difficiles
 - ❑ pointeurs
 - ❑ héritage multiple
 - ❑ gestion de la mémoire
- ▶ plus robuste : gestion automatique de la mémoire
- ▶ plus objet : C++ autorise une programmation mixte
- ▶ moins proche de la machine
 - ❑ mais possibilité d'utiliser JNI pour appeler des librairies C
- ▶ légèrement moins performant à l'origine
 - ❑ l'écart s'est beaucoup réduit avec les JVM optimisées
 - ❑ Java peut prendre l'avantage dans certains cas

Java et les autres langages (3)

32

➤ Java et Smalltalk

- ▶ conceptuellement très proche : machine virtuelle, garbage collector, librairies riches, approche objet obligatoire
- ▶ Java est plus standard
 - bytecode standardisé : plusieurs éditeurs de machine virtuelle
 - librairies standardisées : plusieurs fournisseurs (IBM, Microsoft...)
 - accepté par de nombreux éditeurs
- ▶ Java est plus accessible
 - Java est typé
 - syntaxe moins perturbante
 - nombreuses informations disponibles
- ▶ Java est moins adapté aux développements génériques (typage fort et absence de métaclasses)
- ▶ Java est plus verbeux (typage)

Quiz

- Pourquoi parle-t-on de Java comme d'un langage **portable** ?

- J'ai écrit une application Java simple. Elle marche avec la machine virtuelle de Oracle.
 - ➡ Marchera-t-elle sous une machine virtuelle IBM ?
 - ➡ Faut-il recompiler ?

Java en ligne de commande

34

Présentation du JDK

Structure des programmes

Exemple de programme

Éléments du langage java

JDK (Java Development Kit)

- Contenu du kit de développement
 - ▶ Outils
 - ❑ Compilateur / Interpréteur / Débogueur
 - ❑ Générateur de documentation
 - ▶ Bibliothèques de classes
 - ▶ Le code source des classes
 - ▶ Des exemples
 - ▶ Une base de données embarquée (selon la version du Java)
 - ▶ Documentation (à télécharger en supplément)
 - ❑ Des outils
 - ❑ Des classes
- Disponible sur le site
 - ▶ <https://www.oracle.com/java/technologies/downloads/>

JRE (Java Runtime Environment)

- Minimum requis pour exécuter un programme java
- Sous-ensemble du JDK
 - ➡ Bibliothèque des classes (bytecode seulement), sous forme de fichiers jar
 - ➡ Interpréteur
- A déployer sur toute machine où l'on souhaite simplement exécuter des applications Java

Structure des programmes

37

➤ Code Java

- ▶ **Physiquement** : des fichiers .java organisés en répertoires
- ▶ **Logiquement** : des classes(*) organisées en paquetages (**)
- ▶ Une classe correspond en général à un fichier de même nom
 - Source avec l'extension '.java' : Client.java
 - Après compilation avec l'extension '.class' : Client.class

➤ Une application

- ▶ Regroupe un ensemble de fichiers
- ▶ Possède un « point d'entrée » (dépend du type d'application)

Types de programmes - Main

➤ Application autonome

- ▶ S'exécute sur un poste client
- ▶ Point d'entrée : une classe possédant une méthode « **main** »
- ▶ Livrable sous la forme d'un ou plusieurs **JAR** (Java **ARchive**)
- ▶ *Nos premiers exercices*

Types de programmes - Applet

➤ Applet

- ▶ Application graphique embarquée dans une page HTML
- ▶ Hébergée sur un serveur
- ▶ S'exécute au sein d'un navigateur qui invoque les services de l'applet, en particulier le service « paint » pour la dessiner
- ▶ Modèle de sécurité restreint
- ▶ Livrable sous la forme d'un ou plusieurs **JAR** (Java Archive)
- ▶ **Déprécié depuis Java 7**
- ▶ **Enlevé de Java 11**



Types de programmes – WEB dynamique

40

➤ **Servlet** ou application web

- ▶ Application serveur qui répond aux demandes d'un client HTTP
- ▶ Hébergée sur un serveur
- ▶ S'exécute côté serveur via les classes servlet, en particulier leur méthode « service » pour demander de traiter une requête du client
- ▶ Livrable sous la forme d'un ou plusieurs **WAR** (Java Web **ARchive**)

Types de programmes – WEB applicatif

➤ Application web **d'entreprise**

- ▶ Application serveur qui répond aux demandes d'un client HTTP
- ▶ Hébergée sur un serveur
- ▶ S'exécute côté serveur via les classes EJB
- ▶ Livrable sous la forme d'un ou plusieurs **EAR (EntrepriSe ARchive)**

Exemple de programme

- **main** : point d'entrée pour exécution d'un programme

```
public class Bonjour {  
    public static void main(String[] args) {  
        System.out.println("Bonjour tout le monde");  
    }  
}
```

- **void** : la méthode ne retourne aucun résultat
- **static** : méthode de classe et non d'instance
- Les arguments/paramètres de la méthode main sont des chaînes de caractères (**String[]** args)
- System.out.println(...) affiche sur la sortie standard. On utilise la **méthode println** qui s'applique à l'**attribut out** de la **classe System** et qui prend une chaîne de caractères en **paramètre**.
- le caractère ; est un terminateur d'expression

Bibliothèque de classes

43

- Classes de base
 - ▶ Constituent un noyau pour le développement
 - ▶ Regroupées en packages (concrètement des répertoires)
 - lang, awt, util, applet, sql, ...
 - ▶ Développées en java
 - Certaines sont en fait du code natif (code qui est dans les DLLs ou .SO de la JVM)
- Fournies sous forme de fichier compressé
 - ▶ src.jar ou src.zip : code source (fichiers .java) à la racine du JDK
 - ▶ rt.jar : bytecode résultat de la compilation (fichiers .class) dans le répertoire lib du JRE

Modules et dépendance

Java 9

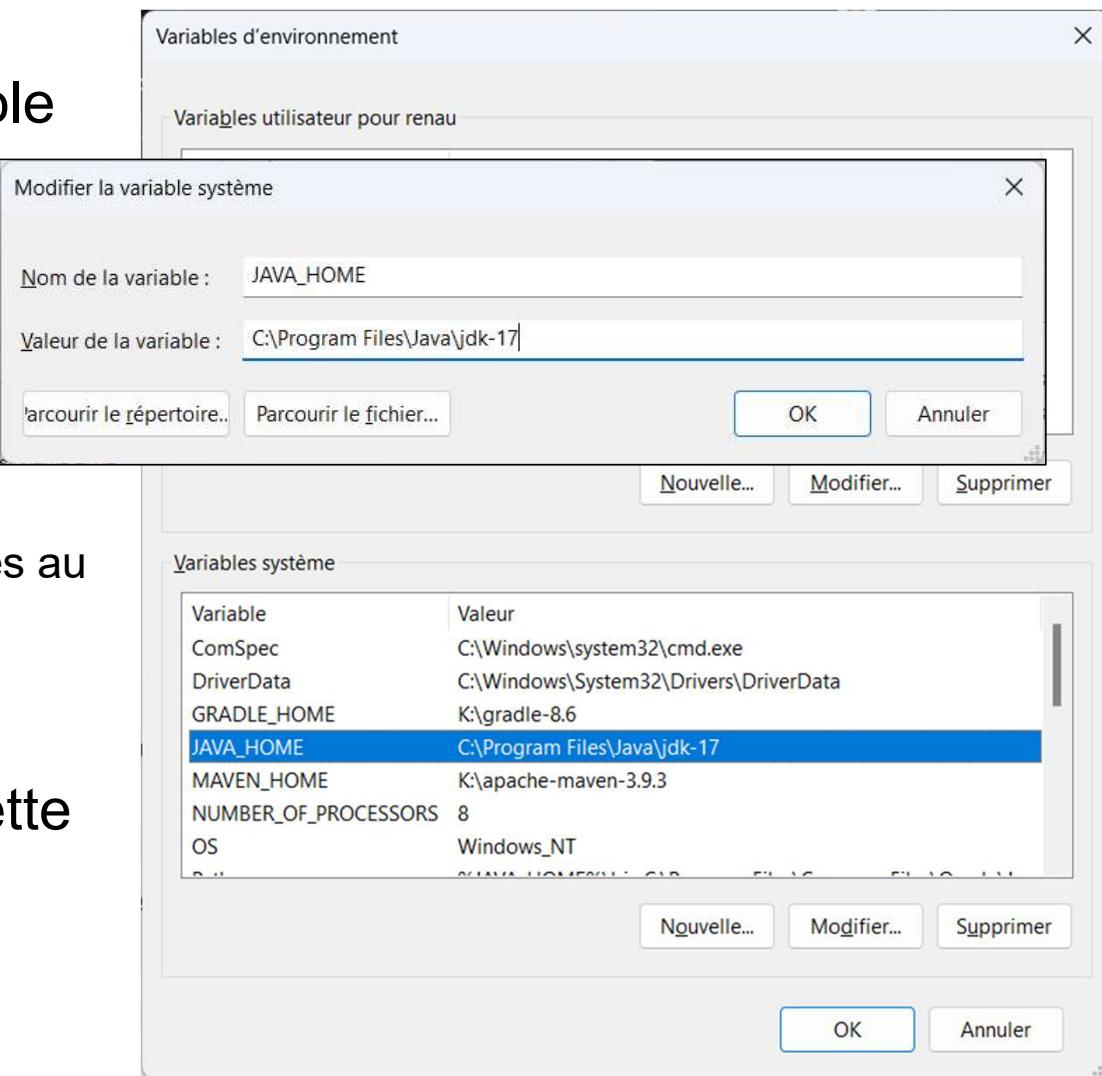
44

- La notion de module a été introduite en Java 9
- Objectif : renforcer la sécurité des accès entre librairies
- Un module
 - regroupement de package
 - une ou plusieurs règles de dépendances sur les modules
 - Une ou plusieurs règles de sécurité (sur la réflexion par exemple)

Variables d'environnements : JAVA_HOME

45

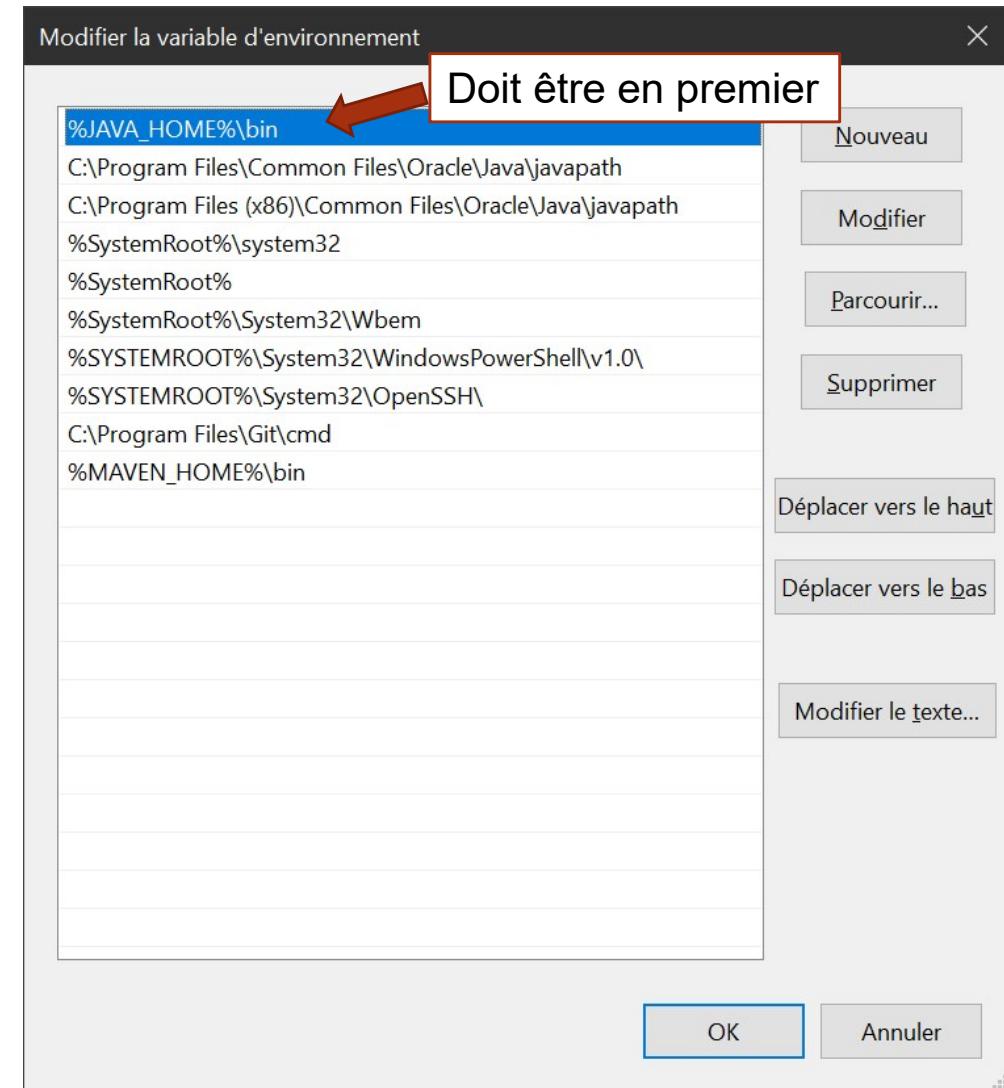
- Conçu pour être utilisé dans une console ou par des scripts
- Nécessite de spécifier le chemin d'installation Java
 - ➡ Variable d'environnement **JAVA_HOME**
 - ➡ **SET JAVA_HOME="C:\Program Files\Java\jdk-17"**
 - ❑ Attention aux guillemets si le chemin d'accès au JDK contient des espaces
 - ❑ Attention aux **majuscules / minuscules**
- Il est très fortement conseillé d'avoir cette variable.
- Elle est en **MAJUSCULES** et c'est une variable **système**



Variables d'environnements : PATH

46

- Modification de sa variable **PATH**
 - ➡ Variable d'environnement **PATH**
 - ➡ **SET PATH=%JAVA_HOME%\bin;%PATH%**
 - On fait usage de **JAVA_HOME**
 - On complète avec un \bin (ou /bin sous Unix)
 - On complète en partant de la gauche, puis on ajoute le PATH initial
- Attention le séparateur d'éléments du **PATH** est
 - ➡ ; pour Windows
 - ➡ : pour Unix
- Il est très fortement conseillé d'ajuster cette variable en faisant usage de **JAVA_HOME**.
- Il faut parfois rebooter (surtout sous Windows)



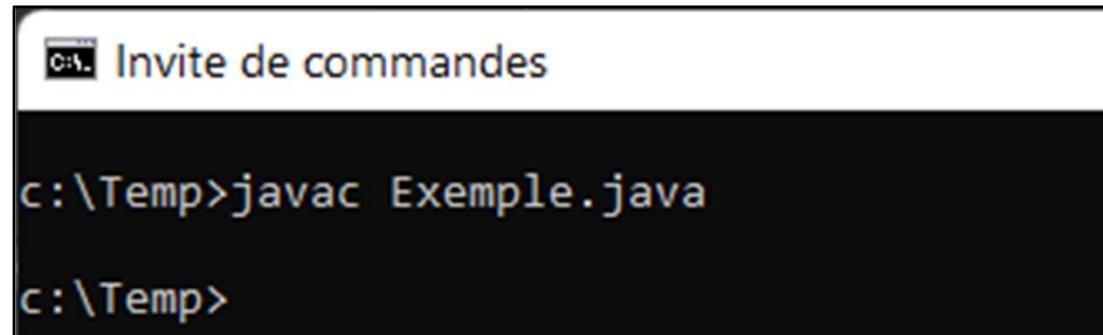
Variables d'environnements : CLASSPATH

47

- Permet de spécifier les chemins des librairies
 - ▶ Variable d'environnement **CLASSPATH**
 - ▶ Indique les fichiers jar et les répertoires contenant des classes
 - ▶ Précise dans quel ordre la recherche s'effectue
 - ▶ SET **CLASSPATH=.:;%JAVA_HOME%\jre\lib**
- Problèmes de conflits en cas d'environnements multiples
 - ▶ Vous pouvez aussi utiliser l'option -classpath ou -cp
- Attention : le séparateur d'éléments du classpath est ; pour Windows et : pour Unix
- Cette variable **n'est pas obligatoire.**

Compilateur : JAVAC

- Produit les fichiers compilés (.class) à partir des sources (.java)
 - ▶ Sous forme de bytecode
- Ligne de commande
 - ▶ javac Fichier.java
 - ▶ Possibilité de compiler tous les fichiers du répertoire :
javac *.java
- Utilise la variable CLASSPATH mais il est maintenant recommandé d'utiliser l'option -classpath à la place
 - ▶ javac -classpath .;netlibs.jar;tests.jar Reseau.java



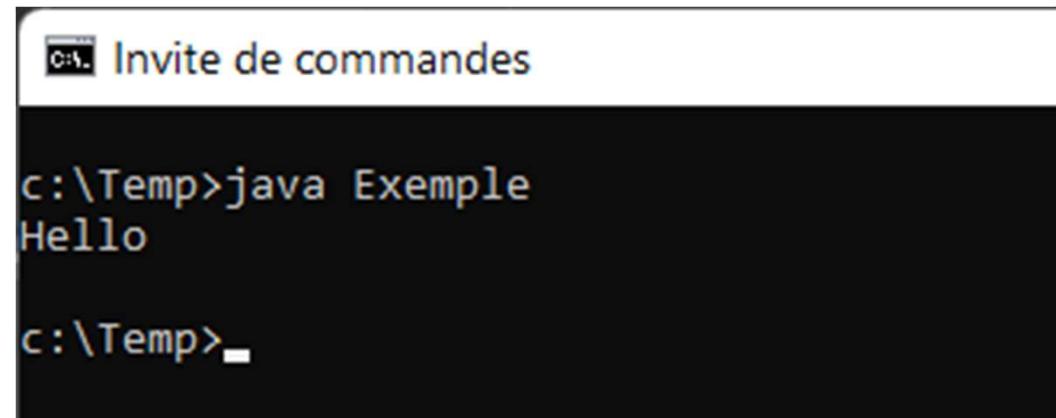
```
c:\ Temp>javac Exemple.java
c:\ Temp>
```

Compilateur : JAVAC - options

- -classpath (ou -cp)
 - ▶ Surcharge la variable CLASSPATH
- -d
 - ▶ Permet d'indiquer un répertoire destination pour les fichiers .class
- -verbose
 - ▶ Affiche des informations pendant la compilation
- -help
 - ▶ Aide sur les options
 - ▶ Fournit d'autres options non mentionnées ici

Lanceur : JAVA

- Exécute le bytecode java
- Ligne de commande
 - java [-options] NomClasse [arguments]
 - Utilise la variable CLASSPATH (option -classpath)
 - Exécute la méthode main de la classe passée en paramètre
 - java Fichier
 - Le nom de la classe doit être complet (packages + classe)
 - java com.sql.formation.BankonetTest
 - Arguments : chaînes de caractères séparées par des espaces
 - java com.sql.formation.BankonetTest compte1 compte2 compte3



```
c:\Temp>java Exemple Hello
c:\Temp>
```

Lanceur : JAVA - options

51

- -classpath ou -cp
 - ▶ Spécifie le chemin de recherche des classes
 - ▶ Substitut recommandé à la variable CLASSPATH
- -version
 - ▶ Indique la version du JDK utilisée
- -Dpropriété=valeur
 - ▶ Permet de passer des propriétés (préférences) au programme
- -verbose
 - ▶ Affiche des informations pendant l'exécution
- -help
 - ▶ Affiche la liste des options (d'autres options sont disponibles)

Fichiers jar : Java ARchive

- Format compressé (ZIP) pour déployer les applications
 - ▶ Sources et/ou bytecode et ressources
 - ▶ Descripteur du contenu : META-INF/MANIFEST.MF
 - ▶ Extension : .jar
- Exécution :
 - ▶ `java -jar nomArchive.jar`
 - ▶ Peut être associée à l'extension .jar sous Windows
- Programme dédié : jar.exe ⇔ tar sous Unix
 - ▶ Création
 - `jar cf monAppli.jar Exemple.class`
 - ▶ Extraction
 - `jar xf monAppli.jar`
 - ▶ Tout autre outil capable de manipuler des zips fonctionne aussi

Débogueur : jdb

- Outil en ligne rudimentaire
- Utilise la variable CLASSPATH (option -classpath)
- Utilisation :
 - ▶ compiler les classes
 - ▶ lancer jdb à la ligne de commande : jdb MaClasse
 - ▶ mettre au moins un point d'arrêt : stop in MaClasse.main
 - ▶ exécuter la classe : run MaClasse (s'arrête au 1er point d'arrêt)
 - ▶ avancer : step, next, cont
 - ▶ inspecter : dump monObjet, print monObjet, locals
 - ▶ voir où on en est : list
 - ▶ pour + d'informations : ? ou help

Documentation du code : javadoc

- Commentaires sur plusieurs lignes
 - ▶ Encadrés par /* et */
 - ▶ Pour les détails techniques
- Commentaires sur une seule ligne
 - ▶ Commence par //
 - ▶ Pour les détails au milieu du code
- Commentaires **javadoc**
 - ▶ Encadrés par /** et */
 - ▶ Pour la documentation technique
 - ▶ Syntaxe normalisée
 - ▶ Automatiquement repris par les outils de production de documentation

Documentation du code : javadoc

```
1 package com.toto;
2
3 /**
4  * Ceci est de la JavaDoc sur ma classe.
5  */
6 public class Exemple {
7
8     // Ceci est un commentaire sur une ligne
9
10    /** Ceci est de la JavaDoc sur mon attribut. */
11    private int a;
12
13 /**
14  * Ceci est de la JavaDoc sur ma méthode.
15  *
16  * @param args
17  *         le paramètre de ma méthode
18  */
19 public static void main(String[] args) {
20     // Ceci est un commentaire sur une ligne
21     /* Ceci est
22      un commentaire sur plusieurs
23      lignes.
24      Ceci N'EST PAS de la JavaDoc
25     */
26 }
27 }
```

Générateur de documentation

- Reprend les commentaires encadrés par /** et */
- Ligne de commande
 - ▶ javadoc Fichier.java
 - ▶ Nombreuses options (voir -help)
- Documentation au format HTML
 - ▶ Possibilité d'inclure du code HTML
- Possibilité d'inclure des tags dans les commentaires (@tag)
 - ▶ @author Julie V.
 - ▶ @see com.formation.Client
 - ▶ @version 2.4, 01/01/2022
 - ▶ ...
- La documentation du JDK est elle-même générée par javadoc

Documentation sur Java

➤ Site principal :

<https://www.oracle.com/java/technologies/>

- ▶ API complète des classes de base
- ▶ Outils standards du JDK (javac, java, jdb,...)
- ▶ Exemples
- ▶ Tutoriaux
- ▶ Changements entre les versions
- ▶ ...



58

Travaux pratiques

Réaliser les travaux pratiques suivants:

TP 00

59

Java



eclipse

Voir support

Java



Voir support

60

Les bases du langage

61

Littéraux

Types primitifs

Opérateurs

Java est un langage **fortement typé**

62

- Quand vous utiliser quelque chose <=> ce quelque chose possède un type
- Vous avez indiqué ce type d'une manière ou d'une autre
 - ▶ Explicitement dans 95% des cas
 - ▶ Automatiquement géré par le Java dans 5% des cas
- Exemple de type :
 - ▶ int, String, Object, float, MaClasseDeMonProjet,
- Il n'est pas possible de changer de type sauf cas particulier
 - ▶ Un chiffre (type = int) n'est pas une chaîne de caractère (type = String)

Littéraux

63

- Représentent des valeurs « constantes » ou « littérales »
- Ils ont tous un type intrinsèque
 - ➡ Types primitifs
 - ❑ Nombres
 - ❑ Caractères
 - ❑ Booléens
 - ➡ Les invariants (immutable) ⇔ Ne peut plus changer de valeur
 - ❑ Chaînes de caractères
 - ❑ Wrappers
 - ❑ Les record (à partir de Java 16)

Types primitifs : Nombres entiers

- Signés (préfixe -, optionnellement +)
- Entiers : Mots clefs : **byte**, **short**, **int**, **long**
 - ▶ 14 (est un **int**)
 - ▶ 0xE (l'entier 14 en représentation hexadécimal)
 - ▶ 0o16 (l'entier 14 en représentation octal)
 - ▶ 164L (est un **long**)
 - Vous pouvez aussi faire usage du l minuscule, mais d'un point de vue qualité il est préférable de faire usage du L majuscule
- Attention : Si vous n'indiquez rien devant le chiffre entier, alors c'est un **int**
 - ▶ 14 (est l'int qui vaut 14)
 - ▶ (byte)14 (est le byte qui vaut 14) ⇔ utilisation explicite d'un cast

Type	Taille
byte	8 bits
short	16 bits
int	32 bits
long	64 bits

Types primitifs : Nombres à virgule

- Signés (préfixe -, optionnellement +)
- Séparateur : le point (~~et surtout pas la -~~)
- Flottants : Mots clefs **float** et **double**
 - ▶ 3.14F (représente un float)
 - ▶ 3.14D (représente un double)
 - ▶ 3.14 (représente un double)
 - ▶ 2.1E-3F (représente un float)
 - ▶ 2.1E-3D ou 2.1E-3 (représente des doubles)
- Vous pouvez aussi faire usage du f ou d minuscule, mais d'un point de vue qualité il est préférable de faire usage du F ou D majuscule
- Attention : Si vous n'indiquez rien devant le chiffre à virgule, alors c'est un **double**
 - ▶ 14.0 (est le double qui vaut 14.0)
 - ▶ (float)14.0 ou 14F (est le float qui vaut 14.0) ⇔ utilisation explicite d'un cast

Type	Taille
float	32 bits
double	64 bits

Type primitif : Caractères

- Mot clef char
- Entre apostrophes (et surtout PAS "")

 - ▶ 'A'

- Précédés d'une barre inverse (backslash) si nécessaire (« échappement » ou « escaping »)
 - ▶ \' (apostrophe) '\\ (barre inverse) '\t' (tabulation 9)
 - ▶ '\n' (ligne 10) '\f (page 12) '\r' (retour 13)
 - ▶ '\"' (guillemet) '\b' (arrière 8)
- Avec un code Unicode en hexadécimal à 4 chiffres
 - ▶ '\u00a9' (©) '\u0153' (œ) '\u20ac' (€)
 - ▶ <https://unicode-table.com/> et <http://www.eteeks.com/tips/tip3.html>
- Remarque : En Java, int et char sont amis
 - ▶ Un char est un int et un int est un char

Type	Taille
char	16 bits

Type primitif : Booléen

- Mot clef **boolean**
- Valeurs possibles:
 - ▶ true (le mot clef **true**)
 - ▶ false (le mot clef **false**)
- Attention :
 - ▶ pas de majuscule !
 - ▶ Pas de conversion possible vers les entiers
 - ◀ ~~false n'est pas 0 et 1 n'est pas true~~

Type	Taille
boolean	1 bit

Types primitifs

- Les valeurs minimales et maximales pour chaque élément numérique

Type	Minimum	Maximum
byte	-128	127
short	-32768	32767
int	-2147483648	2147483647
long	-9223372036854775808	9223372036854775807
	Minimum Positif	Maximum Positif
float	1.4E-45	3.4028235E38
double	4.9E-324	1.7976931348623157E308

Types primitifs

- Taille standard sur toutes les plates-formes
 - ➡ Portabilité assurée
- Utilisés
 - ➡ Pour une meilleure efficacité
 - ➡ Pour une certaine compatibilité avec le C

Type	Taille
byte	8 bits
short	16 bits
int	32 bits
long	64 bits
float	32 bits
double	64 bits
char	16 bits
boolean	1 bit

Objet Invariant : Chaînes de caractères

- Objet/classe `java.lang.String`
- Entre guillemets (~~et surtout pas ''~~)
 - ▶ "Bonjour"
- Utiliser la barre inverse si nécessaire
 - ▶ "Une chaîne avec \" (un guillemet)"
 - ▶ "Une chaîne avec \\ (une barre inverse)"
 - ▶ "29,99 \u20ac" (29,99 €)
 - ▶ "Un n\u0153ud" (un nœud)
 - ▶ ...

Objet Invariant : Chaînes de caractères

- En Java 14, vous pouvez écrire vos chaines sur plusieurs lignes en utilisant """ (3 doubles quotes)

```
String query = """  
    SELECT `EMP_ID`, `LAST_NAME` FROM `EMPLOYEE_TB`  
    WHERE `CITY` = 'INDIANAPOLIS'  
    ORDER BY `EMP_ID`, `LAST_NAME`;  
""";
```

- Notez : que le fait d'aller à la ligne sera compté comme un retour à la ligne dans la chaine
 - ➡ Sauf si vous indiquez \'

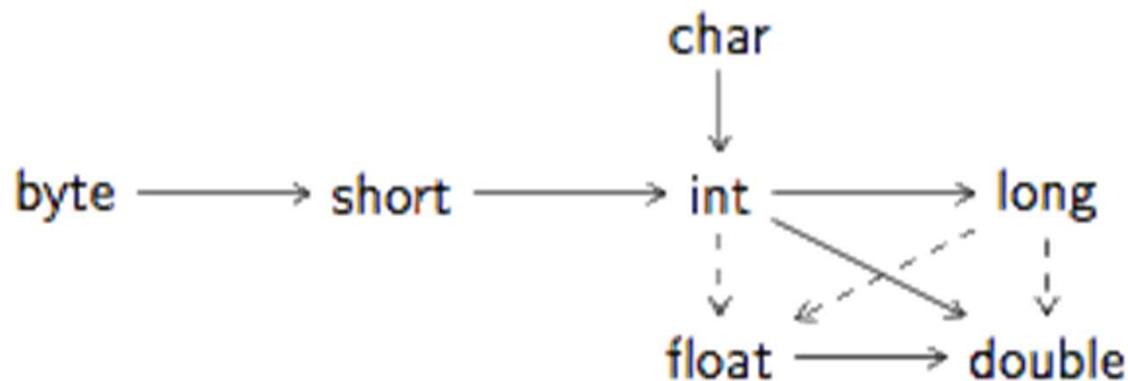
Utilisation

- Dans les initialisations de **variables**
 - ▶ [modificateurs] **type** nom [= valeurInitiale];
 - ❑ **int** i = 1;
 - ❑ **char** a = 'P';
 - ❑ **boolean** trouve = false;
- Dans les expressions (voir plus loin les opérateurs)
 - ❑ k = 10 * i + 1;
 - ❑ a = code + 'A';
 - ❑ i = i + 1;
- Dans les appels de **méthodes**
 - ❑ System.out.println("Bonjour");
 - ❑ effectuerTraitement(a, 100);

Opérateurs

73

- Proviennent du langage C
 - ▶ Pour la compatibilité
 - ▶ Pour la souplesse et les performances
- Assument des conversions implicites
 - ▶ Permettent d'éviter toute perte d'information



Conversion

- D'un point de vue déclaratif :
 - ▶ $5 \leftrightarrow \text{int}$
 - ▶ $6.4 \leftrightarrow \text{double}$
- De manière général, je peux placer quelque chose de petit dans quelque chose de gros (mais pas le contraire)
 - ▶ Par exemple, un short peut se retrouver dans un int sans autre opération (sans cast)
 - ▶ Attention aux chiffres à virgule et à l'utilisation d'opérateurs

```
byte b = 1; // OK - car 1 qui est un int et est aussi < Max valeur de byte - 127  
short s = 5000; // OK - car 5000 qui est un int et est < Max valeur de short  
float f = 5.5; // KO - car 5.5 est un double  
short a = 1, b = 1; // OK - car 1 qui est un int est aussi < Max valeur de short  
short resu = a * b; // KO - car le compilateur va promouvoir a et b en int
```

Conversion

➤ Typage exacte, sinon :

► Si littéral : typage plus gros en restant dans la même famille (boolean, entier, flottant)

□ Sinon Wrapper

- Sinon vararg ...
 - Sinon Object

► Si objet : parent le plus proche en remontant la hiérarchie

Opérateurs arithmétiques

- Les opérations sur les nombres
 - Conversions implicites
 - Valables sur les char (assimilés à leur code)
 - Attention aux débordements (pas d'erreur dans ce cas)
 - + Addition : $7+8=15$, ' A '+3=68
 - - Soustraction : $7-8=-1$
 - * Multiplication : $7*8=56$
 - / Division : $7/8=0$, $7.0/8.0=0.875$
 - % Modulo (reste de la division) : $7\%8=7$, $7.0\%8.0=7.0$
 - ++ Pré/ Post Incrémentation : $i++$, $++index$
 - -- Pré/ Post Décrémentation : $k--$, $--index$

Opérateurs arithmétiques

➤ Concaténation

- ▶ Les chaînes (String) se concatènent avec l'opérateur +

```
int i = 5;  
String result = "indice = " + i;
```

- ▶ Une chaîne ‘+’ n’importe quoi donnera toujours une chaîne

- ▶ Les chaînes (String) se concatènent aussi avec la méthode concat

```
String result = "Bonjour tous le ".concat("monde");
```

➤ Transformation

- ▶ Vous pouvez transformer n’importe quoi en String avec la méthode String.valueOf(xxx);

```
int i = 5;  
String result = String.valueOf(i); // <= "5"
```

Opérateurs arithmétiques

➤ i++ (ou i--)

► Le +/-1 est réalisé après l'instruction

➤ ++i (ou --i)

► Le +/-1 est réalisé tout de suite

```
int counter = 0;  
System.out.println(counter);  
System.out.println(++counter);  
System.out.println(counter);  
System.out.println(counter--);  
System.out.println(counter);
```



0
1
1
1
0

Opérateurs arithmétiques

79

- Exemple complexe (à ne pas reproduire dans la vraie vie) :

```
int x = 3;  
int y = ++x * 5 / x-- + --x;  
System.out.println("x is " + x);  
System.out.println("y is " + y);
```

- Ceci est une question de la certification Java 8

```
y = 4 * 5 / 4 (x passe à 3 après la /) + 2;  
"x is 2"  
"y is 7"
```

Opérateurs relationnels

➤ Opérations de comparaison

- ▶ Valeur booléenne **true** ou **false**
- ▶ Valable pour tous types primitifs
- ▶ == Égal
- ▶ != Différent (non égal)

➤ Opérations de relation d'ordre

- ▶ Valeur booléenne **true** ou **false**
- ▶ Seulement pour les nombres (byte, short, int, char, long float, double)
- ▶ < Plus petit
- ▶ <= Plus petit ou égal
- ▶ > Plus grand
- ▶ >= Plus grand ou égal

Opérateurs logiques

➤ Opérations sur expressions binaires

- ▶ & Et bit-à-bit : $(9 \& 12) = 8$ ($1001 \& 1100 = 1000$)
- ▶ | Ou inclusif : $(9 | 12) = 13$ ($1001 | 1100 = 1101$)
- ▶ ^ Ou exclusif (xor) : $(9 ^ 12) = 5$ ($1001 ^ 1100 = 0101$)

➤ Opération sur les booléens

- ▶ && Conjonction (and) : $(x < 0) \&\& (1/x > -1F)$
- ▶ || Disjonction (or) : $(x == 0) || (1/x > 0.5F)$
- ▶ ! Négation (not) : $!(a == 10)$

Opérateurs logiques

- $X \& Y = \text{true}$ uniquement si X et Y valent true

(And)	$Y = \text{true}$	$Y = \text{false}$
$X = \text{true}$	true	false
$X = \text{false}$	false	false

- $X | Y = \text{false}$ uniquement si X et Y valent false

(OR Inc)	$Y = \text{true}$	$Y = \text{false}$
$X = \text{true}$	true	true
$X = \text{false}$	true	false

- $X ^ Y = \text{true}$ uniquement si X et Y sont différents

(OR Exc)	$Y = \text{true}$	$Y = \text{false}$
$X = \text{true}$	false	true
$X = \text{false}$	true	false

Opérateurs logiques - & - &&

- Si vous utilisez & (ou |) toute l'expression sera analysée

```
boolean t = true;
int i = 0;
if (t | ++i != 0) { // ++i!=0 sera analysée
    System.out.println("ok i=" + i); // i = 1
}
```

- Si vous utilisez && (ou ||) l'expression n'est plus analysée dès que son résultat est **inéluctable**

```
boolean t = true;
int i = 0;
if (t || ++i != 0) { // ++i!=0 ne sera pas analysée
    System.out.println("ok i=" + i); // i = 0
}
```

Opérateurs d'affectations

- Permettent de positionner la valeur d'une variable
 - ▶ Prend la valeur de l'expression à droite et la stocke
 - ▶ = Affectation simple : `a=10; b=(a+1)*3;`
- Affectation combinée avec un opérateur
 - ▶ Forme « var <op>= expression »
 - ▶ Raccourci pour « var = var <op> expression »
 - ▶ Valable pour tous les opérateurs
 - ▶ Exemples

<code>int a += 10;</code>	$\Leftrightarrow a = a + 10;$
<code>int b *= i;</code>	$\Leftrightarrow b = b * i;$
<code>boolean bool = (1/x > 5)</code>	$\Leftrightarrow \text{bool} = \text{bool} (1/x > 5);$

Opérateur conditionnel (ou ternaire)

- Opérateur ternaire permettant des choix ($C?V1:V2$)
 - ▶ (condition) ? valeur pour true : valeur pour false
 - ▶ Simplifie certaines expressions (ne pas en abuser)
 - ▶ Exemples

```
double x = -2.9;
char sign = (x < 0) ? '-' : '+';
String prefix = ((x <= 9) ? "0" : "");
double valeur = (x < 0) ? -x : x;
System.out.println(sign + " " + prefix + " " + valeur); // - 0 2.9
```

Précérence des opérateurs

- expr++, expr--
- ++expr, --expr
- *, /, %
- +, -
- Décalages de bits >>, <<, (>>> unsigned right shift)
- <, <=, >, >=
- ==, !=
- &
- ^
- |
- &&
- ||
- ?:
- Affectations (=, +=, *=, ...)



Simplifier votre écriture avec des parenthèses !

Les objets

87

Bases objet de java

Portée des variables locales

Structures de contrôle

Tableaux

Génériques

Annotations

Classe

- Définit la structure et le comportement d'objets de même type
 - ▶ Structure : **attributs** : les données
 - ▶ Comportement : **méthodes** : les fonctionnalités
- Permet de créer des objets d'un certain type
- Physiquement représenté par un fichier sur le disque dur.
- La classe **Personne** sera obligatoirement dans le fichier **Personne.java**
 - ▶ Une fois compilée elle donnera un fichier **Personne.class**

Classe

89

➤ Exemple : la classe Voiture

```
1 import java.awt.Color;  
2  
3 public class Voiture {  
4     // ----- Les Attributs :  
5     private String marque;  
6     private int km;  
7     private int vitesse;  
8     private double prix;  
9     private Color couleur;  
10    // -----  
11  
12    // ----- Les Méthodes :  
13    public void avancer(int nouvelleVitesse) {  
14        // traitement  
15    }  
16  
17    public void repeindre(Color nouvelleCouleur) {  
18        // traitement  
19    }  
20    // -----  
21 }
```

Le code sera
dans un
fichier
Voiture.java

Toutes les voitures ont

- Un attribut *marque*
- Un attribut *km*
- Un attribut ...

Sur une voiture je peux :

- *avancer*
- *repeindre*

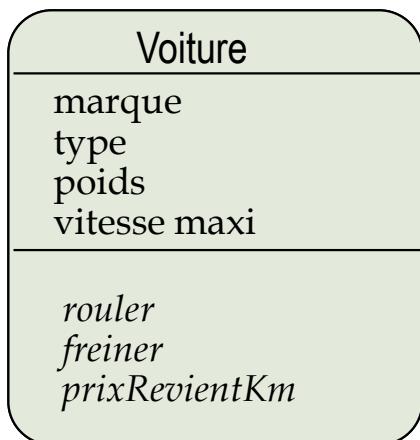
Classe : Nommage

- En Java le nom d'une classe
 - ▶ Commence toujours par une Majuscule
 - ▶ Il n'y a pas de '_' (underscore)
 - ▶ Les noms sont découpés par des Majuscules : ex **MaSuperClasseAMoi**
 - ▶ Il n'y a pas d'espaces
 - ▶ Il n'y a pas de caractères accentués ou autres
 - ▶ Le nom de la classe ne peut pas commencer par un chiffre
- <https://www.oracle.com/java/technologies/javase/codeconventions-contents.html>

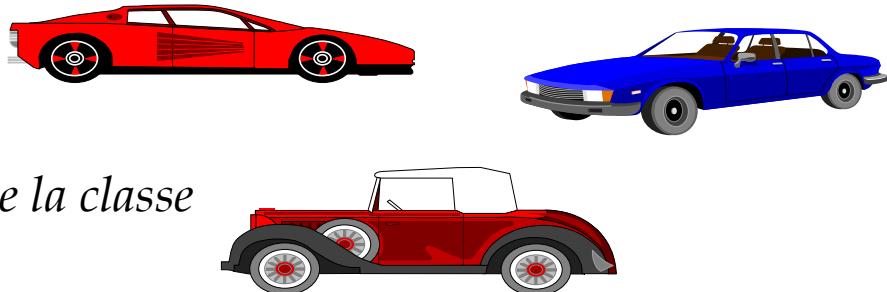
Instance

91

- Objet créé à partir d'une classe
- Possède ses propres valeurs d'attributs
- Partage les comportements (méthodes) des autres instances de la classe



est instance de la classe



Instance

92

➤ Pour créer une instance d'un objet on fait usage du mot clef **new**

```
22 ⊕ public static void main(String[] args) {  
23     Voiture v1 = new Voiture();  
24     v1.repeindre(Color.BLUE);  
25  
26     Voiture v2 = new Voiture();  
27     v2.repeindre(Color.BLACK);  
28  
29     Voiture v3 = new Voiture();  
30     v3.repeindre(Color.PINK);  
31 }
```

- v1, v2, v3 sont
- Des variables locales à la méthode *main*
 - Des instances de la classe Voiture

En mémoire, v1, v2, v3 sont totalement distincts et indépendants

Package

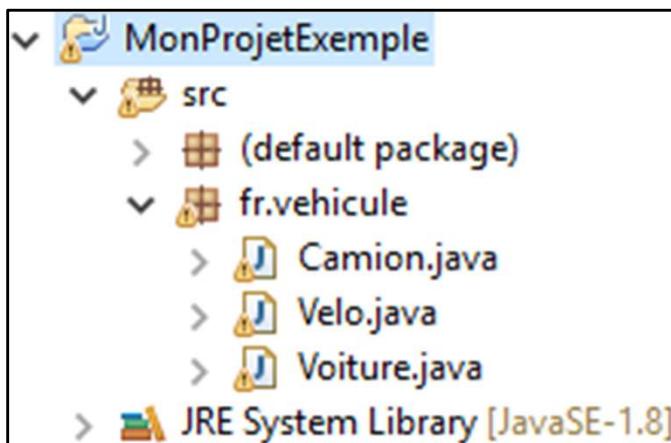
93

- Regroupement fonctionnel (ou technique) de classes
 - ▶ Permet de ranger ses classes
- Niveau de granularité plus grand
- **Physiquement** représenté par **un répertoire** sur le disque dur.
 - ▶ Dans un package je mets des classes ⇔ dans un répertoire je mets des fichiers
- Du coup, vous pouvez avoir deux classes qui ont le même nom, mais elles ne seront pas dans le même package
 - ▶ java.sql.Date
 - ▶ java.util.Date

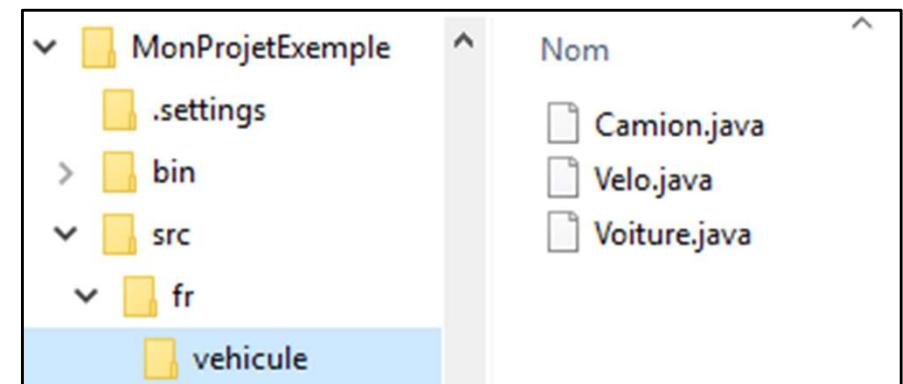
Package

94

➤ Exemple : le package fr.vehicule



Le code sera dans un dossier fr/vehicule



➤ Au niveau de votre code, la première ligne de chaque classe qui se trouve dans le package fr.vehicule sera :

```
package fr.vehicule;

public class Camion {
    // Du code
}
```

```
1 package fr.vehicule;
2
3 public class Velo {
4     // Du code
5 }
```

```
1 package fr.vehicule;
2
3 public class Voiture {
4     // Du code
5 }
```

Package : Nommage

- En Java le nom d'un package
 - ▶ S'écrit toujours en minuscules
 - ▶ Il n'y a pas de ‘_’ (underscore)
 - ▶ Les noms sont découpés par des points: ex mon.super.pkg
 - ▶ Il n'y a pas d'espaces
 - ▶ Il n'y a pas de caractères accentués ou autres

Organisation des packages

- C'est à vous définir vos packages
 - ▶ C'est à vous de définir votre rangement
- Il faut le faire en début de projet
- Il faut réfléchir par regroupement
 - ▶ Métier
 - ▶ Technique
- Il n'y a aucun lien (ou passe-droit) technique entre un package et un sous package
 - ▶ fr est un package, fr.vehicule est un autre package, les deux ne se connaissent pas

Création de classe

97

➤ Une classe

- ▶ Possède un nom **court** et un nom **long**
 - Nom **court** : Velo
 - Nom **long** : fr.vehicule.Velo
- ▶ Une visibilité (représenté par un de ses modificateurs)
 - public, ...
- ▶ Un ensemble d'attributs (0..n) (ses *données*)
- ▶ Un ensemble de constructeur (1..n)
- ▶ Un ensemble de méthodes (1..n) (ses *fonctionnalités*)

➤ Déclaration avec le mot clef **class**

```
[modificateurs] class ExempleDeClasse {  
    attributs  
    constructeurs  
    méthodes  
}
```

Création de classe

- Pour indiquer à quel package une classe appartient il suffit d'indiquer dans le fichier source sur la première ligne de ce dernier :

```
package fr.mon.projet;  
  
[modificateurs] class ExempleDeClasse {  
    attributs  
    constructeurs  
    méthodes  
}
```

- Nom court de la classe : ExempleDeClasse
 - ➡ Ce nom peut porter à ambiguïté (ex: Date)
- Nom long de la classe : fr.mon.projet.ExempleDeClasse
 - ➡ Ce nom est unique et sans ambiguïté (java.util.Date)

Utilisation de classe dans un autre package

99

- Classes connues par défaut (pas besoin de faire un **import**)
 - ▶ classes du package + classes de java.lang
- Sinon, quand vous avez besoin d'une classe qui vient d'un autre packages, vous pouvez
 - ▶ Faire usage de son nom long
 - `java.util.Date maintenant = new java.util.Date();`
 - ▶ Ou importer la classe (préférez cette technique à l'usage des noms longs)
 - plus besoin de faire usage du nom long
 - se fait entre la déclaration de package et la déclaration de classe

```
package fr.mon.projet;

import java.util.Date;

public class ExempleDeClasse {
    private Date dNaissance;
    constructeurs
    méthodes
}
```

dNaissance sera du type `java.util.Date`

Utilisation de classe dans un autre package

- Si vous avez besoin deux classes qui portent le même nom mais sont dans des packages différents

```
package fr.mon.projet;

import java.util.Date;

public class ExempleDeClasse {
    private java.sql.Date dNaissance;
    constructeurs
    méthodes
}
```

dNaissance sera du type java.sql.Date

```
package fr.mon.projet;

import java.util.*;
import java.sql.Date;

public class ExempleDeClasse {
    private Date dNaissance;
    constructeurs
    méthodes
}
```

dNaissance sera du type java.sql.Date
L'import explicite l'emporte sur le nom explicite

Import static

- L'import static a pour objectif de simplifier l'écriture dans votre code lors de l'utilisation de constantes ou de méthodes statiques.

- Exemple, sans import static

- ▶ Math.PI
- ▶ Math.pow

```
package com.sample;

public class RunGenerique {
    public static void main(String[] args) {
        int r =3;
        double air =Math.PI * Math.pow(r, 2);
        System.out.println(air);
    }
}
```

Import static

102

- Exemple, avec un import static (notez le changement sur l'utilisation de PI)

Toujours 'import static'
(et pas ~~static import~~)

```
package com.sample;  
  
import static java.lang.Math.PI;  
  
public class RunGenerique {  
    public static void main(String[] args) {  
        int r = 3;  
        double air = PI * Math.pow(r, 2);  
        System.out.println(air);  
    }  
}
```

Import static

- Vous pouvez aussi importez les méthodes statiques

```
package com.sample;  
  
import static java.lang.Math.pow;  
  
public class RunGenerique {  
    public static void main(String[] args) {  
        int r = 3;  
        double air = Math.PI * pow(r, 2);  
        System.out.println(air);  
    }  
}
```

Import static

- Vous pouvez aussi importez tout ce qui est statique

```
package com.sample;  
  
import static java.lang.Math.*;  
  
public class RunGenerique {  
    public static void main(String[] args) {  
        int r = 3;  
        double air = PI * pow(r, 2);  
        System.out.println(air);  
    }  
}
```

Déclaration des attributs

- [modificateurs] type nom [= valeurInitiale]
- Déclarés de préférence au début
- Attributs (ou variables d'instance)
 - ▶ valeurs différentes pour chaque instance
 - ▶ Chacun la sienne
- Attributs statiques : mot clef **static**
 - ▶ valeurs partagées par les instances et la classe
 - ▶ Tous la même
- Attributs constantes : mot clef **final**
 - ▶ La valeur ne peut plus changer

```
package fr.copera.entity;  
import java.util.Date;  
public class Personne {  
    private Date dNaissance;  
}
```

```
package fr.copera.service;  
public class Bilan {  
    private static float tva = 20F;  
}
```

```
package fr.copera.service;  
public class Bilan {  
    private final int maxIteration = 2;  
}
```

Initialisation des attributs

- Les attributs peuvent être initialisés lors de leur déclaration

```
package fr.copera.entity;  
public class Personne {  
    private int age = 20;  
}
```

- Les attributs peuvent être initialisés dans les constructeurs
- Les attributs peuvent être initialisés dans des blocs de classe
- **Sinon**, tous les attributs non initialisés le seront automatiquement à
 - ▶ null pour les objets
 - ▶ 0 pour les nombres
 - ▶ false pour les booléens
 - ▶ '\u0000' (caractère NUL) pour les caractères

Référencer un attribut

➤ Utilisation du caractère . (point)

➤ Attribut (= Variable d'instance)

► Dans la classe elle-même

□ Dans ses méthodes par exemple

► Ailleurs

□ Dans une méthode *static* ou appartenant à un autre objet

```

1 package fr.vehicule;
2
3 import java.awt.Color;
4
5 public class Voiture {
6     // ----- Les Attributs :
7     private String marque;
8     private int km;
9     private int vitesse;
10    private double prix;
11    private Color couleur;
12    // -----
13
14    // ----- Les Méthodes :
15    public void avancer(int nouvelleVitesse) {
16        vitesse = nouvelleVitesse;
17        this.vitesse = nouvelleVitesse;
18    }
19
20    public void repeindre(Color nouvelleCouleur) {
21        // traitement
22    }
23    // -----
24
25    public static void main(String[] args) {
26        Voiture v1 = new Voiture();
27        v1.vitesse = 50;
28        v1.repeindre(Color.BLUE);
29
30        Voiture v2 = new Voiture();
31        v2.repeindre(Color.BLACK);
32
33        Voiture v3 = new Voiture();
34        v3.repeindre(Color.PINK);
35    }
36 }
```

Référencer un attribut

- Attribut **statique** (sera en *italique* dans Eclipse)
 - ➡ **NomDeLaClasse.nomAttributStatic**

```
1 package fr.vehicule;
2
3 public class Voiture {
4     // ----- Les Attributs :
5     private static int nbRoue = 4;
6     private int vitesse;
7     // -----
8
9     // ----- Les Méthodes :
10    public void avancer(int nouvelleVitesse) {
11        this.vitesse = nouvelleVitesse;
12        // Ou : vitesse = nouvelleVitesse;
13    }
14
15    public void faireQQc() {
16        // traitement
17        int tmp = Voiture.nbRoue * 25;
18        // Ou, mais moins propre : int tmp = nbRoue * 25;
19    }
20    // -----
21
22    public static void main(String[] args) {
23        Voiture v1 = new Voiture();
24        v1.vitesse = 50;
25        System.out.println(Voiture.nbRoue);
26        // Ou, mais moins propre : System.out.println(v1.nbRoue);
27    }
28 }
```

Les variables locales

109

- Une variable locale est toujours typée
- C'est à vous de choisir son nom, elle devra commencer par une minuscule (puis camel case), vous ne pouvez pas commencer par un chiffre.
- Destruction quand hors de portée : C'est-à-dire quand on sort du bloc ⇔ on atteint '}'
- Un bloc est délimité par {...}
- Variable locale : déclarée dans un bloc et utilisable uniquement dans le même bloc

```
1 public class Ok {  
2  
3     public static void main(String[] args) {  
4         double x = -2.9;  
5         {  
6             int i = 5;  
7             // Ici x et i existe  
8         }  
9         // Ici x existe  
10        // Ici i n'existe plus  
11    }  
12 }
```

Les paramètres de méthodes

110

- Un paramètre est toujours typé
- C'est à vous de choisir son nom, il devra commencer par une minuscule (puis camel case)
- Un paramètre de méthode (ou un argument de méthode) n'est manipulable sous son nom que dans la méthode où il est déclaré.

```
1 package fr.vehicule;
2
3 public class Voiture {
4     // ----- Mon Attribut :
5     private int vitesse;
6     // -----
7
8     // ----- Ma Méthode :
9     public void avancer(int nouvelleVitesse) {
10        // nouvelleVitesse est un parametre de la methode avancer
11        // Ici nouvelleVitesse existe
12        this.vitesse = nouvelleVitesse;
13    }
14    // Ici nouvelleVitesse n'existe plus
15
16    // -----
17
18    public static void main(String[] args) {
19        // v1 est une variable locale de la methode main
20        // v1 est une instance de la classe Voiture
21        Voiture v1 = new Voiture();
22        // uneVitesse est une variable locale de la methode main
23        int uneVitesse = 25;
24        // uneVitesse est passee comme valeur de parametre a la methode avancer
25        v1.avancer(uneVitesse);
26        // Combien vaut l'attribut vitesse de l'instance v1 ?
27        System.out.println(v1.vitesse);
28    }
29 }
```

Variable, Attribut, Paramètre

111

- Une **variable**, un **attribut**, un **paramètre** peut être de type objet

```
1 package fr.vehicule;
2
3 import java.awt.Color;
4
5 public class Voiture {
6     // Un attribut de type java.lang.String
7     private String marque;           marque est un attribut
8
9     // Un parametre de type java.awt.Color
10    public void repeindre(Color nouvelleCouleur) {   nouvelleCouleur est un paramètre
11        // Traitement
12    }
13
14    public static void main(String[] args) {
15        // v1 est une variable locale de la methode main de type Voiture
16        // v1 est une instance de la classe Voiture
17        Voiture v1 = new Voiture();                  v1 est une variable
18    }
19 }
```

Variable, Attribut, Paramètre

➤ Contraintes techniques sur le nom

- ▶ Ne peut pas commencer par un chiffre
- ▶ Ne peut pas être identique à un mot clef du langage (if, for, ...)
- ▶ Ne peut pas contenir autre chose que des chiffres, des caractères *classiques* ou \$ ou _
 - Pas de @, *, +, -, ... par exemple

```
int $ = 3;  
int _toto = 64;  
int 3six = 36; // Ne doit pas commencer par un chiffre  
int six3 = 63;  
int tata$ = 44;  
int a.b = 36; // Pas de .,*,+,...  
int $tata = 44;  
int titi_toto = 75;
```

Variable locale en Java 10+

113

Java 10+

- Il est possible de faire usage du mot clef **var**
 - ➡ Le typage n'est plus obligatoire (côté déclaratif)

```
public void maMethodeFaire(int a, int b) {  
    var age = 20;  
    var list = new ArrayList<String>();  
    var maChaine = "Bonjour tour le monde";  
}
```

- **ATTENTION** : ceci n'est valable que dans le cas des **variables locales** (et donc **pas** pour les *paramètres* ou *attributs*)

Création d'objet/instance

114

➤ Mot-clé : **new**

- ▶ doit être suivi par le nom de la classe et ()
- ▶ exemple : **new Voiture()**
 - crée un objet de type Voiture
 - initialise ses attributs en respectant la règle

➤ Voiture() est en fait une méthode particulière de la classe Voiture que l'on nomme un constructeur (nous y reviendrons)

Méthode

- Appartient à un objet
- Code qui permet de regrouper un comportement
- Peut retourner un résultat, prendre des paramètres, lever des exceptions

```
package fr.copera.service;  
public class Bilan {  
    public void calculer() {  
        // Mon Code  
    }  
}
```

- Le nom d'une méthode doit être choisi avec soin (en général on prend un **verbe**)

Déclaration des méthodes (1)

```
[modificateurs] typeRetour nom(typeArg1 nomArg1, typeArg2 nomArg2, ...) {
    code
}
```

- Méthodes d'instance
 - ➡ appliquées uniquement aux instances

```
package fr.copera.service;
public class Bilan {
    public void calculer() {
        // Mon Code
        // Peut utiliser tous les attributs et
        // méthodes de la classe
    }
}
```

- Méthodes statiques
 - ➡ appliquées à la classe (ou à ses instances)

```
package fr.copera.service;
public class Bilan {
    public static void montrer() {
        // Mon Code
        // NE peut utiliser que les
        // attributs et méthodes static
        // de la classe
    }
}
```

Déclaration des méthodes (2)

- Arguments / Paramètres
 - ▶ passés par valeur pour les types primitifs
 - ▶ Passés par référence pour les objets
- Si retour d'une valeur à la fin d'une méthode
 - ▶ mot-clé **return** suivi de l'élément
 - ▶ éventuellement **return null;**
 - ▶ pas de retour par défaut
- Si pas de retour attendu
 - ▶ type de retour void
 - ▶ l'expression **return;** peut être utilisée pour une sortie explicite avant la fin de la méthode

Exécution d'une application

- Une classe peut être exécutée
 - ➡ si elle comporte une méthode particulière : **main**

```
public static void main(String[] args) {  
    // code à exécuter  
}
```

- ou si c'est une sous-classe d'Applet
- Cette notion n'existe pas en Web
 - ➡ Il n'y a pas d'exécution de classe mais exécution de méthodes en fonction des routes

Appeler une méthode

- Utilisation du caractère . (point) et des parenthèses après son nom.
- Méthodes d'instance
 - ▶ Aura comme sujet une instance :
 - ❑ obj.calculerTaux(5);
 - appelle la méthode calculerTaux sur l'instance obj en passant la valeur 5 comme paramètre
 - ❑ this.envoyer();
 - appelle la méthode envoyer sur l'instance courante sans passer de paramètre
 - Méthodes statiques
 - ▶ Aura comme sujet une classe (ou une instance)
 - ▶ Sera en *italique* dans Eclipse
 - ❑ ExempleDeClasse.transformer();
 - ❑ obj.transformer(); (obj instance de ExempleDeClasse)



120

Travaux pratiques

Réaliser les travaux pratiques suivants:

TP 02 Partie 1

Constructeur

- Méthode spéciale sans type de retour
 - ➡ Ne retourne rien, même pas void
- Permet d'initialiser un objet selon vos besoins
- Appelé lors de la création d'une instance via le mot clef **new**

Définition d'un constructeur

- Cette méthode particulière porte **toujours** le même nom que la classe
 - On garde la majuscule au début
- Elle peut prendre un nombre quelconque d'arguments (0..n)
- Vous pouvez réaliser plusieurs constructeurs par classe
 - Cependant, les signatures devront être différentes (pas le même nombre ou le même type d'arguments)
- **JAMAIS** de valeur de retour : Même pas void

Constructeur par défaut

- Constructeur sans argument
- Défini **implicitement** dans chaque classe
 - ▶ sans traitement spécifique autre que la création d'instance
 - ▶ peut-être redéfini pour ajouter des traitements spécifiques
- **Disparaît** dès qu'un autre constructeur est défini dans la classe
 - ▶ il faut, si nécessaire, le redéfinir explicitement

Exemple de constructeurs

124

➤ Deux définitions de constructeurs dans une classe

```
public class CartePostale {  
    private Photo image;  
    private String texte;  
    private Adresse adresse;  
  
    // Définition du constructeur par défaut  
    public CartePostale () {  
        texte = "Nous passons de bonnes vacances";  
    }  
  
    // Second constructeur avec un paramètre  
    public CartePostale(String pTexte) {  
        this.texte = pTexte;  
    }  
}
```

Appel d'un constructeur

125

➤ Pour créer un objet

```
CartePostale carte = new CartePostale();
CartePostale carteRusse = new CartePostale("Bons baisers de Russie");
```

➤ Par un autre constructeur (mot-clé this)

```
public CartePostale () {
    // Chainage des constructeurs
    // Du plus bête vers le plus intelligent
    this("Nous passons de bonnes vacances");
}
```

Destruction d'objet

➤ Automatique en Java

- ▶ une fois que l'objet n'est plus référencé
- ▶ pas de destructeur (C++)

➤ Effectuée par le garbage collector

- ▶ libère la mémoire
- ▶ processus de faible priorité
- ▶ Ce dernier appellera la méthode finalize() lorsque l'objet est sur le point d'être détruit.

Destruction d'objet

- Depuis Java 9, vous ne devez plus utiliser (ou *overrider*) la méthode *finalize* (elle est par ailleurs *deprecated* <=> à ne plus utiliser)
- Vous devez faire usage d'un des objets suivants :
 - ▶ `java.lang.ref.Cleaner`
 - ▶ `java.lang.ref.PhantomReference` (ou autre)



128

Travaux pratiques

Réaliser les travaux pratiques suivants:

TP 02 Partie 2 et 3

Tableaux

129

➤ Le tableau est un objet
(un `java.lang.Object`)

- ▶ Ce n'est pas un type primitif
- ▶ Il faudra l'instancier avant de l'utiliser

➤ C'est une structure de **taille fixe** qui contient des éléments de même type

- ▶ types primitifs ou objets

```
14 // args est un tableau de String
15 // args est un parametre de la methode main
16 public static void main(String[] args) {
17     // tabNoms est une variable locale de la methode main
18     // tabNoms est un tableau de 5 cases de type String
19     // Par defaut, null dans les 5 cases => tabNoms.length = 5
20     String[] tabNoms = new String[5];
21
22     // tabAges est une variable locale de la methode main
23     // tabAges est un tableau de 8 cases de type int
24     // Par defaut, 0 dans les 8 cases => tabAges.length = 8
25     int[] tabAges = new int[8];
26
27     // matrice est un tableau a deux dimensions
28     // <=> matrice.length = 10 et matrice[0].length = 5
29     int[][] matrice = new int[10][5];
30
31     // Vous pouvez aussi initialiser les tableaux via :
32     int[] tabAgesBis = { 0, 10, 25, 32 };
33     // Ici tabAgesBis a une taille de 4 => tabAgesBis.length = 4
34 }
```

Tableaux (2)

130

- Accès aux éléments par leur indice (l'index est un entier positif)
 - ▶ commence à l'indice 0
 - ▶ opérateur d'accès : []
 - ▶ longueur du tableau récupérable via l'attribut length
 - permet de faire des boucles

```
String[] chaines = new String[3];
chaines[0] = "1ers mots";
System.out.println(chaines[0]);      // affiche: 1ers mots
System.out.println(chaines.length); // affiche: 3
```

- Si accès hors des limites du tableau, erreur à l'exécution
 - ▶ `ArrayIndexOutOfBoundsException`

Structures de contrôle

➤ Exécutions conditionnelles

- ▶ if else
- ▶ switch

➤ Itérations

- ▶ for
- ▶ while
- ▶ do-while

if-else

132

➤ La condition if-else

```
if (condition) {  
    instructions;  
}
```

```
if (condition) {  
    instructions;  
} else {  
    instructions;  
}
```

```
if (condition) {  
    instructions;  
} else if (condition2) {  
    instructions;  
} else if (condition3) {  
    instructions;  
} else {  
    instructions;  
}
```

Exemple sur if-else

➤ Exemple

```
if (i == 0) { // retourne un boolean
    System.out.println("i vaut 0");
} else if (i > 0) {
    System.out.println("i est positif");
} else {
    System.out.println("i est négatif");
}
```

switch

134

- La condition **switch**

- La valeur de test peut être

- ▶ une expression renvoyant un entier
(ou wrapper d'entier)
- ▶ Un type énuméré à partir de Java 5
- ▶ Une String à partir de Java 7

- Mot-clef **break** : sortie switch

- Mot-clef **default** : instruction effectuée par défaut, optionnel, peut se positionner où l'on veut (début, milieu, fin)

```
switch (valeur) {  
    case (valeur1) :  
        instructions;  
        break;  
    case (valeur2) :  
        instructions;  
        break;  
    default :  
        instructions;  
}
```

switch

135

```
int maVariable = 5;
switch (maVariable) {
case 4:
    System.out.println("Vaut 4");
case 5:
    System.out.println("Vaut 5");
case 6:
    System.out.println("Vaut 6");
    break;
case 7:
    System.out.println("Vaut 7");
    break;
case 8:
    System.out.println("Vaut 8");
    break;
default:
    System.out.println("Vaut " + maVariable);
}
```

Vaut 5
Vaut 6

➤ Un Switch = s'exécute dans le **case** si il y en a un, sinon passe dans le **default** si il y en a un, attend de trouver l'accolade fermante du switch ou un **break** (ou return)

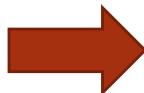
```
int maVariable = 4;
switch (maVariable) {
case 4:
    System.out.println("Vaut 4");
case 5:
    System.out.println("Vaut 5");
case 6:
    System.out.println("Vaut 6");
    break;
case 7:
    System.out.println("Vaut 7");
    break;
case 8:
    System.out.println("Vaut 8");
    break;
default:
    System.out.println("Vaut " + maVariable);
}
```

Vaut 4
Vaut 5
Vaut 6

switch

136

```
int x = 10;  
switch (x % 4) {  
default:  
    System.out.println("Pas modulo 4");  
case 0:  
    System.out.println("Modulo 4");  
case 1:  
    System.out.println("Reste 1");  
}
```



```
Pas modulo 4  
Modulo 4  
Reste 1
```

- Il est important de noter l'endroit où l'on place son **default**
- Un Switch = s'exécute dans le **case** si il y en a un, sinon passe dans le **default** si il y en a un, attend de trouver l'accolade fermante du switch ou un **break** (ou return)

switch

137

- En Java 12, le switch peut aussi retourner un résultat via/grâce au mot clef *break*

```
String x = "10";
int resu = switch (x) {
    case "0" :
        break 0;
    case "10":
        break 10;
    default:
        break 25;
};
```

Le default devient obligatoire

Le ';' précise que c'est une instruction

switch

138

- En Java 13, le mot clef *break* est remplacé par le nouveau mot clef *yield*

```
String x = "10";
int resu = switch (x) {
    case "0" :
        yield 0;
    case "10":
        yield 10;
    default:
        yield 25;
};
```

Le default devient obligatoire

Le ';' précise que c'est une instruction

switch

139

- En Java 12 et 13, vous pouvez faire usage du '->' à la place de l'expression de retour

```
String x = "10";
int resu = switch (x) {
    case "0" -> 0;
    case "10" -> 10;
    default -> 25;
};
```

switch

140

- Le mot clef **yield** n'est pas toujours obligatoire, par exemple :

```
System.out.println(  
    switch (k) {  
        case 1 -> "one";  
        case 2 -> "two";  
        default -> "many";  
    }  
);
```

switch

141

- Vous pouvez faire usage de pattern, de la valeur null,

```
switch (s) {  
    case null          -> System.out.println("Unknown!");  
    case "Java 11", "Java 17" -> System.out.println("LTS");  
    default           -> System.out.println("Ok");  
}
```

switch

142

- Vous pouvez faire usage d'un instanceof masqué

```
static void testTriangle2(Shape s) {  
    switch (s) {  
        case null -> {}  
        case Triangle t && (t.calculateArea() > 100) ->  
            System.out.println("Large triangle");  
        case Triangle t -> System.out.println("Triangle");  
        default -> System.out.println("Unknown!");  
    }  
}
```

for

143

- **Définition** : une boucle contient une série d'instructions exécutées jusqu'à ce qu'un résultat particulier soit obtenu ou qu'une condition prédéterminée soit remplie.
- Les boucles permettent de réutiliser des séries d'instructions et permettent ainsi de limiter le nombre d'instructions.

for

144

- La boucle n'est pas exécutée si la condition est fausse au départ mais l'initialisation est toujours effectuée
- L'incrémentation est une expression qui est exécutée après chaque itération
- Nouvelle syntaxe possible depuis Java 5 (**foreach**)

for

145

➤ Syntaxe :

- ▶ Initialisation, condition de continuation et incrémentation sont facultatifs

```
for (initialisation(s); condition; incrémentation(s)) {  
    instruction(s);  
}
```

➤ Pour un *foreach*

```
for (type variable : collection) {  
    instruction(s);  
}
```

Java 5

Exemple de boucle for

➤ Boucle classique

```
for (int i = 0; i < 10; i++) {  
    // calcul du carré de i  
    int carre = i * i;  
  
    // affichage du résultat  
    System.out.println("le carré de " + i + " est " + carre);  
}
```

➤ Boucle sur un tableau / collections (sans indice)

```
String[] noms = {"Frédéric", "Valérie", "Bernard", "Arlette"};  
for (String unNom : noms) {  
    // affichage de chaque nom  
    System.out.println("un nom: " + unNom);  
}
```

Java 5

Exemple de boucle for

- Boucle for avec plusieurs initialisations et incrémentations

```
for (int i = 0, j = 0; i < 10 && j < 10; i++, j++) {  
    int sol = i * j;  
  
    // affichage du résultat  
    System.out.println("Résultat =" + sol);  
}
```

- Chaque élément est séparé par une ',' (virgule)

while et do-while

148

➤ Différence entre while et do-while

- Les instructions dans le do-while sont exécutées au moins une fois

```
while (condition) {  
    instructions;  
}
```

```
do {  
    instructions;  
} while (condition);
```

Exemple de boucle while

149

➤ Exemple

```
int i = 10;
while (i > 0) {
    // calcul du carré de i
    int carre = i * i;

    // affichage du résultat
    System.out.println("le carré de " + i + " est " + carre);

    // décrémentation de 1
    i--;
}
```

break et continue

- Il est possible d'agir sur le déroulement d'une boucle :
 - ▶ le mot-clef **break** permet de sortir de la boucle.
 - ▶ Le mot-clef **continue** permet d'aller directement à l'évaluation suivante de la condition.
- Ils ne sont utilisables que dans une boucle (le break est aussi utilisable dans un switch)
- Ne pas confondre **break** et **return**
 - ▶ **break** : pour casser une boucle (ou un switch)
 - ▶ **return** : pour casser une méthode

Exemple break et continue

➤ Exemple

```
int i = 10;
while (i > 0) {
    if (i == 3) {
        i--;
        continue;
    }
    if (i == 2) {
        break;
    }
    System.out.println("i=" + i);
    i--;
}
```

Style de codage

- Importance des minuscules/majuscules
- Noms de **packages** : Tout en minuscules, séparés par des ‘.’
- Noms de **classes** : Commencent par une majuscule, chaque mot commence par une majuscule (CamelCase)
- Noms de **variables** et de **méthodes** : Commencent par une minuscule, chaque mot commence par une majuscule
- Noms de **constantes statiques** (final + static): Tout en majuscules, mots séparés par le caractère ‘_’

Style de codage (2)

- Respecter les normes de commentaires pour la génération avec javadoc
- Utiliser systématiquement les {} y compris pour les blocs ne comportant qu'une seule ligne
 - if / for / while / do
- Définir une classe unique par fichier source
- Ne pas abréger les noms de variables ou de méthodes
- <https://www.oracle.com/java/technologies/javase/codeconventions-contents.html>

Rappel – mot clef static

- Le mot clef **static** peut se placer sur
 - ➡ Un *attribut*
 - ❑ Il appartient à la famille = à la classe (et non plus à l'instance)
 - ➡ Une *méthode*
 - ❑ Elle appartient à la famille = à la classe (et non plus à l'instance)
 - ❑ Elle ne peut utiliser que des éléments static
 - ➡ Une *classe* (dans le cas des inner class static)

Rappel – mot clef static

- static **NE VEUT PAS DIRE** constante
 - C'est le mot clef final qui signifie constante
- static **A UNE ODEUR DE** variable globale
 - Car c'est une information plus largement disponible

Rappel – mot clef final

- Le mot clef **final** peut se placer sur
 - ▶ Un attribut
 - Il ne peut plus changer de valeur (ne peut plus être réaffectée)
 - Il doit être initialiser sur sa ligne de déclaration ou les constructeurs
 - ▶ Une variable
 - Elle ne peut plus changer de valeur (ne peut plus être réaffectée)
 - ▶ Un paramètre
 - Il ne peut plus changer de valeur (ne peut plus être réaffectée)
 - ▶ Une méthode
 - Elle ne peut plus être *override* par ses enfants
 - ▶ Une classe
 - Elle ne peut plus avoir d'enfant
- final **SIGNIFIE** constante

Rappel – final static

- Quand on fait usage de **final et static** sur un attribut
 - ❑ Il ne peut plus changer de valeur (ne peut plus être réaffectée)
 - ❑ Il doit être initialisé sur sa ligne de déclaration ou les constructeurs
 - ❑ Il appartient à la famille = à la classe (et non plus à l'instance)
 - ❑ Il est en MAJUSCULE_SEPARE_PAR DES_UNDERSCORES
- **final static ↔ CONSTANTE_GLOBALE**
 - ➡ Les deux mots ensemble

Les énumérations

158

- Un type énuméré est un type dont la valeur fait partie d'un jeu de constantes définies

```
public class TypeCarte {  
    public static final int PIQUE = 1;  
    public static final int COEUR = 2;  
    public static final int CARREAU = 3;  
    public static final int TREFLE = 4;  
    ...  
}
```

- Les inconvénients
 - ▶ Elle n'est pas « type-safe »
 - Pour une variable qui désigne un type de carte, il est possible de lui donner une valeur qui ne fait pas partie de l'énumération (n'importe quel entier)
 - ▶ Elle ne fournit pas de correspondance entre la valeur de la constante et sa description

Les énumérations (2)

➤ Opération sans contrôle de type

```
public class Carte {  
    public String nom;  
    public int type;  
}
```

```
Carte uneCarte = new Carte(); // création d'une nouvelle carte  
  
uneCarte.type = TypeCarte.COEUR; // ok  
  
uneCarte.type = 123; // Techniquement possible,  
                    // mais fonctionnellement incorrecte
```

Le type enum (Java 5)

➤ Depuis Java 5 la notion de type énuméré « type-safe » a été ajoutée

➤ Mot clé **enum**

```
public enum TypeCarte {  
    PIQUE, COEUR, CARREAU, TREFLE  
}
```

➤ Avantages

- Il est "type-safe"
- Il est possible d'ajouter des attributs et des méthodes à chaque valeur

Avec énumérations

- Mon code devient type-safe

```
public class Carte {  
    public String nom;  
    public TypeCarte type;  
}
```

```
Carte uneCarte = new Carte(); // création d'une nouvelle carte  
  
uneCarte.type = TypeCarte.COEUR; // ok  
  
uneCarte.type = 123; // Pas possible, ne compilera pas
```

Exemple de déclaration

➤ Une énumérations avec des attributs

Java 5

```
public enum Vehicule {  
    VÉLO(2, "Bicycle"),  
    VOITURE(4, "Car");  
  
    private final int nombreDeRoues;  
    private final String libelleAnglais;  
  
    Vehicule(int nombreDeRoue, String libelleAnglais) {  
        this.nombreDeRoues = nombreDeRoue;  
        this.libelleAnglais = libelleAnglais;  
    }  
  
    public String getLibelleAnglais() {  
        return libelleAnglais;  
    }  
  
    public int getNombreDeRoues() {  
        return nombreDeRoues;  
    }  
}
```

Arguments variables (Java 5)

- Depuis Java 5 la notion d'arguments variables a été ajoutée
- Permet de déclarer une méthode sans savoir le nombre d'arguments qu'elle va recevoir
 - ▶ Concrètement : raccourci pour passer un tableau en paramètre
- Contrainte
 - ▶ Un seul varargs par méthode
 - ▶ Un paramètre varargs est toujours le dernier de la signature
- Déclaration avec ... (trois petits points)
 - ▶ type... nomVariable

```
public void afficher(int valeur, String... args) {  
    System.out.println("valeur entière: " + valeur);  
    for (String arg : args) {  
        System.out.println("valeur string: " + arg);  
    }  
}
```

Java 5

Annotations (Java 5)

- Les annotations permettent d'ajouter des méta-données au code source
- Syntaxe
 - ▶ `@NomAnnotation(...)`
- Exemple d'utilisation
 - ▶ Informer le compilateur d'ignorer un avertissement
 - `@SuppressWarnings("unused")`
 - ▶ Déclarer que l'on redéfinit une méthode
 - `@Override`
 - ▶ Déclarer qu'une classe ne doit plus être utilisée
 - `@Deprecated`
- Il est possible d'écrire ses propres annotations

Annotations (Java 5)

- Une annotation peut s'appliquer sur
 - Une Classe
 - Un Package
 - Une Méthode
 - Un Attribut
 - Un Paramètre
- Une annotation peut être héritée ou non
- Une annotation peut avoir des paramètres

Types génériques (Java 5)

➤ Problématique

- ▶ Un objet Avion peut transporter soit des objets de type Passager, soit des objets de type Marchandise mais pas les deux en même temps
- ▶ Passager et Marchandise n'ont rien en commun
 - Une classe parente : Object

➤ Avion doit proposer des méthodes pour être chargé et déchargé

- ▶ Pour accepter autant les passagers que les marchandises, les méthodes doivent travailler avec la classe Object
 - Nécessite des conversions
 - Pas de contrôle visant à vérifier que l'on ne mélange pas les passagers et la marchandise dans un même avion
 - Risque d'erreur lors de l'exécution !

Cas d'exemple

➤ Implémentation d'un cas d'exemple sans type générique

```
Passager passager1 = new Passager("Toto");
Passager passager2 = new Passager("Titi");
Marchandise marchandise = new Marchandise("paquet1");
Avion avion = new Avion();
avion.charger(0, passager1);
avion.charger(1, passager2);
avion.charger(2, marchandise); // autorisé !

// récupération de l'élément 2, nécessite une conversion explicite (cast)
// autorisé à la compilation, erreur à l'exécution !
Passager passager = (Passager) avion.decharger(2);
```

➤ Les erreurs de type ne sont détectées qu'à l'exécution du programme

- ➡ Donc trop tard...

Types génériques (Java 5)

- Solution: typer la classe Avion !
- Depuis Java 5, les types génériques ont été ajoutés dans ce but
- Possibilité de paramétriser un type (classe, interface) ou une méthode avec un ou plusieurs types dit génériques
 - ➡ Une liste d'entiers, un dictionnaire de chaînes de caractères, un comparateur de date, ...
- Syntaxe avec <...>
 - ➡ type< identificateur[, identificateur2, ...]>
- Permet au compilateur d'effectuer des contrôles de type

Cas d'exemple

169

- Implémentation d'un cas d'exemple avec type générique

```
Passager passager1 = new Passager("Toto");
Passager passager2 = new Passager("Titi");
Marchandise marchandise = new Marchandise("paquet1");
Avion<Passager> avion = new Avion<Passager>();
avion.charger(0, passager1);
avion.charger(1, passager2);

// la ligne suivante ne compile pas !
avion.charger(2, marchandise);

// récupération de l'élément 1, ne nécessite pas de conversion explicite
Passager passager = avion.decharger(1);
```

Java 5

- Les erreurs de type sont détectées à la compilation
- Le code est plus lisible

Compatibilité avec le code existant

- Cohabitation entre les types génériques et les types bruts
 - Le compilateur Java 5 compile le code mais émet des avertissements
- Comment supprimer les avertissement ?
 - Modifier le code pour utiliser les types génériques
 - Suppression des avertissement avec l'annotation
 @SuppressWarnings("unchecked")
 - Désactiver le contrôle de type au niveau du compilateur
 - –Xlint:unchecked for details.
 - Très fortement déconseillé !

Définition d'un type générique

➤ Définition d'un type générique

```
public class Avion<E> {  
    public void charger(int index, E charge) {  
        ...  
    }  
  
    public E decharger(int index) {  
        ...  
    }  
}
```

Java 5

- ❑ E se manipule comme un type "normal"
- Le type générique E peut s'utiliser comme un type normal au sein de la classe
- Beaucoup d'autre possibilités d'utilisation
 - ▶ Cela devient généralement vite complexe...
 - ▶ <https://docs.oracle.com/javase/tutorial/java/generics/>

Les méthodes de java.lang.Object

172

- Tous les objets héritent de java.lang.Object
- Tous les objets ont les méthodes suivantes qui sont surchargeables :
 - ▶ **toString()** : Donne une représentation de l'objet sous forme de chaînes de caractères. Par défaut donne le nom long de la classe@le hash code en hexadécimal
 - ▶ **equals(Object o)** : Compare l'instance courante avec un autre objet. Par défaut fait == (c.a.d compare les références mémoires)
 - ▶ **hashCode()** : Retourne un entier qui représente l'objet.
 - ▶ **clone()** : Qui fabrique un clone de l'objet. Par défaut lève une exception si la classe n'implémente pas l'interface Cloneable. Sinon fait une shallow copy.
 - ▶ **finalize()** : Méthode appelées par le garbage collector juste avant de libérer la mémoire = déprécié depuis Java 9

Les méthodes de java.lang.Object

- Tous les objets ont les méthodes suivantes qui ne sont pas surchargeables :
 - ▶ **getClass()** : Donne un objet Class qui représente la classe de l'objet
 - ▶ **wait()** : Bloque la thread courante et attend que l'on fasse un appel à notify ou notifyAll
 - ▶ **notify(), notofyAll()** : Débloque la thread courante qui attendait sur un wait

La méthode `toString`

174

- Donne une représentation de l'objet sous forme de chaînes de caractères.
- Par défaut donne le nom long de la classe@le hash code en hexadecimal
- Il est très fortement conseillé de la surcharger
- Attention : Elle sert en priorité pour vous développeur, **elle ne sert pas à l'affichage ou au code métier.**

```
package fr.exemple;

public class Personne {

    private int age;
    private boolean sex;
    private String nom;
    private String prenom;

    @Override
    public String toString() {
        StringBuilder builder = new StringBuilder();
        builder.append(this.getClass().getSimpleName());
        builder.append(" [age=");
        builder.append(this.getAge());
        builder.append(", sex=");
        builder.append(this.getSex() ? "Homme" : "Femme");
        builder.append(", nom=");
        builder.append(this.getNom());
        builder.append(", prenom=");
        builder.append(this.getPrenom());
        builder.append("]");
        return builder.toString();
    }
}
```

La méthode equals

175

- Compare l'instance courante avec un autre objet.
- Par défaut fait == (c.a.d compare les références mémoires)
- Il n'est pas obligatoire de la surcharger. Mais elle peut être utilisée par les APIs des List ou Map
- Attention : Il est conseillé d'écrire aussi la méthode hashCode

```
@Override
public boolean equals(Object obj) {
    // Suis je moi même
    if (this == obj) {
        return true;
    }
    // Je ne peux pas être null
    if (obj == null) {
        return false;
    }
    // Comparaison entre moi et un autre objet Personne
    if (obj instanceof Personne) {
        Personne other = (Personne) obj;
        // Ici on passe tous les attributs en revue
        if (this.getAge() != other.getAge()) {
            return false;
        }
        if (this.getNom() == null) {
            if (other.getNom() != null) {
                return false;
            }
        } else if (!this.getNom().equals(other.getNom())) {
            return false;
        }
        // on fait la même chose pour les autres attributs
    }
    return true;
}
```

La méthode hashCode

176

- Retourne un entier qui représente l'objet.
- Il n'est pas obligatoire de la surcharger. Mais elle peut être utilisée par les APIs des List ou Map

```
@Override
public int hashCode() {
    // Ici on decide que deux personnes ont le meme hashCode si
    // elles ont le meme nom et prenom
    if ((this.getNom()!=null) && (this.getPrenom() != null)){
        return (this.getClass().getName()+"_"+this.getNom()+"_"+this.getPrenom()).hashCode();
    }
    return super.hashCode();
}
```

Les concepts Objet avec Java

177

L'encapsulation

La composition

L'héritage

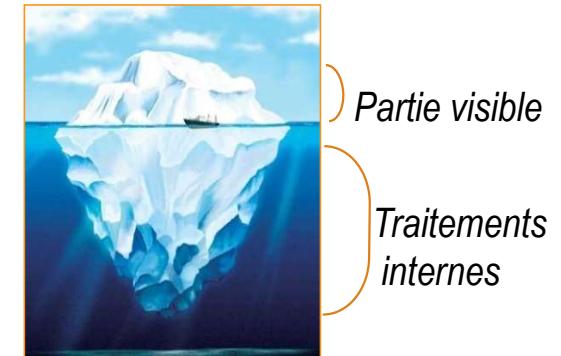
Le polymorphisme

Les interfaces Java

Les Inner classes

Problématique

- Une classe fournit un ensemble de services
 - ▶ Traitements
 - ▶ Transport de données
- Une classe doit être simple d'utilisation
 - ▶ Interface d'utilisation claire et simple
 - ▶ Complexité masquée à l'intérieur
- But : masquer les traitements internes
 - ▶ Dans une classe, possibilité de restreindre la visibilité des attributs et méthodes



Exemple

179

➤ Calcul de la recette d'un aéroport

```
public class RecetteAeroport {  
    public long calculerRecetteTotale() {  
        // calcul recette des ventes billets  
        // calcul recette restaurants  
        // calcul taxes aéroport  
        // ...  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        RecetteAeroport r = new RecetteAeroport();  
        float recette = r.calculerRecetteTotale();  
    }  
}
```

➤ La méthode `calculerRecetteTotale()` est très longue. Comment la diviser en sous-méthodes ?

Exemple (2)

```
public class RecetteAeroport {
    public long calculerRecetteTotale() {
        //appel à toutes les autres méthodes
    }

    public long calculerVentesBilletsAvion() {...}
    public long calculerTaxesAeroport() {...}
    public long calculerRecetteRestaurants() {...}
}
```

```
public class Test {
    public static void main(String[] args) {
        RecetteAeroport r = new RecetteAeroport();
        long recette = r.calculerRecetteTotale();

        long taxes = r.calculerTaxesAeroport();
    }
}
```

- `calculerRecetteTotale()` fait appel aux autres méthodes de la classe
 - ▶ Les autres méthodes sont à usage interne de la classe.
 - ▶ Problème : la classe `Test` a accès à ces autres méthodes ...

Exemple (3)

181

- Solution : les méthodes à usage interne sont privées
 - ➡ Elles ne sont plus visibles de la classe Test

```
public class RecetteAeroport {  
    public long calculerRecetteTotale() {  
        //appel à toutes les autres méthodes  
    }  
  
    private long calculerVentesBilletsAvion() {...}  
    private long calculerTaxesAeroport() {...}  
    private long calculerRecetteRestaurants() {...}  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        RecetteAeroport r = new RecetteAeroport();  
        long recette = r.calculerRecetteTotale();  
  
        long taxes = r.calculerTaxesAeroport();  
    }  
}
```

Règles de visibilité : Attributs et méthodes

182

➤ Attributs et méthodes : plusieurs niveaux de visibilité

- ▶ **public** : élément visible partout.
- ▶ **private** : élément à usage interne de la classe, invisible de l'extérieur.
- ▶ **package** : élément visible partout dans le même package. Il n'y a pas de mot-clé, cas par défaut sauf dans le cas d'une interface. En anglais : *package private visibility*.
- ▶ **protected** : élément à usage des enfants et des membres du même package
 - Attention : Ne concerne pas QUE les enfants

```
public class Test {  
    public int var1;                      // attribut public  
    private int var2;                     // attribut privé  
    int var3;                            // attribut de visibilité 'package'  
  
    public Test() {}                      // constructeur public  
    public int method1() {}               // méthode public  
    private int method2() {}              // méthode privée  
    int method1() {}                     // méthode avec visibilité 'package'  
}
```

Règles de visibilité : Classes & Interfaces

183

- Classes : deux niveaux de visibilité seulement
 - ▶ public : classe visible partout

```
public class Test {  
    // classe publique  
}
```

- ▶ package : classe visible dans son package uniquement (pas de mot clef)

```
class Test {  
    // classe de visibilité package  
}
```

- Pour le cas des classes déclarées dans une classe, elle peuvent faire usage des 4 visibilités indiquées dans le slide précédent.

Règles de visibilité



- **public** : élément visible **partout**.
- **protected** : élément à usage des **enfants et des membres du même package**
- **package** : élément à usage des **membres du même package** (pas de mot-clé, cas par défaut sauf dans le cas d'une interface)
- **private** : élément à usage **interne** de la classe, invisible de l'extérieur.

Encapsulation des variables

➤ Méthodes getter et setter

- Bonne pratique : déclarer les attributs d'une classe **private**
- Utiliser des méthodes **get** et **set** pour accéder aux attributs.

```
public class Avion {  
    private long matricule;  
  
    public long getMatricule() {  
        return matricule;  
    }  
  
    public void setMatricule(long l) {  
        matricule = l;  
    }  
}
```

Encapsulation des variables (2)

- Intérêt des getters et setters ?
 - Il est possible de jouer sur la visibilité des méthodes d'accès.
 - Ex : variable en lecture seule en dehors du package.

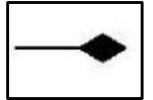
```
public class Avion {  
    private long matricule;  
  
    public long getMatricule() {  
        return matricule;  
    }  
  
    void setMatricule(long l) {  
        matricule = l;  
    }  
}
```

Visibilité publique [

Visibilité package [

- Un traitement spécifique peut être placé à chaque accès à une variable.
 - Message de log...

Définition : composition



- Un lien de composition symbolise l'existence d'une **agrégation** forte entre deux classes.

- Une composition est définie par les points suivants :
 - ▶ Durée de vie : toute classe agrégée est détruite quand la classe composite est détruite.
 - ▶ Exclusivité : une classe agrégée ne peut l'être que par une seule classe composite.
 - ▶ Notion de « fait partie de ».

Définition : composition

- Exemple : Un avion est composé d'un moteur et d'un matricule

```
public class Moteur {  
    private int identifiant;  
    private String dateRevision;  
    //...  
}
```

```
public class Avion {  
    private long matricule;  
    private Moteur moteur;  
  
    public Moteur getMoteur() { return moteur; }  
    public void setMoteur(Moteur moteur) { this.moteur = moteur; }  
    //...  
}
```

Références multiples

➤ Que produit le code suivant ?

```
Avion monAvion = new Avion();
Moteur monMoteur = new Moteur();
monAvion.setMoteur(monMoteur);

monMoteur.setDateRevision("21/02/2007");

System.out.println(monAvion.getMoteur().getDateRevision());
```

➤ Les deux références pointent vers la même zone mémoire !

Gestion de la mémoire

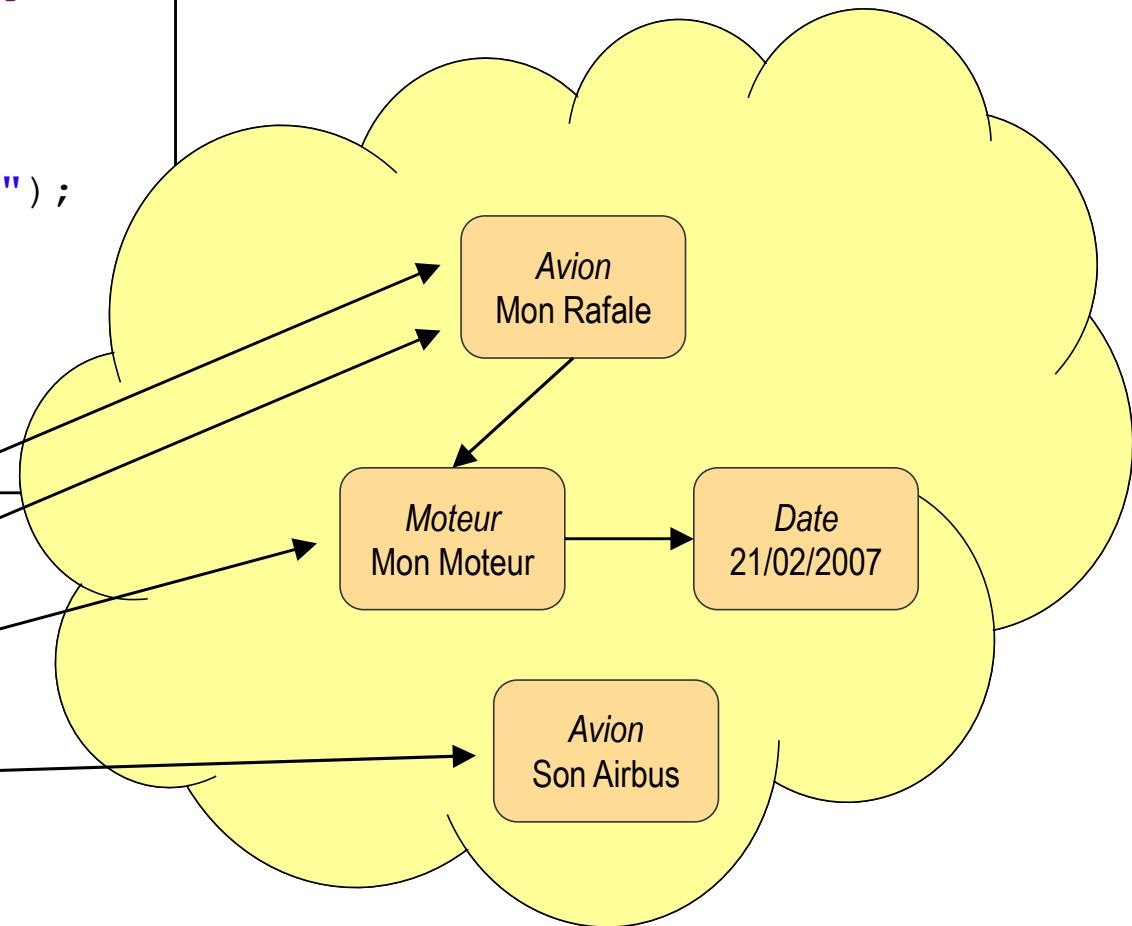
190

```
public static void main(String[] args) {  
    Avion monRafale = new Avion();  
    Moteur monMoteur = new Moteur();  
    monAvion.setMoteur(monMoteur);  
    monMoteur.setRevision("21/02/2007");  
  
    Avion sonAirbus = new Avion();  
  
    Avion lePlusRapide = monRafale;  
}
```

- monRafale
- lePlusRapide
- monMoteur
- sonAirbus

The Stack

The Heap



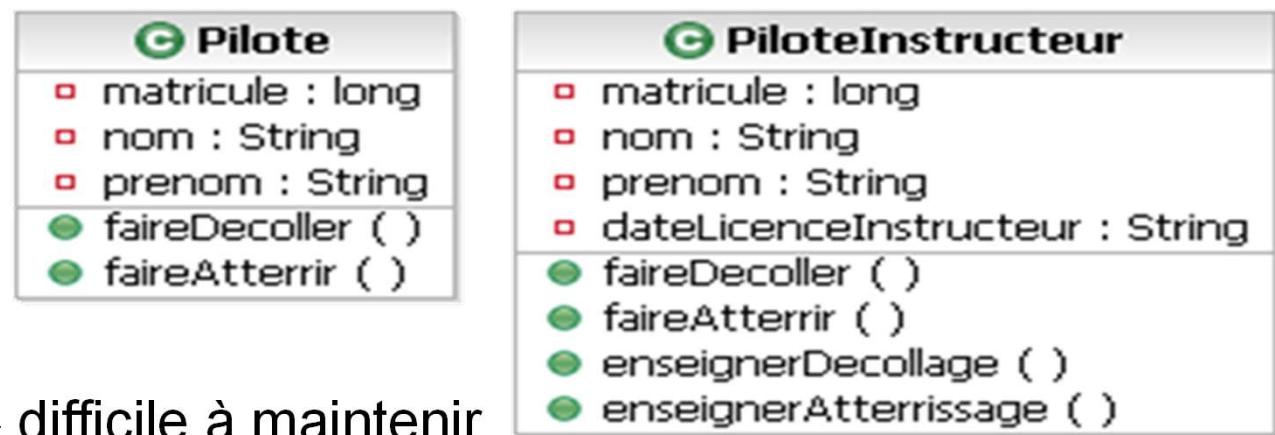
Problématique

191

➤ Exemple

- Un pilote instructeur a un nom, un prénom, un matricule, une licence d'instructeur. Il sait piloter un avion (le faire décoller et atterrir); il peut également enseigner à des apprentis

➤ Modélisation basique



➤ Duplication de code ⇒ difficile à maintenir

Solution : l'héritage

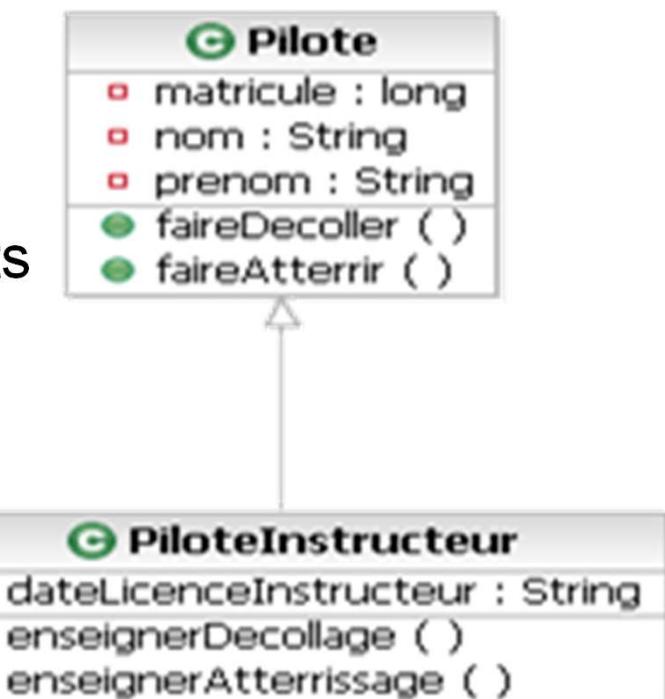
➤ Autre modélisation

- Un pilote instructeur est un type particulier de pilote qui a une licence d'instructeur. Il sait faire tout ce qu'un pilote sait faire, et de plus, il peut enseigner à des apprentis

➤ Relation d'héritage

- Pilote définit les comportements et attributs communs
- PiloteInstructeur ne définit que ce qui lui est propre

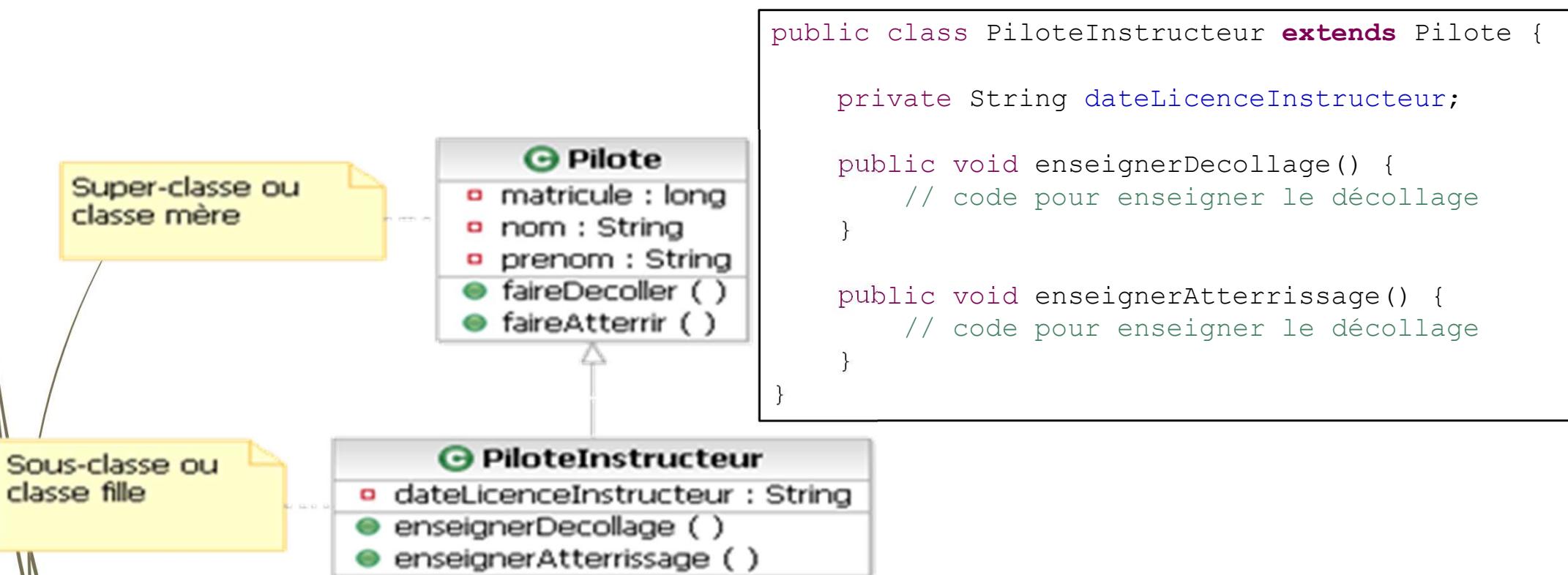
➤ Pas de duplication de code



Relation d'héritage en Java

193

➤ Mot-clef **extends** dans la déclaration de la sous-classe



Comportement (compilation)

194

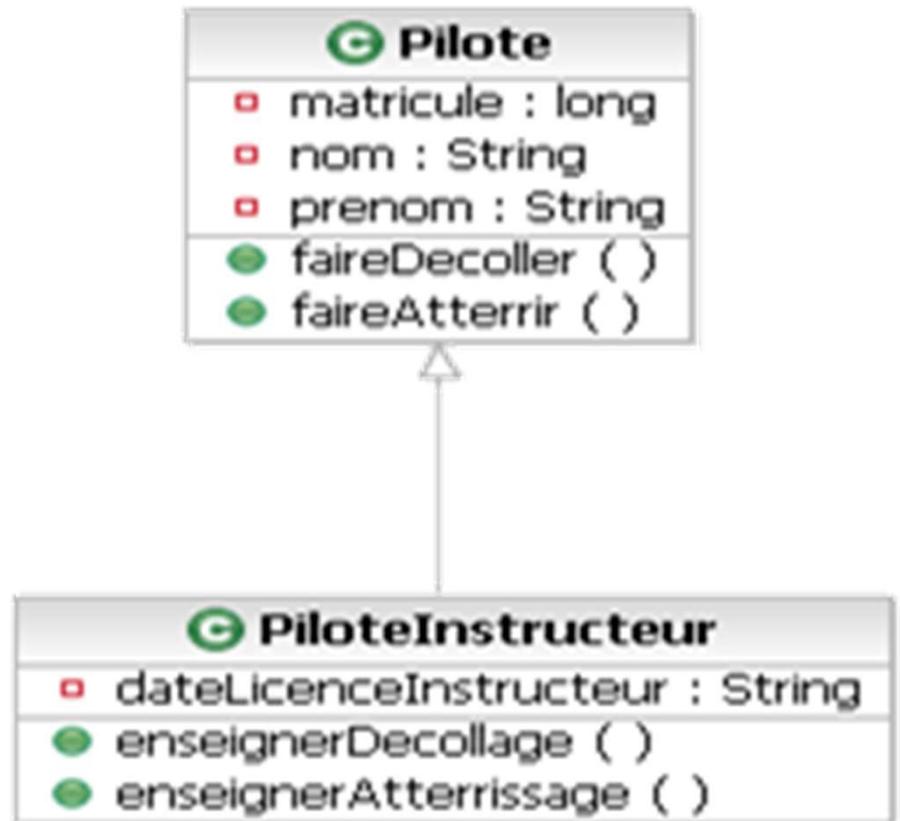
➤ L'état et le comportement d'un objet sont définis le long de la hiérarchie

► Structure interne (attributs)

- matricule
- nom
- prenom
- dateLicenceInstructeur

► Comportement (méthodes)

- enseignerDecollage()
- enseignerAtterrissage()
- faireDecoller()
- faireAtterrir()



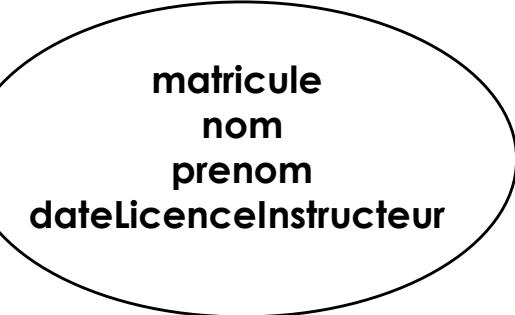
Comportement (exécution)

195

➤ Crédit

```
PiloteInstructeur instructeur = new PiloteInstructeur();
```

instructeur



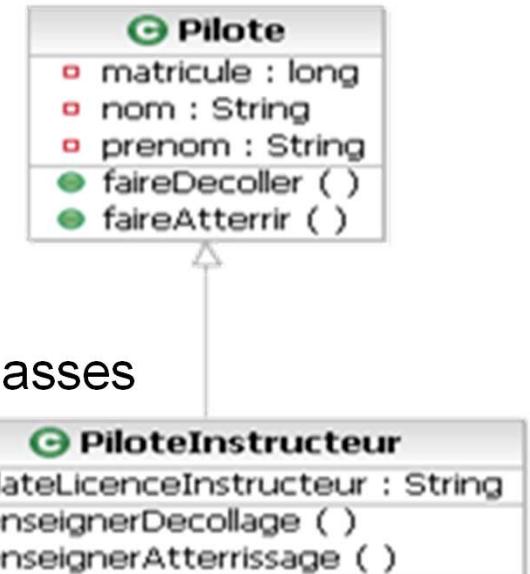
- 1 seul objet créé en mémoire avec les attributs de toutes les super-classes

➤ Appels de méthode

- Parcours de la hiérarchie de classes

- à partir de la classe instanciée vers les super-classes

- Arrêt dès la première implémentation trouvée (et exécution)



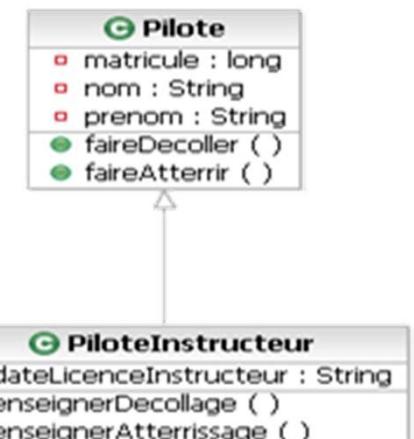
Héritage et visibilité

- Accès aux attributs et méthodes des super-classes soumis aux contraintes de visibilité

- public : oui
- package : dépend du package de la classe
- private : non
- protected : oui

➡ Conséquence: le code suivant ne compile pas

```
PiloteInstructeur instructeur = new PiloteInstructeur();
instructeur.matricule = "1234567"; // attribut invisible
```



- `protected` autorise l'accès depuis les sous-classes
 - ➡ Donne, malheureusement, aussi l'accès aux classes du même package

Hiérarchie des classes

➤ Arborescence

► Une classe a 1 seule super-classe

➤ Racine : java.lang.Object

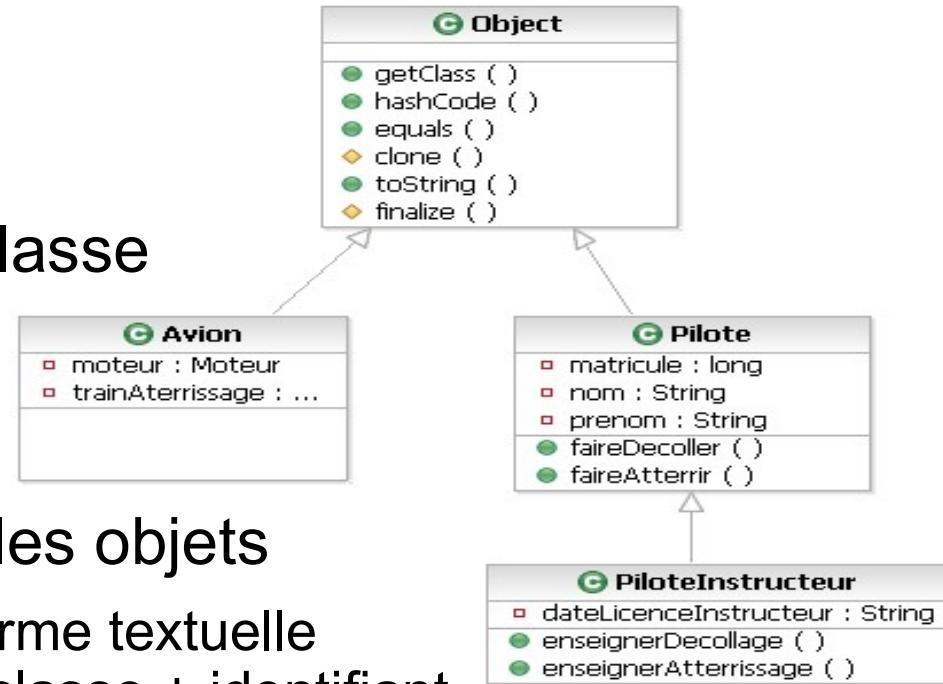
► Super-classe implicite

► Méthodes communes à tous les objets

- String `toString()` : retourne la forme textuelle de l'objet, ici le nom long de la classe + identifiant.

- int `hashCode()` : renvoie un code pour les tables de hachage

- boolean `equals(Object)` : indique si un objet est égal à un autre.
C'est à VOUS de définir la notion d'égalité.



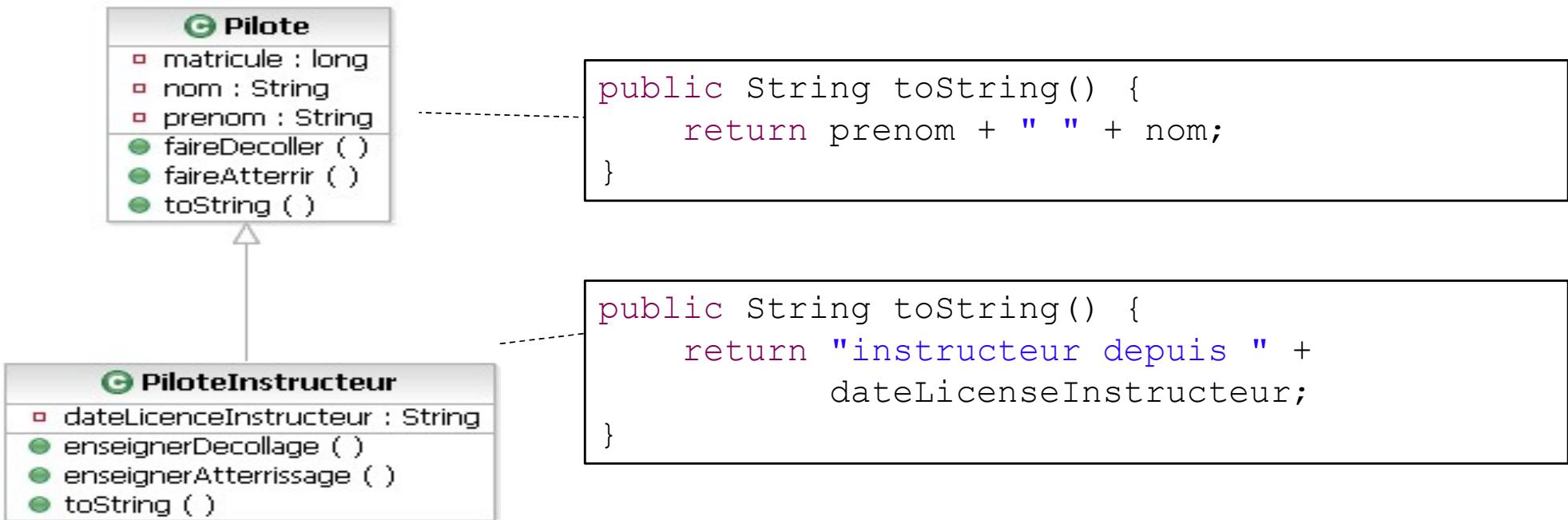
Redéfinition de méthodes

- Définir dans une sous-classe des méthodes avec la même signature que dans la super-classe
- Plusieurs sortes de redéfinitions
 - ▶ la définition de la super-classe est ignorée
 - ▶ extension de la définition de la super-classe
 - appel de la méthode de la super-classe
 - traitements spécifiques

Redéfinition de méthodes (2)

199

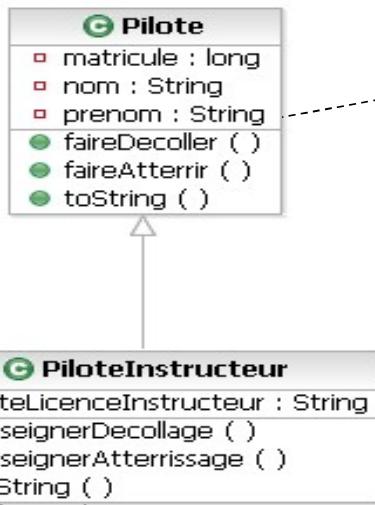
➤ 1er cas : la définition de la super-classe est ignorée



Redéfinition de méthodes (3)

200

- 2ème cas : la définition de la super-classe est reprise
 - ▶ Comment référencer les attributs/méthode de la super-classe ?
 - Si appel au mot-clef **this** : boucle infinie !!!
 - ▶ Mot-clé **super**
 - Commence la recherche de méthode ou d'attribut dans la super-classe



```
public String toString() {
    return prenom + " " + nom;
}

public String toString() {
    // appel à toString() de Pilote
    String nom = super.toString() + "\r\n";
    return nom + "instructeur depuis" +
           dateLicenceInstructeur;
}
```

Redéfinition de méthodes (4)

- Contraintes
 - ▶ Le type de retour doit être :
 - ❑ strictement identique
 - ❑ ou covariant (type spécialisé) Java 5
 - ▶ La visibilité de la méthode de la super-classe ne peut pas être restreinte dans la sous-classe
 - ❑ Par exemple : une méthode définie comme publique dans la super-classe ne peut pas être redéfinie comme privée dans la sous-classe
 - ▶ Contrainte avec le mécanisme d'exception
 - ❑ Sera abordée plus loin dans ce cours
- Ne pas confondre surcharge et redéfinition
 - ▶ **Redéfinition** : exactement la même signature
 - ▶ **Surcharge** : paramètres différents

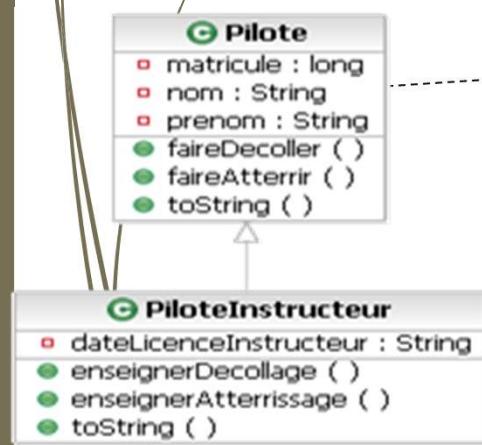
Redéfinition de méthodes (5)

202

➤ Annotation @Override

Java 5

- ▶ Méta-donnée précisant au compilateur que la déclaration d'une méthode redéfinie une méthode de sa super classe.
- ▶ Erreur générée par le compilateur java lorsqu'une méthode est annotée avec @Override mais ne redéfinie pas de méthode.
- ▶ Facultatif mais doublement utile.



```
public String toString() {
    return prenom + " " + nom;
}
```

```
@Override
public String toString() {
    // appel à toString() de Pilote
    String nom = super.toString() + "\r\n";
    return nom + "instructeur depuis" + dateLicenceInstructeur;
}
```

Redéfinition de méthodes - 5 Règles

203

1. La méthode dans la classe enfant doit avoir la même signature que la méthode dans la classe parent.
2. La méthode de la classe enfant doit être au moins aussi accessible ou plus accessible que la méthode dans la classe parente.
3. La méthode de la classe enfant ne peut pas générer une exception (*checked*) nouvelle ou plus large que la classe des exceptions levées dans la méthode de la classe parent.
4. Si la méthode renvoie une valeur, celle-ci doit être identique ou correspondre à une sous-classe de celle de la méthode de la classe parente.
5. La méthode définie dans la classe enfant doit être marquée comme statique si elle est marquée comme statique dans la classe parente (*masquage de méthode*). De même, la méthode ne doit pas être marquée comme statique dans la classe enfant si elle n'est pas marquée comme statique dans la classe parent (*surcharge de méthode*)

Redéfinition de méthodes - Exemples

204

```
public class Animal {  
  
    private void faire() {  
        System.out.println("Animal");  
    }  
  
    public static void faireA() {  
        System.out.println("Static Animal");  
    }  
  
    protected void faireB() {  
        System.out.println("Animal - FaireB");  
    }  
  
    public void faireC() {  
        System.out.println("Animal - FaireC");  
    }  
}
```

```
public class Chien extends Animal {  
  
    public void faire() {  
        // Comme la méthode du parent est private,  
        // Il n'y a pas de @Override (impossible de le mettre)  
        System.out.println("Chien");  
    }  
  
    public static void faireA() {  
        System.out.println("Static Chien");  
    }  
  
    @Override  
    protected void faireB() {  
        // Comportement classique ( @Override )  
        System.out.println("Chien - FaireB");  
    }  
  
    @Override  
    public void faireC() {  
        // Comportement classique ( @Override )  
        System.out.println("Chien - FaireC");  
    }  
}
```

Redéfinition de méthodes - Exemples

205

```
public class Run {  
  
    public static void main(String[] args) {  
        Chien c = new Chien();  
        c.faire(); // Affiche "Chien"  
        c.faireA(); // Affiche "Static Chien"  
                    // <- se fie au typage (gauche du signe =, c est Chien)  
  
        Chien.faireA(); // Affiche "Static Chien"  
        c.faireB(); // Affiche "Chien - FaireB"  
        c.faireC(); // Affiche "Chien - FaireC"  
  
        // Avec le Polymorphisme maintenant  
  
        Animal a = new Chien();  
        a.faireA(); // Affiche "Static Animal"  
                    // <- se fie au typage (gauche du signe =, a est Animal)  
  
        Animal.faireA(); // Affiche "Static Animal"  
        a.faireB(); // Affiche "Chien - FaireB"  
        a.faireC(); // Affiche "Chien - FaireC"  
    }  
}
```

Attention : règle 3 - Override et exceptions

206

- La règle 3 implique qu'un enfant lors d'un override peut :
 - ➡ Reprendre le prototypage du parent tel quel (conseillé)
 - ➡ Reprendre le prototypage du parent en modifiant la clause throws par des enfants des exceptions spécifiées par le parent
 - ➡ Reprendre le prototypage du parent en modifiant la clause throws par des RuntimeException (ou enfant)
 - ➡ Reprendre le prototypage du parent sans reprendre la clause throws (du coup ne lever aucune exception)

Redéfinition de méthodes - Cas particulier

207

- Dans le cas des inner classes,
 - ▶ Le parent a une méthode privée
 - ▶ L'enfant surcharge cette méthode privée et monte sa visibilité
 - ▶ Dans un main, on déclare une variable locale du type du parent mais on l'instancie du type de l'enfant
 - ▶ On appelle la méthode sur l'instance
 - ▶ Dans ce cas, la méthode exécutée est celle du typage et non celle de l'instance.

Redéfinition de méthodes - Cas particulier

208

```
public class Animal {

    private void faire() {
        System.out.println("Animal");
    }

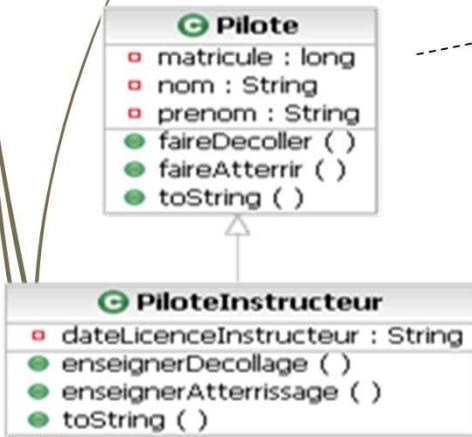
    public static void main(String[] args) {
        Chien c = new Chien();
        c.faire(); // Affichera "Chien"
        // Cas particulier ici
        Animal a = new Chien();
        a.faire(); // Affichera "Animal" et non pas "Chien"
    }
}

class Chien extends Animal {
    public void faire() {
        System.out.println("Chien");
    }
}
```

Héritage et constructeurs

209

- Utilisation d'un constructeur hérité
 - ▶ Appel d'un constructeur de la super-classe avec super(...)
 - Similitude avec utilisation de this(...)
 - ▶ Puis traitements spécifiques
- ou appel implicite à super()



```
public Pilote(String nom, String prenom) {
    // appel implicite à super();
    this.nom = nom;
    this.prenom = prenom;
}
```

```
public PiloteInstructeur(String nom, String prenom) {
    super(nom, prenom);
    this.dateLicenseInstructeur = "21/02/2000";
}
```

 super(...) doit être la première instruction du constructeur

Interdire l'héritage : mot clef final

➤ Classe avec le mot-clef **final**

- ▶ Interdit la création de sous-classes

```
public final class PiloteInstructeur {  
    // attributs, constructeurs, méthodes  
}
```

➤ Méthode avec le mot-clef final

- ▶ Interdit de redéfinir la méthode dans une sous-classe

```
public final void faireDecoller(Avion unAvion) {  
    // code pour faire décoller l'avion  
}
```

➤ Rappel

- ▶ Pour un attribut, final signifie qu'il ne peut pas être redéfini (donc constant pour les primitives et les classes "immuables")



211

Travaux pratiques

Réaliser les travaux pratiques suivants:

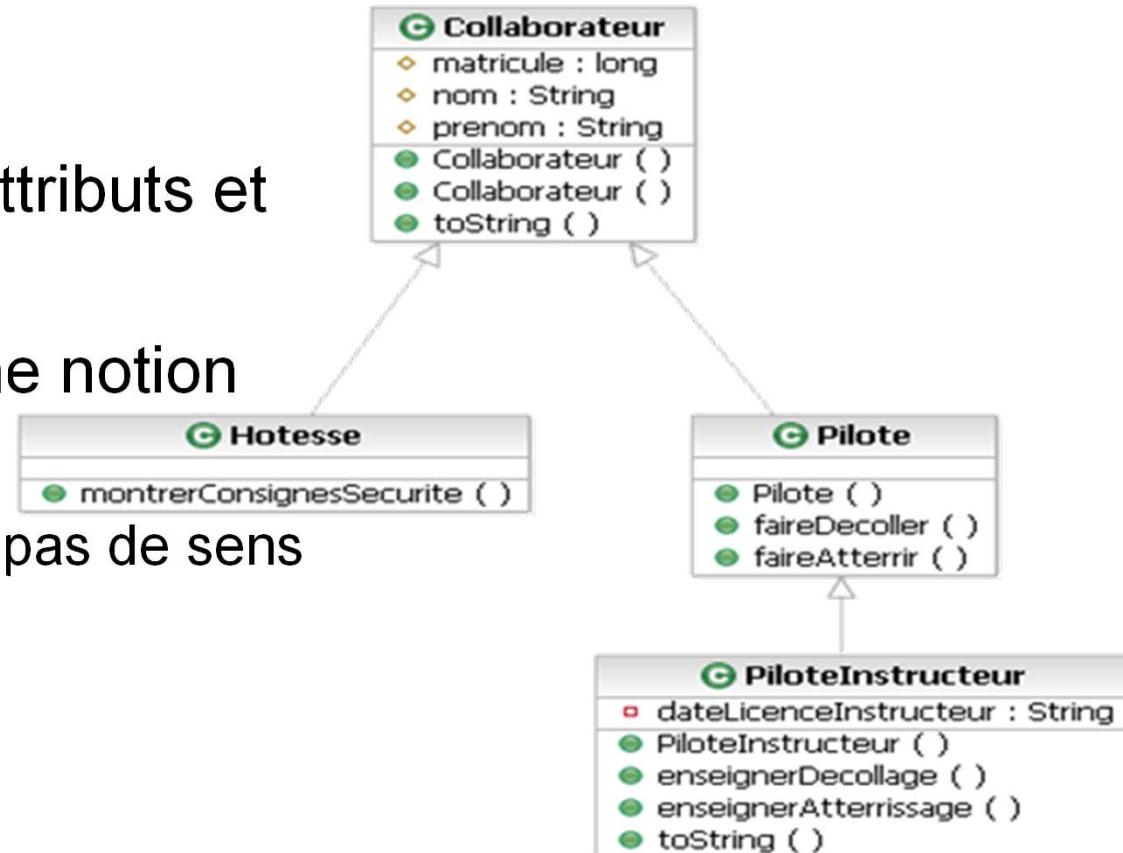
TP 2 partie 4

TP 3 partie 1

Classe abstraite : Problématique

➤ Classe Collaborateur

- ▶ Crée pour factoriser attributs et méthodes
- ▶ Un collaborateur est une notion abstraite
 - ❑ new Collaborateur() n'a pas de sens
 - ❑ Comment l'interdire ?



Classe abstraite

- Classe qui ne possède pas d'instance
- Propose une comportement pour des sous-classes
 - ➡ Fournit plus qu'un comportement commun
 - ❑ Attributs ET Méthodes
 - ➡ Les spécificités sont dans les sous-classes
 - ❑ Ajout d'attributs
 - ❑ Redéfinition/ajout de méthodes
- Utilisée dans le cadre d'une généralisation
 - ➡ Factorisation des attributs et des méthodes
 - ➡ Super-classe d'une hiérarchie

Déclaration de classe abstraite

➤ Collaborateur est abstrait

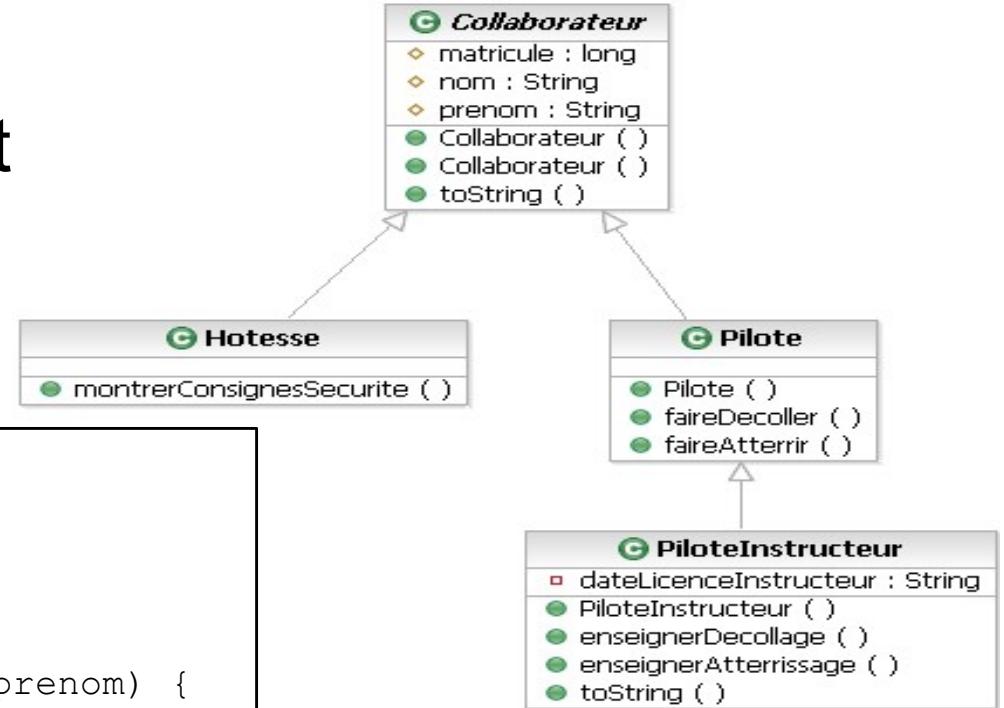
□ Notée en italique en UML

➤ Mot-clef **abstract**

```
public abstract class Collaborateur {
    protected long matricule;
    protected String nom;
    protected String prenom;

    public Collaborateur(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }

    public String toString() {
        return prenom + " " + nom;
    }
}
```



Classe abstraite et constructeurs

215

➤ Une classe abstraite peut avoir des constructeurs

- ▶ Factorisation de l'initialisation
- ▶ Appelés par les constructeurs des sous-classes

□ Mais appel direct (`new`) interdit

```
public abstract class Collaborateur {  
    protected String nom;  
    protected String prenom;  
  
    public Collaborateur(String nom, String prenom) {  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
}  
  
public class Pilote extends Collaborateur {  
    public Pilote(String nom, String prenom) {  
        super(nom, prenom);  
    }  
}
```

Méthode abstraite : Problématique

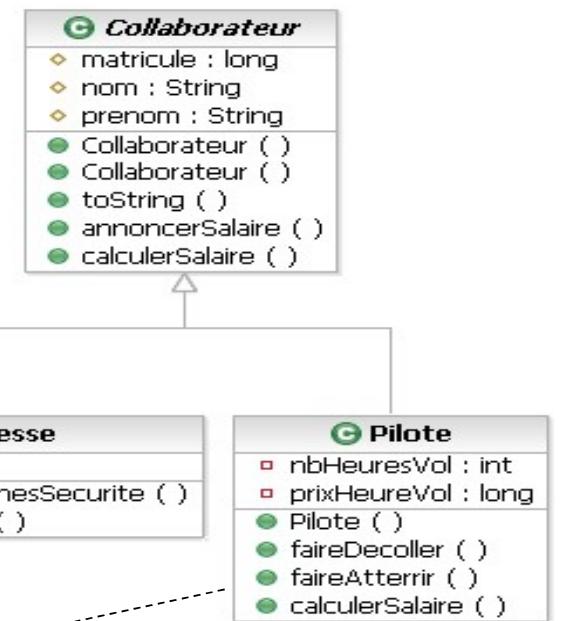
216

➤ Exemple salaires des collaborateurs

- ❑ `calculerSalaire` renvoie le salaire
- ❑ `annoncerSalaire` écrit le salaire dans
- ❑ comment factoriser le code ?

```
public long calculerSalaire() {  
    return salaire;  
}  
  
public void annoncerSalaire(){  
    System.out.println("Salaire : " + calculerSalaire());  
}
```

```
public long calculerSalaire() {  
    return nbHeuresVol * prixHeureVol;  
}  
  
public void annoncerSalaire(){  
    System.out.println("Salaire : " + calculerSalaire());  
}
```



Méthode abstraite

- Mot-clé **abstract**
- Non complète
 - ➡ Signature uniquement
 - ❑ Pas d'accolades { }. Se termine par ;
 - ➡ Pas de code (ou d'algorithme)
 - ➡ Spécifiée dans une classe abstraite

```
public abstract long calculerSalaire();
```

- Redéfinition obligatoire dans les sous-classes concrètes
 - ➡ Définir le code (ou l'algorithme)
- Ne peut pas être : private ou final

Exemple abstraction + contraintes

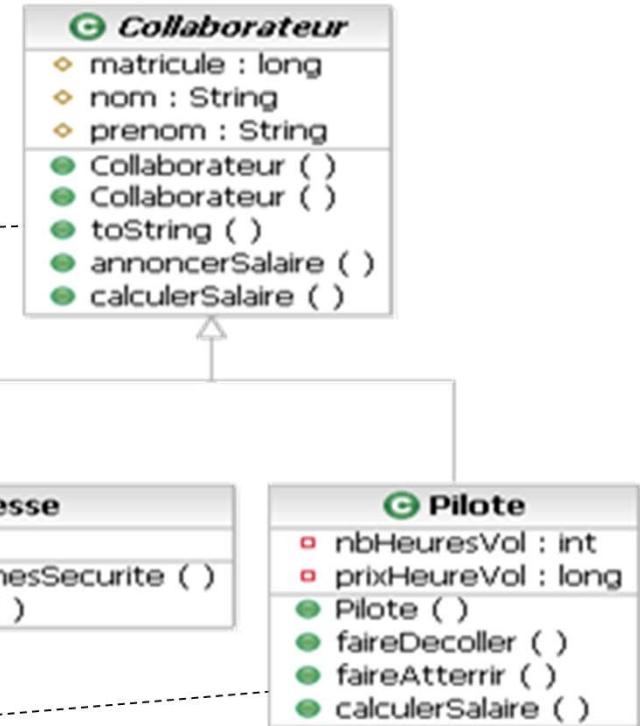
218

```
public abstract long calculerSalaire();

public void annoncerSalaire() {
    System.out.println("Salaire : " + calculerSalaire());
}
```

```
public long calculerSalaire() {
    return salaire;
}
```

```
public long calculerSalaire() {
    return nbHeuresVol * prixHeureVol;
}
```



Rappel : on recherche toujours les méthodes à partir de la classe courante.

Contraintes liées à l'abstraction

- La redéfinition d'une méthode abstraite
 - ▶ Est obligatoire dans une sous-classe concrète
 - ▶ Sinon la sous-classe doit elle-même être abstraite
- Une classe abstraite
 - ▶ Ne peut pas avoir d'instance = pas de new
 - ▶ Peut contenir des attributs
 - ▶ Peut contenir des méthodes concrètes
 - ▶ Peut contenir des constructeurs
 - ❑ Ils ne servent qu'aux enfants car pas de new possible sur la classe abstraite

Polymorphisme : définition

- Un objet peut être vu sous plusieurs formes
 - ➡ Exemple : un pilote instructeur peut être vu comme PilotInstructeur, Pilote, Collaborateur ou Object
- Plusieurs objets différents peuvent être vus sous la même forme
 - ➡ Ex : un pilote instructeur, une hôtesse et un mécanicien peuvent être vus comme des collaborateurs

Un objet peut être vu sous plusieurs formes

221

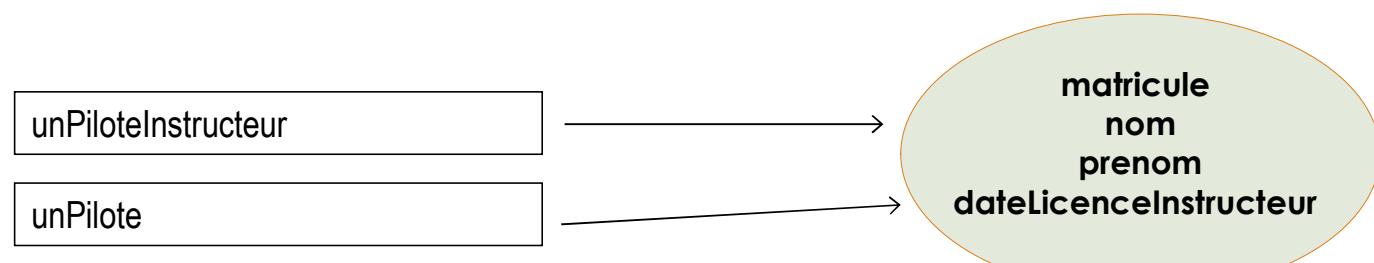
➤ Définitions

- ▶ Type réel d'un objet : classe dont le constructeur est appelé pour créer l'objet
- ▶ Type déclaré : classe utilisée pour manipuler l'objet

➤ Upcasting : type déclaré = superclasse du type réel

- ▶ Utilisation d'une variable du type de la super-classe à la place d'une variable du type réel de l'objet

```
PiloteInstructeur unPiloteInstructeur = new PiloteInstructeur();  
Pilote unPilote = unPiloteInstructeur;
```



Un objet peut être vu sous plusieurs formes

222

➤ Comportement

► Compilation

- Vérification de la compatibilité type déclaré/type réel
 - si pas compatible, erreur de compilation
- Les méthodes/attributs accessibles sont ceux de la classe déclarée

► Exécution

- «liaison dynamique» : détermine le type réel de l'objet à l'exécution, comportement de la classe réelle

Une forme pour plusieurs objets

- Plusieurs objets différents peuvent être vus sous la même forme
- Permet d'avoir des traitements génériques
- Exemple
 - ➡ Une gestionnaire de paie veut calculer le salaire d'un collaborateur
 - ➡ Un collaborateur est un pilote, une hôtesse, un mécanicien ...
 - ➡ On peut manipuler directement le type Collaborateur

Exemple

224

```
public class GestionnairePaie {  
    public void editerAttestationSalaire(Collaborateur collab) {  
        long salaire = collab.calculerSalaire();  
        String texte = "attestation officielle. Montant du salaire : "  
                    + salaire;  
        System.out.println(texte);  
    }  
}
```

```
public static void main(String[] args) {  
    GestionnairePaie paie = new GestionnairePaie();  
  
    Hotesse uneHotesse = new Hotesse();  
    paie.editerAttestationSalaire(uneHotesse);  
  
    Pilote unPilote = new Pilote();  
    paie.editerAttestationSalaire(unPilote);  
}
```

Avantages du polymorphisme

➤ Code plus lisible et plus maintenable

Procédural (pseudo code)
sans polymorphisme

```
si collab = 'Pilote'  
    calculerSalairePilote
```

```
si collab = 'Hotesse'  
    calculerSalaireHotesse
```

```
si collab = 'Mecanicien'  
    calculerSalaireMecanicien
```

Objet avec polymorphisme
collab.calculerSalaire()
=> mise en œuvre de
calculerSalaire dans
toutes les classes concernées
Pilote, Hotesse, Mecanicien

Polymorphisme et redéfinition

- On associe souvent polymorphisme et redéfinition
- Cas pratique
 - `System.out.print(Object unObjet)`

```
public void print(Object obj) {  
    String texte = (obj == null) ? "null" : obj.toString();  
    write(texte);  
}
```

- Méthode générique d'affichage
- Le texte affiché dépend de la classe réelle de l'objet

Downcasting

227

- ▶ Besoin de convertir dans une classe fille, pour avoir accès aux méthodes spécifiques
- ▶ opérateur cast : (TypeDésiré)
 - ❑ exécution ⇒ essaie de convertir dans la classe spécifiée
 - ❑ si impossible, erreur : ClassCastException

▶ Exemple

```
Collaborateur unCollaborateur = new Hotesse();  
/*  
 * Je peux appliquer à unCollaborateur seulement les méthodes  
 * déclarées dans Collaborateur (et éventuellement redéfinies  
 * dans Hotesse) si je veux pouvoir appeler les méthodes  
 * propres à Hotesse  
 */  
Hotesse uneHotesse = (Hotesse) unCollaborateur;
```

Classes 'sealed'

228

- Depuis le Java 15 il est possible de 'fermer' une classe (ou une interface) afin de limiter ses enfants.
- Dans cette approche, c'est lors de sa déclaration que vous allez préciser qui a le droit d'hériter ou non de votre classe/interface.
- Le mot clef **sealed** fera son apparition dans la déclaration ainsi que **permits**

Classes 'sealed'

229

➤ Exemple

```
public abstract sealed class Shape  
    permits Circle, Rectangle, Square {...}
```

► Circle, Rectangle et Square sont les seules classes à pouvoir hériter de Shape

➤ Ne pas confondre *final* et *sealed*

- **final** : la classe ne pourra avoir aucun enfant (jamais)
- **sealed** : la classe définit ses enfants explicitement

Conclusion

230

- ▶ A gauche du signe =
 - ❑ On parle au compilateur
 - ❑ Compile time
- ▶ A droite du signe =
 - ❑ On est dans l'exécution du programme = ce qui se passe pour de vrai en mémoire
 - ❑ Run time
- ▶ Vous pouvez placer à droite du signe = n'importe qu'elle instance à partir du moment où elle appartient à la famille de ce qui est déclaré à gauche du signe =.
 - ❑ Attention : le contraire ne marche pas

Opérateur instanceof

- L'opérateur **instanceof** permet de savoir si une **instance** d'un objet appartient à une famille

```
Object monInstance = "Toto";
if (monInstance instanceof String) {
    String monInstanceCastee = (String) monInstance;
    System.out.println(monInstanceCastee.slit('o'));
} else {
    System.out.println(" monInstance n'appartient pas à la famille des String");
}
```

Opérateur instanceof

232

- Vous pouvez l'utiliser avec une classe ou une interface

```
Object monInstance = "Toto";
if (monInstance instanceof Comparable) {
    Comparable monInstanceCastee = (Comparable) monInstance;
    ...
}
```

Opérateur instanceof ou méthode getClass

233

- L'opérateur instanceof permet de savoir si une instance appartient à une famille
 - ➡ Une instance peut appartenir à une **très grande** famille
- La méthode getClass (présente dans java.lang.Object) donne la classe de l'instance
 - ➡ Une instance ne peut avoir **qu'une seule** Class

Opérateur instanceof en Java 14

234

- Vous pouvez vous passer du cast en donnant directement un nom à votre instance

```
Object monInstance = "Toto";
if (monInstance instanceof String monInstanceCastee) {
    System.out.println(monInstanceCastee.substring(0));
} else {
    System.out.println(" monInstance n'appartient pas à la famille des String");
    // Bien évidemment, la variable monInstanceCastee n'existe pas dans ce bloc
}
```



235

Travaux pratiques

Réaliser les travaux pratiques suivants:

TP 3 partie 2

Problématique

➤ Exemple

- Apprentissage du pilotage : utilisation d'un simulateur puis d'un avion
 - Mêmes boutons, mêmes comportements, même "contrat"
⇒ mêmes signatures de méthode
 - Mais pas la même implémentation
- Utiliser l'héritage ?
 - Inadapté : un simulateur et un avion sont très différents, pas d'attribut ou de méthode à factoriser
 - Un simulateur n'est pas une spécialisation d'avion

Solution : Les interfaces

- Interface : contrat que respecte une classe
 - Signature de méthodes, documentation
 - Pas d'implémentation (sauf en Java 8+)
 - Des constantes (***final et static***)
- Mise en œuvre
 - Déclaration : mot clef **interface** au lieu de class
 - Tous les attributs sont implicitement **public static final** (les trois)
 - Toutes les méthodes sont implicitement **public abstract**



Nom de l'interface : souvent suffixe en "able" : Comparable, Piloteable...

Exemple

238

➤ Interface Pilotable



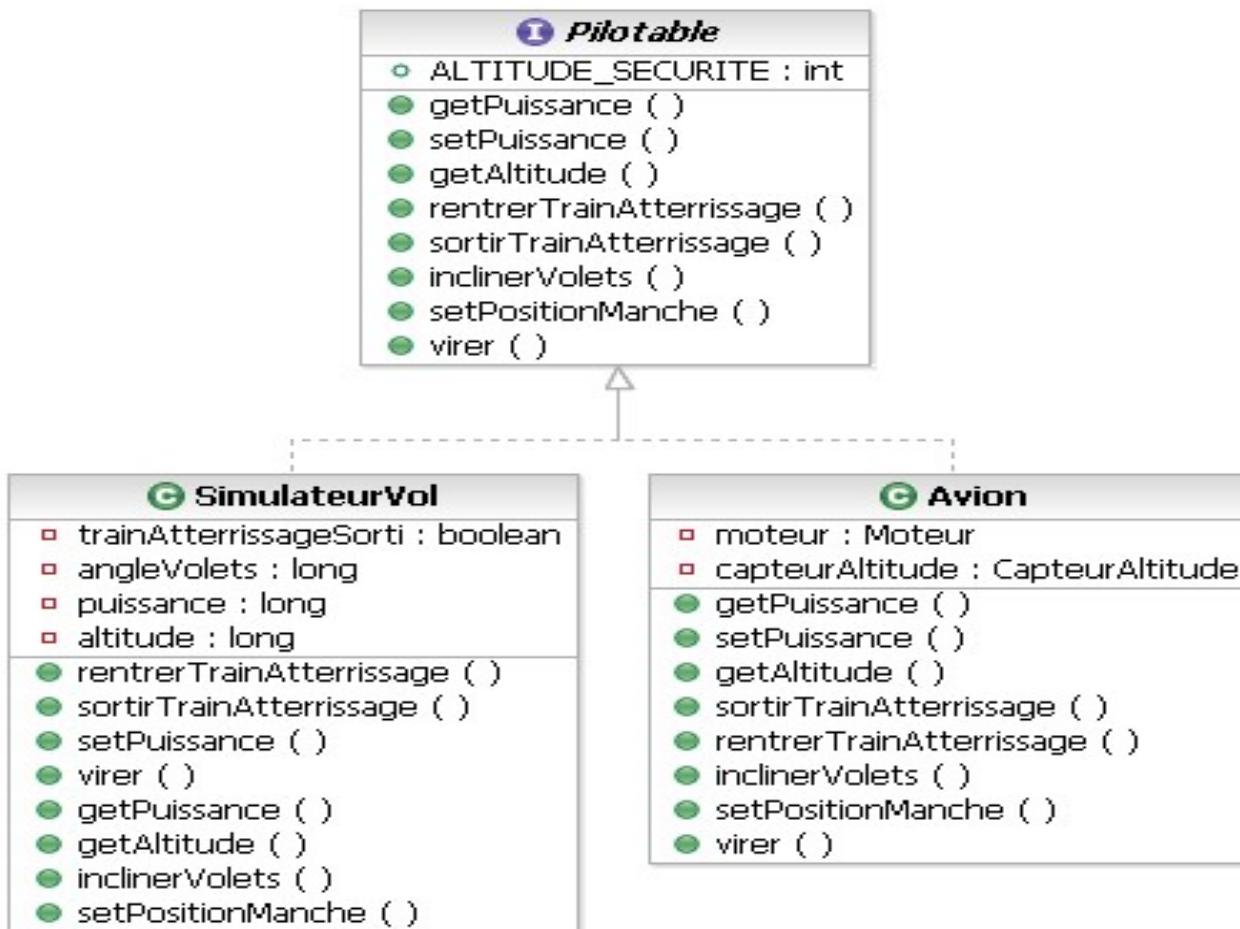
```
public interface Pilotable {
    static final int ALTITUDE_SECURITE = 100;

    long getPuissance();
    void setPuissance(long puissanceCible);
    long getAltitude();
    void rentrerTrainAtterrissage();
    void sortirTrainAtterrissage();
    void inclinerVolets(long angle);
    void setPositionManche(long position);
    void virer(long angle);
}
```

Respect de l'interface

239

- Avion et SimulateurVol respectent l'interface Pilotable



Classes d'implémentation

240

➤ Mot-clé **implements**

- ▶ La classe implémente toutes les méthodes de l'interface
 - Sinon erreur compilation
 - doivent être déclarées public
- ▶ Plus éventuellement d'autres méthodes

```
public class Avion implements Pilotable {  
    private Moteur moteur;  
    private CapteurAltitude capteurAltitude;  
  
    public long getPuissance() { return moteur.getPuissance(); }  
    public void setPuissance(long puissance) { moteur.setPuissance(puissance); }  
    public long getAltitude() { return capteurAltitude.getAltitude(); }  
    public void sortirTrainAtterrissage() { /* code */ }  
    public void rentrerTrainAtterrissage() { /* code */ }  
    public void inclinerVolets(long angle) { /* code */ }  
    public void setPositionManche(long position) { /* code */ }  
    public void virer(long angle) { /* code */ }  
}
```

Utilisation des interfaces

- Interface = groupe de services rendus par la classe
 - ➡ ex : Pilotable, GestionnaireRadio, ...
- Utilisation comme type
 - ➡ Accès seulement aux méthodes et attributs de l'interface (sinon erreur à la compilation)
 - ➡ Permet d'appliquer des méthodes sans connaître la nature réelle de l'objet ⇒ indépendance vis-à-vis de l'implémentation
 - ❑ Implémentation de test
 - ❑ Implémentation "réelle"
 - ❑ Autre implémentation (autre vendeur par exemple)

Exemple d'utilisation comme un type

242

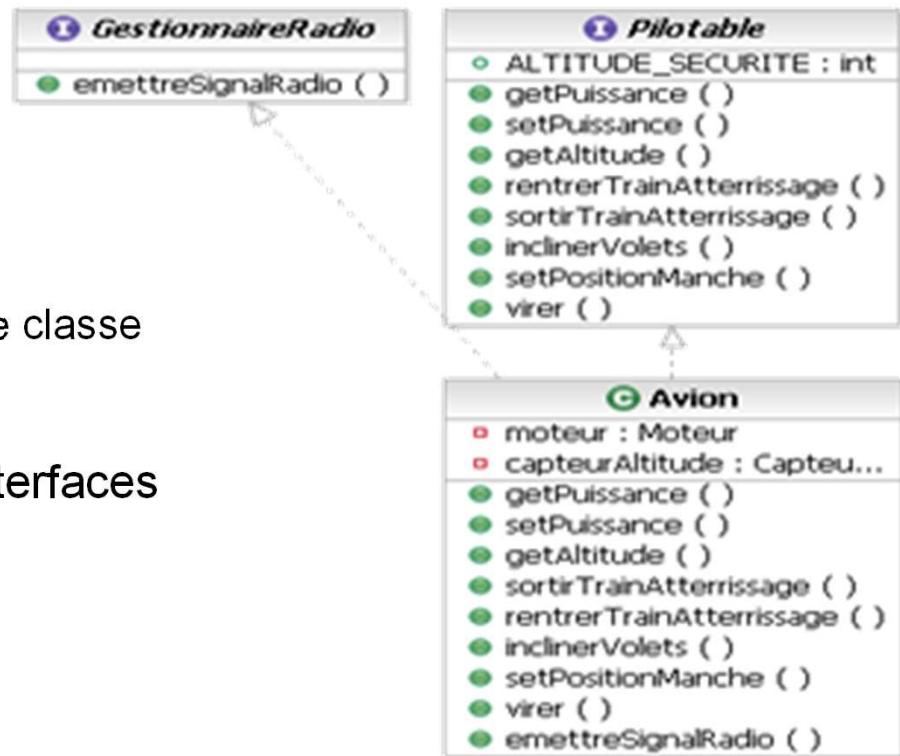
```
public void faireDecoller(Pilotable objetVolant) {  
    // code simpliste pour faire décoller l'avion  
    objetVolant.setPuissance(400);  
    objetVolant.inclinerVolets(10);  
    objetVolant.setPositionManche(3);  
    if (objetVolant.getAltitude() >= Pilotable.ALTITUDE_SECURITE) {  
        objetVolant.inclinerVolets(0);  
        objetVolant.rentrerTrainAtterrissage();  
    }  
}
```

```
public static void main(String[] args) {  
    Pilote unPilote = new Pilote();  
    Pilotable pilotable = new SimulateurVol();  
    // autre possibilité : pilotable = new Avion();  
    unPilote.faireDecoller(pilotable);  
}
```

Héritage multiple

243

- N'existe pas en Java
 - ▶ Une classe ne peut hériter que d'une seule classe
- Une classe peut implémenter plusieurs interfaces
 - ▶ Plusieurs "casquettes"
 - Objet Pilotable
 - Objet avec GestionnaireRadio



```
public class Avion implements Pilotable, GestionnaireRadio {
    // attributs
    // toutes les méthodes de l'interface Pilotable
    public long getPuissance() { return moteur.getPuissance(); }
    public void setPuissance(long puissance) { moteur.setPuissance(puissance); }

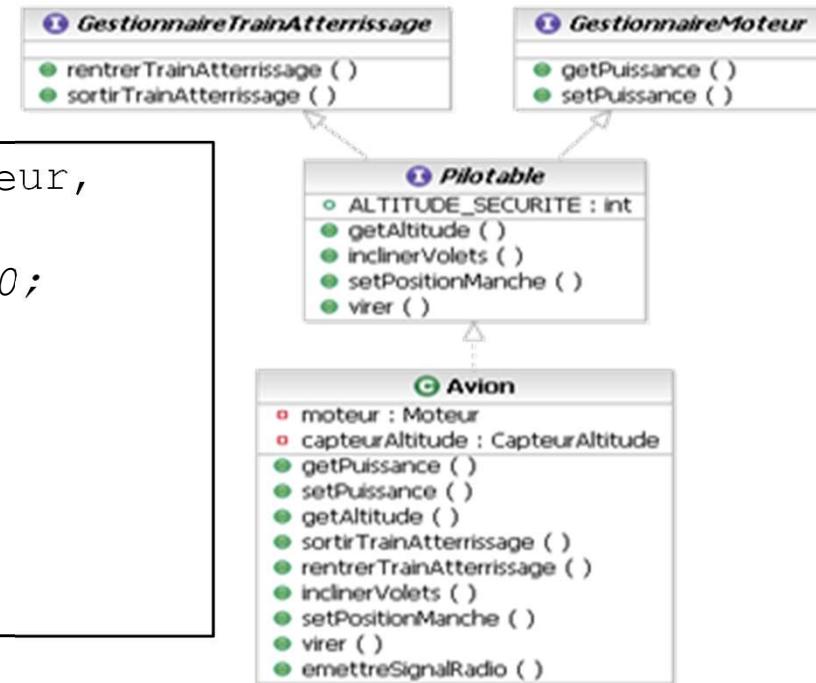
    // toutes les méthodes de l'interface GestionnaireRadio
    public void emettreSignalRadio(String message) { /* code */ }
}
```

Héritage d'interface

244

- Une interface peut hériter de plusieurs interfaces
 - ➡ Ensemble des services des interfaces héritées

```
public interface Pilotable extends GestionnaireMoteur,  
    GestionnaireTrainAtterrissage {  
    public static final int ALTITUDE_SECURITE = 100;  
  
    public long getAltitude();  
    public void inclinerVolets(long angle);  
    public void setPositionManche(long position);  
    public void virer(long angle);  
}
```



- Redéfinition de toutes les méthodes dans les classes d'implémentation

Interface en Java 8

245

- Depuis Java 8 il est possible de réaliser des méthodes concrètes dans vos interfaces
- Elles doivent toujours être public
- Elles feront usage du mot clef **default**
 - ➡ Il ne faut pas y adjoindre le mot clef *abstract*
- Elles ne peuvent faire usage que des méthodes déclarées dans l'interface (ou ses parents)
 - ➡ Elles peuvent aussi faire usage des attributs (*final static*).

Interface en Java 9

246

- Par défaut, si vous n'indiquez pas de visibilité sur une méthode (ou un attribut) d'une interface :
 - ▶ Elle ne peut être que public (Java <= 7)
 - ▶ Elle est de la visibilité de l'interface (Java >= 8)
- En Java 9, vous pouvez déclarer des méthodes *private* dans vos interfaces
 - ▶ Elles ont les mêmes contraintes que les méthodes d'interface du Java 8 mais il ne faut pas y adjoindre le mot clef *default*

Interface en Java 8

247

- Dans le cas où j'ai deux méthodes ayant la même signature, marquées *default* dans deux interfaces différentes

```
public interface InterfaceA {  
    public default void doIt() {  
        System.out.println("ok-A");  
    }  
}
```

```
public interface InterfaceB {  
    public default void doIt() {  
        System.out.println("ok-B");  
    }  
}
```

```
public class ClasseAB implements InterfaceA, InterfaceB {  
    // Ne compile pas car problème avec la méthode doIt()  
}
```

- Il faudra lever l'ambiguïté
 - ➡ En overriding la méthode dans ClasseAB
 - ➡ En supprimant une des méthodes d'une des interfaces

Interface en Java 8 et méthodes static

Java 8

248

- Depuis Java 8 il est aussi possible de réaliser des méthodes **static** dans vos interfaces
- Elles doivent toujours être public
- Elles feront usage du mot clef **static**
 - ▶ Il ne faut pas y adjoindre le mot clef *abstract ou default*
- Elles ont le comportement normal d'une méthode static
 - ▶ Elles ne peuvent faire usage que d'éléments static
- Lors de l'usage d'une méthode static d'interface vous serez dans l'obligation de préfixer l'appel par le nom de l'interface.
 - ▶ Sinon vous ne compilerez pas

Interface en Java 8

249

➤ Exemple de méthode static d'interface

```
public interface InterfaceA {  
    public static void doIt() {  
        System.out.println("ok-A");  
    }  
}
```

```
public class ClasseAB implements InterfaceA{  
    public void realiser() {  
        InterfaceA.faireA();  
        ClasseAB.faireA(); // Ne compilera pas  
        this.faireA(); // Ne compilera pas  
    }  
}
```



250

Travaux pratiques

Réaliser les travaux pratiques suivants:

TP 3 partie 3

Inner classe

251

- Il est possible *d'instancier* une interface ou une classe abstraite
- Ce n'est pas exactement une instantiation au sens classique du terme, mais plus une forme de simplification d'écriture
- En fait, la JVM va créer à la volée une classe à partir de l'interface (ou de la classe abstraite) et l'instancier pour nous.
- Cette classe, créée par la JVM, est anonyme, elle ne porte pas de nom. Elle n'est pas réutilisable comme les autres classes.

Inner classe

252

- Voici une interface avec deux méthodes :

```
1 package com.exemple;
2
3 public interface IFaireQQc {
4     public int faireA();
5
6     public void faireB(float unChiffre);
7 }
```

- Nous pourrions faire la même chose avec une classe abstraite.
- On utilise très souvent la technique des inner classes dans la réalisation d'interface graphique (JavaFX, Swing, Android, ...)
- Vous n'êtes pas obligé d'en faire usage, c'est une facilité syntaxique.
- Elle est complétée en Java8 par les *lambda expressions*

Inner classe

253

➤ Voici à quoi ressemble le code qui va créer une classe anonyme à partir de cette interface.

```
1 package com.exemple;
2
3 public class Run {
4     public static void main(String[] args) {
5         // Creation d'une classe a partir d'une interface
6         IFaireQQc obj = new IFaireQQc() {
7             @Override
8             public void faireB(float pUnChiffre) {
9                 System.out.println("Je suis dans une classe anonyme qui faitB");
10            }
11
12            @Override
13            public int faireA() {
14                System.out.println("Je suis dans une classe anonyme qui faitA");
15                return 56;
16            }
17        };
18
19        // utilisation de ma classe anonyme
20        obj.faireA();
21        obj.faireB(66F);
22    }
23 }
```

```
new NomInterface() {
    methode1
    methode2
};
```

Inner classe : Passes droit

254

- Une inner classe a quelques passe-droits :
 - Elle peut discuter en directe avec sa classe englobante (celle où elle se trouve)
 - ❑ Accéder aux méthodes et attributs sans tenir compte des visibilités
 - ❑ Pour garantir une bonne lisibilité il est conseillé d'utiliser la syntaxe NomDeClasse.this.nomAttribut ou NomDeClasse.this.nomMethode(...)
 - Elle peut accéder aux variables, paramètres de la méthode où elle se trouve
 - ❑ avec quelques restrictions, elle ne peut le faire que si ses éléments sont **final**
 - En Java < 8 vous devez indiquer explicitement final
 - En Java >= 8 ses éléments passeront implicitement final, vous n'avez pas à l'indiquer

Inner classe

255

► Exemple de passes droit :

```
3 public class Run {  
4     private double tva;  
5  
6     private void faireC() { System.out.println("Je suis dans la methode faireC de la classe Run"); }  
7  
8     public void faireQQc(String unArgDeFaireQQc) {  
9         int i = 3;  
10        // Creation d'une classe a partir d'une interface  
11        IFaireQQc obj = new IFaireQQc() {  
12            @Override  
13            public void faireB(float pUnChiffre) {  
14                System.out.println("Je suis dans une classe anonyme qui faitB");  
15                // Je peux faire usage de i qui est une variable locale à la methode faireQQc  
16                int j = i + 3;  
17                // Attention : en Java < 8, i devrait être final  
18  
19                // Je peux appeler la methode faireC de la classe Run  
20                Run.this.faireC();  
21                // La syntaxe Run.this n'est pas obligatoire mais elle est conseillée  
22            }  
23  
24            @Override  
25            public int faireA() {  
26                System.out.println("Je suis dans une classe anonyme qui faitA");  
27                // Je peux faire usage de unArgDeFaireQQc qui est un parametre de la methode faireQQc  
28                System.out.println(unArgDeFaireQQc);  
29                // Attention : en Java < 8, unArgDeFaireQQc devrait être final  
30  
31                // Je peux utiliser l'attribut tva de la classe Run  
32                double r = Run.this.tva * 25.3D;  
33                // La syntaxe Run.this n'est pas obligatoire mais elle est conseillée  
34                return 56;  
35            }  
36        };  
37  
38        // utilisation de ma classe anonyme  
39        obj.faireA();  
40        obj.faireB(66F);  
41    }  
42 }
```

Quizz

256

- Pourquoi créer des constructeurs dans une classe abstraite ?
- Est-ce que je peux avoir des méthodes concrètes dans une interface ?
 - ➡ En Java < 8
 - ➡ En Java ≥ 8
- Donnez les 4 visibilités disponibles en Java de la plus contraignante à la plus ouverte

Quizz (2)

257

➤ Que faut-il faire pour que ce code compile ?

```
public abstract class Collaborateur {  
    protected String nom;  
    protected String prenom;  
  
    public Collaborateur(String nom, String prenom) {  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
  
    public abstract long calculerSalaire();  
  
    public void annoncerSalaire() {  
        System.out.println("Salaire : " + calculerSalaire());  
    }  
}
```

```
public class Mecanicien extends Collaborateur {  
    public void reparer(Avion unAvion) {  
        // code pour reparer un avion  
    }  
}
```

Synthèse

➤ Notions

- ▶ Super-classe
- ▶ Classe abstraite
- ▶ Interface

➤ En pratique, qu'utilise-t-on ?

- ▶ Différents cas d'utilisation
- ▶ Demander à son architecte

Cas pratique 1

- On a déjà une classe (Pilote) et on veut créer un cas particulier (PilotInstructeur)
 - ➡ PilotInstructeur hérite de Pilote
 - ❑ Pilote reste inchangé
 - ❑ Ajout des attributs spécifiques à un instructeur
 - ❑ Ajout des méthodes spécifiques à un instructeur
 - ❑ Redéfinition des méthodes pour lesquelles l'instructeur répond différemment du pilote

Cas pratique 2

- On a plusieurs classes existantes et on s'aperçoit que beaucoup d'attributs et méthodes sont dupliqués
 - ➡ Techniquement
 - ❑ Création d'une super-classe
 - ❑ Changement du lien d'héritage pour les classes existantes
 - ❑ Déplacement des méthodes et des attributs communs vers la super-classe
 - ➡ Y a-t-il un sens à créer des instances de la superclasse, ou ne doit-on manipuler que les sous-classes ?
 - ❑ Si oui, la superclasse est concrète
 - ❑ Si non, la super-classe est abstraite

Cas pratique 3

➤ On veut cacher la complexité d'une classe ou ne montrer qu'une partie de ses méthodes

❑ Ex : TP

► Techniquement

❑ Créer une interface qui ne déclare que les méthodes à exposer

❑ Utiliser l'interface comme type de retour et non la classe d'implémentation

○ Les clients ne doivent connaître/manipuler que l'interface

Cas pratique 4

- On veut pouvoir changer facilement l'implémentation d'un objet
 - ❑ Ex : changer la persistance d'un objet – SGBDR – fichier texte
 - ❑ Ex : remplacer l'objet par une implémentation de test (pilote avec simulateur / avion)
- Techniquement
 - ❑ Créer une interface qui offre les mêmes services de la classe
 - ❑ Référencer cette interface plutôt que la classe
 - ❑ Paramétriser la création de l'objet

Les exceptions

263

Approche conventionnelle
Gestionnaire d'exceptions
Objet de type Exception
Code protégé
Hiérarchie des exceptions

Approche conventionnelle

- Historiquement: utilisation de codes d'erreur
 - les fonctions retournent des codes
 - tester les codes à chaque appel de fonction
 - traiter les erreurs ou continuer l'exécution
- Maintenance difficile
- Pas de catégorie d'erreurs
- Risque de cas d'erreurs non traitées

Principaux rôles des exceptions

- Séparer les traitements “exceptionnels” du code “normal”
 - ➡ Eviter les structures conditionnelles difficiles à maintenir
- Propager les erreurs dans la pile de messages
 - ➡ Mécanisme très simple et intuitif pour le développeur
- Classifier les erreurs
 - ➡ Par la création de classes représentant les exceptions
 - ➡ Utilisation de l'héritage pour regrouper les exceptions

Gestionnaire d'exceptions

- Pour traiter une anomalie
- « Envoi » d'un objet Exception
- Recherche du gestionnaire approprié
 - ➡ Si trouvé (dans la pile d'appels)
 - ❑ Traitement de l'exception
 - ❑ Retour au programme
 - ➡ Sinon
 - ❑ Déclenchement d'une erreur et arrêt de l'exécution

Objet de type Exception

- Creation d'un objet Exception
 - ▶ en general sous-classe de **java.lang.Exception**
 - ▶ **new MaClasseException();**

- Envoi de l'exception
 - ▶ Expression a inserer dans le code de la methode
 - ▶ instruction **throw**
 - **throw new MaClasseException();**

Déclaration des exceptions lancées

268

- La méthode “contient” une liste d’exceptions **potentielles qu’elle peut (ou non) lever**
- Il faut spécifier toutes les exceptions qui peuvent être envoyées dans la déclaration de la méthode via le mot clef **throws**

```
public void calculer() throws E1, E2, E3 {  
    // ...  
    if (x == 0) throw new E1();  
    if (x == 3) throw new E3();  
    // ...  
}
```

E1, E2, E3
représente la liste
d’exceptions
pouvant être
levées par la
méthode

Code protégé

- Blocs try-catch
 - ➡ Permet de sécuriser et de rattraper les exceptions

- Séparation exécution / traitement d'erreur

```
try {  
    // Exécutions d'instructions sécurisées  
} catch (TypeException e) {  
    // Gestion du problème lié à TypeException ou enfant  
}
```

Exemple

270

➤ Exemple d'exception

```
public double diviser(int a, int b) throws DivException {
    if (b == 0) {
        throw new DivException("division par zéro");
    }
    return a / b;
}

public double appelerDiviser(int x, int y) {
    try {
        return diviser(x, y);
    } catch (DivException e) {
        return 0;
    }
}
```

Code protégé - Contraintes

271

- Vous pouvez mettre autant de blocs de catch que vous voulez à partir du moment où :
 - ▶ Ils catch un Throwable (ou enfant)
 - ▶ Vous les placez du **plus spécifique au moins spécifique**
 - Le catch le plus précis en premier, me plus large en dernier
- Le compilateur regardera les lignes de code du bloc de try
 - ▶ Vous ne pouvez pas faire un catch sur un Throwable qui ne peut pas arriver dans le bloc de try

Exemple

272

➤ Exemple d'exception

```
public double diviser(int a, int b) throws DivException {
    if (b == 0) {
        throw new DivException("division par zéro");
    }
    return a / b;
}

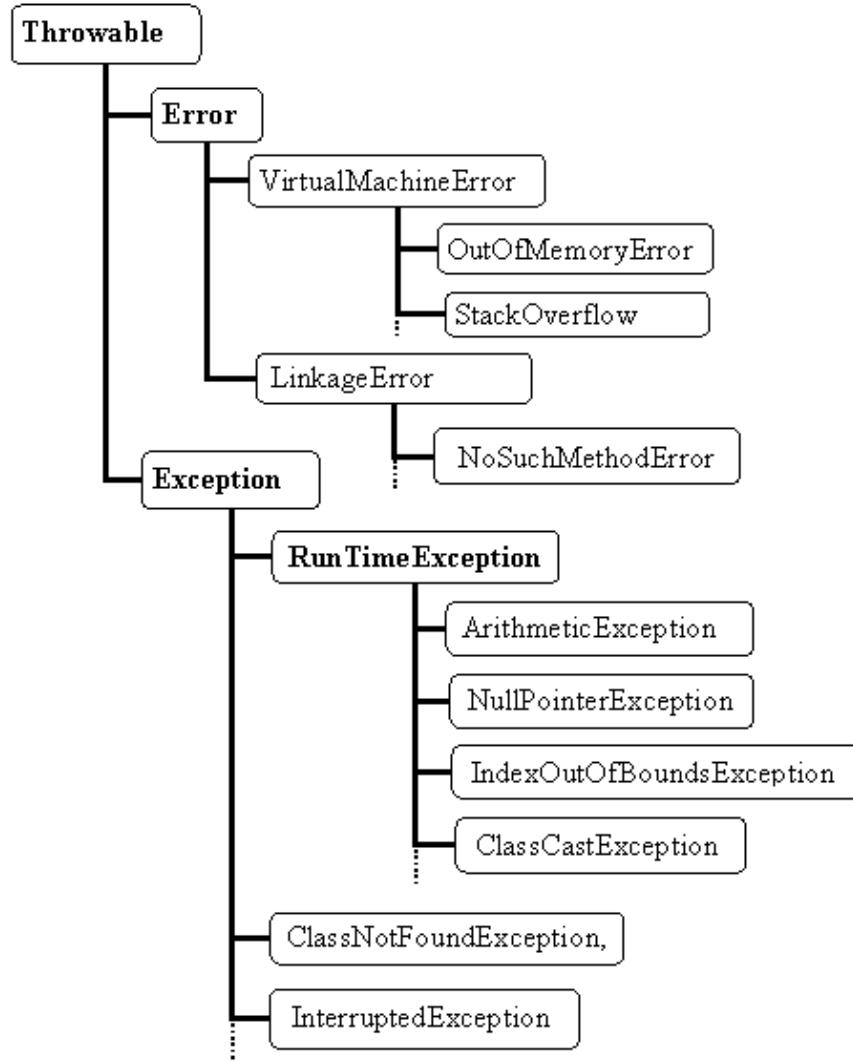
public double appelerDiviser(int x, int y) {
    try {
        return diviser(x, y);
    } catch (DivException e) { // Le plus spécifique en premier
        return 0;
    } catch (Exception e) { // Le moins spécifique en dernier
        return -1;
    }
}
```

Hiérarchie des exceptions (1/2)

- Mère de toutes les exceptions : classe **Throwable**
 - ▶ Signification: « objet qui peut être renvoyé»
 - ▶ Contient des méthodes générales
- 2 sous-classes de Throwable = 2 catégories principales:
 - ▶ **Error** (erreurs système et de compilation)
 - ▶ **Exception**
 - **RuntimeException** (erreurs à l'exécution)
 - ... (erreurs applicatives)

Hiérarchie des exceptions (2/2)

274



Opérations sur une exception

- Une exception peut contenir/porter des messages ou des informations
- Exemples
 - ▶ e.getMessage()
 - ▶ e.printStackTrace()
 - ▶ e.getCause() (à partir de Java 1.4)...

Compilation des programmes

➤ **java.lang.Exception**

► **Doivent** être gérées et traitées

□ try-catch

□ ou throws NomException

○ la méthode ne traite pas l'exception mais la propage

► Sinon erreur de compilation

➤ **java.lang.RuntimeException ou java.lang.Error**

► Pas besoin de try-catch, pas de problème à la compilation, mais erreur déclenchée à l'exécution

► Deux causes :

□ Mauvais développement → à corriger

□ Erreur déclenchée par le système (imprévisible) → pas de correction, sinon code illisible

Créer une nouvelle exception

277

➤ Créer une nouvelle classe

- ▶ Sous-classe directe de Exception ou de RuntimeException
- ▶ Ou utiliser les hiérarchies existantes

```
public class DecollageException extends Exception {  
    public DecollageException() {  
        super();  
    }  
  
    public DecollageException(String msg) {  
        super(msg);  
    }  
}
```

Vos Exceptions

278

- **99,5%** de vos exceptions hériteront de **Exception** (ou *enfant sans être une RuntimeException*)
 - Se sont les **checked** exceptions car il faut les rattraper et elles conditionnent la compilation
- **0,49%** de vos exceptions hériteront de **RuntimeException** (ou *enfant*)
 - Se sont les **unchecked** exceptions car il ne faut pas les rattraper (obligatoirement), elles ne conditionnent pas la compilation
- **0,01%** de vos exceptions hériteront de **Error** (ou *enfant*)
 - Erreurs lourdes liées à des problèmes hardware. Notez que ses enfants de Throwable n'ont pas obligation d'être rattrapés (ils ont un comportement équivalent aux RuntimeException mais si elles ne sont pas des parents directs)

Bloc finally

279

- **finally** assure l'exécution d'un bloc d'instructions même en cas d'exception (**finally** = *quoi qu'il arrive*)
- Assurer un état cohérent dans le programme
 - Fermeture de fichier ouvert *quoi qu'il arrive*
 - Fermeture d'une base de données *quoi qu'il arrive*

```
try {....}           // code protégé
catch (...) {...}   // plusieurs gestionnaires
catch (...) {...}   // d'exceptions
finally {...}       // bloc toujours exécuté
```

Bloc finally - Cas spéciaux

280

- Si la méthode réalise un throws, un try dans cette méthode peut se passer de son catch, mais, dans ce cas particulier, le finally devient obligatoire.
 - ➡ Sinon, un try ne peut pas exister sans catch
- Si vous placez un **return** dans un try ou un catch
 - ➡ Le *finally* sera exécuté avant le **return**
- Si vous placez un **System.exit** dans un try ou un catch
 - ➡ Le *finally* ne sera pas exécuté avant le **System.exit**

Contraintes liées à la redéfinition

- Une méthode redéfinie dans une sous-classe :
 - ➡ Ne doit pas ajouter d'exception supplémentaire à la liste d'exceptions de la méthode qu'elle redéfinie
 - Sauf dans le cas d'un constructeur
 - ➡ Peut déclarer une liste plus restreinte d'exceptions
 - ➡ Peut déclarer une exception plus spécialisée

Gestion du finally par ressource

282

- L'écriture du finally peut parfois être lourde, depuis le **Java 7** il est possible d'utiliser une autre forme d'écriture qui va indiquer à la JVM de faire le finally **automatiquement**.
- Cette technique n'est applicable qu'avec des objets implémentant l'interface **java.lang.AutoCloseable**
 - ➡ C'est déjà le cas de 95% des objets que l'on manipule dans les finally

Gestion du finally par ressource

283

```
1① import java.io.BufferedWriter;
2 import java.io.FileWriter;
3
4 public class Exemple {
5
6②     public static void main(String[] args) {
7         // fw et writer seront fermés quoi qu'il arrive
8         try {
9             FileWriter fw = new FileWriter("dest.txt");
10            BufferedWriter writer = new BufferedWriter(fw);
11        }
12
13        int nombre = 123;
14        writer.write("Bonjour tout le monde");
15        writer.newLine();
16        writer.write("Nous sommes le " + new java.util.Date());
17        writer.write(", le nombre magique est " + nombre + ".");
18    } catch (Exception e) {
19        e.printStackTrace();
20    }
21    // Surtout pas de finally ici
22 }
23 }
```

Java 7

Gestion du finally par ressource

284

- le **try** s'accompagne d'un bloc '()' dans lequel on déclare des variables
 - ▶ locales au bloc de **try**
 - ▶ et qui implémentent l'interface **AutoCloseable**
- Quoiqu'il arrive, la méthode `close()` de ses variables seront automatiquement appelées dès la fin du bloc de **try**

Catch multivalué

285

- Depuis le Java 8 il est possible d'écrire des catch multivalués
 - Ils gèreront plusieurs exceptions via l'opérateur | (ou)

```
6 public class Exemple {  
7  
8     public static void main(String[] args) {  
9         // fw et writer seront fermés quoi qu'il arrive  
10        try (FileWriter fw = new FileWriter("dest.txt");  
11            BufferedWriter writer = new BufferedWriter(fw)) {  
12            int nombre = 123;  
13            writer.write("Bonjour tout le monde");  
14            writer.newLine();  
15            writer.write("Nous sommes le " + new java.util.Date());  
16            writer.write(", le nombre magique est " + nombre + ".");  
17  
18        } catch (NullPointerException | IOException e) {  
19            // Traitement si l'exception est de type  
20            // - NullPointerException  
21            // OU - IOException  
22        }  
23    }  
24}  
25 }
```

Java 8

Rappel : Override et exceptions

286

- Quand une classe parent possède une méthode levant un type d'exception et devant être overriding par un enfant, ce dernier peut
 - ➡ Reprendre le prototypage du parent tel quel (conseillé)
 - ➡ Reprendre le prototypage du parent en modifiant la clause throws par des enfants des exceptions spécifiées par le parent
 - ➡ Reprendre le prototypage du parent en modifiant la clause throws par des RuntimeException (ou enfant)
 - ➡ Reprendre le prototypage du parent sans reprendre la clause throws (du coup ne lever aucune exception)



287

Travaux pratiques

Réaliser les travaux pratiques suivants:

TP 3 partie 4

Les classes de base

288

Wrappers

Boxing / Unboxing

String

StringBuffer / StringBuilder

Dates

Les wrappers

289

- Les types primitifs permettent d'effectuer des opérations simples et rapides

```
int i;  
int j;  
i = 2 + 3;  
j = i + 2;  
j++;
```

- Pour chaque type primitif il existe une classe dite wrapper (emballage).
 - ➡ Une classe wrapper contient une variable d'un type primitif donné et des outils facilitant la manipulation de cette variable.

java.lang.Integer



int

java.lang.Long



long

java.lang.Character



char

• • •

Les wrappers (2)

- Classes wrapper existantes classiques :
 - ➡ Boolean, Byte, Short, Integer, Long, Float, Double, Character
- Méthodes proposées par les classes wrapper :
 - ➡ **toString()** convertit en chaîne de caractères la variable contenue dans le Wrapper
 - ➡ **intValue()**, **doubleValue()**, ... pour récupérer le type primitif encapsulé par le Wrapper.
 - ➡ **parseInt()**, **valueOf()**, ... pour des conversions de texte vers des nombres (type primitif et Wrapper).
 - ➡ Les wrappers de types numériques contiennent des constantes **MAX_VALUE** et **MIN_VALUE**

Les wrappers (3)

➤ Exemples d'utilisation

► Méthodes d'instance

```
Integer i = new Integer(12);
String s = i.toString();
double d = i.doubleValue();
Boolean b = new Boolean(true);
boolean b2 = b.booleanValue();
```

► Méthodes statiques

```
String s = "12";
int i = Integer.parseInt(s);
boolean b = Boolean.valueOf("true");
```



La plupart de ces méthodes ne s'appliquent pas au type Character qui n'enveloppe pas un nombre.

Les wrappers (4)

➤ Classes spéciales de Wrappers

- ▶ `java.util.concurrent.atomic.AtomicInteger`,

- ▶ `java.util.concurrent.atomic.AtomicLong`,

- Garanti que la modification de la valeur est atomic ⇔ permet une utilisation sécurisée en environnement multi thread

- ▶ `java.math.BigDecimal`

- ▶ `java.math.BigInteger`,

- Pour gérer les très très gros chiffres (>64bits)

Les wrappers (5)

- Classes spéciales de Wrappers
 - ▶ `java.util.concurrent.atomic.DoubleAccumulator`,
 - ▶ `java.util.concurrent.atomic.LongAccumulator`
 - Généralement utilisés en environnement multi thread et via une lambda afin de garder un chiffre à jour
 - S'avèrent plus efficaces que les AtomicXxx lors de calculs
 - ▶ `java.util.concurrent.atomic.DoubleAdder`,
 - ▶ `java.util.concurrent.atomic.LongAdder`
 - Même objectif que les Accumulator, mais plus dans une mécanique d'addition que de modification du chiffre
- <https://www.baeldung.com/java-longadder-and-longaccumulator>

Unboxing / Autoboxing

- Depuis Java 5, le mécanisme de boxing permet de convertir automatiquement :

- ▶ un type primitif en wrapper (Autoboxing)

```
int x = 567;  
Integer y = x;
```

- ▶ un wrapper en type primitif (Unboxing)

```
Integer y = new Integer(567);  
int x = y;
```

- ▶ Exemple d'utilisation :

Avant Java 5

```
Integer y = new Integer(567);  
int x = y.intValue();  
x++;  
y = new Integer(x);  
System.out.println("y="+y);
```

```
Integer y = new Integer(567);  
y++;  
System.out.println("y="+y);
```

Depuis Java 5

String

295

- Les String représentent les chaînes de caractères en Java
 - ▶ Les Strings sont des objets invariants
 - ▶ Initialisation facilitée sans le mot-clé new
 - ▶ La taille d'une String n'est pas limitée

```
String s1 = new String("hello world");  
  
// privilégier la façon de faire suivante  
String s2 = "hello world";
```

String (2)

➤ Quelques méthodes utiles

► Comparaisons

- int compareTo(String)
- int compareIgnoreCase(String)

```
String s1 = new String("hello");
String s2 = new String("hello");
System.out.println(s1.compareTo(s2));
```

► Longueur de la chaîne

- int length() (ceci est la méthode length())

```
String s1 = new String("hello");
int longueurChaine = s1.length();
System.out.println(longueurChaine);
```

String (3)

➤ Manipulation de chaînes

- trim() : supprime les espaces superflus

```
String s1 = new String("      hello world ");
System.out.println(s1);
System.out.println(s1.trim());
```

- toUpperCase(), toLowerCase() ...

```
String s1 = new String("hello world");
System.out.println(s1.toUpperCase());
```

String (4)

298

➤ Concaténation

► concat(...) ou opérateur +

```
String s1 = new String("hello");
String s2 = s1 + (" world");
String s3 = s1.concat(" world");
```

➤ Les instances de String sont dites immuables

```
String s = "hello world";
s = "bye bye world";
```

⚠ Utilisée à grande échelle (dans des boucles), la concaténation de String est une opération peu performante.

Le premier objet référencé par s est détruit puis s référence un nouvel objet.

String (5)

299

➤ Découpage

- ▶ substring(début)
- ▶ substring(début, fin)

```
String test = "0123456789";
System.out.println(test.substring(3, 4)); // "3"
System.out.println(test.substring(3, 3)); // ""
System.out.println(test.substring(5)); // "56789"

// Lève une exception StringIndexOutOfBoundsException
// System.out.println(test.substring(15));
// System.out.println(test.substring(5, 4));
```

➤ Attention : substring ne change pas l'état de la chaîne, elle **retourne** une chaîne découpée

StringBuffer / StringBuilder

- Depuis 5.0, on peut utiliser la classe `StringBuilder` (du package `java.lang`) à la place de la classe `StringBuffer`.

Types	Version JDK	Synchronisation ↔ Thread Safe	Performance
StringBuffer	à partir du 1.0	Oui	++
StringBuilder	à partir du 1.5	Non	+++++

StringBuffer / StringBuilder

301

- Ne sont pas des invariants
- Du coup, il est possible d'y insérer / supprimer des éléments

```
StringBuilder test = new StringBuilder("0123456789");
// substring ne change pas l'état du StringBuilder
// Elle retourne un StringBuilder découpé
System.out.println(test.substring(3, 4)); // "3"
System.out.println(test.substring(3, 3)); // ""
System.out.println(test.substring(5)); // "56789"

// delete et insert vont modifier l'état du StringBuilder
// Elles retournent aussi le StringBuilder modifié
System.out.println(test.delete(2, 4)); // "01456789"
System.out.println(test.insert(1, "alpha")); // "0alpha1456789"
```

String / StringBuffer / StringBuilder

302

- Critère de sélection entre String, StringBuffer et StringBuilder:
 - ▶ Si la chaîne de caractères ne va pas changer, utilisez la classe String parce qu'un objet de type String est un objet non mutable.
 - ▶ Si votre chaîne de caractères va beaucoup changer et est accédée par un seul thread, utilisez StringBuilder parce que StringBuilder n'est pas synchronisé et sera donc plus rapide que StringBuffer.
 - ▶ Si votre chaîne de caractères va beaucoup changer et est accédée par plusieurs threads, utilisez StringBuffer parce que StringBuffer est synchronisé.

```
StringBuilder message = new StringBuilder();  
message.append("msg 1");  
message.append( "msg 2");  
message.append(" msg 3");  
System.out.println(message);
```

Flux standards

- Sortie standard : System.out
- Erreur standard : System.err
- Écrire
 - ▶ print(valeur)
 - ▶ println(valeur)
 - valeur : type primitif ou objet
 - ▶ printf(message, args)
 - message : chaîne de caractère avec paramètres
 - args : primitives ou objets à remplacer dans le message

Méthode printf()

- Java 5 introduit une nouvelle méthode permettant de produire des messages formatés
 - ▶ Pattern identique à printf en langage C
 - ▶ Définit dans java.util.Formatter
 - ▶ Raccourcis: String.format(), System.out.format(), System.out.printf()

```
double a = 3.87;
double b = 5.0;
double mult = a * b;
System.out.println(a + " * " + b + " = " + mult);
```

```
double a = 3.87;
double b = 5.0;
double mult = a * b;
System.out.printf("%2$.2f * %1$.2f = %3$.2f", a, b, mult);
// affiche: 5,00 * 3,87 = 19,35
```

Java 5

Dates

305

► `java.util.Date`

- La plupart des méthodes sont dépréciées : existent mais ne doivent pas être utilisées car elles peuvent disparaître dans une future version du JDK
- Pas de gestion des zones (Locale)

► `java.text.DateFormat` et `java.text.SimpleDateFormat`

- conversion Date/String
- constructeur avec un motif pour la conversion
- String ⇒ Date : `Date parse(String)`
- Date ⇒ String : `String format(Date)`

Calendriers

306

► `java.util.Calendar`

- ❑ Tient compte de la zone (fuseaux horaires, etc)
- ❑ Référencent une date/heure avec accès au jour, heure, secondes...
- ❑ `Calendar.getInstance()`
- ❑ `getTime()` : retourne la `Date` associée à ce calendrier

Dates et calendriers

307

```
Calendar calendrierMaintenant = Calendar.getInstance();  
  
Date maintenant = calendrierMaintenant.getTime();  
  
SimpleDateFormat formattage = new SimpleDateFormat("EEEE dd MMMM yyyy");  
  
System.out.println(formattage.format(maintenant));  
// affiche: samedi 31 janvier 2009
```

Dates en Java 8

308

- Une nouvelle API avec de nouveaux objets plus simples : `java.time`

```
// Maintenant
LocalDateTime timePoint = LocalDateTime.now();
// Le 12 decembre 2012
LocalDate d1 = LocalDate.of(2012, Month.DECEMBER, 12);
// milieu de l'annee 1970
LocalDate d2 = LocalDate.ofEpochDay(180);
// 17h 18
LocalTime d3 = LocalTime.of(17, 18);
// A partir d'une chaine
LocalTime d4 = LocalTime.parse("10:15:30");

LocalDate theDate = timePoint.toLocalDate();
Month month = timePoint.getMonth();
int day = timePoint.getDayOfMonth();
int sec = timePoint.getSecond();
```

Dates en Java 8

309

- Les objets ne s'instancient jamais.
- On les récupère via des méthodes statiques.
- Il est possible de chainer les appels (chacun d'eux retournant un objet *cible*)
- Ne JAMAIS mélanger les APIs du Java 8 avec celles du Java 8-
 - ➡ Calendar.MAY = 4 alors que Month.MAY = 5
- Contrairement au SimpleDateFormat, cette API ne tolère pas les écarts
 - ➡ Lèvera une exception :

```
LocalDate date = LocalDate.of(2014, Month.JANUARY, 50);
```

Dates en Java 8

310

- **LocalDate** : Juste une date, sans temps ni time zone

```
System.out.println(LocalDate.now()); // 2015-01-20
```

- **LocalTime** : Juste une heure/minute/seconde/millisecond/nanosecond sans time zone

```
System.out.println(LocalTime.now()); // 12:45:18.401
```

- **LocalDateTime** : Regroupe LocalDate+LocalTime, toujours sans time zone

```
System.out.println(LocalDateTime.now()); // 2015-01-20T12:45:18.401
```

Dates en Java 8

311

➤ Opérations sur les dates (ajout)

```
LocalDate date = LocalDate.of(2014, Month.JANUARY, 20);
System.out.println(date); // 2014-01-20
date = date.plusDays(2);
System.out.println(date); // 2014-01-22
date = date.plusWeeks(1);
System.out.println(date); // 2014-01-29
date = date.plusMonths(1);
System.out.println(date); // 2014-02-28
date = date.plusYears(5);
System.out.println(date); // 2019-02-28
```

Dates en Java 8

312

- Notez que les méthodes plus/minus ne changent pas la date
- Les éléments de l'API sont des invariants (comme la String ou Class)
- Les méthodes retournent le résultat attendu

Dates en Java 8

313

➤ Opérations sur les dates (suppression)

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(5, 15);
LocalDateTime dateTime = LocalDateTime.of(date, time);
System.out.println(dateTime); // 2020-01-20T05:15
dateTime = dateTime.minusDays(1);
System.out.println(dateTime); // 2020-01-19T05:15
dateTime = dateTime.minusHours(10);
System.out.println(dateTime); // 2020-01-18T19:15
dateTime = dateTime.minusSeconds(30);
System.out.println(dateTime); // 2020-01-18T19:14:30
dateTime = dateTime.minusNanos(30);
```

```
LocalDateTime dateTime = LocalDateTime.of(date, time)
.minusDays(1).minusHours(10).minusSeconds(30);
```

Dates en Java 8 - Les périodes

314

- Il existe un objet représentant une période de temps

```
Period annually = Period.ofYears(1); // tous les ans  
Period quarterly = Period.ofMonths(3); // tous les 3 mois  
Period everyThreeWeeks = Period.ofWeeks(3); // toutes les 3 semaines  
Period everyOtherDay = Period.ofDays(2); // tous les 2 jours  
  
// tous les ans, 0 mois et 7 jours  
Period everyYearAndAWeek = Period.of(1, 0, 7);  
  
// Attention : ici seul le dernier appel est pris en compte  
Period wrong = Period.ofYears(1).ofWeeks(1); // Toutes les semaines
```

- Il suffit de l'utiliser sur son objet cible avec les méthodes **plus** ou **minus**

```
LocalDateTime now = LocalDateTime.now();  
now = now.plus(quarterly);
```

Dates en Java 8 - Les périodes

315

➤ Obtenir une période entre deux dates

```
LocalDate startDate = LocalDate.of(2015, 2, 20);
LocalDate endDate = LocalDate.of(2017, 1, 15);

Period p = Period.between(startDate, endDate);
// 1,10,26
System.out.println(p.getYears() + "," + p.getMonths() + "," + p.getDays());
```

➤ Obtenir une période via une String formatée

```
// Via une chaîne respectant le pattern : PnYnMnD
// 2 ans
Period p1 = Period.parse("P2Y");
// 2 ans, 3 mois et 5 jours
Period p2 = Period.parse("P2Y3M5D");
```

Dates en Java 8 - Les durées

316

- Il existe un objet représentant une durée de temps

```
Duration oneHours = Duration.ofHours(1);  
Duration fromDays = Duration.ofDays(1);  
Duration fromMinutes = Duration.ofMinutes(60);
```

```
Duration oneHours2 = Duration.of(1, ChronoUnit.HOURS);
```

- Il suffit de l'utiliser sur son objet cible avec les méthodes *plus* ou *minus*

```
LocalDateTime now = LocalDateTime.now();  
now = now.plus(oneHours);
```

Dates en Java 8 - Les durées

317

➤ Obtenir une durée entre deux dates

```
LocalTime start = LocalTime.of(1, 20, 25, 1024); // h,m,s,nanos  
LocalTime end = LocalTime.of(1, 22, 25, 2024); // h,m,s,nanos  
  
Duration d = Duration.between(start, end);  
System.out.println(d.get(ChronoUnit.SECONDS)); // 120  
System.out.println(d.getNano()); // 1000  
System.out.println(d.getSeconds()); // 120  
// Ne supporte que nano et seconds  
// Ne marche pas : System.out.println(d.get(ChronoUnit.MINUTES));  
  
// Transformation  
System.out.println(d.toMinutes()); // 2  
System.out.println(d.toMillis()); // 120000  
System.out.println(d.toNanos()); // 12000001000
```

Dates en Java 8 - Les durées

318

- Obtenir une durée via une String formatée

```
// Via une chaîne respectant le pattern : PnDTnHnMn.nS  
Duration d = Duration.parse("P1DT1H10M10.5S");
```

Dates en Java 8 - Formater

319

- Vous pouvez récupérer chaque information via sa méthode `get`

```
LocalDateTime date = LocalDateTime.of(2020, Month.JANUARY, 20, 14, 30, 25, 0);

System.out.println(date.getDayOfWeek()); // MONDAY
System.out.println(date.getMonth()); // JANUARY
System.out.println(date.getYear()); // 2020
System.out.println(date.getDayOfYear()); // 20

System.out.println(date.getHour()); // 14
System.out.println(date.getMinute()); // 30
System.out.println(date.getSecond()); // 25
System.out.println(date.getNano()); // 0
```

Dates en Java 8 - Formater

320

- Vous pouvez aussi utiliser des formats prédéfinis

```
LocalDateTime date = LocalDateTime.of(2020, Month.JANUARY, 20, 14, 30, 25, 0);

// 2020-01-20
System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_DATE));

//14:30:25
System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_TIME));

// 2020-01-20T14:30:25
System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
```

Dates en Java 8 - Formater

321

➤ Vous pouvez aussi faire usage du `DateTimeFormatter`

```
LocalDateTime date = LocalDateTime.of(2020, Month.JANUARY, 20, 14, 30, 25, 0);
```

```
DateTimeFormatter shortDate = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);  
System.out.println(shortDate.format(date)); // 20/01/20
```

```
DateTimeFormatter shortTime = DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);  
System.out.println(shortTime.format(date)); // 14:30
```

```
DateTimeFormatter monFormat = DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm");
```

```
LocalDateTime date = LocalDateTime.of(2020, Month.JANUARY, 20, 14, 30, 25, 0);
```

```
// janvier 20, 2020, 02:30  
System.out.println(monFormat.format(date));
```

Dates en Java 8 - Formater

322

- Pour passer d'une chaîne à une date, vous pouvez faire usage de la méthode `parse` de votre cible (avec ou sans `DateTimeFormatter`)

```
// Avec le DateTimeFormatter
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("MM dd yyyy");
LocalDate date = LocalDate.parse("01 02 2015", formatter);
System.out.println(date); // 2015-01-02
```

```
// Sans le DateTimeFormatter
LocalTime time = LocalTime.parse("11:22");
System.out.println(time); // 11:22
```

Les collections et dictionnaires

323

Les collections

Les Listes

Les Sets

Les Maps

Tri d'éléments

Classes utilitaires des collections

Bonnes pratiques

Limites des tableaux

324

- Les collections sont l'évolution logique des tableaux
- Les tableaux peuvent être utilisés pour des opérations simples
- Mais...
 - ▶ La taille d'un tableau est définie lors de sa création
 - ▶ Si on supprime un élément au milieu d'un tableau, il faut gérer le décalage de tous les éléments qui se trouvaient après l'élément supprimé
 - ▶ Les éléments sont obligatoirement indexés par des entiers

```
String[] tableau = new String[10];
tableau[0] = "Hello !";
tableau[1] = "Bonjour !";
```

Les collections

325

- Les collections sont des classes Java qui facilitent la gestion d'ensembles d'éléments
 - ▶ Opérations courantes :
 - ❑ Ajout d'élément
 - ❑ Suppression d'élément
 - ❑ Parcours de la collection
- De nombreuses collections sont fournies avec Java (depuis le JDK 1.2)
 - ▶ Package `java.util`
 - ▶ Implémentent généralement l'interface `java.util.Collection`
 - ▶ Permettent de stocker des références vers tous types d'objets
 - ▶ Pas de taille maximum prédéfinie
 - ▶ Elles sont optimisées en fonction de besoins précis

Types génériques et collections

- Depuis Java 5, toutes les collections peuvent profiter des types génériques
- Les collections peuvent donc être typées
 - ➡ Les exemples du cours utilisent les types génériques
 - ➡ Par exemple, le code suivant :

```
List<String> noms = new ArrayList<String>();  
noms.add("Fred");  
String nom = noms.get(0);
```

Java 5

- ➡ Est, presque, la même chose que ceci :

```
List noms = new ArrayList();  
noms.add("Fred");  
String nom = (String) noms.get(0);
```

Types génériques et collections

327

- En Java 7, quand vous faites usage d'un générique vous n'êtes pas obligé de repréciser les éléments de typage à droite du signe =

```
List<String> noms = new ArrayList<>();  
noms.add("Fred");  
String nom = noms.get(0);
```

Java 7+

- **Attention** : cela n'est valable qu'à partir de Java 7, c'est une erreur sur une version inférieure du Java.

```
List<String> noms = new ArrayList<String>();  
noms.add("Fred");  
String nom = noms.get(0);
```

Java 7-

Types génériques et collections

328

- En Java 9, vous pouvez initialiser une collection(List ou Set) directement via sa méthode static of

```
List<String> noms = List.of("Fred", "Jhon", "Albert");
```

Java 9

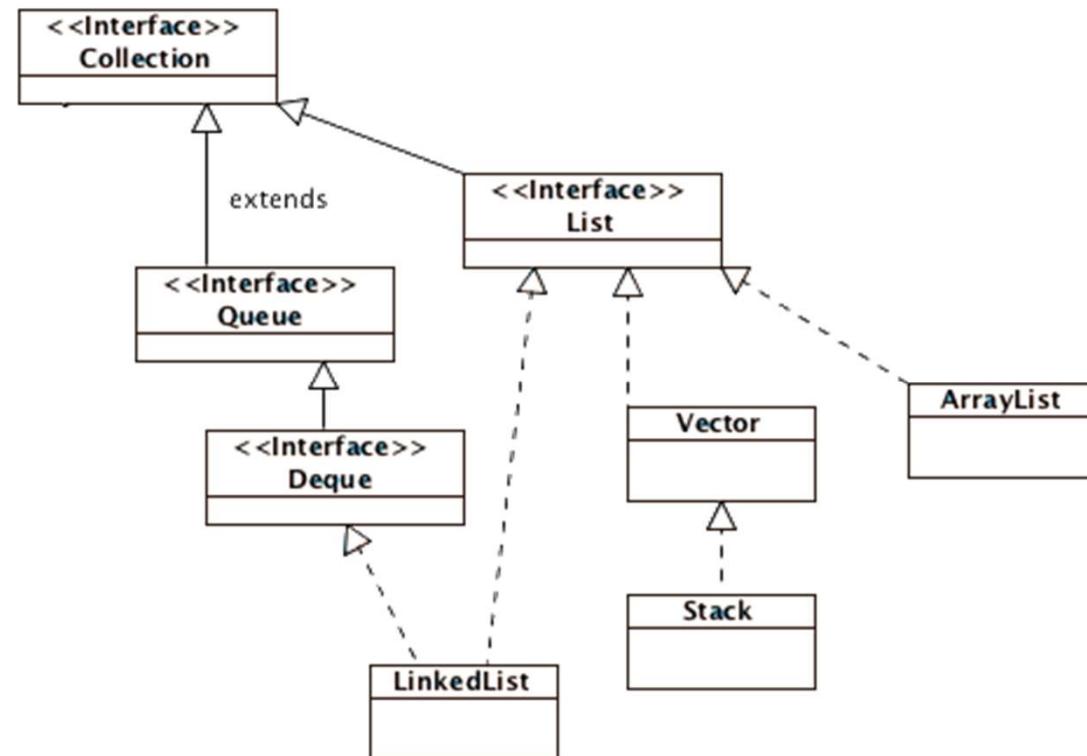
```
Set<String> noms = Set.of("Fred", "Jhon", "Albert");
```

Java 9

Les listes

329

- Les listes sont des collections
 - Implémentent l'interface `java.util.List` qui hérite de l'interface `java.util.Collection`
- Les listes sont indexées
 - Méthode `get(vallIndex)`
- Les collections ne sont pas indexées



Les listes les plus courantes

330

➤ Classe `java.util.ArrayList`

- ▶ La plus utilisée
- ▶ Souvent vue comme un "tableau dynamique"
- ▶ Excellentes performances pour le parcours d'éléments indexés
- ▶ Objet non thread safe

➤ Classe `java.util.Vector`

- ▶ Le plus ancien, retravaillé à chaque version du Java afin de garantir de bonnes performances
- ▶ Souvent vue comme un "tableau dynamique"
- ▶ Excellentes performances pour le parcours d'éléments indexés
- ▶ Objet thread safe

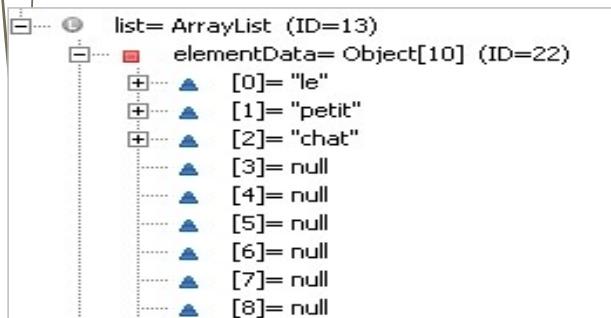
➤ Classe `java.util.LinkedList`

- ▶ Liste chaînée
- ▶ Excellentes performances lors d'insertion/suppression d'éléments
- ▶ Mauvaises performances pour le parcours d'éléments

Ajout d'éléments

331

- Les éléments sont ajoutés avec la méthode add(...)
 - ➡ Par défaut, chaque élément est indexé en fonction de son ordre d'arrivée



```
List<String> list = new ArrayList<>();
list.add("le");
list.add("petit");
list.add("chat");
```

Java 7

Suppression d'un élément

332

➤ Méthode remove(...)

- Possibilité de supprimer l'objet à partir d'une référence à l'objet lui-même, ou à partir de l'index de l'objet dans la liste.

Java 7

```
List<User> list = new ArrayList<>();
User u1 = new User("jean", "dupont");
list.add(u1);
User u2 = new User("jean", "durand");
list.add(u2);
User u3 = new User("jean", "martin");
list.add(u3);

// suppression de la référence vers u3
list.remove(u3);

// suppression de l'élément en position 0
list.remove(0); //
```

Autres méthodes

➤ `size()`

► Renvoie le nombre d'éléments de la collection.

➤ `isEmpty()`

► Renvoie 'true' si la collection est vide

➤ `toArray()`

► Crée un tableau contenant tous les éléments de la liste



Ces méthodes sont déclarées dans l'interface Collection.

Parcours d'une liste

➤ Iterator

- ▶ Permet de parcourir une collection en récupérant les éléments successivement
- ▶ Méthode de parcours homogène pour tous les types de collections
- ▶ Garantit la cohérence de la collection parcourue
 - Permet de retirer d'une collection un élément pointer par l'itérateur
 - Pas de modifications externes autorisées pendant le parcours

Parcours d'une liste (2)

- Récupération d'un Iterator<T> sur une collection donnée
 - ▶ Méthode iterator()
 - ▶ Sur l'objet ArrayList<String>, retourne un Iterator<String>
- Iterator contient 3 méthodes :
 - ▶ boolean **hasNext()**
 - Renvoie true s'il reste des éléments à parcourir dans la liste
 - ▶ T **next()**
 - Renvoie le prochain objet stocké dans la liste
 - ▶ void **remove()**
 - Supprime l'élément en cours de la liste

Parcours d'une liste (3)

336

➤ Exemple

Java 7

```
List<User> list = new ArrayList<>();
User u1 = new User("jean", "dupont");
list.add(u1);
User u2 = new User("jean", "durand");
list.add(u2);
User u3 = new User("jean", "martin");
list.add(u3);

for (Iterator<User> iterator = list.iterator(); iterator.hasNext();) {
    User user = iterator.next();
    System.out.println(user);
}
```



La classe d'implémentation de l'interface Iterator est sans importance.

Parcours d'une liste (4)

➤ Exemple : via l'index

```
List<String> list = new ArrayList<>();  
String s1 = "dupont";  
String s2 = "durand";  
String s3 = "smith";  
list.add(s1);  
list.add(s2);  
list.add(s3);  
  
for (int index = 0; index < list.size(); index++) {  
    String elm = list.get(index);  
    System.out.println(elm);  
}
```



Il est préférable de passer par un Iterateur

Parcours d'une liste (5)

338

➤ Exemple via un ListIterator

```
List<String> list = new ArrayList<>();
String s1 = "dupont";
String s2 = "durand";
String s3 = "smith";
list.add(s1);
list.add(s2);
list.add(s3);

ListIterator<String> lIter = list.listIterator();
while (lIter.hasNext()) {
    String elm = lIter.next();
    System.out.println(elm);
    // lIter.previous();
    // lIter.hasPrevious();
    // lIter.add("Nouvelle");
    // lIter.set("Changement");
}
```

Parcours simplifié d'une liste

339

➤ Nouvelle boucle "for each" depuis Java 5

Java 7

```
List<User> list = new ArrayList<>();
User u1 = new User("jean", "dupont");
list.add(u1);
User u2 = new User("jean", "durand");
list.add(u2);
User u3 = new User("jean", "martin");
list.add(u3);

for (User user : list) {
    System.out.println(user);
}
```

Parcours simplifié d'une liste

340

➤ Via la méthode `forEach` et une lambda expression

Java 8

```
List<Double> list = new ArrayList<>();  
list.add(5.4D);  
list.add(0D);  
list.add(12.8D);  
  
list.forEach( (v) ->System.out.println(" Valeur : " + v));  
  
list.forEach( (v) ->{  
    if(0D == v) {  
        System.out.println("Zero");  
    } else {  
        System.out.println(" Valeur : " + v);  
    }  
} );
```

Gestion des doublons

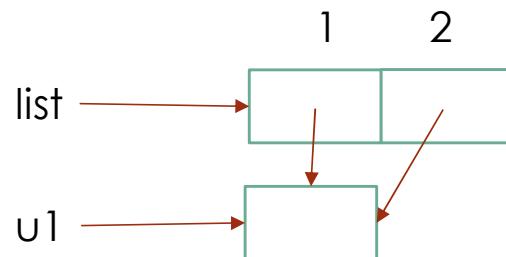
341

➤ Les listes acceptent les doublons

- Ils sont positionnés à un index différent
- La liste stocke plusieurs références vers le même objet

Java 7

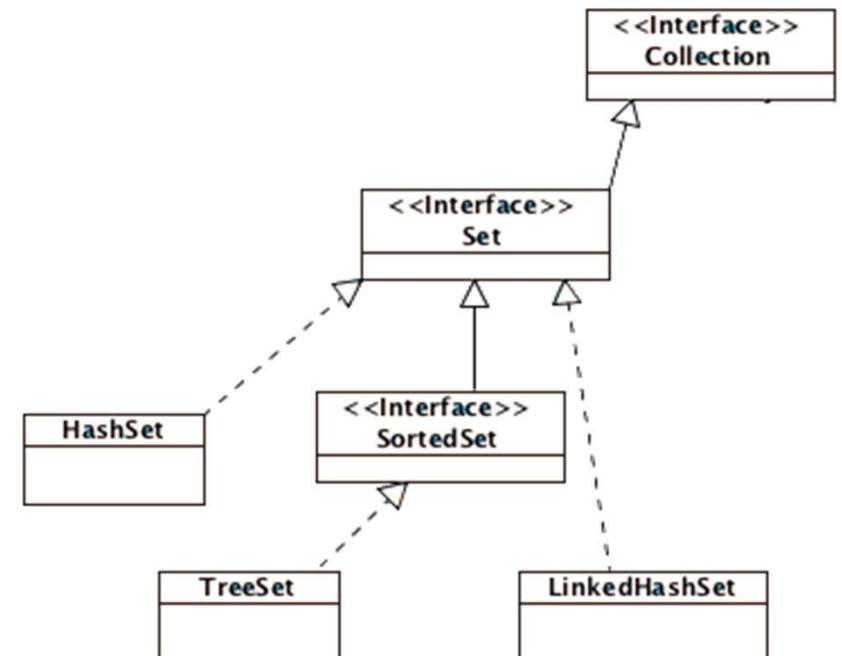
```
List<User> list = new ArrayList<>();  
User u1 = new User("jean", "dupont");  
list.add(u1);  
list.add(u1); // deux références pointent vers le même objet
```



Les sets / Les ensembles

342

- Les sets sont des collections
 - Implémentent l'interface `java.util.Set` qui hérite de l'interface `java.util.Collection`
- Les sets ne sont pas indexés
 - Pas de `get(i)`
- N'acceptent pas les doublons
- Ne garantit pas l'ordre



Les sets les plus courants

➤ Classe `java.util.HashSet`

- ▶ Implémentation de base de l'interface Set
- ▶ Utilise la clé de hachage de ses éléments
- ▶ Set non ordonné

➤ Classe `java.util.TreeSet`

- ▶ Implémente de 2 sous-interfaces de Set :
 - `SortedSet` : force les éléments du set à être triés.
 - `NavigableSet` : permet d'extraire des sous-ensembles.

Java 6

Manipulation des éléments

344

- Les éléments sont ajoutés avec la méthode add(...)
 - ▶ Les éléments ne sont pas indexés.
 - ▶ Tout type d'objet peut être inséré dans un set
 - ▶ Un set peut être typé en utilisant les génériques
- Méthode remove(...)
 - ▶ Suppression de l'objet passé en paramètre
- Un Set peut être parcouru comme une liste (Iterator ou boucle for each)

```
Set<String> set = new HashSet<>();  
set.add("le");  
set.add("petit");  
set.add("chat");  
  
set.remove("petit");
```

Java 7

Gestion des doublons

➤ Les Sets n'acceptent pas les doublons

- Si on ajoute un élément plusieurs fois, il ne sera présent qu'une seule fois dans le set

Java 7

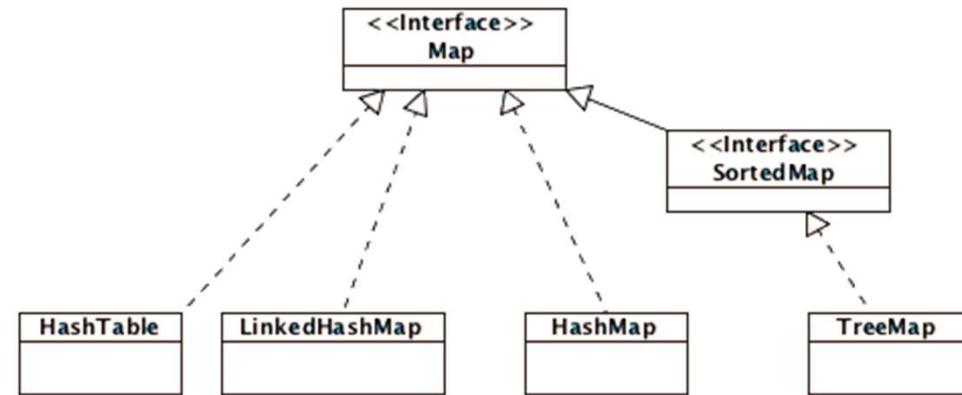
```
Set<String> set = new HashSet<>();  
set.add("le");  
set.add("petit");  
set.add("chat");  
  
set.add("petit"); // sans effet
```



La méthode `add (...)` renvoie un booléen. Elle renvoie `false` si l'élément était déjà présent.

Les maps

- Associations clef-valeur
 - ▶ Aussi appelé dictionnaire
 - ▶ Chaque élément stocké est associé à une clé
 - ▶ Conceptuellement, une map est un ensemble d'éléments
 - ▶ Pour des raisons techniques, l'interface `java.util.Map` n'hérite pas de `java.util.Collection`



Hashtable => Thread safe

Les maps les plus courantes

347

➤ Classe `java.util.HashMap`

- ▶ Implémentation de base de l'interface Map
- ▶ Utilise la clé de hachage de ses éléments
- ▶ Map non ordonnée
- ▶ Non thread safe (donc rapide mais à ne pas partager entre thread)

➤ Classe `java.util.HashTable`

- ▶ Historiquement la plus ancienne
- ▶ Utilise la clé de hachage de ses éléments
- ▶ Map non ordonnée
- ▶ Thread safe (donc plus théoriquement plus lente que HashMap)

Java 6

➤ Classe `java.util.TreeMap`

- ▶ Implémente de sous interfaces de Map
 - SortedMap : force les éléments de la map à être triés selon les clés
 - NavigableMap : permet d'extraire des sous-ensembles.

Ajout et récupération d'éléments

- Les éléments sont ajoutés avec la méthode **put(...)**
 - ➡ **put(K key, V value)**
- Ils sont récupérés avec la méthode **get(...)**

Java 7

```
User u1 = new User("jean", "dupont");
User u2 = new User("jean", "durand");

Map<String, User> map = new HashMap<>();
// utilisation du nom comme clé d'association
map.put(u1.getNom(), u1);
map.put(u2.getNom(), u2);

// récupération en fonction de la clé
User ulbis = map.get("dupont");
```

Suppression d'élément

349

➤ Méthode remove(...)

Java 7

```
User u1 = new User("jean", "dupont");
User u2 = new User("jean", "durand");

Map<String, User> map = new HashMap<>();
// utilisation du nom comme clé d'association
map.put(u1.getNom(), u1);
map.put(u2.getNom(), u2);

// suppression de l'association de la clé dupont (donc u1)
map.remove("dupont");
```



Attention à ne pas passer l'objet mais la clé en paramètre.

Parcours d'une Map

350

➤ Problématique particulière

- ▶ S'agit-il de parcourir les clés ou les valeurs ?
- ▶ Méthode Map.keySet() : permet de récupérer un Set contenant l'ensemble des clés.
- ▶ Méthode Map.values() : permet de récupérer une collection contenant l'ensemble des valeurs.
- ▶ Méthode Map.entrySet() : permet de récupérer un Set contenant l'ensemble des entrées (clef,valeur)

Java 7

```
Map<String, User> map = new HashMap<>();
Iterator<String> keys = map.keySet().iterator();
Iterator<User> values = map.values().iterator();

for (Map.Entry<String, User> entry : map.entrySet()) {
    String key = entry.getKey();
    User value = entry.getValue();
    System.out.println(key + " = " + value);
}
```

Parcours d'une Map

351

➤ Via la méthode `forEach` et une lambda expression

```
Map<String, User> map = new HashMap<>();  
  
// Remplissage ...  
  
map.forEach((k, v) ->  
    System.out.println(" Clef : " + k + " Valeur : " + v));
```

Java 8

Gestion des doublons

- Les maps acceptent les doublons
 - Un élément peut être présent plusieurs fois s'il est référencé par des clés différentes
 - Rajout d'un élément avec une clé déjà existante possible
 - La valeur précédente est écrasée

Problématique de comparaison

- Considérons une classe ayant plusieurs attributs.
 - ▶ Plusieurs objets de cette classe sont placés dans une collection
 - ▶ Comment spécifier les critères de tri des éléments ?
 - ❑ Ex : lors d'un parcours de collection, on veut que les objets User arrivent triés par leur nom
 - ❑ S'ils ont le même nom, le deuxième critère est le prénom



Solution 1 : Comparable

354

➤ Implémentation de l'interface java.lang.Comparable

```
public class User implements Comparable<User> {  
    private String nom;  
    private String prenom;  
    public int compareTo(User otherUser) {  
        // renvoie un nombre négatif si l'objet en cours a un nom  
        // situé avant le nom de l'objet passé en paramètre (ordre  
        // alphabétique)  
        // renvoie un nombre positif si l'objet en cours a un nom  
        // situé après le nom de l'objet passé en paramètre  
        int resultat = nom.compareTo(otherUser.nom);  
        if (resultat == 0) {  
            resultat = prenom.compareTo(otherUser.prenom);  
        }  
        return resultat;  
    }  
}
```

Java 5

Solution 2 : Comparator

355

➤ Crédit d'un java.util.Comparator

Java 5

```
public class UserComparator implements Comparator<User> {  
    @Override  
    public int compare(User u1, User u2) {  
        int resultat = u1.getNom().compareTo(u2.getNom());  
        if (resultat == 0) {  
            resultat = u1.getPrenom().compareTo(u2.getPrenom());  
        }  
        return resultat;  
    }  
}
```



Avantage de cette solution : il est possible de créer plusieurs Comparators pour une même classe, et de choisir lequel appliquer selon les cas.

La classe Collections

356

- Classe utilitaire `java.util.Collections` (avec un **s** à la fin)
 - Méthodes utilitaires pour la manipulation des collections

```
List list = new ArrayList();

// tri de la liste (les éléments doivent implémenter
// l'interface Comparable)
Collections.sort(list);

// tri de la liste en fonction d'un Comparator
Collections.sort(list, new UserComparator());

// renvoie une collection non-modifiable
List l1 = Collections.unmodifiableList(list);

// liste synchronisée pour gérer les accès concurrents
List l2 = Collections.synchronizedList(list);
```

La classe Arrays

357

- Classe utilitaire `java.util.Arrays` (avec un **s** à la fin)
 - Méthodes utilitaires pour les tableaux

Java 5

```
User[] usersArray = new User[3];  
  
// tri d'un tableau  
Arrays.sort(usersArray);  
  
// conversion d'un tableau en liste  
List<User> usersList = Arrays.asList(usersArray);
```

Type déclaré

- Toujours utiliser une interface comme type déclaré d'une collection
 - ▶ Dans la mesure du possible, rester générique
 - ▶ Cela permet de changer l'implémentation sans impact sur le reste du code

Java 7

```
HashSet<User> set = new HashSet<>();  
Set<User> set = new HashSet<>();  
  
ArrayList<User> list = new ArrayList<>();  
List<User> list = new ArrayList<>();  
  
HashMap<String, User> map = new HashMap<>();  
Map<String, User> map = new HashMap<>();
```

Parcours de liste (rappel)

➤ Une pratique à éviter...

```
public static void mauvaisParcours() {  
    List list = ...;  
  
    for(int i = 0; i < list.size(); i++) {  
        list.get(i);  
    }  
}
```

➤ Il faut privilégier l'utilisation (dans l'ordre)

► Du foreach

- Si vous n'avez pas besoin de l'indice

- Si vous ne changez pas la liste

► D'un Iterator ou ListIterator

Quiz (1)

➤ Ce code fonctionne-t-il ?

```
List<String> list;  
list.add("lundi");  
list.add("mardi");
```

Quiz (2)

- La méthode pour ajouter un élément dans une map est ?
 - ▶ put
 - ▶ add
 - ▶ insert
 - ▶ []
- Cette question vous sera posée dans les QCM de CodingGame, vous n'avez que 60s pour y répondre
 - ▶ <https://www.codingame.com>



Quiz (3)

362

➤ A la dernière ligne, que contient chacun des ensembles ci-dessous ?

```
User u1 = new User("jean", "dupont");
User u2 = new User("jean", "durand");

Set<User> set = new HashSet<>();
set.add(u1);
set.add(u1);
set.remove(u1);

List<User> list = new ArrayList<>();
list.add(u1);
list.add(u1);

Map<String, User> map = new HashMap<>();
map.put("dupont", u1);
map.put("durand", u2);
map.put("dupont", u1);
```

Java 7



363

Travaux pratiques

Réaliser les travaux pratiques suivants:

TP 3 partie 5

Accès aux bases de données

364

JDBC
Connection
Statement
ResultSet

Objectifs du chapitre

- Comprendre les architectures liées aux bases de données
- Étudier différents modes d'accès
- Écrire des programmes permettant de consulter et de mettre à jour une base de données

Introduction

366

- JDBC pour Java DataBase Connectivity
- API Java adaptée à la connexion avec les bases de données relationnelles et objet-relationnelles.
- L'API fournit un ensemble de classes et d'interfaces permettant l'utilisation sur le réseau d'un ou plusieurs SGBD à partir d'un programme Java.

Objectifs du JDBC

- Apporter un accès homogène au SGBD
- Abstraction des SGBD cibles
- Requêtes SQL
- Simple à mettre en œuvre
- Portabilité

Avantages

368

- Portabilité sur de nombreux système d'exploitation et sur de nombreuses SGBD (Oracle, DB2, Sybase, Microsoft, ..)
- Uniformité du langage de description des applications, des accès aux bases de données
- Liberté totale vis à vis des constructeurs
 - ➡ Le constructeur fournit juste le pilote
 - ➡ Le code est basé sur l'API JDBC

API JDBC

369

- L'API JDBC est disponible dans les package `java.sql` et `javax.sql`
 - ▶ Permet de formuler et gérer les requêtes aux bases de données relationnelles et objet-relationnelle
 - ▶ Supporte les standard SQL-2 et SQL-3
- Elle offre
 - ▶ Une connexion simultanée à plusieurs DB
 - ▶ La gestion des transactions
 - ▶ L'interrogation
 - ▶ L'appel des procédures stockées

Fonctionnement

- JDBC interagit avec le SGBD par un pilote (driver)
- Il existe des drivers pour Oracle, MySQL , Sybase, DB2, Microsoft SQL Server, ...
- Tous les drivers
 - ➡ <https://www.roseindia.net/tutorial/java/jdbc/listofjdbcdriver.html>

Types de drivers

- Il a essentiellement 4 types de drivers JDBC :
 - ▶ Type 1 : Pont JDBC-ODBC
 - ❑ Traduit les appels de l'API JDBC en appels à l'API ODBC
 - ❑ Est fourni par SUN avec le JDK 1.1 (`sun.jdbc.odbc.JdbcOdbcDriver`)
 - ❑ A utiliser uniquement pour des tests et du prototypage
 - ▶ Type 2 : API native
 - ❑ Traduit les appels à l'API JDBC en appels natifs au client local de la base de données
 - ❑ Livré par les éditeurs de SGBD et généralement payant (question de performance)
 - ❑ Nécessite une installation de composants spécifiques

Types de drivers

► Type 3 : Protocole réseau

- Traduit les appels à l'API JDBC en appels à un protocole indépendant du SGBD
- Nécessite un middleware (serveur d'application)
- Le plus utilisé pour les applications d'entreprise

► Type 4 : Protocole natif

- Traduit les appels à l'API JDBC en appels à un protocole natif lié au SGBD
- Souvent fourni par l'éditeur

Mise en œuvre

373

- Importer le package java.sql
- Enregistrer le driver JDBC
- Etablir la connexion à la base de données
- Créer une zone de description de requête
- Exécuter la requête
- Traiter les données rentrées
- Fermer les différents espaces

Enregistrer le driver JDBC

- Méthode `forName()` de la classe `Class` :

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
Class.forName("oracle.jdbc.driver.OracleDriver");
```

- Quand une classe Driver est chargée, elle doit créer une instance d'elle même et s'enregistrer auprès du `DriverManager`
- Certains compilateurs refusent cette notation et demandent plutôt

```
Class.forName("driver_name").newInstance();
```

URL de connexion

- Accès à la base via un URL de la forme :
 - ▶ jdbc:<sous-protocole>:<nomBD>;param=valeur,...
- L'URL spécifie
 - ▶ L'utilisation de JDBC
 - ▶ Le driver ou le type de SGBDR
 - ▶ L'identification de la base locale ou distante
 - avec des paramètres de configuration éventuels
 - nom utilisateur, mot de passe, ...
- Exemple

```
String url = "jdbc:mysql://10.233.97.5:3306/banque";
```

Connexion à la base

- La méthode `getConnexion()` du `DriverManager` accepte de 1 à 3 arguments :
 - ▶ L'URL de la base de données
 - ▶ Le nom de l'utilisateur de la base
 - ▶ Le mot de passe

```
Connection connection = DriverManager.getConnection(url, user, password);
```

- Le `DriverManager` va essayer tous les drivers qui se sont enregistrés jusqu'à ce qu'il trouve un driver qui puisse se connecter à la base demandée

La classe Statement

377

- La classe Statement possède les méthodes nécessaires pour réaliser les requêtes sur la base de données associée à la connexion dont elle dépend
- 3 types de Statement :
 - ▶ **java.sql.Statement** : requêtes statiques simples
 - ▶ **java.sql.PreparedStatement** : requêtes dynamiques précompilées (avec paramètres d'entrées/sorties)
 - ▶ **java.sql.CallableStatement** : procédures stockées
- **Attention à vos import** :
 - ▶ Tout est dans le package `java.sql`

Création d'un Statement

378

- L'objet Connection offre des méthodes pour créer des objets Statement (ou héritiers)

```
Statement statement1 = connection.createStatement();  
  
PreparedStatement statement2 = connection.prepareStatement(query2);  
  
CallableStatement statement3 = connection.prepareCall(query3);
```

Exécution d'une requête (1/2)

- 3 types d'exécution de requête
- `executeQuery()`
 - ▶ Pour les requêtes (SELECT) qui retournent un ResultSet (tuples résultants)
- `executeUpdate()`
 - ▶ Pour les requêtes (INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE, ...) qui retournent un entier (nombre de tuples traités)
- `execute()`
 - ▶ Pour requêtes diverses
 - ▶ Renvoie true si la requête a donné lieu à la création d'un objet ResultSet

Exécution d'une requête (2/2)

380

- `executeQuery()` et `executeUpdate()` acceptent comme argument une chaîne de caractère (`String`) indiquant la requête SQL à exécuter

```
Statement st = connection.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM user");
int nb = st.executeUpdate("INSERT INTO user(email) "
+ "VALUES ('admin@mywebsite.com')");
```

Traiter les données retournées

- Les instances de l'interface ResultSet contiennent les résultats d'une requête SQL. Ils contiennent les tuples (lignes) satisfaisant les conditions de la requête.
- La structure des ResultSet est très semblable à celle d'une table dans une base de données relationnelle

Le résultat : ResultSet

382

- On pourra parcourir le ResultSet ligne par ligne en utilisant la méthode next()
 - ▶ Retourne false si c'est le dernier tuple lu, sinon true
 - ▶ Chaque appel fait avancer le curseur sur le tuple suivant
 - ▶ Initialement, le curseur est positionné avant le premier tuple
 - Exécuter next() au moins une fois pour avoir le premier

```
ResultSet rs = st.executeQuery("SELECT * FROM user");  
while (rs.next()) {  
    // traitement de chaque tuple  
}
```

- Possibilité de revenir au tuple précédent ou d'accéder à un tuple à partir de son index.

Lire les données dans le ResultSet

383

- Les colonnes sont référencées par leur numéros ou par leurs noms
- L'accès aux valeurs des colonnes se fait par les méthodes de la forme getXxx()
 - ➡ Soit en utilisant le numéro de colonne (déconseillé !)
 - ➡ Soit en utilisant le nom de la colonne

```
// accès à la valeur de la 1ere colonne du tuple courant
int val = rs.getInt(1);

// accès à la valeur de la colonne nommée ID du tuple courant
String id = rs.getString("ID") ;
```

Comparaison : une table et un ResultSet

384

ID	Nom	Prénom	email
1	Toto	Titi	toto.titi@mysite.com
2	Duclaux	Cédric	cduclaux@mysite.com

```
Statement st = connection.createStatement();
ResultSet rs = st.executeQuery("SELECT id, nom, email, FROM user");
while(rs.next()) {
    long id = rs.getLong("id");
    String nom = rs.getString("nom");
    String email = rs.getString("email");
}
```

Conversion de types

- Le driver JDBC traduit le type JDBC retourné par le SGBD en un type Java correspondant
 - ▶ Le Xxx de getXxx() est le nom du type Java correspondant au type JDBC attendu
 - ▶ Chaque driver a des correspondances entre les types SQL du SGBD et les types JDBC
 - ▶ Le programmeur est responsable du choix de ces méthodes
 - SQLException générée si mauvais choix

Correspondance des types

386

Type JDBC	Type Java	Méthode
INT	int	getInt ()
REAL	float	getFloat ()
FLOAT	double	getDouble ()
DOUBLE	double	getDouble ()
CHAR	String	getString ()
VARCHAR	String	getString ()
DATE	java.sql.Date	getDate ()
TIME	java.sql.Time	getTime ()

Les valeurs NULL

387

- Pour repérer les valeurs NULL dans une colonne d'une table de la base de données
 - ▶ Utiliser la méthode `wasNull()` de `ResultSet`
 - ❑ Renvoie true si l'on vient de lire un NULL, sinon false
 - ▶ Les méthodes `getXxx()` de `ResultSet` convertissent une valeur NULL SQL en une valeur acceptable par le type d'objet demandé
 - ❑ Les méthodes retournant un objet (`getString()`, `getObject()`,...) retournent null au sens Java
 - ❑ Les méthodes numériques (`getByte()`, `getInt()`,...) retournent 0
 - ❑ La méthode `getBoolean()` retourne false

Libération des ressources

388

- Pour terminer proprement un traitement, il faut fermer les différentes ressources ouvertes
 - ➡ **Quoi qu'il arrive** (`finally`)
- Chaque objet possède une méthode `close()`

```
resultSet.close();  
  
statement.close();  
  
connection.close();
```

Libération des ressources

389

➤ Attention : Toujours fermer ses éléments dans un finally

```
// Declaration hors du try
Connection ctx = null;
Statement statement = null;
ResultSet resultat = null;
try {
    statement = ctx.createStatement();
    resultat = statement.executeQuery("select * from compte where utilisateurId=5");
    // Traitement
} catch (SQLException sql) {
    // Traitement du probleme
} finally {
    // Tres IMPORTANT, on ferme tout
    if (resultat != null) {
        try { resultat.close(); } catch (SQLException e) {
            // Traitement du probleme
        }
    }
    if (statement != null) {
        try { statement.close(); } catch (SQLException e) {
            // Traitement du probleme
        }
    }
    if (ctx != null) {
        try { ctx.close(); } catch (SQLException e) {
            // Traitement du probleme
        }
    }
}
```

Exceptions

390

- SQLException est levée dès qu'une connexion ou un ordre SQL ne se passe pas correctement
 - ▶ La méthode getMessage() donne le message d'erreur en clair
 - ▶ Renvoie aussi des informations spécifiques au gestionnaire de la base comme :
 - ❑ SQLState
 - ❑ Code d'erreur fabricant
- SQLWarning : avertissements SQL

Accès aux méta-données de la DB

391

- Pour récupérer des informations sur la base de données elle-même, on utilise la méthode `getMetaData()` de l'objet `Connection`
 - ➡ Elle renvoie un objet de type `DataBaseMetaData`
 - ➡ On peut connaître entre autres :
 - ❑ Les tables de la base : `getTables()`
 - ❑ Le nom de l'utilisateur : `getUserName()`
 - ❑ ...
 - ➡ La complétude des méta-données dépend du SGBD avec lequel on travaille

Accès aux méta-données du ResultSet

392

- La méthode `getMetaData()` permet d'obtenir des informations sur les types de données du ResultSet
 - ▶ Elle retourne un objet de type `ResultSetMetaData`
 - ▶ On peut connaître entre autres :
 - ❑ Le nombre de colonnes : `getColumnCount()`
 - ❑ Le nom d'une colonne : `getColumnName(int col)`
 - ❑ Le type d'une colonne : `getColumnType(int col)`
 - ❑ Le nom de la table : `getTableName(int col)`
 - ❑ Si un NULL SQL peut être stocké dans une colonne : `isNullable()`



393

TP 4

Réaliser les travaux pratiques suivants:

Travaux pratiques

Les entrées / Sorties

394

IO
NIO

Introduction

395

- Une entrée/sortie en Java consiste à échanger des données entre le programme et une autre source qui peut être:
 - ▶ la mémoire
 - ▶ un fichier
 - ▶ ou le programme lui-même
- Pour réaliser cela, Java emploie ce qu'on appelle un stream (qui signifie flux) entre la source et la destination des données.
- Un flux est une suite de valeurs (octets, caractères, objets quelconques) successivement lues ou écrites

Mécanisme de base

- Toute opération d'entrée/sortie en Java suit le schéma suivant:
 - ▶ Ouverture d'un flux
 - ▶ Lecture ou écriture des données
 - ▶ Fermeture du flux

Emetteurs et récepteurs

397

- Il existe 3 flux standards pour un système d'exploitation
 - Les octets circulant entre une application (A) et l'écran (E) pour indiquer une information standard
 - System.out
 - Les octets circulant entre une application (A) et l'écran (E) pour indiquer une information d'erreur
 - System.err
 - Les octets circulant entre le clavier (C) et une application (A)
 - System.in

Nommage des flux

	Flux d'octets	Flux de caractères
Flux d'entrée	<code>InputStream</code>	<code>Reader</code>
Flux de sortie	<code>OutputStream</code>	<code>Writer</code>

java.io

399

- Les types utilisés pour les entrées / sorties sont définis dans le paquet java.io
- java.io fournit toutes les classes nécessaires à la création, lecture, écriture et traitement des flux
- Le package java.io contient plusieurs sortes de flux qui peuvent être classés suivant plusieurs critères
 - ▶ Les flux d'entrée et les flux de sortie
 - ▶ Les flux de caractères et les flux binaires (dû au fait que java utilise l'unicode les caractères sont codés sur 2 octets et non sur un seul)
 - ▶ Les flux de communication et les flux de traitement
 - ▶ Les flux avec ou sans tampon

Classe InputStream

- Classe abstraite
- C'est la racine des classes qui concernent la lecture d'octets depuis un flot de données
- Type selon lequel sont vues toutes les classes de flux qui lisent des octets
- Elle possède un constructeur sans paramètre

Flux d'entrée

- Ces flux sont des sous-classes de `InputStream` et peuvent être classés en deux catégories :
 - ▶ Les flux de communication
 - ❑ Servant essentiellement à créer une liaison entre le programme et une autre entité
 - ▶ Les flux de traitement
 - ❑ Comme leur nom l'indique, servent plutôt à traiter les données échangées

Les flux entrant de communication

- Ils sont composés des classes suivantes:
 - ➡ **FileInputStream**
 - ❑ Permet de créer un flux avec un fichier présent dans le système de fichiers
 - ➡ **ByteArrayInputStream**
 - ❑ Permet de lire des données binaires à partir d'un tableau d'octets
 - ➡ **PipedInputStream**
 - ❑ Permet de créer une sorte de tube d'entrée (pipe) dans lequel circuleront des octets

Les flux entrant de traitement

- Réaliser un traitement sur les données lues via un flux de communication
- Classes qui étendent la classe `FilterInputStream`
 - ▶ `BufferedInputStream`
 - Cette classe permet la lecture de données à l'aide d'un tampon. Lorsque cette classe est instanciée elle crée un tableau qui va servir du tampon
 - ▶ `DataInputStream`
 - Sert à lire des données représentant des types primitifs de Java (`int`, `boolean`, `double`, `byte`, ...) qui ont été préalablement écrits par un `DataOutputStream`
 - ▶ `SequenceInputStream`
 - Permet de concaténer deux ou plusieurs `InputStream`
 - ▶ `ObjectInputStream`
 - Permet de «déserialiser» un objet, c'est-à-dire de restaurer un objet préalablement sauvegardé à l'aide d'un `ObjectOutputStream`

Exemple

404

```
FileInputStream fis = null;
try {
    // Création d'un flux d'entrée ayant pour source un fichier nommé source, cette instanciation
    // peut lever une exception de type FileNotFoundException
    fis = new FileInputStream("source");

    // La méthode read() renvoie un int représentant le nombre d'octets lus, la valeur (-1)
    // représente la fin du fichier, read peut lever une exception du type IOException
    byte[] bytes = new byte[1024];
    int bytesRead;
    while ((bytesRead = fis.read(bytes)) != -1) {
        System.out.println(Arrays.toString(bytes));
    }
} catch (FileNotFoundException ef) {
    System.err.println("fichier introuvable");
} catch (IOException e) {
    System.err.println(e + "erreur lors de la lecture du fichier");
} finally {
    // Ne pas oublier de fermer le flux afin de libérer les ressources qu'il utilise
    if (fis != null) {
        try { fis.close(); } catch (IOException e) { e.printStackTrace(); }
    }
}
```

Les flux de sortie

- A chaque flux d'entrée présenté précédemment, correspond un flux de sortie
- Les flux de sortie sont aussi classés en flux de communication et flux de traitement

Classe OutputStream

- Interface publique de cette classe (ajouter throws IOException à toutes les méthodes)

```
public abstract void write(int b)
public abstract void write(byte[] b)
public abstract void write(byte[] b, int début, int nb)
public abstract void flush()
public abstract void close()
```

Les flux sortant de communication

407

- PipedOutputStream
 - ➡ Permet la création d'un tube (pipe)
- FileOutputStream
 - ➡ Permet l'écriture séquentielle de données dans un fichier
- ByteArrayOutputStream
 - ➡ Permet d'écrire les données dans un tampon dont la taille s'adapte en fonction du besoin.
 - ❑ Les méthodes suivantes nous permettent de récupérer les données écrites.
 - toByteArray()
 - toString()

Les flux sortant de traitement

408

- **BufferedOutputStream**
 - ▶ Permet d'écrire dans un tampon puis d'écrire le tampon en entier sur le fichier au lieu d'écrire octet par octet sur le fichier
- **DataOutputStream**
 - ▶ Cette classe permet l'écriture de données sous format Java et assure une portabilité inter-applications et inter-systèmes.

```
// création du flux
DataOutputStream out = new DataOutputStream(new FileOutputStream("sortie.dat"));
out.writeInt(12);
out.writeBoolean(true);
out.writeChars("Hello !");
```

- **ObjectOutputStream**
 - ▶ Permet de sauvegarder l'état d'un objet dans un fichier ou autre.
 - L'objet doit implémenter l'interface java.io.Serializable
 - Seuls les membres non-transients et non statiques seront sauvegardés

Exemple - copie d'un fichier

409

```
BufferedInputStream in = null;
BufferedOutputStream out = null;
try {
    // création des flux
    in = new BufferedInputStream(new FileInputStream("source.dat"));
    out = new BufferedOutputStream(new FileOutputStream("copy.dat"));
    // copie du fichier
    byte[] buffer = new byte[1024];
    int bytesRead = -1;
    while ((bytesRead = in.read(buffer)) != -1) {
        out.write(buffer, 0, bytesRead);
    }
    // forcer l'écriture du contenu du tampon dans le fichier
    out.flush();
} catch (IOException e) {
    System.err.println(e);
} finally {
    // fermeture des flux
    if (in != null) {
        try { in.close(); } catch (IOException e) { e.printStackTrace(); }
    }
    if (out != null) {
        try { out.close(); } catch (IOException e) { e.printStackTrace(); }
    }
}
```

Les flux de caractères

- Ils transportent des données sous forme de caractères
 - ➡ Java les gère avec le format Unicode qui code les caractères sur 2 octets
- Les classes qui gèrent les flux de caractères héritent d'une des deux classes abstraites Reader ou Writer
- Nombreuses sous-classes pour traiter les flux de caractères

La classe Reader

411

➤ Classe abstraite qui est la classe mère de toutes les classes qui lisent des flux de caractères

Méthode	Rôle
boolean markSupported()	Indique si le flux supporte la possibilité de marquer des positions.
boolean ready()	Indique si le flux est prêt à être lu.
void close()	Ferme le flux et libère les ressources qui lui étaient associées.
int read()	Renvoie le caractère lu ou -1 si la fin du flux est atteinte.
int read(char[])	Lit plusieurs caractères et les met dans un tableau de caractères, retourne le nombre de caractères lus ou -1.
int read(char[], int, int)	Lit plusieurs caractères et les met dans un tableau de caractères à partir de l'index et pour la longueur fournie en paramètre, retourne le nombre de caractères lus ou -1.
long skip(long)	Saute autant de caractères dans le flux que la valeur fournie en paramètre. Elle renvoie le nombre de caractères effectivement sautés.
void mark()	Permet de marquer une position dans le flux.
void reset()	Retourne dans le flux à la dernière position marquée.

Lecture d'un fichier

412

- Lecture un fichier texte ligne par ligne
- Composition d'un BufferedReader avec un FileReader

```
BufferedReader reader = null;
try {
    reader = new BufferedReader(new FileReader("source.txt"));
    String ligne = null;
    while ((ligne = reader.readLine()) != null) {
        System.out.println(ligne);
    }
} catch (IOException e) {
    System.err.println("erreur de lecture: " + e.getMessage());
} finally {
    if (reader != null) {
        try { reader.close(); } catch (IOException e) { e.printStackTrace(); }
    }
}
```

La classe Writer

➤ Classe abstraite qui est la classe mère de toutes les classes qui écrivent des flux de caractères

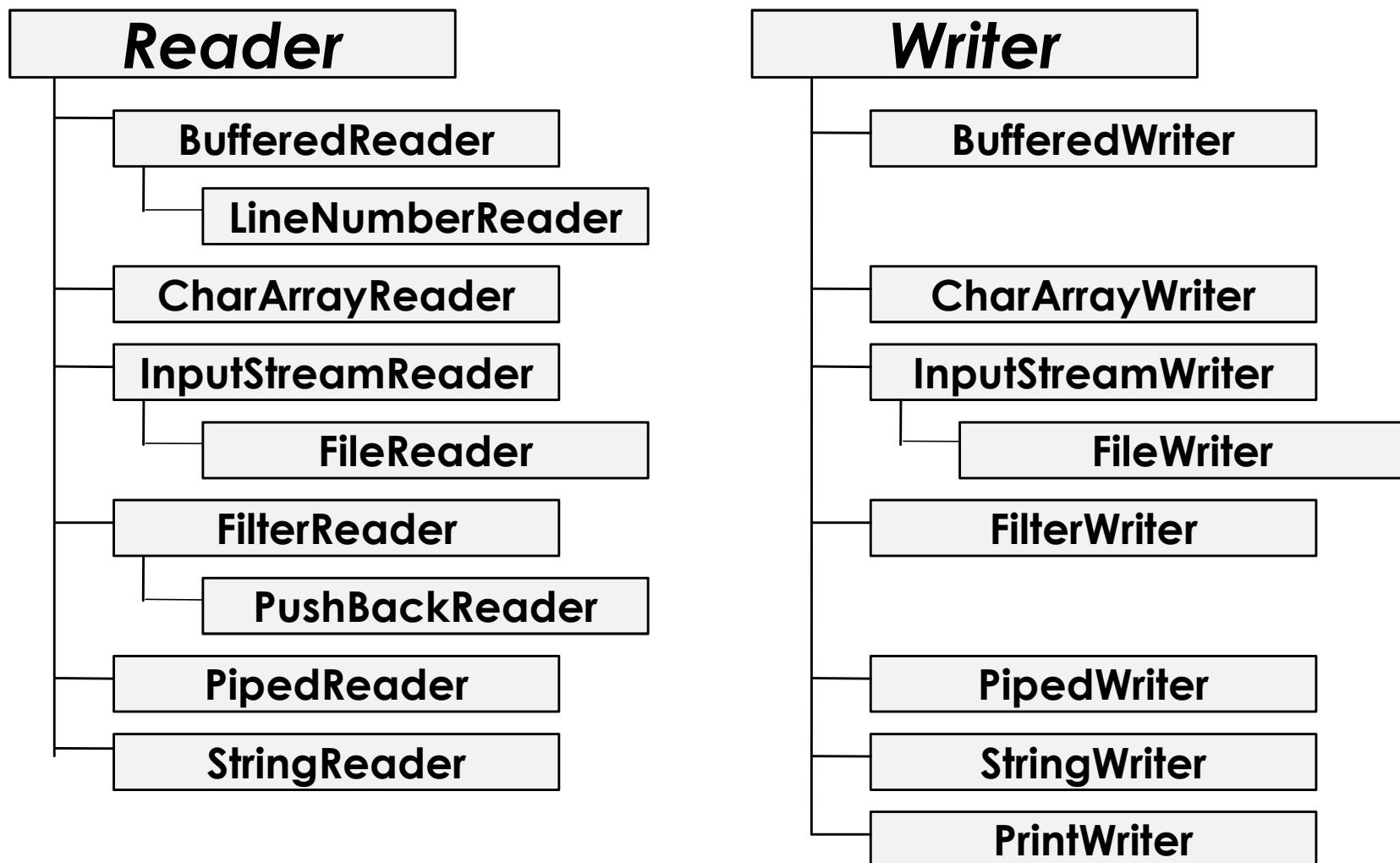
Méthode	Rôle
void close()	Ferme le flux et libère les ressources qui lui étaient associées.
void write(int)	Ecrit le caractère dans le flux.
void write(char[])	Ecrit le tableau de caractères dans le flux.
void write(char[], int, int)	Ecrit le tableau de caractères dans le flux, n'utilise que les caractères entre l'index et pour la longueur fournis en paramètre.
void write(String)	Ecrit la chaîne de caractères dans le flux.
void write(String, int, int)	Ecrit une partie de la chaîne de caractères dans le flux.

Ecrire dans un fichier

- Ecriture dans un fichier texte
- Composition d'un BufferedWriter avec un FileWriter

```
BufferedWriter writer = null;
try {
    int nombre = 123;
    writer = new BufferedWriter(new FileWriter("dest.txt"));
    writer.write("Bonjour tout le monde");
    writer.newLine();
    writer.write("Nous sommes le " + new Date());
    writer.write(", le nombre magique est " + nombre + ".");
} catch (IOException e) {
    System.err.println("erreur d'écriture: " + e.getMessage());
} finally {
    if (writer != null) {
        try { writer.close(); } catch (IOException e) { e.printStackTrace(); }
    }
}
```

La hiérarchie des flux de caractères



La classe File

- java.io.File est une représentation abstraite d'une ressource sur le système de fichier
- Représente soit le chemin d'accès à un dossier ou à un fichier
- La ressource représentée n'existe pas forcément !
- Ses méthodes principales pour la navigation sont :
 - ▶ boolean exists()
 - ▶ boolean isDirectory()
 - ▶ boolean isFile()
 - ▶ String[] list()

```
File fichier = new File("c:/source.dat");

if (fichier.exists()) {
    System.out.println(fichier.getAbsolutePath() +
        " existe");
} else {
    System.out.println(fichier.getAbsolutePath() +
        " n'existe pas");
}
```

- Crédit avec le chemin sous la forme d'une String

Classe Scanner (Java 5)

- Scanner permet de lire les entrées depuis la fenêtre de la console

► La lecture d'une entrée au clavier s'effectue en construisant un Scanner attaché sur le flux d'entrée standard System.in

```
Scanner clavier = new Scanner(System.in);
```

Java 5

- Les diverses méthodes de la classe Scanner permettent ensuite de lire les entrées

► Par exemple, la méthode nextLine() permet de lire une ligne de saisie complète

```
System.out.print("Veuillez entrer votre nom: ");
String nom = clavier.nextLine();
System.out.println("Bonjour " + nom + " !");
```

Java 5

Classe Console (Java 6)

- Java 6 introduit la classe `java.io.Console`
- Permet de saisir un mot de passe sur la console sans qu'il ne s'affiche sur l'écran

```
Console console = System.console();  
if (console != null) {  
    console.printf("Veuillez entrer votre nom: ");  
    String name = console.readLine();  
    console.printf("Veuillez entrer votre mot de passe: ");  
    char[] passwordChars = console.readPassword();  
    String password = new String(passwordChars);  
    console.printf("%s vous êtes connecté"  
                  + " avec le mot de passe %s", name, password);  
} else {  
    System.err.println("Pas de console disponible");  
}
```

Java 6

printf (Java 6)

419

➤ Voir détails sur

<https://docs.oracle.com/javase/tutorial/java/data/numberformat.html>

Converter	Flag	Explanation
d		A decimal integer.
f		A float.
n		A new line character appropriate to the platform running the application. You should always use %n, rather than \n.
tB		A date & time conversion—locale-specific full name of month.
td, te		A date & time conversion—2-digit day of month. td has leading zeroes as needed, te does not.
ty, tY		A date & time conversion—ty = 2-digit year, tY = 4-digit year.
tl		A date & time conversion—hour in 12-hour clock.
tM		A date & time conversion—minutes in 2 digits, with leading zeroes as necessary.
tp		A date & time conversion—locale-specific am/pm (lower case).
tm		A date & time conversion—months in 2 digits, with leading zeroes as necessary.
tD		A date & time conversion—date as %tm%td%ty
	08	Eight characters in width, with leading zeroes as necessary.
	+	Includes sign, whether positive or negative.
	,	Includes locale-specific grouping characters.
	-	Left-justified..
	.3	Three places after decimal point.
	10.3	Ten characters in width, right justified, with three places after decimal point.

printf (Java 6)

420

```
long n = 461012;
System.out.printf("%d%n", n); // --> "461012"
System.out.printf("%08d%n", n); // --> "00461012"
System.out.printf("%+8d%n", n); // --> "+461012"
System.out.printf("%,8d%n", n); // --> " 461,012"
System.out.printf("%+,8d%n%n", n); // --> "+461,012"

double pi = Math.PI;

System.out.printf("%f%n", pi); // --> "3.141593"
System.out.printf("%.3f%n", pi); // --> "3.142"
System.out.printf("%10.3f%n", pi); // --> " 3.142"
System.out.printf("%-10.3f%n", pi); // --> "3.142"
System.out.printf(Locale.FRANCE, "%-10.4f%n%n", pi); // --> "3,1416"

Calendar c = Calendar.getInstance();
System.out.printf("%tB %te, %tY%n", c, c, c); // --> "May 29, 2006"

System.out.printf("%tl:%tm %tp%n", c, c, c); // --> "2:34 am"

System.out.printf("%tD%n", c); // --> "05/29/06"
```

Fichiers de propriétés

- Les fichiers de propriétés permettent de :
 - ▶ Paramétriser une application
 - ▶ Externaliser les ressources pour gérer l'internationalisation
- Une propriété est un couple { clé , valeur }
 - ▶ Clé et valeur sont de type String
 - ▶ Les clés identifient de manière unique les propriétés
- Classe `java.util.Properties`
 - ▶ Hérite de la classe `Hashtable`
 - ▶ Support du format XML
 - ❑ `storeToXML` : génère un fichier XML
 - ❑ `loadFromXML` : lit un fichier de propriétés au format XML

```
<properties>
    <entry key="filename">test.txt</entry>
    <entry key="line">2</entry>
</properties>
```

Communication réseau (1/2)

- Communication par Socket
 - ▶ Protocole TCP/IP
 - ▶ 2 points de connexion : côté client et côté serveur
- Classes du package java.net
 - ▶ InetAddress
 - Représente une adresse IP
 - Encapsule l'accès au serveur de noms (DNS)
 - ▶ Socket
 - Utilisée par les clients TCP
 - Crée un point de connexion
 - Crée une connexion vers le socket serveur
 - ▶ SocketServer
 - Crée un point de communication sur un port particulier
 - Se met en attente de connexions en provenance de clients

Communication réseau (2/2)

423

➤ Exemple d'un serveur de temps

```
// Socket serveur en écoute sur le port 3000
ServerSocket server = new ServerSocket(3000);
while (true) {

    // Accepte la connexion entrante et envoie la date courante
    Socket serviceSocket = server.accept();
    DataOutputStream output = new
    DataOutputStream(serviceSocket.getOutputStream());
    output.writeLong(new Date().getTime());
    serviceSocket.close();
}
```

```
// Récupère la date courante auprès du serveur de temps
Socket clientSocket = new Socket("localhost", 3000);
DataInputStream input = new
DataInputStream(clientSocket.getInputStream());
Date date = new Date(input.readLong());
```

NIO

424

Java 7

- L'API IO a quelques limitations et fait parfois penser à un ensemble de poupée russe
- L'API NIO est arrivée pendant le Java 7 et se complète de version en version
- Ses objectifs
 - ▶ remplacer la classe `java.io.File` (par `java.nio.file.Files`)
 - ▶ étendre les fonctionnalités relatives aux entrées/sorties
 - ▶ Gestions des attributs DOS et POSIX
 - ▶ Gestion des chemins plus générique (via `java.nio.file.Path`)
- Tout est dans le package `java.nio`

NIO

- Les nouvelles fonctionnalités proposées par NIO :
 - ▶ Le support des liens physiques et symboliques
 - ▶ La gestion des attributs sur les fichiers des systèmes Dos et POSIX
 - ▶ L'API WatchService qui surveille un fichier ou un répertoire
 - ▶ La possibilité de parcourir un répertoire
 - ▶ L'utilisation de channels asynchrones pour les opérations de lecture/écriture via un pool de thread
 - ▶ La copie ou le déplacement de fichiers
 - ▶ La possibilité de créer sa propre implémentation d'un système de fichiers.

NIO - Classes principales

426

Java 7

- `java.nio.file.Path` : représente un chemin dans le système de fichiers
- `java.nio.file.Paths` : classe utilitaire pour obtenir des Path
- `java.nio.file.Files` : contient des méthodes statiques pour manipuler les éléments du système de fichiers (Ne pas confondre avec `java.io.File` sans 's')
- `java.nio.file.FileStore` : permet d'obtenir des informations sur le système de stockage comme l'espace total ou l'espace libre.
- `java.nio.file.spi.FileSystemProvider` : service provider qui interagit avec le système de fichiers sous-jacent
- `java.nio.file.FileSystem` : encapsule un système de fichiers et sa fabrique `FileSystems`

NIO - Exemple

427

Java 7

```
1 package com.exemple;
2
3 import java.io.IOException;
4 import java.nio.file.Files;
5 import java.nio.file.Path;
6 import java.nio.file.Paths;
7 import java.util.List;
8
9 public class Run {
10    public static void main(String[] args) {
11        Path chemin = Paths.get("c:/temp/test.txt");
12        List<String> toutesLesLignes = null;
13        try {
14            toutesLesLignes = Files.readAllLines(chemin);
15        } catch (IOException e) {
16            e.printStackTrace();
17        }
18        System.out.println(toutesLesLignes);
19    }
20 }
```



428

TP 5

Réaliser les travaux pratiques suivants:

Travaux pratiques

Les Threads

429

Classe Thread

Interface Runnable

Introduction

- Un thread est une unité d'exécution faisant partie d'un programme.
- Cette unité fonctionne de façon autonome et parallèlement à d'autres threads.
- Le principal avantage des threads est de pouvoir répartir différents traitements d'un même programme en plusieurs unités distinctes pour permettre leur exécution "simultanée".

Interface runnable

- Pour indiquer qu'un code va (ou peut) s'exécuter dans une Thread, il devra implémenter l'interface java.lang.Runnable
- Cette interface n'a qu'une méthode :

```
public void run();
```
- Tout le code contenu dans la méthode sera exécuté dans une Thread.
- Vous ne devrez JAMAIS appeler cette méthode (run) vous-même c'est le Java qui s'en charge.

Classe Thread

432

- C'est cette classe qui gère la notion de traitement en parallèle
- Pour faire usage de cette classe, deux possibilités :
 - ▶ En hériter
 - ▶ En faire usage directement et délégué le traitement à une classe qui implémente l'interface Runnable
- La classe Thread a plusieurs méthodes importantes :
 - ▶ start() : démarre le traitement en parallèle.
 - ▶ destroy() : arrête brutalement le traitement
 - ▶ sleep(long) : pour mettre pause le traitement
 - ▶ run() : à ne pas confondre avec start, contient le code du traitement
- Vous ne devrez **JAMAIS** appeler la méthode run d'une Thread.

Exemple : Implémentation Runnable

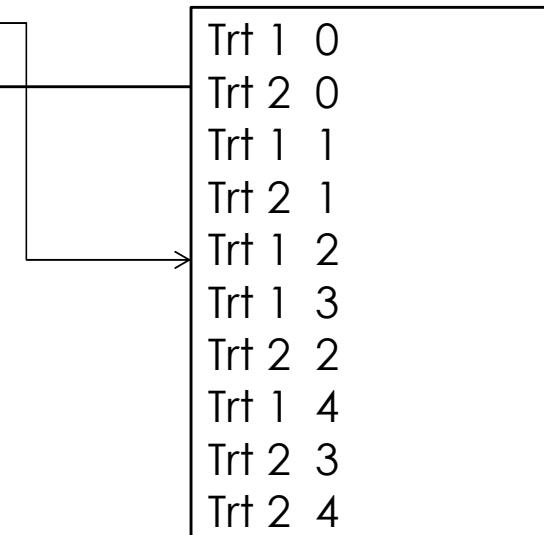
433

```
1 public class MonTraitement implements Runnable {  
2     private String nom;  
3  
4     public MonTraitement(String unNom) {  
5         super();  
6         this.nom = unNom;  
7     }  
8  
9     @Override  
10    public void run() {  
11        for (int lcI = 0; lcI < 5; lcI++) {  
12            System.out.println(this.nom + " " + lcI);  
13        }  
14    }  
15 }
```

Exemple : Crédation d'une Thread

434

```
1 public class Exemple {  
2  
3     public static void main(String[] args) {  
4         Thread t1 = new Thread(new MonTraitement("Trt 1 "));  
5         Thread t2 = new Thread(new MonTraitement("Trt 2 "));  
6         t1.start();  
7         t2.start();  
8     }  
9 }
```



Classe THREAD

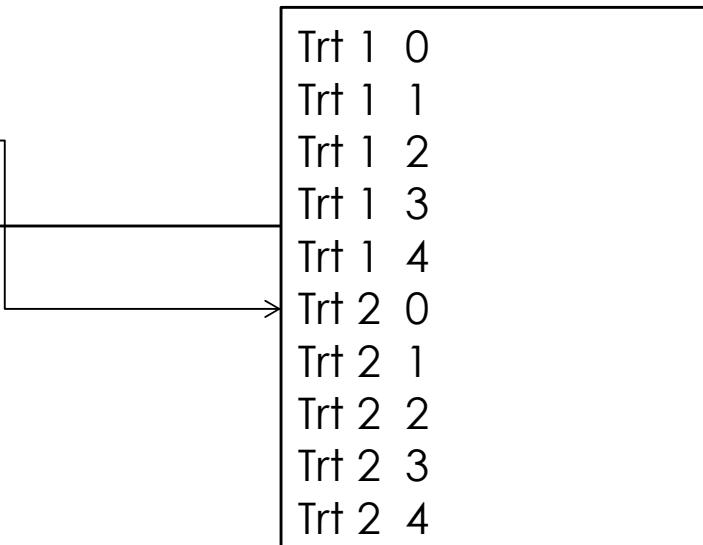
435

- Vous pouvez aussi :
 - ▶ Indiquer une priorité à votre Thread
 - `setPriority(Thread.XXX_PRIORITY)`
 - ▶ Attendre la fin d'une Thread
 - `join()` ou `join(unTempsMaxEnMs)`
 - ▶ Utiliser les méthodes `wait` et `notify` pour faire des verrous
 - ▶ Utiliser un `ThreadGroup` pour gérer la concurences entre plusieurs Thread
- A noter
 - ▶ Les méthodes `wait()`, `notify()`, `notifyAll()` de la classe `Object` servent à gérer les Thread.
 - ▶ Le mot clef `synchronized` peut s'appliquer à une méthode ou à un attribut/variable

Exemple

436

```
1 public class Exemple {  
2  
3     public static void main(String[] args) throws Exception {  
4         Thread t1 = new Thread(new MonTraitement("Trt 1 "));  
5         Thread t2 = new Thread(new MonTraitement("Trt 2 "));  
6         t1.start();  
7         t1.join();  
8         t2.start();  
9     }  
10 }  
11 }
```



```
Trt 1 0  
Trt 1 1  
Trt 1 2  
Trt 1 3  
Trt 1 4  
Trt 2 0  
Trt 2 1  
Trt 2 2  
Trt 2 3  
Trt 2 4
```

Exemple

437

```
1 public class Exemple {  
2  
3     public static void main(String[] args) throws Exception {  
4         ThreadGroup tg = new ThreadGroup("Mon groupe");  
5         Thread t1 = new Thread(tg, new MonTraitement("Trt 1 "));  
6         Thread t2 = new Thread(tg, new MonTraitement("Trt 2 "));  
7         t1.start();  
8         t2.start();  
9     }  
10 }  
11 }
```

The diagram illustrates the execution flow of the Java code. A vertical line starts at the opening brace of the main method (line 11) and branches into two parallel paths. The left path leads to the constructor call 'new MonTraitement("Trt 1 ")' (line 5). The right path leads to the constructor call 'new MonTraitement("Trt 2 ")' (line 6). Both paths converge at a single vertical line that ends at a bracketed box containing the output of the threads.

Trt 1 0
Trt 2 0
Trt 1 1
Trt 2 1
Trt 1 2
Trt 2 2
Trt 1 3
Trt 2 3
Trt 1 4
Trt 2 4

Pour aller plus loin

- Pour la réalisation de codes complexes il existe d'autres classes.
 - ▶ `java.util.concurrent.Executor`
 - ▶ `java.util.concurrent.Executors` : permet de gérer des pools de Thread
 - ▶ `java.util.concurrent.ExecutorService` : gère des Executors et Executors
 - ▶ `java.util.concurrent.Callable<V>` et `java.util.concurrent.Future<V>` : pour réaliser des opérations qui rendent des résultats avec la possibilité de faire des callback.
 - ▶ <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html>

Lambda Expression

439

Java 8

Pourquoi faire

Exemple

Interface fonctionnelle

Historique

- L'ajout des expressions lambda dans le langage Java a été un processus long qui a nécessité de nombreuses années de travaux
- Elle a débuté en 2006 (on est à la sortie du Java 6) et s'est terminée en 2014 (à la sortie du **Java 8**)
- Une lambda expression s'appuie sur les interfaces
- Elle remplace l'écriture sous forme d'inner classe mais sont beaucoup plus efficaces car il n'y a pas de création de classe à la volée.

Rôles & Objectifs

441

Java 8

- Le rôle d'une lambda expression est de simplifier l'écriture du code
 - ▶ C'est donc une expression facilitant/simplifiant l'écriture
- Elle n'est en rien obligatoire : vous n'êtes pas obligé d'en faire usage
 - ▶ Oui mais ..., dans les API modernes elles sont partout
- Elle remplace l'écriture des inner classes dans 95% des cas
- Une lambda expression **n'est pas un type** (comme *int*, *float*, *enum*, *class*, *interface*, ...)
 - ▶ On ne peut donc pas l'affecter à une variable ou la faire passer en paramètre

Rôles & Objectifs

- Les expressions lambda permettent d'écrire du code plus concis, donc plus rapide à écrire, à relire et à maintenir.
- Elles permettent d'écrire du code fonctionnel (on verra le lien avec l'annotation **@FunctionalInterface**).
- Une expression lambda est une fonction dont la définition se fait
 - ▶ sans déclaration explicite du type de retour
 - ▶ sans modificateur d'accès (public, private, ...)
 - ▶ sans nom
 - ▶ sans gestion d'exception explicite (throws)

Rôles & Objectifs

- C'est vraiment un raccourci syntaxique qui simplifie l'écriture de traitements passés en paramètre.
- Elle permet d'encapsuler un traitement pour être passé à d'autres traitements.
- C'est le compilateur qui va se débrouiller pour faire passer les informations vers l'interface fonctionnelle.
 - ➡ Cela lui permet d'obtenir des informations sur les paramètres utilisés, le type de la valeur de retour, les exceptions qui peuvent être levées.
- On les retrouve dans d'autres langages et permettent de réaliser un chainage fonctionnel d'appel de méthodes

Syntaxe

➤ La syntaxe d'une expression lambda est composée de trois parties :

- ▶ un ensemble de paramètres (0 ou n)
- ▶ l'opérateur -> (tiret et supérieur)
- ▶ le corps de la fonction

➤ Exemples :

- ▶ (paramètres) -> paramètres.action()
- ▶ (paramètres) -> { return paramètres.action(); }

Exemple : Comparator

➤ Rappel : voici l'interface Java `java.util.Comparator`

```
public interface Comparator<T> {  
    public int compare(T o1, T o2);  
}
```

- Elle permet de trier / comparer des éléments :
- ▶ Retourne 0 si o1 égal o2
 - ▶ Retourne un chiffre négatif si o1 < o2
 - ▶ Retourne un chiffre positif si o1 > o2

Exemple : Comparator

➤ Ici, on fait une inner classe à la volée afin de trier les voitures en fonction de leur nombre de cheveaux.

```
package com.exemple;
import java.util.*;
public class Run {
    public static void main(String[] args) {
        List<Voiture> listeVoiture = new ArrayList<>();
        listeVoiture.add(new Voiture(5));
        listeVoiture.add(new Voiture(25));
        listeVoiture.add(new Voiture(3));

        System.out.printf("Avant trie : %s%n", listeVoiture);
        listeVoiture.sort(new Comparator<Voiture>() {
            @Override
            public int compare(Voiture v1, Voiture v2) {
                if (v1.getChevaux() == v2.getChevaux()) {
                    return 0;
                }
                return v1.getChevaux() > v2.getChevaux() ? +1 : -1;
            }
        });
        System.out.printf("Apres trie : %s%n", listeVoiture);
    }
}
```

Exemple : Comparator

- Même code,
mais en une
ligne
- Toujours sans
lambda

```
package com.exemple;
import java.util.*;
public class Run {
    public static void main(String[] args) {
        List<Voiture> listeVoiture = new ArrayList<>();
        listeVoiture.add(new Voiture(5));
        listeVoiture.add(new Voiture(25));
        listeVoiture.add(new Voiture(3));

        System.out.printf("Avant tri : %s%n", listeVoiture);
        listeVoiture.sort(new Comparator<Voiture>() {
            @Override
            public int compare(Voiture v1, Voiture v2) {
                return v1.getCheveaux() - v2.getCheveaux();
            }
        });
        System.out.printf("Apres tri : %s%n", listeVoiture);
    }
}
```

Exemple : Comparator

➤ Le même code en lambda expression sur plusieurs lignes

➤ Notez {} et return plus ;

```
package com.exemple;
import java.util.*;
public class Run {
    public static void main(String[] args) {
        List<Voiture> listeVoiture = new ArrayList<>();
        listeVoiture.add(new Voiture(5));
        listeVoiture.add(new Voiture(25));
        listeVoiture.add(new Voiture(3));

        System.out.printf("Avant tri : %s%n", listeVoiture);
        listeVoiture.sort((v1, v2) -> {
            if (v1.getCheveaux() == v2.getCheveaux()) {
                return 0;
            }
            return v1.getCheveaux() > v2.getCheveaux() ? +1 : -1;
        });
        System.out.printf("Apres tri : %s%n", listeVoiture);
    }
}
```

Exemple : Comparator

➤ Le même code en lambda expression sur une ligne

➤ Notez : plus de return, ni {}, ni ;

```
package com.exemple;
import java.util.*;
public class Run {
    public static void main(String[] args) {
        List<Voiture> listeVoiture = new ArrayList<>();
        listeVoiture.add(new Voiture(5));
        listeVoiture.add(new Voiture(25));
        listeVoiture.add(new Voiture(3));

        System.out.printf("Avant tri : %s%n", listeVoiture);
        listeVoiture.sort((v1, v2) ->
            v1.getCheveaux() - v2.getCheveaux());
        System.out.printf("Apres tri : %s%n", listeVoiture);
    }
}
```

Exemple : Runnable

- Rappel : voici l'interface Java `java.lang.Runnable`

```
public interface Runnable {  
    public void run();  
}
```

- Elle permet de créer une Thread, le code de la méthode *run* étant lancé dans ce nouveau processus

Exemple : Runnable

- Ici, on fait une inner classe à la volée

```
package com.exemple;

public class Run {
    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {
                    System.out.printf("Valeur de i=%d%n", i);
                }
            }
        };
        // Fabrication de sa Thread
        Thread t = new Thread(r);
        // Lancement de la Thread
        t.start();
    }
}
```

Exemple : Runnable

➤ Maintenant
en lambda
expression

```
package com.exemple;

public class Run {
    public static void main(String[] args) {
        // Fabrication de sa Thread
        Thread t = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                System.out.printf("Valeur de i=%d%n", i);
            }
        });
        // Lancement de la Thread
        t.start();
    }
}
```

Exemples de syntaxes possibles

- N'accepter aucun paramètre et renvoyer la valeur 123
 - () -> 123 // { et mot clef **return** optionnel dans ce cas et pas de ;
 - () -> { return 123; } // {, mot clef return et ; obligatoire
- Accepter un nombre et renvoyer son double
 - x -> x * 2 // Mot clef return optionnel dans ce cas et pas de ;
- Accepter deux nombres et renvoyer leur somme
 - (x, y) -> x + y // Mot clef return optionnel dans ce cas et pas de ;
 - (int x, int y) -> x + y // Mot clef return optionnel dans ce cas et pas de ;

Exemples de syntaxes possibles

- Accepter une chaîne de caractères et l'afficher sur la sortie standard sans rien renvoyer
 - ➡ (String s) -> System.out.print(s) // Pas de ; dans ce cas
- Renvoyer la taille d'une collection après avoir effacé tous ses éléments.
 - ➡ c -> { int s = c.size(); c.clear(); return s; }

Exemples de syntaxes possibles

- Renvoyer un booléen qui précise si la valeur numérique est impaire
 - `n -> n % 2 != 0 // Mot clef return optionnel dans ce cas et pas de ;`
- Renvoyer un booléen qui précise si le caractère est 'z'
 - `(char c) -> c == 'z' // Mot clef return optionnel dans ce cas et pas de ;`
- Afficher "Hello World" sur la sortie standard
 - `() -> { System.out.println("Hello World"); }`

Exemples de syntaxes possibles

- Renvoyer un booléen qui précise si la première valeur est supérieure ou égale à la seconde
 - ▶ (val1, val2) -> { return val1 >= val2; }
 - ▶ (val1, val2) -> val1 >= val2 // Mot clef return optionnel dans ce cas et pas de ;
- Exécuter la méthode traiter() dix fois
 - ▶ () -> { for (int i = 0; i < 10; i++) traiter(); }
- Renvoyer un booléen pour un objet possédant une méthode getSexe() et getAge() qui vaut true si le sexe est masculin et l'age compris entre 7 et 77 ans
 - ▶ p -> p.getSexe() == Sexe.HOMME && p.getAge() >= 7 && p.getAge() <= 77 // Mot clef return optionnel dans ce cas et pas de ;

Exemples de syntaxes impossibles

- Contrairement à une méthode normale, **il est interdit** de déclarer une variable **qui a le même nom** qu'un paramètre dans une lambda
 - ➡ `(int val1, int val2) -> { int val1 = 0; return val1 >= val2; }`

Lambda : Passes droit

- Une lambda aura les mêmes passe-droits qu'une inner classe :
 - Elle peut discuter en directe avec sa classe englobante (celle où elle se trouve)
 - Accéder aux méthodes et attributs sans tenir compte des visibilités
 - Pour garantir une bonne lisibilité il est conseillé d'utiliser la syntaxe NomDeClasse.this.nomAttribut ou NomDeClasse.this.nomMethode(...)
 - Elle peut accéder aux variables, paramètres de la méthode où elle se trouve
 - avec quelques restrictions, elle ne peut le faire que si ses éléments sont **final**
 - En Java < 8 vous devez indiquer explicitement final
 - En Java >= 8 ses éléments passeront implicitement final, vous n'avez pas à l'indiquer

Lambda et FunctionallInterface

- L'annotation `@FunctionallInterface` n'est pas obligatoire, mais s'applique bien à toute interface qui :
 - ▶ Ne possède qu'une SEULE méthode abstraite
 - ▶ Ne compte pas dans le décompte les méthodes
 - ❑ En *default*
 - ❑ Celles de `java.lang.Object` (`toString`, `equals`, ...)
 - ❑ Les méthodes `static`

- Exemple :

```
package com.exemple;  
  
@FunctionallInterface  
public interface IFaireQQc {  
    public void faire(float unChiffre);  
}
```

Lambda et FunctionallInterface

- Une lambda expression n'est pas un type, mais une interface fonctionnelle est une interface, elle vous permet d'écrire :

```
public static void main(String[] args) {  
    IFaireQQc e = (float f) -> System.out.printf("val f=%f%n", Float.valueOf(f));  
    e.faire(55F);  
    // Affichera : val f=55,000000  
}
```

Méthodes

- Vous pouvez utiliser une expression lambda pour faire appel à
 - ▶ Une méthode
 - Normale nomInstance::nomMethodeNormale
 - Statique NomClasse::nomMethodeStatique
 - ▶ Un constructeur (en faisant attention)
 - nomClasse::new
- On fera usage de l'opérateur ::
- Dans cette forme d'écriture l'expression lambda regardera la compatibilité des formes d'écriture avec ce dont elle a besoin
 - ▶ C'est-à-dire les paramètres de la méthode appelée

Appels de méthodes Syntaxe

462

Java 8

```
package com.exemple;

import javax.swing.JButton;

public class Run {
    public static void main(String[] args) {
        JButton b = new JButton("Click");

        b.addActionListener(System.out::println);
        // Est identique à :
        b.addActionListener((e) -> System.out.println(e));
    }
}
```

Appels de méthodes Syntaxe

463

Java 8

```
package com.exemple;
public class Run {
    public static void main(String[] args) {
        System.out.println("A");
        IFaireQQc f = new FaireQQc();
        System.out.println("B");
        f.faire();
        System.out.println("C");
        IFaireQQc f2 = FaireQQc::new;
        System.out.println("D");
        f2.faire();
        System.out.println("E");
    }
}

class FaireQQc implements IFaireQQc {
    public FaireQQc() {
        System.out.println("Constructeur de la classe 'FaireQQc'");
    }
    @Override
    public void faire() {
        System.out.println("Methode 'faire' dans la classe 'FaireQQc'");
    }
}

@FunctionalInterface
interface IFaireQQc {
    public void faire();
}
```

A

Constructeur de la classe 'FaireQQc'

B

Methode 'faire' dans la classe 'FaireQQc'

C

D

Constructeur de la classe 'FaireQQc'

E

API des FunctionallInterface

464

Java 8

- Dans le Java 8 de base vous avez un très grand nombre d'interface fonctionnelle
- Elles se trouvent dans le package **java.util.function**
- Elles respectent la convention de nommage suivante :
 - ▶ **XxxFunction** : une fonction unaire qui permet de réaliser une transformation. Elle attend un ou plusieurs paramètres et renvoie une valeur. La méthode se nomme **apply()**
 - ▶ **XxxConsumer** : une fonction qui permet de réaliser une action. Elle ne renvoie pas de valeur et attend un ou plusieurs paramètres. La méthode se nomme **accept()**
 - ▶ **XxxPredicate** : une fonction qui attend un ou plusieurs paramètres et renvoie un booléen. La méthode se nomme **test()**
 - ▶ **XxxSupplier** : une fonction qui renvoie une instance. Elle n'attend pas de paramètre et renvoie une valeur. La méthode se nomme **get()**. Elle peut être utilisé comme une fabrique

Exemple : Function

➤ Exemple d'utilisation d'une Function

```
package com.exemple;
import java.util.function.Function;
public class Run {

    public static void main(String[] args) {
        Function<Long, Long> faireX2 = (i) -> {
            // Attention i est un Long - unboxing auto
            return i * 2;
        };

        Function<Long, Long> faireDiv2 = (i) -> {
            // Attention i est un Long - unboxing auto
            return i / 2;
        };

        // Fait : faireDiv2(faireX2(3)) <=> 3*2 puis 6/2 => 3
        System.out.println(faireX2.andThen(faireDiv2).apply(3L));

        // Fait faireX2(faireDiv2(3)) <=> 3/2 puis 1*2 => 2
        System.out.println(faireX2.compose(faireDiv2).apply(3L));
    }
}
```

Exemple : Predicate

➤ Exemple d'utilisation d'un **Predicate**

```
List<String> bunnies = new ArrayList<>();  
bunnies.add("long ear");  
bunnies.add("floppy");  
bunnies.add("hoppy");  
System.out.println(bunnies); // ["long ear", "floppy", "hoppy"]  
bunnies.removeIf(s -> s.charAt(0) != 'h');  
System.out.println(bunnies); // ["hoppy"]
```

Exemple : Predicate

➤ Exemple d'utilisation d'un **Predicate**

```
public class Panda {  
    private int age;  
  
    public static void main(String[] args) {  
        Panda p1 = new Panda();  
        p1.age = 1;  
        Panda.check(p1, p -> p.age < 5);  
    }  
  
    private static void check(Panda panda, Predicate<Panda> pred) {  
        String result = pred.test(panda) ? "match" : "not match";  
        System.out.print(result);  
    }  
}
```

Exécute l'affichage de "match"

l'API Stream

468

Java 8+

Objectifs

Opérations intermédiaires

Opérations terminales

Exemples

Objectifs

469

- l'API Stream permet de mettre en œuvre une approche de la programmation fonctionnelle
- On va écrire notre besoin de traitement comme un enchainement fonctionnel.
 - ➡ On n'est plus sur du sujet.verbe(complement),
 - ❑ mais plus sur du :
 - ➡ traitement1().et().traitement2().sialors().traitement3().sinon().traitement4();

Objectifs

470

- Cette écriture fera usage des lambda expressions
- Rien ne vous oblige à en faire usage
 - ▶ Oui mais ..., dans les API modernes il n'y a que cela (ex : Flux/Mono en Spring)
- Elle permet d'avoir un code plus court et souvent plus lisible fonctionnellement
 - ▶ Du coup il peut être plus complexe à lire/écrire techniquement
- Dans certain cas, le traitement pourra aussi se faire en parallèle de manière totalement transparente
 - ▶ Utilisation de *parallelStream* à la place de *stream*

Exemple

471

➤ Exemple :

```
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");
```

```
myList.stream()  
    .filter(s -> s.startsWith("c"))  
    .map(String::toUpperCase)  
    .sorted()  
    .forEach(System.out::println);
```

Filtre, pour ne prendre que les cases qui commencent par 'c'

→ "c2", "c1"

Transforme en les passant en Majuscules

→ "C2", "C1"

Trie

→ "C1", "C2"

Affiche une par une

C1

C2

Comment faire

- On commencera toujours par l'appel à la méthode `stream()`
 - ➡ Ou `parallelStream()` si l'on veut lancer le traitement en parallèle
- Elle retournera un objet de type `java.util.stream.Stream<T>`
- Sur lequel nous pourrons lancer des opérations

Comment faire

473

➤ Exemple de traitement

```
List<Integer> nombres = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);

List<Integer> troisPremierNombrePairAuCarre = nombres.stream()
    .filter(n -> {
        System.out.println("filter " + n);
        return n % 2 == 0;
    })
    .map(n -> {
        System.out.println("map " + n);
        return n * n;
    })
    .limit(3)
    .collect(Collectors.toList());
System.out.println("");
troisPremierNombrePairAuCarre.forEach(System.out::println);
```

```
filter 1
filter 2
map 2
filter 3
filter 4
map 4
filter 5
filter 6
map 6

4
16
36
```

Comment faire pour obtenir un stream

- Pour obtenir un stream, vous pouvez passer par
 - ➡ Une méthode dans la classe visée (List, String, Files, certain flux ...)
 - Exemple

`Stream<String> instanceBufferedReader.lines()`

→ Les lignes d'un fichier texte ouvert par le buffered reader

`Stream<String> Files.lines(Path)`

→ Les lignes d'un fichier texte ciblé par son chemin

`Stream<T> instanceListT.stream()`

→ Tous les éléments d'une liste

Comment faire pour obtenir un stream

- Ou, pour obtenir un stream, vous pouvez passer par sa/ses fabriques :

```
Stream<Integer> stream = Stream.of(new Integer[ ] { 1, 2, 3 });
```

```
Stream<String> chaines = Stream.of("aaa", "bbb", "ccc");
```

```
Stream<Integer> entiers = Stream.iterate(0, n -> n + 1);  
entiers.limit(10).forEach(System.out::println);
```

Quoi faire avec un Stream

476

- Il existe deux types d'opération
 - ▶ **intermédiaire** : permet d'appliquer une action stateless ou stateful sur un stream (filtrer, transformer par exemple)
 - ▶ **terminale** : qui arrête le traitement du stream et peuvent renvoyer un résultat
- De même, en fonction des opérations, certaines opérations sont exécutées en parallèles ou non selon la méthode sélectionnée

Quoi faire : Les opérations intermédiaires disponibles

477

- Une opération intermédiaire peut être :
 - **stateless** : ne conserve pas d'état lors du traitement d'un élément par rapport au précédent.
 - Chaque élément est donc traité indépendamment des autres.
 - **stateful** : conserve un état par rapport aux éléments traités précédemment lors du traitement d'un élément. Certaines opérations stateful doivent traiter l'ensemble des éléments avant de pouvoir créer leur résultat.
 - Ceci peut avoir des conséquences lors d'un traitements en parallèle : cela peut nécessiter plusieurs itérations sur les données.

Les opérations intermédiaires disponibles

478

Opération		Rôle
filter	Stateless	$\text{Stream} < T > \text{ filter}(\text{Predicate} < ? \text{ super } T > \text{ predicate})$ Filtrer tous les éléments pour n'inclure dans le Stream de sortie que les éléments qui satisfont le Predicat
map	Stateless	$< R > \text{ Stream} < R > \text{ map}(\text{Function} < ? \text{ super } T, ? \text{ extends } R > \text{ mapper})$ Renvoyer un Stream qui contient le résultat de la transformation de chaque élément de type T en un élément de type R
mapToxxx	Stateless	$\text{xxxStream mapToxxx(ToxxxFunction} < ? \text{ super } T > \text{ mapper})$ Renvoyer un Stream qui contient le résultat de la transformation de chaque élément de type T en un type primitif xxx



Les opérations intermédiaires disponibles

479

		$<R> Stream<R> flatMap(Function<T,Stream<? extends R>> mapper)$
flatMap	Stateless	Renvoyer un Stream avec l'ensemble des éléments contenus dans les Stream<R> retournés par l'application de la Function sur les éléments de type T. Ainsi chaque élément de type T peut renvoyer zéro, un ou plusieurs éléments de type R.
		$xxxStream flatMapToxxx(Function<? super T,>? extends xxxStream> mapper)$
flatMapToxxx		Renvoyer un Stream avec l'ensemble des éléments contenus dans les xxxStream retournés par l'application de la Function sur les éléments de type T. Ainsi chaque élément de type T peut renvoyer zéro, un ou plusieurs éléments de type primitif xxx.
		$Stream<T> distinct()$
distinct	Stateful	Renvoyer un Stream<T> dont les doublons ont été retirés. La détection des doublons se fait en invoquant la méthode equals()

Les opérations intermédiaires disponibles

480

sorted	Stateful	<i>Stream<T> sorted()</i> <i>Stream<T> sorted(Comparator<? super T>)</i> Renvoyer un Stream dont les éléments sont triés dans un certain ordre. La surcharge sans paramètre tri dans l'ordre naturel : le type T doit donc implémenter l'interface Comparable car c'est sa méthode compareTo() qui est utilisée pour la comparaison des éléments deux à deux La surcharge avec un Comparator l'utilise pour déterminer l'ordre de tri.
		<i>Stream<T> peek(Consumer <? super T>)</i> Renvoyer les éléments du Stream et leur appliquer le Consumer fourni en paramètre
limit	Stateful	<i>Stream<T> limit(long)</i>
	Short-Circuiting	Renvoyer un Stream qui contient au plus le nombre d'éléments fournis en paramètre

Les opérations intermédiaires disponibles

10.1

		<i>Stream<T> skip(long)</i>
skip	Stateful	Renvoyer un Stream dont les n premiers éléments ont été ignorés, n correspondant à la valeur fournie en paramètre
sequential		<i>Stream<T> sequential()</i> Renvoyer un Stream équivalent dont le mode d'exécution des opérations est séquentiel
parallel		<i>Stream<T> parallel()</i> Renvoyer un Stream équivalent dont le mode d'exécution des opérations est en parallèle
unordered		<i>Stream<T> parallel()</i> Renvoyer un Stream équivalent dont l'ordre des éléments n'a pas d'importance
onClose		<i>Stream<T> onClose(Runnable)</i> Renvoyer un Stream équivalent dont le handler fourni en paramètre sera exécuté à l'invocation de la méthode close(). L'ordre d'exécution de plusieurs handlers est celui de leur déclaration. Tous les handlers sont exécutés même si un handler précédent à lever une exception.

Les opérations terminales disponibles

100

forEach

`void forEach(Consumer<? super T> action)`

Exécuter le Consumer sur chacun des éléments du Stream

`void forEachOrdered(Consumer<? super T> action)`

forEachOrdered

Exécuter le Consumer sur chacun des éléments du Stream en respectant l'ordre de éléments si le Stream en définit un

toArray

`Object[] toArray() // <A> A[] toArray(IntFunction<A[]> generator)`

Renvoyer un tableau contenant les éléments du Stream.

Renvoyer un tableau contenant les éléments du Stream : le tableau est créé par la fonction fournie

`Optional<T> reduce(BinaryOperator<T> accumulator)`

`T reduce(T identity, BinaryOperator<T> accumulator)`

`<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator,`

`BinaryOperator<U> combiner)`

reduce

Réaliser une opération de réduction qui accumule les différents éléments du Stream grâce à la fonction fournie.

Réaliser une opération de réduction qui accumule à partir de la valeur fournie les différents éléments du Stream grâce à la fonction.

Réaliser une opération de réduction avec les fonctions fournies en paramètres.

Les opérations terminales disponibles

collect

$<R,A> R \text{ collect}(\text{Collector}<? \text{ super } T,A,R> \text{ collector})$
 $<R> R \text{ collect}(\text{Supplier}<R> \text{ supplier}, \text{BiConsumer}<R,? \text{ super } T>$
 $\text{accumulator}, \text{BiConsumer}<R,R> \text{ combiner})$

Réaliser une opération de réduction avec le Collector fourni en paramètre
Réaliser une opération de réduction avec les fonctions fournies en paramètres

min

$\text{Optional}<T> \text{ min}(\text{Comparator}<? \text{ super } T> \text{ comparator})$

Renvoyer le plus petit élément du Stream selon le Comparator fourni

max

$\text{Optional}<T> \text{ max}(\text{Comparator}<? \text{ super } T> \text{ comparator})$

Renvoyer le plus grand élément du Stream selon le Comparator fourni

count

$\text{long count}()$

Renvoyer le nombre d'éléments contenu dans le Stream

anyMatch

$\text{boolean anyMatch}(\text{Predicate}<? \text{ super } T> \text{ predicate})$

Retourner un booléen qui indique si au moins un élément valide le Predicate

allMatch

$\text{boolean allMatch}(\text{Predicate}<? \text{ super } T> \text{ predicate})$

Retourner un booléen qui indique si tous les éléments valident le Predicate

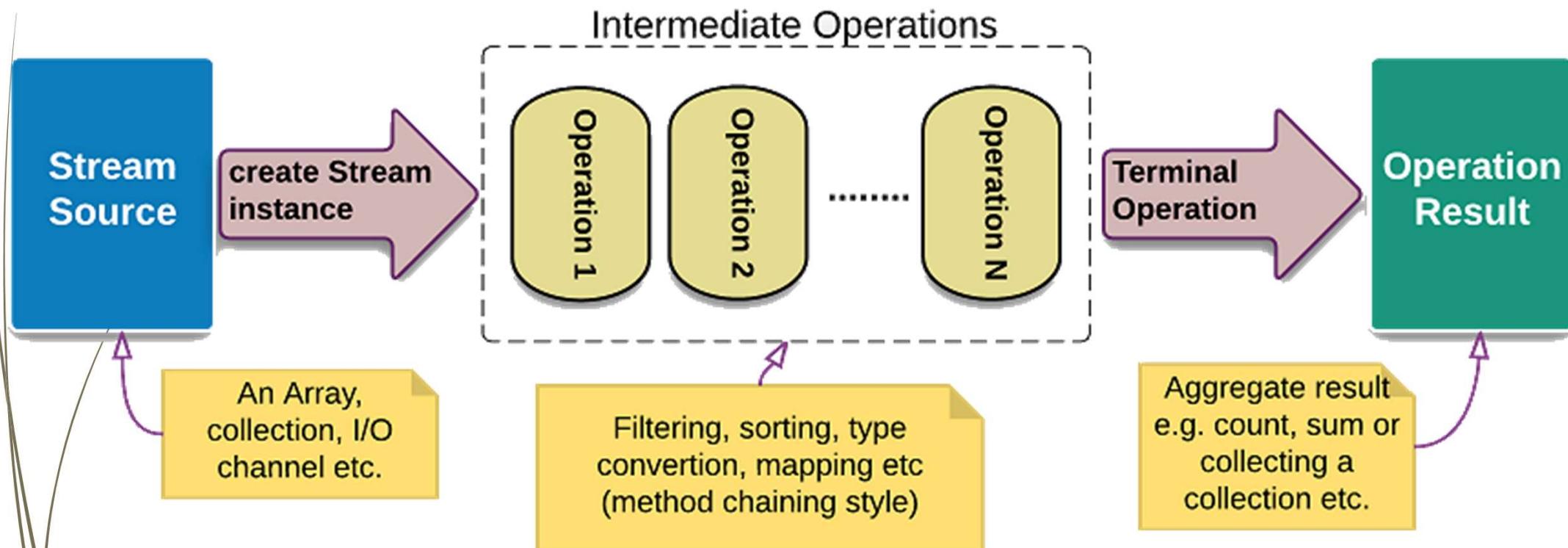
Les opérations terminales disponibles

484

noneMatch	<i>boolean noneMatch(Predicate<? super T> predicate)</i> Retourner un booléen qui indique si aucun élément ne valide le Predicate
findFirst	<i>Optional<T> findFirst()</i> Retourner un Optional qui encapsule le premier élément validant le Predicate s'il existe
findAny	<i>Optional<T> findAny()</i> Retourner un Optional qui encapsule élément validant le Predicate s'il existe
iterator	<i>Iterator<T> iterator()</i> Renvoyer un Iterator qui permet de réaliser une itération sur tous les éléments en dehors du Stream
spliterator	<i>Spliterator<T> spliterator()</i> Renvoyer un Spliterator pour les éléments du Stream

Fonctionnement et enchainement

485



<https://www.logicbig.com/tutorials/core-java-tutorial/java-util-stream/stream-api-intro.html>

Filtering

486

➤ Les filtres se font via la méthode **filter**

```
List<String> prenoms = Arrays.asList("andre", "benoit" "albert", "thierry", "alain", "jean");
prenoms.stream()
    .filter(p -> p.startsWith("a"))
    .forEach(System.out::print);
// andre albert alain
```

```
prenoms.stream()
    .filter(p -> p.startsWith("a"))
    .filter(p -> p.length() == 5)
    .forEach(System.out::print);

// andre alain

// Ou en une fois
// prenoms.stream().filter(p -> p.startsWith("a") && (p.length() == 5)).forEach(System.out::println);
```

Mapping

487

- Il est possible de mapper le résultat via la méthode ***map***
 - ▶ Réaliser une transformation des éléments du stream
 - ▶ **Limitation** : cette méthode ne retourne **qu'un seul** élément
- <R> Stream<R> **map(Function<? super T,? extends R> mapper)**
 - ▶ Renvoyer un Stream qui contient le résultat de l'application de la fonction sur chaque élément du Stream. T est le type des éléments du Stream et R est le type des éléments résultat
- XxxxStream **mapToXxxx(ToXxxxFunction<? super T> mapper)**
 - ▶ Ou Xxxx peut être **Double**, **Int** ou **Long**. Permet de Renvoyer un XxxxStream contenant le résultat de l'application de la fonction passée en paramètre à tous les éléments du Stream
 - **mapToInt**, **mapToDouble**, **mapToLong**

Mapping

488

➤ Exemples

```
Stream<String> chaines = Stream.of("aaa", "bbb", "ccc");
chaines.map(chaine -> chaine.toUpperCase()).forEach(System.out::println);
// chaines.map(String::toUpperCase).forEach(System.out::println);
```

```
Arrays.stream(new int[] {1, 2, 3})
    .map(n -> 2 * n + 1)
    .average()
    .ifPresent(System.out::println); // 5.0
```

```
Stream.of("a1", "a2", "a3")
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .max()
    .ifPresent(System.out::println); // 3
```

FlatMapping

- La fonction ***flatMap*** fait la même chose que ***map*** mais ne retourne pas qu'un seul élément
 - ▶ Elle consomme un stream
 - ▶ Elle renvoie un stream de zéro ou n élément(s)
- `<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)`
- On retrouve, comme pour ***mapToXxx***, les méthodes ***flatMapToInt***, ***flatMapToDouble***, ***flatMapToLong***

FlatMapping

490

➤ Exemple

```
List<Integer> list1 = Arrays.asList(1,2,3);
List<Integer> list2 = Arrays.asList(4,5,6);
List<Integer> list3 = Arrays.asList(7,8,9);

List<List<Integer>> listOfLists = Arrays.asList(list1, list2, list3);

List<Integer> listOfAllIntegers = listOfLists.stream()
    .flatMap(x -> x.stream())
    .collect(Collectors.toList());

System.out.println(listOfAllIntegers); // [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

FlatMapping

491

➤ Exemple

```
String[][] dataArray = new String[][]{{"a", "b"}, {"c", "d"}, {"e", "f"}, {"g", "h"}};

List<String> listOfAllChars = Arrays.stream(dataArray)
    .flatMap(x -> Arrays.stream(x))
    .collect(Collectors.toList());

System.out.println(listOfAllChars); // [a, b, c, d, e, f, g, h]
```

Stream

- Vous pouvez trouver plus d'exemples et d'idées sur l'utilisation des stream ici :
 - ▶ <https://www.jmdoudoux.fr/java/dej/chap-streams.htm>
 - ▶ <https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>



493

TP 10

Réaliser les travaux pratiques suivants:

Travaux pratiques