Chapitre 2

Compilation, exécution, machine virtuelle

JAVA dispose d'outils pour générer et exécuter du code. En particulier, le JRE (JAVA Runtime environment) propose entre autres des bibliothèques de classes et une machine virtuelle dont nous parlerons plus bas. Le JDK (JAVA Development Kit) contient le JRE et contient en plus le compilateur et des outils pour débugger.

Ce cours traite de la programmation en JAVA . Pour pouvoir exécuter le code en JAVA , il faut tout d'abord transformer le code écrit par le programmeur, qu'on appelle le code source, en un code qui est utilisable par la machine. JAVA utilise une stratégie de machine virtuelle : le code source est traduit non dans un langage directement utilisable par la machine mais dans un language spécial appelé `bytecode`. Pour l'exécution, on utilise une machine virtuelle JAVA qui dépend de la machine d'exécution et qui va interpréter le code en `bytecode` dans le langage de la machine. Nous allons parler très succinctement de ces deux phases résumées dans le diagramme ci-dessous.

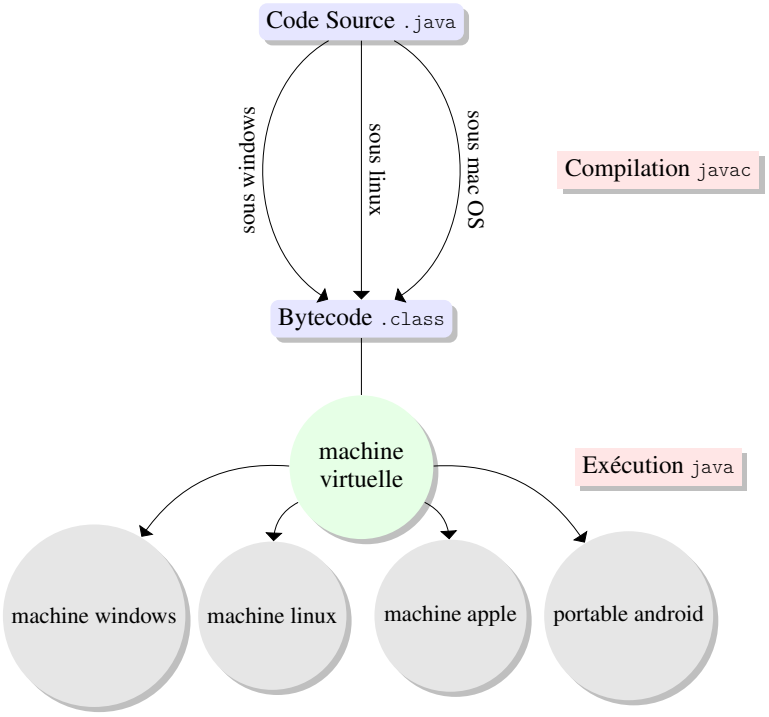

FIGURE 2.1 – Du code source à l'utilisation d'une application Java

1 Compilation

Pour créer un programme, un développeur écrit un ensemble de classes. Chaque classe `<MaClasse>` est enregistrée dans un fichier `<MaClasse>.java` : il porte le même nom que la classe et possède l'extension `.java`.

Ensuite, le développeur doit *compiler* l'ensemble de classes à l'aide d'un programme appelé `javac`. Le but du compilateur est de traduire le code écrit par le développeur en un autre langage qui pourra être exécuté par une machine. Dans le cas de JAVA , un compilateur produit un code dans le langage `bytecode`. Le résultat de la compilation d'une classe `<MaClasse>` sera un fichier nommé `<MaClasse>.class`, i.e., il porte aussi le nom de la classe, mais son extension est `.class`.

Pour simplifier, il y a deux étapes lors de la compilation :

— *analyse syntaxique* : le code est lu, on construit une réprésentation du code à l'aide d'un arbre syntaxique. C'est dans cette phase qu'on vérifie la syntaxe de votre code, i.e. on vérifie la grammaire du code JAVA (pour cela, il suffit d'étudier l'arbre syntaxique créé). Le cours « Automates, Langages et application » donne les outils pour faire cette vérification.

— *analyse sémantique* : l'arbre syntaxique est analysé et traduit en `bytecode`. Pendant l'analyse, les références à des classes extérieures sont vérifiées (on cherche si la classe existe bien, si elle a besoin d'être compilée, etc). Le langage `bycode` est un langage plus simple (plus bas niveau).

2 Exécution

Une fois que les fichiers en `bytecode` sont écrits, on peut exécuter le programme. Dans certains langages de programmation comme le `C` ou le `C++` la compilation génère un fichier qui sera directement exécutable par la machine : le code machine « natif » dépendra donc du système d'exploitation et de l'architecture de la machine (processeur). JAVA utilise une stratégie différente : le `bytecode` est un code qui peut être exécuté non par la machine directement, mais par un programme appelée une machine virtuelle. Pour chaque système opératoire (windows, linux, macintosh, etc...), JAVA fournit une machine virtuelle qui va *interpréter* le `bytecode` dans le langage de la machine pour l'exécuter. Cette solution possède plusieurs avantages :

— le code est *portable* : On peut écrire le code source sur une machine, compiler sur une machine d'architecture différente, et exécuter sur une troisième architecture (cd Figure 2.1). On peut utiliser les mêmes fichiers `.class` quelle que soit l'architecture de la machine (ordinateur, téléphone mobile, caisse enregistreuse, etc).

— la machine virtuelle permet de partager d'une manière sécurisée une machine

— le code est généralement plus compact (pas besoin d'inclure les bibliothèques comme ce serait le cas en `C` ou `C++`).

— la machine virtuelle donne l'impression que l'on dispose d'une machine entière (c'est la machine réelle qui donne du temps processeur à la machine virtuelle).

L'inconvénient majeur est que le coût en ressource de la machine virtuelle : comparé à un code exécuté directement en code natif, l'exécution d'un code JAVA sera généralement plus lente. Cependant, la machine virtuelle JAVA est très efficace et la perte en performance est assez minime.

Chapitre 3

Gestion des classes à l'aide de Packages (espaces de noms)

Un projet en JAVA peut comporter un nombre (parfois important) de classes. De la même façon que pour ordonner les fichiers sur un disque, on peut utiliser une structure hiérarchique de répertoires pour rendre plus facile l'accès aux classes. Une autre motivation des espaces de noms est de pouvoir garantir l'unicité des noms de classes. Par exemple, deux développeur peuvent avoir l'idée de développer une classe `Dauphine`. Tant que les classes se trouvent dans des espaces de noms différents, cela ne posera aucun problème et les deux classes pourront être utilisables.

1 Déclaration d'une classe

Pour simplifier, on va faire l'hypothèse que l'on travaille sur un projet qui se trouve dans le répertoire `ProjetJava` du disque dur. Ce répertoire constituera la racine de l'espace de noms. Si on n'utilise pas de d'espaces de noms, tous les fichiers JAVA se trouveront dans ce répertoire et le nom « qualifié » de la classe est simplement son nom.

Sinon, on trouvera une hiérarchie de répertoires et de fichiers dans le répertoire `ProjetJava`. Par exemple, supposons qu'une classe `MaPremiereClasse.java` se trouve dans le répertoire `ProjetJava/important/premier/`. Cette classe se trouvera dans l'espace de noms `important.premier` : le répertoire `ProjetJava` est la racine (que l'on indique pas), ensuite on indique le répertoire qui la contient, et on sépare le nom des répertoires par le symbole « . ». Le nom qualifié de la classe sera `important.premier.MaPremiereClasse`. Ce nom qualifié sera alors unique et on pourra toujours utiliser une classe grace à son nom qualifié.

Attention, en JAVA , les espaces de noms ne s'emboitent pas les uns dans les autres. `java.util` et `java.util.function` sont deux espaces de noms distincts avec chacun leurs classes et interfaces.

Dans le fichier source JAVA , on commence toujours à indiquer à quel espace de noms la classe appartient en commençant le fichier par la ligne `package <nom_du_package>`. Cette information est redondante si on connait l'emplacement du fichier dans le disque. Cependant, elle peut s'avérer très utile lorsqu'on lit le code dans un éditeur sans pouvoir lire le placement du fichier sur le disque. JAVA vérifiera que le fichier se trouve bien dans le répertoire corespondant (i.e. la classe en question devra se trouver dans le répertoire `ProjetJava/nom_du_package/`, sinon le compilateur indiquera une erreur).

2 Utiliser une classe d'un espace de noms

Pour utiliser une classe qui se trouve dans le même espace de noms, il suffit d'utiliser son nom simple. Prenons un exemple dans lequel l'espace de noms `important.premier` contient les classes `MaPremiereClasse` et `MaSecondeClasse`. Si on a besoin de manipuler un objet de la classe `MaPremiereClasse` dans la classe `MaSecondeClasse`, on peut simplement le référencer comme un objet de classe `MaPremiereClasse`.

Pour utiliser une classe à l'extérieur de son espace de noms, on peut toujours utiliser son nom qualifié. Par exemple, on développe une classe `MaDixiemeClasse` dans l'espace de nom `negligeable/remarque` et on a besoin d'un objet de classe `MaPremiereClasse` de l'espace `important.premier`. On peut la référencer avec son chemin absolu depuis la racine, i.e. `important.premier.MaPremiereClasse`.

Cependant, cela peut être lourd à l'écriture et JAVA propose un mécanisme pour utiliser une classe par son nom simple : c'est le rôle du mot clé **`import`**. Lorsqu'on a besoin plusieurs fois de référencer une classe à l'extérieur de l'espace de nom, on peut *importer* soit toutes les classes d'un espace de nom, soit une classe en particulier : l'import permet de faire comme si la/les classe(s) importée(s) font partie(s) de l'espace de nom courant.

- **`import`** `important.premier` fait comme si toutes les classes de l'espace de noms `important.premier.*` faisaient partie de l'espace de nom courant. On pourra donc maintenant utiliser les classes `MaPremiereClasse` et `MaSecondeClasse`.
- **`import`** `important.premier.MaPremiereClasse` fait comme si la classe `MaPremiereClasse` de l'espace de nom `important.premier` faisait partie de l'espace de nom courant. Dans ce cas, la classe `MaSecondeClasse` ne peut pas être utilisée comme `MaSecondeClasse` (elle peut être utilisée en faisant un autre import, ou en utilisant son chemin absolu `important.premier.MaSecondeClasse`).

L'import se fait *toujours* au début de la classe.

Attention, lorsque vous utilisez `*`, vous importez seulement les classes, pas les sous espaces de noms. Ainsi, vous ne pouvez pas utiliser `import java.*` pour obtenir tous les espaces de noms qui commencent par `java.`.

Attention aussi lorsque vous importez plusieurs espaces de noms (ce qui est fréquent), car il est possible d'avoir des conflits de noms si les espaces de noms contiennent deux classes de même nom. Si on est intéressé par une seule de ces classes, on peut importer la classe spécifique. Sinon (si on veut utiliser les deux classes), il faudra utiliser le nom qualifié complet.

L'utilisation de packages permet alors de désigner sans ambigüité une classe. On peut donc avoir deux méthodes avec exactement la même signature tant que ces deux méthodes appartiennent à des espaces de noms différents.

3 Librairies de classes

Le langage JAVA possède lui-même un ensemble de classes. Sans espace de noms, toutes ses classes se trouveraient dans un seul et même répertoire (bonjour la pagaille !). Grâce aux espaces de noms, les classes sont rangées dans une hiérarchie cohérente.

- `java.lang` contient les classes fondamentales du langage.
- `java.util` contient les classes pour manipuler des collections d'objets, des modèles d'évènements, des outils pour manipuler les dates et le temps, et beaucoup de classes utiles.
- `java.io` contient les classes relatives aux entrées et sorties
- ...

Par exemple, la classe `List` est une classe pour gérer des listes d'objets. Elle se trouve dans le package `java.util`, Evidemment, vous n'avez pas à copier à chaque fois toutes les classes définies par JAVA dans

votre répertoire de travail. L'emplacement du répertoire où se trouve l'ensemble de classes de JAVA est connu de votre système d'exploitation (windows, linux ou mac OS).

4 Compilation et Exécution

Pour compiler une classe, il faut indiquer son chemin depuis la racine.

```
javac important/premier/MaSecondeClasse.java
```

Le compilateur générera le fichier `important/premier/MaSecondeClasse.class`. Si `MaSecondeClasse` possède une méthode `main`, on lance l'exécution en spécifiant le nom complet de la classe. Pour l'exemple, on lancerait donc

```
java important.premier.MaSecondeClasse
```

Chapitre 4

Programmation orientée objet

JAVA est un langage de programmation orienté objet. Dans cette section, on va donc parler d'objets et de classes.

Un *objet* se définit par ses états (on peut aussi parler de ses caractéristiques) et son comportement. Par exemple, une voiture a des états : sa marque, son modèle, sa couleur, etc. Elle a aussi des comportements : elle peut accélerer, passer une vitesse, tourner, consommer du carburant, etc.

Une *classe* peut être comprise comme étant un plan ou un moule pour fabriquer des objets. Une classe va décrire les états possibles et les comportements possibles d'un objet. Les *états* d'un objet vont être représentés par des *variables* et les *comportements* d'un objet seront representés par des *méthodes*. Un *objet* est une *instance* d'une classe, i.e. à l'aide du plan ou du moule qu'est une classe, on fabrique un objet qui possède son état propre et son comportement propre. Par exemple, on peut créer une classe `Personnage` (NB : lorsqu'on nomme une classe, la convention suivie par tous est de commencer le nom de la classe par une lettre capitale). Lorsqu'on instantie la classe Personnage, on crée un objet de type `Personnage`.

La définition plus formelle d'une classe est un type abstrait caractérisé par des propriétés (attributs et méthodes) communes à un ensemble d'objets et permettant de créer des objets ayant ces propriétés. Un objet ou une instance de classe possède un comportement et un état qui ne peut être modifié que par les actions du comportement.

JAVA propose déjà un bon nombre de classes. Par exemple JAVA propose une classe pour manipuler les chaînes de caractères appelée `String`. On peut donc voir cette classe comme étant un moule pour fabriquer des chaînes de caractères et ce moule décrit un ensemble de comportements qui nous aidera à manipuler ces chaînes de caractères. Ainsi, la classe `String` possède toute sorte de méthodes pour manipuler les chaînes de caractères (par exemple une méthode pour retourner le i^{ime} caractère, une méthode pour comparer la chaîne avec une autre, une méthode pour connaitre la longueur de la chaîne, pour concaténer la chaîne avec une autre, etc). Créer une variable de type `String` est synonyme d'instantier la classe `String` et de créez un objet de type `String` : une chaîne de caractères est créée et on va pouvoir la manipuler.

On va illustrer les concepts de ce chapitre à l'aide d'un exemple. Imaginons que l'on veut créer une scène d'un film d'animation. On va donc devoir manipuler les personnages du film. On veut donc commencer avec une classe `Personnage`. On note une nouvelle fois l'utilisation de la convention de nommage d'une classe : son nom commence avec une majuscule. On va donc écrire le code suivant dans le fichier `Personnage.java`

```
1 public class Personnage {
2
3 }
```

le mot clé classe indique que l'on crée une classe, il est suivi par le nom de la classe et le code de la classe se trouvera entre les les accolades { et }.

1 Variables d'instance

La description de l'état d'un objet se fait par des variables qu'on appelle *variables d'instance* (elles sont aussi appelées *attributs*). Ces variables définissent les caractéristiques de l'objet. Par exemple pour la classe `Personnage`, chaque personnage doit avoir un nom.

```
1 public class Personnage {
2    private int age ;
3    private String name ;
4 }
```

Le code ci-dessus implique que chaque objet ou instance de `Personnage` aura ces deux variables.

Notez ici qu'on a indiqué que ces deux variables étaient *privées*, ce qui veut dire que seulement des méthodes de la classe `Personnage` peuvent utiliser ces variables. Il y a plusieurs motivations pour cela. Une première est que vous décidez de la partie du code qui contrôle la modification d'une variable : ici, un autre objet ne peut pas directement changer la valeur de `nom`. Une autre motivation est que le programmeur garde le contrôle de la représentation interne : tant que le comportement de l'objet reste le même, vous pouvez changer tout le code.

De manière plus informelle, les états d'un objets peuvent être connus de tous ou peuvent être cachés. La voiture de James Bond n'est pas une Aston Martin grand publique, elle possède des comportements qui ne sont pas accessibles au conducteurs normaux. Ainsi, chaque variable d'instance ou de classe et chaque méthodes peuvent avoir une *portée*. Si tout le monde peut accéder à la variable ou à la méthode, la portée est publique et on la déclare en utilisant `public`. Si une variable ou une méthode ne peut être accédée que depuis l'intérieur de la classe, la portée est privée et on la déclare en utilisant le mot clé `private`. Ne pas rendre accessible à l'extérieur permet de cacher un mécanisme interne ou permet de le protéger de l'extérieur.

2 Méthodes d'instance

On va maintenant pouvoir implémenter les méthodes qui permettent de modifier l'état d'un objet. On appelle ces méthodes les *méthodes d'instance* : ces méthodes modélise le « comportement » de l'objet. Dans l'exemple ci-dessous, on a deux méthodes d'instance qui ont pour nom `getName` et `birthday`. De la même façon que pour les variables d'instance, les méthodes ont une portée, et ici, les deux méthodes sont publiques. La méthode `getName` n'a pas de paramètre et retourne une chaîne de caractères. La méthode `birthday` quant à elle retourne un entier (`int`) et n'a pas non plus de paramètres.

Il est souvent plus naturel que la portée soit publique, mais pas toujours. On fera une différence entre des méthodes qui ne modifie pas l'objet et des méthodes qui les modifient (en anglais on parle d'un « accessor » et de « mutator »). La méthode `getName` ne modifie pas l'objet alors que la méthode `birthday` modifie l'âge du personnage.

Vous ne devez laisser publique que les méthodes pertinentes pour l'utilisateur de la classe et cacher toutes les autres méthodes (surtout si elles sont dépendentes des détails d'implémentation : si l'implémentation change, vous pourrez changer ou enlever des méthodes privées sans danger pour l'utilisateur).

```
1  public class Personnage {
2
3    private String name ;
4    public int age ;
5
6    public String getName(){
7      return name ;
8     }
9
10   public void birthday(){
11     age++ ;
12   }
13 }
```

Si `asterix` est une instance de la classe `Personnage`, vous pourrez donc appeler les méthodes comme suit :

```
System.out.println(asterix.getName()) ;
asterix.birthday() ;
```

Si la méthode n'a pas de paramètres, il faut quand même utiliser les parenthèses `(` et `)` (cela permet de bien faire la différence avec une variable). Notez que pour le moment, nous n'avons pas de moyen pour afficher l'âge du personnage, il faudrait par exemple créer une méthode `getAge`.

Comme on l'a vu dans la Section 5, on peut définir plusieurs fois la même méthode avec des signatures différentes. Par exemple, on peut avoir plusieurs versions en changeant la liste des arguments. Par exemple si on veut modéliser une ellipse de plusieurs années dans notre scenario, on voudrait pouvoir ajouter d'un coup plusieurs années à notre personnage. On aura donc une méthode avec la signature suivante :

```
public int birthday(int inc)
```

Si besoin est, vous pouvez utiliser le mot clé `this` pour faire référence à l'objet courant. Dans l'exemple ci-dessous, le paramètre de la méthode `setAge` porte le même nom que la variable d'instance `age`. Pour éviter toute confusion, on peut dans ce cas préciser que `this.age` correspond à la variable d'instance et que `age` correspond au paramètre de la méthode.

```
14   public void setAge(int age){
15     this.age = age ;
16   }
```

3 Constructeurs et création d'un objet

Une classe sert de plan ou de moule pour fabriquer un objet, ce qu'on appelle *instancier un objet*. Des méthodes spéciales, appelée des *constructeurs*, sont utilisées pour fabriquer un objet en mémoire. Ces méthodes servent donc à réserver de l'espace mémoire pour l'objet et à initialiser les variables de l'objet.

Un constructeur est une méthode qui porte le nom de la classe et qui n'a pas de type de retour. Le constructeur sans arguments, aussi appelé constructeur *par défaut* aura donc la forme suivante :

```
1 public class <nom classe> {
2    // déclaration des variable d'instances et variables de classe
3
4    // constructeur par défaut
5    public <nom classe>(){
6      // corps de la méthode
7    }
8 }
```

Comme les autres méthodes, on peut surcharger ce constructeur par défaut et avoir des constructeurs avec des signatures différentes. Pour une classe `Personnage`, on peut donc écrire :

```
1  public class Personnage {
2     public String name ;
3
4     // constructeurs
5
6     // constructeur par défaut
7     public Personnage(){
8       name="unknown" ;
9     }
10    public Personnage(String name){
11      nom = name ;
12    }
13
```

Si la valeur d'une variable d'instance n'est pas fixée explicitement dans le constructeur, elle aura automatiquement une valeur par défaut (0 pour un nombre, `false` pour un bouléen, `null` pour un objet).

Lorsque l'on déclare une variable d'instance, on peut lui donner une valeur par défaut lors de la déclaration comme dans l'exemple qui suit où la variable `name` est affectée à la chaine « `unknown` ». Cette affectation est effectuée avant l'appel du constructeur. Un constructeur pourra donc mettre à jour la valeur par défaut.

```
1 public class Personnage {
2    public String name = ''unknown'' ;
3
4    // constructeur
5
6    public Personnage(String name){
7      nom = name ;
8    }
9
```

Vous n'êtes pas obligé d'implémenter un constructeur : dans ce cas, JAVA se charge de donner automatiquement un constructeur par défaut : celui-ci initialisera toutes les variables à leur valeur par défaut (donc soit 0, `false` ou `null` selon le type). Attention cependant, si vous avez déjà défini un constructeur avec des arguments, JAVA ne mettra pas à votre disposition un constructeur par défaut : si vous voulez

aussi avoir un constructeur par défaut, il faudra alors le définir vous-mêmes.

On a vu que pour créer une variable de type primitif (comme un `int`, un `char`, etc.), on devait déclarer le type de la variable et ensuite, éventuellement, on pouvait initialiser la variable. Pour créer une variable de type complexe, i.e. un objet, on doit aussi la déclarer. Pour créer et initialiser l'objet, il faut appeler le constructeur grâce au mot-clé **new**. Ainsi, pour créer un objet de type `Personnage` on écrit le code suivant :

```
1 Personnage asterix = new Personnage("Astérix") ;
```

4 Destruction d'un objet

La destruction des objets est prise en charge par JAVA avec un "garbage collector" (GC), en français litéral, c'est un mécanisme de ramassage d'ordures. Le GC détruit les objets (i.e. efface la mémoire) qui ne sont référencés par aucun autre objet. Les destructions sont asynchrones et il n'y a pas de garantie que les objets soient détruits. Si un objet à une méthode `finalize`, celle-ci est appelée lorsque l'objet est détruit. Elle peut par exemple s'assurer que des fichiers ou des connexions sont bien fermées avant la destruction de l'objet.

5 Manipulation de références : passage par valeur, final, égalité

Commençons par l'exemple suivant où on crée un personnage appelé Astérix en ligne 1. En ligne 2, on déclare une variable qui est une instance de Personnage et on l'affecte à `asterix`.

```
1 Personnage asterix = new Personnage("Astérix") ;
2 Personnage gaulois = asterix ;
3 asterix.setAge(30) ;
4 gaulois.birthday() ;
5 System.out.println(asterix.getAge()) ;
```

Ici, on a donc un seul objet et deux variables pour accéder à cet objet. Ces variables sont deux *références* au même objet. On peut les voir comme deux noms donnés au même objet. Si on exécute le code ci-dessus, l'affichage sera donc bien de `31`. En fait, on ne manipule jamais l'objet lui-même, on manipule toujours une référence à un objet.

Dans la Section 5, on a dit que le passage des arguments d'une méthode se faisait par valeur. C'est toujours le cas. Prenons l'exemple ci-dessous où un mauvais druide possède une méthode pour faire vieillir les personnages.

```
1 public class EvalDruid {
2    public void agingFast(Personnage p){
3      p.birthday(10) ;
4      }
5    }
```

Considérons l'appel suivant

```
saruman.agingFast(asterix) ;
```

Ici, la référence `asterix` est copiée dans la variable paramètre `p`. L'objet est donc a ce moment là référencé par deux noms : `asterix` et `p`. L'objet est modifié avec l'appel de la méthode `birthday`. Après

l'exécution du code, l'objet aura donc pris 10 ans de plus.

On peut déclarer une variable d'instance comme `final` : à la fin de l'exécution du constructeur, il ne sera alors plus possible de changer la valeur de la variable. Par exemple, on pourrait fixer comme `final` la variable `name` de la classe `Personnage`. Pour un type primitif, on ne pourra donc plus changer la valeur de la variable. Lorsqu'on a un objet, c'est de nouveau sur la référence qu'est appliqué le `final`. En effet, on pourra modifier l'objet à l'aide de méthodes qui modifient l'objet, mais on ne pourra pas modifier la référence (par exemple changer l'objet à laquelle elle est associée ou changer pour la référence `null`).

Finalement, un autre élément où la manipulation de références joue un grand rôle est sur l'égalité : il faut bien faire attention si on veut l'égalité entre deux objets ou entre deux références. Considérons l'exemple suivant :

```
1 Personnage asterix = new Personnage("Astérix") ;
2 Personnage asterixBis = new Personnage("Astérix") ;
3 Personnage hero = asterix ;
4 if (asterix == hero) ...
5 if (asterix == asterixBis) ...
```

Les variables qui sont déclarées ne contiennent pas un objet mais une *référence* vers cet objet. Le test à l'aide du symbole == permet de tester si les variables référencent le même objet.

Dans l'exemple ci-dessus, `asterix` et `hero` pointent donc sur le même objet, ainsi, le test de la ligne 4 sera positif. Par contre, la variable `asterixBis` pointe sur un objet différent, même s'il a les mêmes propriétés que l'objet `asterix`, donc, même si les objets `asterix` et `asterixBis` ont les mêmes propriétés, le test de la ligne 5 sera négatif.

Pour tester l'égalité entre les propriétés de deux objets, on doit utiliser une méthode `boolean equals(Object o)` de la classe de l'objet. Dans le cadre de l'exemple, il faudra donc ajouter à la classe `Personnage` cette méthode et définir dans quelles conditions deux personnages sont égaux.

Nous reviendrons sur le problème de l'égalité dans le chapître suivant (Section 6.3).

6 Variables & méthodes de classe

Les variables d'instance sont spécifiques à chaque instance de la classe. Mais parfois, il serait bon d'avoir une variable dont la valeur soit commune à toutes les instances d'une même classe. Par exemple, si on conçoit une classe cercle et qu'on a besoin de la valeur de π, il serait bon qu'il n'y ait qu'une valeur de π en mémoire et que toutes les instances partagent cette valeur. C'est l'idée derrière le mot clé `static`. Nous l'avons déjà rencontré dans le chapitre**??** à propos de méthodes et nous avions alors dit que ces méthodes n'opèrent sur aucun objet.

6.1 Variables de classe

Si une variable est une *variable de classe*, alors il n'y a qu'une seule variable qui sera partagée par toutes les instances de cette classe. Pour déclarer une variable de classe, on utilise le mot clé `static`. Par exemple, on va ajouter la variable de classe `cptPersonnage` :

```java
public class Personnage {

  public static int cptPersonnages ;
  private final String name ;
  public int age ;

  public Personnage(String name){
    this.name = name ;
  }
```

Pour avoir accès à cette variable, il ne faut même pas avoir une instance de la classe ! En effet, pour accéder à la valeur d'une variable de classe, il suffit de concaténer le nom de la classe, le caractère point « . » et le nom de la variable de classe (comme dans l'exemple suivant) :

```java
System.out.println(Personnage.cptPersonnages) ;
```

Une utilisation possible ici est de donner un numéro d'identification unique à chaque personnage. Pour ce faire, à chaque fois que l'on va créer un nouveau personnage, on va incrémenter le compteur de personnages. Ainsi chaque personnage aura un numéro unique [1].

```java
public class Personnage {

  public static int cptPersonnages ;
  private final int id ;
  private final String name ;
  public int age ;

  public Personnage(String name){
    id = cptPersonnages++ ;
    this.name = name ;
  }
```

Comme pour les variables d'instance, on peut aussi avoir des variables de classe qui sont constantes à l'aide du mot clé `final`. Un exemple intuitif est une constante mathématique comme π ou e. Un autre exemple est l'utilisation d'un générateur de nombres aléatoires : au lieu de créer plusieurs instances du générateur de nombres aléatoires de JAVA (c'est la classe `Random`), il est préférable de n'utiliser qu'une seule instance de la classe Random [2]. On pourrait donc avoir le code suivant :

```java
public class Personnage {

  public static int cptPersonnages ;
  private final int id ;
  private final String name ;
  private static final Random generator = new Random() ;
```

S'il y a besoin de faire des calculs pour initialiser des variables static, sachez qu'il existe de blocs

1. Ceci, en faisant l'hypothèse qu'il n'y a qu'un thread qui puisse créer des personnages, si on est dans une situation de programmation concurrente, il faudra remédier à ce problème.
2. En effet, ce qui est aléatoire, c'est la suite des nombres

d'initialisation statiques.

6.2 Méthodes de classe

De même, on a des *méthodes de classe* : ces méthodes ne modifient pas l'état interne d'un objet. On peut se servir de méthodes de classes comme des méthodes utilitaires pour manipuler des objets de la classe en question.

Un exemple de la classe `Math` est le calcul de la puissance : `Math.pow(x,a)`. Ici, il n'y a pas besoin de générer une instance de `Math`, on peut appeler la méthode de classe `pow` qui calcule x^a.

On a en fait déjà vu un exemple de méthode de classe dans le chapitre **??** dans la section sur les tableaux (Section 6). On peut utiliser des méthodes de classe de la classe `Arrays` pour faire des opérations (comme le tri, la recherche binaire, etc) sur des tableaux.

Une utilisations des méthodes de classe est ce qu'on appelle les méthodes usines (« Factory methods ») qui retournent des nouvelles instances de classes.

7 Retour sur la portée : Encapsulation

On a vu que s'il on utilise le mot-clé `public`, une variable ou une méthode était accessible par toute autre classe. Parfois, c'est très pratique, mais cela peut aussi être dangereux. Par exemple on peut changer le nom d'un personnage très facilement !

```
1 Personnage asterix = new Personnage(Astérix) ;
2 asterix.name = "César" ;
```

Parfois on veut donc éviter une modification directe par l'utilisateur des composants de l'objet. Par exemple avec l'exemple ci-dessus, on se dit qu'il faudrait que l'attribut `name` de la classe personnage devienne `private`. L'utilisateur est donc privé d'un accès direct aux (ou à certains) attributs. A la place, on va proposer des méthodes pour accéder aux variables. Ainsi, on peut penser qu'un objet est une boite noire qui rend des services à l'utilisateur : l'utilisateur a un accès direct à un certain nombre de méthodes et de variables qui vont lui rendre service. Le programmeur peut donc choisir de faire évoluer le code sans dommage pour l'application tant que les services sont identiques.

Par exemple pour un projet, on a besoin de manipuler des nombres complexes. Les méthodes vont donc être `re()`, `im()`, `rho()`, `theta()` qui donnent la partie réelle, imaginaire, le module et l'angle du nombre complexe et des méthodes pour faire des opérations sur des complexes (addition, soustraction, multiplication, division, homothétie, rotation). Comment est représenté le nombre complexe dans l'implémentation est sans importance pour l'utilisateur. Ainsi, un programmeur peut faire une première version en utilisant une représentation cartésienne en utilisant `private double partie_re, partie_im ;` comme variables d'instances et écrire toutes les méthodes à partir de cette représentation.

Si le programmeur se rend compte que l'application fait beaucoup de calculs impliquant les rotations, la représentation polaire sera plus efficace. Il peut changer l'implémentation de la classe complexe et écrire les mêmes méthodes en utilisant la représentation polaire. Pour le reste de l'application, il n'y aura aucun changement, le changement est tout à fait transparent. Si toutefois la portée des variables d'instance `partie_re` et `partie_im` avait été laissée `public`, on aurait pu directement utiliser ces variables en dehors de la classe complexe, ce qui rendrait le passage à une implémentation en représentation polaire beaucoup plus compliqué puisqu'il faudrait modifier tout le code utilisant directement `partie_re` et `partie_im`.