INF 321 : mémento de la syntaxe de Java

Table des matières

1 La structure générale d'un programme

1.1 Programme sans enregistrements

```
class  NomProgramme
{
    // déclaration des variables globales
    static  type  nomVariable;

    // définition des fonctions secondaires
    static  type  nomFonction( type1 para1, ..., typeN paraN)
    {
       déclarations de variables locales
       instructions
       return  valeur;
    }
    // définition de la fonction principale
    public static void main(String[] args)
    {
       déclarations de variables locales
       instructions
    }
}
```

Si ce programme s'appelle *NomFichier*.java, il se compile au moyen de la commande
javac *NomFichier*.java
qui produit le fichier *NomProgramme*.class.
Le programme est exécuté au moyen de la commande
java *NomProgramme*

1.2 Programme avec enregistrements

Un programme peut inclure la définition d'un nombre quelconque d'enregistrements (accompagnés d'éventuels constructeurs) *avant* le bloc class *NomProgramme*.

```
class  NomEnregistrement
{
    // déclaration des champs
     type  nomChamp;
}
class  NomProgramme
{
    ...
    public static void main(String[] args)
    {
      ...
    }
}
```

2 Les composants élémentaires de Java

2.1 Les identificateurs

Un identificateur est une suite de caractères parmi :

- les lettres (minuscules ou majuscules),
- les chiffres,
- le "blanc souligné" (`_`),
- le dollar (`$`).

Les majuscules et minuscules sont différenciées.

Conventions.

- **noms de variables :** éviter les noms commençant par `_` ou par `$` car ils sont habituellement réservés respectivement aux variables systèmes et aux variables d'un code généré automatiquement ;
- **noms de fonctions :** juxtaposer un verbe en minuscule et un nom débutant par une majuscule (par exemple, `parcourirListe`) ;
- **noms d'enregistrements et noms de programmes :** commencer par une majuscule.

2.2 Les mots-clefs

Un certain nombre de mots, appelés *mots-clefs*, sont réservés pour le langage lui-même et ne peuvent pas être utilisés comme identificateurs. Le Java compte 50 mots clefs, parmi lesquels `const` et `goto` sont réservés mais ne sont pas utilisés.

```
abstract  assert        boolean   break      byte    case       catch     char
class     const         continue  default    do      double     else      enum
extends   final         finally   float      for     if         goto      implements
import    instanceof    int       interface  long    native     new       package
private   protected     public    return     short   static     strictfp  super
switch    synchronized  this      throw      throws  transient  try       void
volatile  while
```

Parmi les mots interdits pour les identificateurs, il faut ajouter les booléens `false` et `true` ainsi que `null`.

2.3 Les commentaires

Commentaire de fin de ligne :

```
// Ceci est un commentaire
```

Commentaire d'un bloc de code :

```
/*  Ceci est
    un commentaire */
```

Commentaire d'un bloc de code pour la documentation `Javadoc` :

```
/**  Ceci est
     un commentaire */
```

3 Les types primitifs

type	taille (en bits)	intervalle des valeurs	valeur par défaut
`byte`	8	$[-2^7; 2^7[$	0
`short`	16	$[-2^{15}; 2^{15}[$	0
`int`	32	$[-2^{31}; 2^{31}[$	0
`long`	64	$[-2^{63}; 2^{63}[$	0
`char`	16	[`\u0000`; `\uffff`]	`\u0000`
`boolean`	1	{`false, true`}	`false`
`float`	32	[`1.4E − 45`; `3.4028235E38`]	0
`double`	64	[`4.9E − 324`; `1.7917931348623157E308`]	0

3.1 Les constantes entières

Les constantes entières peuvent être représentées dans trois bases :

- **décimale** (par défaut) ;
- **octale :** les constantes octales commencent par un zéro (ex : `0377`) ;
- **hexadécimale :** les constantes hexadécimales commencent par `0x` ou `0X` (ex : `0xff`).

On peut spécifier explicitement qu'une constante entière est de type `long` en la suffixant par `l` ou `L` (ex : `256L`).

3.2 Les constantes flottantes

Par défaut, une constante flottante est de type `double` sauf si elle est suffixée par `f` ou `F`. Par ailleurs, une constante entière suffixée par `d` est de type `double`. Ainsi, `10d` ou `10.` désignent la même constante de type `double`.

3.3 Les constantes caractères

Les caractères Java sont les caractères Unicode. Pour désigner un caractère imprimable, il suffit de le mettre entre apostrophes (par ex. `'A'` ou `'$'`), sauf pour l'antislash et l'apostrophe, qui sont respectivement désignés par `'\\'` et `'\''`. Les caractères peuvent plus généralement être désignés par `'\u`*`CodeHexa`*`'` où *`CodeHexa`* est le code Unicode en hexadécimal (4 symboles) du caractère. Les caractères 8 bits peuvent également s'écrire `'\`*`CodeOctal`*`'` où *`CodeOctal`* est le code octal du caractère, ou `'\x`*`CodeHexa`*`'` où *`CodeHexa`* est le code hexadécimal (2 symboles). Les caractères non-imprimables les plus fréquents disposent aussi d'une notation plus simple :

`\n`	nouvelle ligne	`\r`	retour chariot
`\t`	tabulation horizontale	`\f`	saut de page
`\b`	retour arrière		

3.4 Les constantes chaînes de caractères

Une chaîne de caractères est une suite de caractères entourés par des guillemets. A l'intérieur d'une chaîne de caractères, le caractère `"` doit être désigné par `\"`.

3.5 Les constantes symboliques

Il est souvent souhaitable de donner un nom à une constante plutôt que d'utiliser sa valeur littérale. Pour cela, on la déclare comme suit et on note généralement son identificateur en majuscules, par exemple

```
final int REPONSE = 42;
```

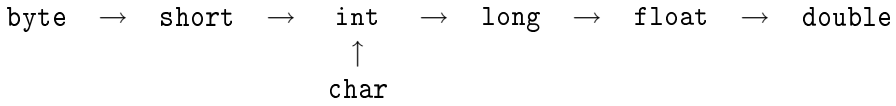
4 Les opérateurs

4.1 L'affectation

variable `=` *expression*

Cette expression a pour effet d'évaluer *expression* et d'affecter la valeur obtenue à *variable*. De plus, cette expression possède une valeur, qui est celle d'*expression*. Ainsi, l'expression `i = 5` vaut 5.

L'affectation effectue une *conversion de type implicite* : la valeur de l'expression (terme de droite) est convertie dans le type du terme de gauche. Cette conversion est possible uniquement si elle s'effectue vers un type de taille supérieure à celle du type de départ :



Ces conversions sont sans perte de précision, sauf celles de `int` et `long` en `float` et `double` qui peuvent parfois entraîner des arrondis.

4.2 Les opérateurs arithmétiques

Les opérateurs arithmétiques classiques sont l'opérateur unaire `-` (opposé) ainsi que les opérateurs binaires

- `+` addition
- `-` soustraction
- `*` multiplication
- `/` division
- `%` reste de la division (modulo)

Ces opérateurs agissent de la façon attendue sur les entiers comme sur les flottants. Leurs seules spécificités sont les suivantes :

- La notation `/` désigne à la fois la division entière et la division entre flottants. Si les deux opérandes sont de type entier, l'opérateur `/` produira une division entière (quotient de la division). Par contre, il délivrera une valeur flottante dès que l'un des opérandes est un flottant. Par exemple,

  ```
  double x;
  x = 3 / 2;
  ```

  affecte à `x` la valeur 1. Par contre

  ```
  x = 3 / 2.;
  ```

  affecte à `x` la valeur 1.5.

– L'opérateur `%` est le reste de la division euclidienne. Si l'un des deux opérandes est négatif, le signe du reste est celui du dividende. Cet opérateur s'applique aussi à des flottants. Dans ce cas, la valeur `r` de `a % b` est donnée par `r = a - bq` où `q` $= \lfloor \frac{a}{b} \rfloor$.

Notons enfin qu'il n'y a pas d'opérateur effectuant l'élévation à la puissance. De façon générale, il faut utiliser la fonction `pow(x,y)` de la classe `java.lang.Math` pour calculer x^y.

4.3 Les opérateurs relationnels

`>`	strictement supérieur
`>=`	supérieur ou égal
`<`	strictement inférieur
`<=`	inférieur ou égal
`==`	égal
`!=`	différent

Leur syntaxe est

expression-1 op expression-2

Les deux expressions sont évaluées puis comparées. La valeur retournée est de type `boolean`.

4.4 Les opérateurs logiques booléens

`&&`	et logique
`||`	ou logique
`!`	négation logique

Comme pour les opérateurs de comparaison, la valeur retournée par ces opérateurs est un `boolean`. Dans une expression de type

expression-1 op-1 expression-2 op-2 ...expression-n

l'évaluation se fait de gauche à droite et s'arrête dès que le résultat final est déterminé.
Par exemple,

```
int x = 5, y = 2, z = 3;
boolean r;
r = (x >= 0) && (y != 2) || !(z > 10);
```

`r` aura comme valeur `true`.

4.5 Les opérateurs logiques bit à bit

Les sept opérateurs suivants permettent de manipuler des entiers au niveau du bit. Ils s'appliquent à tous les types entiers.

`&`	et	`|`	ou inclusif
`^`	ou exclusif	`~`	complément à 1
`<<`	décalage à gauche	`>>`	décalage à droite
`>>>`	décalage à droite sans propagation du bit de signe		

En pratique, les opérateurs `&`, `|` et `^` consistent à appliquer bit à bit les opérations suivantes

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

L'opérateur unaire `~` change la valeur de chaque bit d'un entier. Le décalage à droite `a >> n` et à gauche `a << n` effectuent respectivement une division et une multiplication par 2^n quand ils opèrent sur un entier `a` positif, où `n` est pris modulo 32 pour les `int` et modulo 64 pour les `long`. Quand `a` est un entier négatif, le décalage à droite `a >> n` propage le bit de signe (on remplit les `n` positions de poids fort avec des `1`). Par contre, le décalage à droite sans propagation du signe, `a >>> n`, propage des zéros.

	représentation décimale	représentation binaire
`a`	77	00000000000000000000000001001101
`b`	23	00000000000000000000000000010111
`a & b`	5	00000000000000000000000000000101
`a | b`	95	00000000000000000000000001011111
`a ^ b`	90	00000000000000000000000001011010
`~a`	-78	11111111111111111111111110110010
`b << 5`	736	00000000000000000000001011100000
`b >> 2`	5	00000000000000000000000000000101
`b >> 3`	2	00000000000000000000000000000010
`b >> 32`	23	00000000000000000000000000010111
`c`	-33	11111111111111111111111111011111
`c >> 2`	-9	11111111111111111111111111110111
`c >>> 2`	1073741815	00111111111111111111111111110111

4.6 L'opérateur conditionnel ternaire

L'opérateur conditionnel `?` est un opérateur ternaire. Sa syntaxe est la suivante :

condition `?` *expression-1* `:` *expression-2*

Cette expression est égale à *expression-1* si *condition* est satisfaite, et à *expression-2* sinon. Par exemple, l'expression

```
x >= 0 ? x : -x
```

correspond à la valeur absolue d'un nombre.

4.7 L'opérateur de conversion de type

L'opérateur de conversion de type, appelé *cast*, permet de modifier explicitement le type d'une expression. On écrit

`(`*type*`)` *expression*

De tous les opérateurs, la conversion de type est celui qui a la priorité la plus élevée.

Par exemple, dans

```
int i = 3, j = 2;
double r = (double)i/j;
```

`r` vaut 1.5.

4.8 La concaténation de chaînes de caractères

L'opérateur `+` appliqué à deux objets de type `String` désigne la concaténation de chaînes de caractères. Dès que l'un des deux opérandes est de type `String`, l'autre est converti en `String`.

Par exemple, dans

```
int i = 3;
System.out.println("i = " + i);
```

la chaîne de caractères constante `"i = "` est concaténée avec la valeur de la variable `i` convertie en chaîne de caractères.

4.9 Règles de priorité des opérateurs

Le tableau suivant classe les opérateurs par ordres de priorité décroissants. Les opérateurs placés sur une même ligne ont même priorité. La flèche de la seconde colonne du tableau donne l'ordre d'associativité de ces opérateurs. Par exemple `a || b || c` correspond à `(a || b) || c` alors que `a = b = c` correspond à `a = (b = c)`.

On préférera toutefois mettre des parenthèses en cas de doute...

opérateurs	
`++` `--` -(unaire) `~` `!` `(`*type*`)`	←
`*` `/` `%`	→
`+` -(arithmétiques) +(String)	→
`<<` `>>` `>>>`	→
`<` `<=` `>` `>=` `instanceof`	→
`==` `!=`	→
`&`(et bit-à-bit)	→
`^`	→
`|`	→
`&&`	→
`||`	→
`?:`	
`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `>>>=` `&=` `^=` `|=`	←

TAB. 1 – *Règles de priorité des opérateurs*

Notons que les opérateurs logiques bit-à-bit sont moins prioritaires que les opérateurs relationnels. Cela implique que dans des tests sur les bits, il faut parenthéser les expressions.

5 Le branchement conditionnel

La forme la plus générale est :

```
if ( expression )
  {
    instruction-1
  }
else
  {
    instruction-2
  }
```

expression est évaluée. Si elle vaut `true`, *instruction-1* est exécutée, sinon *instruction-2* est exécutée.

Le bloc

```
else
  {
    instruction-2
  }
```

est facultatif. Chaque *instruction* peut ici être un bloc d'instructions.
Par ailleurs, on peut imbriquer un nombre quelconque de tests, ce qui conduit à :

```
if ( expression-1 )
  {
    instruction-1
  }
else if ( expression-2 )
  {
    instruction-2
         ⋮
  }
else if ( expression-n )
  {
    instruction-n
  }
else
  {
    instruction-∞
  }
```

6 Les boucles

6.1 Boucle `while`

La syntaxe de `while` est la suivante :

```
while ( expression )
  {
   instruction
  }
```

Tant que *expression* est vérifiée (*i.e.*, vaut `true`), *instruction* est exécutée.

6.2 Boucle `do--while`

Il peut arriver que l'on ne veuille effectuer le test de continuation qu'après avoir exécuté l'instruction. Dans ce cas, on utilise la boucle `do--while`. Sa syntaxe est

```
do
  {
    instruction
  }
while ( expression );
```

Ici, *instruction* sera exécutée tant que *expression* vaut `true`. Cela signifie donc que *instruction* est toujours exécutée au moins une fois.

6.3 Boucle `for`

La syntaxe de `for` est :

```
for ( expr-1 ; expr-2 ; expr-3)
  {
    instruction
  }
```

On doit comprendre cette boucle ainsi : *expr-1* est la condition initiale de la boucle, *expr-2* est la condition de poursuite de la boucle et *expr-3* est une expression exécutée après chaque tour de boucle. L'idée est que *expr-1* représente l'état initial d'un compteur dont la valeur sera modifiée après chaque tour par *expr-3*, et la boucle continue tant que la condition *expr-2* est vérifiée.

Par exemple,

```
for (int i=0; i<10; i=i+1)
  {
   System.out.println("Bonjour");
  }
```

affichera à l'écran 10 fois le mot Bonjour suivi d'un retour à la ligne.

Une version équivalente avec la boucle `while` serait :

```
int i = 0;
while (i < 10)
   {
    System.out.println("Bonjour");
    i = i+1;
   }
```

7 Les instructions d'entrée et de sortie

7.1 Affichage

On utilise pour l'affichage les instructions suivantes :

- `System.out.print(`*`expression`*`);` affiche la valeur textuelle d'une *expression* (opère sur les types primitifs, `String`, `Object`).
- `System.out.println(`*`expression`*`);` affichage avec passage à la ligne.

7.2 Lecture

On utilise les fonctions de la classe `Ppl`. Pour lire un entier entré au clavier et stocker sa valeur dans la variable `x`, la syntaxe est

```
x = Ppl.readInt();
```

La classe `Ppl` contient également les méthodes `readChar`, `readDouble` et `readWord`.

La fonction `endOfInput` retourne un booléen qui vaut `true` dès que le caractère de fin de fichier est rencontré. Pour lire un ensemble d'entiers sur l'entrée standard, on utilisera par exemple :

```
do
   {
      x = Ppl.readInt();
      // instructions
   }
while (!Ppl.endOfInput());
```

8 Fonctions

8.1 Définition

```
static  type  nomFonction( type1 para1, ..., typeN paraN)
{
   déclarations de variables locales
   instructions
   return  valeur
```

– Les méthodes dynamiques, elle, ne sont pas précédées du mot-clef `static`.

Par exemple,

```
static long facto(int n)
{
    long resultat = 1;
    for (int i=2; i<=n; i=i+1)
        {
            resultat = resultat * i;
        }
    return resultat;
}
```

n'est rien d'autre que la fonction factorielle.

Il est possible de définir des fonctions ayant le même nom quand leur nombre d'arguments ou le type de leurs arguments diffèrent.

8.2 La fonction `main`

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

Les arguments de `main` sont des chaînes de caractères, rangées dans le tableau `args`. Il s'agit des différents mots passés en arguments de la commande `java NomClass`. L'élément `args[0]` correspond au premier argument, et non au nom de la classe. Pour convertir les arguments de `main` de type `String` en l'un des types primitifs, par exemple `int`, on utilise la fonction

```
int Integer.parseInt(String str)
```

ou, de manière équivalente `Boolean.parseBoolean`, `Byte.parseByte`, `Short.parseByte`, `Long.parseLong`, `Float.parseFloat`, `Double.parseDouble`.

Par exemple, si l'on veut exécuter un programme avec comme paramètres une chaînes de caractères, un nombre entier puis à nouveau une chaîne de caractères, nous aurons la fonction `main` suivante :

```
public static void main(String[] args)
    {
        String pays = args[0];
        int cp = Integer.parseInt(args[1]);
        String ville = args[2];
              ⋮
    }
```

Ainsi, si notre programme se nomme `Prog`, on pourra l'appeler par la ligne de commande

```
java Prog France 91128 Palaiseau
```

9 Les enregistrements

Définition

```
class  NomEnregistrement
{
    // déclaration des champs
    [final]  type  nomChamp;
}
```

Comme pour les variables, il faut qualifier de `final` un champ pour qu'il soit constant.

Déclaration d'un enregistrement

La déclaration d'un enregistrement se fait de la même manière que pour toute variable. Ainsi, la syntaxe est :

NomEnregistrement nomVariable;

où *NomEnregistrement* est le type, et *nomVariable* le nom de la variable.

Allocation d'un enregistrement

Une fois une variable de type enregistrement déclarée, son allocation se fait par appel au constructeur. Par défaut, la syntaxe est :

nomVariable = new *NomEnregistrement*();

Accès à un champ de la variable enregistrement

nomVariable.nomChamp

Constructeurs

On peut définir d'autres constructeurs :

- Le constructeur est une fonction qui porte le même nom que l'enregistrement ;
- Le type de l'objet retourné n'est pas mentionné, et il n'y a pas d'instruction `return;`
- Il permet par exemple d'affecter les champs de l'objet construit, désigné par `this`.

Par exemple,

```
class Point
{
    double abscisse;
    double ordonnee;

    Point(double x, double y)
    {
        abscisse = x;
        ordonnee = y;
    }
}
```

On appelle alors ce constructeur par `Point p = new Point(42, 55.5);`

Remarquons que l'on peut écrire le constructeur ainsi :

```
Point(double abscisse, double ordonnee)
{
  this.abscisse = abscisse;
  this.ordonnee = ordonnee;
}
```

Dans ce cas, l'utilisation du mot-clé `this` est requise pour distinguer les arguments des champs de l'enregistrement. Ainsi, `this.abscisse` correspond au champ `abscisse` d'un enregistrement de type `Point` alloué par le constructeur. La variable `abscisse` correspond, elle, à l'argument `abscisse` du constructeur.

10 Les tableaux

Déclaration

type`[]` *nomTableau*`;`

On peut également initialiser un tableau à sa déclaration :

type`[]` *nomTableau* `= {`*liste des éléments*`};`

où les éléments de la liste sont séparés par des virgules.

Allocation

nomTableau `= new` *type*`[`*expr*`];`

où *expr* est une expression dont la valeur est un entier définissant la longueur du tableau, *i.e.* son nombre d'éléments.

Accès aux éléments

Les éléments d'un tableau de taille N sont numérotés de 0 à $N-1$. On accède à l'élément d'indice `i` par *nomTableau*`[i]`.

Taille d'un tableau

nomTableau`.length`

fournit la longueur du tableau *nomTableau*.

Tableaux multi-dimensionnels

La déclaration d'un tableau à deux dimensions est de la forme

type`[][]` *nomTableau*`;`

et l'accès à un élément se fait par *nomTableau*`[i][j]`.

nomTableau`[i]` désigne la ligne d'indice `i` du tableau. On peut allouer toutes les lignes simultanément si elles ont la même taille :

```
tab = new int[nbLignes][nbCol];
```

ou allouer les lignes séparément, par exemple pour un tableau triangulaire :

```
tab = new int[nbLignes][];
for (int i=0; i< tab.length; i=i+1)
    tab[i] = new int[i+1];
```

11 Les fonctions mathématiques

La classe `java.lang.Math` contient les fonctions mathématiques usuelles. Elle définit les constantes mathématiques e et π, de type `double` : `E` et `PI`.

Les fonctions associées sont :

`int abs (int a)`
`long abs (long a)`
`float abs (float a)`
`double abs (double a)`
	valeur absolue

`double sin (double a)`	sinus
`double cos (double a)`	cosinus
`double tan (double a)`	tangente
`double asin (double a)`	arc sinus
`double acos (double a)`	arc cosinus
`double atan (double a)`	arc tangente
`double atan2 (double y, double x)`	convertit (x,y) en polaire (r,θ) et renvoie θ
`double toDegrees(double angrad)`	convertit un angle en radians en degrés
`double toRadians(double angdeg)`	convertit un angle en degrés en radians
`double cosh(double x)`	cosinus hyperbolique
`double sinh(double x)`	sinus hyperbolique
`double tanh(double x)`	tangente hyperbolique
`double exp (double a)`	exp
`double log (double a)`	log népérien
`double log10 (double a)`	log en base 10
`double sqrt (double a)`	racine carrée
`double cbrt(double a)`	racine cubique
`double pow (double a, double b)`	puissance
`double floor (double a)`	partie entière (inférieure) de a
`double ceil (double a)`	partie entière (supérieure) de a
`double rint (double a)`	entier le plus proche de `a`
`int round (float a)`	entier le plus proche de `a` (stocké dans un `int`)
`long round (double a)`	entier long le plus proche de `a`

`int max (int a, int b)`
`long max (long a, long b)`
`float max (float a, float b)`
`double max (double a, double b)`
	maximum

`int min (int a, int b)`	minimum

`long min (long a, long b)`
`float min (float a, float b)`
`double min (double a, double b)`	

`double random ()`	retourne un nombre aléatoire de $[0; 1[$

12 Le système de fichiers d'Unix/Linux

Les fichiers d'une machine Unix/Linux sont organisés en un arbre dont les noeuds internes sont les répertoires. A chaque utilisateur (*login*) correspond un répertoire particulier, c'est son *home directory* dans lequel il peut construire son arborescence personnelle. Ce répertoire s'écrit `~`*`login`*. Pour un utilisateur `~` (sans le *`login`*) désigne son propre *home directory*.

Un chemin dans l'arborescence s'écrit en énumérant les noeuds rencontrés avec des / comme séparateurs, par exemple `~/INF_321/TD1/` A partir de tout répertoire, `..` désigne le répertoire père et `.` le répertoire lui-même.

12.1 Les principales commandes

`cd` *`nom_de_repertoire`*
: *Change Directory*
permet de se placer dans le répertoire *`nom_de_repertoire`*. Utilisée sans argument, la commande `cd` replace dans le *home directory*.

`ls` *`nom_de_repertoire`*
: liste le contenu du répertoire *`nom_de_repertoire`*. Utilisée sans argument, elle liste le contenu du répertoire courant. L'option `-a` permet de lister également les fichiers dont le nom commence par `.`, l'option `-l` affiche les propriétés des fichiers (droits d'accès, propriétaire, taille, date de modification), l'option `-t` liste les fichiers en les triant suivant la date de la dernière modification.

`mkdir` *`nom_de_repertoire`*
: *MaKe DIRectory*
crée le répertoire *`nom_de_repertoire`*.

`rmdir` *`nom_de_repertoire`*
: *ReMove DIRectory*
supprime le répertoire *`nom_de_repertoire`*, à condition qu'il soit vide.

`cp` *`fichier_depart fichier_arrivee`*
`cp` *`fichier1 [fichier2 fichier3...]repertoire_arrivee`*
: *CoPy*
Si les deux arguments sont des noms de fichier, la commande copie le fichier désigné par le premier argument dans le fichier désigné par le second. Si le dernier argument est un répertoire existant, tous les fichiers désignés par les arguments précédents sont copiés dans ce répertoire.

`mv` *`fichier_depart fichier_arrivee`*
`mv` *`fichier1 [fichier2 fichier3...]repertoire_arrivee`*
: *MoVe*
déplace des fichiers. La syntaxe est identique à celle de `cp`.

`rm` *`fichier1 [fichier2 fichier3...]`*
: *ReMove*

supprime le ou les fichiers donnés en arguments.

`pwd`

Print Working Directory
affiche le répertoire courant.

12.2 Les expansions d'arguments

Certains caractères spéciaux sont interprétés dans les arguments. Les deux les plus utilisés sont :

`?` qui peut être remplacé par n'importe quel caractère ;

`*` qui peut être remplacé par une chaîne quelconque de caractères (incluant la chaîne vide).

Par exemple, la commande

```
rm TD?/*.class
```

supprime tous les fichiers dont le nom est suffixé par `.class` dans tous les répertoires dont le nom est formé de la chaîne de caractères `TD` suivie d'un caractère (par exemple, `TD1`, `TD2`, mais pas `TD10`).

12.3 Redirection des entrées-sorties

Les commandes et programmes lisent et affichent leurs données respectivement dans l'entrée standard (le clavier) et sur la sortie standard (l'écran). Il est possible de modifier ce comportement par défaut en *redirigeant* l'entrée standard ou la sortie standard vers un fichier.

L'opérateur `>` redirige la sortie standard vers un fichier. Par exemple, avec

```
java Prog > fichier_sortie
```

tout ce qui est affiché à l'exécution de `Prog` est écrit dans le fichier `fichier_sortie`. De la même manière, `ls > fichier`, écrit le résultat de la commande `ls` dans le fichier `fichier`.

L'opérateur `<` redirige l'entrée standard vers un fichier. Par exemple, avec

```
java Prog < fichier_entree
```

les données lues au cours de l'exécution de `Prog` ne sont plus celles entrées au clavier mais celles qui se trouvent dans le fichier `fichier_entree`.