UNIVERSITÉ Grenoble Alpes

Initiation à l'informatique et à l'algorithmique (LICENCE MIASHS 1)

8. Actions et fonctions

Parfois, des parties d'un programme se répètent. Par exemple, si on veut lire plusieurs entiers entre 1 et 100, c'est dommage de recopier le même code qui vérifie que l'entier est bien dans l'intervalle attendu. Outre le fait que le programme deviendrait lourd et illisible, sa maintenance serait délicate puisque si l'on voulait modifier ce code, il faudra le faire à différents endroits, ce qui augmenterait le risque d'erreurs.

action

On va donc isoler les parties du programme qui se répètent pour en faire une *action* (appelé aussi une *procédure*) ou une *fonction*. En java, les deux sont aussi appelées *méthodes*. Prenons un exemple. Supposons que nous ayons un programme qui affiche souvent une ligne d'astérisques, par exemple pour mettre en valeur un message. Comme ceci :

```
**********
Le résultat est 144
**********
```

On définit alors une action qui s'appelle, par exemple, afficheAsterisque() et qui s'écrit comme ceci (notez l'indentation, comme d'habitude) :

```
static void afficheAsterisque ()
        System.out.println("**********");
}
```

Les mot-clés *static* et *void* seront expliqués plus tard. Ce code sera placé au début du programme, dans la classe mais juste avant le *main*. Pour faire exécuter cette action (on dit aussi pour *appeler* cette action), il suffit d'écrire dans le programme principal :

```
afficheAsterisque();
```

Voici un exemple d'un programme qui définit cette action et qui l'appelle deux fois. Repérez bien la définition de l'action (lignes 4 à 6) et son utilisation (lignes 9 et 11).

```
import java.util.Scanner;
public class ExempleAction {

   static void afficheAsterisque() {
        System.out.println("**********");
   }

   public static void main(String [] args) {
        afficheAsterisque();
        System.out.println("Bonjour");
        afficheAsterisque();
   }
}
```

On peut aussi *paramétrer* les actions, ce qui permet de faire en sorte que leur code diffère en fonction des paramètres qu'on leur fournit. Par exemple, on peut vouloir paramétrer le nombre d'astérisques à afficher, au lieu d'en afficher toujours le même nombre. Notre action aura donc un paramètre qui est le nombre d'astérisques à afficher. Cela s'écrit donc comme ceci, en mettant le type et le nom des paramètres après le nom de l'action :

paramètre

```
   static void afficheAsterisque(int nb) {
        // affiche une ligne de nb astérisques
        int i=1;
        while (i<=nb)  {
           System.out.print("*");
           i++;
        }
        System.out.println();
   }
```

Si on veut afficher 10 astérisques, on appelera l'action comme ceci :

```
afficheAsterisques(10);
```

On peut aussi appeler l'action avec une variable. Par exemple :

```
x = 10;
afficheAsterisques(x);
```

ou encore

```
        int val;
        System.out.println("Combien d'astérisques ?");
        val = s.nextInt();
        afficheAsterisques(val);
```

La variable de l'action (ici, nb) est appelée le *paramètre formel*. La variable de l'appel (val dans le dernier exemple) se nomme le *paramètre effectif*. C'est important que vous compreniez bien la différence entre les deux et le fait qu'il n'est pas nécessaire qu'ils portent le même nom.

On peut utiliser plusieurs paramètres, en les séparant par des virgules. Voici un exemple dans lequel on ajoute un paramètre qui est le caractère à afficher.

```
import java.util.Scanner;
public class ExempleAction {

   static void afficheAsterisque(int nb, char c) {
        int i=1;
        while (i<=nb)  {
           System.out.print(c);
           i++;
        }
        System.out.println();
   }

   public static void main(String [] args) {
        afficheAsterisque(20,'=');
        System.out.println("Bonjour");
        afficheAsterisque(10,'*');
   }
}
```

Parfois, le code qui se répète a pour objet de calculer une valeur. On a donc besoin de récupérer cette valeur. On utilise alors le mot-clé *return* pour retourner cette valeur au programme appelant. On définit alors une *fonction* et non plus une *action*. Prenons un exemple : il s'agit d'un programme qui fait deviner un nombre entre 1 et 100.

fonction

```
...
System.out.println("Premier joueur, c'est votre tour.");
do {
        System.out.println("Entrez un nombre entre 1 et 100");
        nombreADeviner=s.nextInt();
}
while ((nombreADeviner<1) || (nombreADeviner>100));
do {
        System.out.println("Second joueur, c'est votre tour.");
        do {
                System.out.println("Entrez un nombre entre 1 et 100");
                proposition=s.nextInt();
        }
        while ((proposition<1) || (proposition>100));
        if (proposition < nombreADeviner)
                System.out.println("C'est plus grand.");
        else if (proposition > nombreADeviner)
                System.out.println("C'est plus petit.");
} while (proposition != nombreADeviner);
```

On voit que deux parties du programme sont similaires. Elles consistent à lire un entier entre 1 et 100. On va donc les isoler pour en faire une fonction qui lira un entier et *retournera* (on dit aussi *renvoyer*) la valeur lue. Cette fonction est placée au même niveau que le *main*.

```
import java.util.Scanner;
public class Deviner2 {

   static int lireEntier() {
        // Lecture d'un entier entre 1 et 100
        int y;
        Scanner s = new Scanner(System.in);
        do {
           System.out.println("Entrez un nombre entre 1 et 100");
           y=s.nextInt();
        }
        while ((y<1) || (y>100));
        return y;
   }

   public static void main(String [] args) {
        int nombreADeviner,proposition;
        System.out.println("Premier joueur, c'est votre tour.");
        nombreADeviner=lireEntier();
        do {
           System.out.println("Second joueur, c'est votre tour.");
           proposition=lireEntier();
           if (proposition < nombreADeviner)
                System.out.println("C'est plus grand.");
```

```
            else if (proposition > nombreADeviner)
                System.out.println("C'est plus petit.");
        } while (proposition != nombreADeviner);
    }
}
```

On voit que le programme principal est maintenant plus concis, plus lisible. Grâce à son nom, on voit bien également le rôle de la fonction. Il faut aussi indiquer le type de la valeur qui est retournée par la fonction (ici `int`). Vous comprenez maintenant pourquoi, pour une action qui ne retourne pas de valeur, on lui indique le mot-clé *void* (vide, en anglais). Notez aussi le mot-clé `return` qui permet de renvoyer la valeur au programme principal. A ce niveau, on peut alors le récupérer dans une variable. Une action suit donc ce schéma :

```
// définition de l'action
static void monAction (TYPE VAR, TYPE VAR, ...) {
        ...
}

//appel de l'action
monAction(VALEUR, VALEUR, ...);
```

Une fonction suit ce schéma :

```
// définition de la fonction
static TYPE_FONCTION maFonction (TYPE VAR, TYPE VAR) {
        ...
        return VALEUR;
}

//appel de la fonction
TYPE_FONCTION y;
y=maFonction(VALEUR, VALEUR, ...)
```

La variable qui récupère la valeur de la fonction doit avoir le même type que la fonction.

Voici quelques fonctions prédéfinies en Java :

- `Math.Random()` : retourne un nombre entre 0 et 1. C'est une fonction sans paramètres.
- `Math.sqrt(x)` : retourne la racine carrée de x. Elle a un paramètre de type double. Comme il n'y a pas de perte d'information à transformer un `int`, un short, etc. en double, il n'est pas nécessaire d'utiliser un cast si le paramètre est entier ou float, la conversion se faisant automatiquement par Java.
- `Math.pow(x,y)` : retourne x^y. Attention à l'ordre des paramètres.

Cette dernière fonction existe déjà en Java, mais on peut essayer de la réécrire. Cela s'écrirait par exemple comme ceci :

```
import java.util.Scanner;
public class Puissance {

    static double puissance(double a,double b) {
        // calcule a puissance b
        int i=1;
        double res=1;
```

```
        while (i<=b) {
            res=res*a;
            b--;
        }
        return(res);
    }

    public static void main(String[] args) {
        System.out.println(puissance(4,5));
    }
}
```

Voici un autre exemple avec des chaînes de caractères : il s'agit d'écrire une fonction qui nous indique si une chaîne est préfixe d'une autre. Par exemple, "MI" est préfixe de "MIASHS". Cela nous donne l'occasion d'utiliser un nouveau type prédéfini en Java : le booléen. Un booléen n'a que deux valeurs : vrai ou faux (appelées true et false). On peut, par exemple, écrire :

booléen

```
boolean b;
b=false;
```

ou

```
boolean majeur;
if (age<18)
        majeur=false;
else
        majeur=true;
```

qui peut s'écrire plus simplement comme ceci :

```
        boolean majeur;
        majeur=(age>=18);
```

Voici notre exemple :

```
import java.util.Scanner;
public class Prefixe {

    static boolean prefixe(String a,String b) {
        // retourne vrai si a est prefixe de b
        return (a.equals(b.substring(0,a.length())));
    }

    public static void main(String[] args) {
        Scanner s=new Scanner(System.in);
        String ch1,ch2;
        System.out.println("Vérification de préfixe. Entrez une chaine principale.");
        ch1=s.nextLine();
        System.out.println("Entrez une seconde chaine");
        ch2=s.nextLine();
        if (prefixe(ch2,ch1))
            System.out.println(ch2+" est préfixe de "+ch1);
        else
            System.out.println(ch2+" n'est pas préfixe de "+ch1);
    }
}
```

Exercice : qu'affiche ce programme ?

```
import java.util.Scanner;
public class ExerciceFonctions {

   static boolean f1(int a,int b) {
        return ((a>b) && (a<3));
   }

   static int f2(boolean a, int x,int y) {
        if (a && f1(x,y))
           return(10);
        else
           return(5);
   }

   public static void main(String[] args) {
        System.out.println(f2(true,4,3));
        System.out.println(f2(f1(2,1),1,0));
        System.out.println(f2(true,f2(false,2,2),f2(true,2,0)));

   }
}
```

Réponse : 5, 10 et 5

Étudions quelques exemples d'erreurs que peut vous signaler Java. Voici un programme. Essayer de trouver l'erreur.

```
import java.util.Scanner;
public class ExempleErreur {

   static int tripler(int x) {
        return (3*x);
    }

   public static void main (String [] args) {
        float f=3;
        System.out.println(tripler(f));
   }
}
```

On obtient une erreur parce qu'on appelle la fonction avec un float alors qu'elle attend un entier. Il y a donc un risque de perte d'informations. Java nous dit :

```
ExempleErreur.java:10: tripler(int) in ExempleErreur cannot be applied to (float)
```

Il suffit juste de lui dire que l'on sait ce que l'on fait et de convertir explicitement en entier :

```
System.out.println(tripler((int)f));
```

Même chose avec cet exemple. Essayez de trouver l'erreur...

```
import java.util.Scanner;
public class ExempleErreur {

   static int f(int x) {
        return (Math.sqrt(x));
   }

   public static void main (String [] args) {
        System.out.println(f(16));
   }
}
```

L'erreur provient du fait que Math.sqrt(x) retourne un double et que la fonction retourne un entier. Il y a un risque de perte de précision que Java indique de cette façon : `ExempleErreur.java:5: possible loss of precision`

Visibilité des variables

Dès que l'on écrit des fonctions ou des actions, il faut comprendre que les variables ne sont pas connues de tous les endroits du programme : les variables définies dans une fonction ou une action ne sont pas connues en dehors ! Reprenons l'exemple de la fonction puissance :

```
import java.util.Scanner;
public class Puissance {
    static double puissance(double a,double b) {
        // calcule a puissance b
        int i=1;
        double res=1;
        while (i<=b) {
           res=res*a;
           i++;
        }
        return(res);
   }

   public static void main(String[] args) { // ce programme provoque une erreur !
        System.out.println(puissance(4,5));
        System.out.println("i="+i);
   }
}
```

La variable n'est connue que dans la fonction. Ce programme provoque donc l'erreur suivante : `Puissance.java:17: cannot find symbol - symbol  : variable i`

L'inverse est également vrai. Une variable définie dans le main n'est pas connue dans les fonctions ou actions. Par exemple, ce programme produit une erreur parce que la variable valeur n'est pas connue dans la fonction ajouteDix().

```
import java.util.Scanner;
public class ExempleErreur {

   static int ajouteDix(int x) {
        return (x+valeur);
}

   public static void main (String [] args) { // ce programme est erroné !
        int valeur=10;
        System.out.println(ajouteDix(4));
   }
}
```

Si on veut utiliser la variable valeur dans la fonction, il faut la passer en paramètre :

```
import java.util.Scanner;
public class ExempleErreur {

   static int ajouteDix(int x,int valeur) {
        return (x+valeur);
}

   public static void main (String [] args) {
        int valeur=10;
        System.out.println(ajouteDix(4,valeur));
   }
}
```

Notez qu'il n'est pas nécessaire que le paramètre formel de la fonction ait le même nom que le paramètre effectif du main().

9. Parcours et recherche dans des séquences

On a souvent besoin de considérer des séquences en informatique : une chaîne de caractères est une séquence de lettres, un fichier sur le disque dur est une séquence de caractères, un tableau est une séquence d'éléments, etc.

Tableaux

tableaux

Nous allons justement commencer par présenter la notion de tableau. Un tableau est un regroupement de variables de même type. Chaque variable est repérée par un indice, exactement comme les caractères des chaînes sont repérés par leur indice dans la chaîne. Ainsi si tab est un tableau d'entiers qui contient 100 valeurs, l'entier à l'indice 33 est tab[33]. On pourra aussi modifier cette valeur avec tab[33]=-12 par exemple.

Pour déclarer un tableau, il y a deux méthodes :

- soit on connaît toutes les valeurs du tableau et on utilise cette déclaration

```
<type> [ ] <nom du tableau> = {<valeur1>, <valeur 2>, ..., <valeurn>} ;
```

Par exemple :

```
char [] voyelles = {'a','e','i','o','u','y'} ;
float [] monTableau = {6.7, 5.0, 1.0, 9.5, 11.1} ;
```

- soit on connaît juste le nombre d'éléments du tableau et on le remplira dans la suite du programmme. On déclare alors un tableau vide comme ceci :

```
<type> [ ] <nom du tableau> = new <type> [<nombre d'éléments>] ;
```

Par exemple :

```
String[] noms = new String[50] ;
```

Voici un exemple de manipulation des tableaux. Il s'agit de l'implémentation du crible d'Eratosthène qui permet de déterminer les nombres premiers. Le principe est le suivant : on écrit tous les entiers à partir de 2 (jusque 1000 par exemple). On considère le premier de la liste (2) et on barre tous ses multiples (4,6,8,10,12...) :

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17...

Le suivant non barré est forcément premier (3) et on barre tous ses multiples (6,9,12,15...) : 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17...

Le suivant non barré est forcément premier (5) et on barre tous ses multiples...

```
public class NombresPremiers {
   public static void main(String [] args) {
        // CRIBRE D'ERATOSTHENE
        final int NMAX=1000;
        boolean[] t=new boolean[NMAX+1];
        int i,nb;
        // au début, personne n'est barré
        for(i=1;i<=NMAX;i++)
           t[i]=true;

        // on passe en revue tous les nombres
        nb=2;
        while (nb<Math.sqrt(NMAX)) {
           // on barre les multiples de nb
           i=2*nb; // donc on commence par le suivant
           while(i<NMAX+1) {
                t[i]=false;
                i+=nb;
           }
           nb++;  // nb est premier, on passe au suivant
           while (t[nb]==false)
                nb++;
        }

        // AFFICHAGE
        i=1;
        while (i<=NMAX) {
           if (t[i])
                   System.out.print(i+"-");
           i++;
        }
   }
```

Les cases d'un tableau peuvent également contenir chacune... un tableau. On parle alors de tableaux à deux dimensions, ce qui est très utile pour représenter, par exemple, des matrices. Il suffit juste d'ajouter un nouvel indice entre crochets à la suite du premier. On définira un tableau 2D comme ceci par exemple :

tableaux 2D

```
float[][] tab = new float [20][30];
```

et on accèdera à chaque élément par tab[i][j]. Le nombre de cases sur la première dimension est `tab.length`. Le nombre de cases sur la deuxième dimension est la taille de chacun des tableaux qui sont dans les cases de tab, donc notamment la taille du tableau qui est dans la première case : `tab[0].length`. Voici par exemple une fonction qui calcule le produit de deux matrices.

```
public static int[][] mult(int[][]m1,int[][]m2) {
  // Retourne le produit des deux matrices m1 et m2.
  int[][]res=new int[m1.length][m2[0].length];  // tableau résultat
  for(int i=0;i<m1.length;i++) {
    for(int j=0;j<m2[0].length;j++) {
       res[i][j]=0;
       for(int k=0;k<m2.length;k++)
          res[i][j]+=m1[i][k]*m2[k][j];
    }
  }
  return(res);
}
```

Maintenant que nous connaissons les séquences de caractères (String) et les tableaux, on peut étudier la manière de les traiter. On distingue deux grandes opérations sur les séquences : la *parcourir* en entier pour appliquer un traitement à tous les éléments ou *chercher* un élément ayant une certaine propriété en s'arrêtant dès qu'on l'a trouvé. On parle donc de *parcours* et de *recherche*.

Voici des exemples de *parcours* : mettre toutes les lettres d'un mot en minuscule, compter le nombre de fois que le mot "MIASHS" apparaît dans un fichier texte, afficher tous les numéros de téléphone se terminant par 97, remplacer tous les espaces doubles par un seul, etc.

Voici des exemples de *recherche* : déterminer si un fichier texte comporte une ligne vide, chercher le nom d'une personne connaissant son numéro de téléphone, vérifier si "Alan Turing" est dans la liste des étudiants de la licence MIASHS, indiquer si un fichier contient au moins 3 fois le mot "BSHM".

Pour écrire des schémas d'algorithme de parcours et de recherche, on va distinguer quatre éléments :

- *ElémentCourant* représente l'élément courant de la séquence
- *FindeSéquence* représente l'expression logique qui caractérise la fin de la séquence
- *Démarrer* représente la partie algorithmique consistant à se positionner sur le premier élément de la séquence
- *Avancer* représente la partie algorithmique consistant à passer de l'élément courant au l'élément suivant .

9.1 Schéma général de parcours

```
InitialisationDuTraitement
while (! FindeSéquence) {
        Traitement ElementCourant
        Avancer
}
```

On utilise une boucle while parce qu'il ne faut pas entrer dans la boucle si la séquence est vide.

Par exemple, si on veut doubler tous les caractères d'une chaîne (afficher MMIIAASSHHSS si l'utilisateur entre MIASHS), il s'agit bien d'un parcours. Voici la solution :

```
import java.util.Scanner;
public class DoublerLesCaracteres {

   public static void main(String[] args) {
        Scanner s=new Scanner(System.in);
        String ch;
        System.out.println("Doublage de caractères. Entrez une chaîne principale.");
        ch=s.nextLine();
        int i=0;  // Démarrer
        while (!(i>=ch.length())) {   // FindeSéquence
           System.out.print(""+ch.charAt(i)+ch.charAt(i));// Trait. ElementCourant
           i++;   // Avancer
        }
   }
}
```

La condition `(!(i>=ch.length()))` pourrait être écrite de manière plus simple : `(i<ch.length())`, mais l'objectif était d'utiliser le schéma général.

9.2 Schéma avec traitement séparé de la séquence vide

Parfois, on a besoin de séparer le traitement de la séquence vide. Par exemple, si on veut faire la somme d'une séquence de nombres, on ne peut rien répondre si la séquence est vide, même pas 0 ! Le schéma est le suivant :

```
if (SéquenceVide) {
        Traitement SéquenceVide
}
else {
        InitialisationDuTraitement
        do
                Traitement ElementCourant
                Avancer
        } while (! FindeSéquence)
}
```

On a pu utiliser une boucle do-while puisque la chaîne contient au moins un élément.

9.3 Schéma général de recherche

Il s'agit de déterminer si, dans la séquence, un élément vérifie une propriété P. Deux cas se présentent : l'élément est présent ou l'élément n'est pas présent. On a donc deux raisons d'arrêter la recherche : on a trouvé l'élément ou on est parvenu à la fin de la séquence. La condition de la boucle est donc double :

```
while (! FindeSéquence) && (! P (EC)) {
        Avancer
}
if (! FindeSéquence) {// élément trouvé
        …
}
else {    // élément non trouvé
        …
}
```

On remarque que l'on ne fait aucun traitement de l'élément courant, contrairement au parcours, ce qui n'empêche pas qu'il y ait d'autres traitements dans la boucle.
Prenons un exemple. On veut écrire un programme qui, étant donné une chaîne de caractères et un caractère, déterminer si le caractère est présent dans la chaîne. Il n'est pas nécessaire de tout parcourir : dès qu'on a trouvé l'élément on arrête. Mais comme on n'est pas certain qu'il est présent, on est obligé d'utiliser les deux conditions.

```
import java.util.Scanner;
public class ChercherOccurenceCar {

   public static void main(String [] args) {
        Scanner s= new Scanner(System.in);
        String ch,ch1;
        char c;
        int i;
        System.out.print("Entrez la chaîne : ");
        ch=s.nextLine();
        System.out.print("Entrez le caractère : ");
        ch1=s.nextLine();
        c=ch1.charAt(0);
        i=0;
        while((i<ch.length())&&(c!=ch.charAt(i))) // (!FindeSéquence) && (!P (EC))
           i++;   // Avancer
        if (i<ch.length())  // (! FinDeSéquence)
           System.out.println(c+" est présent dans la chaîne.");
        else
           System.out.println(c+" n'est pas présent dans la chaîne.");
   }
}
```

Prenons un autre exemple : on veut écrire un programme qui lit une chaîne composée d'un prénom et d'un nom séparés par un ou plusieurs espaces. Par exemple : "Gérard Menvussa"...
Il nous faut donc chercher le premier espace puis chercher la première lettre du nom. Il y a donc deux recherches successives.

```
public class Initiales {

   public static void main(String [] args) {
        Scanner s= new Scanner(System.in);
        String ch,ch1;
        char initialePrenom, initialeNom;
        int i;
        System.out.print("Nom et prénom séparés par un ou plusieurs espaces : ");
        ch=s.nextLine();
        if (ch.length() == 0)
           System.out.println("La chaîne est vide !");
        else {
           initialePrenom=ch.charAt(0);
           i=0;
           while ((i<ch.length()) && (ch.charAt(i)!=' '))
                i++;
           if (i<ch.length()) {   // on a trouvé l'espace
                   while ((i<ch.length()) && (ch.charAt(i)==' '))
                        i++;
                   if (i<ch.length())  {
                        initialeNom=ch.charAt(i);
                        System.out.println(initialePrenom+"."+initialeNom+".");
                   }
                   else
                        System.out.println("Il n'y a pas de nom !");
           }
           else
                System.out.println("Il n'y a pas d'espace !");
        }
   }
}
```

9.4 Les fichiers texte

On étudie maintenant un autre type de séquences : les fichiers texte. Il est possible de lire les informations qui se trouvent dans un fichier ou d'écrire dans un fichier pour y stocker des informations de manière pérenne.

Le mode d'accès aux fichiers est *séquentiel*. On ne peut pas accéder à un élément au milieu d'un fichier, il faut pour cela avoir parcouru tout ce qui précède. De la même façon, on ne peut ajouter des éléments qu'à la fin d'un fichier, on ne peut pas écrire au milieu d'un fichier.

Lecture dans un fichier texte

Lecture de fichier

La *lecture* d'un fichier se fait en définissant une variable de type BufferedReader. Il faut ensuite se positionner sur le début du fichier (*Démarrer* dans les schémas précédents) et lire ligne à ligne. L'opération *Avancer* est effectuée avec la fonction readLine() qui retourne la chaîne de caractère suivante dans le fichier. La condition de fin de fichier (*FinDeSéquence* dans les schémas) est obtenue lorsque la chaîne que l'on lit vaut *null*. Il faut aussi penser à fermer le fichier à la fin.

Le schéma de parcours dans un fichier est donc le suivant (le schéma générique, à connaître par cœur, est à droite) :

```
BufferedReader fr;
String chaineLue;
fr=newBufferedReader(new FileReader("<fichier>"));
chaineLue=fr.readLine();
while (chaineLue != null) {
    ...
    chaineLue=fr.readLine();
}
fr.close();
```

```
InitialisationDuTraitement
while (! FindeSéquence) {
        Traitement
ElementCourant
        Avancer
}
TerminaisonDuTraitement
```

Pour utiliser les fichiers en Java, il y a deux choses importantes :

1. il faut ajouter au début du programme l'instruction : `import java.io.*;`
2. il faut indiquer à java qui gère les éventuelles erreurs (fichier inexistant, par exemple). On verra en deuxième année comment gérer nous-mêmes les erreurs. En attendant, on laisse à Java le soin de le faire en ajoutant à la fin de la ligne du main cette instruction : `throws IOException`

Par exemple, on veut compter combien de lignes d'un fichier commencent par #.

```
// Compte combien de lignes d'un fichier commencent par un #
import java.io.*;
import java.util.Scanner;
public class CompterLesLignesCommencantPar {
   public static void main(String [] arg) throws IOException {
        BufferedReader f;
        String chaineLue,nomFichier;
        int nbDieses=0;

        Scanner s=new Scanner(System.in);
        System.out.print("Quel est le nom du fichier ? ");
        nomFichier=s.nextLine();

        f=new BufferedReader(new FileReader(nomFichier));
        chaineLue=f.readLine();
        while (chaineLue != null) {
           if (chaineLue.length()>0 && chaineLue.charAt(0)=='#')
                nbDieses++;
           chaineLue=f.readLine();
        }
        f.close();
        System.out.println("Ce fichier contient "+nbDieses+" #.");
   }
}
```

Le schéma de recherche est le suivant (à droite, le schéma générique à connaître par cœur !).

```
BufferedReader f;
String chaineLue;
f=new(BufferedReader(new FileReader("<fichier>"));
chaineLue=fr.readLine();
while ((chaineLue != null) && ... {
    chaineLue=fr.readLine();
}
if (chaineLue!=null) { // élément trouvé
    ...
}
else {    // élément non trouvé
    ...
}
f.close();
```

```
while (!FindeSéquence) && (!P
(EC)){
        Avancer
}
if (! FindeSéquence) {// trouvé
        ...
}
else {    // non trouvé
        ...
}
```

Par exemple, voici un programme qui cherche un mot dans un fichier de mots (un mot par ligne). On pourrait facilement le compliquer en supposant qu'une ligne contient plusieurs mots. On aurait alors deux recherches : (1) chercher une ligne à l'intérieur de laquelle on aurait à (2) chercher un mot.

```
Import java.util.Scanner ;
import java.io.*;
public class MotPresent {
   public static void main(String [] arg) throws IOException {
        Scanner s = new Scanner(System.in);
        BufferedReader f;
        String chaineRecherchee,chaineLue;
        System.out.println("Quel mot rechercher ? ");
        chaineRecherchee=s.nextLine();
        f=new BufferedReader(new FileReader (new File ("datamots.txt")));
        chaineLue=f.readLine();
        while ((chaineLue!= null) && !(chaineLue.equals(chaineRecherchee)))
           chaineLue=f.readLine();
        if (chaineLue==null)
           System.out.println("la chaîne n'a pas été trouvée.");
        else
           System.out.println("la chaîne a été trouvée.");
        }
}
```

Écriture dans un fichier

Écriture dans un fichier

Pour écrire dans un fichier (et le créer s'il n'existait pas), on crée une variable de type BufferedWriter. Pour écrire, on utilise l'action write à laquelle on passe en paramètre une chaîne. Pour ajouter un saut de ligne, on utilise l'action newLine(). Voici le schéma général d'écriture dans un fichier :

```
BufferedWriter f;
String ch;
f=new BufferedWriter(new FileWriter ("<nom fichier>"));
ch=...
f.write(ch);
f.newLine();
...
f.close();
```

Par exemple, pour recopier dans un fichier toutes les lignes qui contiennent un entier positif, on utilise ce programme qui parcourt le premier fichier.

```
// Recopie des entiers positifs d'un fichier dans un autre
// un entier par ligne !
import java.io.*;
public class RecopiePositifs {
  public static void main(String [] arg) throws IOException {
    BufferedWriter fw;
    BufferedReader fr;
    String chaineLue;
    int entierLu;
    fw=new BufferedWriter(new FileWriter ("recopieout.txt"));
    fr=new BufferedReader(new FileReader("recopiein.txt"));
    chaineLue=fr.readLine();
    while (chaineLue != null) {
      entierLu=Integer.parseInt(chaineLue);
      if (entierLu>0) {
        fw.write(chaineLue);
        fw.newLine();
      }
      chaineLue=fr.readLine();
    }
    fr.close();
    fw.close();
  }
}
```

Voici un exemple consistant à recopier les lignes d'un fichier dans un autre, mais sans les espaces. On a donc deux parcours : parcours de lignes et parcours de caractères dans la ligne.

```
import java.io.*;
public class CarParCar {
   public static void main(String [] arg) throws IOException {
        BufferedWriter fw;
        BufferedReader fr;
        String chaineLue,chaineEcrite;
        int i;
      fr=new BufferedReader(new FileReader("fichierin.txt"));
        fw=new BufferedWriter(new FileWriter ("fichierout.txt"));
      chaineLue=fr.readLine();
      while (chaineLue != null) {  // tant qu'on n'est pas à la fin du fichier
```

```
            chaineEcrite="";
            i=0;
            while (i<chaineLue.length()) {
                if (chaineLue.charAt(i)!=' ')
                    chaineEcrite=chaineEcrite+chaineLue.charAt(i);
                i++;
            }
            fw.write(chaineEcrite);
            fw.newLine();
            chaineLue=fr.readLine();  // ligne suivante
        }
        fr.close();
        fw.close();
    }
}
```

Si l'on veut ajouter les éléments *à la fin* d'un fichier existant, et non pas écraser les anciennes valeurs, il faut créer le fichier avec un paramètre supplémentaire égale à true :

```
f=new BufferedWriter(new FileWriter ("<nom fichier>", true));
```