Table des matières

1 Préambule

Java est un language orienté objet, il n'est pas possible d'y couper. Dans toute la suite, il sera supposé que vous connaissez déjà le principe de la Programmation Orienté Objet (POO).

En Java, l'exécution d'un programme va lancer la méthode **main()** de l'objet associé. Le code ci-dessous est le code minimum d'un programme Java qui défini la classe **sdz1**. Ce code ne fait rien.

```
1  public class sdz1 {
2
3    /**
4    * @param args
5    */
6
7    public static void main(String[] args) {
8    // TODO Auto-generated method stub
9
10   }
11 }
```

Le même code, affichant le sempiternel "Hello World" :

```
1  import java.util.Scanner;
2
3
4  public class sdz1 {
5
6    /**
7      * @param args
8      */
9    public static void main(String[] args) {
10     // TODO Auto-generated method stub
11
12     System.out.println("Hello World!");
13   }
14
15 }
```

Quelques informations de base :

- Le programme commence par le lancement de la méthode **.main()** de la classe du programme principal.
- Les lignes doivent se terminer par "**;**"
- Il faut déclarer les variables avant de les utiliser (voir [§ 2.2 on the facing page])
- Il faut compiler le programme avant de pouvoir l'utiliser (via une plateforme java, ce n'est pas un binaire mais un bytecode utilisable uniquement par un environnement Java)

2 Les bases

2.1 Commentaires

Il existe deux types de commentaires :

- les commentaires unilignes : introduits par les symboles //, ils mettent tout ce qui les suit en commentaires, du moment que le texte se trouve sur la même ligne que les //.

```
1 public static void main(String[] args){
2   //Un commentaire
3   //un autre
4   //Encore un autre
5   Ceci n'est pas un commentaire ! ! ! !
6 }
```

- les commentaires multilignes : ils sont introduits par les symboles /* et se terminent par les symboles */.

```
1 public static void main(String[] args){
2
3   /*
4   Un commentaire
5   Un autre
6   Encore un autre
7   */
```

```
8|   Ceci n'est pas un commentaire ! !
9| }
```

Il existe en fait un troisième type de commentaires : les *javadocs*. Ce sont des commentaires avec une mise en forme particulière (ils commencent par `/**` au lieu de `/*`) qui vont permettre de générer de la documentation de manière automatisée. En voici un exemple :

```
1|  /**
2|   * Retourne une chaine de caracteres selon le resultat de la comparaison
3|   * @param v1
4|   *             objet Ville
5|   * @return comparaison de deux ville
6|   */
```

Dans *Eclipse*, il vous suffit d'aller dans **Project** et de cliquer sur **Generate Javadoc**.

2.2 Types de variables

En Java, nous avons deux type de variables :
- des variables de type simple ou "primitif"
- des variables de type complexe : des **objets**. Le type *String* est un type complexe puisque c'est un objet, mais primitif puisque vous n'avez pas à en définir la classe correspondante pour qu'il existe.

2.2.1 Variables primitives

Ce qu'on appelle des types simples, ou types primitifs, en Java ce sont tout bonnement des nombres entiers, des nombres réels, des booléens ou encore des caractères.

Commençons par les variables de type numérique

- Le type *byte* (1 octet) peut contenir les entiers entre -128 et $+127$.

```
1| byte temperature;
2| temperature = 64;
```

- Le type *short* (2 octets) contient les entiers compris entre -32768 et $+32767$.

```
1| short vitesseMax;
2| vitesseMax = 32000;
```

- Le type *int* (4 octets) va de $-2 * 10^9$ à $2 * 10^9$ (2 et 9 zéros derrière...).

```
1| int temperatureSoleil;
2| temperatureSoleil = 15600000;
```

  C'est en kelvins...

- Le type *long* (8 octets) de $-9 * 10^{18}$ à $9 * 10^{18}$.

```
1| long anneeLumiere;
2| anneeLumiere = 9460700000000000;
```

- Le type *float* (4 octets) correspond à des nombres avec virgule flottante.

```
1| float pi;
2| pi = 3.141592653f;
```

  ou encore

```
1| float nombre;
2| nombre = 2.0f;
```

  Vous remarquerez que nous ne mettons pas de virgule mais un point ! Et vous remarquerez aussi que même si le nombre en question est rond, on met tout de même .0 derrière celui-ci !

- Le type *double* (8 octets) est identique à float, si ce n'est qu'il contient un nombre plus grand derrière la virgule.

```
1| double division;
2| division = 0.33333333333333333333333333333333333333333333334;
```

Le type *char* contient UN caractère stocké entre de simples quotes ' ' comme ceci...

```
1| char caractere;
2| caractere = 'A';
```

Le type *boolean* qui lui contient **true** (vrai) ou **false** (faux).

```
boolean question;
question = true;
```

2.2.2 Astuces

On peut très bien compacter la phase de déclaration et d'initialisation en une seule phase ! Comme ceci :

```
int entier = 32;
float pi = 3.1416f;
char carac = 'z';
String mot = new String("Coucou");
```

Et lorsque nous avons plusieurs variables d'un même type, nous pouvons compacter tout ceci en une déclaration comme suit :

```
int nbre1 = 2, nbre2 = 3, nbre3 = 0;
```

On peut aussi convertir une variable ou une formule dans un autre type de donnée via ce qu'on appelle un **cast** :

```
int i = 123;
double j = (double)i;
```

⚠ Il peut y avoir des pertes de précision au sein même des opérations mathématiques que la conversion du résultat via un cast n'empêchera pas. Ex :

```
int i = 123, j = 5;
double k = (double) (i / j);
```

2.2.3 Un type complexe particulier : la Chaîne de caractères

Le type *String* correspond à de la chaîne de caractères. Ici, il ne s'agit pas d'une variable mais d'un objet qui instancie une classe qui existe dans Java; nous pouvons l'initialiser en utilisant l'opérateur unaire *new()* dont on se sert pour réserver un emplacement mémoire à un objet, ou alors lui affecter directement la chaîne de caractères.

Vous verrez que celle-ci s'utilise très facilement et se déclare comme ceci :

```
String phrase;
phrase = "Titi et gros minet";
//Deuxieme methode de declaration de type String
String str = new String();
str = "Une autre chaine de caracteres";
//La troisieme
String string = "Une autre chaine";
//Et une quatrieme pour la route
String chaine = new String("Et une de plus ! ");
```

String étant un objet, il possède des méthodes afin de les manipuler. En voici quelques exemples :

- *toLowerCase()*
  Cette méthode permet de transformer toute saisie clavier de type caractère en minuscules. Elle n'a aucun effet sur les nombres, puisqu'ils ne sont pas assujettis à cette contrainte. Vous pouvez donc utiliser cette fonction sur une chaîne de caractères comportant des nombres. Elle s'utilise comme ceci :

  ```
  String chaine = new String("COUCOU TOUT LE MONDE !");
  String chaine2 = new String();
  chaine2 = chaine.toLowerCase();//donne "coucou tout le monde !"
  ```

- *toUpperCase()*
  Il s'agit de l'opposée de la précédente. Elle transforme donc une chaîne de caractères en majuscules. Et s'utilise comme suit :

  ```
  String chaine = new String("coucou coucou"), chaine2 = new String();
  chaine2 = chaine.toUpperCase();//donne "COUCOU COUCOU"
  ```

– *concat()*
Cette méthode permet de concaténer deux chaînes de caractères.

```
String str1 = new String("Coucou "), str2 = new String("toi !");
String str3 = new String();
str3 = str1.concat(str2);//donne "Coucou toi !"
```

– *length()*
Celle-là permet de donner la longueur d'une chaîne de caractères (en comptant les espaces blancs).

```
String chaine = new String("coucou ! ");
int longueur = 0;
longueur = chaine.length();//donne 9
```

– *equals()*
Permet de voir si deux chaînes de caractères sont identiques. Donc, de faire des tests. C'est avec cette fonction que vous ferez vos tests de conditions, lorsqu'il y aura des *String*. Exemple concret :

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

if (str1.equals(str2))//Si les deux chaines sont identiques
        System.out.println("Les deux chaines sont identiques !");

else
        System.out.println("Les deux chaines sont differentes !");
```

Vous pouvez aussi demander la non vérification de l'égalité grâce à l'opérateur de négation « ! », ce qui nous donne :

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

if (!str1.equals(str2))//Si les deux chaines sont differentes
        System.out.println("Les deux chaines sont differentes !");

else
        System.out.println("Les deux chaines sont identiques !");
```

Le principe de ce genre de condition fonctionne de la même façon pour les boucles. Et dans l'absolu, cette fonction retourne un booléen. C'est pourquoi nous pouvons utiliser cette fonction dans les tests de condition.

```
String str1 = new String("coucou"), str2 = new String("toutou");
boolean Bok = str1.equals(str2);//ici Bok prendra la valeur false
```

– *charAt()*
Le résultat de cette méthode sera un caractère, car il s'agit d'une méthode d'extraction de caractères, je dirais même d'UN caractère. Elle ne peut s'opérer que sur des *String* ! Elle possède la même particularité que les tableaux, c'est-à-dire que, pour cette méthode, le premier caractère sera le numéro 0. Cette méthode prend un entier comme argument.

```
String nbre = new String("1234567");
char carac = ' ';
carac = nbre.charAt(4);//renverra ici le caractere 5
```

– *substring()*
Comme son nom l'indique, elle permet d'extraire une sous-chaîne de caractères d'une chaîne de caractères. Cette méthode prend 2 entiers comme arguments. Le premier définit le début de la sous-chaîne à extraire inclus, le deuxième correspond au dernier caractère à extraire exclus. Et le premier caractère est aussi le numéro 0.

```
String chaine = new String("la paix niche"), chaine2 = new String();
chaine2 = chaine.substring(3,13);//permet d'extraire "paix niche"
```

– *indexOf()*/*lastIndexOf()*
indexOf() permet d'explorer une chaîne de caractères depuis son début. *lastIndexOf()* depuis sa fin, mais renvoie l'index depuis le début de la chaine. Elle prend un caractère, ou une chaîne de caractères comme argument, et renvoie un int. Tout comme *charAt()* et *substring()*, le premier caractère est à la place 0. Je crois qu'ici un exemple s'impose, plus encore que pour les autres fonctions :

```
String mot = new String("anticonstitutionnellement");
int n = 0;
```

```
3
4 n = mot.indexOf('t');       // n vaut 2
5 n = mot.lastIndexOf('t');         // n vaut 24
6 n = mot.indexOf("ti");     // n vaut 2
7 n = mot.lastIndexOf("ti");       // n vaut 12
8 n = mot.indexOf('x');       // n vaut -1
```

2.2.4 Les types complexes

Les types complexes, ou objets se déclarent un peu différemment des types primitifs. En effet, la déclaration est en fait une instanciation, c'est à dire qu'on crée un objet appartenant à une certaine classe d'objet :

```
1 Ville bdx = new Ville();
```

On définit ici la variable **bdx** comme étant un objet de la classe **Ville** à laquelle on affecte l'instanciation **new Ville()**. *new* signifie qu'on créer un nouvel objet de la classe, le constructeur étant appelé sans aucun argument (vu que les parenthèses n'en contiennent pas). Pour plus de détails sur ce point, voir [§ 3 on page 11].

2.3 Entrées clavier

Afin de récupérer ce qu'on tape au clavier, il faut importer une nouvelle classe

```
1 import java.util.Scanner;
```

Voici l'instruction pour permettre à Java de récupérer ce que vous avez saisi et ensuite de l'afficher :

```
1 Scanner sc = new Scanner(System.in);
2 System.out.println("Veuillez saisir un mot :");
3 String str = sc.nextLine();
4 System.out.println("Vous avez saisi : " + str);
```

Dans le cas où on récupère autre chose qu'une chaîne de caractère, il faut vider la ligne via un `sc.nextLine();` avant de chercher à récupérer une chaîne de caractère.

```
1 Scanner sc = new Scanner(System.in);
2
3 System.out.println("Saisissez un entier : ");
4 int i = sc.nextInt();
5
6 System.out.println("Saisissez une chaine : ");
7 //On vide la ligne avant d'en lire une autre
8 sc.nextLine();
9 String str = sc.nextLine();
10 System.out.println("FIN ! ");
```

2.4 Opérateur ternaire

La particularité des conditions ternaires réside dans le fait que trois opérandes (variable ou constante) sont mises en jeu mais aussi que ces conditions sont employées pour affecter des données dans une variable. Voici à quoi ressemble la structure de ce type de condition :

```
1 int x = 10, y = 20;
2 int max = (x < y) ? y : x ; //Maintenant max vaut 20
```

On peut faire par exemple :

```
1 int x = 10;
2 String type = (x % 2 == 0) ? "C' est pair" : "C' est impair" ;
3 //Ici type vaut "C' est pair"
4
5 x = 9;
6 type = (x % 2 == 0) ? "C' est pair" : "C' est impair" ;
7 //Ici type vaut "C' est impair"
```

2.5 Boucles

2.5.1 Boucle if

```
if(//condition)
  {
  // execution des instructions si la condition est remplie


  }
else
  {
  // execution des instructions si la condition n'est pas remplie


  }
```

Exemple :

```
int i = 10;

if (i < 0)
  System.out.println("Le nombre est negatif");

else
  System.out.println("Le nombre est positif");
```

Remarque : On n'est pas obligés de mettre les accolades quand il n'y a qu'une seule ligne d'instruction dans la boucle.

On peut aussi mettre des tests multiples :

```
int i = 0;

if (i < 0)
  System.out.println("Ce nombre est negatif !");

else if(i > 0)
  System.out.println("Ce nombre est positif !!");

else
  System.out.println("Ce nombre est nul !!");
```

Si on souhaite faire beaucoup de tests, on peut souhaiter utiliser la structure *switch* à la place.
Avec des chaînes de caractère

```
String entry = new String("t");

if (entry.equals("t"))
  System.out.println("le texte est 't'");
else
  System.out.println("le texte n'est pas 't'");
```

Voici un autre exemple pour utiliser par exemple les modulo :

```
int i = 25;

if (i%5==0)
  System.out.println("Ce nombre est multiple de 5");
else
  System.out.println("Ce nombre n'est pas multiple de 5");
```

2.5.2 Boucle Switch

```
int nbre = 5;

switch (nbre)
{
  case 1:
    System.out.println("Ce nombre est tout petit");
    break;
```

```
8
9     case 2:
10      System.out.println("Ce nombre est tout petit");
11      break;
12
13    case 3:
14      System.out.println("Ce nombre est un peu plus grand");
15      break;
16
17    case 4:
18      System.out.println("Ce nombre est un peu plus grand");
19      break;
20
21    case 5:
22      System.out.println("Ce nombre est la moyenne");
23      break;
24
25    case 6:
26      System.out.println("Ce nombre est tout de meme grand");
27      break;
28
29    case 7:
30      System.out.println("Ce nombre est grand");
31      break;
32
33    default:
34      System.out.println("Ce nombre est compris entre 8 et 10");
35
36  }
```

2.5.3 Boucle while

```
int a = 1, b = 15;
while (a < b)
{
        System.out.println("coucou " +a+ " fois !!");
        a++;
}
```

On peut aussi faire :

```
//Une variable vide
String prenom;
// On initialise celle-ci a 0 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 reponse donnee est egale a oui
while (reponse == 'O')
{
  //On affiche une instruction
  System.out.println("Donnez un prenom : ");
  //On recupere le prenom saisi
  prenom = sc.nextLine();
  // On affiche notre phrase avec le prenom
  System.out.println("Bonjour " +prenom+ " comment vas-tu ?");
  //On demande si la personne veut faire un autre essai
  System.out.println("Voulez-vous reessayer ?(O/N)");
  //On recupere la reponse de l'utilisateur
  reponse = sc.nextLine().charAt(0);
}

System.out.println("Au revoir...");
//Fin de la boucle
```

2.5.4 Boucle do while

```
do{
  blablablablablablablabla
}while(a < b);
```

2.5.5 Boucle for

```
for(int i = 1; i <= 10; i++)
{
  System.out.println("Voici la ligne "+i);
}
```

On peut aussi boucler sur les éléments d'un tableau :

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

for(String str : tab)
   System.out.println(str);
```

Cette forme de boucle **for** est particulièrement adaptée au parcours de tableau. On peut naturellement se demander comment faire de même pour des tableaux multidimensionnels. La chose à retenir est que la variable en premier paramètre de la boucle **for** doit être du même type que la valeur de retour du tableau. Dans le cas d'un tableau multi-dimensionnel, cette dernière sera un tableau de dimension inférieure. En conséquence, on peut boucler sur des sous tableaux, puis sur les éléments de ces derniers via des boucles imbriquées :

```
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 a l'indice ["
      + j + "][" +i +"] est : " +  tab[j][i] + "\n");
    i++;
  }
  j++;
}
```

2.6 Tableaux

On définit des tableaux de la même manière que les éléments qui le constituent. Un tableau a donc un type associé et ne peut stocker que des éléments de ce type là.

Pour définir un tableau sans l'initialiser on fait :

```
int tableauEntier[] = new int[6];
//ou encore
int[] tableauEntier2 = new int[6];
```

mais la définition d'un tableau initialisé se fait elle de la façon suivante :

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

On peut définir des tableaux multi-dimensionnels :

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

Nous voyons bien ici les deux lignes de notre tableau symbolisées par les doubles crochets `[][]`. Ce genre de tableau n'est rien d'autre que plusieurs tableaux en un. Ainsi, pour passer d'une ligne à l'autre, nous jouerons avec la valeur du premier crochet.

Exemple : `premiersNombres[0][0]` correspondra au premier élément de la colonne paire.
Et `premiersNombres[1][0]` correspondra au premier élément de la colonne impaire.

Remarque : La longueur d'un tableau **tab** est donnée par :

```
tab.length
```

On peut définir des sortes de tableaux pouvant contenir n'importe quel type de données, ce sont des objets de type *ArrayList* qu'on appelle des *collection d'objet*.

2.6.1 ArrayList

ArrayList permet de stocker un nombre inconnu d'objets. C'est à dire que contrairement à un tableau classique, on ne sait pas *a priori* combien d'élément on a besoin de stocker.

On peut définir un tableau de la façon suivante :

```
ArrayList liste = new ArrayList();
liste.add('d');
liste.add("une chaine");
liste.add(3);
liste.add(3.14f);
```

Il y a en particulier les méthodes suivantes, disponible pour les objets **ArrayList** :
- **add()** : permet d'ajouter un élément.
- **get(int index)** : retourne l'élément à l'index 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'objet.

Ça permet par exemple de remplir une liste à l'aide de l'utilisateur.

```
ArrayList<Player> players = new ArrayList(); // The list of players we want to define
String name = new String();
//To get info via stdin
Scanner sc = new Scanner(System.in);
//On affiche une instruction
System.out.println("Please enter the names of players (one line per name, leave a
    blank line to end the process) :");
while(true) {
  //On recupere le prenom saisi
  name = sc.nextLine();

  if (name.length()!=0) {
    // We add a new player in the list
    players.add(new Player(name));
  }
  else
    break;
```

2.6.2 HashMap

Dans le même style, une **HashMap** est un peu l'équivalent d'un dictionnaire en python. C'est à dire que c'est une liste non ordonnée où on stocke des couples (clé, valeur), ce qui permet de référencer des données via d'autres types de données.

Un exemple rapide :

```
HashMap dict = new HashMap();
dict.put(1, "printemps");
dict.put(10, "ete");
dict.put(12, "automne");
dict.put(45, "hiver");
```

Il y a en particulier les méthodes suivantes, disponible pour les objets **HashMap** :
- **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.
- **put(Object key, Object value)** : ajoute le couple key/value dans l'objet.
- **elements()** : retoune une énumération des éléments de l'objet.
- **keys()** : retourne la liste des clés (peut servir dans une boucle)
- **values()** : retourne la liste des valeurs (peut servir dans une boucle)

Remarque : On peut contraindre le type de données pour la clé ou la valeur lors de la déclaration de l'objet **HashMap** :

```
HashMap<String, int> dict = new HashMap();
```

Ici, les clés devront être des chaînes de caractère, et les valeurs seront des entiers.

Un exemple d'utilisation (en reprenant l'exemple pour les **ArrayList**) :

```
// We start by giving the name of the players
HashMap<String, Player> players = new HashMap<String, Player>();
String name = new String();

Scanner sc = new Scanner(System.in);
//On affiche une instruction
System.out.println("Please enter the names of players (one line per name, leave a
    blank line to end the process) :");
while(true) {
  //On recupere le prenom saisi
  name = sc.nextLine();

  if (name.length()!=0) {
    // We add a new player in the list
    players.put(name, new Player(name));
  }
  else
    break;
}
```

3 Les classes

Toute classe possède un constructeur (une méthode particulière, lancée lors de l'initialisation d'une instance de classe), ayant le même nom que la classe elle-même.

Prenons un exemple avec une classe **Ville** :

```
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 nbreHabitant;

  /**
   * Constructeur par defaut
   */
  public Ville(){
    System.out.println("Creation d'une ville !");
    nomVille = "Inconnu";
    nomPays = "Inconnu";
    nbreHabitant = 0;
  }

}
```

Remarque : Il est possible de surcharger [§ 5.1 on page 17] le constructeur et ainsi avoir plusieurs constructeurs en fonction des paramètres passés lors de l'initialisation de l'instance de classe.

La création d'une instance de classe se fait via le mot clé **new** :

```
Ville bordeaux = new Ville();
```

3.1 Bien gérer les variables

Dans l'exemple précédent, les *variables d'instance* sont publiques et modifiables directement. La philosophie de la POO est d'utiliser des *accesseurs* getVar (qui renvoient la valeur d'une variable var) et des *mutateurs* setVar (qui modifient la valeur de la variable var). En conséquence, on mettra plutôt les variables avec une portée privée, et on créera des méthodes permettant d'afficher ou modifier ces dernières. La classe devient alors :

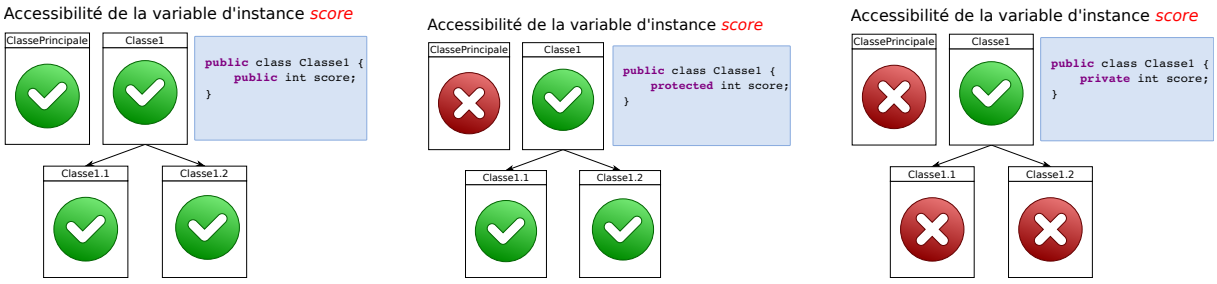
```java
public class Ville {
  /**
  * Stocke le nom de notre ville
  */
  private String nomVille;
  /**
  * Stocke le nom du pays de notre ville
  */
  private String nomPays;
  /**
  * Stocke le nombre d'habitants de notre ville
  */
  private int nbreHabitant;

  /**
  * Constructeur par defaut
  */
  public Ville(){
    System.out.println("Creation d'une ville !");
    nomVille = "Inconnu";
    nomPays = "Inconnu";
    nbreHabitant = 0;
  }

  /**
  * Constructeur d'initialisation
  * @param pNom
  *                       Nom de la Ville
  *  @param pNbre
  *                       Nombre d'habitants
  *  @param pPays
  *                       Nom du pays
  */
  public Ville(String pNom, int pNbre, String pPays)
  {
    System.out.println("Creation d'une ville avec des parametres !");
    nomVille = pNom;
    nomPays = pPays;
    nbreHabitant = pNbre;
  }

 //*****************************************************************************
  //                                        ACCESSEURS
  //*****************************************************************************

  /**
   * Retourne le nom de la ville
   * @return le nom de la ville
   */
  public String getNom()
  {
    return nomVille;
  }

  /**
   * Retourne le nom du pays
   * @return le nom du pays
   */
  public String getNomPays()
  {
    return nomPays;
  }

  /**
   * Retourne le nombre d'habitants
   * @return nombre d'habitants
   */
 public int getNombreHabitant()
 {
    return nbreHabitant;
 }

 //*****************************************************************************
```

```java
74  //                                          MUTATEURS
75  //*************************************************************************
76
77  /**
78   * Definit le nom de la ville
79   * @param pNom
80   *              nom de la ville
81   */
82  public void setNom(String pNom)
83  {
84     nomVille = pNom;
85  }
86
87  /**
88   * Definit le nom du pays
89   * @param pPays
90   *              nom du pays
91   */
92  public void setNomPays(String pPays)
93  {
94     nomPays = pPays;
95  }
96
97  /**
98   * Definit le nombre d'habitants
99   * @param nbre
100  *              nombre d'habitants
101  */
102  public void setNombreHabitant(int nbre)
103  {
104    nbreHabitant = nbre;
105  }
106
107 }
```

3.2 Portée des variables

Lors de la déclaration de variables ou de méthodes, il y a plusieurs attributs possible, dont le type bien entendu (mais que je ne détaillerai pas ici).

L'un des attributs est la **portée** de la variable ou la méthode :
- *public* : La variable/méthode est accessible par tout le monde
- *private* : La variable/méthode n'est accessible qu'à l'intérieur de la classe où elle est définie
- *protected* : La variable/méthode est accessible à l'intérieur de la classe où elle est définie ainsi que toutes ses classes dérivées (*héritage*)



(a) Variable publique : accès pour tout le monde

(b) Variable protégée : accès réservé à la classe et ses classes dérivées

(c) Variable privée : accessible uniquement à l'intérieur de la classe.

FIGURE 1 – Illustration de la portée des variables dans une classe

3.3 Variable de classe

Tout d'abord les *variables de classes*, qui sont définies dans la classe, avec l'attribut *static*. Cette variable sera commune à toutes les instances de la classe. On déclare une variable de classe de la façon suivante :

```
public class Ville {

   /**
    * Variable de classe
    */
   static String nomVille;
```

Pour accéder à une *variable de classe* **var** (de la classe **Ville** par exemple) à l'intérieur même de celle-ci (ou une de ses instances), on y accède via :

```
Ville.var
```

3.4 Méthode de classe

Toutes les **méthodes** d'une **classe** n'utilisant que des *variables de classes* doivent être déclarées *static* ! On les appelle des **méthodes de classes** car elles sont globales à toutes vos instances !

Remarque : Par contre ceci n'est plus vrai si une méthode utilise des variables d'instances et des variables de classes...

L'*accesseur* d'une *variable de classe* doit aussi être déclarée *static*, c'est une règle !

3.5 Héritage

L'*héritage* permet de créer des classes filles d'une classe donnée. L'intérêt de l'héritage est que toutes les méthodes et variables créées pour la classe parente se retrouveront dans les classes filles.

```
class Capitale extends Ville {

private String president;

 /**
  *Constructeur par defaut
  */
  public Capitale(){
    //Ce mot cle appelle le constructeur de la classe mere.
    super();
    president = "aucun";
  }
}
```

Pour plus de détails sur *super()* voir [§ 5.1 on page 17]. [§ 6.1 on page 18] détaille comment empêcher l'héritage.

3.6 Généricité

La généricité, c'est pouvoir créer une classe qui puisse travaille avec n'importe quel type d'objet sans avoir à créer une classe différente à chaque fois. La particularité, c'est qu'une fois instancié, le type de variable utilisable par l'objet ne peut pas changer (chaque instance de la classe peut avoir un type de variable différent, mais une fois définit, on ne peut pas le changer).

L'astuce est ici de définir une variable `T` pour désigner le type avec lequel travaille la classe. Exemple :

```
public class Solo<T> {

   /**
     * Variable d'instance
     */
   private T valeur;

   /**
     * Constructeur par defaut
     */
   public Solo(){
     this.valeur = null;
   }

   /**
```

```
16      * Constructeur avec parametre
17      * Inconnu pour l'instant
18      * @param val
19      */
20    public Solo(T val){
21      this.valeur = val;
22    }
23
24
25    /**
26      * Definit la valeur avec le parametre
27      * @param val
28      */
29    public void setValeur(T val){
30      this.valeur = val;
31    }
32
33    /**
34      * retourne la valeur deja "castee" par la signature de la methode !
35      * @return
36      */
37    public T getValeur(){
38      return this.valeur;
39    }
40  }
```

Dans cette classe, le `T` n'est pas encore défini. Vous le ferez à l'instanciation de cette classe. Par contre, une fois instancié avec un type, l'objet ne pourra travailler qu'avec le type de données que vous lui avez spécifié !

Exemple :

```
1  public class Test {
2
3    public static void main(String[] args) {
4      // TODO Auto-generated method stub
5      Solo<Integer> val = new Solo<Integer>(12);
6      int nbre = val.getValeur();
7      Solo<String> valS = new Solo<String>("TOTOTOTO");
8      Solo<Float> valF = new Solo<Float>(12.2f);
9      Solo<Double> valD = new Solo<Double>(12.202568);
10 }
11 }
```

On n'est pas limité à une seule variable. On peut définir par ce même principe, plusieurs variables génériques dans une classe donnée :

```
1  public class Duo<T, S> {
2
3    /**
4      * Variable d'instance de type T
5      */
6    private T valeur1;
7    /**
8      * Variable d'instance de type S
9      */
10   private S valeur2;
11 }
```

que l'on utilise de la façon suivante :

```
1  Duo<String, Boolean> dual = new Duo<String, Boolean>("toto", true);
2  System.out.println("Valeur de l'objet dual: val1 = " + dual.getValeur1() + ", val2 =
       " + dual.getValeur2());
3
4  Duo<Double, Character> dual2 = new Duo<Double, Character>(12.25895, 'C');
5  System.out.println("Valeur de l'objet dual2: val1 = " + dual2.getValeur1() + ", val2
       = " + dual2.getValeur2());
```

On peut contourner l'interdiction d'utiliser un type différent en utilisant le point d'interrogation « ? » lors de la déclaration du type des variables. Ça permet, pour une même variable, d'accepter l'écrasement

de l'objet existant par un nouvel objet de type différent (mais l'objet défini n'acceptera toujours qu'un seul type de variable.

Exemple :

```
1 Duo<?, ?> dual = new Duo<String, Boolean>("toto", true);
2
3 System.out.println("Valeur de l'objet dual: val1 = " + dual.getValeur1() + ", val2 =
      " + dual.getValeur2());
4 dual = new Duo<Double, Character>();
5 dual = new Duo<Integer, Float>();
6 dual = new Duo<Solo, Solo>();
```

4 Les instances de classe

4.1 Variable d'instance

Il y a ensuite les *variables d'instance*, qui sont définies dans la classe. Cette variable **var** est accessible, à l'intérieur de la classe via **this.var** où *this* désigne l'instance de classe. On défini une variable d'instance dans l'entête de la classe, de la façon suivante :

```
1 public class Ville {
2
3   /**
4    * Variable d'instance
5    */
6   String nomVille;
```

Pour accéder à une variable d'instance à l'intérieur d'une instance, on procède de la façon suivante :

```
1 this.var
```

Pour y accéder depuis l'extérieur, il est conseillé de configurer des accesseurs « getVariable() », voir [§ 3.1 on page 11] pour plus de détails. On peut créer des variables privées (notamment pour faire des variables locales) en rajoutant un attribu privé à une variable. Il est recommandé de définir ces variables avec l'attribut *private* et de créer des *accesseurs* et des *mutateurs* afin de voir le contenu de la variable et de le modifier.

4.2 Méthodes d'instance

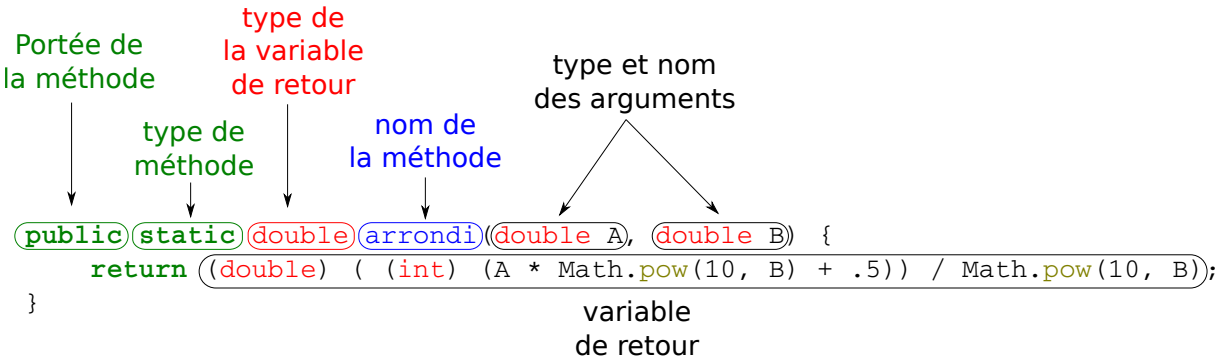


FIGURE 2 – Explication des différents attributs d'une méthode. Le but ici est de présenter à quoi correspond la syntaxe, sans présenter les différents choix possibles.

Les méthodes ne sont pas limitées en nombre de paramètres (s'il n'y a pas d'argument, il faut au minimum « `String[] args` ».

Il existe trois grands types de méthodes :

- celles qui ne renvoient rien. Elles sont de type *void*. Ces types de méthodes n'ont pas d'instruction **return** !
- celles qui retournent des types primitifs (*double*, *int*...). Elles sont de type *double*, *int*, *char*... Celles-ci ont une instruction **return**.

– celles qui retournent des objets. Par exemple, une méthode qui retourne un objet de type *String*. Celles-ci aussi ont une instruction **return**.

Remarque : On n'imbrique pas les méthodes et elles doivent toutes faire partie d'une classe.

⚠ Les méthodes de la classe **main**, c'est à dire la classe lancée par défaut au début du programme et qui contient le *main()*, doivent être *static*

4.3 Accéder aux méthodes et variables d'un objet

En règle générale, on appelle des méthodes ou des paramètres grâce à notre variable qui nous sert de référence, celle-ci suivie de l'opérateur ".", puis du nom de la dite méthode/variable.

```
String str = new String("opizrgpinbzegip");
str = str.subString(0,4);
```

Mais pour accéder aux données ou méthode de notre objet à l'intérieur même de celui-ci, cette méthode n'est pas possible. Pour y remédier, il faut utiliser le mot-clé *this* :

```
public class Ville {

  /**
  * Variable d'instance
  */
  String nomVille;

  public Ville(String nom) {
    this.nomVille = "nom";
  }
}
```

où **this** désigne l'instance courante de l'objet (qui n'existe pas au moment de l'écriture du programme, mais qui existera quand le mot clé **this** aura besoin de pointer vers quelque chose.

5 Les méthodes

Pour des infos sur les méthodes de classes ([§ 3.4 on page 14]) et les méthodes d'instance ([§ 4.2 on the preceding page]), se référer aux sections correspondantes.

5.1 La surcharge de méthode

La surcharge de méthode consiste à garder un nom de méthode (donc un type de traitement à faire, pour nous, lister un tableau) et de changer la liste ou le type de ses paramètres.

Nous allons surcharger notre méthode afin qu'elle puisse travailler avec des **int** par exemple :

```
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);
  }
```

On peut aussi faire de même avec les tableaux à 2 dimensions ou ajouter des paramètres à la méthode.

Le mot-clé *super()* permet de faire appel à la version parente d'une méthode donnée que l'on souhaite surcharger. Ça signifie concrètement que *super()* va exécuter la version parente de la méthode en lieu et place du mot-clé. Ceci fonctionne bien entendu pour le constructeur, ce qui permet d'initialiser la classe parente de manière normale, tout en rajoutant l'initialisation des variables propre à la classe courante :

```
class Capitale extends Ville {

private String president;

 /**
  *Constructeur par defaut
  */
  public Capitale(){
    //Ce mot cle appelle le constructeur de la classe mere.
    super();
    president = "aucun";
  }
}
```

Remarque : Dans le cas d'une méthode qui nécessite une variable, imaginez que *super()* se comporte comme un alias. Il suffit de mettre en paramètre de *super()* les valeurs que vous souhaitez :

```
super(nom, hab, pays);
```

Pour aller plus loin, voir [§ 6.1] pour apprendre comment empêcher la surcharge et l'héritage

5.2 Polymorphisme

Le *polymorphisme*, c'est le fait qu'une même méthode soit implémentée dans plusieurs classes, autorisant cette méthode à être utilisée indifféremment sur plusieurs types d'objets.

 Il ne faut pas confondre le *polymorphisme*, c'est à dire le fait de définir une même méthode avec un squelette identique (même nombre et type d'arguments) dans plusieurs classes avec la surcharge de méthodes, qui permet au sein d'une même classe de définir la même méthode plusieurs fois mais avec un nombre ou type d'argument différents.

Le polymorphisme permet d'agir sur la classe mère, dans laquelle les méthodes sont définies, et agir indifféremment sur des objets de sous-classes ayant redéfini les méthodes de la classe mère.

Remarque : L'utilisation d'*interfaces* permet aussi d'utiliser le polymorphisme dans des classes non issues de la même classe mère (voir [§ 6.3 on the next page]).

5.3 Méthode abstraite

Le mot-clé *abstract* permet de créer une classe ou une méthode abstraite. Une méthode abstraite n'a pas de corps.

```
abstract class Animal{
   abstract void manger();//une methode abstraite
}
```

Remarque : Une classe déclarée abstraite n'est plus instanciable, mais elle n'est nullement obligée d'avoir des méthodes abstraites ! En revanche, une classe ayant une méthode abstraite doit être déclarée abstraite !

6 Les classes : avancé

6.1 Empêcher la surcharge de méthode ou l'héritage

En déclarant le mot-clé *final* comme attribut d'une méthode, on empêche la surchage de méthode. Ces méthodes sont figées et vous ne pourrez *jamais* redéfinir une méthode déclarée *final*. Un exemple de ce type de méthode est la méthode **getClass()** de la classe **Object**.

Remarque : Il existe aussi des classes déclarées *final*. Vous avez compris que ces classes sont immuables... Et vous ne pouvez donc pas faire hériter un objet d'une classe déclarée *final*.

6.2 Super-classes non instantiables : Classe abstraite

Le mot-clé *abstract* permet de créer une classe ou une méthode abstraite. Ça signifie concrêtement que la classe ne peut pas être instanciée. Elle n'est utile que pour définir des sous classes, ayant des méthodes ou attributs communs qu'on peut définir dans cette super-classe qui n'est pas utilisable en l'état.

Exemple : Imaginez une méta-classe abstraite **Animal** qui servirait à définir des sous classes **Chien**, **Chat**, **Lion**, **Ours** etc...

```
1 abstract class Animal{
2
3 }
```

Une telle classe peut avoir le même contenu qu'une classe normale. Ses enfants pourront utiliser tous ses éléments déclarés (attributs et méthodes) public. Cependant, ce type de classe permet de définir des méthodes abstraites, voir [§ 5.3 on the facing page].

Remarque : Les classes abstraites sont à utiliser lorsque qu'une classe mère ne doit pas être instanciée.

6.3 Compilations de définitions de méthodes : Interfaces

Une *interface* permet de définir le squelette (le nom de la méthode, le nombre et le type des arguments) de différentes méthodes généralistes. Ensuite, un peu comme un module, on appelle cette interface dans la classe voulue. On doit alors implémenter *toutes* les méthodes définies dans l'interface.

On définit une interface **I** de la façon suivante :

```
1 public interface I{
2
3    public void A();
4    public String B();
5
6 }
```

On appelle une *interface* **I** dans la classe **X**

```
1 public class X implements I{
2     public void A(){
3       //.......
4     }
5
6     public String B(){
7       //.......
8     }
9 }
```

6.4 Conversion d'objets

Il est possible, en particulier pour l'utilisation de certaines méthodes, de convertir des objets. Imaginons qu'on ait une classe **Ville**, définissant la méthode **.decrisToi()**, le code suivant est possible :

```
1 //Def d'un tableau de ville null
2 Ville[] tableau = new Ville[6];
3
4 /*
5 On initialise les objets dans cette partie non ecrite
6 */
7
8 //il ne nous reste plus qu'a decrire tout notre tableau !
9 for(Object v : tableau){
10   System.out.println(((Ville)v).decrisToi()+"\n");
11 }
```

On a ainsi converti la variable **v** afin de pouvoir utiliser la méthode **decrisToi()** qui n'existe pas dans la classe **Object**. Ceci fonctionne sans doute parce que la classe Object est la classe parente de toute classe par défaut, c'est à dire que toute classe :

```
1 class Ville {
2
3 }
```

est en fait équivalente à :

```
class Ville extends Object {

}
```

Pour plus de détails sur *extends*, voir [§ 3.5 on page 14].

6.5 Covariance des variables

Il existe ce qu'on appelle la *covariance des variables*, c'est à dire qu'il est possible d'utiliser une variable, déclarée comme appartenant à une classe **ClasseParente**, pour stocker une instance de n'importe quel objet d'une classe héritée **ClasseFille**. Un exemple :

```
//Def d'un tableau de ville null
Ville[] tableau = new Ville[6];

//Definition d'un tableau de noms de Villes et d'un tableau de nombres d'habitants
String[] tab = {"Marseille", "lille", "caen", "lyon", "paris", "nantes"};
int[] tab2 = {123456, 78456, 654987, 75832165, 1594,213};

/* Les 3 premiers elements 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", "Sarko");
    tableau[i] = C;
  }
}

//il ne nous reste plus qu'a decrire tout notre tableau !
for(Ville v : tableau){
  System.out.println(v.decrisToi()+"\n");
}
```

7 Avancé

7.1 Lire/écrire des objets dans un fichier : sérialisation

7.1.1 Écrire des objets dans un fichier

```
FileOutputStream fos = new FileOutputStream("task_list.task"); // Lie un flux de
    donnees a un fichier physique
ObjectOutputStream oos = new ObjectOutputStream(fos); // permet de convertir un objet
     en flux de donnees
for (Task task : taskList) {
  oos.writeObject(task);
}
oos.close();
```

Afin de pouvoir être enregistré dans un fichier, l'objet doit être déclaré de la manière suivante :

```
public class Task implements Serializable{

  /**
    * To follow, in the saving of tasks, the eventual modification of the class
        itself
    */
  private static final long serialVersionUID = 1L;
}
```

Cette interface **Serializable** est dite de type marqueur, c'est à dire qu'elle n'a aucune méthode à implémenter. Elle permet juste d'indiquer que cette classe est sérialisable, et donc enregistrable dans un fichier par la méthode ci-dessus.

En résumé :

1. Il faut créer un FileOutputStrem. C'est objet permet de se connecter à un fichier (voire en créer un).
2. Il faut ensuite créer un ObjectOutputStrem. Avec pour paramètre un FileOutputStream, il peut écrire des objets dans un fichier.
3. Ce dernier objet est ensuite celui avec qui on interagit pour écrire des objets, le reste s'opère en interne.

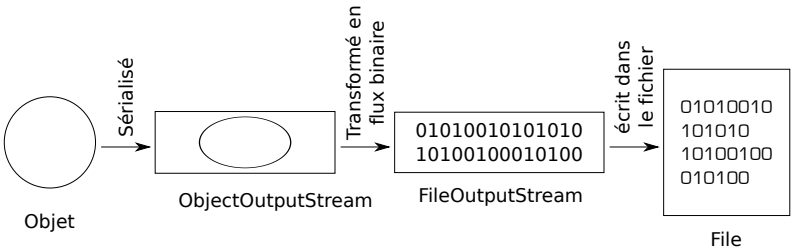


FIGURE 3 – Les différentes étapes de l'enregistrement d'un objet dans un fichier.

7.1.2 Lire des objets à partir d'un fichier

```
1  ArrayList<Task> taskList = new ArrayList<Task>();
2  FileInputStream fis = new FileInputStream("task_list.task");
3  ObjectInputStream ois = new ObjectInputStream(fis);
4  try {
5    Task obj = null;
6    while ((obj = (Task)ois.readObject()) != null)
7      taskList.add(obj);
8  } catch (EOFException ex) { //This exception will be caught when EOF is reached
9  System.out.println("List of tasks read successfully");
10 } finally {
11   ois.close();
12 }
```

Même principe que pour l'écriture, la lecture du fichier pour récupérer les objets.

8 Eclipse

Il est impossible de faire un tutoriel Java sans s'attarder sur Eclipse, excellent logiciel pour coder en Java et qui facilite vraiment la vie des développeurs.

8.1 Générer automatiquement les accesseurs et les mutateurs avec Eclipse

Une fois qu'on a créé une classe avec beaucoup de variables d'instances ou de classe, *eclipse* permet de générer automatiquement les *accesseurs* et les *mutateurs*.

Il faut pour cela aller dans le menu **Source**, puis l'option générale **Generate Getters and Setters**. *Eclipse* propose alors la liste des variables présentes dans la classe courante.

8.2 Récupérer un projet déjà existant sur un SVN distant

Il faut installer le plugin *subclipse* au préalable.

Il faut pour celà créer un nouveau "Projet" : File>New>Other>Checkout projet from SVN (dans le dossier SVN).

Suivre les instructions et renseigner l'adresse du dépot SVN (sans mettre le nom du sous-dossier où se trouve le projet). Dans l'onglet suivant, vous pourrez spécifiez le nom du dossier qui vous intéresse.

8.3 Synchroniser un projet en le sauvegardant sur un SVN distant

En prenant un projet déjà existant localement (mais qui n'existe pas sur le SVN distant). Il faut faire clic droit sur le nom du projet. Puis **Team** et enfin **Share Project...**

Index