Algorithmie et Programmation

Chapitre 3 : les instructions de sélection

Sommaire

La programmation est la combinaison de trois formants élémentaires. Nous en avons déjà vu un, la séquence. Les deux formants restants sont des instructions de contrôle. Jusqu'à présent, les instructions d'un programme sont exécutées les unes après les autres du début jusqu'à la fin. Cela est plutôt limité comme flux d'exécution. C'est pourquoi, il existe des instructions de contrôle permettant de modifier la progression du flux d'exécution par branchement (ou saut) contrôlé.

Ces instructions de contrôle sont :
- la sélection (de test) : permettant d'exécuter un ordre sous certaines conditions
- la répétition (les boucles) : permettant de répéter un ordre plusieurs fois

Cette partie n°2 est consacrée aux instructions de sélection.

1.De quoi parle-t-on ?

Reprenons le cas de notre « programmation algorithmique du touriste égaré ». Normalement, notre algorithme ressemblera à quelque chose comme : « Allez tout droit jusqu'au prochain carrefour, puis prenez à droite et ensuite la deuxième à gauche, et vous y êtes ».
Mais en cas de doute, cela pourrait devenir : « Allez tout droit jusqu'au prochain carrefour et là regardez à droite. Si la rue est autorisée à la circulation, alors prenez la et ensuite c'est la deuxième à gauche. Mais si en revanche elle est en sens interdit, alors continuez jusqu'à la prochaine à droite, prenez celle-là, et ensuite la première à droite ».
Ce deuxième algorithme a ceci de supérieur au premier qu'il prévoit, en fonction d'une situation pouvant se présenter de deux façons différentes, deux façons alternatives d'agir. Cela suppose que l'interlocuteur (le touriste) sache analyser la condition que nous avons fixée à son comportement pour effectuer la série d'actions correspondantes (« la rue est-elle en sens interdit ? »).

En ce qui concerne la programmation informatique, les ordinateurs possèdent cette aptitude. Il est possible d'indiquer des séries d'instructions à effectuer selon que la situation se présente d'une manière ou d'une autre. C'est le rôle des instructions de sélection, on parle aussi de structure conditionnelle. Concrètement, celles-ci permettent de contrôler le flux de l'exécution d'un algorithme en fonctions de conditions qui ne sont **connues qu'au moment de l'exécution**.

2.Structure des instructions de sélection.

Pour bien comprendre ce qui suit, écrivons le programme de notre touriste perdu :

```
Allez tout droit jusqu'au prochain carrefour
Si la rue à droite est autorisée à la circulation Alors
        Tournez à droite
        Avancez
        Prenez la deuxième à gauche
Sinon
        Continuez jusqu'à la prochaine rue à droite
        Prenez cette rue
        Prenez la première à droite
Fin De Si
```

Cet algorithme du touriste perdu utilise une instruction de contrôle de sélection

Il n'y a que deux formes possibles pour une instruction de sélection ; la forme de gauche est la forme complète, celle de droite la forme simple.

if (*condition*){
  Bloc d'instructions 1
} else {
  Bloc d'instructions 2
}
	if (*condition*){
	  Bloc d'instructions
	}

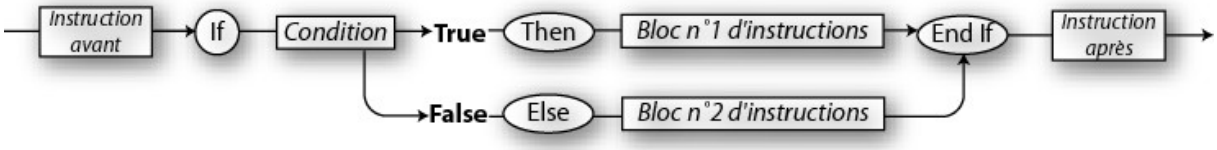
If est l'instruction de branchement conditionnel. Elle sert à orienter l'exécution d'un algorithme selon deux chemins distincts. Ceci appelle quelques explications.

La *condition* désigne n'importe quelle expression retournant une valeur booléenne.

Une condition évalue donc, soit :
- une variable de type **boolean** qui contient **true** ou **false**
- une comparaison qui, par son évaluation, donne une valeur **true** ou **false**

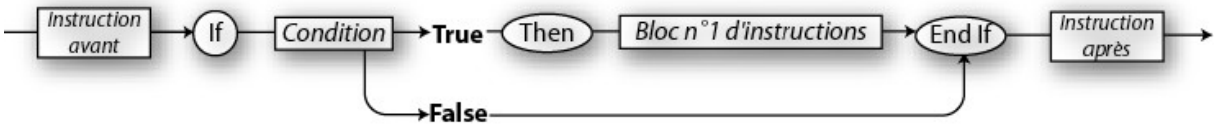
Le fonctionnement d'un test est donc le suivant :
- On évalue la condition
- Si l'évaluation vaut **true**, on exécute le bloc d'instructions 1, mais on n'exécute **PAS** le bloc d'instructions 2
- Si par contre l'évaluation vaut **false**, on exécute le bloc d'instructions 2, mais **PAS** le bloc d'instructions 1

Forme complète :



Donc, seul un des deux blocs entre en jeu, cela dépend du résultat de l'évaluation de la condition.

Forme simple :



- la forme simplifiée correspond au cas où l'un des deux blocs du **if** est vide. Dès lors, plutôt que d'écrire **else {}**, il est plus simple de supprimer le **else**.

Remarques :
- veillez à respecter l'indentation !

Illustrons tout cela avec l'exemple suivant :

```
import java.util.*;

public class Selection {
      //constante pour représenter l'age de majorité
      static final int MAJORITE = 18;

      static int lireAge(){
          Scanner lectureClavier = new Scanner(System.in);
          System.out.print("Quel est votre âge ? ");
          int age = lectureClavier.nextInt();
          return age;
      }
```

```
	public static void main(String[] args) {
		int age = lireAge();
		//Test de l'age :
		//Si l'age est supérieur ou égal à la majorité
		//Alors afficher "Vous avez accès au site."
		//Sinon afficher "Accès au site interdit."
		if (age >= MAJORITE) {
			System.out.println("Vous avez accès au site.");
		} else {
			System.out.println("Accès au site interdit.");
		}
	}
}
```

A l'exécution de cet algorithme, le message affiché dépend de l'âge saisi **au moment de l'exécution**. En effet, la condition évalue l'expression booléenne *age >= MAJORITE* qui donne soit **true**, et on exécute le premier bloc, soit **false** et on exécute le second bloc.

3.Utilisation des comparaisons.

Comparaison simple.

Rappelons qu'une comparaison est composée de trois éléments (cf. Partie n°1) :

- une valeur
- un opérateur de comparaison (<, <=, >, >=, ==, !=)
- une autre valeur

Les valeurs peuvent être a priori de n'importe quel type, mais la comparaison doit se faire sur deux valeurs de même type. Cette comparaison constitue donc une affirmation, qui a un moment donné est **true** ou **false**.

A noter que les opérateurs de comparaison s'emploient tout-à-fait avec des caractères. Chaque caractère étant codé au travers de la table des caractères Unicode, on obtient :

- 't' < 'w' est **true** (la lettre t possède un code décimal plus petit que la lettre w)
- "Maman" > "Papa" est **false** (la lettre M a un code inférieur à la lettre P)
- "maman" > "Papa" est **true** (la lettre m a un code supérieur à la lettre P)

Comparaisons composées.

Attention à certains raccourcis du langage courant qui peuvent mener à des non-sens informatiques. Prenons un exemple : « une note est-elle comprise entre 0 et 100 ? ». On peut être tenté de traduire ce test par : 0 <= uneNote <= 100. Or, une telle expression, qui a du sens en mathématiques, ne veut rien dire en programmation.
En programmation, une telle comparaison s'exprime par l'association de plusieurs comparaisons. En effet, « une note comprise entre 0 et 100 » cache non une, mais deux comparaisons. Car elle revient à dire que « la note est supérieur à 0 **et** la note est inférieur à 100». Il y a donc bien deux comparaisons, reliées par l'opérateur logique ET (**&&** en Java**)**. (cf. Partie n°1).

On peut alors écrire ce qui suit :

```
import java.util.*;
```

```
public class ComparaisonComposee {
      //constantes pour représenter la note minimum et la note maximum
      static final int NOTE_MAX = 6;
      static final int NOTE_MIN = 1;

      static double lireNote(){
          Scanner lectureClavier = new Scanner(System.in);
          System.out.print("Saisissez une note: ");
          double note = lectureClavier.nextDouble();
          return note;
      }

      public static void main(String[] args) {
             double note = lireNote();
             // test de la validité d'une note
             if (note >= NOTE_MIN && note <= NOTE_MAX) {
                    System.out.println("Cette note est valide.");
             } else {
                    System.out.println("Cette note n'est pas valide.");
             }
       }
}
```

4.Les opérateurs logiques.

- Le **ET (&&)** a le même sens en informatique que dans le langage courant.
  Pour que l'expression :

  expression1 **&&** expression2 soit **true**

  il faut impérativement que expression1 soit **true** ET que expression2 soit **true**.

  Ex. : `estUneNoteValide = (note >= NOTE_MIN && note <= NOTE_MAX)`

- Il faut se méfier un peu plus du **Ou** (**||** en Java). Pour que l'expression :

  expression1 **||** expression2 soit **true**

  il suffit que expression1 soit **true** OU que expression2 soit **true**.

  Ex. : `noteEstInvalide = ( note < NOTE_MIN || note > NOTE_MAX)`
  `obtientUnRabais = (unAge <= 16 || estEtudiant)`

Donc, si une seule des deux expressions est True, alors l'expression est True, de plus, si expression1 est **true** et que expression2 est **true** aussi, expression1 **||** expression2 est **true**. Le **||** ne veut donc pas dire « ou bien ».

- L'erreur à ne pas faire, c'est bien sûr de formuler une expression qui ne pourra jamais être **true**. En effet, une expression dont on sait à l'avance qu'elle sera toujours fausse est une expression qui ne servirait à rien.

  Par exemple : i**f** (note<0 **&&** note>100) **{**…

  Il est évident qu'il n'existe pas de valeur de note pouvant répondre à cette condition !

- L'opérateur **!** inverse une expression. Par exemple :
  !(x > 15 **)** revient donc à écrire **(**x <= 15**)**

- Attention tout de même lors de l'inversion d'une expression composée !
  !(x > 15 **&&** x<30**)** revient donc à écrire **(**x <= 15 **||** x>=30**)**
  !(x < 10 **||** x > 20**)** revient donc à écrire **(**x >= 10 **&&** x<=20**)**
  Ces théories logiques s'appellent **les lois de De Morgan.**

5.Tests imbriqués.

Graphiquement, on peut très facilement représenter un **if** comme un aiguillage de chemin de fer. Un **if** ouvre donc deux voies, correspondant à deux traitements différents. Mais il y a des tas de situations où deux voies ne suffisent pas.
Par exemple, un programme devant donner une appréciation sur une note :

```
import java.util.*;

public class TestImbrique {

      static double lireNote(){
          Scanner lectureClavier = new Scanner(System.in);
          System.out.print("Saisissez une note: ");
          double note = lectureClavier.nextDouble();
          return note;
      }

      public static void main(String[] args) {
             double note = lireNote();
             //Test si la note est inférieure à 1, et donc invalide
             if (note < 1) {
                    System.out.println("Note invalide");
             }
             //Test l'appartenance à [1;3[
             if (note >= 1 && note < 3) {
                    System.out.println("Très insuffisant");
             }
             //Test l'appartenance à [3;4[
             if (note >= 3 && note < 4) {
                    System.out.println("A confirmer");
             }
             //Test l'appartenance à [4;5[
             if (note >= 4 && note < 5) {
                    System.out.println("Suffisant");
             }
             //Test l'appartenance à [5;6]
             if (note >= 5 && note <= 6) {
                    System.out.println("Bien");
             }
             //Test si la note est supérieure à 6, et donc invalide
             if (note > 6) {
                    System.out.println("Note invalide")
             }
       }
}
```

Mais quelque-chose ne va pas dans ce code !

En effet, les conditions se ressemblent plus ou moins, et on oblige la machine à examiner six tests successifs alors que tous portent sur une même chose, la variable **note**.
Il serait ainsi bien plus rationnel d'imbriquer les tests de cette manière :

```
import java.util.*;


public class TestImbrique {

	static double lireNote(){
	    Scanner lectureClavier = new Scanner(System.in);
	    System.out.print("Saisissez une note: ");
	    double note = lectureClavier.nextDouble();
	    return note;
	}

	public static void main(String[] args) {
		double note = lireNote();
		//Test d'abord si la note est invalide, cad lorsque
		//la note est hors de l'intervalle [1;6]
		if (note < 1 || note > 6) {
			System.out.println("Note invalide");
		} else {
		    //A ce niveau la note est valide.
		    if (note < 3) {
		        System.out.println("Très insuffisant");
		    } else {
		        if (note < 4) {
		            System.out.println("A confirmer");
		        } else {
		            if (note < 5) {
		                System.out.println("Suffisant");
		            } else {
		                //celle-ci est logiquement dans [5;6]
		                //aucun nouveau test n'est donc nécessaire
		                System.out.println("Bien");
		            }
		        }
		    }
		}
	}
}
```

Cette version est bien plus logique et en plus nous avons fait des économies au niveau de la saisie de code : au lieu de devoir saisir six comparaisons, dont quatre composées, nous n'avons plus que quatre comparaisons, dont une composée.
Mais surtout, la conséquence, c'est que nous avons fait des économies sur le temps d'exécution par la machine. Si la note est inférieure strict à zéro ou supérieure strict à 100, on écrit dorénavant "Note invalide" et passe directement à la fin, sans être ralenti par l'examen d'autres possibilités (qui sont forcément fausses).
Notez également, que les cas d'invalidité de la note sont regroupés en une seule condition. Cela permet notamment de n'avoir à changer le message d'affichage `"Note invalide"` qu'à un seul endroit du code si cela est nécessaire.

Cette deuxième version n'est donc pas seulement plus simple à écrire et plus lisible, elle est également plus performante à l'exécution et plus simple à modifier.

Les structures de tests imbriqués sont donc un outil indispensable à la simplification et à l'optimisation des algorithmes.

Remarque : dans l'écriture de tests imbriqués, l'indentation prend encore plus d'importance si on ne veut pas s'y perdre.

Comme ce genre d'imbrication est très fréquent, il existe une écriture plus simple syntaxiquement nommée **if else if** (si sinon si) . Voilà le même code que ci-dessus écrit avec ce raccourcis d'écriture:

```
import java.util.*;


public class TestImbrique {

      static double lireNote(){
          Scanner lectureClavier = new Scanner(System.in);
          System.out.print("Saisissez une note: ");
          double note = lectureClavier.nextDouble();
          return note;
      }

      public static void main(String[] args) {
             double note = lireNote();
             //Test d'abord si la note est invalide
             if (note < 1 || note > 6) {
                    System.out.println("Note invalide");
             } else if (note < 3) {
                    System.out.println("Très insuffisant");
             } else if (note < 4) {
                    System.out.println("A confirmer");
             } else if (note < 5) {
                    System.out.println("Suffisant");
             } else {
                     System.out.println("Bien");
             }
       }
}
```

On obtient un code compacte, plus facile à lire et à comprendre. Mais attention, ce raccourcis d'écriture est plus restrictif que la version précédente ! En effet, dans notre exemple, il est maintenant plus difficile de rajouter une message indiquant que la note est valide, alors que dans la version précédente, on pouvait facilement le rajouter à la place du commentaire "//A ce niveau la note est valide."

6.Utilisation d'une variable booléenne.

Le plus souvent, l'expression qui sert à contrôler un **if** implique des opérateurs de comparaison. Techniquement, pourtant, cela n'est pas indispensable. Il est en effet possible de contrôler un **if** grâce à une seule variable de type **boolean**. Rappelons que ce type de variable est prévu pour stocker une des deux valeurs **true** ou **false**. Or, le résultat d'une comparaison est **true** ou **false**, on peut donc stocker ce résultat dans une variable de type **boolean**, puis l'utiliser ensuite dans un **if**.

Voyons cela dans l'exemple suivant :

```
import java.util.*;
```

```
public class TestImbrique {

	static double lireTemperature(){
	    Scanner lectureClavier = new Scanner(System.in);
	    System.out.print("Entrez la température de l'eau : ");
	    double temperature = lectureClavier.nextDouble();
	    return temperature;
	}

	static boolean eauGlace(double temperature){
	    boolean estDeLaGlace = (temperature <= 0);
	    return estDeLaGlace;
	}

	static boolean eauLiquide(double temperature){
	    boolean estLiquide = (temperature > 0 && temperature < 100);
	    return estLiquide;
	}

	public static void main(String[] args) {
		double temperature = lireTemperature();
		if (eauGlace(temperature)) {
			System.out.println("eau sous forme de la glace");
		} else if (eauLiquide(temperature)) {
			System.out.println("eau liquide");
		} else {
			System.out.println("eau sous forme de vapeur");
		}
	 }
}
```

A priori, cette technique ne présente guère d'intérêt : on a alourdi plutôt qu'allégé l'algorithme de départ, en lui faisant recourir à deux variables supplémentaires.
Cependant, nous pouvons argumenter quelques intérêts notables :

- une variable booléenne n'a besoin que d'**un seul byte au maximum** pour être stockée. L'alourdissement de ce point de vue n'est donc pas considérable.
- dans certains cas, notamment celui de comparaisons composées très lourdes (avec plusieurs **&&** et **||**) cette technique peut faciliter le travail du programmeur en augmentant la lisibilité.
- les fonctions ainsi crées sont prêtes à être facilement réutilisées dans d'autres programmes.