Entrées/Sorties

FIP ING 39

Serge Rosmorduc
serge.rosmorduc@lecnam.net
Conservatoire National des Arts et Métiers

2017-2022

But du cours

Donner les éléments de base pour comprendre les entrées/sorties en java (et dans les autres langages de programmation, tant qu'on y est).

- Cours 1 : lecture/écriture de texte
- Cours 2 : Approfondissement ; flux binaires ; manipulation de fichiers.

Définition

- Entrées (Input) : opération, pour une machine, de récupérer (lire) une donnée depuis un périphérique : une webcam, le réseau, un fichier sur disque...
- Sortie (Output) : opération, pour un ordinateur, d'envoyer (écrire) une donnée vers un périphérique : un écran, une led, un fichier sur disque, le réseau.
- on prend le point de vue de l'ordinateur : c'est la machine qui lit ou écrit ;
- I/O : sigle pour entrées/sorties.

Entrées/sorties texte séquentielles

Aujourd'hui, nous étudierons les entrées/sorties texte séquentielles.

- texte : on va lire et écrire des *caractères* (chars) ;
- séquentielles : on va lire caractère par caractère

Exemple simple de sortie texte

```
public static void main(String[] args) throws IOException {
  char [] texte= {
   'u', 'n', '␣', '2', '\n', 't', 'r', 'o', 'i','s'
  };
  FileWriter w= new FileWriter("demo.txt");
  for (char c: texte) {
   w.write(c);
  }
  w.close();
}
```

```
public static void main(String[] args) throws IOException
```

presque toutes les méthodes d'I/O peuvent lever des exceptions.

```
FileWriter w= new FileWriter("demo.txt");
```

le FileWriter va servir à écrire dans le fichier. Sa création entraîne celle du fichier « demo.txt »

```
w.write(c);
```

écrit le caractère c dans le fichier, puis avance la tête d'écriture.

```
w.close();
```

il est très important de fermer les Writer et les Reader.

Lecture de texte

Plus complexe : quand on écrit, on sait ce qu'on doit écrire. Quand on lit, tout dépend du contenu du fichier !

```
public static void main(String[] args) throws IOException {
  FileReader r= new FileReader("toto.txt");
  int c= r.read();
  while (c != -1) {
   char cc= (char) c;
   System.out.println(cc);
   c= r.read();
  }
  r.close();
}
```

```
FileReader r= new FileReader("toto.txt");
```

Crée et ouvre le Reader, qui lira le contenu du fichier toto.txt.

```
int c= r.read();
```

*On va lire **caractère par caractère** dans la variable c - c'est un int, on en reparle sur la diapo suivante.*

```
while(c != -1)
```

Quand on arrive à la fin du fichier, read() renvoie -1.

```
    char cc= (char) c;
    System.out.println(cc);
```

On affiche le caractère de code c.

```
    c= r.read();
```

on n'oublie pas de lire le caractère suivant !!!

```
    r.close();
```

...ni de fermer.

Pourquoi read() renvoie-t-il un int ?

- Logique de la lecture de fichier en langage C. Au lieu d'avoir une opération de lecture et une opération pour savoir si on est en fin de fichier, on en a une seule : read().
- read() renvoie **le code du caractère lu** (pour java, un caractère et son code, c'est presque pareil). Comme -1 n'est le code d'aucun caractère, cette valeur peut être utilisée sans risque pour indiquer une fin de fichier.
- $\rightarrow$ *valeur sentinelle*.
- Il est **très important que c soit déclarée de type int.**
- ensuite on peut parfois avoir besoin d'utiliser un cast comme nous l'avons fait.
- ... mais pas toujours : on peut parfois travailler avec c int de bout en bout.

Flux

...	b	o	n	j	o	u	r		m	o	n	d	e	\n	...
		⇑													

Définition

Suite de données, finie ou infinie, dotée d'un curseur.

- Deux grands types de flux : en lecture (curseur = tête de lecture) et flux en écriture ;
- Les lectures (resp. écritures) dans le flux se font à la position du curseur.
- L'opération de lecture (resp. d'écriture) avance le curseur ;
- On suppose qu'à l'ouverture du flux, la tête se trouve sur une position d'attente *avant la première cellule*.

Exemple de flux : canal de communication en réseau, fichier sur disque, entrée standard au clavier...

Remarque sur read() et write()

- read() fait deux choses : il avance d'un cran dans le flux et renvoie le code du caractère lu ;
- si on appelle deux fois de suite read(), on aura à priori des résultats différents (un caractère, puis le suivant dans le flux).
- fonctionnement similaire pour write().

Fichier

Le **fichier,** *(file)* c'est un ensemble de données enregistrées sur votre disque dur, doté d'un nom, et localisé par un **chemin** (*path)*.

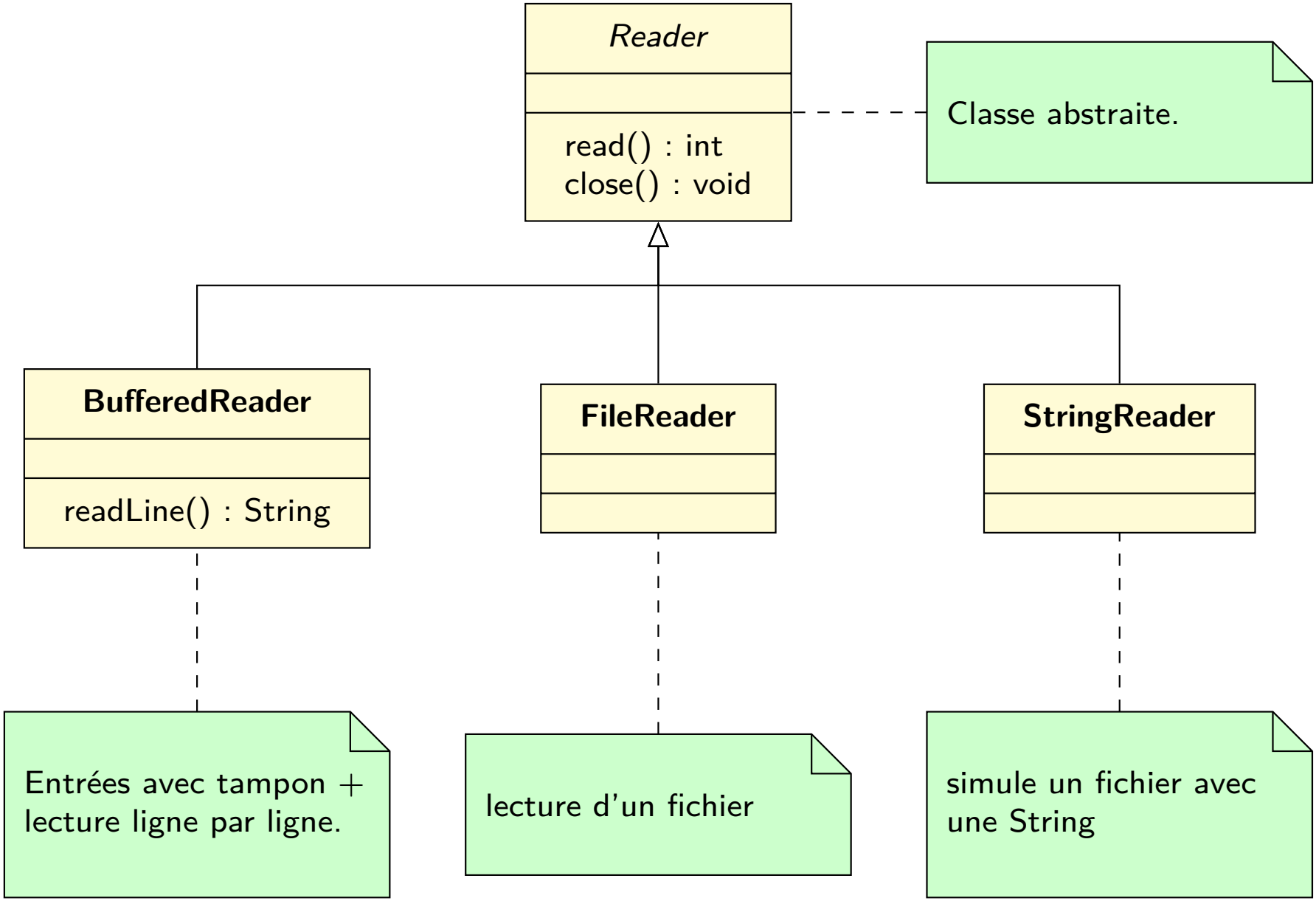
Nom	Date de modification
cs224n.stanford.edu.html	25 janvier 2015 17:25
NAACL2013-Socher-Manning-DeepLearning.pdf	9 juin 2013 14:35
pa4_ner.pdf	21 novembre 2014 22:11
pa4-ner	31 janvier 2015 10:52
pa4-ner.zip	21 novembre 2014 22:11

On considère deux types de fichiers :

- les dossiers, ou répertoire (*folders, directories*) qui contiennent d'autres fichiers ;
- les fichiers « normaux », qui contiennent un flux, texte (par exemple du code java), ou binaire (une image jpeg ou un fichier .class).

On parlera plus des fichiers au prochain cours.

Architecture simplifiée des I/O Textes



Reader

Classe **abstraite**, ancêtre de toutes les classes de flux d'entrée texte.

A priori, toutes les méthodes qui suivent lèvent IOException.

Méthodes

int read()

avance, puis renvoie le code du caractère sous la tête de lecture, ou -1 si on est à la fin du fichier.

void close()

ferme le flux (important !)

FileReader

Reader qui lit dans un fichier.

Constructeurs

FileReader(String fileName)

Ouvre un reader sur un fichier dont on fournit le nom (en fait, le chemin d'accès comme `/home/rosmord/toto.txt`, `toto.txt` ou `C:\Data\toto.txt`)

FileReader(File file)

Ouvre un reader sur le fichier file.

(pas de méthode spécifique à FileReader)

FileReader (2)

Exemple

```
FileReader r= new FileReader("toto.txt");
int c= r.read(); int nb= 0;
while (c!= -1) {
  nb++;
  c= r.read();
}
r.close();
System.out.println("taille "+ nb + " caractères");
```

BufferedReader

Reader avec tampon. Souvent utilisé pour lire ligne par ligne.

BufferedReader

Reader avec tampon. Souvent utilisé pour lire ligne par ligne.

Définition : tampon *(buffer)*

Zone de mémoire dans laquelle la machine va stocker des informations, en provenance ou à destination d'un périphérique.
idée : lire ou écrire les données bloc par bloc et non octet par octet $\rightarrow$ gain de temps.

BufferedReader

Reader avec tampon. Souvent utilisé pour lire ligne par ligne.

Constructeur

BufferedReader(Reader in)

Crée un buffered reader qui lit ses données à partir d'un autre Reader.

Méthodes

String readLine()

renvoie la prochaine ligne dans le flux, ou `null` en fin de fichier.

BufferedReader (2)

Exemple de lecture ligne par ligne.

```
FileReader f0= new FileReader("toto.txt");
BufferedReader r= new BufferedReader(f0);
String s= r.readLine();
while (s != null) {
  System.out.println(s);
  s= r.readLine();
}
r.close();
```

StringReader

Flux lisant dans une chaîne de caractères... pratique pour écrire des tests.

Constructeur

StringReader(String s)

Crée un reader qui lira le contenu de s. Par exemple, si s= "toto", read() renverra successivement « t », « o », « t » et « o ».

Exemple idiot

```
StringReader r= new StringReader("hello");
int c= r.read();
while (c!= -1) {
  System.out.println((char)c);
  c= r.read();
}
r.close();
```

StringReader (2)

Utilisation de StringReader pour un test...

Soit à tester la méthode statique `String Inverseur.inverser(Reader r)` qui renvoie le texte lu à partir du Reader r, mais à l'envers...

```
@Test
public void testerInverse() {
  StringReader entree= new StringReader("hello");
  String attendu= "olleh";
  String resultat= Inverseur.inverser(entree);
  Assert.assertEquals(attendu, resultat);
}
```

Utilisation de la hiérarchie des classes

- Soit la fonction :

```
1 static String lireEnMajuscule(FileReader r) {
2   ... lit r et renvoie le texte lu en majuscules.
3 }
```

Utilisation de la hiérarchie des classes

- Soit la fonction :

```
1 static String lireEnMajuscule(FileReader r) {
2     ... lit r et renvoie le texte lu en majuscules.
3 }
```

- le type choisi pour est-il le bon ?

Utilisation de la hiérarchie des classes

- Soit la fonction :

```
1 static String lireEnMajuscule(FileReader r) {
2     ... lit r et renvoie le texte lu en majuscules.
3 }
```

- le type choisi pour est-il le bon ?
- tel quel, lireEnMajuscule ne peut utiliser que des FileReader !

Utilisation de la hiérarchie des classes

- Soit la fonction :

```
1  static String lireEnMajuscule(FileReader r) {
2     ... lit r et renvoie le texte lu en majuscules.
3  }
```

- le type choisi pour est-il le bon ?
- tel quel, lireEnMajuscule ne peut utiliser que des FileReader !
- en mettant Reader à la place :

```
1  static String lireEnMajuscule(Reader r)
```

Utilisation de la hiérarchie des classes

- Soit la fonction :

```
1  static String lireEnMajuscule(FileReader r) {
2     ... lit r et renvoie le texte lu en majuscules.
3  }
```

- le type choisi pour est-il le bon ?
- tel quel, lireEnMajuscule ne peut utiliser que des FileReader !
- en mettant Reader à la place :

```
1  static String lireEnMajuscule(Reader r)
```

- on pourra passer en argument n'importe quelle sorte de `Reader`, y compris `FileReader`

Utilisation de la hiérarchie des classes

- Soit la fonction :

```
1  static String lireEnMajuscule(FileReader r) {
2     ... lit r et renvoie le texte lu en majuscules.
3  }
```

- le type choisi pour est-il le bon ?
- tel quel, lireEnMajuscule ne peut utiliser que des FileReader !
- en mettant Reader à la place :

```
1  static String lireEnMajuscule(Reader r)
```

- on pourra passer en argument n'importe quelle sorte de `Reader`, y compris `FileReader`
- morale : utiliser le type le plus général possible.

Writer

Flux texte ouvert en écriture. Classe abstraite. *Comme pour les Readers, les méthodes lèvent IOException.*

Méthodes

void write(int c)

écrit le caractère de code c sur le flux. pas besoin de cast.

void write(String s)

écrit la chaîne s sur le flux.

Writer (2)

Méthodes

void flush()

vide (écrit) le contenu du tampon, s'il y en a un.

void close()

Ferme le flux. Important !!!

FileWriter

Flux d'écriture dans un fichier.

Constructeurs

FileWriter(String nomFichier)

Crée un flux texte qui écrit dans le fichier `nomFichier`. Le fichier est créé, et, s'il existe déjà, *vidé.*

FileWriter(File file)

Crée un flux texte qui écrit dans le fichier `file` (voir ci-dessus).

FileWriter (2)

Exemple simple

```
1 FileWriter w= new FileWriter("test.txt");
2 w.write("un texte\nde deux lignes");
3 w.close();
```

StringWriter

Writer qui écrit en mémoire.

- Le texte écrit peut être récupéré comme une chaîne de caractères.
- utile pour les tests.

Constructeur

StringWriter()

crée un StringWriter.

Méthodes

String toString()

permet de récupérer la chaîne écrite dans le StringWriter.

StringWriter

Exemple d'utilisation pour un test.

Soit à tester la méthode statique `MesFichiers.copier(Reader r, Writer w)` qui copie le texte lu par le reader r sur le writer w.

```
@Test
public void testerCopie() throws IOException {
  StringReader r= new StringReader("un texte");
  StringWriter w= new StringWriter();
  MesFichiers.copier(r,w);
  // récupération du texte écrit dans w:
  String ecrit= w.toString();
  Assert.assertEquals("un texte", ecrit);
}
```

Fermeture des flux

Pourquoi fermer les flux ?

- En écriture : si le système utilise un tampon, il est possible que le texte ne soit réellement écrit sur disque qu'après fermeture.
- si on ne ferme pas, la fin du texte peut manquer.
- En lecture : il y a un nombre maximum de flux simultanés ouverts par un programme donné (dépend du système d'exploitation) ;
- si on ne ferme pas les flux, il se peut que l'ouverture de nouveaux flux échoue à partir d'un moment.

Fermeture des flux

Pour être sûr de fermer les flux, même en cas d'exception, on peut utiliser try... finally :

```
FileReader r= null; FileWriter w= null;
try {
  r= new FileReader("a.txt");
  w= new FileWriter("b.txt");
  int c= r.read();
  while (c!= -1) {
    w.write(c);
    c= r.read();
  }
} finally {
  if (w != null) w.close();
  if (r != null) r.close();
}
```

Complexe quand il y a plusieurs flux à fermer (le code ci-dessus est incorrect en cas d'exception lors de `w.close()`).

Fermeture des flux (java 1.7+)

Solution automatisée : le try - with - resources.

```
try (
 FileReader r= new FileReader("a.txt");
 FileWriter w= new FileWriter("b.txt");
) {
  int c= r.read();
  while (c!= -1) {
    w.write(c);
    c= r.read();
  }
}
```

La méthode close() des objets créés dans la parenthèse qui suit le try est automatiquement appelée.

Analyse de problèmes d'entrées/sortie

Conception d'un algorithme de lecture

questions à se poser :

- que veut-on lire ? quelle est la forme de ce qu'on veut lire ?
- comment lire
  - lecture ligne par ligne ?
  - "token" par "token" ?
  - caractère par caractère ?

Exemple : Recherche de lignes contenant un texte donné

On parle de lignes → lecture ligne par ligne probablement indiquée.
Algorithme :

```
pour toute ligne l du fichier
  si l contient le texte
    afficher l
```

```java
public static void main(String[] args) throws IOException {
  String texte= args[0];
  String fichier= args[1];
  BufferedReader reader=
    new BufferedReader(new FileReader(fichier));
  String s= reader.readLine();
  while(s != null) {
   if (s.contains(texte)) {
     System.out.println(s);
   }
   s= reader.readLine();
  }
  reader.close();
}
```

Lecture caractère par caractère

Exemple : calcul de la liste des mots dans un texte.

- utilisation de séparateur ou définition par le contenu ?
  - « un mot est délimité par des espaces » ?
  - ou ... « un mot est une suite de lettres » ?
- la dernière définition semble la plus solide.

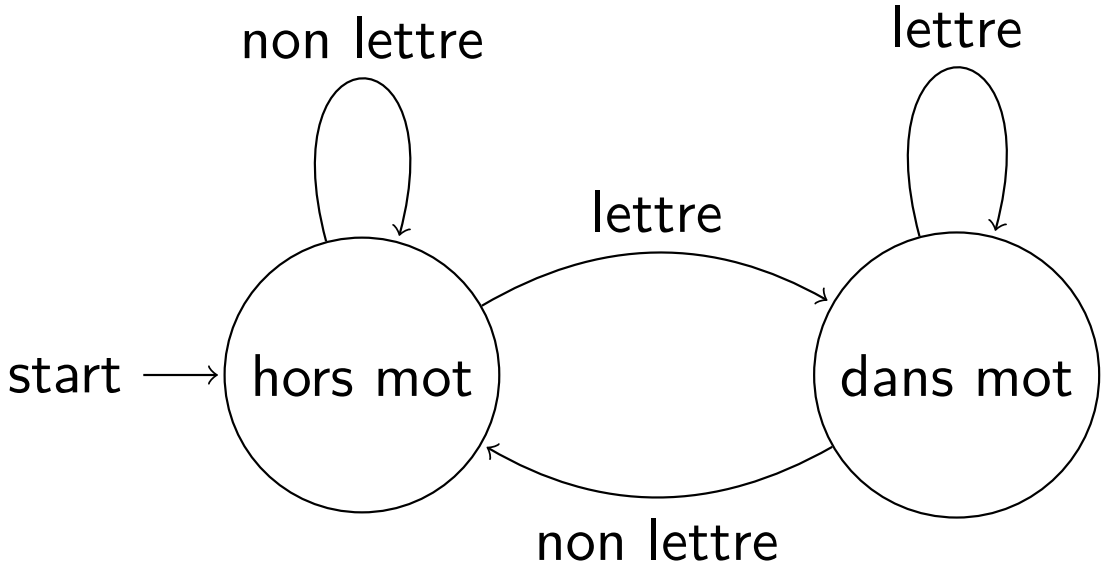
On réfléchit en utilisant notion **d'état.**
Qu'est-ce qu'un état ?

- qu'est-ce que je m'attends à lire ?
- que dois-je faire de ce que je lirai ?

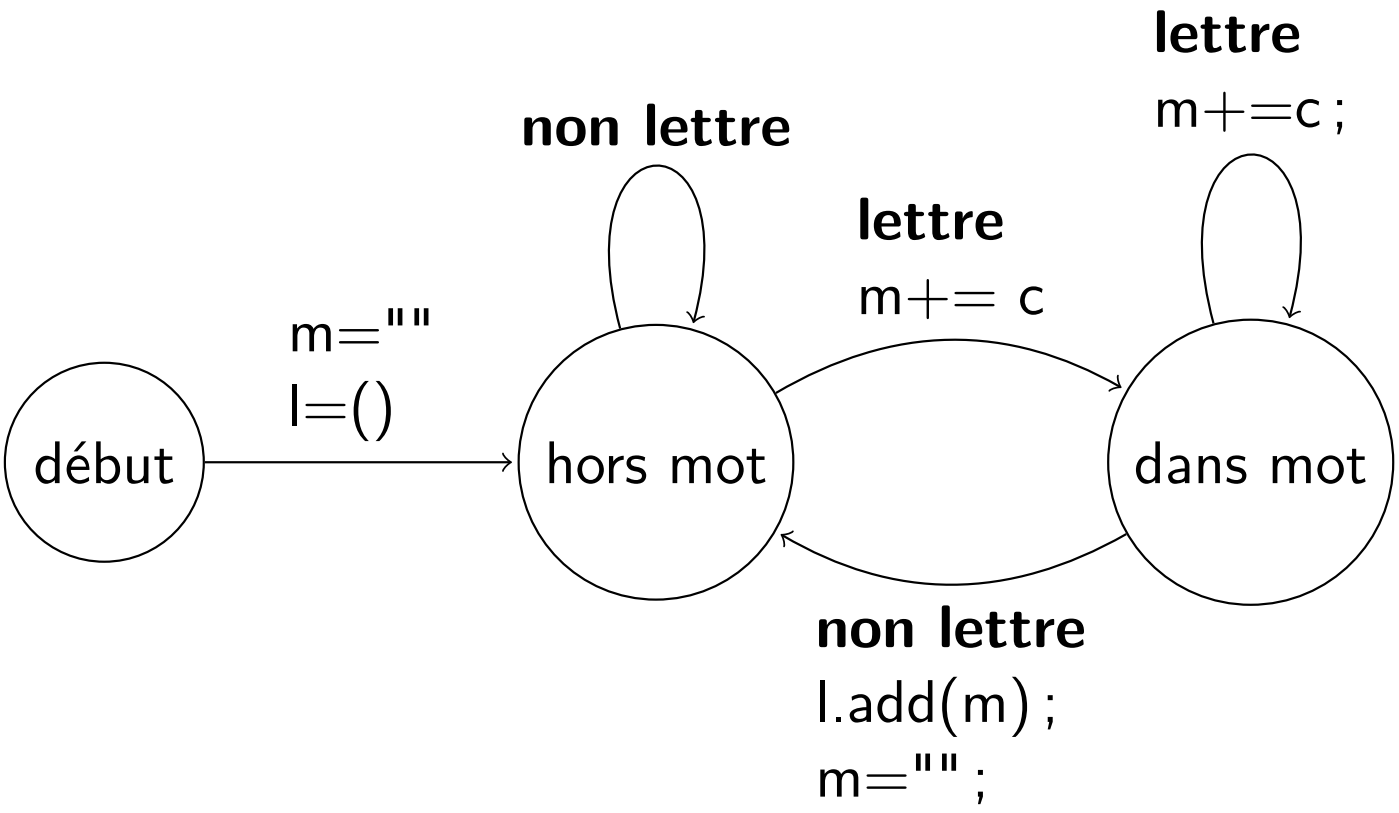
permet une lecture robuste (avec traitement possible des erreurs)

Liste des mots d'un texte

On réfléchit à partir de deux états : DansMot et HorsMot.



On réfléchit à partir de deux états : DansMot et HorsMot.



- on définit un mot comme une suite de lettres...
- quand on lit un caractère :
  - soit c'est une lettre ;
    - ★ si on avait déjà lu une lettre → on reste dans le même mot ;
    - ★ sinon, on commence un nouveau mot.
  - soit c'est autre chose ;
    - ★ si on avait lu une lettre : on termine le mot ;
    - ★ si le précédent caractère n'était pas une lettre, on continue...

```
etat = HorsMot ; mots= liste vide
motCourant= ""; c= lire()
tant que c != -1 faire
  si etat == DansMot
    si c est lettre alors
      motCourant= motCourant + c
    sinon
       ajouter motCourant à mots
       motCoutant= ""
       etat= HorsMot
  sinon
    si c est lettre alors
       motCourant= motCourant + c
       etat= DansMot
    sinon
       (on ne fait rien)
    fin si
  fin si
  c= lire()
fin tant que
// fin du fichier:
si etat== dansMot alors
  ajouter motCourant à mots.
fin si
```

En java

```
boolean dansMot= false;
String courant= "";
List<String> mots= new ArrayList<>();
int c;
do {
  c= reader.read();
  if (dansMot) {
    if (Characters.isLetter(c)) {
      courant+= c;
    } else {
        mots.add(courant);
        courant= "";
        dansMot= false;
    }
  } else {
    if (Characters.isLetter(c)) {
      courant+= c;
      dansMot= true;
    }
  }
} while (c != -1);
// dernier mot correctement traité quand c= -1.
```

Note sur le codage des états

- implicite (servent simplement à mieux réfléchir) - ici, on conserverait par exemple « caracterePrecedent ».
- explicite, codé par des variables (booléens, entiers ou enum) (comme ici) ;
- codés par des objets (un objet état a une méthode qui reçoit une valeur et renvoie un nouvel état).

Lecture d'une heure

Exemple : lecture et interprétation « à la main » d'une heure, entrée sous la forme « 10h14 ».

- On commence par spécifier précisément **la forme du texte** (qu'est ce qui est optionnel, qu'est-ce qui ne l'est pas...)
- ici :
  - exemple lecture d'une heure, de la forme :
  - un ou deux chiffres, zéro ou plusieurs espaces, la lettre « h », zéro ou plusieurs espaces, un ou deux chiffres.
  - expression régulière : `[0-9][0-9]? * h * [0-9][0-9]?`
- on lit un caractère, *puis* on l'interprète → on termine une interprétation en lisant le caractère suivant ;
- On raisonne en terme *d'états*. À moment donné, où en sommes-nous, quels caractères pouvons nous attendre ?
- lié à la notion *d'automates*

Début : lecture d'un ou deux chiffres pour l'heure

```
// On lit le premier caractère avant de commencer
c= lire()
si c n'est pas un chiffre
   erreur()
fin si
h= valeur de c
c= lire() //on avance...
//lecture du chiffre "optionnel"
si c est un chiffre
   h= h * 10 + valeur de c
   c= lire() // on avance
fin si
```

*à la fin, c correspond au caractère qui **suit** le texte reconnu*

- intérêt de « -1 » comme « fin de flux » ;
- on est prêt à lire la suite...

Lecture de zéro ou n espaces,

```
// c a déjà été lu...
tant que c = espace
   c= lire()
fin tant que
```

- on est sûr que c n'est pas un espace après la boucle ;
- on s'arrête forcément : fin du fichier, `lire()` renvoie -1.

Lecture de « h »

Normalement, après les espaces, le caractère qui suit est un « `h` ».

- si ça n'est pas le cas, on a une erreur ;
- si c'est le cas, on « saute » le « h » et on avance au caractère suivant.

```
si c != 'h' alors
  erreur()
fin si
c= lire()
```

On réunit le tout...

```
c= lire()
si c n'est pas un chiffre alors erreur() fin si
h= valeur de c
c= lire()
si c est un chiffre
   h= h * 10 + valeur de c
   c= lire()
fin si
tant que c = espace faire c= lire() fin tant que
si c != 'h' alors erreur() fin si
c= lire()
tant que c = espace faire c= lire() fin tant que
si c n'est pas un chiffre alors erreur() fin si
m= valeur de c
c= lire()
si c est un chiffre
   m= m * 10 + valeur de c
   c= lire()
fin si
```