Initiation à la programmation avec le shell Bash

Cours n°2

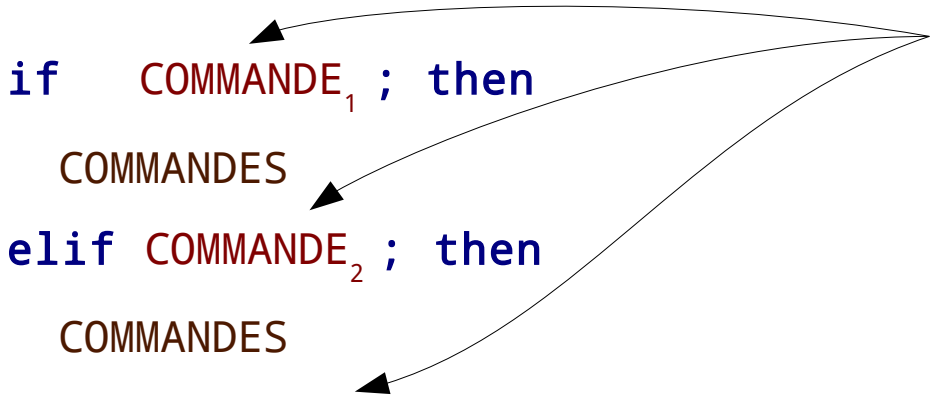
Jean-Vincent Loddo

Sommaire du cours n°1

- Retour sur la conditionnelle (**elif**, **case**)
- Notion n°5 : itérations ou boucles sans condition (**for**)
- Notion n°6 : boucles avec condition (**while**)
- Notion n°7 : Paramétrisation
- Notion n°8 : sous-programmes (**fonctions**)

Retour sur la conditionnelle

- Syntaxe générale (voir **`help if`**) :



```
if   COMMANDE1 ; then
  COMMANDES
elif COMMANDE2 ; then
  COMMANDES
elif COMMANDE3 ; then
  COMMANDES
...
else
  COMMANDES
fi
```

Commandes **d'aiguillage** :

succès => aller à la partie « then » suivante

échec => aller à la partie « elif » ou « else » suivante

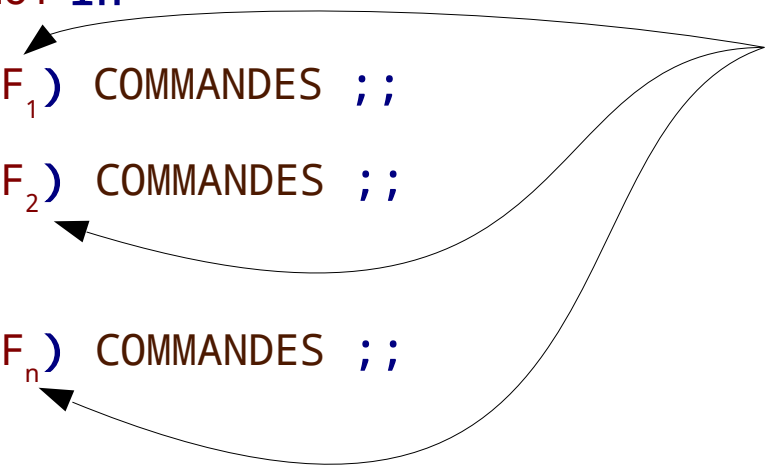
Remarque n.1 : les parties « elif » et la partie « else » **sont optionnelles**

Remarque n.2 : au plus une des COMMANDES après les parties « then » ou « else » **sera exécutée**

Retour sur la conditionnelle : le **case** (switch)

- Syntaxe (voir **`help case`**) :

```
case MOT in
   MOTIF1) COMMANDES ;;
   MOTIF2) COMMANDES ;;
   ...
   MOTIFn) COMMANDES ;;
esac
```



Motifs **d'aiguillage** :

succès => exécuter les commandes correspondantes

échec => passer au motif suivant

- Les motifs peuvent être construits avec les méta-caractères **`* ? [...]`**

Remarque : rien de vraiment nouveau, ce n'est pas fondamental !
Il s'agit d'un **if-then-elif-else** bien déguisé,
c'est-à-dire qu'on pourrait écrire un **if-then-elif-else** équivalent,
en exploitant le programme **grep** pour tester la correspondance des motifs

Retour sur la conditionnelle : le **case** (switch)

- Exemple :

Valeur analysée

```
X="16/03/2005"
case ${X} in
  ??/??/????) echo "J'ai deviné, c'est une date" ;;
        [A-Z]) echo "J'ai deviné, c'est une majuscule" ;;
        [a-z]) echo "J'ai deviné, c'est une minuscule" ;;
         [01]) echo "J'ai deviné, c'est un chiffre binaire" ;;
        [0-9]) echo "J'ai deviné, c'est un chiffre décimal" ;;
esac
```

Motifs **d'aiguillage**

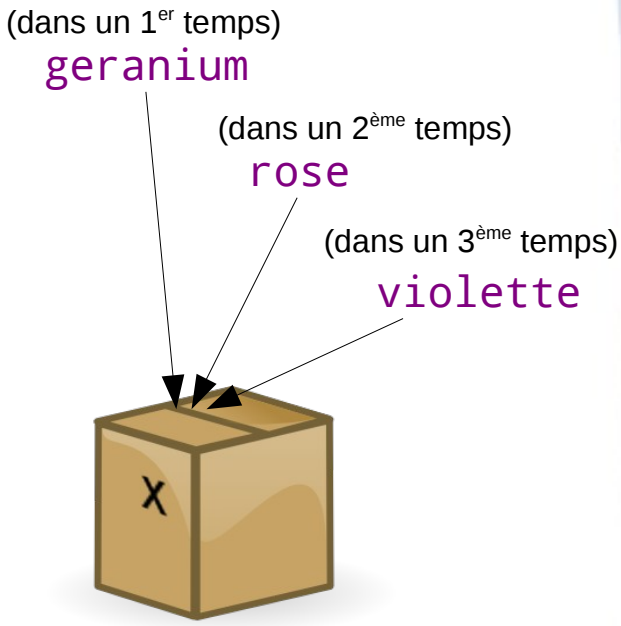
Notion n°5 : **itérations** ou boucles sans condition (for) (1)

- On doit répéter la **même action** plusieurs fois
  - Pas tout à fait la même action, mais à quelque chose près…
  - D'autres langages appellent cette construction plutôt « **foreach** »
  - Exemple : vous avez trois fichiers JPEG
    `geranium.jpg  rose.jpg  violette.jpg`
    que vous voulez convertir au format PNG
  - Vous pouvez faire (méthode bovine) :

```
convert  geranium.jpg  geranium.png
convert  rose.jpg  rose.png
convert  violette.jpg  violette.png
```

  - Vous pouvez faire (méthode rusée) :

```
for X in geranium rose violette; do
  convert  ${X}.jpg  ${X}.png
done
```



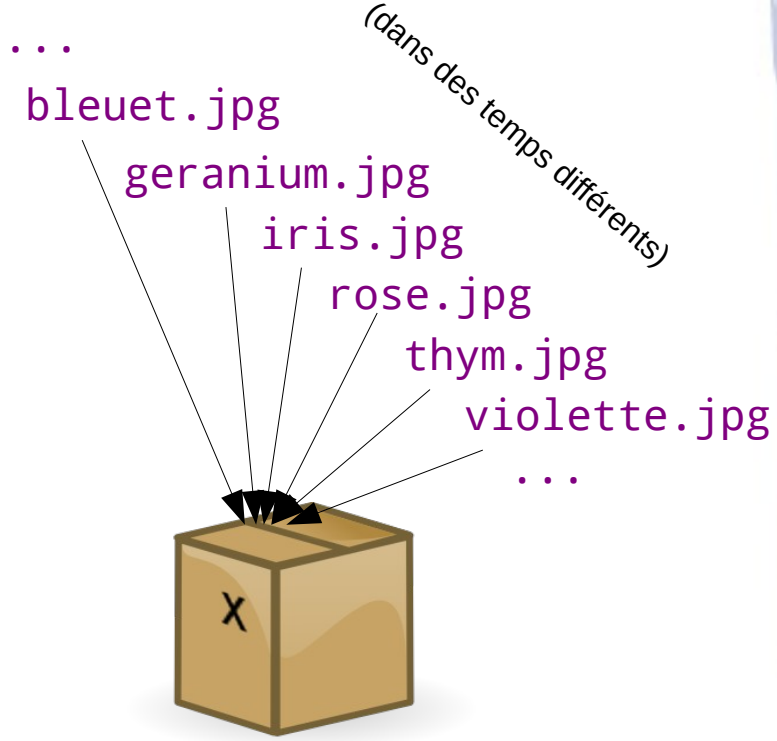
Action répétée 3 fois (pour chacune des valeurs de X)

Notion n°5 : **itérations** ou boucles sans condition (for) (2)

- Les boucles « for » sont d'autant plus intéressantes en Bash qu'on peut utiliser les **métacaractères d'expansion** des noms de fichiers ou répertoires **`* ? [...]`**

```
for X in *.jpg; do
  convert  ${X}  ${X%.jpg}.png
done
```

Action répétée pour tous les fichiers `.jpg` du répertoire courant en ordre alphabétique



- Autres exemples :

```
for X in photo_?.jpg; do
  convert  ${X}  ${X%.jpg}.png
done
```

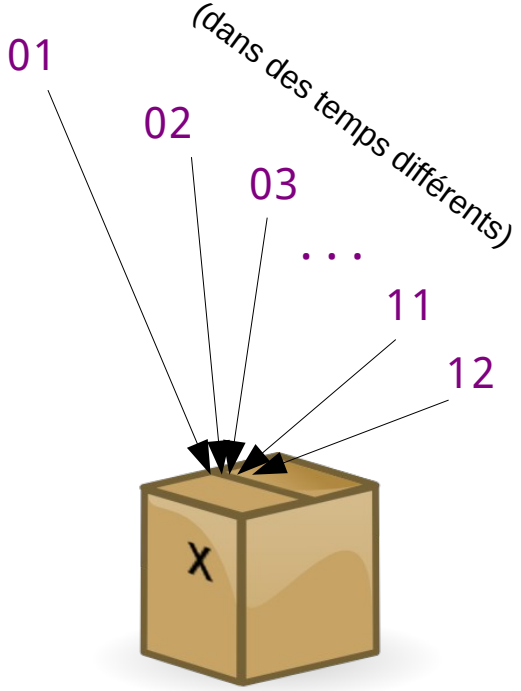
```
for X in *.[jJ][pP][gG]; do
  convert  ${X}  ${X%.jpg}.png
done
```

Notion n°5 : **itérations** ou boucles sans condition (for) (3)

- Les boucles « for » sont d'autant plus intéressantes en Bash qu'on peut aussi utiliser l'**expansion des accolades** (brace expansion)
  - Syntaxe « foreach » : `for NOM in MOTS; do COMMANDES; done`

```
for X in {01..12}; do
  mkdir 2015/${X}
done
```

Action répétée pour tous les mots 01 02 03 ... 12



  - Autre exemple :

```
for X in {2014,2015,2016}/{01..12}; do
  mkdir ${X}
  touch ${X}/README
done
```

Actions répétées pour tous les mots 2014/01 .. 2014/12 .. 2015/01 .. 2015/12 .. 2016/01 .. 2016/12 (donc 36 fois !)

Notion n°5 : **itérations** ou boucles sans condition (for) (4)

- Il existe aussi la boucle « **for arithmétique** », similaire à celle de la plupart des langages de programmation (par exemple C)
  - Syntaxe « arithmétique » :

    `for ((EXPR1; EXPR2; EXPR3)); do COMMANDES; done`

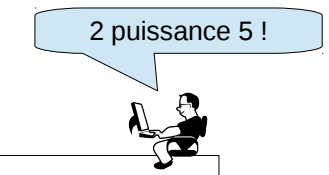
    Les trois expressions doivent être acceptables pour `let` (voir `help let`)

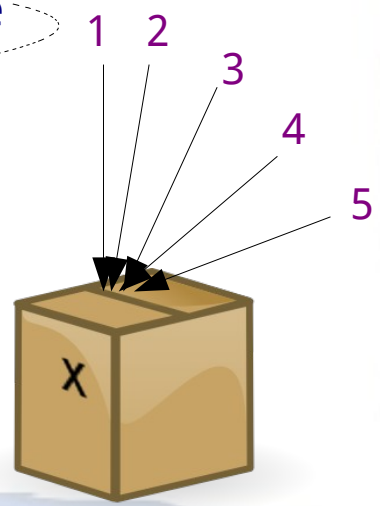
    EXPR1 initialise la **variable de boucle**

    EXPR2 test arithmétique d'aiguillage (succès => on continue)

    EXPR3 mise à jour de la variable de boucle avant le ré-aiguillage
  - Exemple :

```
N=1
for ((X=1; X<=5; X=X+1)); do
  N=$((N*2))
done
```





Action répétée 5 fois, pour X égal à 1 2 3 4 5

Notion n°6 : boucles avec condition (while, until) (1)

- **while** : on répète des actions plusieurs fois, **tant qu'une commande réussit**
  - c'est-à-dire : on répète tant qu'une condition est **satisfaite**, on sort dès qu'elle ne l'est plus
  - ```
    X=1
    while zenity --question --text="Continuer ?"; do X=$((X*2)); done
    ```
- **until** : on répète des actions plusieurs fois, **tant qu'une commande échoue**
  - C'est-à-dire : on répète tant qu'une condition est **insatisfaite**, on sort dès qu'elle le devient
  - ```
    X=1
    until zenity --question --text="Arrêter ?"; do X=$((X*2)); done
    ```

Notion n°6 : boucles avec condition (while, until) (2)

- **while** : on répète des actions plusieurs fois, **tant qu'une commande réussit**
  - Syntaxe (**help while**) : `while COMMANDE; do COMMANDES; done`
  - ```
    X=1
    while zenity --question --text="Continuer ?"; do X=$((X*2)); done
    ```
- **until** : on répète des actions plusieurs fois, **tant qu'une commande échoue**
  - Syntaxe (**help until**) : `until COMMANDE; do COMMANDES; done`
  - ```
    X=1
    until zenity --question --text="Arrêter ?"; do X=$((X*2)); done
    ```

Notion n°6 : boucles avec condition (while) (3)

- **while** : on répète des actions plusieurs fois, **tant qu'une commande réussit**

  - ```
    while COMMANDE; do
      COMMANDES;
    done
    ```

Commande **d'aiguillage** :

succès => aller à la partie « do », puis relancer à nouveau COMMANDE et aiguiller

échec => sortir, c'est-à-dire aller à la partie qui suit le « done »

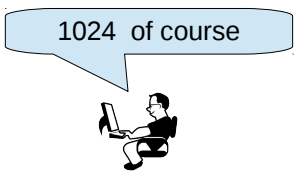
- **Exemples :**

  - ```
    while read X; do echo "Vous avez saisi la ligne : $X"; done
    ```

  - ```
    while read X; do echo "Ligne de foo.txt : $X"; done < foo.txt
    ```

  - ```
    N=1
    while read X; do echo "Ligne n°$N : $X"; N=$((N+1)); done < foo.txt
    ```

  - ```
    N=1
    while test $N -le 1000; do N=$((N*2)); done
    echo "La valeur de N à la sortie de la boucle est $N"
    ```


    ```

Notion n°6 : boucles avec condition (while) (4)

- **Remarque** : le « **for arithmétique** » est un **while** déguisé en **for**
  - `for ((EXPR1; EXPR2; EXPR3)); do COMMANDES; done`

    est équivalent à :

  - `let "EXPR1"; while let "EXPR2"; do COMMANDES; let "EXPR3"; done`
- **Exemple :**
  - ```
    N=1
    for ((X=1; X<=5; X=X+1)); do N=$((N*2)); done
    ```

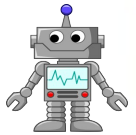
    est équivalent à :

  - ```
    N=1
    let "X=1"; while let "X<=5"; do N=$((N*2)); let "X=X+1"; done
    ```

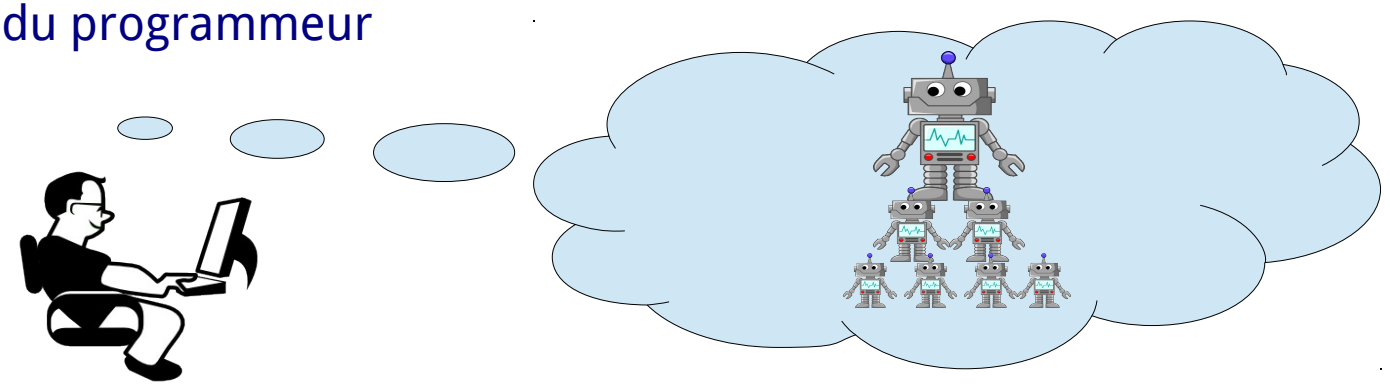
Notion n°7 : paramétrisation (des programmes ou sous-programmes) (1)

- Les programmes interactifs sont intéressants pour l'utilisateur :
  - ```
    JPG=$(zenity --file-selection --text="Choisissez un fichier jpeg")
    convert ${JPG} -resize 50% ${JPG%.jpg}.png
    ```

- mais... il n'y a que lui qui peut **en faire appel**, puisque il n'y a que lui pour répondre interactivement aux questions posées (clavier, souris,...)



- C'est bien aussi qu'un robot (programme) puisse faire appel à... un autre robot (sous-programme) déjà construit
  - C'est un principe de sous-traitance des services à rendre, qui facilite la tâche du programmeur


    ```

Notion n°7 : paramétrisation (des programmes ou sous-programmes) (2)

- Pour que les (sous-)programmes puissent se faire appel, on utilise les **paramètres d'appel** :
  - Tous les (sous-)programmes ont une sorte de « **casier** » où on peut déposer des informations (valeurs) utiles au service qu'ils doivent rendre
  - De cette façon, ils ne doivent pas poser de question (interagir) : les informations sont **dès le départ** dans le casier des paramètres
  - N'importe qui (utilisateur ou autre programme) peut faire appel à leur service en remplissant le casier avec les bonnes informations ; c'est en réalité un principe que vous connaissez très bien, puisque vous l'utilisez :

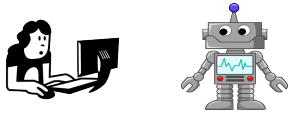
    ```
    convert rose.jpg -resize 50% rose.png
    ```

    Vous faites appel au programme `convert` en remplissant son casier :

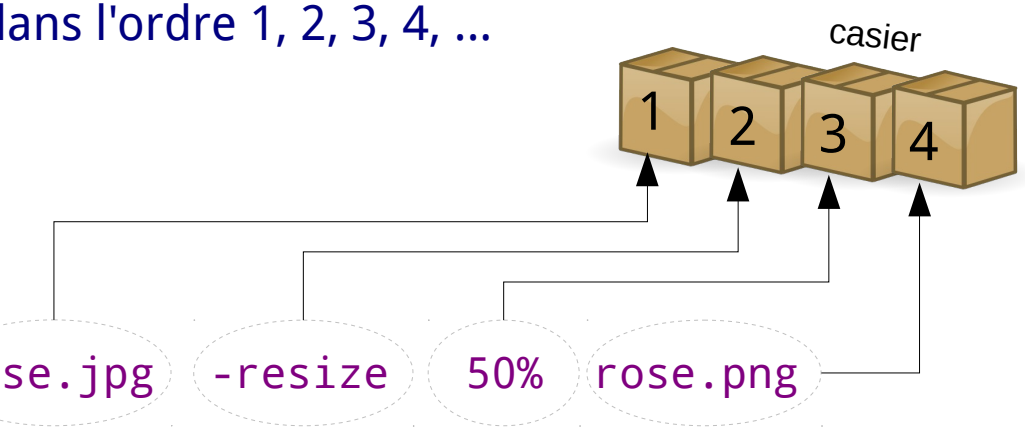
    - Dans la case n°**1** du casier vous mettez la valeur `"rose.jpg"`
    - Dans la case n°**2** du casier vous mettez la valeur `"-resize"`
    - Dans la case n°**3** du casier vous mettez la valeur `"50%"`
    - Dans la case n°**4** du casier vous mettez la valeur `"rose.png"`

Notion n°7 : paramétrisation (des programmes ou sous-programmes) (3)

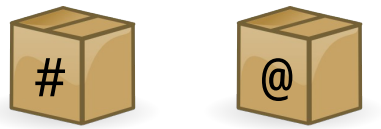
- Comment le développeur (Bash) peut manipuler les valeurs déposées dans le casier du programme par l'appelant (utilisateur ou autre programme) ?



  - Les cases du casier sont **comme des variables**, avec la différence qu'elles sont nommées dans l'ordre 1, 2, 3, 4, ...



  - Le développeur peut donc accéder à ces informations (qu'il ne connaît pas mais ce n'est pas un problème) par ${1} ${2} ${3} ${4} ...

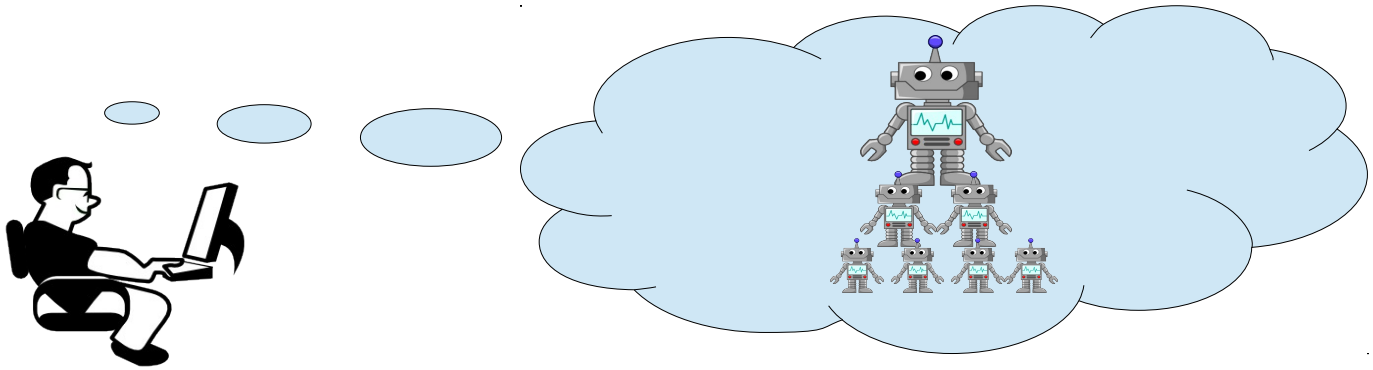


  - Autres variables intéressantes # et @ :
    $# : est le **nombre** de cases « remplies » par l'appelant
    $@ : est la **concaténation** de toutes les cases « remplies » par l'appelant

  -

Notion n°7 : paramétrisation (des programmes ou sous-programmes) (4)

- Grâce aux paramètres un programme peut faire appel à d'autres programmes (déjà construits) pour rendre son service, dans une logique de sous-traitance

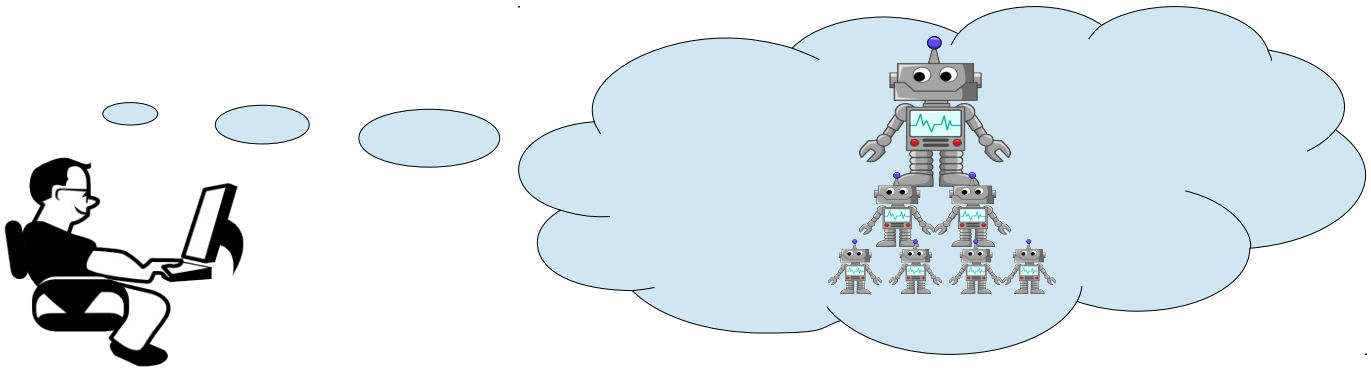


Terminologie :

- les paramètres traités par le développeur ($1, $2, ...) sont appelés **paramètres** (ou **arguments**) **formels**
- Les valeurs utilisées à l'appel (rose.jpg, -resize, ...) sont appelés **paramètres** (ou **arguments**) **actuels**

Notion n°8 : Sous-programmes internes (fonctions) (1)

- Grâce aux paramètres un programme peut faire appel à d'autres programmes (déjà construits) pour rendre son service, dans une logique de sous-traitance



- Or, il est **possible** mais n'est **pas nécessaire** que ces sous-traitants soient des programmes (fichiers exécutables) **indépendants**
- On peut aussi définir des sous-programmes **à l'intérieur** du programme lui même (dans le même fichier exécutable) :
  - Syntaxe de définition : `function NOM { COMMANDES ; }`
  - Syntaxe d'appel (habituelle) : `NOM ARG1 … ARGn`

Notion n°8 : Sous-programmes internes (fonctions) (2)

- Les fonctions sont des sous-programmes définissables **à l'intérieur** du programme lui même
  - Syntaxe de définition : `function NOM { COMMANDES ; }`
  - Syntaxe d'appel (habituelle) : `NOM ARG1 … ARGn`
- Les conventions sur les paramètres formels ($1, $2, ..., $#, $@) et actuels sont les mêmes que pour les exécutables indépendants

```
#!/bin/bash
# Définition de la fonction foo :
function foo { convert ${1} -resize 50% ${1%.jpg}.png ; }
# Interaction avec l'utilisateur :
JPG1=$(zenity --file-selection --text="Choisissez un fichier jpeg")
JPG2=$(zenity --file-selection --text="Choisissez un fichier jpeg")
# Appels de la fonction foo :
foo ${JPG1}
foo ${JPG2}
```

Adresse des images utilisées

- Boite fermée https://openclipart.org/detail/15872/closed-box-by-mcol
- Robot sympa https://openclipart.org/detail/170101/cartoon-robot-by-sirrob01
- Développeur https://openclipart.org/detail/37129/personnage_ordinateur-by-antoine
- Utilisateur https://openclipart.org/detail/37135/personnage_ordinateur-by-antoine-37135