Structures de contrôle
Deuxième partie

Exercices de mise en train

Exercice 1 : Sujet créé par France-IOI

Vous arrivez devant un pont que vous devez absolument emprunter pour arriver avant la nuit au village situé de l'autre côté de la rivière. Cependant, la traversée du pont n'est pas gratuite et le tarif dépend de votre chance au jeu. En effet, le gardien va lancer deux dés et le prix dépendra des valeurs que vous obtiendrez. Vous décidez d'écrire un programme pour vérifier qu'il applique bien le bon tarif. Pour l'instant vous n'écrirez que l'algorithme de ce programme.

Ce que votre programme doit faire :

Votre programme doit lire deux entiers, compris entre 1 et 6, la valeur de chaque dé. L'algorithme doit vérifier que les valeurs fournies, sont correctes.

Si la somme est supérieure ou égale à 10, alors vous devez payer une taxe spéciale (36 pièces). Sinon, vous payez deux fois la somme des valeurs des deux dés. Votre programme devra afficher selon le cas le texte « Taxe spéciale ! » ou bien « Taxe régulière », puis la somme à payer (sans indiquer l'unité).

Exercice 2 : Le jeu « Trouver le nombre ». L'ordinateur choisit un nombre au hasard entre 0 et 20. Vous devez trouver ce nombre.

Pour cette mise en train, votre algorithme AlgoBox fonctionnera de la façon suivante :

- Un nombre entre 0 et 20 est mémorisé : Algobox offre une fonction ALGOBOX_ALEA_ENT(p,n) qui renvoie un nombre entier pseudo-aléatoire compris entre p et n .
- L'utilisateur propose un nombre
- Si le nombre proposé est égal au nombre mémorisé, l'algorithme répond « Vous avez trouvé », sinon il répond « Plus petit » ou « Plus grand » suivant que le nombre proposé est supérieur ou inférieur au nombre mémorisé.
- Il affiche le nombre qu'il fallait trouver.

Exercice 3 : Étude d'un programme

Simulation d'un lancer de dés avec Scratch

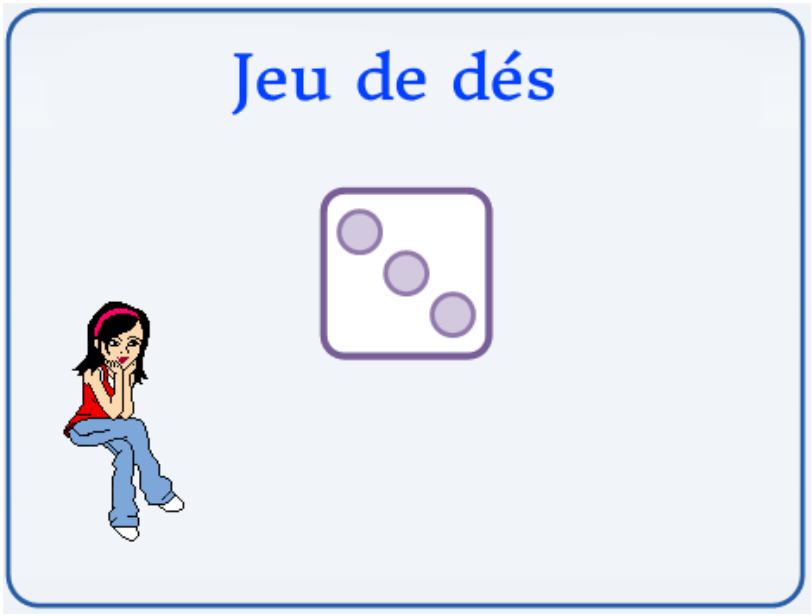
Dans Scratch, chaque lutin peut envoyer un message ou envoyer un message et attendre.

Ces messages arrivent chez tous les lutins (même celui qui a émis le message).

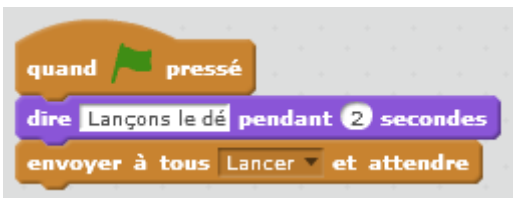
Les lutins peuvent réagir à un message particulier, s'ils se sont placés en attente de ce message.

Nous pouvons utiliser cette diffusion-réception de messages pour synchroniser nos lutins.

Dans le projet JeuDés0 nous avons deux lutins : un personnage et un dé



Le script du personnage Ruby est :



Le dé est muni de 6 costumes représentant chacun une face du dé.

Le script du dé est :

Lorsqu'on démarre le programme, Ruby dit « Lançons le dé » et **envoie le message** « Lancer ».

Le dé qui **attendait ce message**, change alors de costume 50 fois.

Le numéro du costume est tiré au sort entre 1 et 6.

Le script du dé, effectue une **boucle**.

L'instruction incluse dans la boucle est exécutée 50 fois.

Tester ce projet pour bien comprendre comment il fonctionne.

Les boucles

Les boucles permettent d'exécuter plusieurs fois une ou plusieurs instructions. Il existe plusieurs variantes de boucles.

Les trois familles principales permettent de :

- répéter un bloc d'instructions un nombre de fois donné
- répéter un bloc d'instructions **jusqu'à ce** qu'une condition soit vérifiée
- répéter un bloc d'instructions **tant qu'**une condition est vérifiée.

Dans chaque langage de programmation certaines variantes sont implémentées et pas d'autres.

Boucle Pour

Pour répéter un bloc d'instructions un nombre de fois donné :

Instruction avant boucle

Pour i **de** n1 **à** n2 **faire**

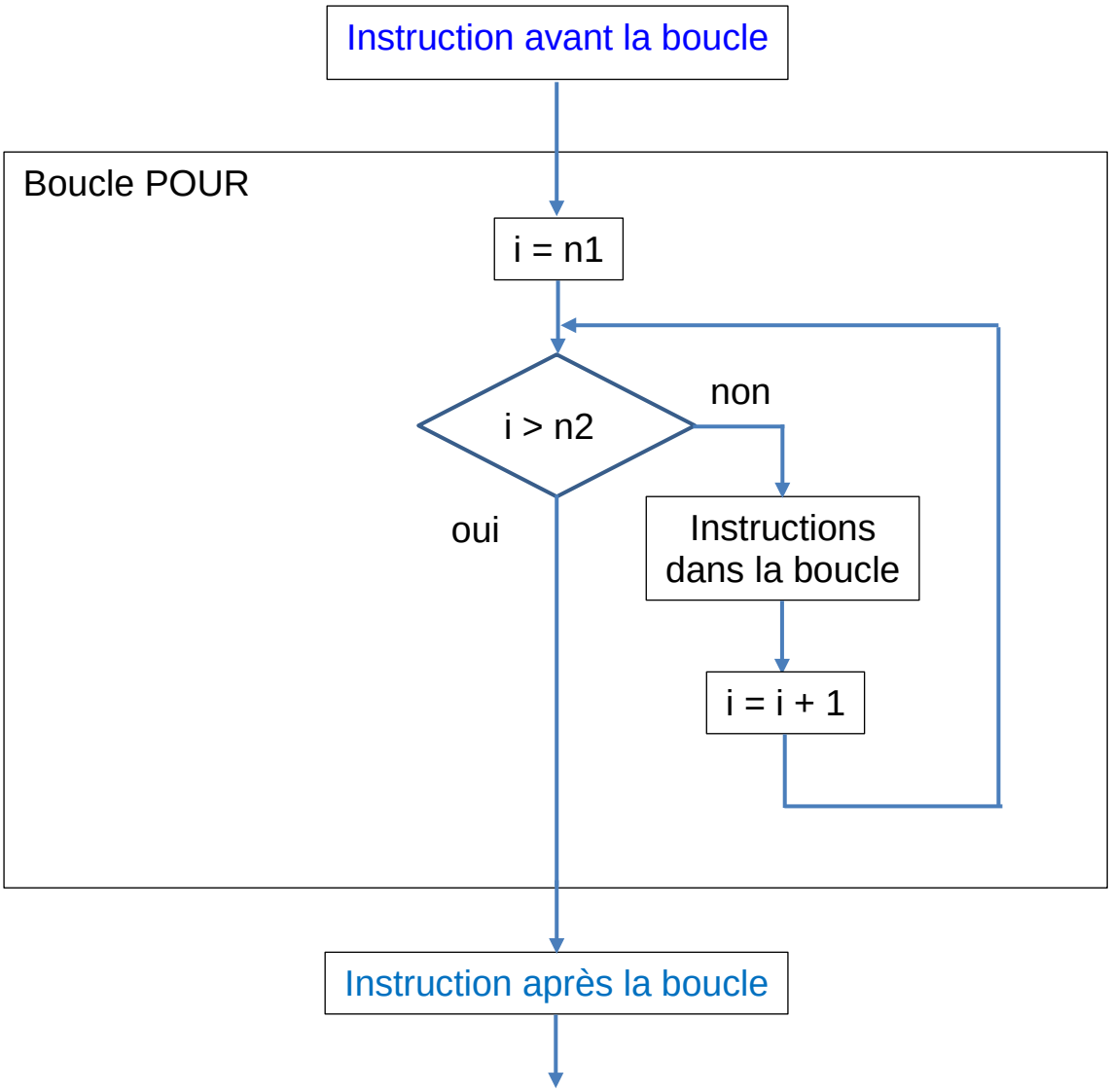
    Instructions dans la boucle

Fin pour

Instruction après boucle

– La variable i est un compteur dont la valeur va varier de n1 à n2

– Elle augmente automatiquement de 1 à chaque tour


Instruction avant la boucle
Boucle POUR
i = n1
i > n2
non
oui
Instructions
dans la boucle
i = i + 1
Instruction après la boucle

Exécution :

a. Lorsqu'on entre pour la première fois dans la boucle, le compteur i prend la valeur n1.
b. La valeur du compteur i est comparée à la valeur n2.
   - Si i > n2 alors **on n'entre pas** dans la boucle et **on saute en f**, à la première instruction située sous la boucle.
   - Si i ≤ n2, alors **on continue** dans la boucle, en c.
c. Les instructions dans la boucle sont exécutées.
d. Lorsqu'on arrive à la fin de la boucle, **le compteur i est augmenté de 1**.
e. On revient en b au début de la boucle.
f. On exécute l'instruction située sous la boucle.

Il existe une autre variante de la boucle « Pour » dans laquelle il est possible de faire varier le pas de progression du compteur.

Pour i **de** n1 **à** n2 **pas** n3 **faire**

    Instructions à effectuer

Fin pour

Le compteur i est augmenté de n3 à chaque tour.

Dans Algobox nous avons la structure :

POUR la variable … **ALLANT DE** … **A** …

Dans Scratch pour n'avons pas la possibilité d'indiquer le nom d'une variable, ni les bornes entre lesquelles la valeur de cette variable varie. Nous ne pouvons qu'indiquer le nombre de répétitions.

Ce nombre de répétitions est égal à n2 – n1

Boucle Répéter jusqu'à

Pour répéter un bloc d'instructions jusqu'à ce qu'une condition soit vérifiée :

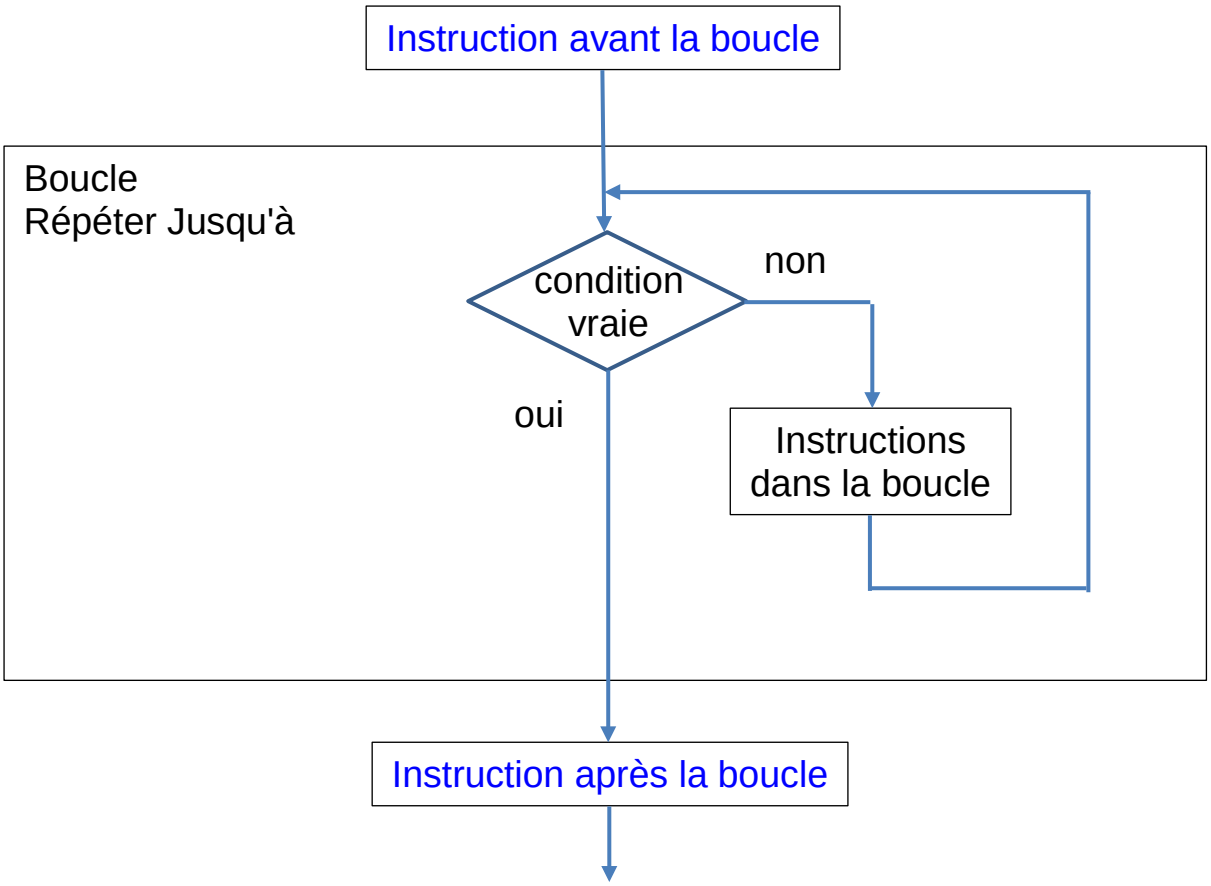
Instruction avant boucle

Répéter Jusqu'à condition vraie

    Instructions à effectuer dans la boucle

Fin répéter

Instruction après boucle



Exécution

a. On exécute l'instruction avant la boucle

b. On teste la condition à l'entrée de la boucle

- Si la condition est **fausse** on **entre** dans la boucle en c.
- Si la condition est **vraie** on **n'entre pas** dans la boucle et on saute en e.

c. On exécute les instructions de la boucle.

d. Lorsqu'on arrive à la fin de ces instructions on remonte en b.

e. On exécute l'instruction située en dessous de la boucle.

Algobox ne dispose pas de ce type de structure.

En Scratch nous avons par exemple:

Si au moment où on arrive sur la boucle n1 ≥ 6, alors on entre dans la boucle. Pour ne pas tourner indéfiniment dans la boucle, il faudra donc qu'au moins une des instructions situées dans la boucle, modifie la valeur de n1.

Si au moment où on arrive sur la boucle, n1 < 6 alors on n'entre pas dans la boucle.

Boucle Tant que

Pour répéter un bloc d'instructions tant qu'une condition est vérifiée :

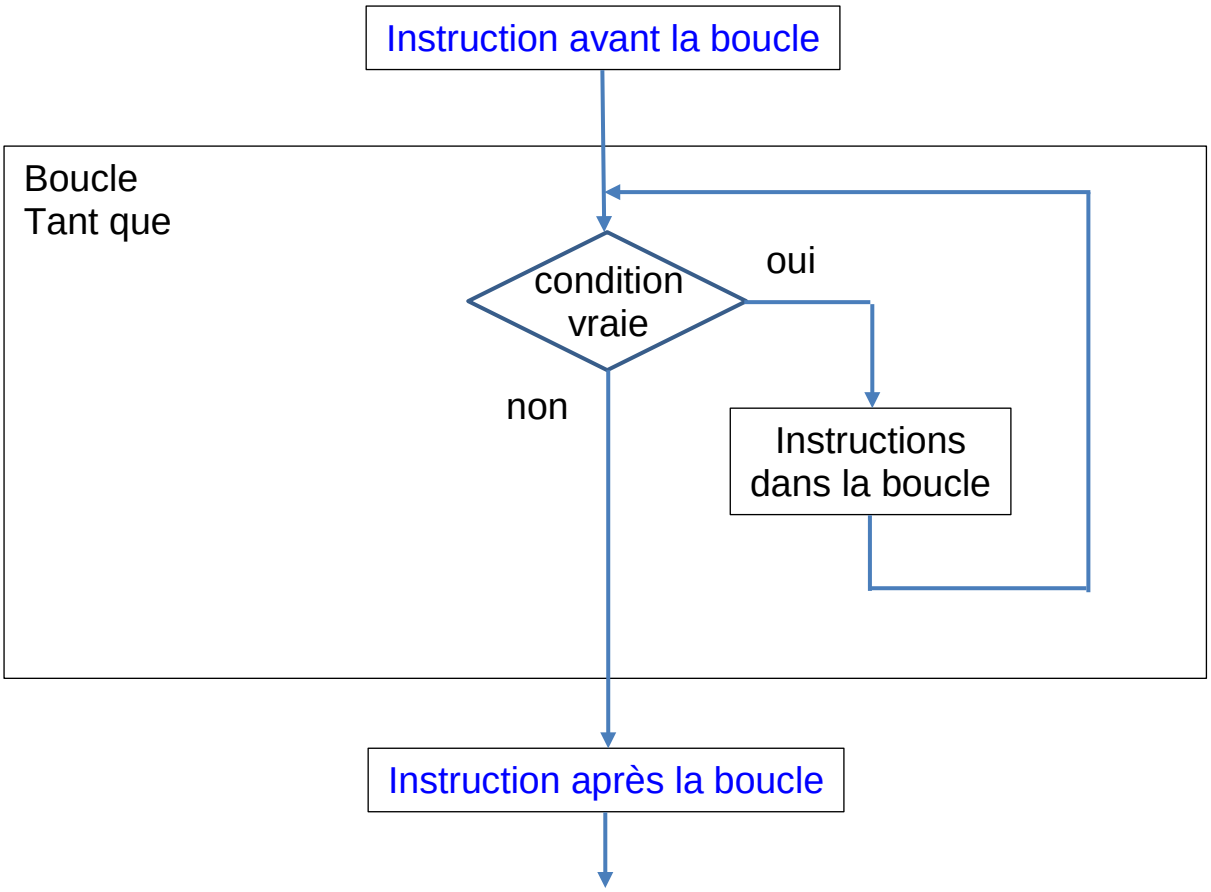
Instruction avant boucle

Tant que condition vraie faire

    Instructions à effectuer dans la boucle

Fin tant que

Instruction après boucle



Exécution :

a. On exécute l'instruction avant la boucle

b. On teste la condition

- Si la condition est **vraie** on **entre** dans la boucle en c.
- Si la condition est **fausse** on **n'entre pas** dans la boucle et on saute en e.

c. On exécute les instructions de la boucle.

d. Lorsqu'on arrive à la fin de ces instructions on remonte en b.

e. On exécute l'instruction située en dessous de la boucle.

Scratch ne dispose pas de ce type de structure.

Avec Algobox nous avons :

TANT QUE condition …. est vérifiée **Faire**

DEBUT_TANT_QUE

FIN_TANT_QUE

Par exemple :

TANT QUE n1<6 **FAIRE**

DEBUT_TANT_QUE

…

FIN_TANT_QUE

Si au moment où on arrive sur la boucle n1 ≥ 6, alors on n'entre pas dans la boucle.

Si au moment où on arrive sur la boucle, n1 < 6 alors on entre dans la boucle. Pour ne pas tourner indéfiniment dans la boucle, il faudra donc qu'au moins une des instructions situées dans la boucle, modifie la valeur de n1.

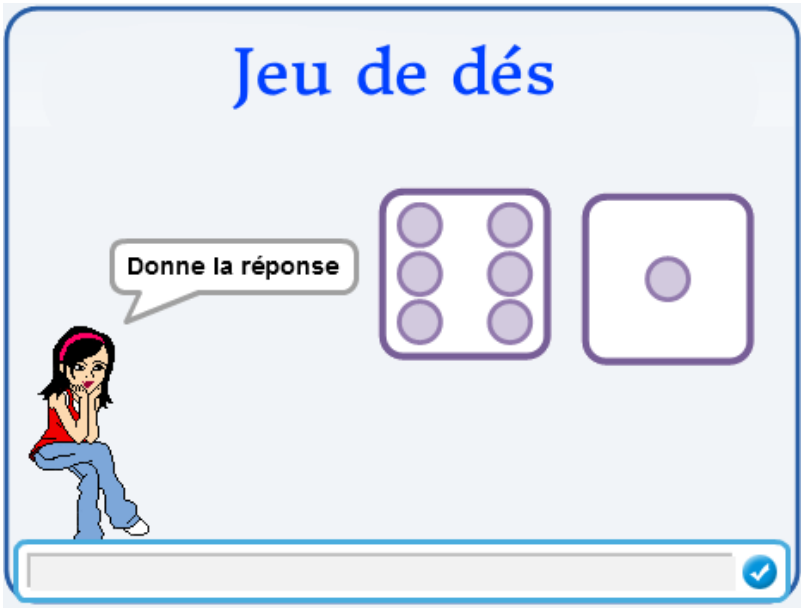
Boucles Répéter Jusqu'à ⇔ Tant que

Nous pouvons remarquer, qu'il est très simple de passer d'une structure à l'autre. Il suffit d'inverser le sens de la condition.

Exercices

Exercice 1 : Reprendre l'algorithme du jeu « Trouver le nombre » et permettre au joueur de rejouer plusieurs fois, jusqu'à ce qu'il trouve le nombre. À chaque proposition du joueur, l'algorithme répond « plus petit » ou « plus grand » tant que le joueur n'a pas trouvé le nombre. Lorsqu'il a trouvé, l'algorithme affiche « Bravo » et indique le nombre d'essais.

Exercice 2 : En partant du projet Scratch JeuDe0, modifier ce projet pour qu'il fonctionne de la façon suivante :



Ruby lance les dés et attend que les dés s'arrêtent.

Elle dit ensuite « Donne la réponse » et attend que le joueur donne sa réponse.

Le joueur doit donner la valeur de la somme des 2 dés.

Si la réponse fournie est exacte, Ruby doit dire « Bravo ! », sinon elle dira « Non, tu t'es trompé ! »

Exercice 3 : Le joueur doit maintenant répondre 5 fois, les deux dés étant relancés à chaque fois. À chaque bonne réponse, le joueur gagne un point, s'il se trompe, il perd un point. Son score est affiché, ainsi que le nombre d'essais. Lorsque les 5 essais sont passés, Ruby dit « C'est terminé »

Exercice 4 : Implémenter l'algorithme PayerLaTaxe (exercice 1 de la mise en train) en Scratch. Voir la fiche correspondant à ce projet.

Il s'agit ici d'apprendre à réutiliser ce que nous savons déjà faire.

Un programmeur doit :

- connaitre les techniques de base utiles pour tous les langages de programmation.
- être capable de réutiliser ce qu'il a déjà fait, en le réaménageant si nécessaire.
- savoir chercher, comprendre et réutiliser ce que d'autres programmeurs ont fait.