Java

Table des matières

[Installer Java 4](#_Toc471485887)

[JRE ou JDK ? 4](#_Toc471485888)

[Eclipse 4](#_Toc471485889)

[Premier programme : 5](#_Toc471485890)

[Variables et opérateurs : 5](#_Toc471485891)

[Rappel : 5](#_Toc471485892)

[Les types de variables : 5](#_Toc471485893)

[Quelques calculs : 7](#_Toc471485894)

[Les casts (ou conversion) : 7](#_Toc471485895)

[Depuis Java7, formatage des nombres : 8](#_Toc471485896)

[Lire les entrées claviers : 9](#_Toc471485897)

[Class Scanner : 9](#_Toc471485898)

[Récupérer ce que l’on veut : 9](#_Toc471485899)

[Les conditions : 10](#_Toc471485900)

[If : 10](#_Toc471485901)

[Switch : 11](#_Toc471485902)

[Ternaire : 11](#_Toc471485903)

[Les boucles : 12](#_Toc471485904)

[While : 12](#_Toc471485905)

[Do .. while 13](#_Toc471485906)

[Boucle for : 13](#_Toc471485907)

[Les tableaux : 14](#_Toc471485908)

[Tableaux à une dimension : 14](#_Toc471485909)

[Les tableaux multidimensionnels : 14](#_Toc471485910)

[Rechercher dans un tableau 14](#_Toc471485911)

[Les méthodes de classe : 16](#_Toc471485912)

[Quelques méthodes utiles : 16](#_Toc471485913)

[Créer sa propre méthode : 17](#_Toc471485914)

[La surcharge de méthode : 17](#_Toc471485915)

[Notre première classe : 19](#_Toc471485916)

[Constructeur : 19](#_Toc471485917)

[Les accesseurs et mutateurs : 20](#_Toc471485918)

[This : 21](#_Toc471485919)

[Les variables de classe : 21](#_Toc471485920)

[L’encapsulation : 22](#_Toc471485921)

[L’héritage : 22](#_Toc471485922)

[Le polymorphisme : 24](#_Toc471485923)

[Final : 26](#_Toc471485924)

[La classe object : 26](#_Toc471485925)

[Modéliser ses objets grâce à UML 27](#_Toc471485926)

[Présentation d’UML : 27](#_Toc471485927)

[Modéliser ses objets : 27](#_Toc471485928)

[Modéliser les liens : 28](#_Toc471485929)

[Les packages 29](#_Toc471485930)

[Créer un package : 29](#_Toc471485931)

[Droit d’accès entre les packages : 29](#_Toc471485932)

[Classes abstraites et interface 30](#_Toc471485933)

[Les classes abstraites : 30](#_Toc471485934)

[Les interfaces : 31](#_Toc471485935)

[Le pattern Strategy : 32](#_Toc471485936)

[Les exceptions : 37](#_Toc471485937)

[Try … catch : 37](#_Toc471485938)

[Exceptions personnalisées : 37](#_Toc471485939)

[Gestion de plusieurs exceptions : 39](#_Toc471485940)

[Depuis Java7 : multi catch : 40](#_Toc471485941)

[Les énumérations 41](#_Toc471485942)

[Avant les enum : 41](#_Toc471485943)

[Après les enum : 41](#_Toc471485944)

[Les collections d’objet 43](#_Toc471485945)

[Les différents types de collection : 43](#_Toc471485946)

[Objet List : 43](#_Toc471485947)

[LinkedList : 43](#_Toc471485948)

[L’objet ArrayList : 44](#_Toc471485949)

# Installer Java

## JRE ou JDK ?

JRE contient tout ce qu’il faut pour exécuter des programmes dans une machine virtuelle Java. JDK contient JDE et permet de développer et compiler des programmes.

Les machines virtuelles Java ne comprennent que le **Byte Code**. C’est le rôle de l’IDE Eclipse que de renvoyer notre programme en Byte Code.

Il y a des environnements pour chaque type de programme :

* J2SE(Java 2 Standard Edition, celui qui nous intéresse dans cet ouvrage) : permet de développer des applications dites « client lourd », par exemple Word, Excel, la suite OpenOffice.org… Toutes ces applications sont des « clients lourds » . C'est ce que nous allons faire dans ce cours
* J2EE(Java 2 Enterprise Edition) : permet de développer des applications web en Java. On parle aussi de clients légers.
* J2ME(Java 2 Micro Edition) : permet de développer des applications pour appareils portables, comme des téléphones portables, des PDA…

## Eclipse

L’interface :

* File: C'est ici que nous pourrons créer de nouveaux projets Java, les enregistrer et les exporter le cas échéant.  
  Les raccourcis à retenir sont :
  + **ALT** + **SHIFT** + **N** : nouveau projet ;
  + **CTRL** + **S** : enregistrer le fichier où l'on est positionné ;
  + **CTRL** + **SHIFT** + **S** : tout sauvegarder ;
  + **CTRL** + **W** : fermer le fichier où l'on est positionné ;
  + **CTRL** + **SHIFT** + **W** : fermer tous les fichiers ouverts.
* Edit: Dans ce menu, nous pourrons utiliser les commandes «copier» , «coller», etc.
* Window: Dans celui-ci, nous pourrons configurer Eclipse selon nos besoins.

## Premier programme :

Rappel : println renvoie à la ligne, print non.

1. \rva insérer un retour chariot, parfois utilisé aussi pour les retours à la ligne ;
2. \tva faire une tabulation.

# Variables et opérateurs :

## Rappel :

| **Raccourcis** | **Traduction** | **Correspondance** |
| --- | --- | --- |
| b | Bit | C'est la plus petite valeur informatique : soit 0 soit 1 |
| o | Octet | regroupement de 8 bits, par exemple : 01011101 |
| Ko | Kilo Octet | regroupement de 1024 octets |
| Mo | Mega Octet | regroupement de 1024 ko |
| Go | Giga Octet | regroupement de 1024 Mo |
| To | Tera Octet | regroupement de 1024 Go |

## Les types de variables :

En Java, nous avons deux types de variables :

1. des variables de type simple ou « primitif » ;
2. des variables de type complexe ou des « objets ».
3. Le type byte (1 octet) peut contenir les entiers entre -128 et +127.
4. byte temperature;
5. temperature = 64;
6. Le type short (2 octets) contient les entiers compris entre -32768 et +32767.
7. short vitesseMax;
8. vitesseMax = 32000;
9. Le type int (4 octets) va de -2\*109 à 2\*109 (2 et 9 zéros derrière… ce qui fait déjà un joli nombre).
10. int temperatureSoleil;
11. temperatureSoleil = 15600000; //La température est exprimée en kelvins
12. Le type long (8 octets) peut aller de −9×1018  à 9×1018 (encore plus gros…).
13. long anneeLumiere;
14. anneeLumiere = 9460700000000000L;
15. Afin d'informer la JVM que le type utilisé est **long**, vous **DEVEZ** ajouter un "**L**" à la fin de votre nombre, sinon le compilateur essaiera d'allouer ce dernier dans une taille d'espace mémoire de type entier et votre code ne compilera pas si votre nombre est trop grand...
16. Le type float (4 octets) est utilisé pour les nombres avec une virgule flottante.
17. float pi;
18. pi = 3.141592653f;
19. Ou encore :
20. float nombre;
21. nombre = 2.0f;
22. Vous remarquerez que nous ne mettons pas une virgule, mais un point ! Et vous remarquerez aussi que même si le nombre en question est rond, on écrit « .0 » derrière celui-ci, le tout suivi de « f ».
23. Le type double (8 octets) est identique à float, si ce n'est qu'il contient plus de chiffres derrière la virgule et qu'il n'a pas de suffixe.
24. double division;
25. division = 0.333333333333333333333333333333333333333333334d;
26. Ici encore, vous devez utiliser une lettre - le « d » - pour parfaire la déclaration de votre variable.

##### Des variables stockant un caractère

1. Le type char contient un caractère stocké entre apostrophes (« ' ' »), comme ceci :
2. char caractere;
3. caractere = 'A';

##### Des variables de type booléen

1. Le type boolean, lui, ne peut contenir que deux valeurs  : true (vrai) ou false (faux), sans guillemets (ces valeurs sont natives dans le langage, il les comprend directement et sait les interpréter).
2. boolean question;
3. question = true;

String est un objet, il comence par une majuscule.

int entier = 32;

float pi = 3.1416f;

char carac = 'z';

String mot = new String("Coucou");

Convention :

* **tous vos noms de classes doivent commencer par une majuscule ;**
* **tous vos noms de variables doivent commencer par une minuscule ;**
* **si le nom d'une variable est composé de plusieurs mots, le premier commence par une minuscule, le ou les autres par une majuscule, et ce, sans séparation ;**
* **tout ceci sans accentuation !**

## Quelques calculs :

int nbre1, nbre2, nbre3; //Déclaration des variables

nbre1 = 1 + 3; //nbre1 vaut 4

nbre2 = 2 \* 6; //nbre2 vaut 12

nbre3 = nbre2 / nbre1; //nbre3 vaut 3

nbre1 = 5 % 2; //nbre1 vaut 1, car 5 = 2 \* 2 + 1

nbre2 = 99 % 8; //nbre2 vaut 3, car 99 = 8 \* 12 + 3

nbre3 = 6 % 3; //là, nbre3 vaut 0, car il n'y a pas de reste

Afficher sa variable :

double nbre1 = 10, nbre2 = 3;

int resultat = (int)(nbre1 / nbre2);

System.out.println("Le résultat est = " + resultat);

## Les casts (ou conversion) :

int i = 123;

float j = (float)i;

D'un type int en double :

int i = 123;

double j = (double)i;

Et inversement :

double i = 1.23;

double j = 2.9999999;

int k = (int)i; //k vaut 1

k = (int)j;

Exemple, on a deux int et on a un résultat double. On aura quand même un resultat tronqué car la JVM fait d’abords le calcule et ensuite fait le cast. On devra donc plutôt faire :

int nbre1 = 3, nbre2 = 2;

double resultat = (double)(nbre1) / (double)(nbre2);

System.out.println("Le résultat est = " + resultat);

//affiche : Le résultat est = 1.5

Cast avec un String :

int i = 12;

String j = new String();

j = j.valueOf(i);

int k = Integer.valueOf(j).intValue();

## Depuis Java7, formatage des nombres :

double nombre = 1000000000000d; // cast en d

//Peut s'écrire ainsi

double nombre = 1\_\_\_\_000\_\_\_\_000\_\_\_000\_000d; // cast en d

//Le nombre d'underscore n'a pas d'importance

//Voici quelques autres exemple d'utilisation

int entier = 32\_000;

double monDouble = 12\_34\_56\_78\_89\_10d; // cast en d

double monDouble2 = 1234\_5678\_8910d; // cast en d

Les underscore doivent être placés entre deux caractères numériques :

double d = 123\_.159;

int entier = \_123;

int entier2 = 123\_;

Avant Java 7, il était possible de déclarer des expressions numériques en hexadécimal, en utilisant le préfixe « 0x » :

int entier = 255; //Peut s'écrire « int entier = 0xFF; »

int entier = 20; //Peut s'écrire « int entier = 0x14; »

int entier = 5112; //Peut s'écrire « int entier = 0x13\_F8; »

Depuis java 7, vous avez aussi la possibilité d'utiliser la notation binaire, en utilisant le préfixe « 0b » :

int entier = 0b1111\_1111; //Est équivalent à : « int entier = 255; »

int entier = 0b1000\_0000\_0000; //Est équivalent à : « int entier = 2048; »

int entier = 0b100000000000; //Est équivalent à : « int entier = 2048; »

# Lire les entrées claviers :

## Class Scanner :

Scanner sc = new Scanner(System.in);

On a besoin d’importer une classe pour le faire fonctionner :

//Ceci importe la classe Scanner du package java.util

import java.util.Scanner;

//Ceci importe toutes les classes du package java.util

import java.util.\*;

## Récupérer ce que l’on veut :

Scanner sc = new Scanner(System.in);

System.out.println("Veuillez saisir un nombre :");

int str = sc.nextInt();

System.out.println("Vous avez saisi le nombre : " + str);

Ou avec un String :

Scanner sc = new Scanner(System.in);

System.out.println("Veuillez saisir un mot :");

String str = sc.nextLine();

System.out.println("Vous avez saisi : " + str);

(Attention, Char n’est pas pris en compte par Scanner 🡺 nextChar() ).

Pour fait l’équivalent, on utilise charAt().

System.out.println("Saisissez une lettre :");

Scanner sc = new Scanner(System.in);

String str = sc.nextLine();

char carac = str.charAt(0);

System.out.println("Vous avez saisi le caractère : " + carac);

Si on doit utiliser plusieurs nextXxx(), il faut penser à vider la ligne avant d’en écrire une autre. Normalement la tête de lecture est placée sur la ligne suivante, mais pas quand on demande plusieurs variables différentes !

import java.util.Scanner;

public class Main {

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.println("Saisissez un entier : ");

int i = sc.nextInt();

System.out.println("Saisissez une chaîne : ");

String str = sc.nextLine();

System.out.println("FIN ! ");

}

}

On aura dirrectement « fin » ici. Pour pallier ce problème, il suffit de vider la ligne après les instructions ne le faisant pas automatiquement :

import java.util.Scanner;

public class Main {

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.println("Saisissez un entier : ");

int i = sc.nextInt();

System.out.println("Saisissez une chaîne : ");

//On vide la ligne avant d'en lire une autre

sc.nextLine();

String str = sc.nextLine();

System.out.println("FIN ! ");

}

}

# Les conditions :

## If :

int i = 10;

if (i < 0)

System.out.println("le nombre est négatif");

else

System.out.println("le nombre est positif");

Les conditions multiples :

int i = 58;

if(i < 100 && i > 100)

System.out.println("Le nombre est bien dans l'intervalle.");

else

System.out.println("Le nombre n'est pas dans l'intervalle.");

## Switch :

int note = 10; //On imagine que la note maximale est 20

switch (note)

{

case 0:

System.out.println("Ouch !");

break;

case 10:

System.out.println("Vous avez juste la moyenne.");

break;

case 20:

System.out.println("Parfait !");

break;

default:

System.out.println("Il faut davantage travailler.");

}

## Ternaire :

int x = 10, y = 20;

int max = (x < y) ? y : x ; //Maintenant, max vaut 20

* Nous cherchons à affecter une valeur à notre variable max, mais de l'autre côté de l'opérateur d'affectation se trouve une condition ternaire…
* Ce qui se trouve entre les parenthèses est évalué : x est-il plus petit que y ? Donc, deux cas de figure se profilent à l'horizon :
  + si la condition renvoie true (vrai), qu'elle est vérifiée, la valeur qui se trouve après le ?sera affectée ;
  + sinon, la valeur se trouvant après le symbole**:** sera affectée.
* L'affectation est effective : vous pouvez utiliser votre variable max.

# Les boucles :

## While :

//Une variable vide

String prenom;

//On initialise celle-ci à O pour oui

char reponse = 'O';

//Notre objet Scanner, n'oubliez pas l'import de java.util.Scanner !

Scanner sc = new Scanner(System.in);

//Tant que la réponse donnée est égale à oui…

while (reponse == 'O')

{

//On affiche une instruction

System.out.println("Donnez un prénom : ");

//On récupère le prénom saisi

prenom = sc.nextLine();

//On affiche notre phrase avec le prénom

System.out.println("Bonjour " +prenom+ ", comment vas-tu ?");

//On demande si la personne veut faire un autre essai

System.out.println("Voulez-vous réessayer ? (O/N)");

//On récupère la réponse de l'utilisateur

reponse = sc.nextLine().charAt(0);

}

System.out.println("Au revoir…");

//Fin de la boucle

On peut le forcer à mettre O ou N :

String prenom;

char reponse = 'O';

Scanner sc = new Scanner(System.in);

while (reponse == 'O')

{

System.out.println("Donnez un prénom : ");

prenom = sc.nextLine();

System.out.println("Bonjour " +prenom+ ", comment vas-tu ?");

//Sans ça, nous n'entrerions pas dans la deuxième boucle

reponse = ' ';

//Tant que la réponse n'est pas O ou N, on repose la question

while(reponse != 'O' && reponse != 'N')

{

//On demande si la personne veut faire un autre essai

System.out.println("Voulez-vous réessayer ? (O/N)");

reponse = sc.nextLine().charAt(0);

}

}

System.out.println("Au revoir…");

Une autre synthaxe :

int a = 1, b = 15;

while (a++ < b)

System.out.println("coucou " +a+ " fois !!");

## Do .. while

String prenom = new String();

//Pas besoin d'initialiser : on entre au moins une fois dans la boucle !

char reponse = ' ';

Scanner sc = new Scanner(System.in);

do{

System.out.println("Donnez un prénom : ");

prenom = sc.nextLine();

System.out.println("Bonjour " +prenom+ ", comment vas-tu ?");

do{

System.out.println("Voulez-vous réessayer ? (O/N)");

reponse = sc.nextLine().charAt(0);

}while(reponse != 'O' && reponse != 'N');

}while (reponse == 'O');

System.out.println("Au revoir…");

Elle s’éxecute au moins une fois. Il y a un point virgule après le while !

## Boucle for :

for(int i = 1; i <= 10; i++)

{

System.out.println("Voici la ligne "+i);

}

# Les tableaux :

## Tableaux à une dimension :

int tableauEntier[] = new int[6];

//Ou encore

int[] tableauEntier2 = new int[6];

Ou :

int tableauEntier[] = {0,1,2,3,4,5,6,7,8,9};

double tableauDouble[] = {0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0};

char tableauCaractere[] = {'a','b','c','d','e','f','g'};

String tableauChaine[] = {"chaine1", "chaine2", "chaine3" , "chaine4"};

## Les tableaux multidimensionnels :

int premiersNombres[][] = { {0,2,4,6,8},{1,3,5,7,9} };

## Rechercher dans un tableau

Recherche avec une boucle :

char tableauCaractere[] = {'a','b','c','d','e','f','g'};

for(int i = 0; i < tableauCaractere.length; i++)

{

System.out.println("À l'emplacement " + i +" du tableau nous avons = " + tableauCaractere[i]);

}

Exemple :

char tableauCaractere[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g'};

int i = 0;

char reponse = ' ',carac = ' ';

Scanner sc = new Scanner(System.in);

do {//Boucle principale

do {//On répète cette boucle tant que l'utilisateur n'a pas rentré une lettre figurant dans le tableau

i = 0;

System.out.println("Rentrez une lettre en minuscule, SVP ");

carac = sc.nextLine().charAt(0);

//Boucle de recherche dans le tableau

while(i < tableauCaractere.length && carac != tableauCaractere[i])

i++;

//Si i < 7 c'est que la boucle n'a pas dépassé le nombre de cases du tableau

if (i < tableauCaractere.length)

System.out.println(" La lettre " +carac+ " se trouve bien dans le tableau !");

else //Sinon

System.out.println(" La lettre " +carac+ " ne se trouve pas dans le tableau !");

}while(i >= tableauCaractere.length);

//Tant que la lettre de l'utilisateur ne correspond pas à une lettre du tableau

do{

System.out.println("Voulez-vous essayer à nouveau ? (O/N)");

reponse = sc.nextLine().charAt(0);

}while(reponse != 'N' && reponse != 'O');

}while (reponse == 'O');

System.out.println("Au revoir !");

Tableau avec les String :

String tab[][]={{"toto", "titi", "tutu", "tete", "tata"}, {"1", "2", "3", "4"}};

int i = 0, j = 0;

for(String sousTab[] : tab)

{

i = 0;

for(String str : sousTab)

{

System.out.println("La valeur de la nouvelle boucle est : " + str);

System.out.println("La valeur du tableau à l'indice ["+j+"]["+i+"] est : " + tab[j][i]);

i++;

}

j++;

}

# Les méthodes de classe :

## Quelques méthodes utiles :

toLowerCase() :

String chaine = new String("COUCOU TOUT LE MONDE !"), chaine2 = new String();

chaine2 = chaine.toLowerCase(); //Donne "coucou tout le monde !"

toUpperCase() :

String chaine = new String("coucou coucou"), chaine2 = new String();

chaine2 = chaine.toUpperCase(); //Donne "COUCOU COUCOU"

length() :

String chaine = new String("coucou ! ");

int longueur = 0;

longueur = chaine.length(); //Renvoie 9

equals():

String str1 = new String("coucou"), str2 = new String("toutou");

if (str1.equals(str2))

System.out.println("Les deux chaînes sont identiques !");

else

System.out.println("Les deux chaînes sont différentes !");

La méthode substring() extrait une partie d'une chaîne de caractères. Elle prend deux entiers en arguments : le premier définit le premier caractère (inclus) de la sous-chaîne à extraire, le second correspond au dernier caractère (exclu) à extraire. Là encore, le premier caractère porte le numéro 0.

String chaine = new String("la paix niche"), chaine2 = new String();

chaine2 = chaine.substring(3,13); //Permet d'extraire "paix niche"

La méthode indexOf() explore une chaîne de caractères à la recherche d'une suite donnée de caractères, et renvoie la position (ou l'index) de la sous-chaîne passée en argument. la méthode indexOf() explore à partir du début de la chaîne, lastIndexOf() explore en partant de la fin, mais renvoie l'index à partir du début de la chaîne. Ces deux méthodes prennent un caractère ou une chaîne de caractères comme argument, et renvoient un int. Tout comme charAt() et substring(), le premier caractère porte le numéro 0. Je crois qu'ici, un exemple s'impose, plus encore que pour les autres fonctions :

String mot = new String("anticonstitutionnellement");

int n = 0;

n = mot.indexOf('t'); //n vaut 2

n = mot.lastIndexOf('t'); //n vaut 24

n = mot.indexOf("ti"); //n vaut 2

n = mot.lastIndexOf("ti"); //n vaut 12

n = mot.indexOf('x'); //n vaut -1

Les fonctions mathématiques :

double X = 0.0;

X = Math.random();

//Retourne un nombre aléatoire

//compris entre 0 et 1, comme 0.0001385746329371058

double sin = Math.sin(120); //La fonction sinus

double cos = Math.cos(120); //La fonction cosinus

double tan = Math.tan(120); //La fonction tangente

double abs = Math.abs(-120.25); //La fonction valeur absolue (retourne le nombre sans le signe)

double d = 2;

double exp = Math.pow(d, 2); //La fonction exposant

//Ici, on initialise la variable exp avec la valeur de d élevée au carré

//La méthode pow() prend donc une valeur en premier paramètre, et un exposant en second

## Créer sa propre méthode :

Les méthodes dans la classe main doivent être static :

public class Sdz1

{

public static void main(String[] args)

{

String[] tab = {"toto", "tata", "titi", "tete"};

parcourirTableau(tab);

}

static void parcourirTableau(String[] tabBis)

{

for(String str : tabBis)

System.out.println(str);

}

}

## La surcharge de méthode :

Le principe conssite à ne pas changer de nom de méthode mais à changer ses paramètres :

static void parcourirTableau(String[] tab)

{

for(String str : tab)

System.out.println(str);

}

static void parcourirTableau(int[] tab)

{

for(int str : tab)

System.out.println(str);

}

Vous pouvez faire de même avec les tableaux à deux dimensions. Voici à quoi pourrait ressembler le code d'une telle méthode (je ne rappelle pas le code des deux méthodes ci-dessus) :

static void parcourirTableau(String[][] tab)

{

for(String tab2[] : tab)

{

for(String str : tab2)

System.out.println(str);

}

}

On pourra donc faire, avec ces trois méthodes :

String[] tabStr = {"toto", "titi", "tata"};

int[] tabInt = {1, 2, 3, 4};

String[][] tabStr2 = {{"1", "2", "3", "4"}, {"toto", "titi", "tata"}};

//La méthode avec un tableau de String sera invoquée

parcourirTableau(tabStr);

//La méthode avec un tableau d'int sera invoquée

parcourirTableau(tabInt);

//La méthode avec un tableau de String à deux dimensions sera invoquée

parcourirTableau(tabStr2);

# Notre première classe :

## Constructeur :

public class Ville{

String nomVille;

String nomPays;

int nbreHabitants;

}

Avec le constructeur :

public class Ville{

//Stocke le nom de notre ville

String nomVille;

//Stocke le nom du pays de notre ville

String nomPays;

//Stocke le nombre d'habitants de notre ville

int nbreHabitants;

//Constructeur par défaut

public Ville(){

System.out.println("Création d'une ville !");

nomVille = "Inconnu";

nomPays = "Inconnu";

nbreHabitants = 0;

}

}

Création d’un objet :

public class Sdz1{

public static void main(String[] args){

Ville ville = new Ville();

}

}

Ou avec plusieurs types de constructeurs :

public class Ville {

//Stocke le nom de notre ville

String nomVille;

//Stocke le nom du pays de notre ville

String nomPays;

//Stocke le nombre d'habitants de notre ville

int nbreHabitants;

//Constructeur par défaut

public Ville(){

System.out.println("Création d'une ville !");

nomVille = "Inconnu";

nomPays = "Inconnu";

nbreHabitants = 0;

}

//Constructeur avec paramètres

//J'ai ajouté un « p » en première lettre des paramètres.

//Ce n'est pas une convention, mais ça peut être un bon moyen de les repérer.

public Ville(String pNom, int pNbre, String pPays)

{

System.out.println("Création d'une ville avec des paramètres !");

nomVille = pNom;

nomPays = pPays;

nbreHabitants = pNbre;

}

}

Il faut contrôler les variables de classe, on les met donc en private :

public class Ville {

private String nomVille;

private String nomPays;

private int nbreHabitants;

//…

}

## Les accesseurs et mutateurs :

Ils permettent de lire et modifier les variables de classe :

public class Ville {

//Les variables et les constructeurs n'ont pas changé…

//\*\*\*\*\*\*\*\*\*\*\*\*\* ACCESSEURS \*\*\*\*\*\*\*\*\*\*\*\*\*

//Retourne le nom de la ville

public String getNom() {

return nomVille;

}

//Retourne le nom du pays

public String getNomPays()

{

return nomPays;

}

// Retourne le nombre d'habitants

public int getNombreHabitants()

{

return nbreHabitants;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\* MUTATEURS \*\*\*\*\*\*\*\*\*\*\*\*\*

//Définit le nom de la ville

public void setNom(String pNom)

{

nomVille = pNom;

}

//Définit le nom du pays

public void setNomPays(String pPays)

{

nomPays = pPays;

}

//Définit le nombre d'habitants

public void setNombreHabitants(int nbre)

{

nbreHabitants = nbre;

}

}

## This :

Pour simplifier, this fait référence à l'objet courant ! Bien que la traduction anglaise exacte soit « ceci », il faut comprendre « moi ». À l'intérieur d'un objet, ce mot clé permet de désigner une de ses variables ou une de ses méthodes.

## Les variables de classe :

Les variables de classes sont disponible avec la totalité des classes.

Afin qu'une variable soit une variable de classe, elle doit être précédée du mot clé static. Cela donnerait dans notre classe Ville :

public class Ville {

//Variables publiques qui comptent les instances

public static int nbreInstances = 0;

//Variable privée qui comptera aussi les instances

private static int nbreInstancesBis = 0;

//Les autres variables n'ont pas changé

public Ville(){

//On incrémente nos variables à chaque appel aux constructeurs

nbreInstances++;

nbreInstancesBis++;

//Le reste ne change pas.

}

public Ville(String pNom, int pNbre, String pPays)

{

//On incrémente nos variables à chaque appel aux constructeurs

nbreInstances++;

nbreInstancesBis++;

//Le reste ne change pas

}

public static int getNombreInstancesBis()

{

return nbreInstancesBis;

}

//Le reste du code est le même qu'avant

}

## L’encapsulation :

Voilà, vous venez de construire votre premier objet « maison ». Cependant, sans le savoir, vous avez fait plus que ça : vous avez créé un objet dont les variables sont protégées de l'extérieur. En effet, depuis l'extérieur de la classe, elles ne sont accessibles que via les accesseurs et mutateurs que nous avons défini. C'est le principe d'encapsulation !

# L’héritage :

class Capitale extends Ville {

}

On ne peut pas utiliser les variables de la classe mère si ceux-ci sont protégés, exemple :

public class Capitale extends Ville{

public Capitale(){

this.nomVille = "toto";

}

}

Ici, erreur s’ils sont en private !

Solution : passer de *private* à *protected* :

public class Ville {

public static int nbreInstances = 0;

protected static int nbreInstancesBis = 0;

protected String nomVille;

protected String nomPays;

protected int nbreHabitants;

protected char categorie;

//Tout le reste est identique.

}

On peut accéder aux paramètres de la classe mère, on utilise *super*. Par exemple dans le constructeur :

class Capitale extends Ville {

private String monument;

//Constructeur par défaut

public Capitale(){

//Ce mot clé appelle le constructeur de la classe mère

super();

monument = "aucun";

}

}

On peut aussi l’utiliser ainsi : cf polymorphisme :

class Capitale extends Ville {

private String monument;

public Capitale(){

//Ce mot clé appelle le constructeur de la classe mère

super();

monument = "aucun";

}

public String decrisToi(){

String str = super.decrisToi() + "\n \t ==>>" + this.monument+ " en est un monument";

System.out.println("Invocation de super.decrisToi()");

return str;

}

}

Au complet, on obtient comme classe fille :

public class Capitale extends Ville {

private String monument;

//Constructeur par défaut

public Capitale(){

//Ce mot clé appelle le constructeur de la classe mère

super();

monument = "aucun";

}

//Constructeur d'initialisation de capitale

public Capitale(String nom, int hab, String pays, String monument){

super(nom, hab, pays);

this.monument = monument;

}

/\*\*

\* Description d'une capitale

\* @return String retourne la description de l'objet

\*/

public String decrisToi(){

String str = super.decrisToi() + "\n \t ==>>" + this.monument + "en est un monument";

return str;

}

/\*\*

\* @return le nom du monument

\*/

public String getMonument() {

return monument;

}

//Définit le nom du monument

public void setMonument(String monument) {

this.monument = monument;

}

}

Ici décritToi est une méthode **polymorphe**.

## Le polymorphisme :

Vous aurez sans doute remarqué que je n'utilise que des objets Ville dans ma boucle : on appelle ceci la **covariance des variables** ! Cela signifie qu'une variable objet peut contenir un objet qui hérite du type de cette variable. Dans notre cas, un objet de typeVillepeut contenir un objet de typeCapitale. Dans ce cas, on dit queVilleest la **superclasse** deCapitale. La covariance est efficace dans le cas où la classe héritant redéfinit certaines méthodes de sa superclasse.

Attention à ne pas confondre la surcharge de méthode avec une méthode polymorphe.

* Une méthode surchargée diffère de la méthode originale par le nombre ou le type des paramètres qu'elle prend en entrée.
* Une méthode polymorphe a un squelette identique à la méthode de base, mais traite les choses différemment. Cette méthode se trouve dans une autre classe et donc, par extension, dans une autre instance de cette autre classe.

Un exemple :

//Définition d'un tableau de villes null

Ville[] tableau = new Ville[6];

//Définition d'un tableau de noms de Villes et un autre de nombres d'habitants

String[] tab = {"Marseille", "lille", "caen", "lyon", "paris", "nantes"};

int[] tab2 = {123456, 78456, 654987, 75832165, 1594, 213};

//Les trois premiers éléments du tableau seront des Villes

//et le reste des capitales

for(int i = 0; i < 6; i++){

if (i <3){

Ville V = new Ville(tab[i], tab2[i], "france");

tableau[i] = V;

}

else{

Capitale C = new Capitale(tab[i], tab2[i], "france", "la tour Eiffel");

tableau[i] = C;

}

}

//Il ne nous reste plus qu'à décrire tout notre tableau !

for(Object obj : tableau){

System.out.println(obj.toString()+"\n");

}

Une précision s'impose : si vous avez un objetvde typeVille, par exemple, que vous n'avez pas redéfini la méthode toString()et que vous testez ce code :

System.out.println(v);

… vous appellerez automatiquement la méthode toString()de la classe Object! Mais ici, comme vous avez redéfini la méthode toString()dans votre classe Ville, ces deux instructions sont équivalentes :

System.out.println(v.toString());

//Est équivalent à

System.out.println(v);

**Une méthode n’est invocable que si elle est défini dans cet objet**. Exemple :

public class Sdz1 {

public static void main(String[] args){

Ville[] tableau = new Ville[6];

String[] tab = {"Marseille", "lille", "caen", "lyon", "paris", "nantes"};

int[] tab2 = {123456, 78456, 654987, 75832165, 1594, 213};

for(int i = 0; i < 6; i++){

if (i <3){

Ville V = new Ville(tab[i], tab2[i], "france");

tableau[i] = V;

}

else{

Capitale C = new Capitale(tab[i], tab2[i], "france", "la tour Eiffel");

tableau[i] = C;

}

}

//Il ne nous reste plus qu'à décrire tout notre tableau !

for(Object v : tableau){

System.out.println(v.decrisToi()+"\n");

}

}

}

Ne fonctionne pas.

Pour qu'il fonctionne, vous devez dire à la JVM que la référence de type Objectest en fait une référence de type Ville, comme ceci :((Ville)v).decrisToi();. Vous transtypez la référence v en Ville par cette syntaxe. Ici, l'ordre des opérations s'effectue comme ceci :

* Vous transtypez la référence v en Ville;
* Vous appliquez la méthode decrisToi()à la référence appelante, c'est-à-dire, ici, une référence Object changée en Ville.

Il y a deux autres méthodes qui sont très souvent redéfinies :

* public boolean equals(Object o), qui permet de vérifier si un objet est égal à un autre ;
* public int hashCode(), qui attribue un code de hashage à un objet. En gros, elle donne un identifiant à un objet. Notez que cet identifiant sert plus à catégoriser votre objet qu'à l'identifier formellement.

Il faut garder en tête que ce n'est pas parce que deux objets ont un même code de hashage qu'ils sont égaux (en effet, deux objets peuvent avoir la même « catégorie » et être différents…) ; par contre, deux objets égaux ont forcément le même code de hashage ! En fait, la méthodehashcode()est utilisée par certains objets (que nous verrons avec les collections) afin de pouvoir classer les objets entre eux.

Eclipse peut générer equals et hasCode avec Source/Generate hashcode and equals.

## Final :

Une classe déclarée finale est figé, elle ne peut être hérité et ne peut être modifiée. Il en va de même pour les variables.

## La classe object :

Depuis Java7 on peut utiliser :java.util.Objects. Avec on peut utiliser hasCode() et equals() :

public int hashCode() {

return Objects.hash(categorie, nbreHabitants, nomPays, nomVille);

}

Pour l’objet Ville on aurait cette méthode :

public boolean equals(Object obj) {

//On vérifie si les références d'objets sont identiques

if (this == obj)

return true;

//On s'assure que les objets sont du même type, ici de type Ville

if (getClass() != obj.getClass())

return false;

//Maintenant, on compare les attributs de nos objets

Ville other = (Ville) obj;

return Objects.equals(other.getCategorie(), this.getCategorie()) &&

Objects.equals(other.getNom(), this.getNom()) &&

Objects.equals(other.getNombreHabitants(), this.getNombreHabitants()) &&

Objects.equals(other.getNomPays(), this.getNomPays());

}

# Modéliser ses objets grâce à UML

## Présentation d’UML :

UML permet de modéliser les liens entre les objets. Il en existe plusieurs comme :

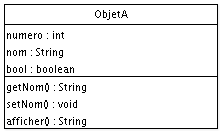
Cependant, il en existe d'autres, comme :

* boUML,
* Together,
* Poseidon,
* Pyut
* etc.

Ou encore <http://argouml.tigris.org> écrit en Java et donc multiplatform. On peut réaliser différents diagrammes comme :

* le diagramme de *use case* (cas d'utilisation) permet de déterminer les différents cas d'utilisation d'un programme informatique ;
* le diagramme de classes ; c'est de celui-là que nous allons nous servir. Il permet de modéliser des classes ainsi que les interactions entre elles ;
* les diagrammes de séquences, eux, permettent de visualiser le déroulement d'une application dans un contexte donné ;
* et d'autres encore…

## Modéliser ses objets :



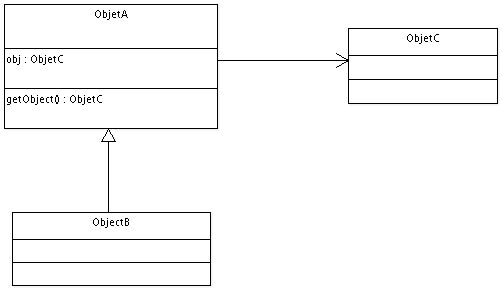
Voici une classe nommée ObjetA qui a comme attributs :

* numero de type int ;
* nom de type String ;
* bool de type boolean.

Ses méthodes sont :

* getNom() qui retourne une chaîne de caractères ;
* setNom() qui ne renvoie rien ;
* afficher() qui renvoie également une chaîne de caractères.

## Modéliser les liens :



Ici on a un objet dans un autre.

# Les packages

## Créer un package :

Ceux-ci regroupent des classes pour une meilleur lisibilité.

Il existe aussi une convention de nommage pour les packages :

* ceux-ci doivent être écrits entièrement en minuscules ;
* les caractères autorisés sont alphanumériques (de a à z, de 0 à 9) et peuvent contenir des points (.) ;
* tout package doit commencer par *com*, *edu*, *gov*, *mil*, *net*, *org* ou les deux lettres identifiant un pays (ISO Standard 3166, 1981) ; « fr » correspond à la France, « en » correspond à l'Angleterre (pour England)etc.
* aucun mot clé Java ne doit être présent dans le nom, sauf si vous le faites suivre d'un underscore (« \_ »), comme ceci : com.sdz.package\_.

## Droit d’accès entre les packages :

Les classes publics sont accessibles à l’exterieur des packages, les autres ont une portée en « default ».

Exemple, on va créer deux classes :

package com.sdz.test;

class B {

public String str ="";

}

package com.sdz.test;

public class A {

public B b = new B();

}

Vous aurez remarqué que les classes contenues dans un package ont en toute première instruction la déclaration de ce package.

Maintenant que cela est fait, afin de faire le test, créez une classe contenant la méthode main, toujours dans le même package, comme ceci :

package com.sdz.test;

public class Main {

public static void main(String[] args){

A a = new A();

B b = new B();

//Aucun problème ici

}

}

Mais si on met ce code dans com.sdz.2, on aura un problème à l’importation de com.sdz.test.b.

# Classes abstraites et interface

## Les classes abstraites :

C’est une classe normale sauf qu’on ne peut pas l’instancier.

Création d’une classe et d’une méthode abstraite :

abstract class Animal{

abstract void manger(); //Une méthode abstraite

}

On pourra donc faire :

public class Test{

public static void main(String args[]){

Animal loup = new Loup();

Animal chien = new Chien();

loup.manger();

chien.crier();

}

}

Exemple :

public class Test{

public static void main(String[] args){

Object obj = new Loup();

((Loup)obj).manger();

}

}

En revanche, ceci pose problème :

public static void main(String[] args){

Object obj = new Loup();

Loup l = obj; //Problème de référence

}

Eh oui ! Nous essayons de mettre une référence de type Object dans une référence de type Loup : pour avertir la JVM que la référence que vous voulez affecter à votre objet de type Loup est un Loup, vous devez utiliser le transtypage ! Revoyons notre code :

public static void main(String[] args){

Object obj = new Loup();

Loup l = (Loup)obj;

//Vous prévenez la JVM que la référence que vous passez est de type Loup.

}

Exemple pour ce code :

* Nos objets seront probablement tous de couleur et de poids différents. Nos classes auront donc le droit de modifier ceux-ci.
* Ici, nous partons du principe que tous nos animaux mangent de la viande. La méthode manger() sera donc définie dans la classe Animal.
* Idem pour la méthode boire(). Ils boiront tous de l'eau (je vous voyais venir).
* Ils ne crieront pas et ne se déplaceront pas de la même manière. Nous emploierons donc des méthodes polymorphes et déclarerons les méthodes deplacement() et crier() abstraites dans la classe Animal.

## Les interfaces :

Si on veut utiliser notre appli pour un chenil, on ne pourra pas mettre faireCalin() ou autre dans animal parce que c’est presque impossible pour un lion par exemple.

SI on le met directement dans chien :

1. vous allez devoir mettre en place une convention de nommage entre le programmeur qui va utiliser vos objets et vous. Vous ne pourrez pas utiliser la méthode faireCalin(), alors que le programmeur oui ;
2. si vous faites cela, adieu au polymorphisme ! Vous ne pourrez pas appeler vos objets par le biais d'un supertype. Pour pouvoir accéder à ces méthodes, vous devrez obligatoirement passer par une référence à un objet Chien. Pas terrible, tout ça !

On utilise donc les interfaces :

Une interface est 100% abstraite, pas besoin d’écrire abstract :

public interface I{

public void A();

public String B();

}

public interface I2{

public void C();

public String D();

}

Et pour faire en sorte qu'une classe utilise une interface, il suffit d'utiliser le mot clé implements. Ce qui nous donnerait :

public class X implements I{

public void A(){

//…

}

public String B(){

//…

}

}

Exemple :

public static void main(String[] args){

//Avec cette référence, vous pouvez utiliser les méthodes de l'interface I

I var = new X();

//Avec cette référence, vous pouvez utiliser les méthodes de l'interface I2

I2 var2 = new X();

var.A();

var2.C();

}

Utilisation concrète :

public class Test {

public static void main(String[] args) {

//Les méthodes d'un chien

Chien c = new Chien("Gris bleuté", 20);

c.boire();

c.manger();

c.deplacement();

c.crier();

System.out.println(c.toString());

System.out.println("--------------------------------------------");

//Les méthodes de l'interface

c.faireCalin();

c.faireLeBeau();

c.faireLechouille();

System.out.println("--------------------------------------------");

//Utilisons le polymorphisme de notre interface

Rintintin r = new Chien();

r.faireLeBeau();

r.faireCalin();

r.faireLechouille();

}

}

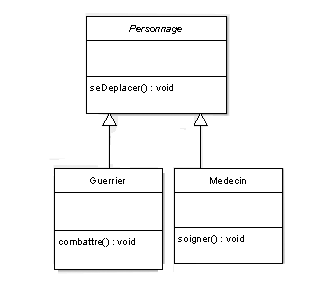
## Le pattern Strategy :

Le fait de toucher à votre hiérarchie peut amener des erreurs indésirables, voire des absurdités : tout cela parce que vous allez changer une structure qui fonctionne à cause de contraintes que l'on vous impose. Pour remédier à ce problème, il existe un concept simple (il s'agit même d'un des fondements de la programmation orientée objet) : **l'encapsulation !**

On va utiliser un pattern design (un patron de conception) pour construire la hiérarchie.

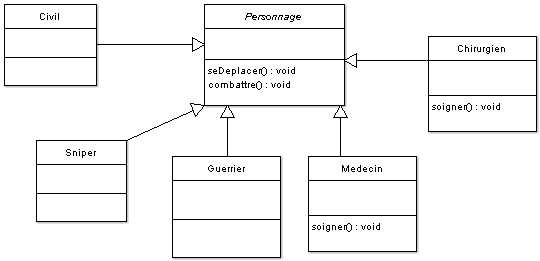
**Exemple :**

On prend le jeux z-army :



Sauf que dans le deuxième vollet on veux se battre aussi avec les médecins, on déplace combattre dans la superclasse.

On doit aussi créer beaucoup d’autres personnages :



Classe de test :

public static void main(String[] args) {

Personnage[] tPers = {new Guerrier(), new Chirurgien(), new Civil(), new Sniper(), new Medecin()};

String[] tArmes = {"pistolet", "pistolet", "couteau", "fusil à pompe", "couteau"};

for(int i = 0; i < tPers.length; i++){

System.out.println("\nInstance de " + tPers[i].getClass().getName());

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

tPers[i].combattre();

tPers[i].setArmes(tArmes[i]);

tPers[i].combattre();

tPers[i].seDeplacer();

tPers[i].soigner();

}

}

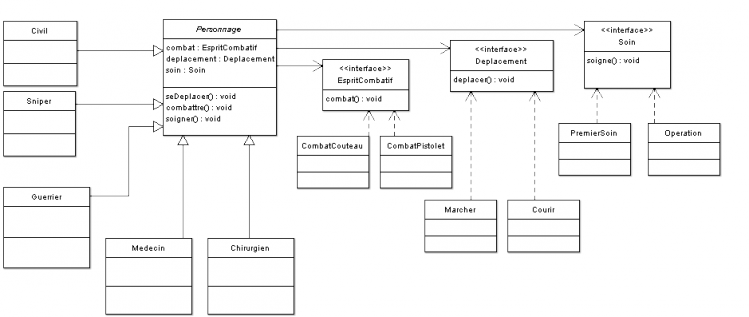
Faisons un point de la situation :

* du code dupliqué s'insinue dans votre code ;
* à chaque modification du comportement de vos personnages, vous êtes obligés de retoucher le code source de la (ou des) classe(s) concernée(s) ;
* votre code perd en « réutilisabilité » et du coup, il n'est pas extensible du tout !

La solution : le pattern strategy :

Le principe de base de ce pattern est le suivant : « isolez ce qui varie dans votre programme et encapsulez-le ! »

Nouveau diagramme :



##### Implémentations de l'interface EspritCombatif

package com.sdz.comportement;

public class Pacifiste implements EspritCombatif {

public void combat() {

System.out.println("Je ne combats pas !");

}

}

package com.sdz.comportement;

public class CombatPistolet implements EspritCombatif{

public void combat() {

System.out.println("Je combats au pitolet !");

}

}

package com.sdz.comportement;

public class CombatCouteau implements EspritCombatif {

public void combat() {

System.out.println("Je me bats au couteau !");

}

}

##### Implémentations de l'interface Deplacement

package com.sdz.comportement;

public class Marcher implements Deplacement {

public void deplacer() {

System.out.println("Je me déplace en marchant.");

}

}

package com.sdz.comportement;

public class Courir implements Deplacement {

public void deplacer() {

System.out.println("Je me déplace en courant.");

}

}

##### Implémentations de l'interface Soin

package com.sdz.comportement;

public class PremierSoin implements Soin {

public void soigne() {

System.out.println("Je donne les premiers soins.");

}

}

package com.sdz.comportement;

public class Operation implements Soin {

public void soigne() {

System.out.println("Je pratique des opérations !");

}

}

package com.sdz.comportement;

public class AucunSoin implements Soin {

public void soigne() {

System.out.println("Je ne donne AUCUN soin !");

}

}

Ainsi la classe personnage :

import com.sdz.comportement.\*;

public abstract class Personnage {

//Nos instances de comportement

protected EspritCombatif espritCombatif = new Pacifiste();

protected Soin soin = new AucunSoin();

protected Deplacement deplacement = new Marcher();

//Constructeur par défaut

public Personnage(){}

//Constructeur avec paramètres

public Personnage(EspritCombatif espritCombatif, Soin soin, Deplacement deplacement) {

this.espritCombatif = espritCombatif;

this.soin = soin;

this.deplacement = deplacement;

}

//Méthode de déplacement de personnage

public void seDeplacer(){

//On utilise les objets de déplacement de façon polymorphe

deplacement.deplacer();

}

// Méthode que les combattants utilisent

public void combattre(){

//On utilise les objets de déplacement de façon polymorphe

espritCombatif.combat();

}

//Méthode de soin

public void soigner(){

//On utilise les objets de déplacement de façon polymorphe

soin.soigne();

}

//Redéfinit le comportement au combat

public void setEspritCombatif(EspritCombatif espritCombatif) {

this.espritCombatif = espritCombatif;

}

//Redéfinit le comportement de Soin

public void setSoin(Soin soin) {

this.soin = soin;

}

//Redéfinit le comportement de déplacement

public void setDeplacement(Deplacement deplacement) {

this.deplacement = deplacement;

}

}

# Les exceptions :

## Try … catch :

Exception est une classe de Java. On peut l’utiliser par exemple, pour les divisons par 0 :

public static void main(String[] args) {

int j = 20, i = 0;

try {

System.out.println(j/i);

} catch (ArithmeticException e) {

System.out.println("Division par zéro !");

}

System.out.println("coucou toi !");

}

Finally est l’action qui sera faite quoi qu’il arrive :

public static void main(String[] args){

try {

System.out.println(" =>" + (1/0));

} catch (ClassCastException e) {

e.printStackTrace();

}

finally{

System.out.println("action faite systématiquement");

}

}

## Exceptions personnalisées :

On peut créer nos propres exceptions, par conventions ils ont Exception comme suffixe !

* throws : ce mot clé permet de signaler à la JVM qu'un morceau de code, une méthode, une classe… est potentiellement dangereux et qu'il faut utiliser un bloc try{…}catch{…}. Il est suivi du nom de la classe qui va gérer l'exception.
* throw : celui-ci permet tout simplement de lever une exception manuellement en instanciant un objet de type Exception (ou un objet hérité). Dans l'exemple de notre ArithmeticException, il y a quelque part dans les méandres de Java un throw new ArithmeticException().

Exemple :

class NombreHabitantException extends Exception{

public NombreHabitantException(){

System.out.println("Vous essayez d'instancier une classe Ville avec un nombre d'habitants négatif !");

}

}

Maintenant qu’on a créé la classe, on peut l’utiliser avec notre constructeur :

public Ville(String pNom, int pNbre, String pPays)

throws NombreHabitantException

{

if(pNbre < 0)

throw new NombreHabitantException();

else

{

nbreInstance++;

nbreInstanceBis++;

nomVille = pNom;

nomPays = pPays;

nbreHabitant = pNbre;

this.setCategorie();

}

}

A partir de maintenant, on ne pourra plus faire new Ville simplement. On devra le mettre entre try…catch dans le main avec notre Exception personnalisée.

**Attention !**

public static void main(String[] args)

{

try {

Ville v = new Ville("Rennes", 12000, "France");

} catch (NombreHabitantException e) { }

System.out.println(v.toString());

}

Ne fonctionnera pas parce que v est déclaré dans un sous-bloc et n’est donc défini que dans celui-ci. Il faut donc faire :

public static void main(String[] args)

{

Ville v = null;

try {

v = new Ville("Rennes", 12000, "France");

} catch (NombreHabitantException e) { }

System.out.println(v.toString());

}

Mais le programme plantera quand même puisque v sera null :

public static void main(String[] args)

{

Ville v = null;

try {

v = new Ville("Rennes", 12000, "France");

} catch (NombreHabitantException e) { }

finally{

if(v == null)

v = new Ville();

}

System.out.println(v.toString());

}

Ici on a résolu le problème.

On peut créer un deuxième construceur dans notre classe Exception pour renseigner le nombre choisi :

public NombreHabitantException(int nbre)

{

System.out.println("Instanciation avec un nombre d'habitants négatif.");

System.out.println("\t => " + nbre);

}

Il suffit maintenant de modifier le constructeur de la classe Ville en conséquence :

public Ville(String pNom, int pNbre, String pPays)

throws NombreHabitantException

{

if(pNbre < 0)

throw new NombreHabitantException(pNbre);

else

{

//Le code est identique à précédemment

}

}

## Gestion de plusieurs exceptions :

Imaginons que l’on veuille lever une exception si le nom de la ville comporte moins de 3 caractères :

public class NomVilleException extends Exception {

public NomVilleException(String message){

super(message);

}

}

Vous avez remarqué que nous avons utilisé super ? Avec cette redéfinition, nous pourrons afficher notre message d'erreur en utilisant la méthode getMessage().

Dans le code suivant, nous ajoutons une condition dans le constructeur Ville :

public Ville(String pNom, int pNbre, String pPays) throws NombreHabitantException, NomVilleException

{

if(pNbre < 0)

throw new NombreHabitantException(pNbre);

if(pNom.length() < 3)

throw new NomVilleException("le nom de la ville est inférieur à 3 caractères ! nom = " + pNom);

else

{

nbreInstance++;

nbreInstanceBis++;

nomVille = pNom;

nomPays = pPays;

nbreHabitant = pNbre;

this.setCategorie();

}

}

Enfin, dans le main :

Ville v = null;

try {

v = new Ville("Re", 12000, "France");

}

//Gestion de l'exception sur le nombre d'habitants

catch (NombreHabitantException e) {

e.printStackTrace();

}

//Gestion de l'exception sur le nom de la ville

catch(NomVilleException e2){

System.out.println(e2.getMessage());

}

finally{

if(v == null)

v = new Ville();

}

System.out.println(v.toString());

## Depuis Java7 : multi catch :

On peut séparer deux Exceptions grace à un pipe | :

public static void main(String[] args){

Ville v = null;

try {

v = new Ville("Re", 12000, "France");

}

//Gestion de plusieurs exceptions différentes

catch (NombreHabitantException | NomVilleException e2){

System.out.println(e2.getMessage());

}

finally{

if(v == null)

v = new Ville();

}

System.out.println(v.toString());

}

# Les énumérations

## Avant les enum :

On pouvait faire :

public class AvantEnumeration {

public static final int PARAM1 = 1;

public static final int PARAM2 = 2;

public void fait(int param){

if(param == PARAM1)

System.out.println("Fait à la façon N°1");

if(param == PARAM2)

System.out.println("Fait à la façon N°2");

}

public static void main(String args[]){

AvantEnumeration ae = new AvantEnumeration();

ae.fait(AvantEnumeration.PARAM1);

ae.fait(AvantEnumeration.PARAM2);

ae.fait(4);

}

}

## Après les enum :

Ils se construisent comme une classe mais on écrit enum à la place :

public enum Langage {

JAVA,

C,

CPlus,

PHP;

}

Ils s’utilisent comme une classe static déclarée en public.

public class Main {

public static void main(String args[]){

for(Langage lang : Langage.values()){

if(Langage.JAVA.equals(lang))

System.out.println("J'aime le : " + lang);

else

System.out.println(lang);

}

}

}

Exemple complet :

public enum Langage {

//Objets directement construits

JAVA("Langage JAVA", "Eclipse"),

C ("Lanage C", "Code Block"),

CPlus ("Langage C++", "Visual studio"),

PHP ("Langage PHP", "PS Pad");

private String name = "";

private String editor = "";

//Constructeur

Langage(String name, String editor){

this.name = name;

this.editor = editor;

}

public void getEditor(){

System.out.println("Editeur : " + editor);

}

public String toString(){

return name;

}

public static void main(String args[]){

Langage l1 = Langage.JAVA;

Langage l2 = Langage.PHP;

l1.getEditor();

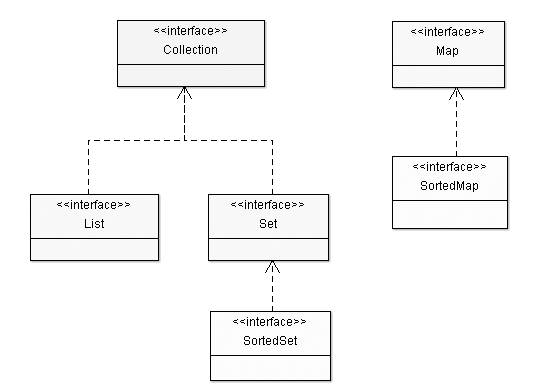
l2.getEditor();

}

}

# Les collections d’objet

## Les différents types de collection :

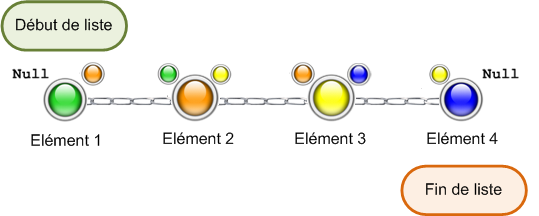


L’objet set est assez restrictif car il n’autorise qu’une seule fois le même objet à l’intérieur.

## Objet List :

On y trouve les objets Vector, ArrayList et LinkedList.

### LinkedList :



LinkedList implémente l’interface Itérator, on peut donc l’utiliser pour lister les éléments de l’objet.

import java.util.LinkedList;

import java.util.List;

import java.util.ListIterator;

public class Test {

public static void main(String[] args) {

List l = new LinkedList();

l.add(12);

l.add("toto ! !");

l.add(12.20f);

for(int i = 0; i < l.size(); i++)

System.out.println("Élément à l'index " + i + " = " + l.get(i));

System.out.println("\n \tParcours avec un itérateur ");

System.out.println("-----------------------------------");

ListIterator li = l.listIterator();

while(li.hasNext())

System.out.println(li.next());

}

}

### L’objet ArrayList :

Les ArrayList acceptent tout type de donnée mais null. On peut donc y mettre n’importe quoi.

import java.util.ArrayList;

public class Test {

public static void main(String[] args) {

ArrayList al = new ArrayList();

al.add(12);

al.add("Une chaîne de caractères !");

al.add(12.20f);

al.add('d');

for(int i = 0; i < al.size(); i++)

{

System.out.println("donnée à l'indice " + i + " = " + al.get(i));

}

}

}

Les méthodes des ArrayList :

* add() permet d'ajouter un élément ;
* get(int index) retourne l'élément à l'indice demandé ;
* remove(int index) efface l'entrée à l'indice demandé ;
* isEmpty() renvoie « vrai » si l'objet est vide ;
* removeAll() efface tout le contenu de l'objet ;
* contains(Object element) retourne « vrai » si l'élément passé en paramètre est dans l'ArrayList.

## Les objets Map :

Collection qui fonctionne avec un couple clé valeur. On y trouve les objets

* Hashtable
* HashMap
* TreeMap
* WeakHashMap

### Hashtable :

On parcourt l’objet Hashtable grace à la classe Enumeration :

import java.util.Enumeration;

import java.util.Hashtable;

public class Test {

public static void main(String[] args) {

Hashtable ht = new Hashtable();

ht.put(1, "printemps");

ht.put(10, "été");

ht.put(12, "automne");

ht.put(45, "hiver");

Enumeration e = ht.elements();

while(e.hasMoreElements())

System.out.println(e.nextElement());

}

}

Les méthodes :

* isEmpty() retourne « vrai » si l'objet est vide ;
* contains(Object value) retourne « vrai » si la valeur est présente. Identique à containsValue(Object value) ;
* containsKey(Object key) retourne « vrai » si la clé passée en paramètre est présente dans la Hashtable ;
* put(Object key, Object value) ajoute le couple key - value dans l'objet ;
* elements() retourne une énumération des éléments de l'objet ;
* keys() retourne la liste des clés sous forme d'énumération.

La classe Enumeration a deux methodes :

|  |  |
| --- | --- |
| boolean | [**hasMoreElements**](https://docs.oracle.com/javase/7/docs/api/java/util/Enumeration.html#hasMoreElements())()  Tests if this enumeration contains more elements. |
| [**E**](https://docs.oracle.com/javase/7/docs/api/java/util/Enumeration.html) | [**nextElement**](https://docs.oracle.com/javase/7/docs/api/java/util/Enumeration.html#nextElement())()  Returns the next element of this enumeration if this enumeration object has at least one more element to provide. |

### L’objet HashMap :

Quasi identique sauf qu’il :

* il accepte la valeur null ;
* il n'est pas Thread Safe.

## Les objets set :

Objet qui n’accepte pas les doublons (même null).

On trouve les objets :

* HashSet
* TreeSet
* KinkedHashSet
* …

### HashSet :

Le plus utilisé, on utilise un Iterator :

import java.util.HashSet;

import java.util.Iterator;

public class Test {

public static void main(String[] args) {

HashSet hs = new HashSet();

hs.add("toto");

hs.add(12);

hs.add('d');

Iterator it = hs.iterator();

while(it.hasNext())

System.out.println(it.next());

System.out.println("\nParcours avec un tableau d'objet");

System.out.println("-----------------------------------");

Object[] obj = hs.toArray();

for(Object o : obj)

System.out.println(o);

}

}

Les méthodes :

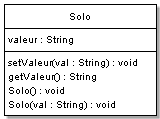
* add() ajoute un élément ;
* contains(Object value) retourne « vrai » si l'objet contient value ;
* isEmpty() retourne « vrai » si l'objet est vide ;
* iterator() renvoie un objet de type Iterator ;
* remove(Object o) retire l'objet o de la collection ;
* toArray() retourne un tableau d'Object.

# Généricité en Java

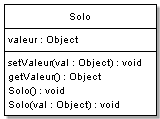
Le principe de la généricité est de faire des classes qui n’acceptent qu’un certain type d’objet ou de donnée de façon dynamique.

## Principe de base :

**Cas d’une classe qui ne l’utilise pas :**



Pour faire une classe qui accepte tout type de donnée, première idée :

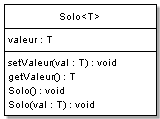


Sauf qu’avec cette configuration on est obligé de faire un cast.

Solo val = new Solo(12);

int nbre = (Integer)val.getValeur();

**Classe avec la généricité :**



Voici le code :

public class Solo<T> {

//Variable d'instance

private T valeur;

//Constructeur par défaut

public Solo(){

this.valeur = null;

}

//Constructeur avec paramètre inconnu pour l'instant

public Solo(T val){

this.valeur = val;

}

//Définit la valeur avec le paramètre

public void setValeur(T val){

this.valeur = val;

}

//Retourne la valeur déjà « castée » par la signature de la méthode !

public T getValeur(){

return this.valeur;

}

}

On définit T à l’instanciation de la classe :

public static void main(String[] args) {

Solo<Integer> val = new Solo<Integer>(12);

int nbre = val.getValeur();

}

Seulement cela ne fonctionne qu’avec le type que l’on a spécifié. Si on écrit Integer on ne pourra plus rentrer de String.

On peut aussi faire :

public static void main(String[] args) {

Solo<Integer> val = new Solo<Integer>();

Solo<String> valS = new Solo<String>("TOTOTOTO");

Solo<Float> valF = new Solo<Float>(12.2f);

Solo<Double> valD = new Solo<Double>(12.202568);

}

Quand on déclare une variable de type primitif, on peut utiliser ses classes enveloppes, aussi appelé WRAPPER. Elles ajoutent automatiquement les méthodes du type (String pas exemple), ainsi que des méthodes pour caster leurs valeurs.

Depuis Java5, Java gère l’AUTOBOXING, qui permet de transformer un type primitif en classe WRAPPER (le boxing) et inversement (unboxing). :

public static void main(String[] args){

int i = new Integer(12); //Est équivalent à int i = 12

double d = new Double(12.2586); //Est équivalent à double d = 12.2586

Double d = 12.0;

Character c = 'C';

al = new ArrayList();

//Avant Java 5 il fallait faire al.add(new Integer(12))

//Depuis Java 5 il suffit de faire

al.add(12);

//…

}

### Plus loin dans la généricité :

On pourrait très bien créer une classe avec deux variables génériques :

public class Duo<T, S> {

//Variable d'instance de type T

private T valeur1;

//Variable d'instance de type S

private S valeur2;

//Constructeur par défaut

public Duo(){

this.valeur1 = null;

this.valeur2 = null;

}

//Constructeur avec paramètres

public Duo(T val1, S val2){

this.valeur1 = val1;

this.valeur2 = val2;

}

//Méthodes d'initialisation des deux valeurs

public void setValeur(T val1, S val2){

this.valeur1 = val1;

this.valeur2 = val2;

}

//Retourne la valeur T

public T getValeur1() {

return valeur1;

}

//Définit la valeur T

public void setValeur1(T valeur1) {

this.valeur1 = valeur1;

}

//Retourne la valeur S

public S getValeur2() {

return valeur2;

}

//Définit la valeur S

public void setValeur2(S valeur2) {

this.valeur2 = valeur2;

}

}

Et ainsi les instancier :

public static void main(String[] args) {

Duo<String, Boolean> dual = new Duo<String, Boolean>("toto", true);

System.out.println("Valeur de l'objet dual : val1 = " + dual.getValeur1() + ", val2 = " + dual.getValeur2());

Duo<Double, Character> dual2 = new Duo<Double, Character>(12.2585, 'C');

System.out.println("Valeur de l'objet dual2 : val1 = " + dual2.getValeur1() + ", val2 = " + dual2.getValeur2());

}

Mais attention à ne pas redéfinir la variable que l’on a déjà instancié. Par exempel duo<Integer,Boolean> val en duo <Integer, Character> val.

## Généricité et collection :

On peut aussi utiliser la généricité sur les objets servant à gérer les collections.

Exemple : Avec une ArrayList on n’est jamais sûr du type sur lequel on va tomber en parcourant le tableau puisqu’il peut tout contenir. On va utiliser le polymorphisme.

public static void main(String[] args) {

System.out.println("Liste de String");

System.out.println("------------------------------");

List<String> listeString= new ArrayList<String>();

listeString.add("Une chaîne");

listeString.add("Une autre");

listeString.add("Encore une autre");

listeString.add("Allez, une dernière");

for(String str : listeString)

System.out.println(str);

System.out.println("\nListe de float");

System.out.println("------------------------------");

List<Float> listeFloat = new ArrayList<Float>();

listeFloat.add(12.25f);

listeFloat.add(15.25f);

listeFloat.add(2.25f);

listeFloat.add(128764.25f);

for(float f : listeFloat)

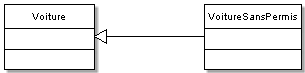
System.out.println(f);

}

Ce sont les mêmes règles que pécédement : pas de float dans une liste de String et invérsement.

### Héritage et généricité :

Imaginons que l’on ait une classe Voiture et une classe VoitureSansPermis qui hérite de Voiture :



public static void main(String[] args) {

List<Voiture> listVoiture = new ArrayList<Voiture>();

List<VoitureSansPermis> listVoitureSP = new ArrayList<VoitureSansPermis>();

listVoiture = listVoitureSP; //Interdit !

}

C’est interdit car on aura à un moment une référence de type voitureSansPermis à laquelle on va tenter de lui affecter la référence Voiture.

Une des solutions consiste à utiliser le wildcard : « ? ». Le fait de déclarer une collection avec le wildcard, comme ceci :

ArrayList<?> list;

… revient à indiquer que notre collection accepte n'importe quel type d'objet.

Par exemple, pour définir une liste qui n’accepte qu’un objet de class Voiture ou un de ses sous-classes :

public static void main(String[] args) {

List<? extends Voiture> listVoitureSP = new ArrayList<VoitureSansPermis>();

afficher(listVoitureSP);

}

//Méthode générique !

static void afficher(ArrayList<? extends Voiture> list){

for(Voiture v : list)

System.out.println(v.toString());

}

Mais en utilisant le wildcard, la liste se met en lecture seule, on ne peut plus insérer.

Du coup, ça fonctionne bien en lecture :

public static void main(String[] args){

//Liste de voiture

List<Voiture> listVoiture = new ArrayList<Voiture>();

listVoiture.add(new Voiture());

listVoiture.add(new Voiture());

List<VoitureSansPermis> listVoitureSP = new ArrayList<VoitureSansPermis>();

listVoitureSP.add(new VoitureSansPermis());

listVoitureSP.add(new VoitureSansPermis());

affiche(listVoiture);

affiche(listVoitureSP);

}

//Avec cette méthode, on accepte aussi bien les collections de Voiture que les collection de VoitureSansPermis

static void affiche(List<? extends Voiture> list){

for(Voiture v : list)

System.out.print(v.toString());

}

On aura au moins une seule méthode pour lire tout ce qui déscend de Voiture.

Si on remplace extends par super, on autorise n’importe quelle classe superclass de Voiture dont Voiture elle-même :

public static void main(String[] args){

//Liste de voiture

List<Voiture> listVoiture = new ArrayList<Voiture>();

listVoiture.add(new Voiture());

listVoiture.add(new Voiture());

List<Object> listVoitureSP = new ArrayList<Object>();

listVoitureSP.add(new Object());

listVoitureSP.add(new Object());

affiche(listVoiture);

}

//Avec cette méthode, on accepte aussi bien les collections de Voiture que les collections d'Object : superclasse de toutes les classes

static void affiche(List<? super Voiture> list){

for(Object v : list)

System.out.print(v.toString());

}

En définitif, le wildcard sert surtout à retrouver le polymorphisme avec les Collections :

import java.util.ArrayList;

import java.util.List;

public class Garage {

List<Voiture> list = new ArrayList<Voiture>();

public void add(List<? extends Voiture> listVoiture){

for(Voiture v : listVoiture)

list.add(v);

System.out.println("Contenu de notre garage :");

for(Voiture v : list)

System.out.print(v.toString());

}

}

Un petit test rapide :

public static void main(String[] args){

List<Voiture> listVoiture = new ArrayList<Voiture>();

listVoiture.add(new Voiture());

List<VoitureSansPermis> listVoitureSP = new ArrayList<VoitureSansPermis>();

listVoitureSP.add(new VoitureSansPermis());

Garage garage = new Garage();

garage.add(listVoiture);

System.out.println("--------------------------");

garage.add(listVoitureSP);

}

# Les flux entrée/sortie

Pour réaliser des entrées sorties, on utilise des stream. Des objets différents pour les deux catégories de flux :

* Les objets travaillant avec des flux d’entrée : (in) pour la lecture.
* Les objets travaillant avec des flux de sortie : (out) pour l’écriture.

## Utilisation de Java.io

### L’objet File :

Avant de commencer on créer un fichier texte à la racine du projet. Exempel test.txt

//Package à importer afin d'utiliser l'objet File

import java.io.File;

public class Main {

public static void main(String[] args) {

//Création de l'objet File

File f = new File("test.txt");

System.out.println("Chemin absolu du fichier : " + f.getAbsolutePath());

System.out.println("Nom du fichier : " + f.getName());

System.out.println("Est-ce qu'il existe ? " + f.exists());

System.out.println("Est-ce un répertoire ? " + f.isDirectory());

System.out.println("Est-ce un fichier ? " + f.isFile());

System.out.println("Affichage des lecteurs à la racine du PC : ");

for(File file : f.listRoots())

{

System.out.println(file.getAbsolutePath());

try {

int i = 1;

//On parcourt la liste des fichiers et répertoires

for(File nom : file.listFiles()){

//S'il s'agit d'un dossier, on ajoute un "/"

System.out.print("\t\t" + ((nom.isDirectory()) ? nom.getName()+"/" : nom.getName()));

if((i%4) == 0){

System.out.print("\n");

}

i++;

}

System.out.println("\n");

} catch (NullPointerException e) {

//L'instruction peut générer une NullPointerException

//s'il n'y a pas de sous-fichier !

}

}

}

}

On peut aussi,

* Supprimer le fichier avec *delete()*
* Ajouter un répertoire avec *mkdir()*

### Les objets FileInputStream et FileOutputStream :

C’est grace à ces deux objets que l’on va pouvoir lire et écrire dans un fichier.

Ces classes héritent des classes abstraites *InputStream* et *OutputStream*.

Exemple pour travailler avec le fichier :

//Packages à importer afin d'utiliser les objets

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

// Nous déclarons nos objets en dehors du bloc try/catch

FileInputStream fis = null;

FileOutputStream fos = null;

try {

// On instancie nos objets :

// fis va lire le fichier

// fos va écrire dans le nouveau !

fis = new FileInputStream(new File("test.txt"));

fos = new FileOutputStream(new File("test2.txt"));

// On crée un tableau de byte pour indiquer le nombre de bytes lus à

// chaque tour de boucle

byte[] buf = new byte[8];

// On crée une variable de type int pour y affecter le résultat de

// la lecture

// Vaut -1 quand c'est fini

int n = 0;

// Tant que l'affectation dans la variable est possible, on boucle

// Lorsque la lecture du fichier est terminée l'affectation n'est

// plus possible !

// On sort donc de la boucle

while ((n = fis.read(buf)) >= 0) {

// On écrit dans notre deuxième fichier avec l'objet adéquat

fos.write(buf);

// On affiche ce qu'a lu notre boucle au format byte et au

// format char

for (byte bit : buf) {

System.out.print("\t" + bit + "(" + (char) bit + ")");

}

System.out.println("");

//Nous réinitialisons le buffer à vide

//au cas où les derniers byte lus ne soient pas un multiple de 8

//Ceci permet d'avoir un buffer vierge à chaque lecture et ne pas avoir de doublon en fin de fichier

buf = new byte[8];

}

System.out.println("Copie terminée !");

} catch (FileNotFoundException e) {

// Cette exception est levée si l'objet FileInputStream ne trouve

// aucun fichier

e.printStackTrace();

} catch (IOException e) {

// Celle-ci se produit lors d'une erreur d'écriture ou de lecture

e.printStackTrace();

} finally {

// On ferme nos flux de données dans un bloc finally pour s'assurer

// que ces instructions seront exécutées dans tous les cas même si

// une exception est levée !

try {

if (fis != null)

fis.close();

} catch (IOException e) {

e.printStackTrace();

}

try {

if (fos != null)

fos.close();

} catch (IOException e) {

e.printStackTrace();

}

}

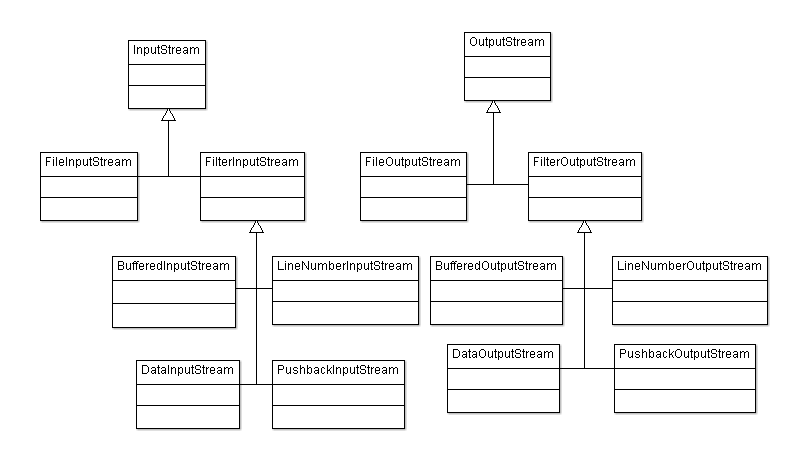
}

}

Il est impossible de lire dans un fichier inexistant, on aura une Exception *FileNotFoundException* par contre on peut écrire dans un fichier inexistant, il sera créé.

* les espaces :SPpour « SPace », code décimal 32 ;
* les sauts de lignes :LFpour « Line Feed », code décimal 13 ;
* les retours chariot :CRpour « Carriage Return », code décimal 10.

### Les objets FilterInputStream et FilterOutputStream :



Ainsi les classes Filter, étant abstraites, définissent un comportement pour leurs classes filles.

* DataInputStream : offre la possibilité de lire directement des types primitifs (double, char, int) grâce à des méthodes comme readDouble(),readInt()…
* BufferedInputStream : cette classe permet d'avoir un tampon à disposition dans la lecture du flux. En gros, les données vont tout d'abord remplir le tampon, et dès que celui-ci est plein, le programme accède aux données.
* PushbackInputStream : permet de remettre un octet déjà lu dans le flux entrant.
* LineNumberInputStream : cette classe offre la possibilité de récupérer le numéro de la ligne lue à un instant T.

Puisque ces classes acceptent une instance de leur superclasse en paramètre, vous pouvez cumuler les filtres et obtenir des choses de ce genre :

FileInputStream fis = new FileInputStream(new File("toto.txt"));

DataInputStream dis = new DataInputStream(fis);

BufferedInputStream bis = new BufferedInputStream(dis);

//Ou en condensé :

BufferedInputStream bis = new BufferredInputStream(

new DataInputStream(

new FileInputStream(

new File("toto.txt"))));

Si on met un autre fichier test, plus gros, on peut comparer le temps d’exécution en milliseconde :

//Packages à importer afin d'utiliser l'objet File

import java.io.BufferedInputStream;

import java.io.DataInputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

//Nous déclarons nos objets en dehors du bloc try/catch

FileInputStream fis;

BufferedInputStream bis;

try {

fis = new FileInputStream(new File("test.txt"));

bis = new BufferedInputStream(new FileInputStream(new File("test.txt")));

byte[] buf = new byte[8];

//On récupère le temps du système

long startTime = System.currentTimeMillis();

//Inutile d'effectuer des traitements dans notre boucle

while(fis.read(buf) != -1);

//On affiche le temps d'exécution

System.out.println("Temps de lecture avec FileInputStream : " + (System.currentTimeMillis() - startTime));

//On réinitialise

startTime = System.currentTimeMillis();

//Inutile d'effectuer des traitements dans notre boucle

while(bis.read(buf) != -1);

//On réaffiche

System.out.println("Temps de lecture avec BufferedInputStream : " + System.currentTimeMillis() - startTime));

//On ferme nos flux de données

fis.close();

bis.close();

} catch (FileNotFoundException e) {

e.printStackTrace();

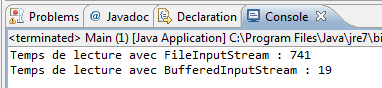
} catch (IOException e) {

e.printStackTrace();

}

}

}



Si on fait le test avec l’écriture :

//Packages à importer afin d'utiliser l'objet File

import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

//Nous déclarons nos objets en dehors du bloc try/catch

FileInputStream fis;

FileOutputStream fos;

BufferedInputStream bis;

BufferedOutputStream bos;

try {

fis = new FileInputStream(new File("test.txt"));

fos = new FileOutputStream(new File("test2.txt"));

bis = new BufferedInputStream(new FileInputStream(new File("test.txt")));

bos = new BufferedOutputStream(new FileOutputStream(new File("test3.txt")));

byte[] buf = new byte[8];

//On récupère le temps du système

long startTime = System.currentTimeMillis();

while(fis.read(buf) != -1){

fos.write(buf);

}

//On affiche le temps d'exécution

System.out.println("Temps de lecture + écriture avec FileInputStream et FileOutputStream : " + (System.currentTimeMillis() - startTime));

//On réinitialise

startTime = System.currentTimeMillis();

while(bis.read(buf) != -1){

bos.write(buf);

}

//On réaffiche

System.out.println("Temps de lecture + écriture avec BufferedInputStream et BufferedOutputStream : " + (System.currentTimeMillis() - startTime));

//On ferme nos flux de données

fis.close();

bis.close();

fos.close();

bos.close();

} catch (FileNotFoundException e) {

e.printStackTrace();

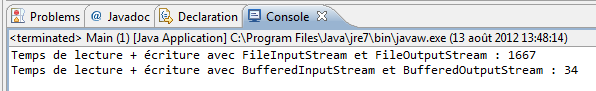
} catch (IOException e) {

e.printStackTrace();

}

}

}



**Les DataInput/OuputStream** :

Fonctionnent comme les Buffered.

//Packages à importer afin d'utiliser l'objet File

import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

//Nous déclarons nos objets en dehors du bloc try/catch

DataInputStream dis;

DataOutputStream dos;

try {

dos = new DataOutputStream(

new BufferedOutputStream(

new FileOutputStream(

new File("sdz.txt"))));

//Nous allons écrire chaque type primitif

dos.writeBoolean(true);

dos.writeByte(100);

dos.writeChar('C');

dos.writeDouble(12.05);

dos.writeFloat(100.52f);

dos.writeInt(1024);

dos.writeLong(123456789654321L);

dos.writeShort(2);

dos.close();

//On récupère maintenant les données !

dis = new DataInputStream(

new BufferedInputStream(

new FileInputStream(

new File("sdz.txt"))));

System.out.println(dis.readBoolean());

System.out.println(dis.readByte());

System.out.println(dis.readChar());

System.out.println(dis.readDouble());

System.out.println(dis.readFloat());

System.out.println(dis.readInt());

System.out.println(dis.readLong());

System.out.println(dis.readShort());

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

}

}

Travailler aves des objets :

### ObjectInputStream / ObjectOutputStream :

Ecrire un objet **dans** un fichier = SERIALISATION

Exemple de serialisation :

//Package à importer

import java.io.Serializable;

public class Game implements Serializable{

private String nom, style;

private double prix;

public Game(String nom, String style, double prix) {

this.nom = nom;

this.style = style;

this.prix = prix;

}

public String toString(){

return "Nom du jeu : " + this.nom + "\n

Style de jeu : " + this.style + "\n

Prix du jeu : " + this.prix + "\n";

}

}

**L’interface Serializable est une « interface marqueur »**

Rien qu’en implémentant, Java sait que l’on peut serialiser ses objets.

Si une superclass implémente Serializable, ses classes filles seront considérées comme telle aussi.

Exemple :

On va serialiser deux, trois objets, puis les désérialiser pour les réutiliser :

//Packages à importer afin d'utiliser l'objet File

import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

public class Main {

public static void main(String[] args) {

//Nous déclarons nos objets en dehors du bloc try/catch

ObjectInputStream ois;

ObjectOutputStream oos;

try {

oos = new ObjectOutputStream(

new BufferedOutputStream(

new FileOutputStream(

new File("game.txt"))));

//Nous allons écrire chaque objet Game dans le fichier

oos.writeObject(new Game("Assassin Creed", "Aventure", 45.69));

oos.writeObject(new Game("Tomb Raider", "Plateforme", 23.45));

oos.writeObject(new Game("Tetris", "Stratégie", 2.50));

//Ne pas oublier de fermer le flux !

oos.close();

//On récupère maintenant les données !

ois = new ObjectInputStream(

new BufferedInputStream(

new FileInputStream(

new File("game.txt"))));

try {

System.out.println("Affichage des jeux :");

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

System.out.println(((Game)ois.readObject()).toString());

System.out.println(((Game)ois.readObject()).toString());

System.out.println(((Game)ois.readObject()).toString());

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

ois.close();

} catch (FileNotFoundException e) {

e.printStackTrace();

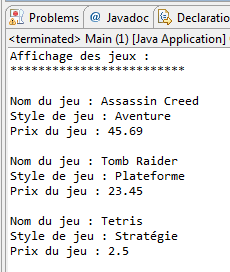
} catch (IOException e) {

e.printStackTrace();

}

}

}



**Et si notre objet contenait un autre objet ?**

Par exemple une notice :

public class Notice {

private String langue ;

public Notice(){

this.langue = "Français";

}

public Notice(String lang){

this.langue = lang;

}

public String toString() {

return "\t Langue de la notice : " + this.langue + "\n";

}

}

Nous allons maintenant implémenter une notice par défaut dans notre objetGame. Voici notre classe modifiée :

import java.io.Serializable;

public class Game implements Serializable{

private String nom, style;

private double prix;

private Notice notice;

public Game(String nom, String style, double prix) {

this.nom = nom;

this.style = style;

this.prix = prix;

this.notice = new Notice();

}

public String toString(){

return "Nom du jeu : " + this.nom + "\n

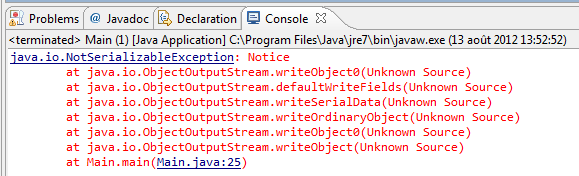
Style de jeu : " + this.style + "\n

Prix du jeu : " + this.prix + "\n";

}

}

On obitent :



L’erreur est due au fait que notice n’est pas serialisable. Soit :

* On fait en sorte de rendre notice serialisable
* Soit on dit dans *Game* que notice n’est pas à serialiser

Pour la première option on implémente simplement *Serializable*. Pour le second on doit déclarer l’objet notice comme étant *transient* :

import java.io.Serializable;

public class Game implements Serializable{

private String nom, style;

private double prix;

//Maintenant, cette variable ne sera pas sérialisée

//Elle sera tout bonnement ignorée !

private transient Notice notice;

public Game(String nom, String style, double prix) {

this.nom = nom;

this.style = style;

this.prix = prix;

this.notice = new Notice();

}

public String toString(){

return "Nom du jeu : " + this.nom + "\n

Style de jeu : " + this.style + "\n

Prix du jeu : " + this.prix + "\n";

}

}

**Important, on ne met pas la notice dans le toString, car celle-ci est ignorée, on aura donc un *NullPointerException*.**

### Les objets CharArray(Writer/Reader) et String(Writer/Reader) :

Font presque la même chose et ont les mêmes méthodes que leur classe mère. Ils servent juste à écrire un flux de caractère dans un buffer adaptatif.

**CharArray(Writer/Reader) :**

//Packages à importer afin d'utiliser l'objet File

import java.io.CharArrayReader;

import java.io.CharArrayWriter;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

CharArrayWriter caw = new CharArrayWriter();

CharArrayReader car;

try {

caw.write("Coucou les Zéros");

//Appel à la méthode toString de notre objet de manière tacite

System.out.println(caw);

//caw.close() n'a aucun effet sur le flux

//Seul caw.reset() peut tout effacer

caw.close();

//On passe un tableau de caractères à l'objet qui va lire le tampon

car = new CharArrayReader(caw.toCharArray());

int i;

//On remet tous les caractères lus dans un String

String str = "";

while(( i = car.read()) != -1)

str += (char) i;

System.out.println(str);

} catch (IOException e) {

e.printStackTrace();

}

}

}

**String(Writer/Reader) :**

//Packages à importer afin d'utiliser l'objet File

import java.io.IOException;

import java.io.StringReader;

import java.io.StringWriter;

public class Main {

public static void main(String[] args) {

StringWriter sw = new StringWriter();

StringReader sr;

try {

sw.write("Coucou les Zéros");

//Appel à la méthode toString de notre objet de manière tacite

System.out.println(sw);

//caw.close() n'a aucun effet sur le flux

//Seul caw.reset() peut tout effacer

sw.close();

//On passe un tableau de caractères à l'objet qui va lire le tampon

sr = new StringReader(sw.toString());

int i ;

//On remet tous les caractères lus dans un String

String str = "";

while(( i = sr.read()) != -1)

str += (char) i;

System.out.println(str);

} catch (IOException e) {

e.printStackTrace();

}

}

}

### File(Writer/Reader) et Print(Writer/Reader) :

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

File file = new File("testFileWriter.txt");

FileWriter fw;

FileReader fr;

try {

//Création de l'objet

fw = new FileWriter(file);

String str = "Bonjour à tous, amis Zéros !\n";

str += "\tComment allez-vous ? \n";

//On écrit la chaîne

fw.write(str);

//On ferme le flux

fw.close();

//Création de l'objet de lecture

fr = new FileReader(file);

str = "";

int i = 0;

//Lecture des données

while((i = fr.read()) != -1)

str += (char)i;

//Affichage

System.out.println(str);

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Java.io existe depuis le jdk 1.1, alors que Java.nio est récent.**

## Java.nio :

Nio : New input output.

Rappel : java.io traite par octet, alors que java.nio traite des blocs de données 🡺 lecture accelérée !

Tout repose sur deux objets : *channels* et *buffers*

*Channels* : sont des flux, ils travaillent avec un *buffers* dont on définit la taille.

En gros, quand on ouvre un flux vers un fichier, on peut récupérer un canal vers ce fichier. Combiné à un buffer, on lira encore plus vite qu’avec *BufferedReaderStream*.

Si on reprend le gros fichier text :

//Packages à importer afin d'utiliser l'objet File

import java.io.BufferedInputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

import java.nio.ByteBuffer;

import java.nio.CharBuffer;

import java.nio.channels.FileChannel;

public class Main {

public static void main(String[] args) {

FileInputStream fis;

BufferedInputStream bis;

FileChannel fc;

try {

//Création des objets

fis = new FileInputStream(new File("test.txt"));

bis = new BufferedInputStream(fis);

//Démarrage du chrono

long time = System.currentTimeMillis();

//Lecture

while(bis.read() != -1);

//Temps d'exécution

System.out.println("Temps d'exécution avec un buffer conventionnel : " + (System.currentTimeMillis() - time));

//Création d'un nouveau flux de fichier

fis = new FileInputStream(new File("test.txt"));

//On récupère le canal

fc = fis.getChannel();

//On en déduit la taille

int size = (int)fc.size();

//On crée un buffer correspondant à la taille du fichier

ByteBuffer bBuff = ByteBuffer.allocate(size);

//Démarrage du chrono

time = System.currentTimeMillis();

//Démarrage de la lecture

fc.read(bBuff);

//On prépare à la lecture avec l'appel à flip

bBuff.flip();

//Affichage du temps d'exécution

System.out.println("Temps d'exécution avec un nouveau buffer : " + (System.currentTimeMillis() - time));

//Puisque nous avons utilisé un buffer de byte afin de récupérer les données

//Nous pouvons utiliser un tableau de byte

//La méthode array retourne un tableau de byte

byte[] tabByte = bBuff.array();

} catch (FileNotFoundException e) {

e.printStackTrace();

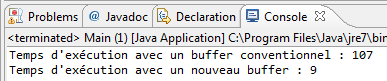
} catch (IOException e) {

e.printStackTrace();

}

}

}



Ce package offre un buffer par type primitif pour la lecture sur le channel, vous trouverez donc ces classes :

* IntBuffer;
* CharBuffer;
* ShortBuffer;
* ByteBuffer;
* DoubleBuffer;
* FloatBuffer;
* LongBuffer.

Pour s’assurer que le flux est bien fermé :

//Packages à importer afin d'utiliser l'objet File

//…

public class Main {

public static void main(String[] args) {

//Nous déclarons nos objets en dehors du bloc try / catch

ObjectInputStream ois;

ObjectOutputStream oos;

try {

//On travaille avec nos objets

} catch (FileNotFoundException e) {

//Gestion des exceptions

} catch (IOException e) {

//Gestion des exceptions

}

finally{

if(ois != null)ois.close();

if(oos != null)oos.close();

}

}

}

On met un test dans finally.

Avec Java7, plusieurs nouveauté, à part la gestion de la mémoire qui est déléguée au *Garbage Collector* (ramasse miette), le reste est à ouvrir et fermer manuellement (flux de donnée, connexion BDD …)

Depuis Java7, Java a aussi initialiser *try-with-ressources* qui ferme automatiquement ce qui a été ouvert dans le try.

try(FileInputStream fis = new FileInputStream("test.txt");

FileOutputStream fos = new FileOutputStream("test2.txt")) {

byte[] buf = new byte[8];

int n = 0;

while((n = fis.read(buf)) >= 0){

fos.write(buf);

for(byte bit : buf)

System.out.print("\t" + bit + "(" + (char)bit + ")");

System.out.println("");

}

System.out.println("Copie terminée !");

} catch (IOException e) {

e.printStackTrace();

}

**A noter que chaque paramètre est séparé d’un ; dans le try !**

Mais si on encapsule :

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("test.txt"))) {

//…

}

FileInputStream ne sera pas fermer automatiquement, il faut bien penser à les déclarer un à un séparés par un point-virgule.

## Depuis Java7 : Java.nio II :

Nouveau package : java.nio.file, en remplacement à java.io.file :

* une meilleure gestion des exceptions : la plupart des méthodes de la classe File se contentent de renvoyer une valeur nulle en cas de problème, avec ce nouveau package, des exceptions seront levées permettant de mieux cibler la cause du (ou des) problème(s) ;
* un accès complet au système de fichiers (support des liens/liens symboliques, etc.) ;
* l'ajout de méthodes utilitaires tels que le déplacement/la copie de fichier, la lecture/écriture binaire ou texte…
* récupérer la liste des fichiers d'un répertoire via un flux ;
* remplacement de la classe java.io.File par l'interface java.nio.file.Path.

Path path = Paths.get("test.txt");

System.out.println("Chemin absolu du fichier : " + path.toAbsolutePath());

System.out.println("Est-ce qu'il existe ? " + Files.exists(path));

System.out.println("Nom du fichier : " + path.getFileName());

System.out.println("Est-ce un répertoire ? " + Files.isDirectory(path));

La classe Files vous permet aussi de lister le contenu d'un répertoire mais via un objet DirectoryStream qui est un itérateur. Ceci évite de charger tous les fichiers en mémoire pour récupérer leurs informations. Voici comment procéder :

//On récupère maintenant la liste des répertoires dans une collection typée

//Via l'objet FileSystem qui représente le système de fichier de l'OS hébergeant la JVM

Iterable<Path> roots = FileSystems.getDefault().getRootDirectories();

//Maintenant, il ne nous reste plus qu'à parcourir

for(Path chemin : roots){

System.out.println(chemin);

//Pour lister un répertoire, il faut utiliser l'objet DirectoryStream

//L'objet Files permet de créer ce type d'objet afin de pouvoir l'utiliser

try(DirectoryStream<Path> listing = Files.newDirectoryStream(chemin)){

int i = 0;

for(Path nom : listing){

System.out.print("\t\t" + ((Files.isDirectory(nom)) ? nom+"/" : nom));

i++;

if(i%4 == 0)System.out.println("\n");

}

} catch (IOException e) {

e.printStackTrace();

}

}

On peut aussi filtrer pour n’afficher qu’il ne liste qu’un certain fichier.

try(DirectoryStream<Path> listing = Files.newDirectoryStream(chemin, "\*.txt")){ … }

//Ne prendra en compte que les fichier ayant l'extension .txt

Avec File :

**Copie de fichier :**

Path source = Paths.get("test.txt");

Path cible = Paths.get("test2.txt");

try {

Files.copy(source, cible, StandardCopyOption.REPLACE\_EXISTING);

} catch (IOException e) { e.printStackTrace(); }

Le troisième argument permet de spécifier les options de copie. Voici celles qui sont disponibles :

* StandardCopyOption.REPLACE\_EXISTING: remplace le fichier cible même s'il existe déjà ;
* StandardCopyOption.COPY\_ATTRIBUTES: copie les attributs du fichier source sur le fichier cible (droits en lecture etc.) ;
* StandardCopyOption.ATOMIC\_MOVE: copie atomique ;
* LinkOption.NOFOLLOW\_LINKS: ne prendra pas en compte les liens.

**Déplacement de fichier :**

Path source = Paths.get("test2.txt");

Path cible = Paths.get("test3.txt");

try {

Files.move(source, cible, StandardCopyOption.REPLACE\_EXISTING);

} catch (IOException e) { e.printStackTrace(); }

Dans le même genre vous avez aussi :

* une méthodeFiles.delete(path)qui supprime un fichier ;
* une méthodeFiles.createFile(path)qui permet de créer un fichier vide.

**Ouvrir des flux :**

:

Path source = Paths.get("test.txt");

//Ouverture en lecture :

try ( InputStream input = Files.newInputStream(source) ) { … }

//Ouverture en écriture :

try ( OutputStream output = Files.newOutputStream(source) ) { … }

//Ouverture d'un Reader en lecture :

try ( BufferedReader reader = Files.newBufferedReader(source, StandardCharsets.UTF\_8) ) { … }

//Ouverture d'un Writer en écriture :

try ( BufferedWriter writer = Files.newBufferedWriter(source, StandardCharsets.UTF\_8) ) { … }

Pour en savoir plus sur ce que permet la nouvelle classe java.nio.file.Files, je vous invite à regarder [la documentation Java](https://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html).

Java 7 vous permet également de gérer les fichier ZIP grâce à l'objet FileSystem:

// Création d'un système de fichiers en fonction d'un fichier ZIP

try (FileSystem zipFS = FileSystems.newFileSystem(Paths.get("monFichier.zip"), null)) {

//Suppression d'un fichier à l'intérieur du ZIP :

Files.deleteIfExists( zipFS.getPath("test.txt") );

//Création d'un fichier à l'intérieur du ZIP :

Path path = zipFS.getPath("nouveau.txt");

String message = "Hello World !!!";

Files.write(path, message.getBytes());

//Parcours des éléments à l'intérieur du ZIP :

try (DirectoryStream<Path> stream = Files.newDirectoryStream(zipFS.getPath("/"))) {

for (Path entry : stream) {

System.out.println(entry);

}

}

//Copie d'un fichier du disque vers l'archive ZIP :

Files.copy(Paths.get("fichierSurDisque.txt"), zipFS.getPath("fichierDansZIP.txt"));

}

Il est également possible d'être averti *via* l'objetWatchServicelorsqu'un un fichier est modifié, de gérer des entrées/sorties asynchrones via les objets AsynchronousFileChannel, AsynchronousSocketChannel ou AsynchronousServerSocketChannel. Ceci permet de faire les actions en tâche de fond, sans bloquer le code pendant l'exécution. Il est aussi possible d'avoir accès aux attributs grâce à 6 vues permettant de voir plus ou moins d'informations, à savoir :

* BasicFileAttributeViewpermet un accès aux propriétés généralement communes à tous les systèmes de fichiers ;
* DosFileAttributeViewajoute le support des attributs MS-DOS (readonly,hidden,system,archive) à l'objet ci-dessus ;
* PosixFileAttributeViewajoute les permissions POSIX du monde Unix au premier objet cité ;
* FileOwnerAttributeViewpermet de manipuler le propriétaire du fichier ;
* AclFileAttributeViewpermet de manipuler les droits d'accès au fichier ;
* UserDefinedFileAttributeView: permet de définir des attributs personnalisés.

## Le pattern decorator :

Les objets que l’on utilise utilisent des instances de même supertype :

DataInputStream dis = new DataInputStream(

new BufferedInputStream(

new FileInputStream(

new File("sdz.txt"))));

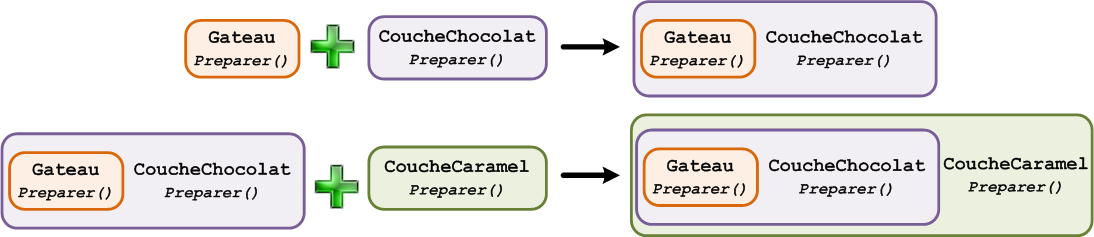
Avant de récupérer des données dans notre objet, elles vont d’abords transiter par des objets passés en paramètre et dans un certain ordre : *pattern decorator*.

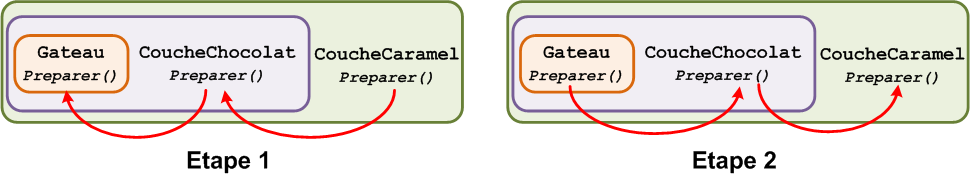
Nous allons procéder de la façon suivante :

* nous allons créer un objetGateau;
* nous allons lui ajouter uneCoucheChocolat;
* nous allons aussi lui ajouter uneCoucheCaramel;
* nous appellerons la méthode qui confectionnera notre gâteau.

Tout cela démarre avec un concept fondamental : l'objet de base et les objets qui le décorent *doivent* être du même type, et ce, toujours pour la même raison, le polymorphisme, le polymorphisme, et le polymorphisme !

En fait, les objets qui vont décorer notre gâteau posséderont la même méthode preparer()que notre objet principal, et nous allons faire fondre cet objet dans les autres. Cela signifie que nos objets qui vont servir de décorateurs comporteront une instance de type Patisserie ; ils vont englober les instances les unes après les autres et du coup, nous pourrons appeler la méthode preparer()de manière récursive !





On obtiendrait au final :

##### Patisserie.java

public abstract class Patisserie {

public abstract String preparer();

}

##### Gateau.java

public class Gateau extends Patisserie{

public String preparer() {

return "Je suis un gâteau et je suis constitué des éléments suivants. \n";

}

}

##### Couche.java

public abstract class Couche extends Patisserie{

protected Patisserie pat;

protected String nom;

public Couche(Patisserie p){

pat = p;

}

public String preparer() {

String str = pat.preparer();

return str + nom;

}

}

##### CoucheChocolat.java

public class CoucheChocolat extends Couche{

public CoucheChocolat(Patisserie p) {

super(p);

this.nom = "\t- Une couche de chocolat.\n";

}

}

##### CoucheCaramel.java

public class CoucheCaramel extends Couche{

public CoucheCaramel(Patisserie p) {

super(p);

this.nom = "\t- Une couche de caramel.\n";

}

}

##### CoucheBiscuit.java

public class CoucheBiscuit extends Couche {

public CoucheBiscuit(Patisserie p) {

super(p);

this.nom = "\t- Une couche de biscuit.\n";

}

}

Et voici un code de test ainsi que son résultat, représenté à la figure suivante.

public class Main{

public static void main(String[] args){

Patisserie pat = new CoucheChocolat(

new CoucheCaramel(

new CoucheBiscuit(

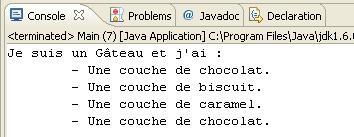
new CoucheChocolat(

new Gateau()))));

System.out.println(pat.preparer());

}

}



# Java et la réflexivité :

Aussi appelé *introspéction* consiste à découvrir de façon dynamique, des informations relatives à une classe ou un objet.

## Objet class :

Quand on crée une classe, la JVM créé un objet class pour chacune d’elle. On peut récupérer des infos sur cette classe.

public static void main(String[] args) {

Class c = String.class;

Class c2 = new String().getClass();

//La fameuse méthode finale dont je vous parlais dans le chapitre sur l'héritage

//Cette méthode vient de la classe Object

}

Maintenant qu’on a récupéré la class :

### Connaitre la superclass :

System.out.println("La superclasse de la classe " + String.class.getName() + " est : " + String.class.getSuperclass());

### Liste des interfaces d’une class :

public static void main(String[] args) {

//On récupère un objet Class

Class c = new String().getClass();

//Class c = String.class; est équivalent

//La méthode getInterfaces retourne un tableau de Class

Class[] faces = c.getInterfaces();

//Pour voir le nombre d'interfaces

System.out.println("Il y a " + faces.length + " interfaces implémentées");

//On parcourt le tableau d'interfaces

for(int i = 0; i < faces.length; i++)

System.out.println(faces[i]);

}

### Liste des méthodes de la class :

public static void main(String[] args) {

Class c = new String().getClass();

Method[] m = c.getMethods();

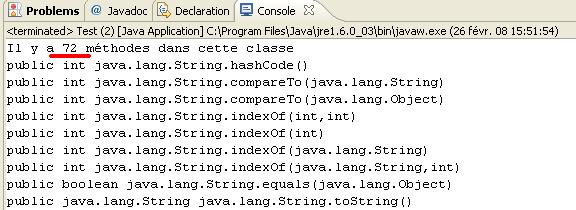
System.out.println("Il y a " + m.length + " méthodes dans cette classe");

//On parcourt le tableau de méthodes

for(int i = 0; i < m.length; i++)

System.out.println(m[i]);

}



Method() a aussi des méthodes intéressantes :

public static void main(String[] args) {

Class c = new String().getClass();

Method[] m = c.getMethods();

System.out.println("Il y a " + m.length + " méthodes dans cette classe");

//On parcourt le tableau de méthodes

for(int i = 0; i < m.length; i++)

{

System.out.println(m[i]);

Class[] p = m[i].getParameterTypes();

for(int j = 0; j < p.length; j++)

System.out.println(p[j].getName());

System.out.println("----------------------------------\n");

}

}

### Connaître liste des champs (variable de class ou d’instance) :

public static void main(String[] args) {

Class c = new String().getClass();

Field[] m = c.getDeclaredFields();

System.out.println("Il y a " + m.length + " champs dans cette classe");

//On parcourt le tableau de méthodes

for(int i = 0; i < m.length; i++)

System.out.println(m[i].getName());

}

Ce qui nous donne :

Il y a 7 champs dans cette classe

value

offset

count

hash

serialVersionUID

serialPersistentFields

CASE\_INSENSITIVE\_ORDER

### Liste des constructeurs de la class :

public static void main(String[] args) {

Class c = new String().getClass();

Constructor[] construc = c.getConstructors();

System.out.println("Il y a " + construc.length + " constructeurs dans cette classe");

//On parcourt le tableau des constructeurs

for(int i = 0; i < construc.length; i++){

System.out.println(construc[i].getName());

Class[] param = construc[i].getParameterTypes();

for(int j = 0; j < param.length; j++)

System.out.println(param[j]);

System.out.println("-----------------------------\n");

}

}

## Instancianciation dynamique :

Si on crée une nouvelle class :

public class Paire {

private String valeur1, valeur2;

public Paire(){

this.valeur1 = null;

this.valeur2 = null;

System.out.println("Instanciation !");

}

public Paire(String val1, String val2){

this.valeur1 = val1;

this.valeur2 = val2;

System.out.println("Instanciation avec des paramètres !");

}

public String toString(){

return "Je suis un objet qui a pour valeur : " + this.valeur1 + " - " + this.valeur2;

}

public String getValeur1() {

return valeur1;

}

public void setValeur1(String valeur1) {

this.valeur1 = valeur1;

}

public String getValeur2() {

return valeur2;

}

public void setValeur2(String valeur2) {

this.valeur2 = valeur2;

}

}

Le but du jeu est de créer un objet paire sans utiliser l’opérateur new :

public static void main(String[] args) {

String nom = Paire.class.getName();

try {

//On crée un objet Class

Class cl = Class.forName(nom);

//Nouvelle instance de la classe Paire

Object o = cl.newInstance();

//On crée les paramètres du constructeur

Class[] types = new Class[]{String.class, String.class};

//On récupère le constructeur avec les deux paramètres

Constructor ct = cl.getConstructor(types);

//On instancie l'objet avec le constructeur surchargé !

Object o2 = ct.newInstance(new String[]{"valeur 1 ", "valeur 2"} );

} catch (SecurityException e) {

e.printStackTrace();

} catch (IllegalArgumentException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

} catch (InstantiationException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

} catch (NoSuchMethodException e) {

e.printStackTrace();

} catch (InvocationTargetException e) {

e.printStackTrace();

}

}

Si on appelle le toString :

public static void main(String[] args) {

String nom = Paire.class.getName();

try {

//On crée un objet Class

Class cl = Class.forName(nom);

//Nouvelle instance de la classe Paire

Object o = cl.newInstance();

//On crée les paramètres du constructeur

Class[] types = new Class[]{String.class, String.class};

//On récupère le constructeur avec les deux paramètres

Constructor ct = cl.getConstructor(types);

//On instancie l'objet avec le constructeur surchargé !

Object o2 = ct.newInstance(new String[]{"valeur 1 ", "valeur 2"} );

//On va chercher la méthode toString, elle n'a aucun paramètre

Method m = cl.getMethod("toString", null);

//La méthode invoke exécute la méthode sur l'objet passé en paramètre

//Pas de paramètre, donc null en deuxième paramètre de la méthode invoke !

System.out.println("----------------------------------------");

System.out.println("Méthode " + m.getName() + " sur o2: " +m.invoke(o2, null));

System.out.println("Méthode " + m.getName() + " sur o: " +m.invoke(o, null));

} catch (SecurityException e) {

e.printStackTrace();

} catch (IllegalArgumentException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

} catch (InstantiationException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

} catch (NoSuchMethodException e) {

e.printStackTrace();

} catch (InvocationTargetException e) {

e.printStackTrace();

}

}

Avec ces méthodes, on peut par exemple stocker le nom d’une variable ou d’une classe en BDD.