Initiation à la programmation avec le shell Bash

Cours n°1

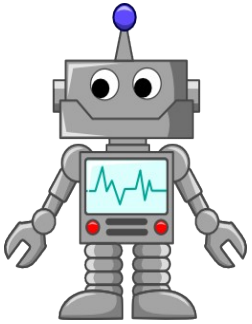
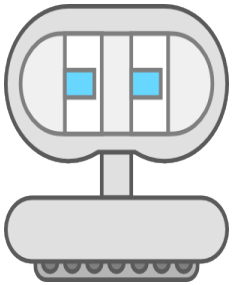
Jean-Vincent Loddo

Sommaire du cours n°1

- Notion n°1 : programmer = **automatiser un service**
- Notion n°2 : les **valeurs**
- Notion n°3 : les **variables**
- Notion n°4 : la **conditionnelle**

Notion n°1 : programmer = automatiser un service

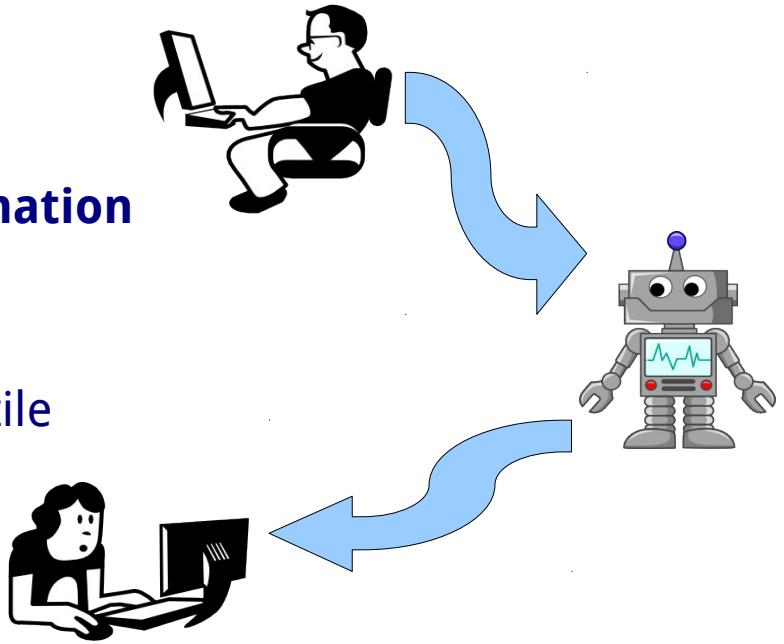
- On peut imaginer un **programme** comme un **robot**
- Même si il n'a pas un corps


- Et même si son ectoplasme est emprisonné dans une fenêtre graphique ou textuelle (**terminal**) d'un ordinateur
- Comme un robot : il fait un travail, il questionne l'utilisateur, il réagit aux réponses et autres stimuli (clavier, souris, réseau, etc)
- Comme un robot : quelqu'un le construit, quelqu'un l'utilise

Développeur → Programme → Utilisateur

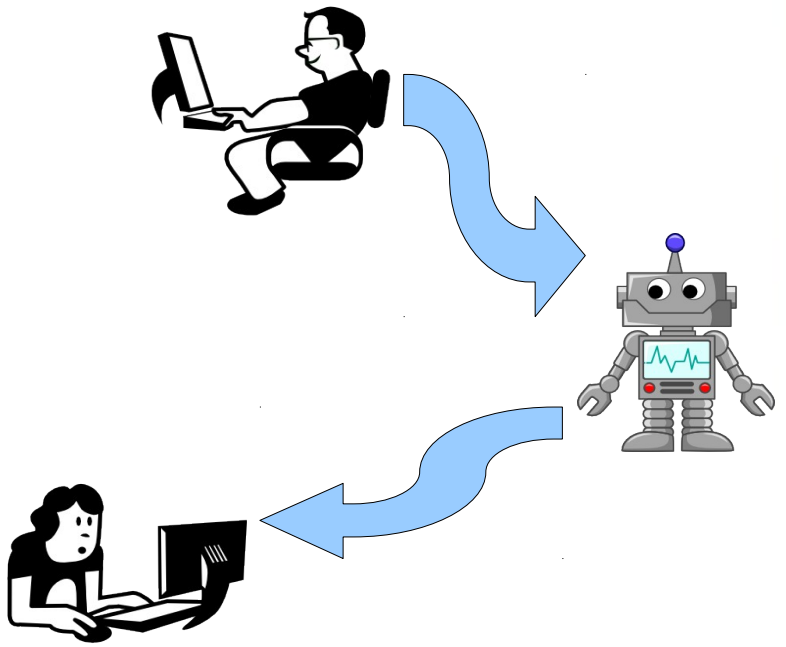
- Comme un robot : quelqu'un le construit, quelqu'un l'utilise
- Qui le construit ?
  - C'est le **programmeur** (ou **développeur**)
  - Comment : avec un **langage de programmation**
  - Combien de fois : une fois !
  - Pourquoi : parce qu'**il rendra** un service utile
- Qui l'utilise ?
  - C'est l'**utilisateur**
  - Comment : avec une interface (graphique ou textuelle)
  - Combien de fois : autant qu'il le souhaite
  - Pourquoi : parce qu'**il rend** un service utile



Développeur → Programme → Utilisateur

- Mais alors, apprendre à programmer c'est apprendre à **automatiser un service utile** ?

OUI



Développeur → Programme → Utilisateur

- Est-ce que le développeur peut être aussi l'utilisateur ?
- Ce n'est pas sa vocation et ce n'est pas souvent le cas, mais c'est possible
- Sauf... lorsque le développeur **teste** (avant de le livrer) si le programme rend effectivement le service qu'on attend de lui
- Un peu de terminologie à propos :
  - si le comportement est erroné on dit que le programme a un **bug** (ou **bogue**)
  - l'activité qui consiste à corriger un programme s'appelle **debugging** (ou **debogage**)

Exemple de programme (1)

- **Service à rendre** : afficher la date et encourager l'utilisateur à travailler (quitte à lui écrire quelques petits mensonges)

- ```
  #!/bin/bash
  date
  echo "Je vous souhaite une belle journée"
  echo "Le $(date +%A) est le meilleur jour de la semaine !"
  echo "En plus, le $(date +%j)-ème jour de l'année est l'un des plus propices aux bonnes surprises."
  ```
  ```

Exemple de programme (2)

- **Service à rendre** : afficher la date et encourager l'utilisateur à travailler (quitte à lui écrire quelques petits mensonges)

```
#!/bin/bash
date
echo "Je vous souhaite une belle journée"
echo "Le $(date +%A) est le meilleur jour de la semaine !"
echo "En plus, le $(date +%j)-ème jour de l'année est l'un des plus propices aux bonnes surprises."
```

1ère ligne du programme (ou "script"). Ne pas oublier non plus de rendre le fichier exécutable (chmod +x)

Par exemple : le mardi 23/09/2014
$(**date** +%A) sera remplacé par "mardi"
et $(**date** +%j) sera remplacé par "266"

Substitutions de commandes : la partie $(COMMANDE) est remplacée par le résultat (**sortie standard**) de COMMANDE

Remarque : ce programme fait appel aux programmes **date** (3 fois) et **echo** (3 fois) : le service rendu est donc un **assemblage** de sous-services rendus par des programmes pré-existants

Exemple de programme (3)

- **Service à rendre** : afficher la date et encourager l'utilisateur à travailler (quitte à lui écrire quelques petits mensonges)

- Exécution du programme, que nous avons nommé date_et_encouragements.sh, rendu exécutable (chmod +x) et exécuté le mardi 23/09/2014 :

- ```
  $ ./date_et_encouragements.sh
  mardi 23 septembre 2014, 12:50:32 (UTC+0200)
  Je vous souhaite une belle journée
  Le mardi est le meilleur jour de la semaine !
  En plus, le 266-ème jour de l'année est l'un des plus propices aux
  bonnes surprises.
  ```
  ```

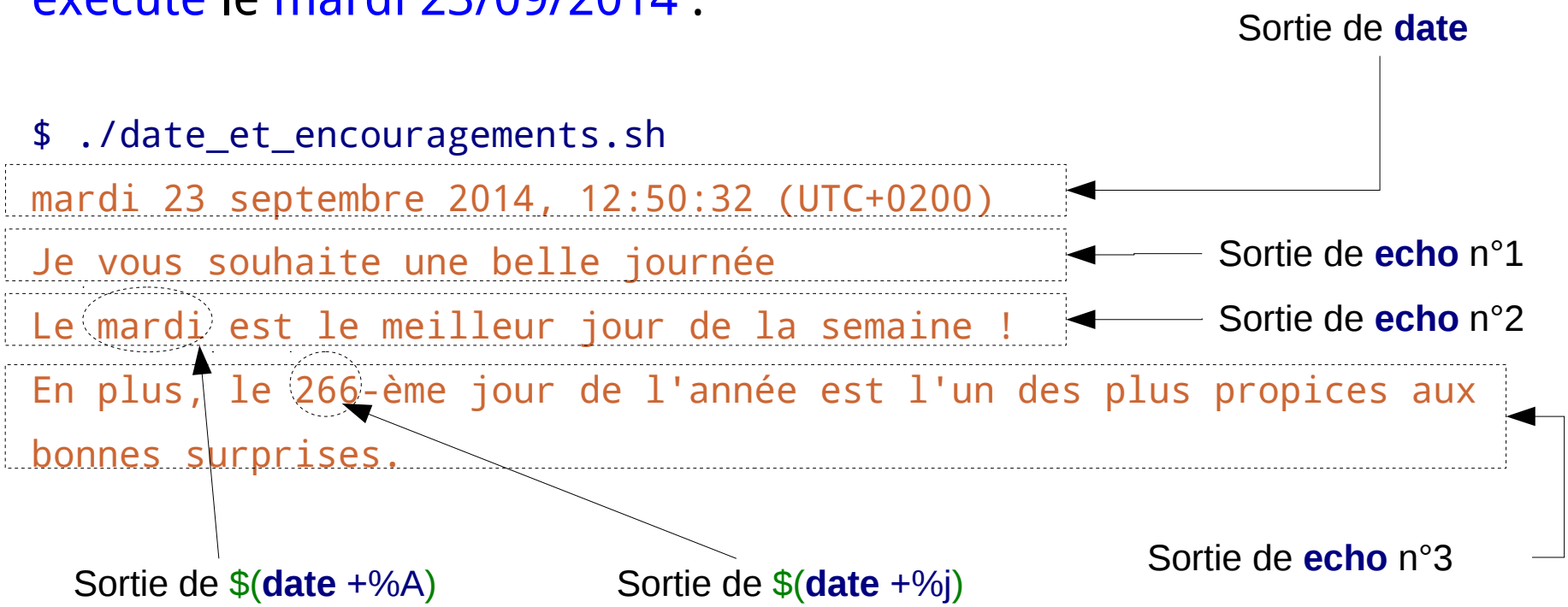
Exemple de programme (4)

- **Service à rendre** : afficher la date et encourager l'utilisateur à travailler (quitte à lui écrire quelques petits mensonges)

- Exécution du programme, que nous avons nommé date_et_encouragements.sh, rendu exécutable (chmod +x) et exécuté le mardi 23/09/2014 :

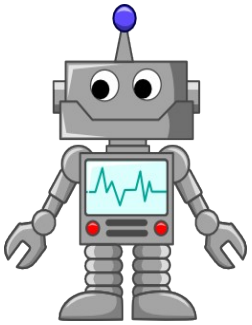
- `$ ./date_et_encouragements.sh`

```
mardi 23 septembre 2014, 12:50:32 (UTC+0200)
Je vous souhaite une belle journée
Le mardi est le meilleur jour de la semaine !
En plus, le 266-ème jour de l'année est l'un des plus propices aux
bonnes surprises.
```



Notion n°1 : programmer = automatiser un service

- On peut imaginer un **programme** comme un **robot**



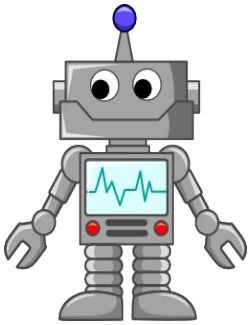
- Ok, ok, mais emprisonné dans la fenêtre d'un ordinateur il ne pourra pas faire le **ménage** ! Ni **repasser le linge** !

- Alors, que peut-il rendre comme service intéressant ?
  - récupérer des informations (p.e. sur Internet, dans des fichiers, ou données par l'utilisateur), présenter des informations (p.e. la date), calculer des informations

- Autrement dit : un programme **élabore des informations**

Notion n°1 : programmer = automatiser un service

- Admettons : un programme **élabore des informations**



- Quels types d'information sont traitées ?
  - Textes ? Nombres ? Dates ? Noms de fichiers ? Adresses Internet ? Autre chose ?
  - C'est la notion de **valeur**
  - Et ça dépend du langage de programmation...

Notion n°2, les informations ou **valeurs** élaborées

- Les **valeurs** d'un langage de programmation sont les informations que les programmes sont capables de traiter
- Par exemple :
  - Nombres entiers (0 42 -16 100)
  - Nombres flottants (3.14159 2.71828)
  - Booléens (true false)
  - Caractères (a z A Z 0 9 #)
  - Chaînes de caractères (salut HeLLo /etc/bashrc)
  - Tableaux, listes, arbres, ...

Les **valeurs** en Bash (1)

- En Bash c'est bien simple : les informations manipulées ne sont que des **chaînes de caractères** !
- En Bash, tout est chaînes de caractères :
  - `echo "salut le monde"`
  - `echo "salut" > "/tmp/foo.txt"`
  - `ls "/tmp/"`
- On utilise des guillemets pour délimiter (mais s'il n'y a pas de blancs, on peut les éviter) :
  - `echo salut > /tmp/foo.txt`
  - `ls /tmp/`

Les **valeurs** en Bash (2)

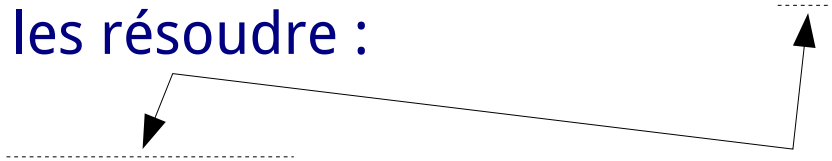
- En Bash les informations manipulées sont des chaînes de caractères
- **Pourquoi** ?
- **Parce que** Bash est avant tout un **interpréteur de commandes** :
  - `ls -l /home`
  - `head -n 20 /home/secrets | grep "42"`
  - `find /etc -name "*passwd*" > bingo`
- Et les **commandes** sont des chaînes de caractères (**phrases**, séparées par des retour-chariot), composées des sous-chaînes de caractères (**mots**, séparées par des blancs)

Les **valeurs** en Bash (3)

- Prenons par exemple :
  - `find /etc -name "*passwd*" 1> bingo`
- Le nom du programme à exécuter (find) est une chaîne
- Les paramètres fournis sont des chaînes (/etc -name *passwd*)
- L'indicateur de redirection (1>) et le nom du fichier de redirection (bingo) sont des chaînes
- Le résultat de la commande est une chaîne (elle sera écrite dans le fichier bingo)
- La commande toute entière est une chaîne (phrase) avec plusieurs mots (six) séparés par des blancs

Les **valeurs** en Bash (4)

- Il est vrai cependant qu'on peut travailler avec les **nombres entiers**, à condition de demander explicitement à Bash d'interpréter certaines chaînes, entourées par l'opérateur $((?)), comme des **formules arithmétiques** et de les résoudre :



  - `echo "Le résultat est $((10*(7+4)/2))"`

se transforme avant d'être véritablement exécutée en :

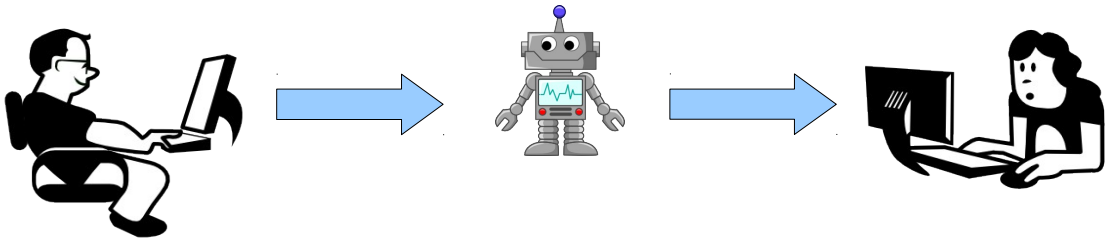
  - `echo "Le résultat est 55"`

Les **valeurs** en Bash

- Que fait-on avec les chaînes de caractères ?
- Ce qu'on fait tout le temps avec les mots et les phrases : on les rassemble, c'est-à-dire qu'on les « colles » les unes aux autres (**concatenation** ou **juxtaposition**)
- Pour créer des commandes (séparateur blanc) :
  - `tar -cvzf foo.$((10*(7+4)/2)).tgz /usr /var`
- Pour créer des fichiers (séparateur retour-chariot \n) :
  - `echo "Les infos secrètes"      > foo.txt`
  - `echo "sont :"                 >> foo.txt`
  - `lynx -dump http://hacker.zone >> foo.txt`

Résumé pour l'instant et prochaine question...

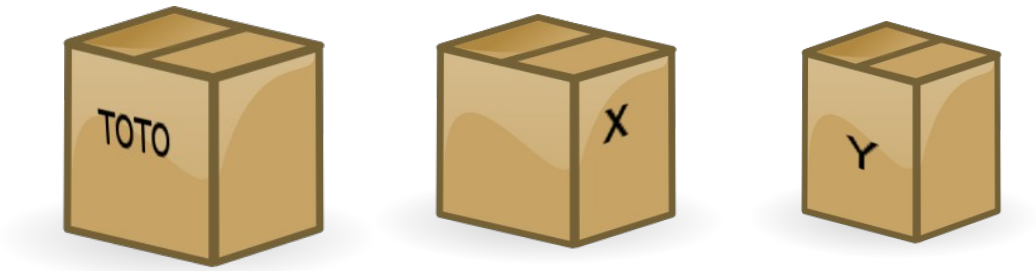
- Un programme est comme un robot sauf que son job est de manipuler des informations, par exemple des chaînes de caractères (Bash)
- Un programmeur le construit, un utilisateur l'utilise



- Parce que cela rend service
- Mais comment le programmeur peut planifier le comportement du robot sans savoir à quel moment et par qui il sera utilisé ?
  - Imaginons ce **service à rendre** : le robot doit demander l'age de l'utilisateur et lui écrire ensuite une gentillesse du style « c'est génial d'avoir ... ans »
  - Comment le programmeur peut remplir les pointillés ?

Notion n°3, les **variables**

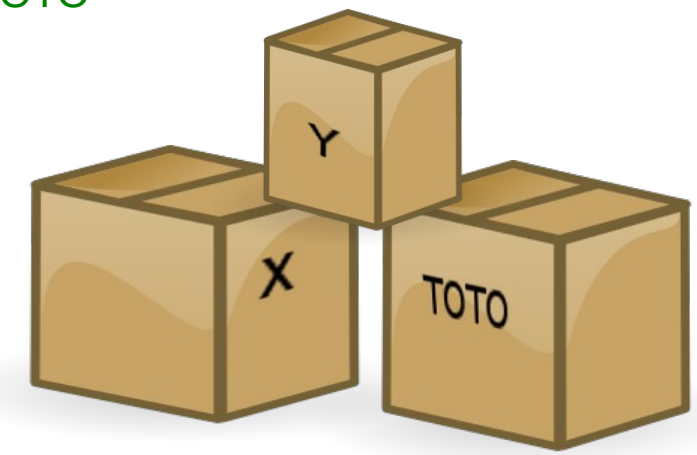
- Pour traiter l'information que le programmeur connaît mais surtout celle qu'il **ne connaît pas**, le langage de programmation propose les « variables »
- Les variables sont des boîtes qui ont un **nom** et un **contenu**



- Le contenu est une **valeur**, c'est-à-dire une information traitée par le langage de programmation
- Comment on stocke une information dans une boîte ?
- Comment on la récupère ?

Variables : comment on stocke une information dans une variable ?

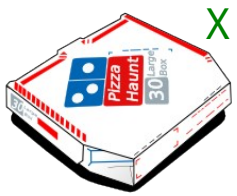
- Par lecture des caractères saisis au clavier par l'utilisateur.
  En C cela se fait avec **scanf**, en Bash avec **read** :
  - `read X` ← Toute la phrase saisie dans X
  - `read X Y` ← Le premier mot dans X, le reste de la phrase saisie dans Y
  - `read X Y TOTO` ← Les deux premiers mots dans X et Y, le reste de la phrase saisie dans TOTO
- Par **affectation** du contenu :
  - `TOTO="salut le monde"`
  - `X="aaa $(ls /etc) zzz"`
  - `Y=16`

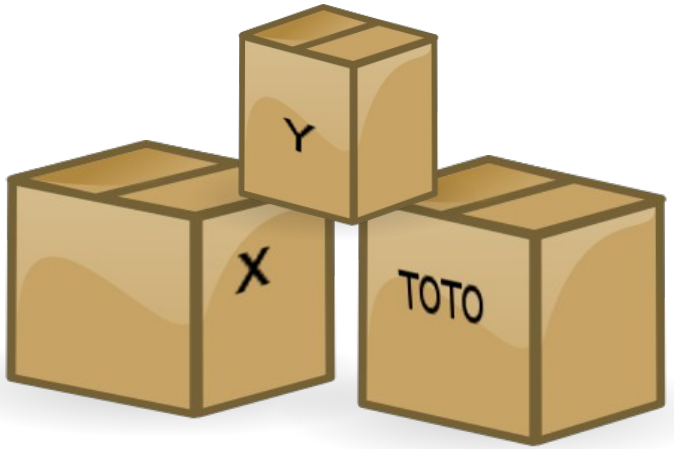


Variables : comment on récupère l'information stockée dans une variable ?

- En Bash, par l'opérateur ${?}

  où ? est le **nom** de la boîte à ouvrir





- Exemple :

```
TOTO="La réponse"
Y=42
echo "${TOTO} à la question ultime est ${Y}"
```

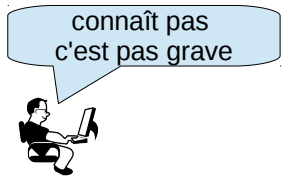
- Affiche :
  - La réponse à la question ultime est 42

Variables : remarque sur les informations manipulées

- Le **développeur** programme (écrit) cette ligne en **1999** :
  - `read X`

- Un **robot** (une instance du programme) est exécuté en **2017**
- L'**utilisateur** saisi une chaîne de caractères en **2017**

- Le **robot** stocke cette chaîne dans la variable X en **2017**
- Le **développeur**, toujours en **1999**, ne connaît pas la chaîne saisie
  - mais il sait qu'elle se trouve dans X
  - il peut donc y accéder par ${X} dans la suite du programme
- Le développeur planifie aussi bien le traitement de l'information qu'il **connaît** que de celle qu'il **ne connaît pas**



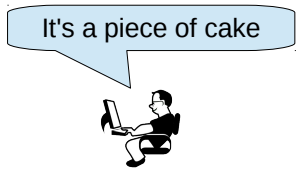
Variables : remarque (on insiste) sur les informations manipulées

- Le **développeur** programme (écrit) cette ligne en **1999** :
  - `X="aaa $(ls /etc) zzz"`
- Un **robot** (une instance du programme) exécute en **2017** :
  - `ls /etc`
  - `X="aaa  zzz"` où est le résultat (sortie) de
- Le **développeur**, toujours en **1999**, ne connaît pas exactement la chaîne qui sera affectée (juste le début aaa et la fin zzz)
  - mais il sait qu'elle se trouve dans X
  - il peut donc y accéder par ${X} dans la suite du programme

Un petit robot bien gentil

- Revenons sur la question : comment le programmeur peut planifier le comportement du robot sans savoir à quel moment et par qui il sera utilisé ?
  - Imaginons ce **service à rendre** : le robot doit demander l'âge de l'utilisateur et lui écrire ensuite une gentillesse du style « c'est génial d'avoir ... ans »
  - Comment le programmeur peut remplir les pointillés ?

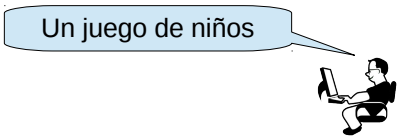
```
- echo "Saisissez votre âge svp"
- read AGE
- echo "C'est génial d'avoir ${AGE} ans"
```



Un robot pour tout âge

- Supposons à présent de vouloir écrire une phrase différente en fonction de l'âge. **Service à rendre** :
  - Si l'utilisateur a moins de 13 ans le robot devra écrire « Alors vous jouez à Call of Duty »
  - Sinon il écrira « Alors vous jouez à Pokemon »
- Comment faire ?
  - Avec la construction (commande) **if-then-else** !

```
read AGE
if  test ${AGE} -lt 13 ;  then
  echo "Alors vous jouez à Call of Duty"
else
  echo "Alors vous jouez à Pokemon"
fi
echo "En tous cas c'est génial d'avoir ${AGE} ans"
```



Notion n°4, la « conditionnelle »

- Il est possible de planifier des actions **conditionnelles** en utilisant un test :
  - Si le test a **succès**, le robot exécutera certaines actions
  - Si le test **échoue**, le robot exécutera d'autres actions
  - **le test est lui-même une commande (« d'aiguillage »)**, en profitant du fait que toutes les commandes Unix peuvent réussir (code de sortie 0) ou échouer (code de sortie différent de 0)

Commande **d'aiguillage**

- `if COMMANDE1; then`

  `COMMANDES2`

  Commandes exécutées en cas de **succès** du test

  `else`

  `COMMANDES3`

  Commandes exécutées en cas d'**échec** du test

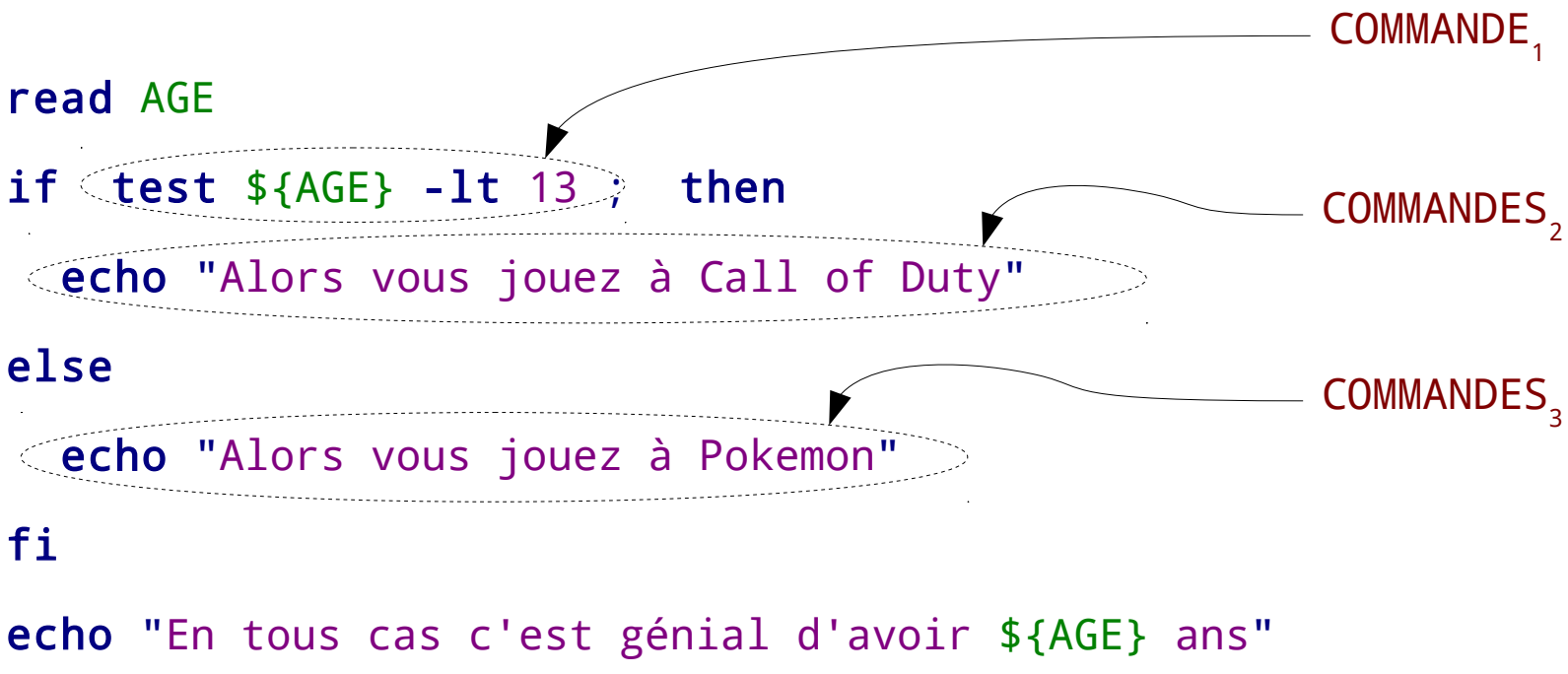
  `fi`

Un robot pour tout âge

- Syntaxe

```
if COMMANDE1; then COMMANDES2 ; else COMMANDES3 ; fi
```

- Par rapport à notre exemple :




```
read AGE
if test ${AGE} -lt 13 ; then
 echo "Alors vous jouez à Call of Duty"
else
 echo "Alors vous jouez à Pokemon"
fi
echo "En tous cas c'est génial d'avoir ${AGE} ans"
```

Remarque : la commande **test** (ou **[[** … **]]**) est parfaite comme commande d'aiguillage (voir **help test**) mais n'est pas la seule possible (**grep awk find ls tar** …) même si c'est la plus courante

Un robot pour tout âge
Le mot de la fin (du 1er cours)

```
read AGE
if  test ${AGE} -lt 13 ;  then
  echo "Alors vous jouez à Call of Duty"
else
  echo "Alors vous jouez à Pokemon"
fi
echo "En tous cas c'est génial d'avoir ${AGE} ans"
```

Non seulement on peut stocker (dans une variable)
une information **qu'on ne connaît pas**,
mais on peut aussi **traiter** cette information
en adaptant le comportement du robot aux différentes possibilités

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
- Robot chenille https://openclipart.org/detail/168755/cartoon-robot-by-qubodup
- Laptop https://openclipart.org/detail/24817/-by--24817
- Développeur https://openclipart.org/detail/37129/personnage_ordinateur-by-antoine
- Utilisateur https://openclipart.org/detail/37135/personnage_ordinateur-by-antoine-37135
- Pizza box https://openclipart.org/detail/171767/pizza-haunt-by-jakoriginal-171767
- Pizza https://openclipart.org/detail/189439/pepperoni-pizza-by-toons4biz-189439