Chapitre 3
Programmation objet

1. Introduction à la POO

Avec Java, la notion d'objet est omniprésente et nécessite un minimum d'apprentissage. Nous allons donc voir dans un premier temps les principes de la programmation objet et le vocabulaire associé, puis nous verrons comment mettre cela en application avec Java.

Dans un langage procédural classique, le fonctionnement d'une application est réglé par une succession d'appels aux différentes procédures et fonctions disponibles dans le code. Ces procédures et fonctions sont chargées de manipuler les données de l'application qui sont représentées par les variables de l'application. Il n'y a aucun lien entre les données et le code qui les manipule. Dans un langage objet, on va au contraire essayer de regrouper le code. Ce regroupement est appelé une classe. Une application développée avec un langage objet est donc constituée de nombreuses classes représentant les différents éléments manipulés par l'application. Les classes vont décrire les caractéristiques de chacun des éléments. C'est ensuite l'assemblage de ces éléments qui va permettre le fonctionnement de l'application.

Ce principe est largement utilisé dans d'autres domaines que l'informatique. Dans l'industrie automobile, par exemple, il n'existe certainement pas, chez aucun constructeur, un plan complet décrivant les milliers de pièces constituant un véhicule. Cependant, chaque sous-ensemble d'un véhicule peut être représenté par un plan spécifique (le châssis, la boîte de vitesses, le moteur...). Chaque sous-ensemble est également décomposé jusqu'à la pièce élémentaire (un boulon, un piston, un pignon...). C'est l'assemblage de tous ces éléments qui permet la fabrication d'un véhicule.

En fait, ce n'est pas l'assemblage des plans qui permet la construction du véhicule, mais l'assemblage des pièces fabriquées à partir de ces plans. Dans une application informatique, c'est l'assemblage des objets créés à partir des classes qui va permettre le fonctionnement de l'application. Les deux termes classe et objet sont souvent confondus, mais ils représentent des notions bien distinctes. La classe décrit la structure d'un élément alors que l'objet représente un exemplaire créé sur le modèle de cette structure. Après sa création, un objet est indépendant des autres objets construits à partir de la même classe. Par exemple, une portière de voiture pourra après fabrication être peinte d'une couleur différente des autres portières fabriquées selon le même plan.

Les classes sont constituées de champs et de méthodes. Les champs représentent les caractéristiques des objets. Ils sont représentés par des variables et il est donc possible de lire leur contenu ou de leur affecter une valeur directement. Le robot qui va peindre une portière va modifier le champ couleur de cette portière. Les méthodes représentent les actions qu'un objet peut effectuer. Elles sont mises en œuvre par la création de procédures ou de fonctions dans une classe.

Ceci n'est qu'une facette de la programmation orientée objet. Trois autres concepts sont également fondamentaux :

– l'encapsulation,

– l'héritage,

– le polymorphisme.

© Editions ENI - All rights reserved

L'encapsulation consiste à cacher les éléments qui ne sont pas nécessaires pour l'utilisation d'un objet. Cette technique permet de garantir que l'objet sera correctement utilisé. C'est un principe qui est aussi largement utilisé dans d'autres domaines que l'informatique. Pour reprendre l'exemple de l'industrie automobile, savez-vous comment fonctionne la boîte de vitesses de votre voiture ?

Pour changer de vitesse allez-vous directement modifier la position des différents engrenages ? Heureusement que non. Les constructeurs ont en fait prévu des solutions plus pratiques pour la manipulation de la boîte de vitesse.

Les éléments d'une classe visibles de l'extérieur de la classe sont appelés l'interface de la classe. Dans le cas de notre voiture, le levier de vitesses constitue l'interface de la boîte de vitesses. C'est par son intermédiaire que l'on peut agir sans risque sur les mécanismes internes de la boîte de vitesses.

L'héritage permet la création d'une nouvelle classe à partir d'une classe existante. La classe servant de modèle est appelée classe de base, classe mère ou super-classe. La classe ainsi créée hérite des caractéristiques de sa classe de base. Elle peut aussi être personnalisée en y ajoutant des caractéristiques supplémentaires. Les classes créées à partir d'une classe de base sont appelées classes dérivées, classes filles ou sous-classes. Ce principe est bien sûr aussi utilisé dans le monde industriel. La boîte de vitesses de votre voiture comporte peut-être cinq rapports. Les ingénieurs qui ont conçu cette pièce ne sont certainement pas repartis de zéro. Ils ont repris le plan de la génération précédente (quatre rapports) et y ont ajouté des éléments. De même que les ingénieurs qui ont réfléchi à la boîte de vitesses à six rapports sont repartis de la version précédente.

Le polymorphisme est une autre notion importante de la programmation orientée objet. Par son intermédiaire, il est possible d'utiliser plusieurs classes de manière interchangeable même si le fonctionnement interne de ces classes est différent. Si vous savez changer de vitesse sur une voiture Peugeot, vous savez également comment le faire sur une voiture Renault et pourtant les deux types de boîtes de vitesses ne sont pas conçus de la même façon.

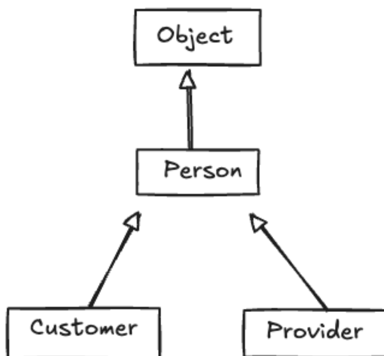
Deux autres concepts sont également associés au polymorphisme. La surcharge et la substitution de méthodes. La surcharge est utilisée pour concevoir dans une classe des méthodes ayant le même nom, mais ayant un nombre de paramètres différent ou des types de paramètres différents. La substitution est utilisée lorsque dans une classe dérivée, on souhaite modifier le fonctionnement d'une méthode dont on a hérité. Le nombre et le type des paramètres restent identiques à ceux définis dans la classe de base.

Voici mis en place les quelques principes de base de la programmation objet. Le sujet de cet ouvrage n'étant pas la mécanique automobile nous allons tout de suite voir la mise en œuvre de ces principes avec le langage Java.

2. Mise en œuvre de la POO avec Java

2.1 Contexte

Dans le reste de ce chapitre, nous allons travailler sur la classe `Person` et ses sous-classes. Cette classe `Person` représentera une personne au sens courant du terme en français. Notez qu'il est généralement recommandé de nommer ses classes en anglais, et nous respecterons cette règle. Vous trouverez ci-dessous une représentation simplifiée en UML (*Unified Modeling Language*) de cette hiérarchie de classes.



© Editions ENI - All rights reserved

UML est un langage graphique dédié à la représentation des concepts de programmation orientée objet. Pour plus d'informations sur ce langage, vous pouvez consulter l'ouvrage UML 2.5 dans la collection Ressources Informatiques des Éditions ENI.

En UML, une flèche triangulaire vide décrit une dérivation. La classe dérivée est la classe de base, mais avec des propriétés additionnelles ou modifiées, comme le concept d'héritage présenté précédemment.

Nous avons la classe `Person` qui hérite de la classe `Object` (comme toute classe en Java). Cette classe `Person` possédera deux classes filles : `Customer` (un client) et `Provider` (un fournisseur).

2.2 Création d'une classe

La création d'une classe passe par la déclaration de la classe elle-même et de tous les éléments la constituant.

2.2.1 Déclaration de la classe

La déclaration d'une classe se fait en utilisant le mot-clé `class` suivi du nom de la classe puis d'un bloc de code délimité par les caractères { et }. Dans ce bloc de code, on trouve des déclarations de variables qui seront les champs de la classe et des fonctions qui seront les méthodes de la classe. Plusieurs mots-clés peuvent être ajoutés pour modifier les caractéristiques de la classe. La syntaxe générale de déclaration d'une classe est donc la suivante :

```
[modificateurs] class NomDeLaClasse
              [extends NomDeLaClasseDeBase]
              [implements NomDeInterface1,NomDeInterface2, ...] {
{
    Code de la classe
}
```

Les signes `[` et `]` sont utilisés ici pour indiquer le caractère optionnel de l'élément. Ils ne doivent pas être utilisés dans le code de déclaration d'une classe.

Les modificateurs permettent de déterminer la visibilité de la classe et la manière de l'utiliser. Voici la liste des modificateurs disponibles :

`public` : indique que la classe peut être utilisée par toutes les autres classes. Sans ce modificateur, la classe ne sera utilisable que par les autres classes faisant partie du même `package`. Pour plus d'informations sur les packages, veuillez vous référer à la section éponyme un peu plus loin dans le chapitre.

`abstract` : indique que la classe est abstraite et ne peut donc pas être instanciée. Elle ne peut être utilisée que comme classe de base dans une relation d'héritage. En général, dans ce genre de classe, seules les déclarations de méthodes sont définies, et il faudra écrire le contenu des méthodes dans les classes dérivées.

`final` : la classe ne peut pas être utilisée comme classe de base dans une relation d'héritage. Autrement dit, il n'est pas possible de la dériver. Elle peut uniquement être instanciée.

La signification des mots-clés `abstract` et `final` étant contradictoire, leur utilisation simultanée est bien sûr interdite.

Pour indiquer que votre classe récupère les caractéristiques d'une autre classe par une relation d'héritage, vous devez utiliser le mot-clé `extends` suivi du nom de la classe de base. Vous pouvez également implémenter dans votre classe une ou plusieurs interfaces en utilisant le mot `implements` suivi de la liste des interfaces implémentées. Ces deux notions seront vues en détail plus loin dans ce chapitre.

Le début de la déclaration de la classe `Person` est donc le suivant :

```
public class Person {

}
```

Ce code doit obligatoirement être saisi dans un fichier ayant le même nom que la classe (`Person`) et l'extension `.java`.

© Editions ENI - All rights reserved

2.2.2 Création des champs

Intéressons-nous maintenant au contenu de la classe. Nous devons créer les différents champs (ou variables membres) de la classe. Pour cela, il suffit de déclarer des variables à l'intérieur du bloc de code de la classe en indiquant la visibilité de la variable, son type et son nom.

```
[private | protected | public] typeDeLaVariable nomDeLaVariable;
```

Il est possible de définir des variables de classe ou des constantes en utilisant les mots-clés `static` et `final`. Veuillez vous référer au chapitre sur les bases du langage pour plus d'informations.

La visibilité de la variable répond aux règles suivantes :

`private` : la variable n'est accessible que dans la classe où elle est déclarée.

`protected` : la variable est accessible dans la classe où elle est déclarée, dans les autres classes faisant partie du même package et dans les classes héritant de la classe où elle est déclarée.

`public` : la variable est accessible à partir de n'importe où.

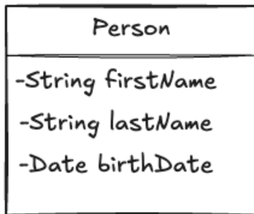
Si aucune information n'est fournie concernant la visibilité, la variable est accessible à partir de la classe où elle est déclarée et des autres classes faisant partie du même package.

Lorsque vous choisissez la visibilité d'une variable, vous devez autant que possible respecter le principe d'encapsulation et limiter au maximum la visibilité des variables. L'idéal étant de toujours avoir des variables `private` ou `protected` mais jamais `public`.

La variable doit également avoir un type. Il n'y a pas de limitation concernant le type d'une variable et vous pouvez donc utiliser aussi bien les **types primitifs** du langage Java, tels que `int`, `float`, `char`, que des **types références** (comme `String`, `LocalDate` ou des objets que vous avez vous-même créés).

Le nom de la variable doit quant à lui simplement respecter les règles de nommage (pas d'utilisation de mot-clé du langage). Veuillez vous référer au chapitre sur les conventions de nommage (cf. chapitre Comprendre un programme - Les conventions de nommage) pour plus d'informations.

On peut représenter la classe `Person` avec ses variables membres en UML de la façon suivante :



La classe est composée de trois variables privées (le – devant les noms de variables indique ce caractère privé). Les variables `firstName` et `lastName` (nom et prénom) sont de type `String` et la variable `birthDate` (date de naissance) est de type `Date`. Le type `Date` est un type UML standard. Lors de la transposition en code, il est possible d'utiliser un type différent adapté au langage utilisé.

La classe `Person` prend donc maintenant la forme suivante :

```
public class Person {
    private String firstName;
    private String lastName;
    private LocalDate birthDate;
}
```

2.2.3 Création des méthodes

Les méthodes sont simplement des fonctions définies à l'intérieur d'une classe. Elles sont en général utilisées pour manipuler les champs de la classe. La syntaxe générale de déclaration d'une méthode est décrite ci-dessous.

```
[modificateurs] typeRetour nomMethode ([listeParametres])
                                      [throws listeException] {

}
```

© Editions ENI - All rights reserved

Les modificateurs de visibilité suivants sont disponibles :

`private` : indique que la méthode ne peut être utilisée que dans la classe où elle est définie.

`protected` : indique que la méthode peut être utilisée dans la classe où elle est définie, dans les sous-classes de cette classe et dans les classes faisant partie du même package.

`public` : indique que la méthode peut être utilisée depuis n'importe quelle autre classe.

Si aucun de ces mots-clés n'est utilisé, alors la visibilité sera limitée au package dans lequel la classe est définie.

Des modificateurs complémentaires sont disponibles.

`static` : indique que la méthode est une méthode de classe.

`abstract` : indique que la méthode est abstraite et qu'elle ne contient pas de code. La classe où elle est définie doit elle aussi être abstraite.

`final` : indique que la méthode ne peut pas être substituée dans une sous-classe.

`native` : indique que le code de la méthode se trouve dans un fichier externe écrit dans un autre langage.

`synchronized` : indique que la méthode ne peut être exécutée que par un seul thread à la fois.

Le type de retour peut être n'importe quel type de données, type valeur ou type référence. Si la méthode n'a pas d'information à renvoyer, vous devez utiliser le mot-clé `void` en remplacement du type de retour.

La liste des paramètres est identique à une liste de déclaration de variables. Il faut spécifier le type du paramètre et le nom du paramètre. Si plusieurs paramètres sont attendus, il faut séparer leur déclaration par une virgule. Même si aucun paramètre n'est attendu, les parenthèses sont tout de même obligatoires.

Chapitre 3
Les conditionnelles

1. Présentation

Jusqu'à présent, les algorithmes présentés sont complètement linéaires, c'est-à-dire que les instructions sont toutes exécutées dans l'ordre. Les structures de contrôle permettent, suivant les cas, de ne pas exécuter les mêmes instructions. Pour reprendre l'exemple de l'algorithme de calcul du prix TTC, il serait possible de demander à l'utilisateur si le produit a le droit à un taux de TVA réduit et, seulement si la réponse est "oui", de demander à l'utilisateur le taux de TVA à appliquer.

2. La structure de contrôle Si (forme simple)

La première possibilité pour n'effectuer des instructions que dans certaines situations est d'utiliser la structure de contrôle `Si`. Voici la syntaxe à utiliser pour ce test :

```
Si conditionBooléenne Alors
 SuiteDInstructions
FSi
```

Après le `Si`, il faut indiquer la condition qui doit être remplie pour exécuter la suite d'instructions comprise entre les mots-clés `Alors` et `FSi`. Le mot-clé `FSi` correspond à "Fin du Si". La condition doit être de type booléen : elle doit avoir pour valeur soit vrai soit faux. Si la condition a pour valeur vrai, alors l'ensemble de la suite des instructions sont exécutées, sinon elles sont sautées et l'exécution continue avec les instructions situées après le `FSi`.

Exemple :

```
Algo Tva2
# calcule le prix TTC d'un article
Variable prixHT : réel
Variable tvaReduite : texte
Variable tva : réel <- 20/100
Début
   écrire("Prix HT de l'article ?")
   prixHT <- saisir()
   écrire("Ce produit bénéficie-t-il d'un taux de TVA réduit ?")
   tvaReduite <- saisir()
   Si tvaReduite = "oui" Alors
     écrire("Quel est le taux (%) ?")
     tva  <- saisir()/100
   FSi
   écrire("Prix TTC de l'article : " & prixHT×(1+tva) & "€")
Fin
```

Dans cet exemple, la condition est `tvaReduite = "oui"`. Cette expression a bien une valeur booléenne puisque l'opérateur de comparaison égal donne comme résultat un booléen (cf. chapitre précédent). Les instructions comprises entre `Alors` et `FSi` ne sont exécutées qu'à la condition que la variable `tvaReduite` vaille "oui". Si elle vaut une autre valeur, alors ces instructions seront sautées et l'exécution se poursuivra avec l'affichage du prix TTC de l'article.

En Java, il faut utiliser le mot-clé `if` pour effectuer l'équivalent. La condition est positionnée entre un couple de parenthèses, et les instructions qui ne sont exécutées que si la condition est vraie sont positionnées dans un bloc d'instructions. Un bloc est matérialisé en plaçant les instructions entre un couple d'accolades.

© Editions ENI - All rights reserved

Exemple :

```
import java.util.Scanner;

public class Tva2 {
 public static void main(String[] args) {
   double tva = 20.0 / 100;
   double prixHT;
   Scanner console = new Scanner(System.in);
   System.out.println("Prix HT de l'article ?");
   prixHT = console.nextDouble();
   console.nextLine();
   System.out.println(
         "Ce produit bénéficie-t-il d'un taux de TVA réduit ?");
   String tvaReduite = console.nextLine();
   if(tvaReduite.equals("oui")) {
     System.out.println("Quel est le taux (%) ?");
     tva = console.nextDouble()/100;
     console.nextLine();
   }
   System.out.printf("Prix TTC de l'article : %.2f€%n",
                     prixHT * (1 + tva));
   console.close();
 }
}
```

Une variable qui est déclarée est utilisable uniquement entre le moment où elle est déclarée et la fin du bloc dans lequel elle a été déclarée. C'est ce qui se nomme la portée d'une variable. Si vous déclarez une variable au sein du bloc d'instructions, elle n'existera plus à la fin de celui-ci.

En Java, il ne faut pas utiliser les opérateurs == et != pour tester l'égalité de deux variables de type `String`. Il faut utiliser la méthode `equals()`. Dans l'exemple précédent, la condition pour comparer la valeur saisie avec le texte "oui" s'écrit donc :

```
tvaReduite.equals("oui")
```

S'il n'y a qu'une seule instruction qui ne doit être exécutée de manière conditionnelle, il est possible d'omettre les accolades.

Exemple :

```
if(tva < 5.5)
 System.out.println ("Taux de TVA très réduit");
```

3. La structure de contrôle Si (forme double)

La structure de contrôle `Si` peut se présenter dans une forme double. Dans ce cas-là, en plus des éléments présents dans sa version simple, s'ajoute après le mot-clé `Sinon` la suite d'instructions à effectuer lorsque la condition est fausse.Voici donc la syntaxe :

```
Si conditionBooléenne Alors
 suiteDInstructions
Sinon
 autreSuiteDInstructions
FSi
```

Si la condition est vraie, alors la suite d'instructions comprises entre `Alors` et `Sinon` est exécutée et l'exécution se poursuit après le `FSi`. Si par contre la condition est fausse, la première suite d'instructions est sautée et les instructions qui sont exécutées sont celles comprises entre `Sinon` et `FSi`, puis l'exécution se poursuit après le `FSi`.

Exemple :

```
Algo Age
Variable age : entier
Début
 age <- saisir("Quel est votre âge ? ")
 Si age ≥ 18 Alors
   écrire("Vous êtes majeur !")
 Sinon
   écrire("Vous êtes mineur !")
 FSi
Fin
```

Dans cet exemple, suivant la valeur saisie par l'utilisateur, le message "Vous êtes majeur !" ou "Vous êtes mineur !" s'affiche. En aucun cas les deux messages ne peuvent s'afficher.

© Editions ENI - All rights reserved

En Java, voici la syntaxe équivalente :

```
if(conditionBooléenne) {
  suiteDInstructions
} else {
 autreSuiteDInstructions
}
```

Voici donc le code Java correspondant à l'algorithme d'exemple précédemment présenté :

```
import java.util.Scanner;

public class Age {

   public static void main(String[] args) {
         Scanner console = new Scanner(System.in);
         System.out.println("Quel est votre âge ?");
         int age = console.nextInt();
         if(age>=18) {
                System.out.println("Vous êtes majeur !");
         } else {
                System.out.println("Vous êtes mineur !");
         }
         console.close();
   }
}
```

4. L'imbrication des structures de contrôle

Les structures de contrôle peuvent être imbriquées les unes dans les autres. C'est-à-dire que parmi la suite d'instructions contenues entre `Alors` et `Sinon` ou celles contenues entre `Sinon` et `FSi`, il est possible de positionner une autre structure de contrôle.

Exemple :

```
Algo Age2
Variable age : entier
Début
 age <- saisir("Quel est votre âge ? ")
 Si age < 0 Alors
   écrire("Ce n'est pas possible !")
```

```
 Sinon
   Si age ≥ 18 Alors
     écrire("Vous êtes majeur !")
   Sinon
     écrire("Vous êtes mineur !")
   FSi
 FSi
Fin
```

Dans cet exemple, il y a deux structures de contrôle imbriquées. Il y a tout d'abord la structure de contrôle `Si` englobante indiquée en gras ci-dessous et ensuite le `Si` englobé indiqué en italique ci-dessous :

```
Algo age2
Variable age : entier
Début
 age <- saisir("Quel est votre âge ? ")
 Si age < 0 Alors
   écrire("Ce n'est pas possible !")
 Sinon
   Si age ≥ 18 Alors
     écrire("Vous êtes majeur !")
   Sinon
     écrire("Vous êtes mineur !")
   FSi
 FSi
Fin
```

Si l'âge est négatif, alors le message "Ce n'est pas possible !" est affiché puis les instructions jusqu'au `FSi` en gras sont sautées. Si l'âge n'est pas négatif, alors un nouveau test est effectué pour savoir si celui-ci est plus grand ou égal à 18. Si c'est le cas, le message "Vous êtes majeur !" est affiché, sinon c'est le message "Vous êtes mineur !" qui est affiché.

En Java, il est bien évidemment possible également d'imbriquer les structures de contrôle. Voici l'algorithme précédent codé en Java :

```
import java.util.Scanner;

public class Age2 {
   public static void main(String[] args) {
         Scanner console = new Scanner(System.in);
         System.out.println("Quel est votre âge ?");
         int age = console.nextInt();
```

© Editions ENI - All rights reserved

```
            if (age < 0) {
                   System.out.println("Ce n'est pas possible");
            } else {
                   if (age >= 18) {
                         System.out.println("Vous êtes majeur !");
                   } else {
                         System.out.println("Vous êtes mineur !");
                   }
            }
            console.close();
      }
}
```

Il est à noter qu'après le premier `else`, le bloc ne contient que la seconde structure de contrôle `if`, il n'est donc pas obligatoire de mettre les accolades autour de celle-ci. Cela permet de rendre le code un peu plus lisible :

```
import java.util.Scanner;

public class Age3 {
   public static void main(String[] args) {
         Scanner console = new Scanner(System.in);
         System.out.println("Quel est votre âge ?");
         int age = console.nextInt();
         if (age < 0) {
               System.out.println("Ce n'est pas possible");
         } else if (age >= 18) {
               System.out.println("Vous êtes majeur !");
         } else {
               System.out.println("Vous êtes mineur !");
         }
         console.close();
   }
}
```

Remarque

Dans cet exemple, une structure de contrôle `Si` a été imbriquée au sein d'une autre structure de contrôle `Si`. Pour le moment, c'est la seule structure de contrôle qui a été présentée, mais il en existe beaucoup d'autres. Il est à noter qu'il est également possible d'imbriquer n'importe quelle structure de contrôle au sein de n'importe quelle structure de contrôle.

5. La structure de contrôle Selon

La structure de contrôle `Selon` permet également de n'effectuer que certaines instructions de manière conditionnelle. `Selon` permet d'effectuer les instructions souhaitées en fonction de la valeur d'une variable. Voici la syntaxe de cette structure de contrôle :

```
Selon nomDeLaVariable
    cas premièreListeDeValeurs : premièreSuiteDInstructions
    cas deuxièmeListeDeValeurs : deuxièmeSuiteDInstructions
    cas troisièmeListeDeValeurs : troisièmeSuiteDInstructions
...
    autre : autreSuiteDInstructions
FSelon
```

Si la valeur de la variable fait partie de la première liste de valeurs, alors c'est la première suite d'instructions qui est exécutée, puis l'exécution reprend après `FSelon`. Si ce n'est pas le cas, la valeur de la variable est recherchée dans la deuxième liste de valeurs. Si elle est présente, alors la deuxième suite d'instructions est exécutée, puis l'exécution reprend après `FSelon`. Cela est réalisé jusqu'à trouver le cas contenant dans sa liste la valeur de la variable.

Le cas `autre` est facultatif. Si celui-ci est présent et si la valeur de la variable n'a été trouvée dans aucun cas, ce sont les instructions situées après `autre` qui sont exécutées.

Exemple :

```
Algo TriSelectif
Variable dechet : texte
Début
 dechet <- saisir("Que souhaitez-vous jeter ?")
 Selon dechet
   cas "papier"; "carton"; "boîte de conserve" :
     écrire("à recycler")
   cas "végétaux"; "épluchures" :
     écrire("à composter")
   autre :
     écrire("à mettre à la poubelle")
 FSelon
Fin
```

© Editions ENI - All rights reserved