Nono le petit robot

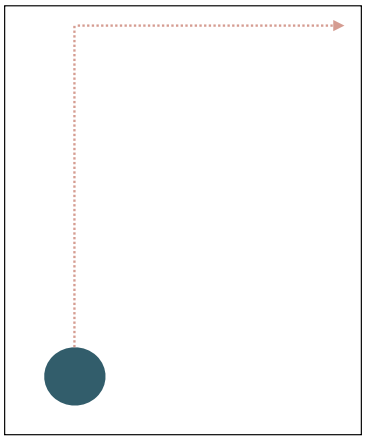
- Nono est un robot qui sait se déplacer seul, un peu comme R2D2.
- Ses compétences sont les suivantes :
  - Il peut initialiser, modifier, comparer des registres avec des constantes ou d'autres registres.
  - Il sait **avancer**, **pivoter**, tourner à **gauche** à **droite**, s'orienter au **nord**, à l'**ouest**, à l'**est** au **sud**, comme Peter, sauf que ses déplacements se font **un pas à la fois**.
  - Il sait également faire des répétitions avec un compteur.
  - Il dispose en outre d'une opération **toucher** qui lui indique s'il est à moins d'un pas d'un obstacle (détails plus loin…).

Nono le petit robot

- Les algorithmes que nous allons concevoir ici consistent à expliquer à Nono comment se déplacer pour :
  - aller dans une autre pièce en passant par une porte.
  - se placer à un endroit précis d'une pièce dont on connaît la configuration.
- Dans ces algorithmes, nous verrons également un autre type de répétition qui ne repose pas sur un compteur mais sur une **condition**.

Premier exemple : énoncé

- Nono est dans une pièce rectangulaire dont il ignore les dimensions. Au départ, il ne connaît pas sa position, mais il sait qu'il doit aller se placer dans le coin supérieur droit de la pièce.



Premier exemple : début de **l'algorithme**

- On voit qu'il faut que Nono aille d'abord sur le mur nord, puis se déplace jusqu'au mur est.

- Il faut donc réfléchir au sous-problème : « se déplacer jusqu'à un mur, quelle que soit sa position initiale ».

Des idées ?

Premier exemple : suite de l'algorithme

- Comme on ne sait pas combien de pas il faut faire pour atteindre le mur nord, on ne peut pas utiliser de compteur de pas.

- Par contre, on dispose des opérations « avancer d'un pas » et « toucher »…

- L'opération **toucher()** est simple : quand on l'exécute, elle met « **vrai** » dans le registre **obstacle** si Nono est à moins d'un pas d'un obstacle. Sinon, elle met « **faux** ».

Premier exemple : résumé…

- Il faut orienter Nono vers le nord : on sait le faire avec l'opération **nord()**.
- Il faut *avancer jusqu'au mur*.
- Il faut orienter Nono vers l'est : on sait le faire avec l'opération **est()**.
- Il faut *avancer jusqu'au mur*.

Si l'on sait résoudre « avancer jusqu'au mur », on a donc fini.

Avancer jusqu'au mur

- Pour avancer jusqu'au mur, il suffit de se mettre à la place de Nono, c'est-à-dire d'un aveugle qui avance avec ses bras tendus jusqu'à toucher le mur…

- Autrement dit : « tant qu'il n'y a pas d'obstacle, avancer d'un pas ». Mais, pour savoir s'il y a un obstacle, il faut utiliser **toucher()**.

- Dans le langage de Nono, cela s'écrit :

en algorithmique :
tant que obstacle = faux faire

```
toucher()
while obstacle == faux:
    avancer()
    toucher()
```

Boucle while (tant que)

- La boucle « **tant que** *condition* **faire** *instructions* » est une autre forme algorithmique pour exprimer la répétition d'une suite d'actions.

```
for cpteur_pas in range(nb_pas):
   avancer()
```

```
toucher()
while obstacle == faux:
    avancer()
    toucher()
```

Boucle while (tant que)

- Dans un **pour**, nous avons vu que ce qui pilotait la répétition était la valeur d'une variable servant de **compteur**.

```
for cpteur_pas in range(nb_pas):
    avancer()
```

- Dans un **tant que**, c'est la **condition** qui pilote la répétition : si cette condition est vraie, on répète. Quand la condition est fausse, on s'arrête.

```
toucher()
while obstacle == faux:
    avancer()
    toucher()
```

Essayez de trouver des phrases de tous les jours dans lesquelles vous utilisez un tant que… Vous verrez, il y en a beaucoup !

Boucle while (tant que)

- Pour « entrer dans une boucle tant que », il faut donc que sa condition soit vraie : si elle est fausse dès le départ, on ne fait aucune répétition.

- Réciproquement, si on est entré dans une boucle tant que, c'est parce que la condition était vraie…

- Pour en ressortir, il faudra donc que le contenu de la boucle, à un moment donné, remette cette condition à faux : sinon, on est dans une boucle sans fin.

Boucle while (tant que)

- Vérifions que notre boucle fonctionne dans tous les cas :
  - Si, dès le départ, Nono est contre le mur nord : le premier **toucher()** met **obstacle** à **vrai** et on ne rentre donc pas dans la boucle. Il n'y a pas de déplacement de Nono vers le nord.
  - S'il n'était pas contre le mur nord, le premier **toucher()** met **obstacle** à **faux** : on entre donc dans la boucle dont on ne sortira que lorsque **toucher()** mettra **obstacle** à **vrai**, c'est-à-dire lorsqu'on sera contre le mur.
  - Même principe après avoir pivoté vers l'est…

```
toucher()
while obstacle == faux:
    avancer()
    toucher()
```

Boucle while (tant que)

- Étudiez, par exemple, cet extrait :

```
age = 20                # on initialise age à 20
while age < 50:         # age est inférieur à 50 donc on entre dans la boucle
    age = age + 1       # à un moment donné, age va forcément arriver à 50…
                        # La condition deviendra fausse et on sortira du while
```

- et celui-ci :

```
age = 70                # on initialise age à 70
while age < 50:         # age est supérieur à 50, donc on ne fait jamais la boucle
    age = age + 1       # ce code ne sera jamais exécuté…
```

Bilan : for ou while ?

- Il est important de bien différencier ces deux types de répétitions et de choisir celle qui convient.

- On choisit un **for** quand **on sait d'avance le nombre de répétitions que l'on va faire** : *avancer 10 fois d'un pas*, par exemple, ou *frapper 4 fois à la porte*.

- On choisit un **while** dans le cas contraire : on sait juste que la boucle devra être répétée tant qu'une certaine condition n'est pas réalisée : *avancer d'un pas tant qu'on n'a pas rencontré d'obstacle*, par exemple, ou *tant que la porte ne s'ouvre pas, frapper*.

Premier exemple : fin…

- Vous savez tout ce que vous avez besoin de savoir pour écrire le programme de déplacement de Nono dans le coin supérieur droit…
- À vous de jouer : écrivez l'algorithme complet et traduisez-le dans le langage de Nono.
- Malheureusement, vous ne pourrez pas exécuter ce programme sur les machines car elles ne savent pas simuler Nono : c'est vous qui devrez jouer le rôle de Nono en lisant le programme et en l'exécutant "sur le papier".

Sous-programmes : premiers indices…

- Vous aurez probablement écrit l'algorithme suivant :

```
S'orienter au nord
Avancer jusqu'au mur
S'orienter vers l'est
Avancer jusqu'au mur
```

- Les compétences de Nono vous permettent de traduire directement "S'orienter vers le nord" par **nord()** et "S'orienter vers l'est" par **est()**.

- Nous avons vu comment détailler "Avancer jusqu'au mur" pour les compétences de Nono.

Sous-programmes : premiers indices…

- Ce qui devrait donner le programme suivant :

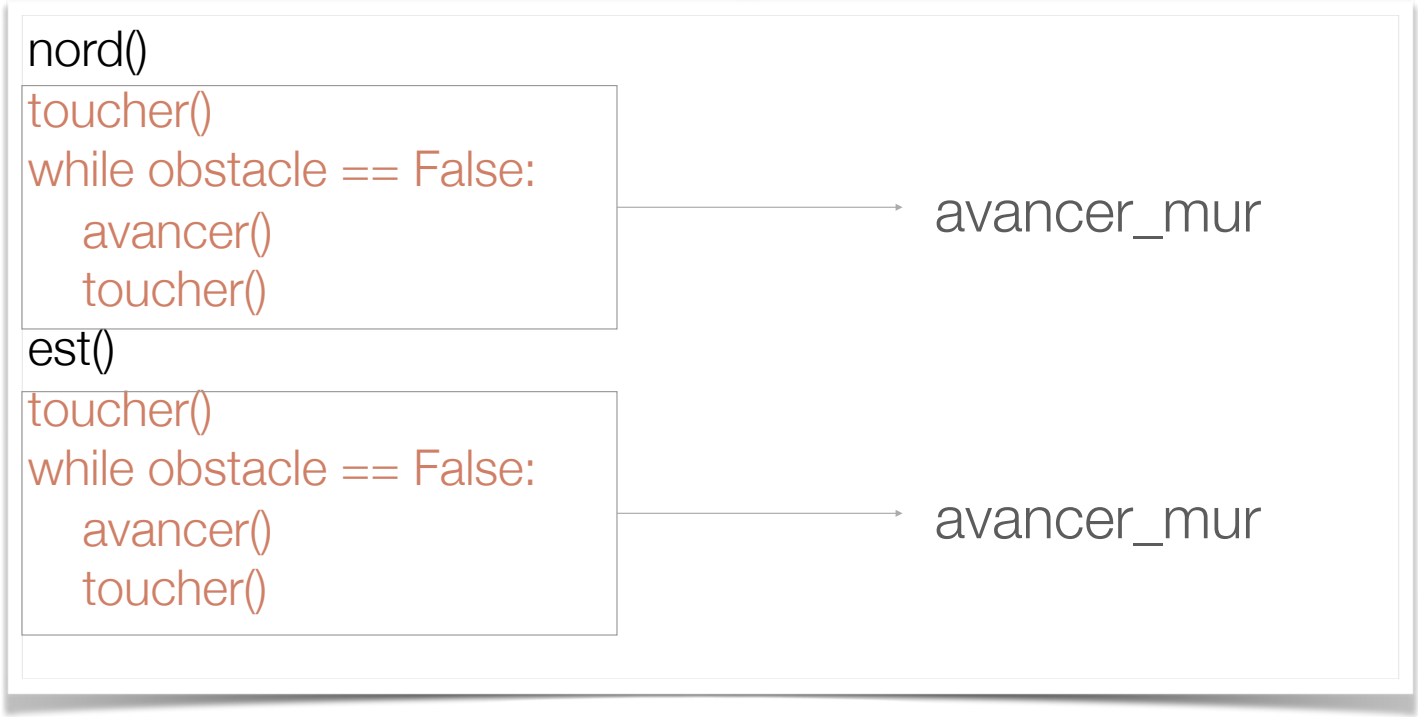
```
nord()
toucher()
while obstacle == False:
    avancer()
    toucher()
est()
toucher()
while obstacle == False:
    avancer()
    toucher()
```

S'orienter au nord
Avancer jusqu'au mur
S'orienter vers l'est
Avancer jusqu'au mur

- On remarque que l'on répète à l'identique les 4 mêmes lignes…

Sous-programmes : premiers indices…

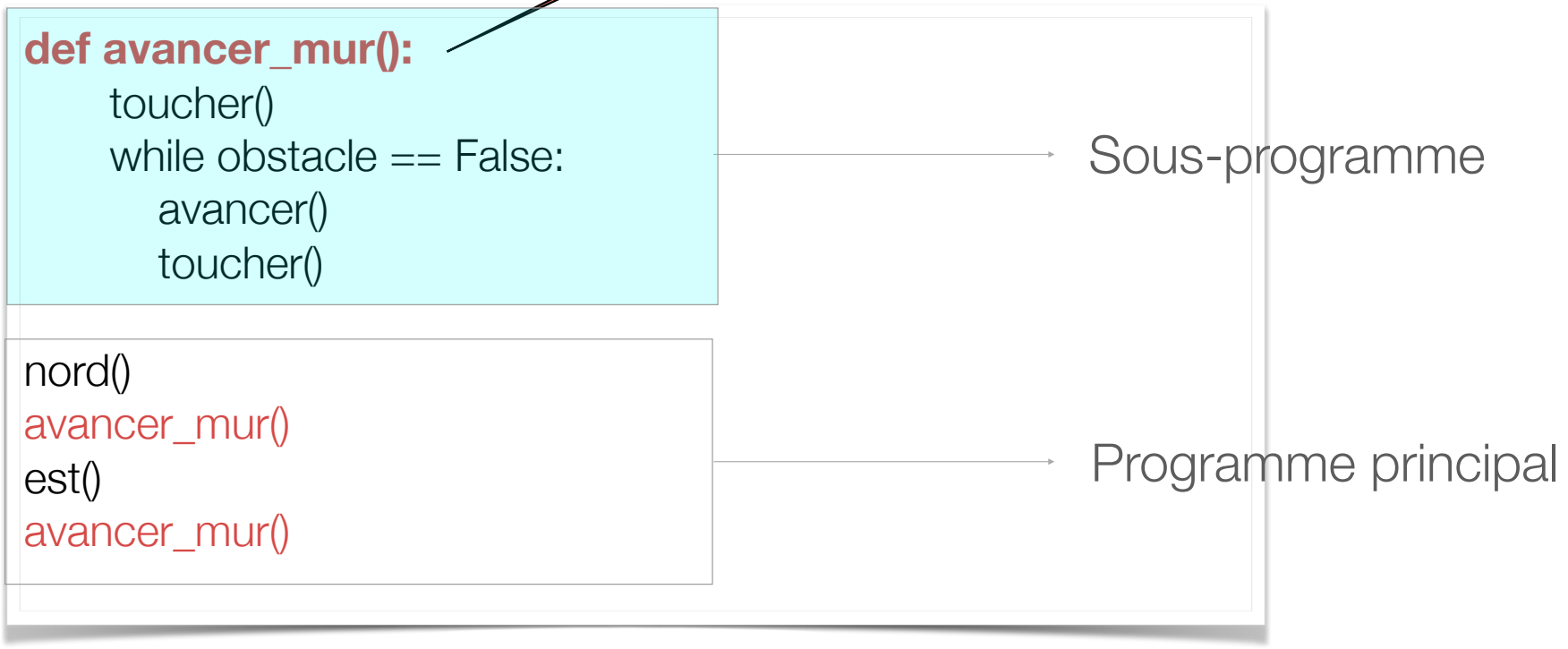
- Pourquoi ne pas donner un nom à ces 4 lignes de code et réutiliser ce nom à chaque fois que l'on en a besoin ?



Sous-programmes : premiers indices…

attention à la syntaxe !

- Les 4 lignes de code définissent un **sous-programme** et le programme principal fait appel au sous-programme chaque fois qu'il en a besoin :



Sous-programmes : premiers indices…

- Avec cette approche, le programme principal est le reflet exact du premier niveau de l'algorithme.
- La création d'un sous-programme permet de détailler un sous-problème indépendamment des autres.
- Elle évite de dupliquer le code et rend le programme plus modulaire et plus lisible.
- Ici, l'ajout d'un sous-programme peut être vu comme l'*ajout d'une compétence* au robot : il sait maintenant se déplacer jusqu'au mur qui est dans la direction courante.

Sous-programmes : premiers indices…

- Un sous-programme peut avoir des *paramètres*.
- Par exemple, revenons à Peter :
  - supposons qu'il ne sait avancer que d'un seul pas avec **avancer()**
  - on aurait pu créer un sous-programme pour lui ajouter la compétence "avancer de n pas" :

```
def avancer_de_n_pas(nb_pas):
    for x in range(nb_pas):
        avancer()
```

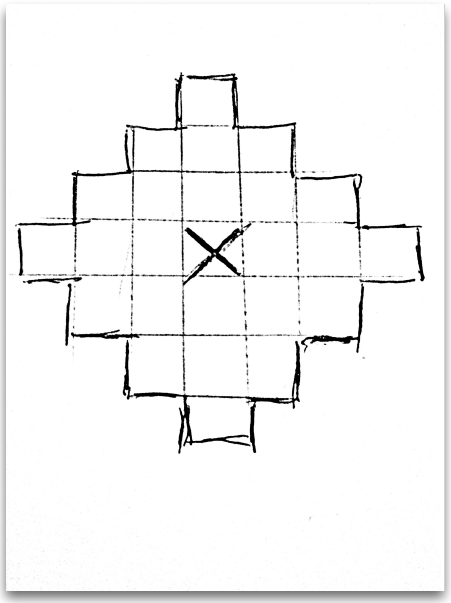
- Ici, le sous-programme attend un paramètre : le nombre de pas dont il doit avancer.

Sous-programmes : premiers indices…

- Nous reviendrons sur les sous-programmes car c'est un élément essentiel de la programmation.
- En attendant, écrivez les algorithmes puis les programmes pour les exercices suivants en essayant d'identifier les sous-problèmes que vous traduirez par des sous-programmes.

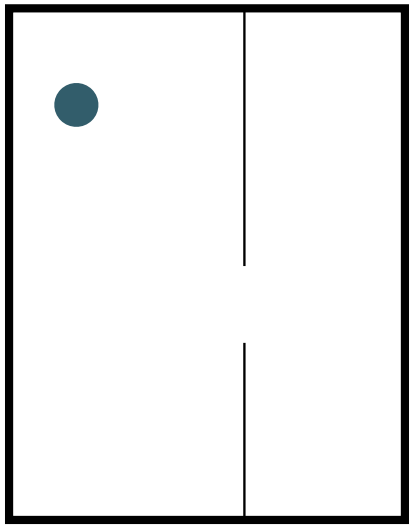
Deuxième exemple

- La pièce a maintenant la configuration suivante (chaque "case" fait un pas de côté) :



- Au départ, Nono est n'importe où et il faut le faire arriver sur la case marquée d'une croix. Écrivez l'algorithme puis le programme utilisant les compétences de Nono.

Troisième exemple



- Au départ, Nono est n'importe où dans la pièce de gauche : il faut le faire arriver dans la pièce de droite.

- La porte est n'importe où (***sauf aux extrémités du mur de séparation***) et fait un pas de large