Synthèse d'Informatique
Q1 - LFSAB1401

Benoît Legat Antoine Paris

Compilation: 23/09/2019 (21:32)

Dernière modification: 13/01/2016 (10:22) `6a6c420a`

Informations importantes Ce document est grandement inspiré de l'excellent cours donné par Olivier Bonaventure et Charles Pecheur à l'EPL (École Polytechnique de Louvain), faculté de l'UCL (Université Catholique de Louvain). Il est écrit par les auteurs susnommés avec l'aide de tous les autres étudiants, la vôtre est donc la bienvenue. Il y a toujours moyen de l'améliorer, surtout si le cours change car la synthèse doit alors être mise à jour en conséquence. On peut retrouver le code source et un lien vers la dernière version du pdf à l'adresse suivante

`https://github.com/Gp2mv3/Syntheses`.

On y trouve aussi le contenu du `README` qui contient de plus amples informations, vous êtes invités à le lire.

Il y est indiqué que les questions, signalements d'erreurs, suggestions d'améliorations ou quelque discussion que ce soit relative au projet sont à spécifier de préférence à l'adresse suivante

`https://github.com/Gp2mv3/Syntheses/issues`.

Ça permet à tout le monde de les voir, les commenter et agir en conséquence. Vous êtes d'ailleurs invités à participer aux discussions.

Vous trouverez aussi des informations dans le wiki

`https://github.com/Gp2mv3/Syntheses/wiki`

comme le statut des documents pour chaque cours

`https://github.com/Gp2mv3/Syntheses/wiki/Status`

Vous pouvez d'ailleurs remarquer qu'il en manque encore beaucoup, votre aide est la bienvenue.

Pour contribuer au bug tracker et au wiki, il vous suffira de créer un compte sur GitHub. Pour interagir avec le code des documents, il vous faudra installer LaTeX. Pour interagir directement avec le code sur GitHub, vous devrez utiliser `git`. Si cela pose problème, nous sommes évidemment ouverts à des contributeurs envoyant leurs changements par mail (à l'adresse contact.epldrive@gmail.com) ou par n'importe quel autre moyen.

Table des matières

Première partie

Instructions, expressions et variables

1 Instructions

Un programme est composé principalement d'instructions qu'il exécute l'une à la suite de l'autre.

Chaque instruction se termine par un `;`, on les écrit tout le temps une par ligne pour aider à la lecture du programme mais le compilateur pourrait comprendre le programme s'ils n'étaient que sur une seule ligne.

1.1 Blocs d'instruction

Certaines structures de Java (comme le `if`) demandent un bloc. Un bloc, c'est simplement un regroupement d'instructions. On le délimite avec des accolades (`{` et `}`).

```
{
  <instruction_1>;
  <instruction_2>;
  ...
  <instruction_n>;
}
```

Portée des variables Les variables déclarées dans un bloc ne sont pas accessibles à l'extérieur du bloc.

2 Expressions

Un composé fondamental de ces instructions est l'expression. Ces expressions sont composées d'opérations entre d'autres expressions. Une valeur ou une variable sont tous les deux des expressions. Chaque expression est évaluée lors de l'exécution du programme.

3 Variables

Dans un programme, on utilise des variables comme en mathématiques. Par contre, en informatique, les variables peuvent changer de valeur.

Chaque variable a un type et en Java, on ne peut pas changer son type.

Pour créer une variable, il faut lui donner un type et ce type ne changera plus par la suite.

3.1 Types

Il y a deux sortes de type en Java, les types primitifs et les classes.

Il y a 8 types primitifs mais les plus importants sont les

- `int`, ce sont des nombres entiers, ils valent par exemple `0` ou `-8` ou encore `42`;
- `double`, ce sont les nombres à virgule, il valent par exemple `3.14` ou `1.41`;
- `char`, ce sont les charactères, il valent par exemple `'a'` ou `'A'` ou `'0'` ou même un retour à la ligne `'\n'`;
- `boolean`, ils valent `true` ou `false`.

Les autres sont comme les `int` ou les `double` mais utilisent plus ou moins de mémoire. Cependant, à l'époque actuelle, la mémoire n'est plus si limitée et ils sont moins utilisés.

Par contre il est important de remarquer que ces deux types ne savent pas contenir des nombres d'une taille arbitraire, ils ont chacun leur limite

- Un `int` sait seulement contenir un entier appartenant à $[-2^{31}; 2^{31}[$. On retient que

$$2^{31} = 2 \cdot \left(2^{10}\right)^3 = 2 \cdot (1024)^3 \approx 2 \cdot (1000)^3 = 2 \cdot 10^9;$$

- Un `double` retient un nombre à virgule flottante, c'est à dire qu'il le retient en écriture scientifique. Il retient ses décimales et la puissance de 10. Il peut donc être très grand mais n'est pas si précis que ça.

  Vous pouvez vous en apercevoir en vérifiant que l'expression

```
0.1 + 0.2 == 0.3
```

est évalué à `false`. C'est du au fait que les décimales sont écrite en binaire dans la mémoire et que ces nombres ne sont pas si rond en binaire.
En règle générale, on ne test jamais l'égalité entre deux `double` mais plutôt la valeur absolue de leur différence.

3.2 Manipulation de variables

On peut utiliser les variables de 3 manières différentes, on peut les créer, leur assigner une valeur et la lire dans une expression. Dans les exemples de cette sous-section, on considèrera qu'on utilise des `int` mais c'est évidemment le même principe pour les autres types primitifs.

3.2.1 Création d'une variable

Pour créer une variable, il faut lui choisir un nom qui n'est pas encore utilisé, il ne peut pas commencer par un chiffre mais peut contenir chiffres, lettres et underscores (_).
Supposons qu'on crée un `int` qu'on appelle `foobar`.

```
int foobar;
```

Lors de leur création, les variables sont initialisées à `0` ce qui pour les `boolean` signifie `false`.
On peut aussi donner une valeur lors de l'initialisation de la variable comme dans l'exemple suivant

```
int foobar = 1;
```

3.2.2 Assignation

On peut modifier la valeur d'une variable à tout moment du programme en lui assignant la valeur d'une expression. Il faut bien évidemment que la valeur de l'expression soit du même type. La syntaxe est la suivante

```
foobar = <expression>;
```

Il est important de remarquer que le `=` de l'assignation n'est pas le même qu'en mathématique, on peut par exemple faire

```
foobar = foobar + 1;
```

Dans ce cas ci, il est important d'insister sur l'ordre à l'exécution. On évalue d'abord le membre de droite qui n'est rien d'autre qu'une expression puis on assigne cette valeur dans le membre de gauche.

3.2.3 Casting

Supposons que `a` soit de type `<type_a>` et `b` soit de type `<type_b>` et qu'on veuille assigner à `a` la valeur de `b`.
Si `<type_a>` et `<type_b>` ne sont pas les mêmes, on a deux possibilités.

— Soit on fait un *cast explicite*, c'est à dire qu'on spécifie à JAVA qu'on veut transformer la valeur de `b` pour qu'elle ait le type `<type_a>`. La syntaxe est la suivante

```
(<type_a>) b
```

Évidemment, ça ne marche pas pour n'importe quel type. On peut caster n'importe quel type primitif dans n'importe quel type primitif mais on ne peut pas caster un type primitif dans un objet ni le contraire (sauf entre les types primitifs et leur classe wrapper comme `int` et `Integer`). Par contre, on peut caster un objet dans un autre si et seulement si, l'un hérite de l'autre ou le contraire ;

— Soit on fait un *cast implicite*, comme son nom l'indique, on ne dit rien à Java mais il effectue tout de même un cast. Entre type primitifs, en règle générale, Java fait un cast quand il ne perd pas de précision. Les casts se faisant implicitement sont donc

- `char` à `int` ;
- `int` à `double` ;
- `int` à `Integer` et vice versa ;
- `char` à `Character` et vice versa ;
- `double` à `Double` et vice versa ;
- `boolean` à `Boolean` et vice versa ;
- `<class_1>` à `<class_2>` si `<class_1>` hérite de `<class_2>`.

Sans parler des `byte`, `short` et `float`. De plus, il y a une relation de transitivité entre les casts implicites, c'est à dire que si on sait caster implicitement `<type_1>` dans `<type_2>` et `<type_2>` dans `<type_3>`, on sait caster implicitement `<type_1>` dans `<type_3>`.

Encodage Le cast de `char` en `int` peut paraitre étrange si on ne comprend pas comment un caractère est socké dans un ordinateur. Un ordinateur ne sait stocker que des nombres, il a donc une tableau de traduction des nombres en caractère.

En castant un `char` en `int`, on voit le nombre qui est réellement stocké dans l'ordinateur. Ce tableau de traduction dépend de l'encodage. Comme l'informatique a commencé au États-Unis, le premier encodage, l'ASCII, ne traitait que les caractères de l'alphabet latin non-accentués et les autres caractères du clavier qwerty. Cet encodage n'utilisait que 128 caractères alors qu'on peut avoir 256 nombres différents sur un byte.

Il restait donc un peu de place et presque un encodage différent pour chaque langue a été créé en donnant en utilisant ces places restantes (c'est le cas du latin-1). Ainsi tous les caractères pouvaient être stockés sur 1 byte (sauf pour certaines langues).

Le problème de tous ces encodages c'est qu'on ne pouvait pas utiliser 2 alphabets différents dans un même texte et qu'il y avait souvent des problèmes de compatibilité.

L'unicode a donc été créé avec l'UTF-8 qui permet d'écrire presque tous les caractères possibles et imaginables. Les caractères simples n'utilisent toujours qu'un seul byte et les caractères plus complexes comme ceux de l'alphabet Klingon prennent plus de bytes. Le fait que tous les caractères n'utilisent pas le même nombre de byte permet de gagner beaucoup de place car on utilise le plus souvent des caractères simples.

Heureusement, la plupart de ces encodages (dont l'UTF-8) utilisent la même traduction que l'ascii pour les caractères simples, c'est pour ça que quand vous avez des problèmes d'encodage, il n'y a que les caractères spéciaux comme les caractères accentués qui rende bizarrement.

Si on veut voir la valeur d'un caractère représentant un chiffre, il faut prendre en compte le fait que les chiffres se suivent dans la table comme on peut voir dans la table 1. Si `c` vaut `'1'` et qu'on fait `int i = c`, `i` vaudra 49. Pour avoir 1, il faut faire `int i = c - '0'`. On peut aussi faire `int i = Character.digit(c , 10)`.

Casting et héritage On a vu qu'on pouvait caster des objets dans une classe parente. Mais quand on fait cela, on ne sait évidemment plus appeler ses méthodes spécifiques que la classe parente n'a pas. Grâce au keyword `instanceof`, on peut néanmoins recaster l'objet dans la classe de départ en vérifiant que c'est bien une instance de cette classe. Par exemple, on peut implémenter la méthode `equals` comme suit

```
1 public class Foo {
2   int data;
3   public Foo (int data) {
4     this.data = data;
5   }
6   public boolean equals (Object o) {
7     if (o instanceof Foo) {
8       Foo foo = (Foo) o;
9       return this.data == foo.data;
```

Décimal	Symbol
⋮	⋮
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
⋮	⋮

TABLE 1 – Tableau ascii pour les chiffres.

```
10      }
11      return false;
12    }
13  }
```

Ainsi, ça permet de donner comme argument à `equals` un objet de n'importe quel classe car toute classe hérite de `Object`.

3.2.4 Les constantes

En informatique, on aime bien se donner des règles supplémentaires à respecter qui sont vérifiées par le compilateur pour diminuer le plus possible les chances que notre programme compile mais ne fonctionne pas.

Pour cela, il est possible de spécifier que des variables sont constantes. Dès qu'on voudra modifier leur valeur, le code ne compilera pas.

Pour cela, à la création de la variable, on ajoute le keyword `final` de la manière suivante.

```
1 final <type> <name>[ = <expression>];
```

Si on ne donne pas `= <expression>`, on peut encore lui faire une assignation. Ça permet d'initialiser les variables dans le constructeur en fonction de ses paramètres.

Assignation de la même valeur En fait, on a le droit d'assigner une valeur à une constante autant de fois qu'on veut, tant que cette valeur est la même que celle de la constante ou que la contante n'a pas encore été initialisée.

Convention Par convention, on écrit tout le temps les constantes en majuscule.

3.2.5 Les raccourcis

Certaines assignations sont très courrantes, elles possèdent donc un raccourcis.

```
1 foo = foo + bar;
```

peut s'écrire

```
1 foo += bar;
```

Il y a des raccourcis similaire pour les opérateurs `-`, `*` et `/`.

Encore plus raccourcis, au lieu d'écrire

```
foo += 1;
```

on peut écrire

```
foo++;
```

Il y a un raccourci pour l'opérateur `-`.

4 Les opérateurs

Les expressions seraient bien pauvres si on ne pouvait pas les manipuler entre elles. Il existe les opérateurs suivants dont le résultat est du même type que les deux variables sur lesquels ils sont appliqués

`a + b`	Vaut la somme de `a` et `b`
`a - b`	Vaut la soustraction de `a` par `b`
`a * b`	Vaut la multiplication de `a` et `b`
`a / b`	Vaut la division de `a` par `b`
`a % b`	Vaut le reste de la division euclidienne de `a` par `b`

Il existe aussi des opérateurs dont le résultat est un `boolean`

`a == b`	Vaut `true` si `a` est égal à `b`
`a != b`	Vaut `true` si `a` est différent de `b`
`a < b`	Vaut `true` si `a` est plus petit strictement que `b`
`a <= b`	Vaut `true` si `a` est plus petit ou égal à `b`
`a > b`	Vaut `true` si `a` est plus grand strictement que `b`
`a >= b`	Vaut `true` si `a` est plus grand ou égal à `b`

Tous ces opérateurs sont des opérateurs binaires car ils recquièrent deux valeurs. Il existe aussi un opérateur unaire et un opérateur ternaire

Deuxième partie
Structures de contrôle

Quand on exécute un programme, les instructions s'exécutent l'une après l'autre, dans l'ordre dans lequel elles ont été écrites.

Seulement, il existe des moyens de passez outre cet ordre.

Les conditions Une condition est toute expression s'évaluant en `boolean`.

5 Bloc, instruction ou structure de contrôle ?

Ce qui est magique avec les stuctures de contrôle, c'est qu'elles permettent de manipuler, non plus des variables, mais carrément des instructions ! Elles permettent de déterminer si certaines instructions vont être exécutées en fonction de la valeur d'une condition.

Comme on veut parfois appliquer cela sur plusieurs instructions, on dit que ces structures conditionnelles s'appliquent sur des blocs d'instructions. Seulement, on peut aussi donner une seule instruction où même directement une structure de contrôle sans les mettre dans un bloc (sans accolade).

En fait, si on ne leur donne pas un bloc mais directement des instructions, ces structures prennent la première instruction.

Attention néanmoins car la pratique de donner une instruction au lieu d'un bloc à ces structures, bien qu'acceptée par JAVA, n'enchante pas tous les programmeurs. En effet, elle ajoute souvent de l'ambiguïté et rappelle donc pour beaucoup du temps perdu à débugger.

Pour illustrer cela, remarquez que, contrairement aux apparences,

```
if (a_number % another_number == 0)
  amount_of_divisors++;
  is_prime = false;
```

est équivalent à

```
if (a_number % another_number == 0) {
  amount_of_divisors++;
}
is_prime = false;
```

et non à

```
if (a_number % another_number == 0) {
  amount_of_divisors++;
  is_prime = false;
}
```

6 Les structures conditionnelles

Il est possible d'exécuter un bloc d'instruction seulement si une condition est vraie.

```
if (<condition>) <bloc>
```

Le bloc sera exécuté seulement si la condition vaut `true`.

Il est possible de demander d'exécuter un autre bloc si la condition est fausse.

```
if (<condition>) <bloc_1>
else <bloc_2>
```

Si la condition vaut `true`, le premier bloc sera exécuté, sinon, le second bloc sera exécuté.

6.1 Quid des else if ?

Rappelons-nous (voir Section 5) qu'à la place d'un bloc, on peut donner une structure de contrôle. On peut donc dire que

```
if (<condition_1>) <bloc_1>
else if (<condition_2>) <bloc_2>
else <bloc_3>
```

est équivalent à

```
if (<condition_1>) <bloc_1>
else {
  if (<condition_2>) <bloc_2>
  else <bloc_3>
}
```

Les `else if` ne sont donc pas vraiment un nouveau concept lorsqu'on connait les `else` et les `if`.

7 Les boucles

Une boucle exécute un bloc *tant que* la condition est vraie. Avant chaque exécution du bloc, il vérifie si la condition vaut `true`. Si c'est le cas, le bloc est exécuté. Sinon, le programme poursuit son court.

La syntaxe est fort semblable au `if`.

```
1 while (<condition>) <bloc>
```

7.1 Les boucles for

Lorsqu'on veut itérer une opération, souvent, on a un itérateur qu'on initialise avant la boucle et qu'on incrémente après chaque boucle.

```
1 <init>;
2 while (condition) {
3   ...
4   <inc>;
5 }
```

La boucle `for` permet d'écrire ça de façon plus condensée

```
1 for (<init>; <condition>; <inc>) {
2   ...
3 }
```

— `<init>` peut être n'importe quelle instruction mais si elle crée des variables, ces dernières seront locales au `for` ;

— `<inc>` peut être n'importe quelle instruction, ça peut même contenir plusieurs instructions séparées par des virgules mais cette pratique n'est pas appréciée par tout le monde.

Le fait que les variables déclarées dans `<init>` ne soit pas accessibles en dehors de la boucle est assez pratique car souvent, on y déclare des variables dont on aura pas besoin après le `for`. C'est souvent des variables qui nous permettent de parcourir un tableau, c'est pourquoi on les appelle des itérateurs.

Troisième partie
Les tableaux

Le tableau de Java est une structure de donnée assez intéressante. Elle permet de stocker un nombre fixe de variables de même type ou héritant d'une même classe (Voir Section 3.2.3).

8 Création de tableaux

Une variable ne contient jamais un tableau en lui même, tout comme les objects, elle contient une référence vers un tableau. À sa création, elle vaut `null`.

Pour créer un tableau, un utilise le keyword `new`. Il crée un nouveau tableau et renvoit sa référence.

8.1 Création de tableau à une dimension

On utilise la syntaxe suivante [1]

```
1 <type>\[\] <name> = new <type>\[<length>\];
```

1. Ici, les
devant les [,] servent à dire que ce n'est pas de crochets pour dire que c'est optionnel mais vraiment des crochets qu'il faut mettre.

Par exemple, pour faire un tableau qu'on nomme `tab` de 42 `int`, on écrit

```
int[] tab = new int[42];
```

8.2 Création de tableau à plusieurs dimension

On fait un tableau à plus d'une dimension en faisant des tableaux de tableau. Par exemple, si dans chaque case d'un tableau à 2 éléments, on met un tableau de `int` à 4 éléments, on a une matrice 2×4. Ça s'écrit

```
int[][] matrix = new int[2][4];
```

9 Manipulation de tableau

Lorsqu'on a créé un tableau, on peut changer ses valeurs (sauf si c'est un tableau de constante bien entendu) ou juste y accéder.

Il faut cependant savoir qu'en informatique, on commence à compter à partir de 0, donc la $\texttt{n}^{\text{e}}$ case est à l'indice `n-1`.

Pour obtenir la 21^{e} case du tableau `tab`, il faut donc faire

```
tab[20]
```

Pour assigner à la 2^{e} colonne de la 1^{re} ligne de `matrix` la valeur 42, il faut écrire

```
matrix[0][1] = 42;
```

9.1 Taille de tableau

La taille d'un tableau est son nombre d'éléments. Elle est fixe. Pour obtenir la taille du tableau `tab`, on fait

```
tab.length
```

et ça vaudrait 2.

Si on voulait le nombre de colonnes d'une matrice, il suffit de demander le nombre d'élément de n'importe quelle ligne (s'il y en a au moins une !).

```
matrix[0].length
```

Mais dans la plupart des cas, on peut juste redemander la longueur pour chaque ligne.

9.2 Exemples

Pour illustrer cela, voici comment faire une méthode qui fait la somme des éléments d'un tableau de `int` à deux dimension.

```
public static int sumMatrix (int[][] matrix) {
  sum = 0;
  for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
      sum += matrix[i][j];
    }
  }
  return sum;
}
```

Et voici une méthode qui fait la copie d'un tableau. En effet, quand on fait une assignation d'un tableau avec la valeur d'un autre tableau, on fait une copie de la référence mais le tableau reste le même !

```java
public static int[] arrayCopy (int[] array) {
  int[] copy = new int[array.length];
  for (int i = 0; i < array.length; i++) {
    copy[i] = array[i];
  }
  return copy;
}
```

Quatrième partie
Les classes

10 La portée

Les variables et les méthodes ont une certaine portée, c'est à dire qu'on ne peut pas y accéder depuis n'importe où.

À chaque fois qu'on défini une méthode ou une variable de classe ou d'instance, on doit lui attribuer une portée.

Il existe 3 portées différentes

- `public` : peut être accédé depuis n'importe quelle classe et n'importe quel object ;
- `protected` : peut être accédé depuis toute classe ou object de la classe en question ou d'une classe fille ;
- `private` : peut être accédé uniquement depuis la classe en question et ses instances.

De plus, on ne peut pas accéder à une variable ou méthode d'instance depuis une méthode de classe.

Pour spécifier la portée d'une variable ou d'une méthode, on rajoute le keyword `public`, `protected` ou `private` en fonction de la portée au début de sa définition.

En anglais, portée se dit *scope*.

11 Variables et méthodes

11.1 Statique ou non-statique

Dans une classe, on peut définir des variables et des méthodes. Ces dernières peuvent être soit statiques, soit non-statiques.

statiques Ce sont les méthodes et variables de classe. Elles sont dans la classe uniquement pour des raisons de portée. Les variables statiques ne sont créées qu'une seule fois même si 42 instances de la classe sont créées. C'est pourquoi les constantes sont souvent statiques.

non-statiques Ce sont les méthodes et variables d'instance. De nouvelles variables sont créées à chaque création d'instance de la classe. Les variables d'instance référencées dans les méthodes d'instance sont celles spécifique de l'object en question. Les variables d'instances constituent le *contexte* de l'objet. Lorsqu'on appelle des méthodes d'instances sur un objet, ces méthodes sont dans le contexte de cet objet, elle peuvent le lire et le modifier.

Pour spécifier si une variable ou une méthode est statique, on ajoute le keyword `static` à la suite du keyword pour la portée. Si le keyword n'est pas présent, elle est non-statique.

11.1.1 Le secret des méthodes non-statiques

La difficulté de comprendre ce concept vient du fait qu'on est jamais obligé d'écrire une méthode en non-statique, *toute méthode non-statique peut être écrite en méthode statique*. D'ailleurs, on pourrait

imaginer que JAVA transforme toutes les méthodes non-statiques en méthodes statiques juste avant la compilation pour se simplifier la vie, les méthodes non-statiques ne seraient alors que là pour faciliter la vie du programmeur. Par exemple, prenons la classe suivante

```
public class Soldier {
  private int HP;
  private int attack;
  public Soldier (int HP0, int attack) {
    HP = HP0;
    this.attack = attack;
  }
  public void getShot (int damage) {
    HP -= damage;
  }
  public boolean isAlive () {
    return HP > 0;
  }
  public void shoot (Soldier s) {
    if (isAlive()) {
      s.getShot(attack);
    }
  }
  public static void main (String[] args) {
    Soldier a = new Soldier(100, 10);
    Soldier b = new Soldier(100, 11);
    while (a.isAlive() && b.isAlive()) {
      a.shoot(b);
      if (b.isAlive()) {
        b.shoot(a);
      }
    }
  }
}
```

JAVA commencerait par ajouter des `this` pour lever toute ambiguité. Puis changerait toutes les méthodes non-statiques en statiques juste en rajoutant un argument et le keyword `static`. Ces méthodes pourraient toujours accéder aux variables instances même si elles sont `private` car elles sont dans la même classe. JAVA changerait ensuite tous les appels en appelant les méthodes sur la classe et non sur les objets et donnerait ces objets en argument.

```
public class Soldier {
  private int HP;
  private int attack;
  public Soldier (int HP0, int attack) {
    this.HP = HP0;
    this.attack = attack;
  }
  public static void getShot (Soldier this, int damage) {
    this.HP -= damage;
  }
  public static boolean isAlive (Soldier this) {
    return this.HP > 0;
  }
  public static void shoot (Soldier this, Soldier s) {
    if (HP > 0) {
      Soldier.getShot(s, this.attack);
    }
  }
  public static void main (String[] args) {
    Soldier a = new Soldier(100, 10);
    Soldier b = new Soldier(100, 11);
```

```
22      while (Soldier.isAlive(a) && Soldier.isAlive(b)) {
23        Soldier.shoot(a, b);
24        if (Soldier.isAlive(b)) {
25          Soldier.shoot(b, a);
26        }
27      }
28    }
29  }
```

Ici, `Soldier.` devant les méthodes est facultatif car on est dans la méthode `Soldier`.

11.2 Méthodes

Une méthode est comme une fonction mathématique, on lui donne des argument et elle renvoit une valeur en fonction de ces arguments.

Si c'est une méthode d'instance, son comportement dépendra aussi de son contexte, c'est à dire de la valeur des variables d'instance de l'objet sur lequel la méthode est appelée.

11.2.1 Définition de méthode

Pour définir une méthode, on doit spécifier le type de la valeur qu'elle renvoit ainsi que le type de chacun des paramètres et le nom avec lesquels on les référencera à l'intérieur de la méthode. La syntaxe est la suivante

```
1  <scope> <return_type> <name> (<type_1> <param_1>, ...) bloc
```

Cette ligne (sans le `bloc`) est appelée la *signature* de la méthode.

- `<scope>` est la portée de la méthode. Il peut être suivit par d'autres keywords ;
- `<return_type>` est le type de la valeur retournée par la méthode. Si la méthode ne retourne rien, `<return_type>` vaut `void` ;
- `<name>` est le nom de la méthode ;
- `<type_i>` est le type du i^e paramètre et `<param_i>` est son nom.
  `<type_i> <param_i>` est appelé le *paramètre formel* ;
- `bloc` est un bloc d'instruction doté d'un keyword supplémentaire : `return`. Sa syntaxe est la suivante

  ```
  1      return <expression>;
  ```

  `return` termine la méthode immédiatement (même s'il restait des instructions à exécuter) et renvoie `<expression>` comme valeur de retour de la méthode.

  - Si `<return_type>` vaut `void`, on peut quand même arrêter l'exécution de la méthode avec l'instruction `return;` ;
  - Si `<return_type>` ne vaut pas void, le compilateur doit pouvoir s'assurer qu'il pourra toujours trouver quoi retourner sinon, le code ne compilera pas. Par exemple, si, un `return` est dans une structure de contrôle et pas après, il doit alors y en avoir un dans chaque embranchement et le `else` doit être présent. Par exemple,

    ```
    1  public static double div (double a, double b) {
    2    if (b != 0) {
    3      return a / b;
    4    }
    5  }
    ```

    ne compilera pas.

11.2.2 Appel de méthode

Pour appeler une méthode, il faut utiliser la syntaxe suivante [2]

```
[<locator>.]<name>(<arg_1>, ...)
```

Si la méthode retourne `void`, cet appel est une *instruction*. Sinon, c'est une *expression* dont le type est le type de retour de la méthode appelée.

- `<locator>` dépend du cas dans lequel on est
  - Si on veut accéder à une méthode de classe extérieure à la classe dans laquelle on est, il vaut le nom de cette classe. Si on est dans la classe en question, `<locator>.` est optionnel ;
  - Si on veut accéder à une méthode d'instance, il vaut le nom de la variable référençant cet objet. Si on est dans l'objet en question, `<locator>.` est optionnel sauf si la variable qu'on veut accéder est cachée par une variable locale du même nom. On utilise le keyword `this` pour obtenir la référence vers l'objet dans lequel une méthode est exécutée. Par exemple, dans le code suivant, `this` est obligatoire ;

```
public class ImageBitmap {
  private int[][] map;
  public ImageBitmap (int[][] map) {
    this.map = map;
  }
}
```

- `<name>` est le nom de la méthode ;
- `<arg_i>` est une *expression* qui donne la valeur du `i`^{e} argument de la méthode. Ça équivaut à faire une assignation

```
    <type_i> <param_i> = <arg_i>;
```

  à la différence que le type auquel est évalué de l'expression `<arg_i>` sert à déterminer quelle fonction est appelée.

  `<arg_i>` est appelé le *paramètre effectif*.

  Attention Comme c'est une assignation, si `<arg_i>` est une variable, `<param_i>` sera une copie de cette variable ! Dès lors, changer `<param_i>` ne change pas la valeur de `<arg_i>`. Néanmoins, si `<arg_i>` est la référence vers un object, c'est la référence qui est copiée, pas l'objet ! Donc si on modifie l'objet vers qui `<param_i>` fait référence, l'objet vers qui `<arg_i>` fait référence est aussi modifié car c'est le même.

Java va localiser la fonction à appeler en grâce au `<locator>`, au `<name>` et au type des `<arg_i>` (dans l'ordre dans lequels ils sont !).

11.3 Variables

La syntaxe pour utiliser les variables dans une classe est la même que celle des méthodes sauf qu'on ne donne pas d'arguments. On la définit comme suit [3]

```
<scope> <return_type> <name>[ = <expression>];
```

où l'initialisation avec `<expression>` est optionnelle. Elle s'utilise dans une expression comme suit

```
[<locator>.]<name>
```

Et on lui fait une assignation comme suit

2. [et] sont là pour indiquer que c'est optionnel.
3. [et] sont là pour indiquer que c'est optionnel.

```
[<locator>.]<name> = <expression>;
```

JAVA distingue les variables des méthodes au fait qu'on ne mette pas de parenthèses. Par exemple,

```
x.length
```

c'est la variable `length` de l'objet `x` et

```
x.length()
```

c'est la méthode `length` qui ne prend pas d'argument de l'objet `x`.

Objet ou instance "objet" et "instance" sont presque des synonymes. En fait, ils veulent dire la même chose mais on ne les utilise pas au même moment. On ne dit pas "c'est un objet de la classe `String`" mais plutôt "c'est une instance de la classe `String`" ou encore "c'est un objet de type `String`". Aussi, pour parler de variables ou de méthodes non-statiques, on dit des "variables d'instance" et des "méthodes d'instance" et non "d'objet".

12 Classes et objets

12.1 Définition de classes

Une classe est en quelque sorte la définition d'une "famille d'objet" qu'on pourrait aussi appeler une "classe d'objet", d'où le nom.

En JAVA, il y a une bijection entre classes et fichiers sources. Chaque classe `<name>` doit se trouver dans un fichier `<name>.java`. Dans ce dernier, la syntaxe est la suivante

```
public class <name> {
  ...
}
```

Mettez toujours `public` dans le cadre de ce cours, les `...` sont à remplacer par les définitions de variables et de méthodes dont la syntaxe a été vue à la section 11.

12.2 Création et manipulation d'objets

12.2.1 Références

Il y a un concept clé à comprendre sur les objets. Une variable ne contient jamais un objet, elle contient une *référence* vers un objet. On ne manipule jamais d'objet, on manipule des références vers des objets.

Une référence vaut soit `null`, soit la référence vers un objet.

Lorsqu'on crée une variable de type non-primitif sans l'initialiser, sa valeur est `null`.

12.2.2 Création d'objets

Pour créeer un objet, on utilise le keyword `new`. Sa syntaxe est la suivante

```
new <class>(<arg_1>, ...)
```

C'est une expression qui renvoie une référence vers l'objet créé. Le type de cette référence est donc bien évidemment `<class>`.

Pour stocker cette référence, il faut créer une variable de type `<class>` ou une classe parente. Par exemple, dans

```
ArrayList array = new ArrayList();
```

array vaut une référence vers un objet de type `ArrayList` qui vient d'être créé.

12.2.3 Constructeur

Un objet est un contexte pour ses méthodes d'instances. Lorsque vous définissez une classe, vous donnez un sens à vos variables d'instances.

Pour vos méthodes d'instance, en plus du devoir de respecter la pre et la post, vous devez respecter ce sens que vous donnez à vos variables d'instance. Pour ce faire, écrivez une méthode telle que, si la pre et la cohérence de vos variables d'instance à l'entrée est respectée, la post et la cohérence de vos variables d'instance à la sorties est respectée aussi.

Ainsi, vos variables d'instances seront toujours cohérentes, non ? Mais quand est-il de la cohérence de vos variable d'instance à la création de votre objet ? Rappelez vous que pour qu'un raisonnement par récurrence marche, il faut une induction et une *initialisation*.

Cette initialisation, c'est votre constructeur. Vous pouvez créer différents constructeurs avec une signature différente, pour créer différentes valeurs de départ cohérentes pour votre classe.

Ce constructeur est appelé par le keyword `new`. JAVA cherche le constructeur avec la signature correspondante aux types des arguments d'entrées `(<arg_1>, ...)`.

Un constructeur, ça ressemble à une méthode mais ce n'est pas une méthode.

— Premièrement, il ne faut pas lui stipuler de valeur de retour ;

— Deuxièmement, pour l'appeler, il y a deux façon, soit `this(<arg_1>, ...)`, ça appelle le constructeur de la classe dans laquelle on est. Soit `super(<arg_1>, ...)`, ça appelle le constructeur de la classe parente à celle dans laquelle on écrit (ce n'est pas spécialement la classe parente à celle de `this`, voir la section 13.2). Ces appels ne peuvent que se trouver à la première ligne d'un constructeur. En fait, si il n'y a pas de ligne comme ça à la première ligne d'un constructeur, JAVA considère qu'on a mis `super()`. Si la classe `super` n'a pas de constructeur sans paramètre, JAVA ne compilera pas et vous devrez appeler le constructeur de `super` avec des arguments.

Voici deux exemples :

1. Une classe `Number` contenant un nombre.

```
public class Number {
  private int value;
  public Number (int value) {
    this.value = value;
  }
  public Number () {
    this(42); // default value
  }
}
```

Ici, `super` vaut `Object` qui a un constructeur sans arguements.

2. Une classe `Human`

```
public class Human {
  private String name;
  public Human (String name) {
    this.name = name;
  }
}
```

et une classe `Woman`.

```
public class Woman extends Human {
  public Woman (String name) {
    super(name);
  }
}
```

12.2.4 Manipulation d'objets

On s'interdit de rendre les variables d'instance publiques, c'est le principe d'*encapsulation*. Dès lors, on peut manipuler les objets qu'en appelant leur méthodes d'instances.

Lorsqu'on implémente une méthode d'instance, on peut non seulement accéder à tous les éléments statiques à portée mais aussi aux variables et méthodes d'instance. En y fesant référence, on référence celles de l'objet dans lequel on est.

Attention Dans le code suivant

```
String s = "42";
String t = s;
```

`s` et `t` référencent vers le même objet. L'objet n'a pas été copié. Seulement la référence a été copié. C'est pareil pour le passage de variable à une fonction.

12.2.5 Destruction d'objets

Un élément de JAVA appelé le *Garbage Collection* garde en mémoire, pour chaque objet créé, le nombre de variables qui le référence. Dès que ce nombre tombe à zéro, JAVA sait que plus personne ne pourra toucher à cette objet et marque la place qu'il prenait dans la mémoire comme libre.

13 Héritage

La programmation orientée objet n'aurait que peut d'intérêt sans l'héritage et le polymorphisme (voir section 13.1). L'héritage permet de n'écrire du code commun à deux classes qu'une seule fois.

Dire qu'une classe hérite d'une autre signifie que toutes les méthodes et variables définies par l'autre et par ses classes parentes sont également définies pour la classe fille.

On peut également les redéfinir.

13.1 Polymorphisme

Quand on appelle une méthode sur un objet qu'on a casté dans sa classe parente mais que sa classe redéfini la méthode. Quelle implémentation de la méthode va être appelée ?

Le polymorphisme, c'est quand c'est la méthode de la classe fille qui est appelée. C'est à dire que l'objet, même après avoir été casté, se souvient de ce qu'il est vraiment.

Par exemple, supposons qu'on ait créé une classe `Animal` ainsi qu'une classe `Dog` et `Cat`.

Supposons qu'on ait une référence `a` de type `Animal` mais qu'on mette dans `a` une référence vers un objet de type `Dog`.

```
Animal a = new Dog();
```

Que va faire `a.shout()` ?

Grâce au polymorphisme, ça va faire `Waf waf`.

13.2 Liens statiques et dynamiques

À cause de l'héritage, quand vous implémentez une méthode dans une classe vous ne savez jamais vraiment si elle va être appelée sur des objets de cette classe ou d'une classe fille.

Dès lors, quand vous utilisez `this` et `super`, à quelle classe faites-vous référence ?

Lorsque vous utilisez `this`, vous faites juste référence à l'objet dans lequel vous êtes, c'est un lien dynamique car si le type de votre objet est une classe fille et quelle a réimplémenté la méthode qu'on essaie d'appeler avec `this`, c'est la réimplémentation qu'on appelle.

`super` par contre est un lien statique, il fait référence à la classe parente à la classe dans laquelle on écrit. On essaie d'éviter les liens statiques, essayez de n'utiliser `super` que dans la première ligne d'un constructeur.

13.3 Classes abstraites et interfaces

Parfois, on aimerait pas qu'on instantie une classe. Par exemple, ça a du sens de faire `new Dog()` mais ça n'a aucun sens de faire `new Animal()`. Aussi, on aimerait bien dire que tous les classes filles de `Animal` doivent implémenter `shout` comme ça on peut implémenter la méthode `makeNoise` dans la classe `Animal`.

JAVA permet de faire ça de manière élégante avec le concept de classe abstraite. L'idée, c'est de ne pas implémenter `shout` et de dire "Je ne l'implémente pas, et d'ailleurs, vous ne pouvez même pas créer d'animaux car je n'ai pas implémenté toutes les méthodes". Voilà comment faire cela

```
public abstract class Animal {
  public abstract void shout ();
  public void makeNoise () {
    for (int i = 0; i < 42; i++) {
      shout();
    }
  }
}
```

Maintenant, toutes les classes qui héritent de `Animal` ont deux choix : implémenter `shout` ou être abstraite.

13.3.1 L'héritage multiple

L'héritage, avant tout, c'est une relation "est". Mais que faire quand on veut faire deux relation "est" ? Par exemple, un chien est un animal mais c'est aussi un carnivore ! Du coup, j'aimerais pouvoir écrire une classe `Carnivore` pour pouvoir utiliser les bienfaits du polymorphisme sur les animaux carnivores.

Pour faire celà, l'héritage multiple a été inventé. L'héritage multiple, c'est tout simplement hériter de 2 classes. Seulement, en JAVA, on *ne peut pas* faire d'héritage multiple. Pourquoi ? Imaginez que `Carnivore` et `Animal` implément la méthode `walk` et que `Dog` ne la réimplémente pas. Que faire quand on appelle `walk` sur un `Dog` ?

13.3.2 Interfaces

L'idée la plus simple pour régler le problème est de dire qu' on a le droit d'hériter de plusieurs classe mais une seule des classes qu'on hérite doit avoir implémenté des méthodes. Les autres doivent n'avoir que des méthodes abstraites. Comme ça, plus de soucis !

Pour ça, le JAVA a inventé les interfaces, c'est tout simplement des classes complètement abstraites. C'est à dire qu'elles n'implémentent aucune de leur méthodes.

On ne dit plus hériter dans ce cas-ci mais implémenter une interface. La syntaxe est la suivante

Définition d'interfaces La syntaxe pour définir une interface `<name_1>` est la suivante

```
public interface <name_1> {
  ...
}
```

Implémentation d'interface La syntaxe pour dire que `<name_2>` implémente l'interface `<name_1>` est la suivante

```
public class <name_2> implements <name_1> {
}
```

La règle de l'héritage multiple en JAVA est alors : "une classe peut hériter que d'une seule classe mais peut implémenter autant d'interface qu'elle veut".

14 Exception

Lorsque vous écrivez une méthodes, des cas exceptionnels peuvent arriver. Pour que votre programme soit robuste, vous devez gérez ces cas.

14.1 Exceptions contrôlées et non-contrôlées

On peut différentier ces cas exceptionnels en deux cas

— ceux dus à une erreur de la part du programmeur ;

— ceux dus à une erreur de la part de l'utilisateur.

La manière de gérer ça proprement en Java, c'est avec les exceptions. Chaque exception différente est une classe qui hérite de la classe `Exception`.

— Les exceptions dues à une erreur de la part du programmeur héritent de `RuntimeException` qui hérite lui de `Exception`, c'est les exceptions non-contrôlées (unchecked) ;

— Les exceptions dues à une erreur de la part de l'utilisateur n'héritent pas de `RuntimeException` mais héritent bien de `Exception`, c'est les exceptions contrôlées (checked).

La différence entre les deux types exceptions, c'est que si une méthode lance une exception contrôlées ou qu'elle appelle une méthode qui en lance une sans la catcher, il doit rajouter à la fin de sa signature `throws <name>` où `<name>` est le nom de la classe de l'exception.

14.2 Lancer une expression

Pour lancer une exception, il faut utiliser le keyword `throw`. La syntaxe est la suivante

```
throw new <name>(<arg_1>, ...);
```

où `<name>` est le nom de la classe de l'exception à lancer. `new <name>(<arg_1>, ...)` crée un objet en appelant le constructeur ce `<name>` avec les arguments `(<arg_1>, ...)`.

Il arrête immédiatement l'exécution de la méthode, de la méthode qui l'appelait et ainsi de suite jusqu'à une méthode qui catch l'exception. Si aucune méthode ne la catch, l'exécution s'arrête avec une erreur.

14.3 Catcher une exception

Pour catcher une exception dont le nom de la classe est `<name>` lancée par un bloc d'instruction `<bloc_1>`, la syntaxe est la suivante

```
try <bloc>
catch (<name_1> <var_1>) <bloc_1>
catch (<name_2> <var_2>) <bloc_2>
...
```

où `<var_i>` est un nom de variable et `<bloc_i>` est un bloc d'instruction.

Si `<bloc>` lance une exception de type `<name_i>`, `<bloc_i>` est exécuté avec la variable `<var_i>` valant une référence vers l'exception lancée.

14.4 Les assertions

Les assertions en Java permettent de vérifier si les préconditions sont respectées en utilisant le mécanisme des exceptions. Une assertion s'utilise de la manière suivante :

```
assert <condition> : <message>;
```

Si la condition n'est pas respectée, le programme s'arrêtera et le message d'erreur `message` s'affichera.

15 Lecture et écriture dans un fichier texte

Package nécessaire Toutes les méthodes utilisées dans cette section se trouvent dans le package `java.io`, il est donc indispensable de l'importer à l'aide de la ligne de code suivante :

```
1 import java.io.*;
```

Gestion des exceptions Lorsqu'on manipule des fichiers, il est important de traiter les différentes exceptions (notamment les `IOException`) qui peuvent survenir. Le traitement de ces exceptions ne sera pas indiqué dans cette section.

15.1 Lire un fichier texte

Pour lire un fichier texte, nous aurons besoin de deux classes : `BufferedReader` et `FileReader`. L'ouverture du fichier se fait de la manière suivante :

```
1 BufferedReader file = new BufferedReader(new FileReader("filename.txt"));
```

Il est ensuite possible de parcourir le texte ligne par ligne grâce à la méthode `readLine()` de la classe `BufferedReader`. Cette méthode ne prend pas de paramètre et retourne `null` une fois à la fin du fichier :

```
1 file.readLine();
```

Enfin, il est important de bien fermer le flux une fois toutes les manipulations terminées. Le flux se ferme avec la ligne de code suivante :

```
1 file.close();
```

15.2 Ecrire dans un fichier texte

Pour écrire dans un fichier texte, nous aurons encore besoin de deux classes : `PrintWriter` et `FileWriter` . Comme pour la lecture, il faut avant tout ouvrir le fichier :

```
1 PrintWriter file = new PrintWriter(new FileWriter("filename.txt"));
```

Il est ensuite possible d'écrire du texte ligne par ligne grâce à la méthode `println()` de la classe `PrintWriter`. Par exemple :

```
1 file.println("Blablablablabla...");
```

Ici aussi, il ne faut pas oublier de fermer le fichier une fois toutes les manipulations terminées.

16 Les collections

16.1 La classe `ArrayList`

La classe `ArrayList` est une classe :

- Qui utilise un tableau comme composant ;
- Dont la taille est extensible (contrairement aux tableaux classique de Java) ;
- Qui offre des méthodes supplémentaires.

Un `ArrayList` s'initialise de la façon suivante :

```
1 ArrayList<E> name = new Arraylist<E>();
```

Où `E` peut être n'importe quel type d'objet (on parle alors de type *générique*). On ne peut pas créer d'`ArrayList` de types primitifs, pour créer un `ArrayList` contenant des entiers par exemple, il faudra utiliser la classe /wrapper| de `int`, à savoir `Integer`.

Un `ArrayList` possède entre autre les méthodes suivantes :

- `boolean add(E obj)` : ajoute `obj` en fin de liste ;
- `void add(int index, E obj)` : ajoute `obj` à la position `index` ;
- `E get(int index)` : retourne une référence vers l'objet à la position `index` ;
- `void set(int index, E obj)` : remplace l'élément à la position `index` par `obj` ;
- `E remove(int index)` : retire et retourne une référence vers l'élément situé à la position `index` ;
- `boolean contains(E obj)` : renvoie `true` si `obj` appartient à la liste.

Attention Un élément d'un `ArrayList` ne contient pas d'objet directement mais bien la référence vers cet objet (comme d'habitude).

Remarque La classe `ArrayList<E>` implémente l'interface `List<E>` qui implémente elle même l'interface `Collection<E>`.

16.2 Parcourir une collection : les itérateurs

Il est possible de parcourir une collection de manière élégante grâce à des objets de types `Iterator<E>`. Pour utiliser un itérateur, il faut connaître les deux méthodes suivantes :

- `boolean hasNext()` : renvoie `true` s'il y a encore des éléments à parcourir dans la collection, `false` sinon ;
- `E next()` : renvoie une référence vers l'élement suivant dans la liste. Lance une `NoSuchElementException` si `hasNext()== false`.

Chaque collection a son itérateur. Pour obtenir une référence vers un itérateur, il existe une méthode que toute collection implémente : il s'agit de la méthode `iterator()`.

Voici un exemple complet d'utilisation d'un itérateur sur une liste contenant des objets de types `Clients`. Chaque `Clients` possède un nom.

```
1 Collection<Client> listeClients = ... ;
2 Iterator<Client> iterator = listeClients.iterator();
3 Client current;
4
5 while(iterator.hasNext())
6 {
7   current = iterator.next();
8   System.out.println(current.getName());
9 }
```

Cinquième partie

Annexes

A Installation

A.1 Sous GNU/Linux

Vous avez deux choix, Oracle JDK ou OpenJDK. Il ne sera pas discuté ici quel est la meilleure.

A.1.1 OpenJDK

OpenJDK se trouve dans votre gestionnaire de packets. Par exemple, sur Ubuntu, c'est dans le Ubuntu Software Center. Les packets nécessaires sont `openjdk-7-jre` et `openjdk-7-jdk`.

A.1.2 Oracle JDK

Pour Ubuntu, suivez le guide suivant
`http://www.webupd8.org/2012/01/install-oracle-java-jdk-7-in-ubuntu-via.html`
Pour Debian, suivez le guide suivant
`http://www.webupd8.org/2012/06/how-to-install-oracle-java-7-in-debian.html`

B Écrire, compiler et exécuter

Nous allons différencier la programmation par IDE et en console. Les remarques sur l'écriture, la programmation et la compilation sont dans la partie console mais elles valent aussi pour la programmation à travers un IDE.

B.1 IDE

Un **IDE** (**I**ntegrated **D**evelopment **E**nvironment) fourni un éditeur de texte et des boutons pour compiler et exécuter. Tout ça avec une interface graphique.

Pour Java, on peut citer notamment

— Eclipse ;

— Netbeans ;

— BlueJ.

Ils sont tous les trois libres donc gratuits.

B.2 Console

Programmer en console signifie utiliser l'éditeur de texte et le compilateur séparément. On appelle ça programmer à la console car on appelle souvent le compilateur et on exécute le programme à travers l'invite de commande.

Supposons qu'on crée le projet `Foobar` contenant la classe `Foo` et la classe `Bar`, la méthode `main` étant dans la classe `Foo`.

Une structure classique serait de placer `Foo.java` et `Bar.java` dans un dossier `src/` et placer un fichier `README` à la racine du projet expliquant brièvement ce que fait le programme.

B.2.1 Se déplacer dans la console

Dans une console, on se situe toujours dans un dossier. Lorqu'on ouvre une console, on se trouve dans le dossier personnel. Pour savoir dans lequel on se trouve, exécutez la commande suivante (**P**rint **W**orking **D**irectory) :

```
$ pwd
/home/jean
```

Pour voir les fichiers dans le dossier courant, exécutez la commande suivante :

```
$ ls
Desktop                 Music                   Templates
Documents               Pictures                Videos
Downloads               Public
```

Pour se déplacer, il faut utiliser la commande `cd` (**C**hange **D**irectory).

```
$ cd Documents
$ pwd
/home/jean/Documents
$ cd ..
$ pwd
/home/jean
```

B.2.2 Écrire

Pour écrire `Foo.java`, `Bar.java` et `README`, il nous faut un éditeur de texte.

— Sur Linux, les plus simples sont *Gedit* et *Kate* ;

— Sur Mac OS, *TextMate* est un excellent choix ;

— Sur Windows, je conseille *Notepad++*.

Il y en a plein d'autre évidemment. Comment ne pas citer *Emacs* et *Vim* qui, bien qu'étant très anciens, sont considérés par beaucoup pour être les plus complets et les plus puissants. Attention néanmoins, la *learning curve* de ces derniers est assez raide.

Si vous n'avez pas d'interface graphique et que vous ne voulez pas utiliser un éditeur de texte aussi compliqué que *Emacs* ou *Vim*, considérez *nano* ou *pico*.

Pour lancer un éditeur de texte depuis la console, positionnez-vous dans le même dossier que le fichier à ouvrir et exécutez la commande suivante :

```
$ vim Foo.java
```

Pour un éditeur de texte graphique, il vaut mieux l'appeler ajouter un `&` pour qu'il s'exécute en background et que l'invite de commande ne se bloque pas :

```
$ gedit Foo.java &
```

B.2.3 Compiler

Il faut compiler chaque classe en fichier `.class`. Pour cela, la commande à utiliser est `javac`. Avec un terminal ouvert et positionné dans le fichier `src/`, exécutez la commande suivante :

```
$ javac Foo.java Bar.java
```

Notez que l'ordre entre les fichiers `.java` n'a aucune importance.

B.2.4 Exécuter

À nouveau, ouvrez un terminal et positionnez le dans le fichier `src/`. Il faut dire à Java dans quelle classe se trouve la méthode `main`. Dans notre cas, c'est dans la classe `Foo`. Dès lors, exécutez la commande suivante :

```
$ java Foo
```

Notez qu'on écrit pas d'extension à `Foo`, on écrit ni `Foo.java` ni `Foo.class`.