Initiation à la programmation orientée objet avec Alice

version du 26/09/2010

David Roche

Notes de l'auteur :

Ce document est grandement inspiré du livre de Wanda P. Dann, Stephen Cooper et Randy Pausch *Learning to program with Alice* (Pearson, Practice Hall), mais il n'en constitue pas une traduction.

Alice est gratuitement téléchargeable sur www.alice.org

Ce document a été utilisé par trois enseignants du lycée G Fichet de Bonneville (Nicolas Bechet, David Berthier et moi même) pour initier des élèves (3 groupes de 20 élèves) de seconde, à la programmation orientée objet pendant l'année scolaire 2009-2010. Un grand merci à Nicolas pour le temps consacré à améliorer ce document.

Un cours destiné aux élèves de première faisant le lien entre Alice et la programmation en Java est en cours de préparation.

David Roche enseignant au lycée G Fichet à Bonneville (74)

Préface

Alice ?

Alice est un logiciel ***" open source "*** qui a été réalisé à l'Université de Carnegie Mellon (Pittsburgh – Pennsylvanie – USA), dans le cadre d'une démarche pédagogique visant à motiver des élèves de collèges ou de lycée à s'initier à la programmation.

Le principe d'Alice est de permettre la construction d'une scène 3D incluant des décors, des personnages (humains ou humanoïdes), des animaux, des objets, du texte, du son ou d'autres éléments issus de bibliothèques fournies avec le logiciel et pouvant être enrichies à partir d'Internet ou bien avec des créations élaborées avec des logiciels spécialisés tels que 3D Studio par exemple. Outre ces éléments, l'utilisateur d'Alice dispose aussi de la possibilité de placer et de faire évoluer une caméra et de régler des questions d'éclairage, de cadrage, etc.

A tous les éléments sont associés des " méthodes ", c'est à dire des commandes ou des actions agissant sur tout ou partie de l'élément : agrandir un objet, le tourner, déplacer une caméra, plier un bras, faire marcher un personnage, etc. Chaque méthode peut disposer de paramètres (par exemple : distance à parcourir, angle de rotation…). De nouvelles méthodes plus ou moins complexes peuvent être créées ou modifiées. Certaines actions peuvent être contrôlées à partir du clavier ou de la souris qui disposent donc de leurs propres méthodes (appuyer sur une touche, clic droit, etc).

La programmation d'une séquence 3D animée commence donc en général par l'écriture d'un projet (scénario, storyboard ou autre forme de cahier des charges), puis par la sélection et la mise en place des différents éléments sur la scène, et enfin la programmation de ces différents éléments par glisser-déposer puis paramétrage des méthodes. La vérification du bon déroulé de la programmation s'effectue en " jouant " la séquence à l'écran. Il est ensuite possible d'exporter cette séquence sous forme d'images, de vidéo, ou encore de fichier partageable par d'autres utilisateurs d'Alice.

Quelques avantages d'Alice :

- Légèreté de l'application qui ne nécessite pas d'ordinateurs particulièrement puissants ou évolués sur le plan graphique. Il suffit simplement de disposer d'une plate-forme Java (gratuite) installée sur son ordinateur, et d'environ 400Mo d'espace disque pour Alice 2.2
- Gratuité du logiciel
- Communauté importante d'utilisateur (enseignants, enfants, passionnés…) issus des plus de 1.500 établissements scolaires utilisateurs du logiciel dans le monde. Cette communauté est essentiellement anglophone, mais nous allons nous employer à développer une communauté francophone dans le même esprit d'échange de pratiques et d'échange d'expériences.
- Accessibilité de l'interface et du logiciel pour les novices
- Exigence du concept, en pleine conformité avec les exigences de la programmation objet.

Apprendre la programmation

L'apprentissage de la programmation est un domaine pédagogique particulièrement intéressant permettant de donner aux élèves l'expérience de systèmes complexes en développant un mode de pensée structuré, méthodique et rigoureux qu'ils pourront ensuite mettre en œuvre dans de nombreuses autres matières de la Biologie à l'Économie, mais aussi dans la vie pratique, que ce soit au contact des appareils technologiques de la vie courante, ou bien de systèmes complexes dont le fonctionnement dépend de celui de ses éléments constitutifs et de leurs interactions (automobile, météo, organisation de la production dans une usine, plan de développement économique, élaboration d'une politique environnementale,…).

La pratique de la programmation permet de se trouver confronté à des problèmes plus ou moins complexes nécessitant à la fois une démarche logique basée sur l'analyse, une approche par hypothèses, essais successifs et élimination, une capacité imaginative, un esprit d'initiative propice à l'auto-apprentissage et enfin le développement de qualités plus génériques telles que la patience, l'ouverture d'esprit, l'adaptabilité, la précision.

Enfin la programmation s'appuie sur des langages qui ont leur vocabulaire et leur syntaxe, elle renforce donc la formation de l'esprit aux pratiques linguistiques. D'une façon plus directe, les élèves, sans forcément se trouver une vocation pour les métiers de l'informatique, seront confrontés tout au long de leur vie à l'environnement informatique et nombre d'entre eux seront appelés à collaborer de façon étroite avec des professionnels de l'informatique ou encore de mettre directement en application leurs savoir-faire en programmation, comme par exemple dans le cadre de la mise en place de formules sous tableur Excel, de la manipulation d'informations issues de bases de données marketing ou commerciales, ou de la préparation d'un e-mailing.

Mais l'apprentissage de la programmation est aussi potentiellement porteur de frustrations ! Les tâtonnements pour obtenir un résultat satisfaisant peuvent être longs et stériles, la patience trop malmenée, les problèmes jugés trop complexes, leurs résolutions trop obscures. D'où la conception du logiciel Alice, un outil de programmation dans un environnement 3D permettant de manipuler des objets et des personnages afin de réaliser des séquences animées, voire des jeux interactifs.

Alice a été pensé afin de lever deux des principaux obstacles rencontrés par les programmeurs débutants :

- Les erreurs de syntaxe
- La non visibilité du résultat

Pour cela, la programmation s'effectue par glisser-déposer de blocs de code incluant les concepts classiques abordés dans ce domaine : itérations et boucles, branchements conditionnels, méthodes, paramètres, variables, tableaux et récursivité. De plus, l'apprenant peut à tout moment observer le résultat, ajuster et corriger ses erreurs, en visualisant l'animation 3D en cours de réalisation.

Alice ouvre l'esprit

Heureusement, Alice n'est pas qu'un simple outil d'apprentissage à la programmation. C'est aussi un outil qui permet aux élèves d'**apprendre à apprendre** : le travail par essais successifs, qui est une base de la programmation, s'applique aussi à bien d'autres activités, c'est une démarche qui permet aussi d'apprendre à mesure les risques et prendre des précautions en conséquence (comme par exemple effectuer une copie de sécurité de son travail avant d'y effectuer des changements importants, ou bien encore décomposer un travail important en étapes progressives élémentaires faciles à contrôler).

Alice permet aussi de mettre en œuvre de nouvelles méthodes pédagogiques particulièrement intéressantes à l'ère de l'informatique et de l'Internet : celles notamment reposant sur la progression de son apprentissage à partir de l'acquisition d'un savoir-faire pré-existant. Ainsi un élève pourra exercer son acquis et l'enrichir en travaillant sur une séquence déjà réalisée par un tiers (qu'on lui aura fournie ou qu'il aura trouvé sur un espace communautaire dédié à Alice sur Internet) avec pour objectif de modifier celle-ci pour l'enrichir ou l'adapter à un nouveau besoin. C'est une méthode de travail largement répandue dans le monde de la programmation qui permet non seulement d'acquérir rapidement de nouvelles connaissances, mais aussi de confronter et de faire évoluer les siennes, de découvrir de nouvelles façons de programmer, etc.

Cette forme de travail permet aussi de développer chez les jeunes un esprit " social " qui les incitera à s'insérer dans des communautés partageant les mêmes centres d'intérêt pour y mettre à disposition leur compétence et certaines de leurs réalisations dans un esprit d'ouverture et de service.

Daniel Bouillot, Délégué à l'action économique, la formation et la recherche – CITIA -

Sommaire

Chapitre I : Les premiers pas p 8

Exercices chapitre I p 12

Chapitre II : Premier programme p 14

Exercices chapitre II p 25

Chapitre III : Les fonctions et les structures de contrôle p 27

Exercices chapitre III p 36

Chapitre IV (1er partie) : Les classes, les instances et les méthodes p 38

Exercices Chapitre IV (1er partie) p50

Chapitre IV (2e partie) : Les méthodes Class-Level et l'héritage p 52

Exercices chapitre IV (2e partie) p 62

Chapitre V : Interaction utilisateur – machine p 64

Exercices chapitre V p 69

Chapitre VI : Retour sur les fonctions et le couple if/else p 71

Exercices chapitre VI p 79

Chapitre VII : L'instruction while p 81

Exercices chapitre VII p 87

Chapitre VIII : Les variables p 89

Exercices chapitre VIII p 99

Annexe 1 : Les coordonnées dans Alice p 101

Annexe 2 : Les méthodes move, turn, roll.... p 104

Annexe 3 : La caméra p 108

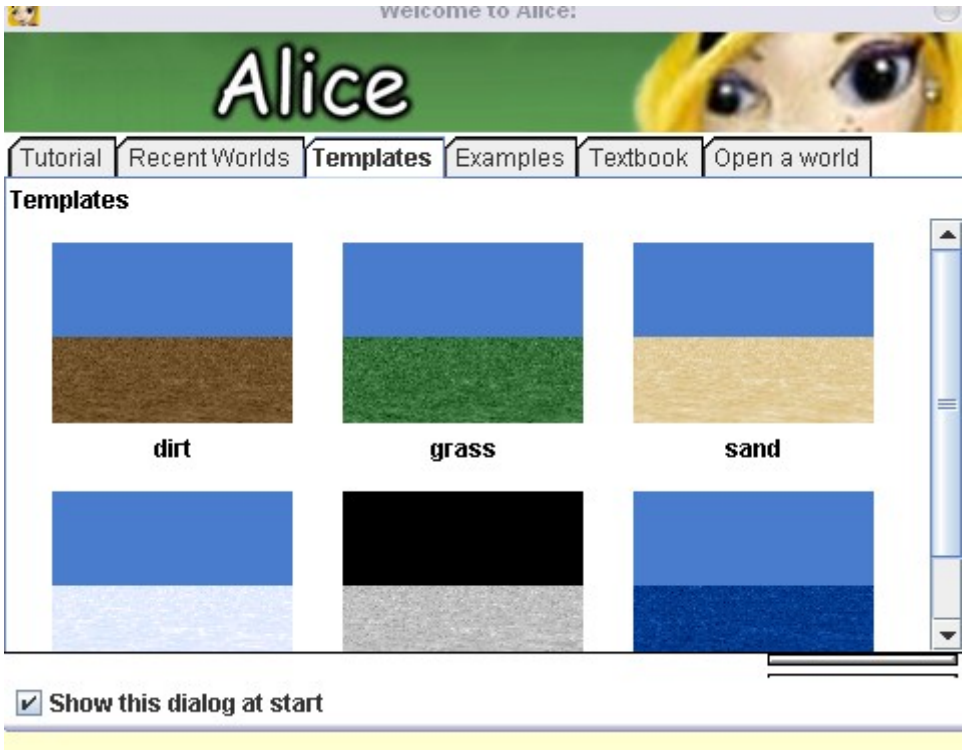
Annexe 4 : rendre les objets invisibles p 111

Annexe 5 : Créer ses propres personnages p 112

Annexe 6 : Nombres aléatoires p 114

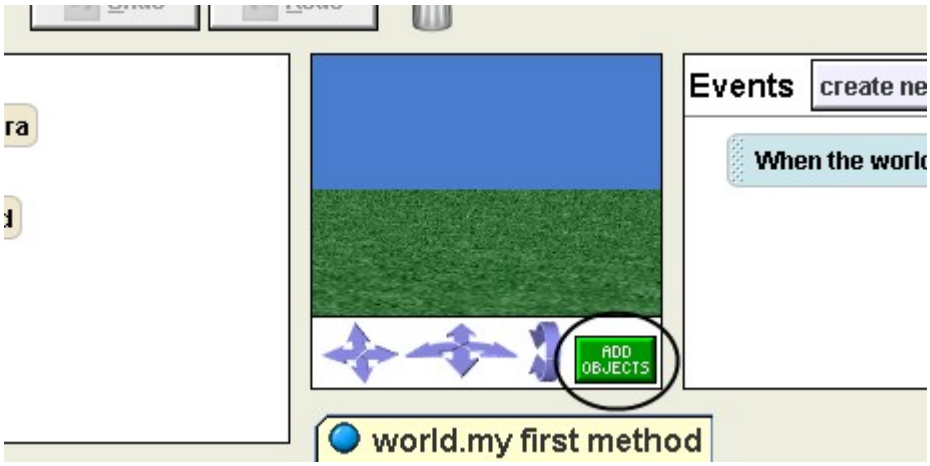
Chapitre I
Les premiers pas

Alice 2.2 va vous permettre de créer un monde virtuel rempli de personnages de toutes sortes. À l'ouverture du logiciel, vous devez choisir un monde vide (ou un monde déjà partiellement peuplé)

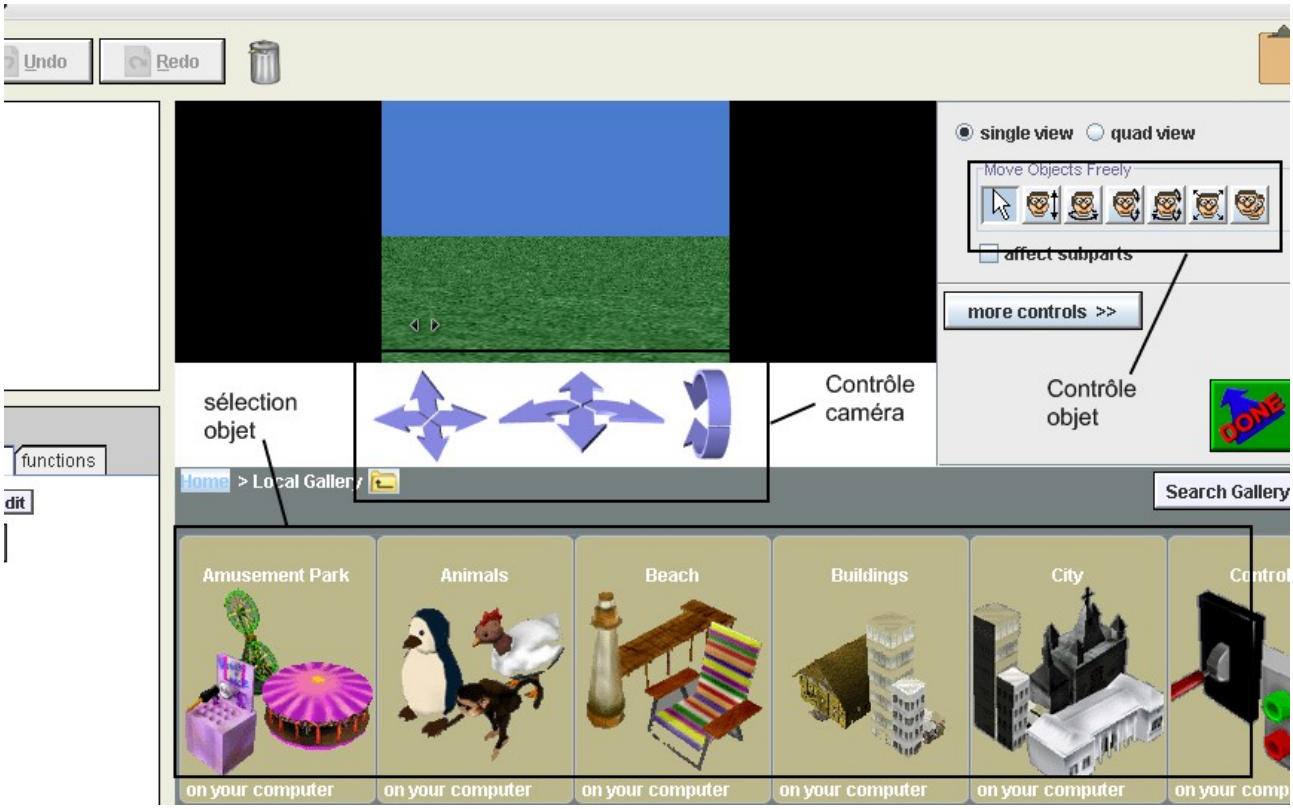


" Templates " permet d'ouvrir un monde vide, " Textbook " permet d'ouvrir un monde partiellement peuplé

Une fois " le monde " vide choisi, il faut le peupler avec des personnages ou des objets inanimés (dans les 2 cas, nous avons affaire à des instances de classe au sens de la POO, chaque objet ayant des méthodes, des fonctions et des attributs (variables)).
Le bouton " ADD OBJECTS " permet d'ajouter des objets (personnages ou objets inanimés).



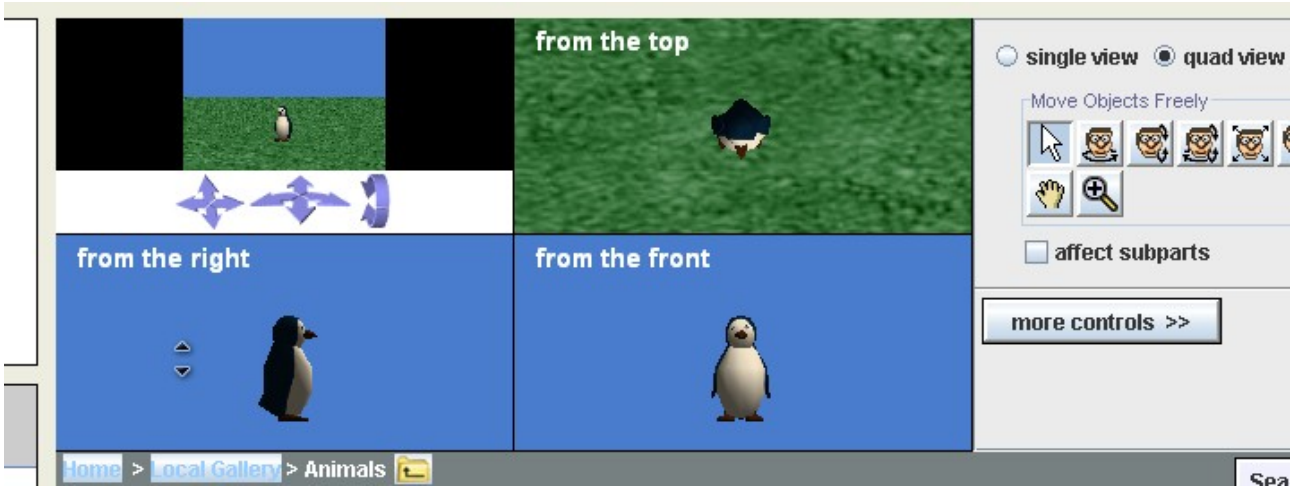
" ADD OBJECTS " nous donne accès à l'éditeur de scène.



La première chose à faire est de choisir un objet parmi la collection proposée (il existe aussi d'autres objets sur le net). L'objet est placé dans le monde par cliquer-déplacer-déposer.

Vous pouvez modifier la taille et la position de vos objets (voir schéma ci-dessus " contrôle objet "). En 3D, vous pouvez déplacer vos objets (*translation*), vers le haut, vers le bas, vers la gauche, vers la droite, vers l'avant et vers l'arrière. Vous pouvez aussi faire tourner vos objets (*rotation autour de l'axe des X, l'axe des Y et l'axe des Z*). Bien sûr tout cela est à tester.

Pour vous repérer dans cet univers 3D (ce qui est loin d'être évident !), vous pouvez utiliser, dans l'éditeur de scène, la vue " quad view "



Une fois votre scène mise en place cliquez sur " DONE "

Pour chaque objet, leur créateur a défini un centre, ce centre correspond souvent au centre géométrique de l'objet (*plutôt centre de masse*), mais pas forcément : pour les personnages, le centre de l'objet se trouve entre les 2 pieds, pour la batte de baseball, le centre se trouve au niveau du manche....bref, à voir au cas par cas.

Le centre de l'objet est très important, car les coordonnées d'un objet sont données par rapport au centre de l'objet.

Exercices chapitre I

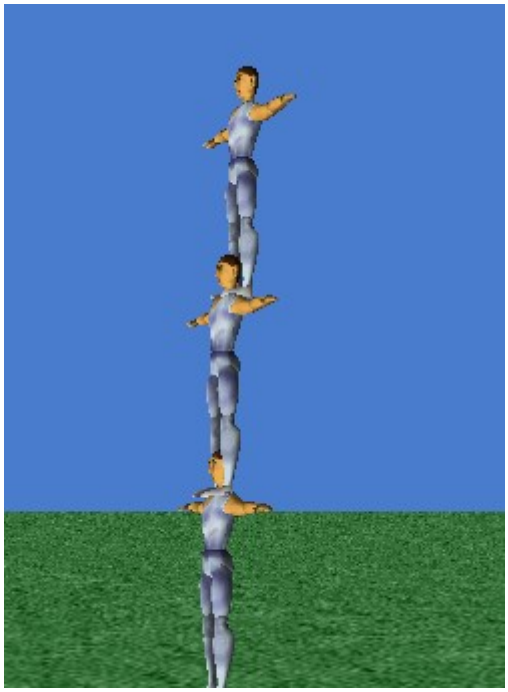
Voici quelques exercices pour vous entrainer à maitriser la mise en place d'une scène dans Alice 2.2
Dans ces exercices il est indispensable d'utiliser la vue " quad view "

Exercice 1.1

Dans un paysage enneigé, placer 2 rangées de 3 pingouins serrés les uns contre les autres (une rookerie).

Exercice 1.2

Reproduire la scène suivante (personnage : MaleBalletDancer).



Exercice 1.3 (voir l'annexe 1 pour cet exercice)

Placer une poule dans un monde vide de telle façon à ce qu'elle ait pour coordonnées (2,0,1) (à peu près !).
On veut placer un chat 2 m derrière la poule (il est en train de la poursuivre !). Quelles doivent être ses coordonnées ? Placer correctement le chat.
Placer une tortue qui regarde la scène (à la gauche de la poule et du chat).
Enfin, placer un oiseau au milieu du triangle formé par la poule, le chat et la tortue à 1,5 m au-dessus du sol

Exercice 1.4

Créer une scène avec un aéroport, une tour de contrôle, 1 avion au sol et 1 avion en train de décoller.

Chapitre II
Premier programme

.

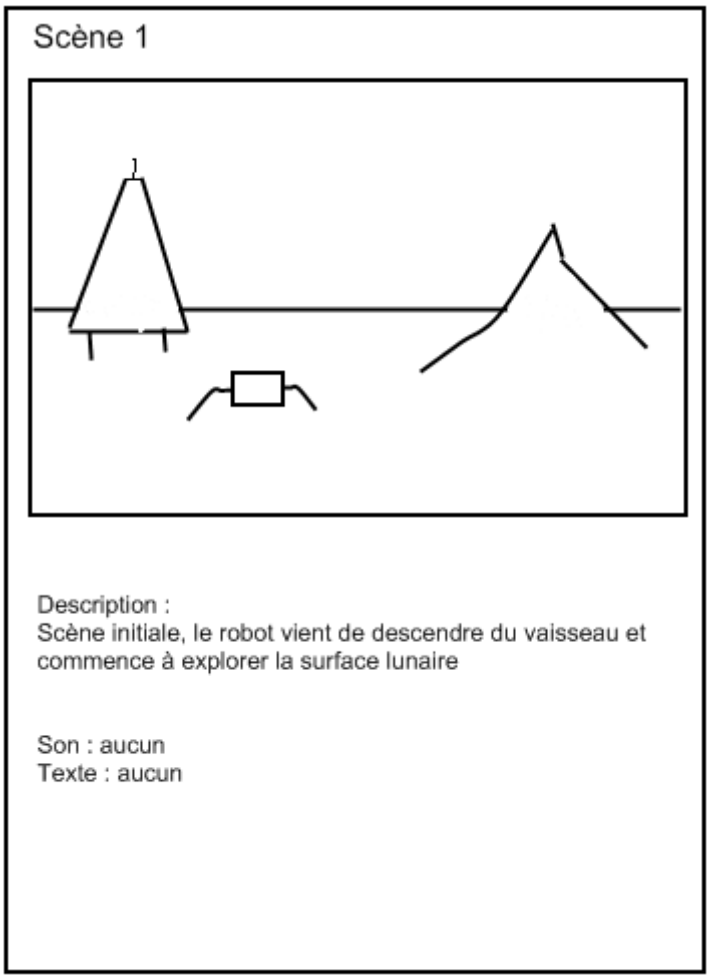
Placer les personnages dans une scène, c'est bien, animer une scène, c'est mieux ! Nous allons donc, dans ce chapitre, écrire notre premier programme.

Scénario et story-board

Avant d'aborder l'écriture d'un programme dans Alice, il va falloir nous familiariser avec l'écriture des scénarios et des story-boards. Nous allons commencer par un petit exemple relativement simple, mais au contenu très riche :

Scénario : La scène se déroule sur la Lune, un robot (“terrien”), vient juste de descendre de son vaisseau spatial, quand il est interpelé par un extra terrestre qui jusqu'à présent était caché derrière un rocher. Surpris, le robot tourne la tête vers l'intrus et se précipite vers lui, mais trop tard, l'extra terrestre a disparu !
Le robot tourne alors la tête vers la caméra; en signe de danger cette dernière devient rouge. Le robot dit la phrase restée célèbre (Apollo 13) :
“Houston, nous avons un problème !”

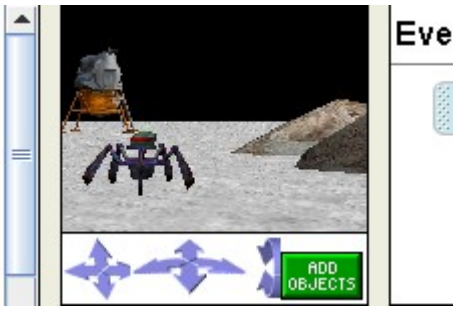
Maintenant que la trame de l'histoire est connue, il nous faut passer au story-board :
Le story-board va nous permettre de découper notre histoire en scène.
Il comportera : le numéro de la scène, un dessin de la scène (non obligatoire), une description de la scène, les éventuels sons et les éventuels textes. Pour notre première scène, cela pourrait donner ceci :



Voici la suite du story-board (sans les dessins) :
Scène 2 : Description : Un extra terrestre surgit derrière un rocher
Son : aucun
Texte : l'extra terrestre dit : “?r#'(t&ç^”
Scène 3 :
Description : le robot tourne la tête (360°) pour chercher l'extra terrestre
Son : aucun
Texte : aucun
Scène 4 :
Description : Le robot se tourne puis se dirige vers l'extra terrestre. Mais ce dernier disparaît derrière le rocher.
Son : aucun
Texte : aucun
Scène 5 :
Description : le robot se tourne vers la caméra, sa tête devient rouge.
Son : aucun
Texte : le robot dit : “Houston, nous avons un problème !”

du story-board à la programmation

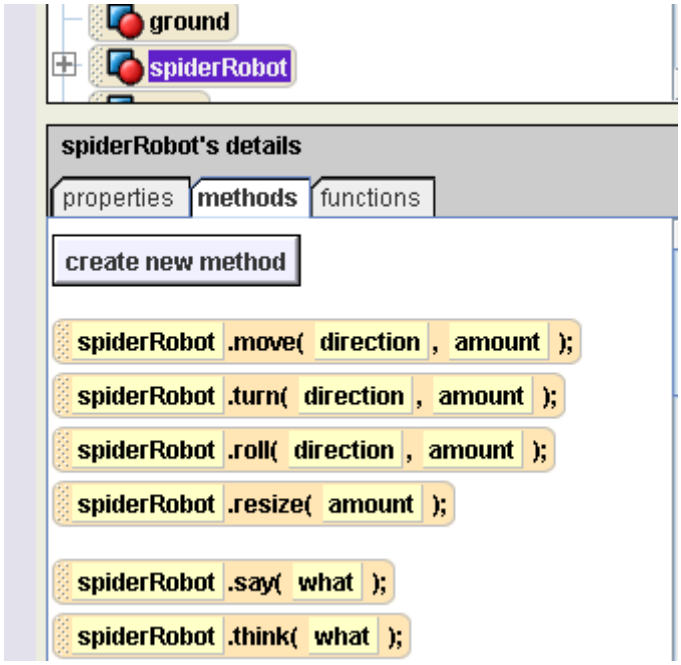
Nous allons utiliser une scène “toute prête” : ouvrir le fichier " chap2-1.a2w "
Vous devriez avoir à l'écran :



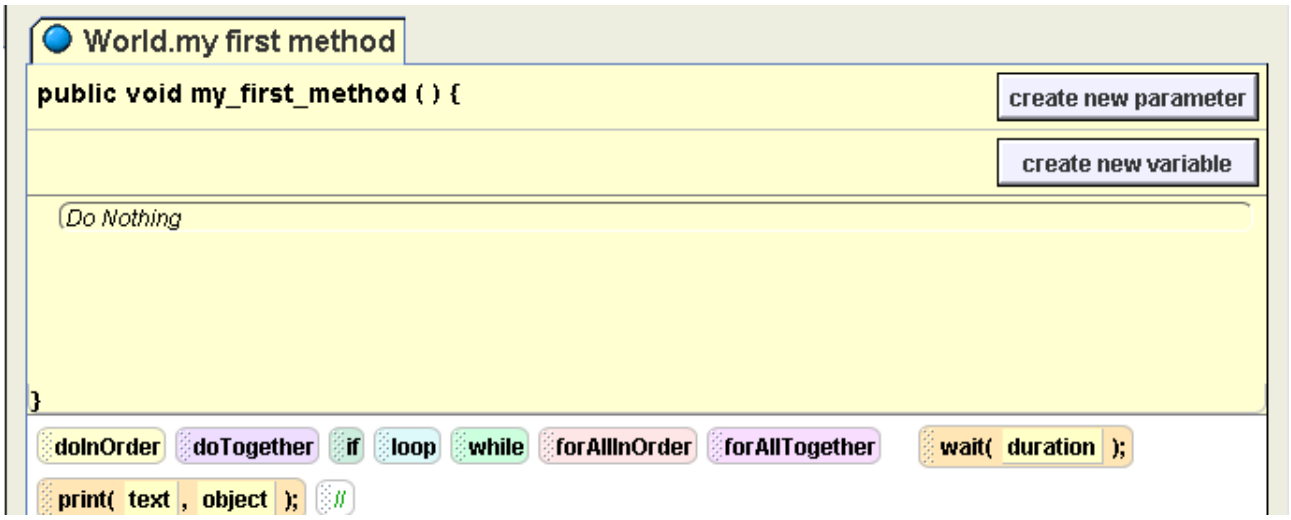
Tous les acteurs sont déjà en place (l'extra terrestre est caché derrière un rocher).

Comme déjà dit auparavant, Alice est à 100 % objet, avec Alice, on ne fait qu'une seule chose : manipuler des objets. Qui dit POO dit classe, instance, méthodes, attributs, fonctions. Alice a déjà créé les instances dont nous aurons besoin (spiderRobot, rock, rock2, alienOnWheels et lunarlander).

Chacune de ces instances possède ses propres méthodes, ses propres attributs et ses propres fonctions.
Voici par exemple, une partie des méthodes de l'instance “spiderRobot” :



Nous allons créer une nouvelle méthode pour la classe « World » grâce à la fenêtre « World.myfirstmethod » (une fois créé, nous aurons une méthode de la classe « World » qui aura pour nom « myfirstmethod ».)



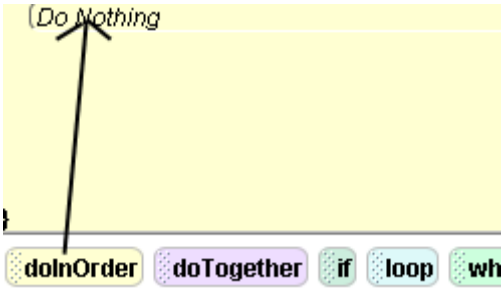
Nous reviendrons sur les détails plus tard, pour l'instant, vous devez juste savoir que nous allons remplacer “Do Nothing” par des instructions (qui pourront être des méthodes ou des fonctions (oui, des méthodes dans une méthode !) mais pas seulement).

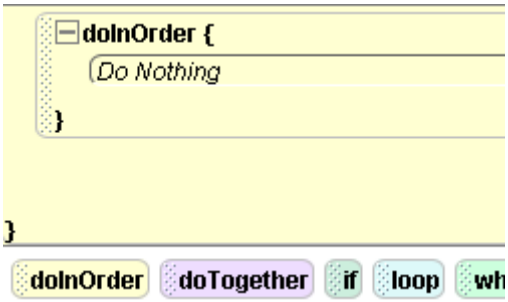
Quand nous lancerons notre programme, la première chose que fera Alice, c'est d'appeler notre méthode « World.myfirstmethod ».

Car comme nous le montre l'image ci-dessus : When the world starts , do World.myfirstmethod ( ); (nous reviendrons aussi plus en détail sur la fenêtre “Events” plus tard.)

Dernière chose avant d'entrer dans le vif du sujet, il existe 2 façons d'exécuter les instructions : l'une après l'autre ou plusieurs en même temps. Pour les exécuter l'une après l'autre il faudra utiliser ”DoInOrder” et “DoTogether” pour les exécuter toutes en même temps.

La première chose à faire et de “déposer” DoInOrder à la place de DoNothing
Ceci étant fait, vous devriez avoir ceci :

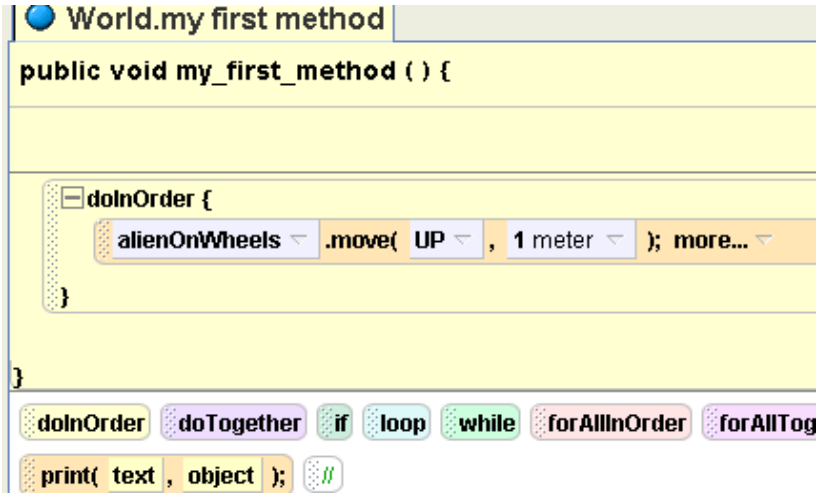




Nous allons maintenant pouvoir déposer nos premières instructions :

La scène commence avec l'apparition de l'extra terrestre (AlienOnWheel), cliquer sur “AlienOnWheel”, puis sur l'onglet “methods”, puis déposer la méthode “AlienOnWhell.move(direction, amount);” dans “doInOrder” (à la place de “Do Nothing”), le programme ouvre alors un menu déroulant, choisir “up”, dans l'autre menu déroulant sélectionner “1 meter”.

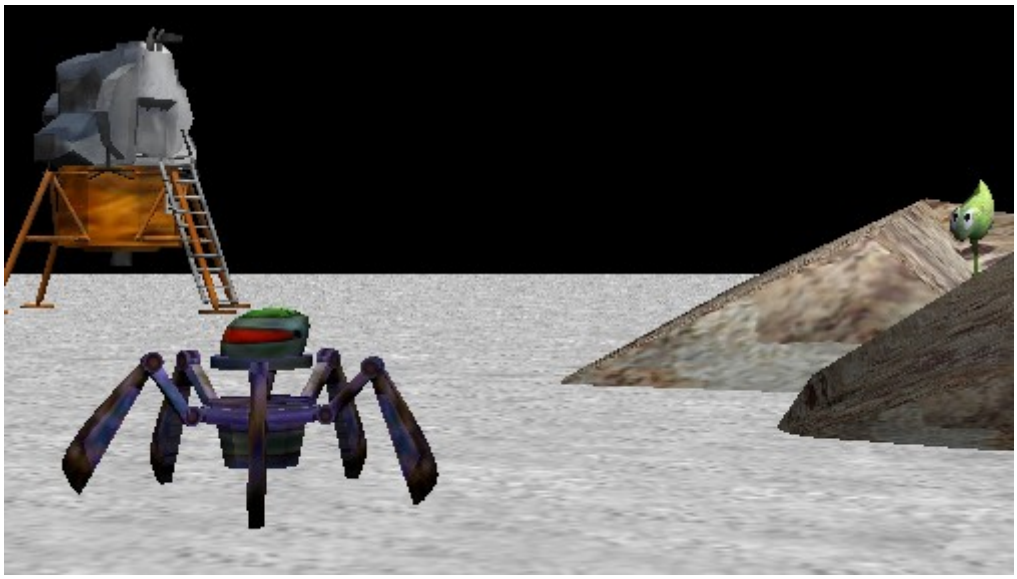
Vous devriez alors avoir ceci :



Nous avons utilisé la méthode move de “alienOnWheels” avec pour argument “UP” (vers le haut) et “1 meter” (d'une distance d'un mètre).

Vous devriez ici prendre conscience de la “puissance” d'Alice, car faire bouger un objet de 1 mètre vers le haut est d'une simplicité enfantine.

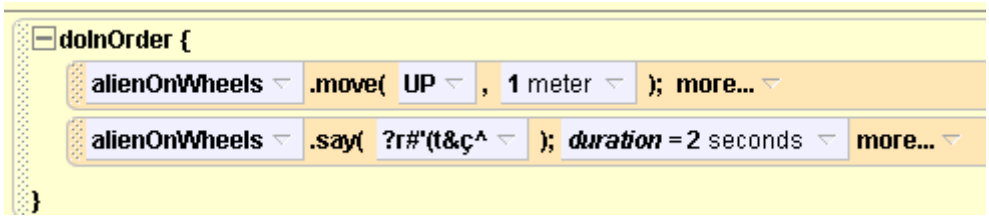
Le programme vous propose automatiquement les arguments liés à la méthode employée, d'autres arguments optionnels sont disponibles (“more...”)) !
Nous pouvons d'ores et déjà tester notre programme en cliquant sur le bouton “Play”.



Tout fonctionne comme prévu, notre extra terrestre sort de sa cachette !

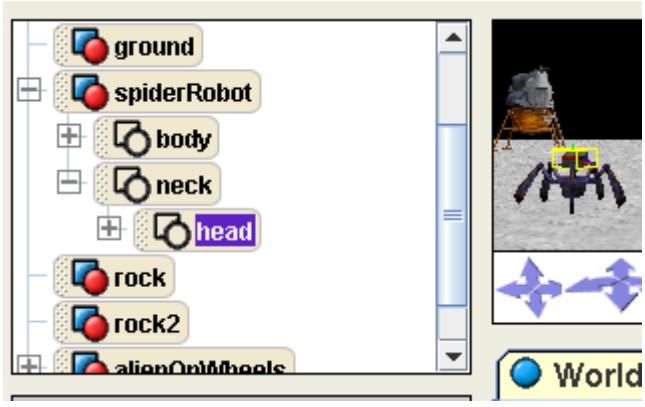
Pour la deuxième instruction, nous utiliserons la méthode “say” (toujours pour alienOnWheels), dans le menu déroulant, sélectionner “other...” et entrer votre texte dans la fenêtre.

Nous pourrions tester l'effet de notre deuxième instruction, mais avant nous allons choisir la durée d'affichage de notre texte en sélectionnant “...more”-> “duration”-> “2 seconds”.

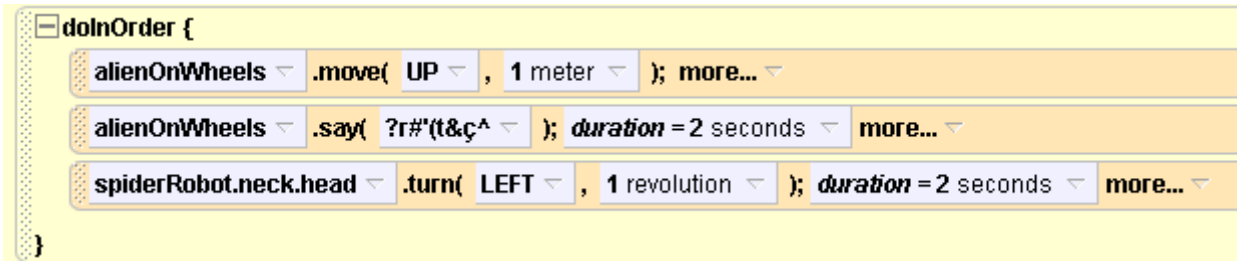


et voilà le résultat :

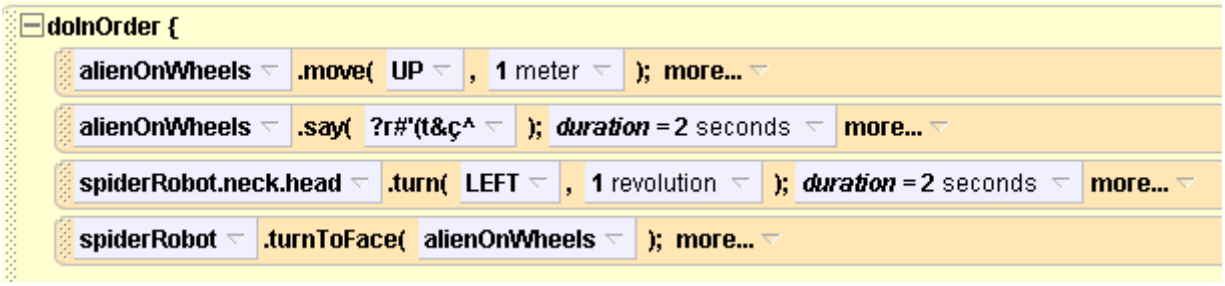
Enchaînons avec la rotation de la tête du robot :
Jusqu'à présent nous avons toujours utilisé les méthodes de l'instance “alienOnWheels”, nous allons maintenant utiliser une méthode de l'instance “spiderRobot”.
Mais si nous utilisons la méthode “turn” pour “spiderRobot”, c'est le robot en entier qui va tourner alors que dans notre scénario, cela doit être uniquement sa tête !
Les utilisateurs d'Alice ont prévu “le coup” et les objets sont souvent divisés en plusieurs parties. Pour voir ces parties, il suffit de cliquer sur le + qui se trouve devant “spiderRobot”.



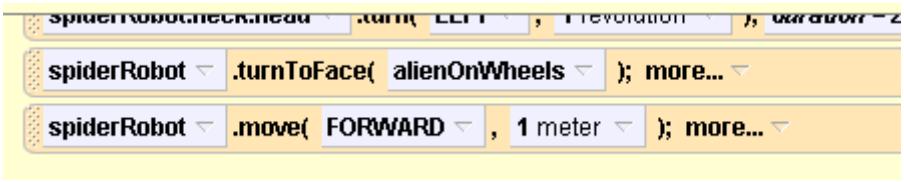
Sélectionner “head” (comme ci-dessus), il suffit maintenant de sélectionner la méthode “turn” dans l'onglet “methods”.Choisir alors “LEFT” puis “1 revolution” puis dans “more...” choisir “duration” et “2 seconds”.



Le robot doit maintenant se tourner vers l'extra terrestre. Pour ce faire, on sélectionne la méthode “turnToFace“ (se tourner vers) de l'instance “spiderRobot”, comme “target” choisir “alienOnWheels”-> “the entire alienOnWheels “ (on aurait pu choisir seulement une partie de alienOnWheels)

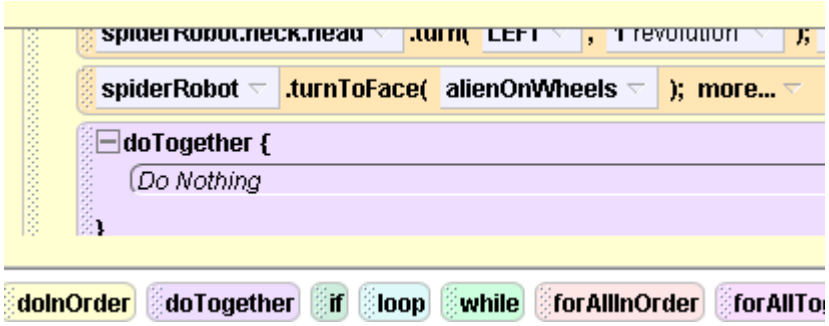


Faisons avancer le robot droit devant lui (“FORWARD”) sur 1 mètre. Pour cela nous allons utiliser la méthode move de spiderRobot.



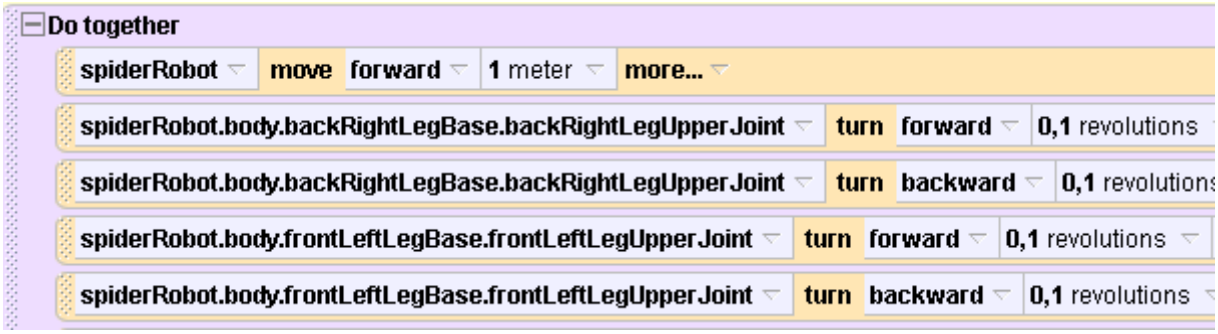
Après avoir testé cette nouvelle instruction, il est évident que le déplacement du robot n'est pas très “naturel” (les pattes ne bougent pas). Essayons d'améliorer cela.

Pour “spiderRobot”, il n'existe pas de méthode “marcher”, nous devons la programmer nous même. Il faut en même temps avancer et bouger les pattes, pour se faire nous allons utiliser “doTogheter” à la place de “doInOrder” (“faire ensemble” à la place de faire “dans l'ordre”).
Effaçons la dernière ligne ajoutée et “déposons” “ doTogheter” à la place.



Nous allons ajouter la méthode “move” comme indiqué précédemment et appliquer la méthode “turn” vers l'avant sur 0,1 révolution puis vers l'arrière toujours sur 0,1 révolution à la jointure entre les 2 parties de la patte arrière droite (la rotule) (ouf !), même chose pour la patte avant gauche (par souci de simplicité nous n'animerons que 2 pattes sur 4).

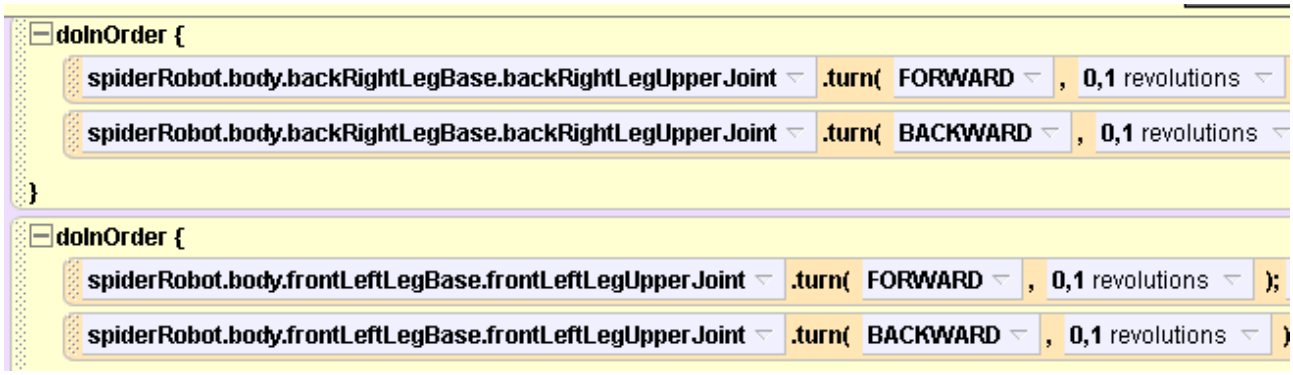
Tout cela est un peu compliqué à expliquer, alors voyons, ce que cela donne à l'écran :



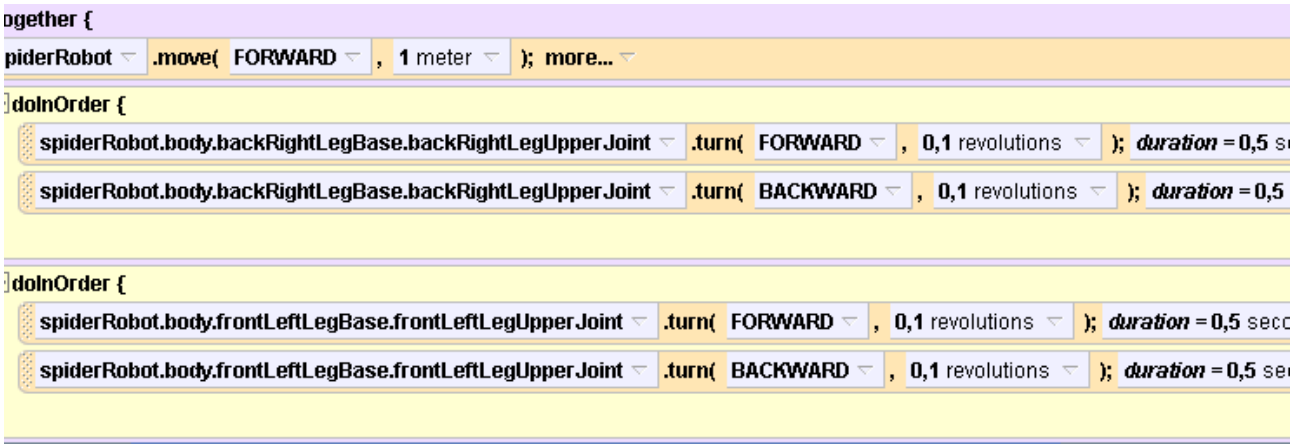
Testons, cela ! Horreur, cela ne fonctionne pas (on dit qu'il y a un bug !) Nous venons tout simplement de faire une erreur de programmation (une erreur de logique !)

Réfléchissons un peu :
finalement, il est tout à fait normal que cela ne fonctionne pas, en effet nous demandons au robot de tourner en même temps la jambe vers l'avant et vers l'arrière, bref de ne rien faire !!
Pour que cela fonctionne il faut, dans un premier temps qu'il tourne vers l'avant puis ensuite vers l'arrière. Ce mot “ensuite” doit immédiatement vous faire penser à “doInOrder”. Cela sera un “doInOrder” dans un “doTogheter” ! Je vous laisse réfléchir à tout cela !

Voilà le code corrigé :



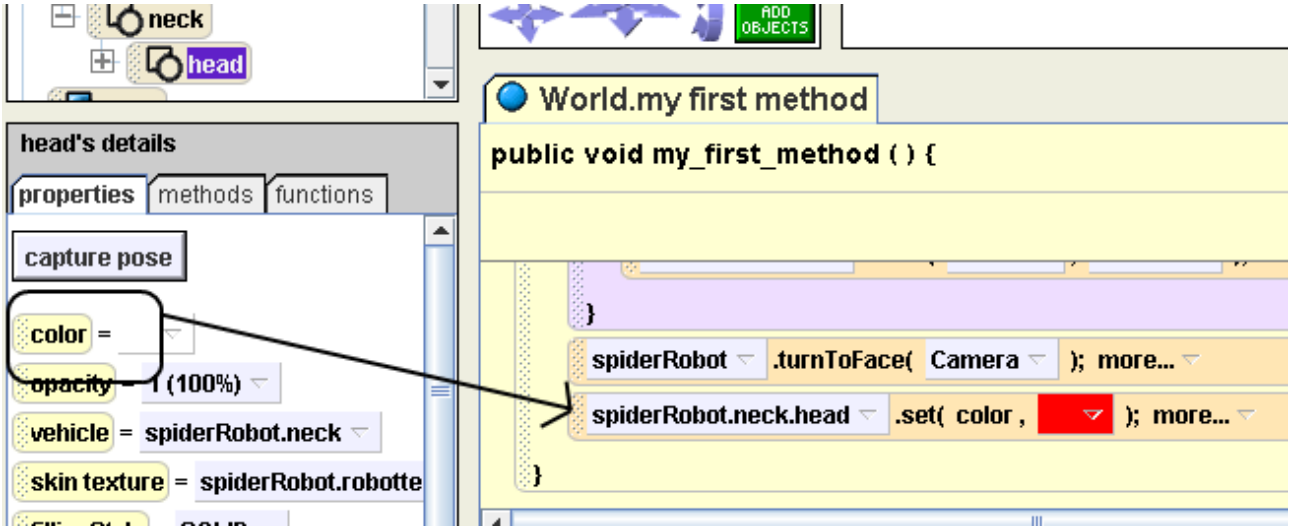
Cela va mieux, mais ce n'est pas encore “génial”.
Il y a un manque de synchronisation entre le mouvement du robot et l'animation de ses pattes.
Il faudrait que le mouvement “avant-arrière” des pattes soit plus rapide. Utilisons “more..” → “duration”-> “0.5 seconds” pour chaque mouvement de la patte (pour qu'en tout le mouvement des pattes dure 1 seconde)



L'extra terrestre doit se cacher (mouvement inverse de celui du départ), on applique donc la méthode “MOVE” “DOWN” “1 meter”de “alienOnWheels”.
Pour que ce mouvement vers le bas se fasse pendant le mouvement du robot, il faut veiller à ce que cette instruction soit dans le “doTogether” du mouvement du robot (voir les différentes couleurs de fond)

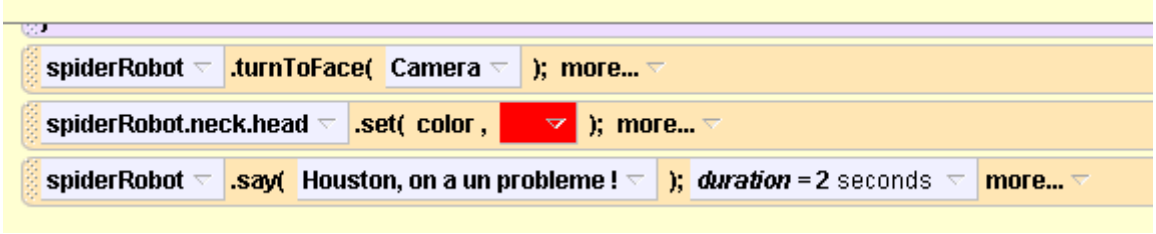
Dans notre story-board, le robot doit maintenant se tourner vers la caméra. On va donc utiliser la méthode “turn to face” de “spiderRobot” avec l'argument “camera”.

La tête du robot doit maintenant devenir rouge. Nous allons modifier un des attributs de la tête du robot (avec Alice pour accéder aux attributs d'une instance, il faut cliquer sur l'onglet “properties”), l'attribut “color”.
Déplacer l'attribut “color” à la suite de nos instructions et choisir “RED”



et la tête devient bien rouge.

Il ne nous reste plus qu'à envoyer le message à Houston (avec “duration = 2 seconds”)
Attention pas d'accent (système US) !



Exercices chapitre II

Exercice 2.1

Robert, Gertrude et Davina sont 3 bonshommes de neige. Robert a toujours eu un petit faible pour Gertrude, mais n'a jamais eu le courage de lui déclarer sa flamme. La scène débute alors que Gertrude et Davina sont en pleine discussion.Robert choisit ce moment pour se faire remarquer par Gertrude (à vous de voir comment !). Gertrude se retourne pour faire face à Robert et reprend sa conversation avec Davina sans le moindre mot pour ce pauvre Robert.
Écrire la fin de ce scénario, écrire un story-board, programmer la scène.

Début de la scène

Exerice 2.2 (Pour cet exercice voir l'annexe 2)

Créer un monde (paysage marin) et déposer dans ce monde une île (Environments/Island) et un poisson. Dans la scène initiale, le poisson sort légèrement de l'eau et se trouve devant l'île.
Programmer la scène suivante :

- le poisson tourne en rond devant l'île (1 tour)
- le poisson tourne autour de l'île (1 tour)
- le poisson saute hors de l'eau avant de disparaître dans les abysses.

scène initiale

Exercice 2.3 (Pour cet exercice voir l'annexe 2)

Créer un monde peuplé : d'une petite fille (People/Mana), d'un aimant (Objects/Magnet), d'un ressort (Objects/Spring), d'un casque de scaphandrier (Objects/divingHelmet), d'une clé (Objects/windupKey) et d'une voiture (Vehicles/car).
Mana tiens l'aimant dans sa main gauche et dirige ce dernier vers le ressort, le ressort est alors attiré vers l'aimant. Mana reproduit la même opération avec le casque de scaphandrier et la clé. Elle dirige alors l'aimant vers la voiture, mais au lieu d'attirer la voiture, c'est la voiture qui attire Mana. Mana abandonne alors l'aimant, qui reste collé à la voiture, et revient en marchant vers la caméra.
Programmer cette scène.

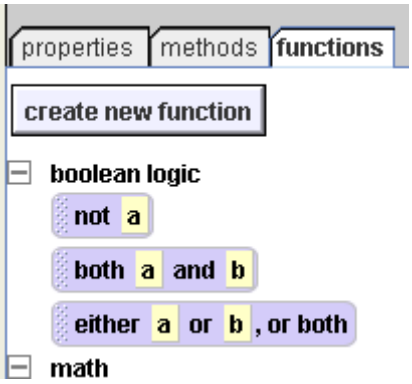
scène initiale

chapitre III
Les fonctions et les structures de contrôle

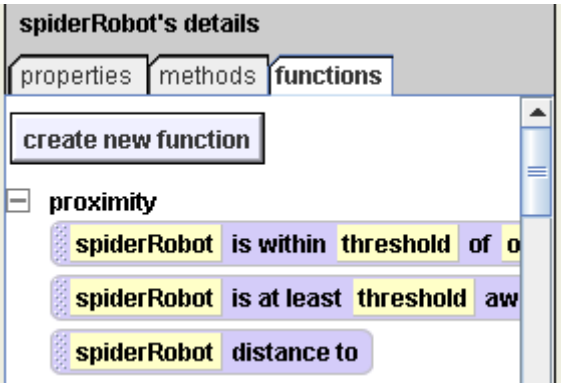
Jusqu'à présent, tous nos programmes écrits avec Alice étaient linéaires (à chaque exécution toutes les instructions étaient toujours parcourues dans le même ordre).
Pourtant, programmer, c'est souvent demander à l'ordinateur de choisir les instructions à exécuter. Nous allons maintenant étudier une technique permettant d'aiguiller le déroulement du programme dans différentes directions, en fonction des circonstances rencontrées. Pour ce faire, nous devons disposer d'instructions capables de tester une certaine condition et de modifier le comportement du programme en conséquence.
La plus simple de ces instructions conditionnelles est l'instruction if. If en anglais signifie " si ". Nous allons donc avoir une structure du type : " si ceci est vrai alors fait cela ". Mais avant d'étudier ces conditions, nous devons d'abord nous arrêter sur les fonctions dans Alice 2.2.

les fonctions dans Alice

Nous avons déjà utilisé l'onglet "properties", l'onglet "methods", il nous reste donc à découvrir l'onglet "functions"

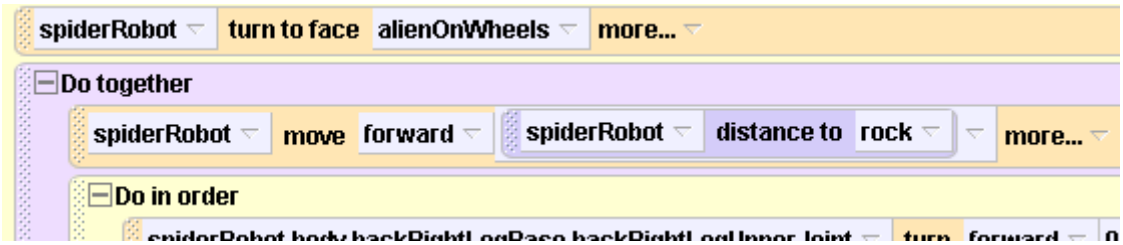


Les fonctions ressemblent fortement aux méthodes, on peut même dire que les fonctions sont des méthodes qui renvoient des valeurs. Quand vous utiliserez une fonction, le résultat de l'exécution sera forcément une valeur (alors qu'une méthode ne renvoie aucune valeur : la méthode "move" déplace un objet, mais ne retourne aucune valeur).
Par valeur je n'entends pas forcément un nombre, en effet une valeur peut être un booléen (voir plus loin), une chaîne de caractère (suite de caractère) ou bien sûr, un nombre.
Un petit exemple s'impose :
reprenons l'exemple du robot et de l'extra terrestre. Le robot avance d'un mètre vers le rocher, mais pourquoi un mètre ? Pourquoi pas 2, 3 ou 4 mètres ? Tout simplement parce qu'avec un cela fonctionne ! Si j'avais pris une trop grande valeur, mon robot aurait pénétré dans le rocher ! Comment faire pour que le robot s'arrête juste au bord du rocher ? Il faut dans un premier temps connaître la distance robot-rocher. C'est là que les fonctions entrent dans la dance. En effet, il existe une fonction qui retourne comme valeur la distance entre 2 objets :



Nous allons utiliser la fonction "spiderRobot distance to" avec l'argument "rock", nous aurons alors la distance entre le robot et le rocher. Mais où placer cette fonction ?
Il ne faut pas oublier qu'une fonction renvoie une valeur, tout se passe comme si au moment de l'exécution du programme, Alice remplaçait la fonction (ici "spiderRobot distance to rock") par la valeur retournée par cette même fonction.

Où avons-nous besoin de cette valeur ?
Dans la fonction "move" (pour remplacer nos "1 meter"), nous allons donc remplacer "1 meter" par la fonction "spiderRobot distance to rock" dans la méthode "spiderRobot move foreward.....". Simple, non ?



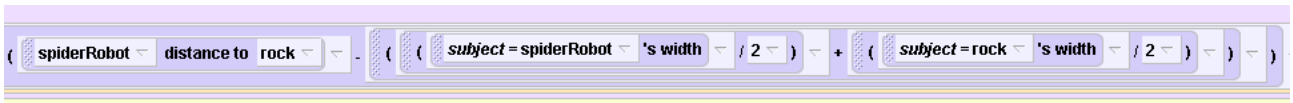
On constate quelques problèmes :
- Le robot se déplace trop vite, mais bon, un petit coup de "duration" devrait pouvoir résoudre le problème.
- L'animation n'est plus vraiment adaptée, mais là encore rien qui ne puisse être résolu.
- Le robot "entre" carrément dans le rocher (il n'y a pas de détection de collision dans Alice).

Là, nous devons plus sérieusement nous pencher sur la question :
la fonction "spiderRobot distance to rock" renvoie la distance entre les 2 centres des objets (ici le robot et le rocher), le résultat obtenu, même s'il ne nous convient pas, est tout à fait logique.
Pour que cela fonctionne il faudrait, à la distance centre rocher centre robot, retrancher la moitié de la largeur du robot plus la moitié de la largeur du rocher (faites un petit schéma pour vous en convaincre !)

résumé :
distance à parcourir par le robot = distance centre robot centre rocher- (largeur du robot/2+largeur du rocher/2)

Bien évidemment, Alice est capable d'effectuer ce genre d'opération avec des fonctions (ne jamais oublier, qu'au cours de l'exécution du programme, Alice "remplace" la fonction par la valeur renvoyée par la fonction).

Il existe des fonctions "largeur des objets" (ex "rock's width" pour la largeur du rocher) et la possibilité d'effectuer les 4 opérations avec les fonctions (utiliser la fonction "math" du menu déroulant)



Pour les opérations il faut d'abord mettre une valeur quelconque (par exemple : "spiderRobot distance to rock-1") puis remplacer la valeur quelconque par la fonction qui nous intéresse (ici on remplace "1" par "spiderRobot's width").

les structures de contrôles

Les booléens

Les valeurs renvoyées par les fonctions ne sont pas forcément des nombres. Une fonction peut aussi renvoyer un booléen.
Qu'est-ce qu'un booléen ?
Un booléen peut prendre uniquement 2 valeurs vraies ou faux (true ou false), un peu comme le binaire. Imaginons la fonction "est une fille", si l'on applique la fonction à l'objet "Robert", la fonction renverra "faux" (à condition que Robert soit un garçon !), si l'on applique cette fonction à "Gertrude", la fonction renverra "vrai" (à condition que Gertrude soit une fille).

Autre fonction possible : "a est égal à b" (traduction Alice : "a = = b", noté le double = qui montre que l'on a affaire à une fonction).
Imaginons le programme suivant:
a=15
b=20
si maintenant on applique la fonction a = = b, elle nous renverra "faux".
Autre programme:
a=15
b=15
la même fonction renverra "vrai".

Le couple if/else

Comme dit dans l'introduction nous allons pouvoir utiliser l'instruction if couplée à une fonction pour "aiguiller" le programme dans une direction ou une autre, exemple :
soit une fonction booléenne (qui renvoie vrai ou faux) : fon1
soient 3 instructions quelconques : inst1,inst2 et inst3

Nous allons trouver la structure suivante :

if fon1 : inst1
else : inst2
inst3

Voilà ce qui va se passer :
- si la fonction fon1 renvoie "vrai" le programme exécutera inst1 puis inst3
- si la fonction fon1 renvoie "faux" le programme va exécutera inst2 puis inst3

L'instruction inst3 est donc quoiqu'il arrive exécutée (elle est en dehors du if/else), en revanche le programme devra "choisir" entre inst1 et inst2 (selon la valeur retournée par fon1)

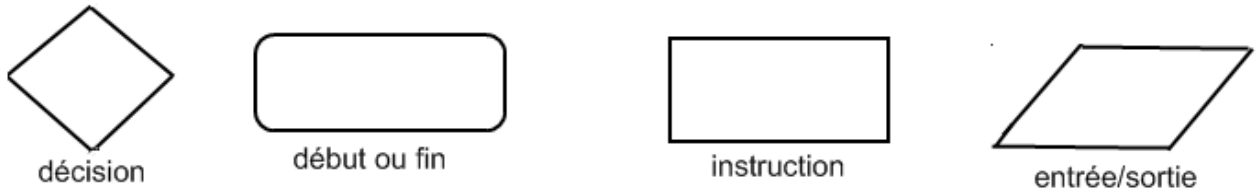
Le else n'est pas obligatoire, on trouve alors la structure suivante :

if fon1 :inst1
inst3

avec pour résultat :
- si la fonction fon1 renvoie "vrai" le programme exécutera inst1 puis inst3
- si la fonction fon1 renvoie "faux" le programme exécutera uniquement inst3

diagramme

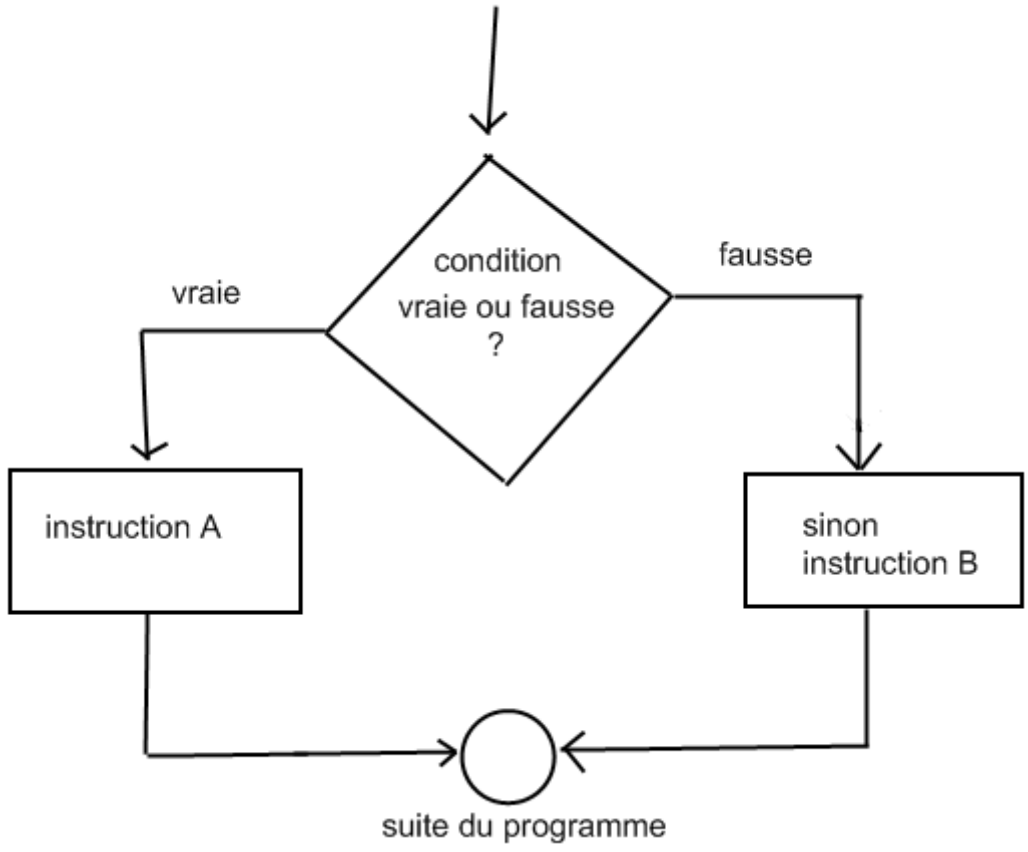
Les programmeurs font souvent appel à des schémas pour représenter la structure de leurs programmes. Ces schémas sont tellement courants que certains symboles sont standardisés :



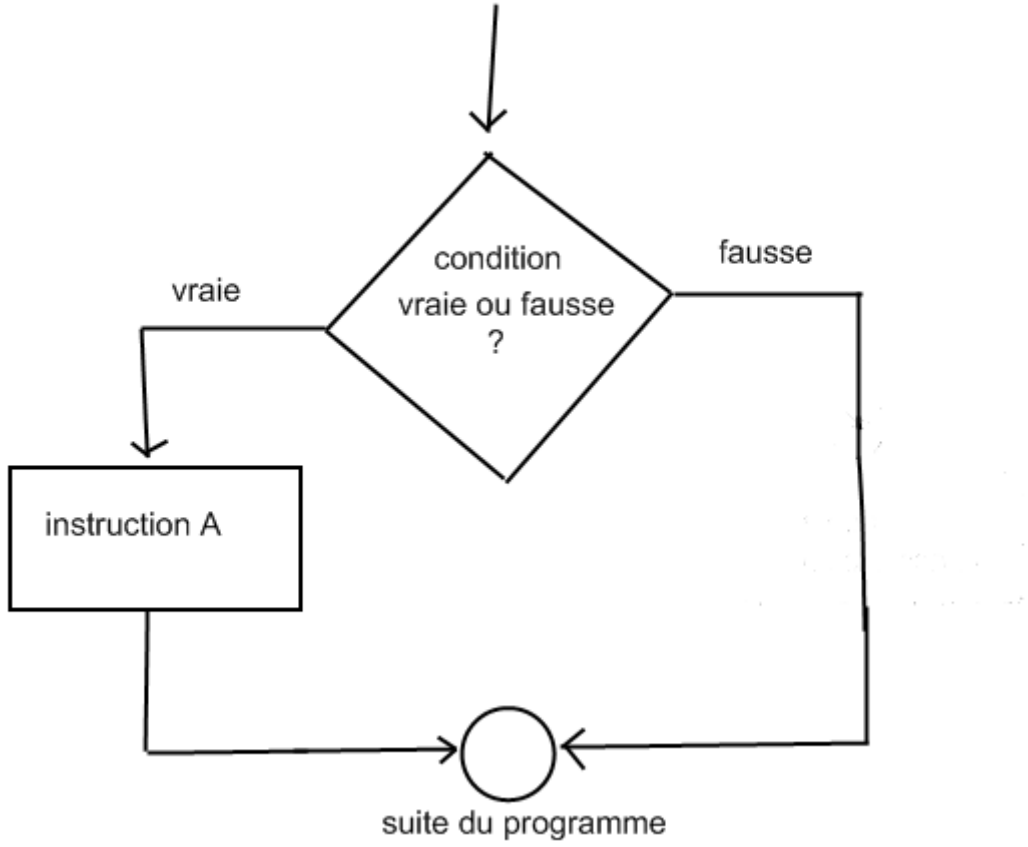
On utilise aussi le rond qui signifie "la suite du programme sur une autre page"

le programme :
if condition :
    instruction A
else :
    instruction B
suite du programme.....

Se traduit donc

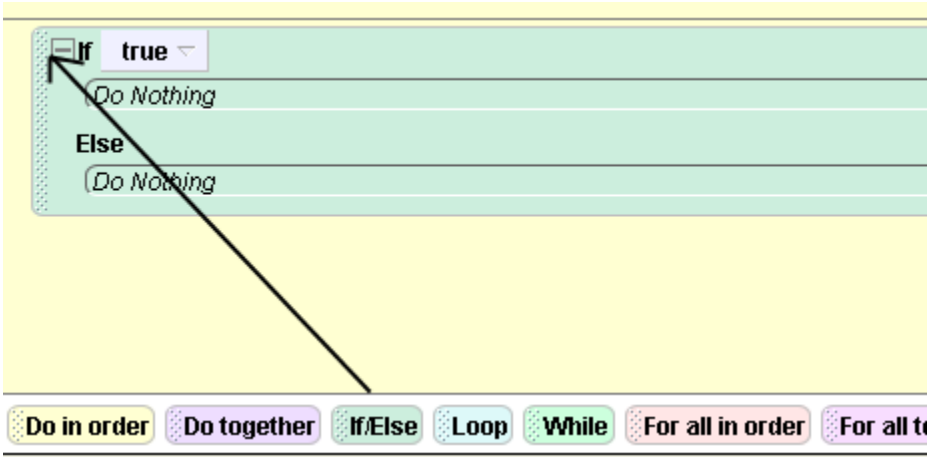


Bien évidemment, là aussi, le else n'est pas obligatoire :



Et dans Alice alors ?

Créer ce genre de structure avec Alice est relativement simple (comme d'habitude !?)



Il suffit (une fois de plus) de faire un "glisser-déposer" et de choisir "true" dans le menu déroulant. Ensuite, il faut remplacer "true" par la fonction qui nous intéresse (cette fonction devra bien sûr renvoyer un booléen).

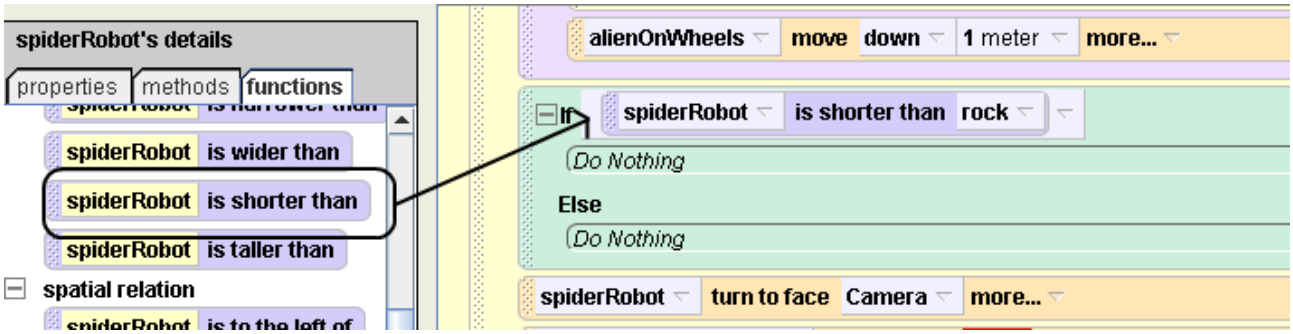
Prenons tout de suite un exemple concret :

Revenons une fois de plus à notre rencontre robot extra-terrestre.

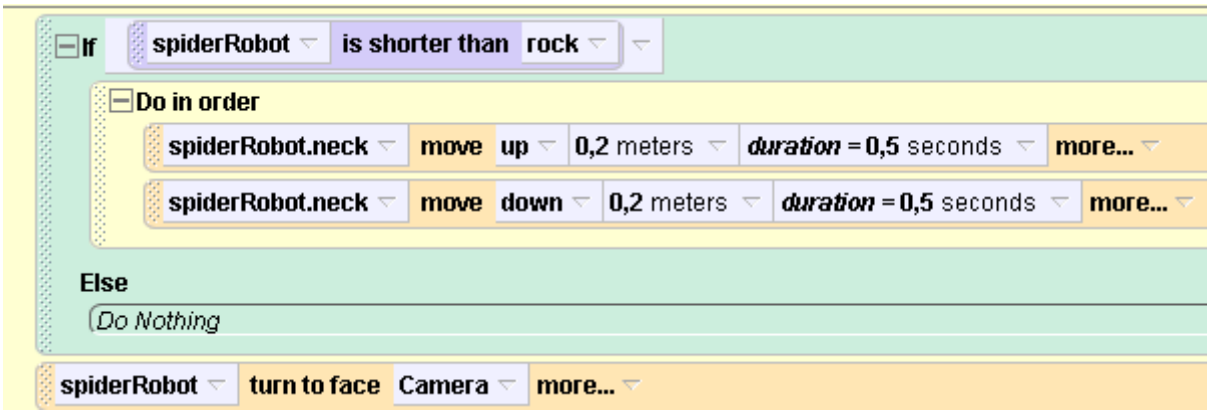
Après avoir été interpelé par l'extra-terrestre, le robot se dirige vers ce dernier. De notre point de vue (le point de vue de la caméra), le rocher est trop grand et l'extra-terrestre n'est plus visible. Mais il n'est pas totalement exclu que la perspective nous trompe et que le robot soit suffisamment grand pour voir l'extra terrestre par-dessus le rocher.

Nous allons donc modifier le programme comme suit :

Si le robot est plus petit que le rocher alors le robot doit allonger son cou sinon (sous entendu le robot n'est pas plus petit que le rocher !) : ne rien faire de plus.
Nous allons utiliser la fonction "spiderRobot is shorter than" avec "rock" comme argument. La fonction nous renverra true si le robot est plus petit et false si le robot n'est pas plus petit.



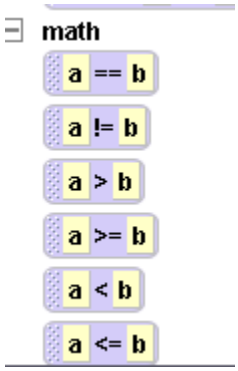
Il ne reste plus qu'à compléter la ligne juste en dessous du if (le robot allonge son cou puis reprend sa position initiale). La ligne en-dessous du else restant vide (sinon : ne rien faire de plus)



Dans ce cas précis, la fonction renvoie true à chaque exécution : le robot est plus petit que le rocher !
Mais juste pour voir ce qui se passe, modifier (suffisamment) la taille du rocher pour que la fonction "spiderRobot is shorter than rock" renvoie " false ", constatez par vous même le résultat : le cou du robot ne s'allonge pas !

Encore quelques fonctions...

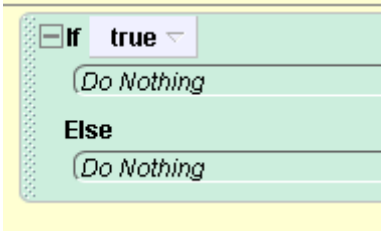
D'autres exemples de fonctions renvoyant des booléens :



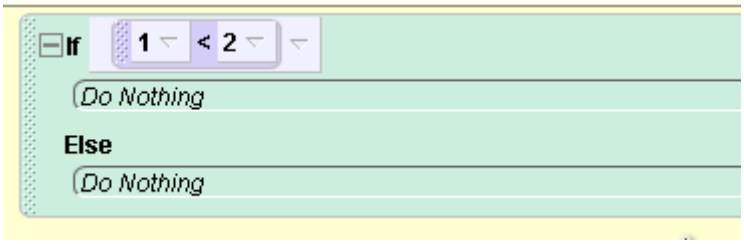
a = = b a est égal à b (si a égal b renvoie true)
a!=b a est différent de b
a>b a est supérieur à b
a>=b a est supérieur ou égal à b
etc..

Pratiquement, comment utilise-t-on ces fonctions dans un if ?

Sélectionner le if comme précédemment (en choisissant true)



sélectionner la fonction qui vous intéresse (par exemple avec a<b choisir une valeur pour a et une valeur pour b.



Remplacer une des 2 valeurs par une autre fonction (en effet dans l'état actuel des choses la fonction renvoie toujours true, car 1 est bien inférieur à 2, l'intérêt du if est donc plus que limité !)
On peut par exemple remplacer le 1 par une fonction qui donne la taille du robot (spiderRobot's height) :

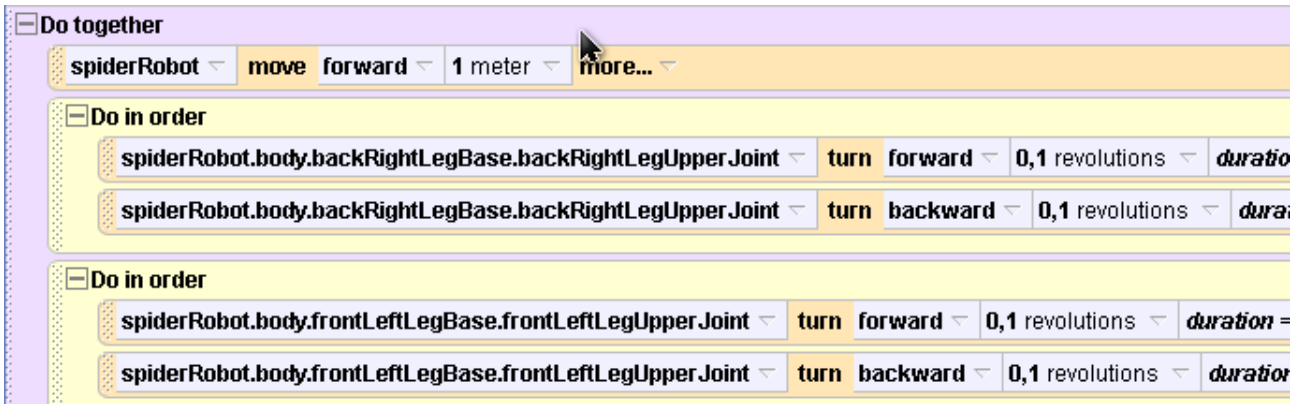
Voilà, si la taille du robot est inférieure à 2 l'instruction sous le if sera exécutée (la fonction a<b renverra true)

les répétitions

Parfois, il est nécessaire de répéter plusieurs fois la même instruction (on pourrait recopier plusieurs fois la même instruction, mais bon .....).

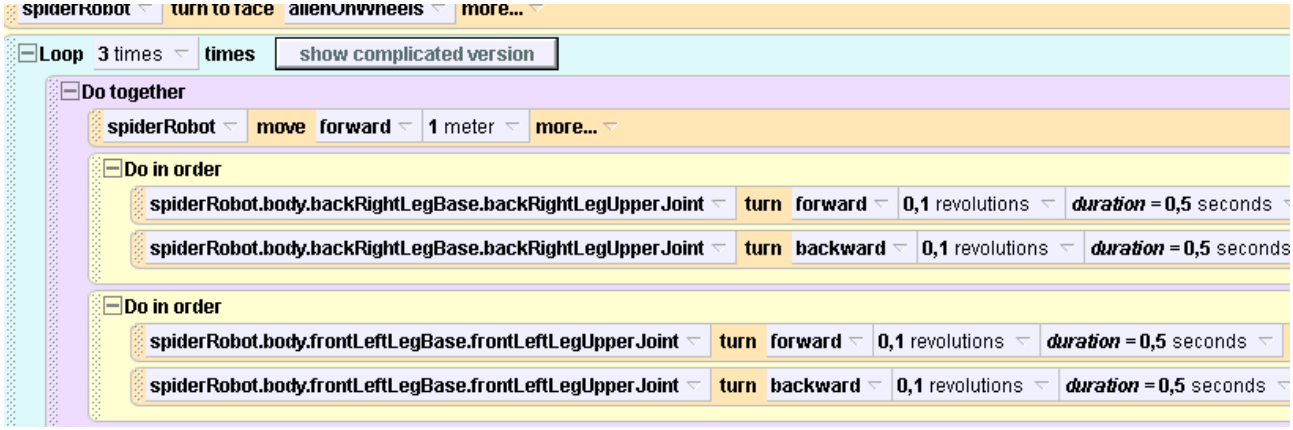
Une fois de plus, revenons à notre robot et notre extraterrestre !

Dans la dernière version de notre programme " fétiche ", l'animation du robot n'est pas (plus) très réaliste, car ce dernier parcourt une grande distance en n'effectuant qu'un seul "pas". Si nous voulons améliorer cette situation, il faudrait revenir à la situation précédente où le robot n'avançait que d'un mètre.
Mais bon, sommes-nous condamnés à ne faire avancer notre robot que d'un mètre ?
Bien sûr que non, si nous voulons que notre robot avance de 3 mètres, il suffit de répéter 3 fois le Dotogether suivant :



Pour cela nous allons utiliser "Loop".

Glisser-déposer "Loop" juste avant le Do together en choisissant "other..." et "3" (les instructions dans la boucle seront exécutées 3 fois)
On obtient donc :



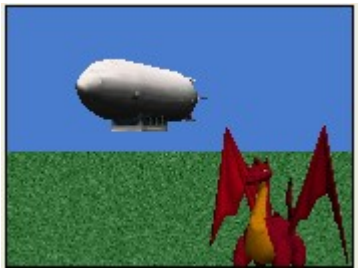
Voilà cela fonctionne !

Exercices chapitre III

Exercice 3.1
Créer une scène avec un loup et une vache. Le loup doit se diriger vers la vache jusqu'à qu'il y ait contact entre les 2 animaux.

Exercice 3.2
Créer une scène avec "skatergirl" et "Pj" (à rechercher dans "People"). Faites sauter en l'air ces 2 personnages, sachant qu'ils sont tous les deux capable de s'élever de ¼ de leur taille.

Exercice 3.3
Un dragon ("Medieval") est intrigué par la présence d'un ballon et décide de se diriger vers lui. Arrivé à 30 m du centre du ballon, il décide d'en faire le tour 3 fois pour mieux l'examiner, à la fin de chaque tour le dragon essaye d'interpeler les occupants du ballon en leur criant "Hello". Désespéré, le dragon décide de se diriger vers la caméra. Écrire le programme correspondant à cette scène.



Exercice 3.4
Créer un monde avec un bonhomme de neige et un tabouret (Kitchen)

Le bonhomme de neige doit se diriger vers le tabouret. Vous devrez utiliser une boucle (Loop). La fonction "distance to" devra être utilisée pour déterminer le nombre de répétitions de la boucle. Il faut savoir que la fonction "distance to" renverra une valeur entière (sans les décimales) si elle est utilisée comme paramètre de "loop". Déplacer le bonhomme de neige pour vérifier que votre programme fonctionne quelque soit la position de départ du bonhomme de neige.

Chapitre IV (1er partie)
Les classes, les instances et les méthodes

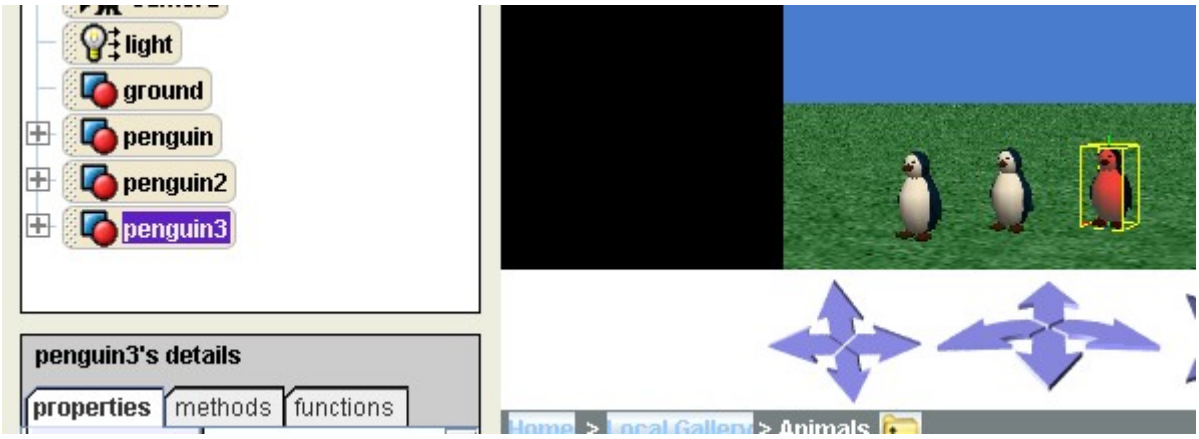
Notion de classe et d'instance

Comme déjà dit précédemment, Alice 2.2 permet l'apprentissage de la programmation orientée objet. Revenons sur ce concept si important en programmation moderne.

Tous les objets proposés dans Alice sont aussi des objets au sens informatique du terme, on parlera d'ailleurs plutôt de classe au lieu d'objet (vous remarquerez que dans le sélecteur d'objet d'Alice, le nom des objets est précédé du mot class). Le nom d'une classe commence toujours par une majuscule (et pas seulement dans Alice).

A chaque fois que vous incorporez un objet dans le monde, vous créez une instance de classe. La classe est en quelque sorte le moule, et l'instance est l'objet que vous fabriquez avec le moule. Comme pour un moule, il est tout à fait possible de créer plusieurs instances différentes (dans une même scène) toutes issues de la même classe (vous l'avez d'ailleurs fait dès le premier chapitre : pour aligner les pingouins, vous avez créé plusieurs instances de la classe pingouin).

Pour vous y retrouver, il est conseillé de donner des noms différents à chaque instance d'une même classe (Alice le fait automatiquement en ajoutant des chiffres: penguin, penguin2, ...). Chaque instance issue d'une même classe est à l'origine identique (tous les objets issus du même moule sont identiques), mais par la suite chaque instance pourra avoir "sa propre vie" (rien ne nous empêche de peindre une pièce en rouge et une pièce en bleu malgré le fait qu'elles sortent du même moule !).



Nous avons créé 3 instances de la classe "Penguin" (penguin, penguin2,penguin3), qui au départ étaient toutes identiques. Ensuite, nous avons modifié un des attributs (properties) de penguin3, sa couleur. Bien évidemment, cette modification n'a eu aucune conséquence sur les autres instances de la classe pingouin.
Enfin pour terminer, il faut noter que la classe "World" contient toutes les autres classes, cela a son importance pour la suite des événements.

les méthodes

Nous avons déjà, à de nombreuses reprises, utilisé les méthodes; nous allons maintenant un peu entrer dans les détails.
Chaque classe (et donc chaque instance de classe) possède des méthodes. Certaines méthodes sont communes à plusieurs classes, on parle alors de "world-level methods" (le nom de ces méthodes commence par World.). Il existe aussi des méthodes, dites "class-level", qui au contraire, ne peuvent être utilisées que par une seule classe (nous verrons des exemples plus loin).

Utilisation des world-level-methods

L'écriture de nos premiers programmes nous a amené à utiliser des méthodes "toutes prêtes" par exemple la méthode "move". Il est possible d'écrire ses propres méthodes, ce que nous avons d'ailleurs déjà fait à de nombreuses reprises, en écrivant la méthode "World.my first method".

Au lieu d'écrire le programme dans une seule méthode (dans "World.my first method"), il est souvent beaucoup plus judicieux de décomposer le programme en plusieurs méthodes et d'appeler ces méthodes au moment opportun.

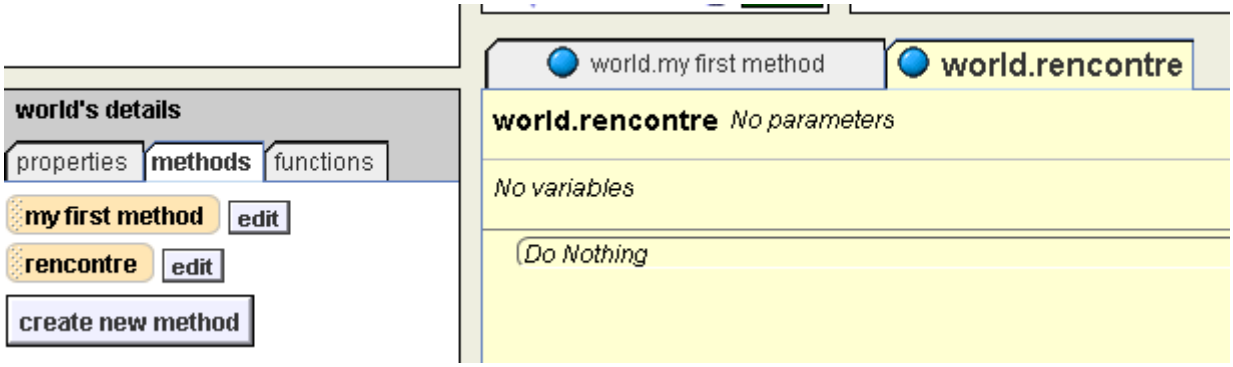
Prenons un exemple simple:
nous allons créer un monde avec un lièvre et une tortue.

- le lièvre et la tortue se dirigent l'un vers l'autre
- la tortue demande au lièvre s'il veut faire la course, le lièvre répond "non, pas aujourd'hui"
- le lièvre et la tortue s'éloignent l'un de l'autre
-

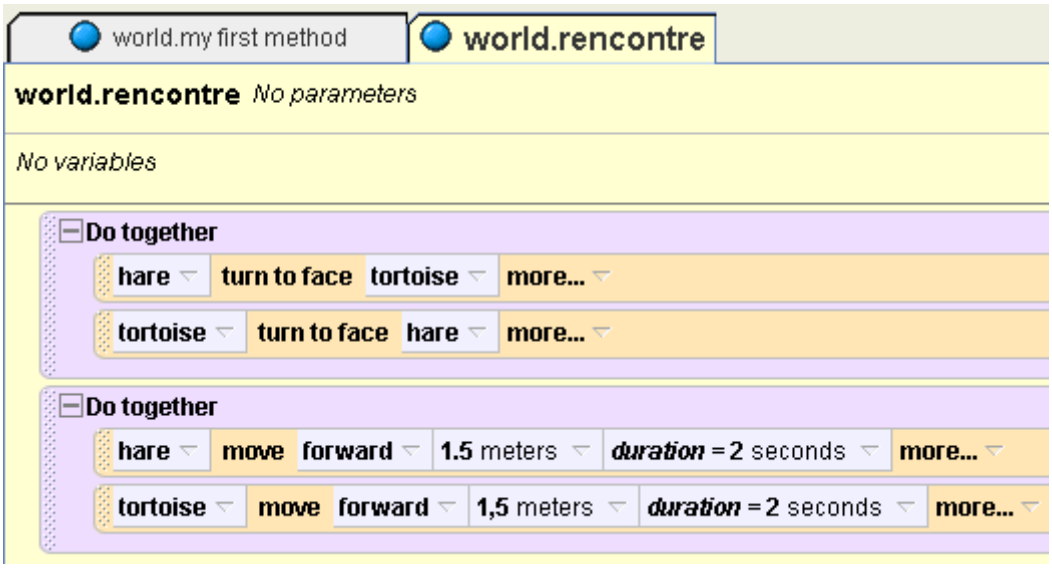
Bien évidemment, nous pourrions très bien écrire le programme dans sa totalité dans la méthode "World.my first method" comme nous l'avons toujours fait jusqu'à présent (c'est sans doute la bonne méthode dans un cas aussi simple), mais pour illustrer notre propos nous allons scinder notre programme en 3 parties en créant 3 méthodes (une par étape).

Ces méthodes seront forcément des "world-class method" car elles concernent 2 classes différentes (Class Tortoise et Class Hare). Passons à la pratique :

En prenant bien soin de sélectionner World (ce sont des "world-class method"), cliquer sur "create new method". Nous allons nommer cette première méthode "rencontre"



Programmons cette méthode "world.rencontre" :



Rien de spécial à signaler, que du connu.

Si nous lançons notre programme, il ne se passera strictement rien, car en faisant preuve d'un peu d'observation, nous pouvons remarquer que la première chose que fait Alice au lancement d'un programme, c'est d'appeler la méthode "world.myfirst method"

Or, la méthode "world.myfirst method" est pour l'instant vide. Passons à la programmation de la 2e méthode, nous reviendrons sur ce point plus tard.

Avant d'attaquer la deuxième méthode, faisons une petite parenthèse et parlons un peu des commentaires. En programmation, il est très important de commenter ses programmes pour expliquer à d'éventuels lecteurs ce que l'on a voulu faire. Pour ajouter un commentaire, il faut cliquer sur "//" en bas à gauche.

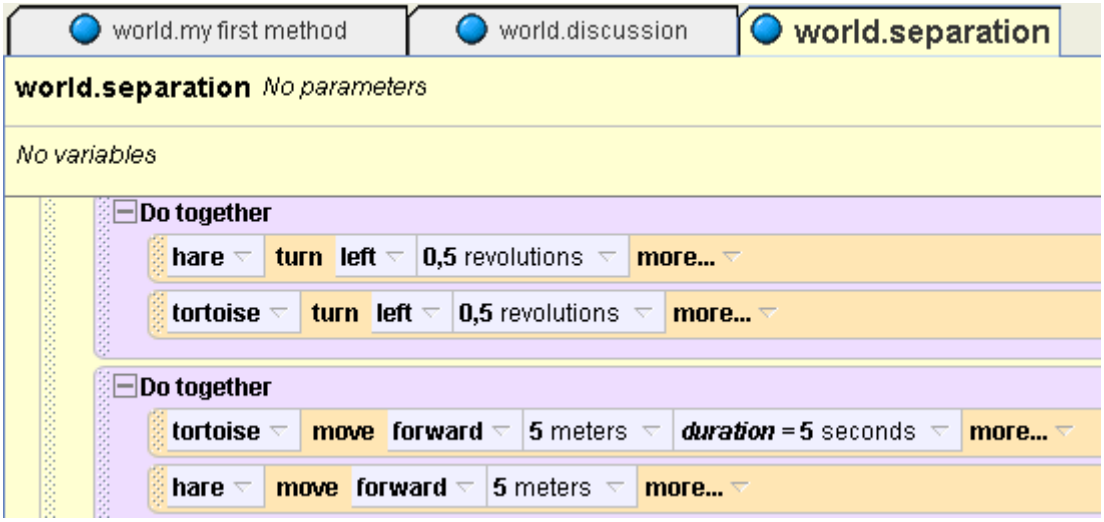
Un petit exemple avec la méthode world.rencontre

Vous pouvez ajouter des commentaires où bon vous semble, fin de la parenthèse.

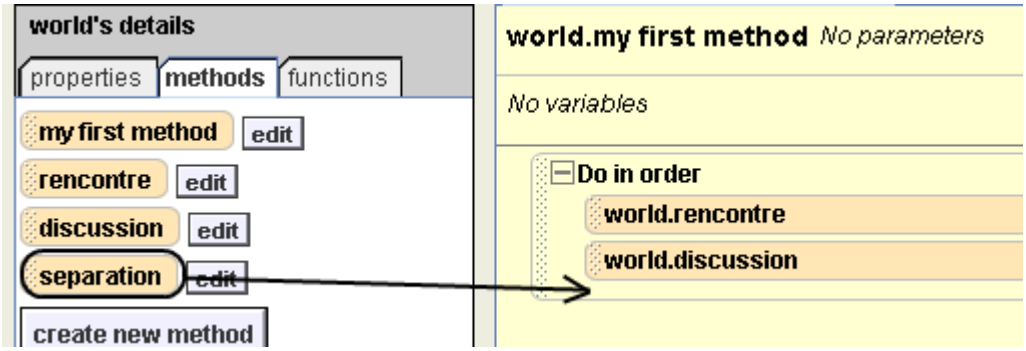
Revenons maintenant à notre seconde méthode.
Nous allons créer une méthode "discussion" (toujours une "world-class method")

En suivant le même cheminement que précédemment nous arrivons à :

et pour la 3e méthode ("separation ", attention pas d'accent pour le nom des méthodes) :



Pour lancer nos 3 méthodes (l'une après l'autre) nous pourrions modifier "Events". Nous allons plutôt choisir une deuxième stratégie :
Quand Alice exécute un programme, il lance par défaut la méthode "world.my first method" (on dira plutôt qu'Alice appelle la méthode "world.my first method"). Il est possible d'appeler une méthode dans une autre méthode.
Nous allons donc compléter la méthode "world.my first method" avec des appels aux méthodes "world.rencontre", "world.discussion" et "world.separation" (une fois de plus tout se fait avec du "cliquer-déplacer" !)



Voilà, vous pouvez maintenant "admirer" le résultat.

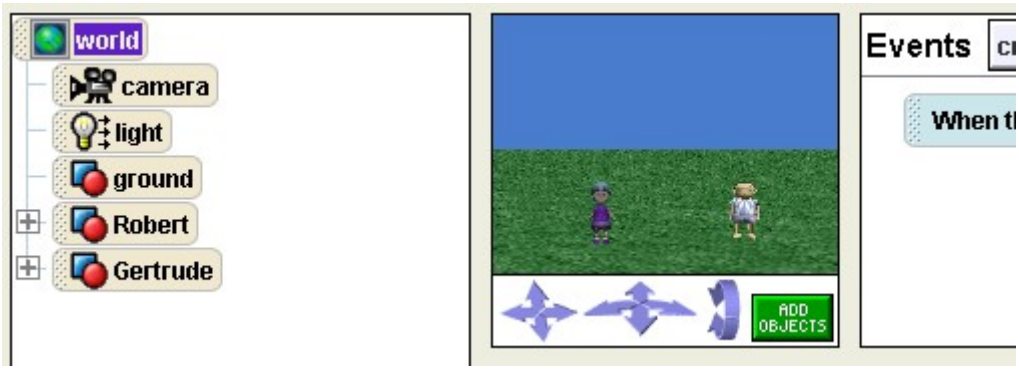
En programmation, il est très courant de segmenter un programme en un grand nombre de méthodes au lieu d'écrire une seule méthode "géante". En effet, cela permet :

- de gérer plusieurs problèmes simples au lieu d'avoir à gérer 1 seul problème complexe
- de travailler à plusieurs sur un même programme (moi je m'occupe de la méthode "rencontre" et toi de la méthode "discussion")
- si une action doit se répéter plusieurs fois dans un même programme, il suffit d'écrire une seule méthode et de faire plusieurs fois appel à cette méthode.
- De rendre les programmes beaucoup plus clairs (lecture facilitée notamment si les méthodes sont correctement commentées).

En bref, il faudra privilégier cette façon de travailler (c'est plus qu'un conseil !!).
Tout développement sérieux devra forcément passer par la phase "fragmentation du problème" et "mise en place de la liste des méthodes à écrire".

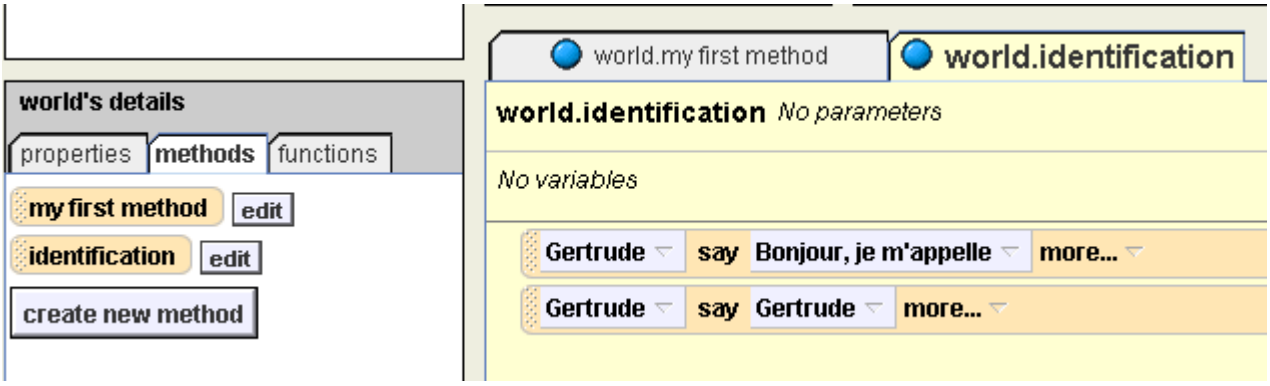
méthodes et paramètres

Créons une scène avec 2 personnages : class Pj et class Mana.L'instance créée à partir de la class Mana se nommera Gertrude et l'instance créée à partir de la class Pj se nommera Robert (on renommera donc les 2 instances)



Nous allons écrire un programme permettant au personnage de se présenter. La présentation se fera en 3 parties : le personnage donne son nom, le personnage avance vers la caméra, le personnage incline sa tête en guise de salut. Nous allons donc écrire 3 méthodes par personnage : une méthode "identification", une méthode "avance" et une méthode "incline" (toutes ces méthodes devront être des "world-class method").

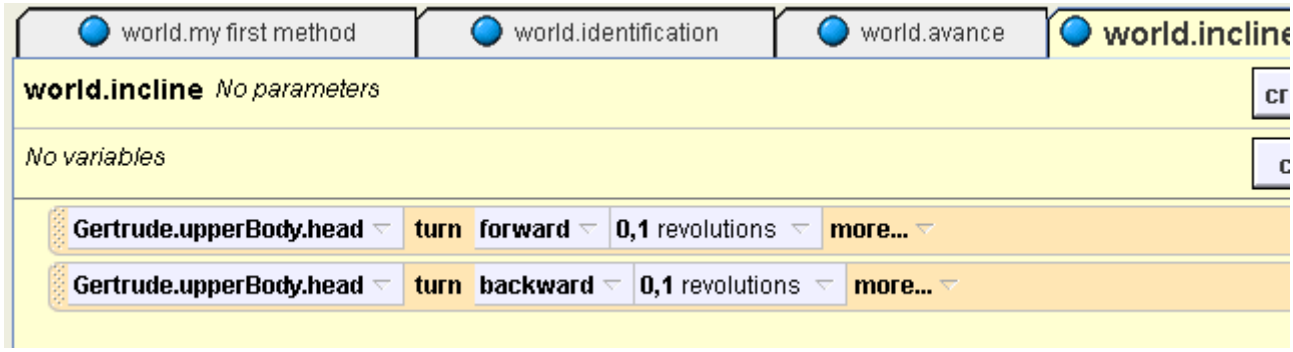
Voici la méthode "identification" pour Gertrude :



Pour tester cette méthode, il suffit de l'appeler dans "world.my first method" comme précédemment.

Voici la méthode "avance" :

et enfin la méthode "incline":

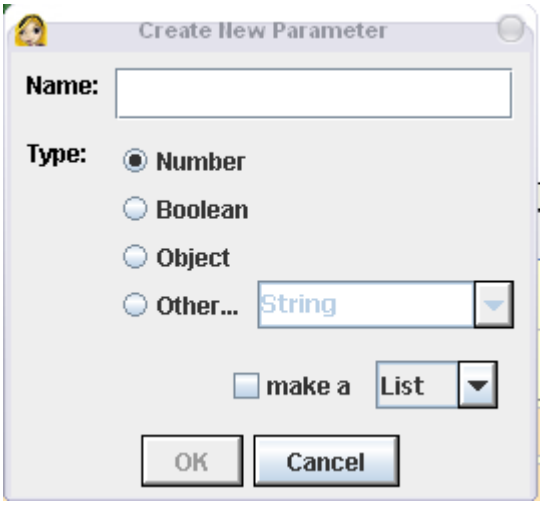


Et maintenant il n'y a plus qu'à réécrire exactement les mêmes méthodes, mais cette fois-ci pour Robert ! Eh bien non ! Il y a beaucoup plus pratique à faire :

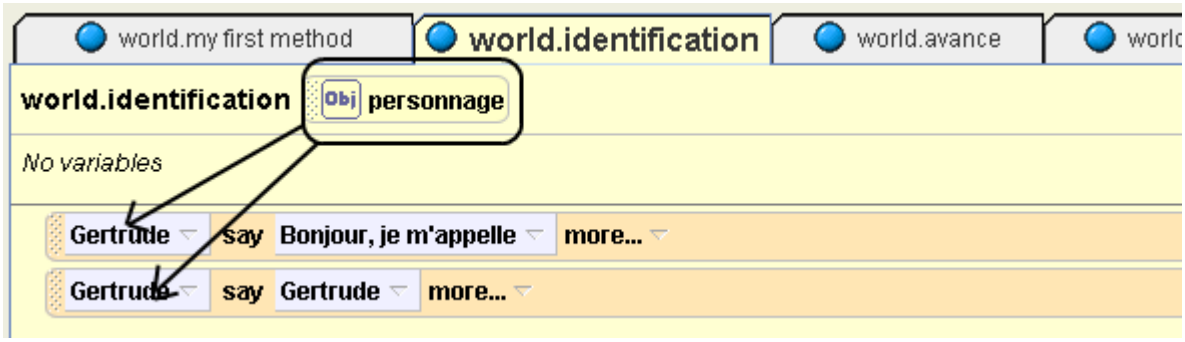
En effet, si l'on regarde toutes ces méthodes de plus prêt, seules 2 choses vont changer : l'instance concernée (c'était Gertrude, cela sera Robert) et la chaîne de caractère (Gertrude dit :"je m'appelle Gertrude" et Robert dira :"je m'appelle Robert"). Cela vaut-il vraiment le coup de tout réécrire pour si peu ?

Pour éviter cela, nous allons utiliser les paramètres des méthodes. L'idée c'est d'écrire une seule méthode (pour les 2 instances) en remplaçant ce qui doit changer d'une instance à l'autre par un paramètre.

Créons un paramètre pour la méthode identification :

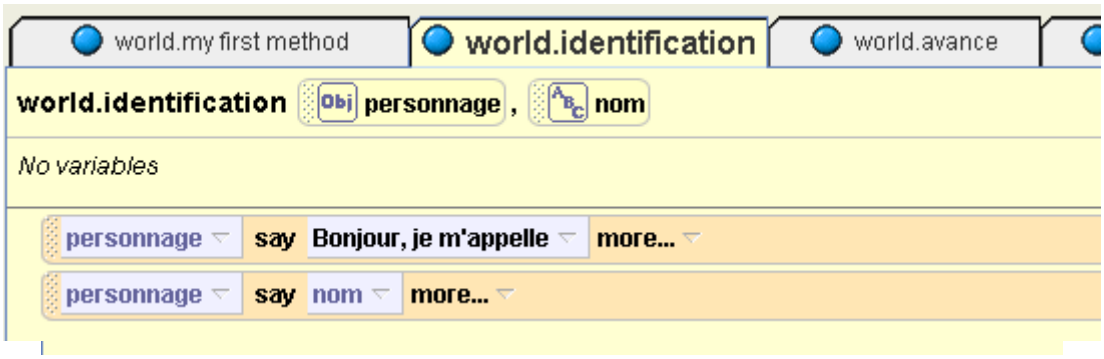


Vous devez entrer un nom (ici cela sera "personnage") et le type de paramètre (ici cela sera un objet)

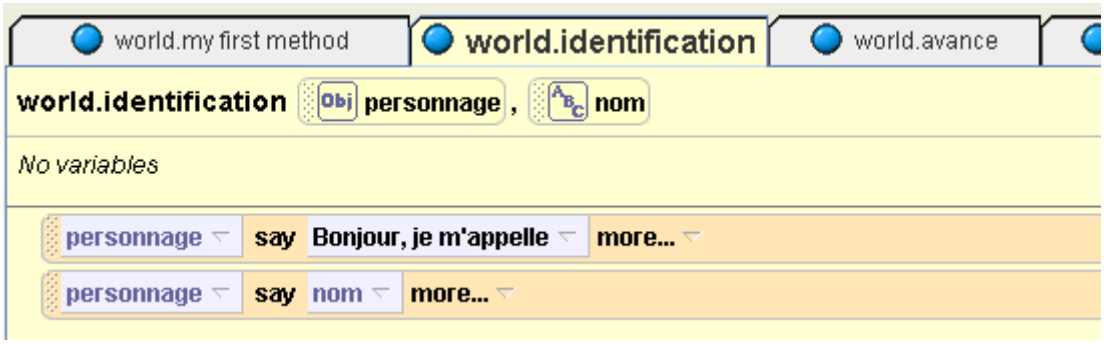


Vous devez ensuite cliquer-déposer le paramètre nouvellement créé à la place de l'objet Gertrude Attention il existe un 3e Gertrude, mais ce 3e Gertrude n'est pas un objet, mais une chaîne de caractère (ce qui s'affiche dans les bulles), pour remplacer ce 3e Gertrude nous aurons à créer un nouveau paramètre.

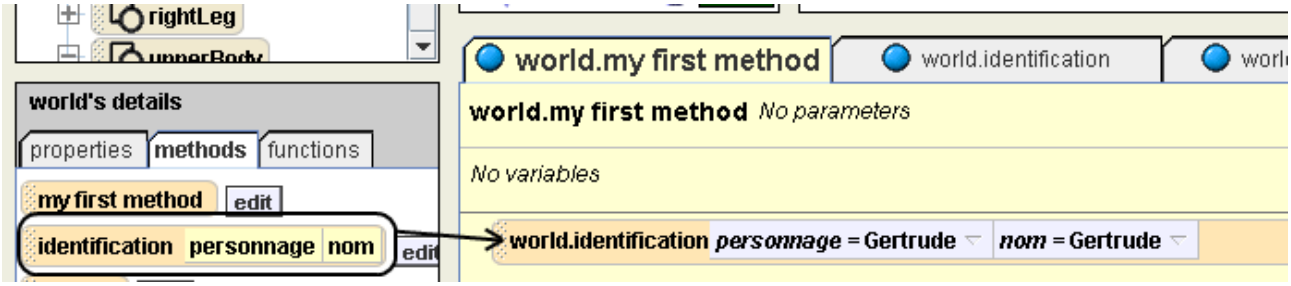
Nous obtenons donc :



Créons un deuxième paramètre de type String (chaîne en anglais) on le nommera "nom". Il devra remplacer le 3e "Gertrude" :

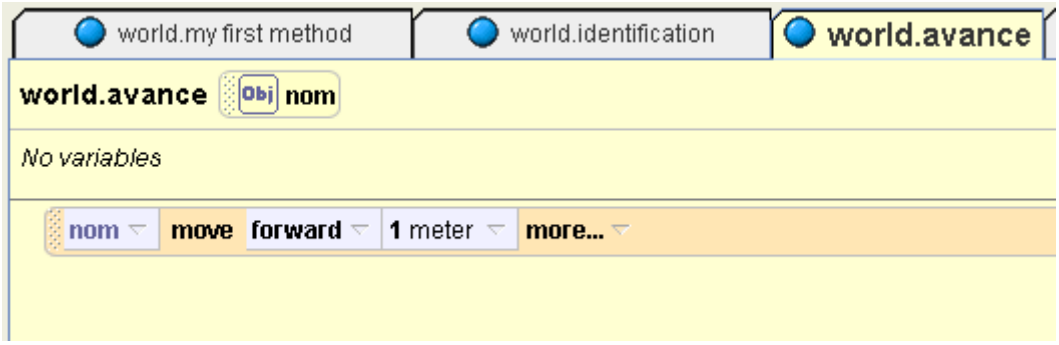


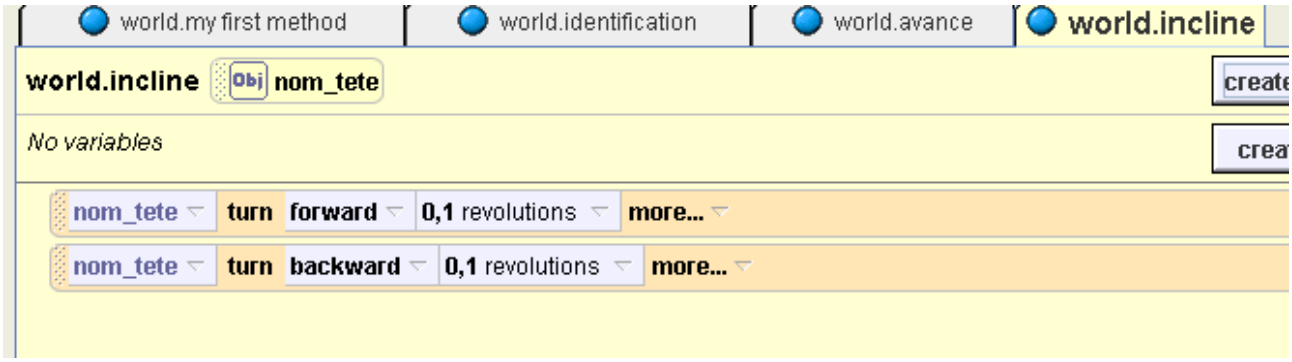
"personnage" et "nom" seront remplacés par les valeurs désirées au moment de l'appel de la méthode world.identification (dans world.my first method)



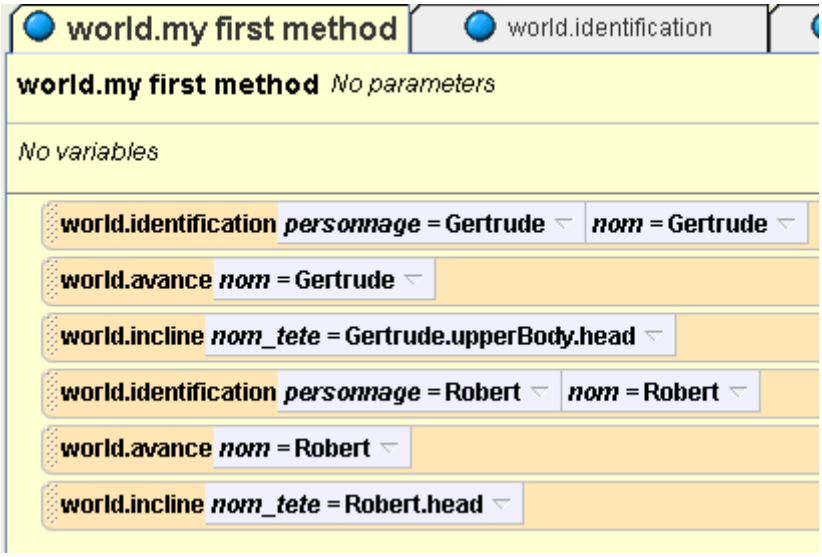
Alice vous demandera par quoi voulez-vous remplacer les paramètres personnage et nom au moment du cliquer-déposer de la méthode identification dans world.my first method.
Vous pouvez d'ores et déjà tester le programme avec uniquement la méthode identification.

Il nous reste à modifier les autres méthodes en créant un paramètre nom de type objet pour la méthode avance et un paramètre nom_tete (attention pas d'accent) de type objet pour la méthode incline.

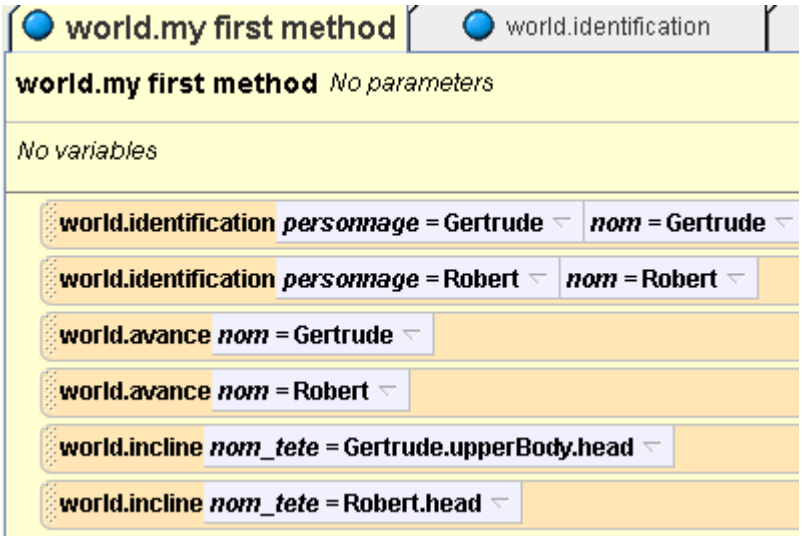




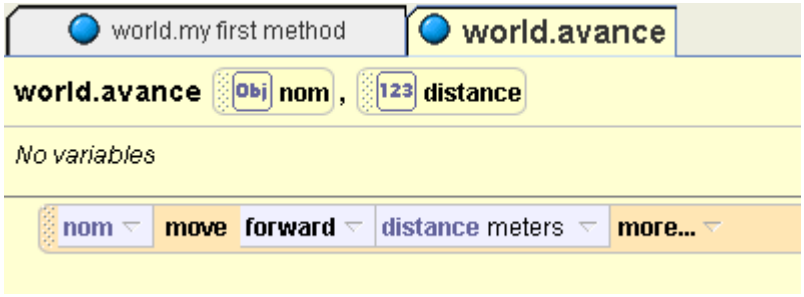
Complétons la méthode world.my first method avec les appels aux différentes méthodes.

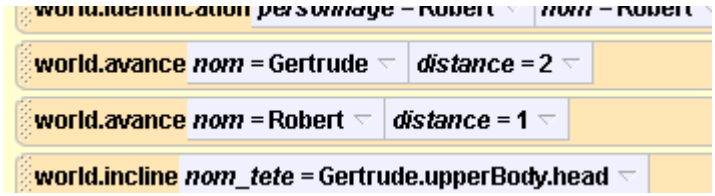


Vous pouvez aussi modifier l'ordre des appels aux méthodes



Vous pouvez aussi créer un paramètre "distance" de type nombre, si vous voulez décider de la distance à faire parcourir au personnage au moment de l'appel de la méthode avance.





Exercices chapitre IV
1ère partie

Exercice 4.1

Un paysage enneigé, un bonhomme de neige, une bonnefemme de neige. Le bonhomme de neige salut la bonnefemme de neige en enlevant puis en remettant son chapeau. La bonnefemme de neige lui repond avec le même geste.
Ecrire une World-level-method permettant au bonhomme de neige et à la bonnefemme de neige de se saluer comme décrit ci-dessus (attention une seule méthode pour les deux !). Pourquoi n'écrit-on pas une Class-level-method ?

Exercice 4.2

Composer une scène avec un magicien, une assistante de magicien, un lapin et une table. L'assistante du magicien est allongée sur la table. Ecrire une méthode "levitation" permettant au magicien de faire voler (lévitation) aussi bien le lapin que l'assistante.

Exercice 4.3

Placer 4 dragons au sol pour qu'ils forment un losange. Créer une méthode "vol_du_dragon" permettant à 2 dragons d'échanger leur place. Les déplacements devront se faire en volant. Les dragons ne devront pas voler à la même altitude à cause du risque de collision. La méthode "vol_du_dragon" devra comporter 4 paramètres.

Chapitre IV (2e partie)
Les méthodes Class-Level et l'héritage

les Class-Level method

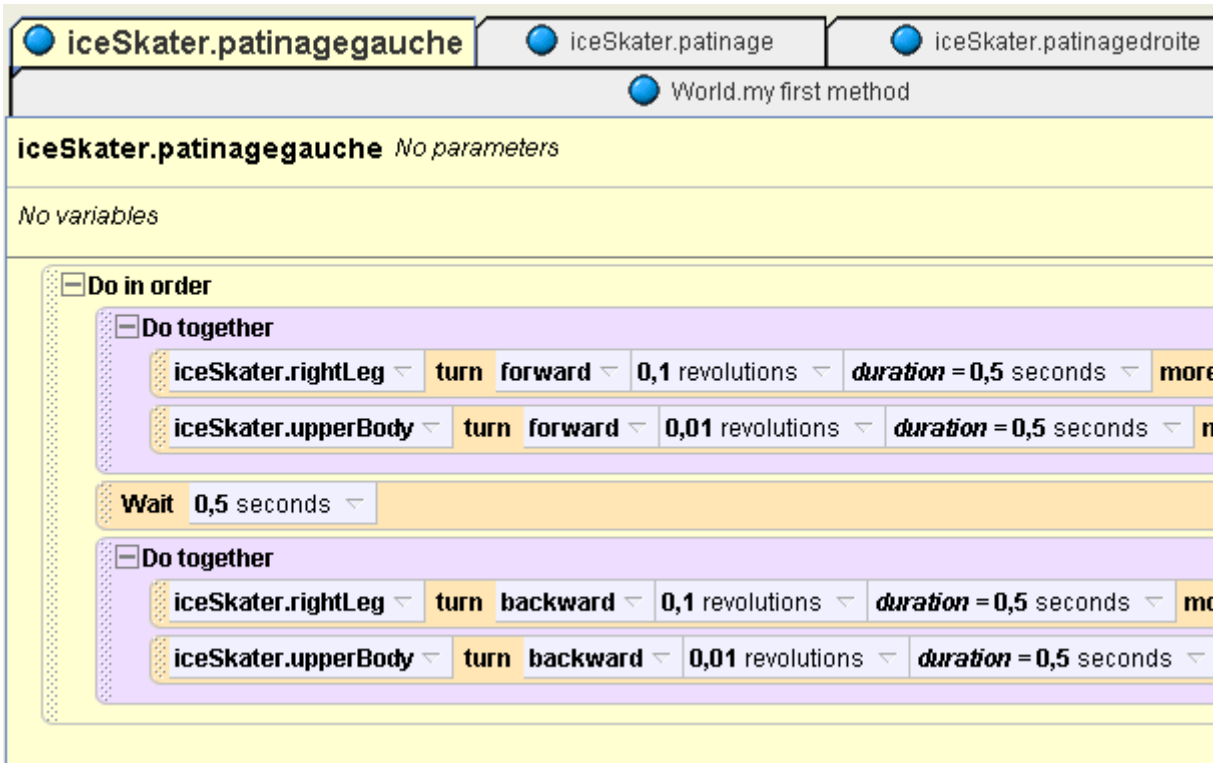
Nous allons maintenant nous intéresser aux Class-Level method. Une Class-Level method est écrite pour une classe donnée. Par exemple la méthode penguin.wing_flap est une Class-Level method de la Class Penguin (on peut remarquer que le nom de la méthode ne commence plus par world, mais par le nom de la classe et que cette méthode admet un paramètre "times" (nombre de battements d'aile)).
Voici l'appel de la méthode :

Nous allons ajouter une Class-Level method qui permettra à une instance de la classe IceSkater de patiner, car comme vous pouvez le constater la classe IceSkater ne possède pas ce genre de méthode. Dit autrement, nous allons apprendre à la patineuse à patiner !

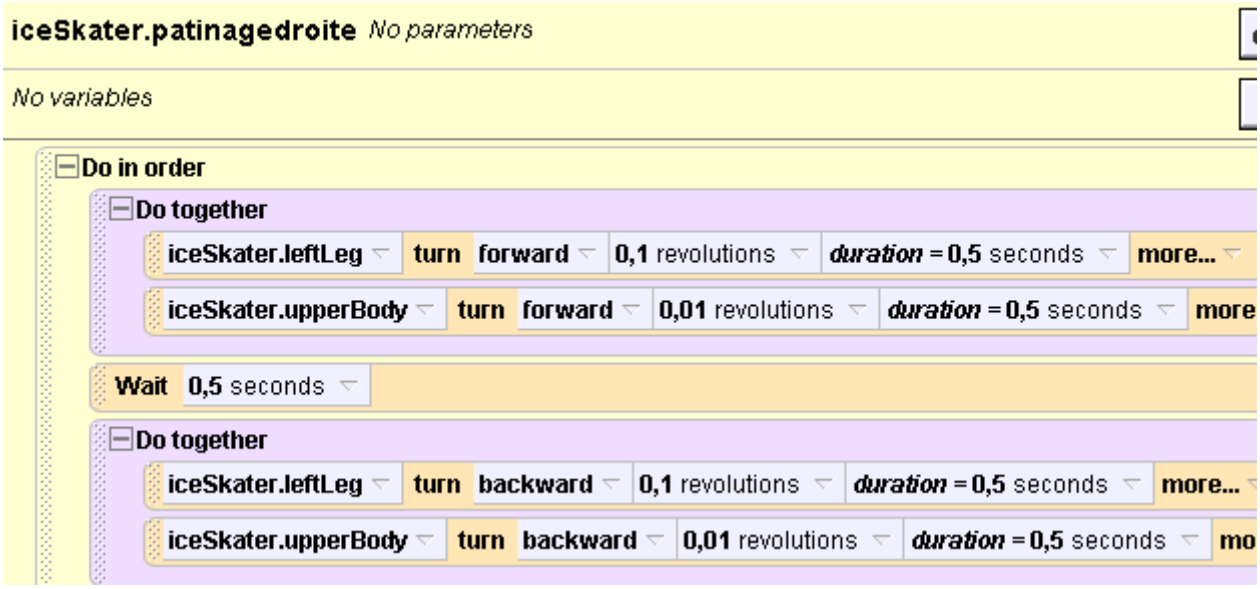
Le patinage est un mouvement complexe à simuler, car il fait appel à plusieurs parties du corps.
Pour patiner, la patineuse doit faire glisser vers l'avant sa jambe gauche puis faire glisser vers l'avant sa jambe droite, la difficulté réside dans le fait que le haut du corps doit aussi suivre ce mouvement.
En programmation quand on a affaire à un problème complexe, on essaye de le fragmenter en plusieurs problèmes moins complexes. Ce n'est donc pas une méthode que nous allons créer, mais trois :
Une méthode patinage, qui fera avancer la patineuse et qui servira à appeler tour à tour les 2 autres méthodes (patinagegauche et patinagedroite).

Créons la méthode patinagegauche :
Dans un premier temps, sélectionnez l'instance iceskater puis cliquez sur "create new method ", appelez-la " patinagegauche ", voilà, vous avez créé votre première Class-Level method (vous pouvez constater que le nom de la méthode commence bien par IceSkater)

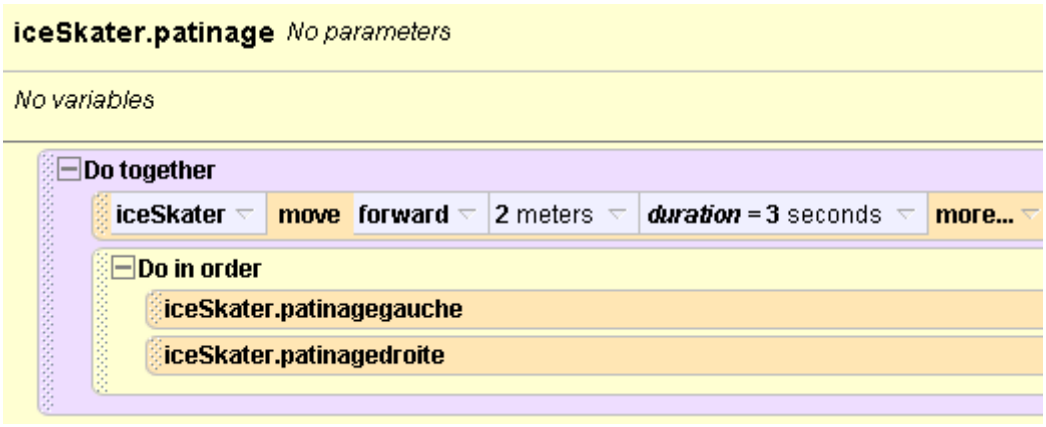
Il ne nous reste plus qu'à compléter notre méthode (il n'y a rien de spécial à dire, que du connu, sauf l'instruction wait) :



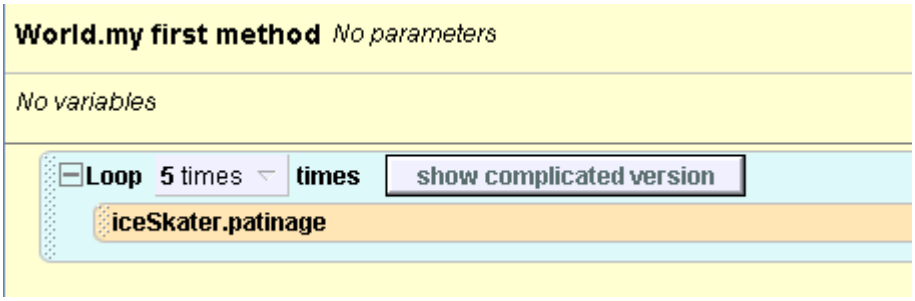
Voici patinagedroite :



et enfin la méthode patinage :



N'oublions pas la méthode world.my first method (si vous voulez avoir le temps d'admirer la patineuse, pourquoi ne pas faire appel à une petite boucle ?)

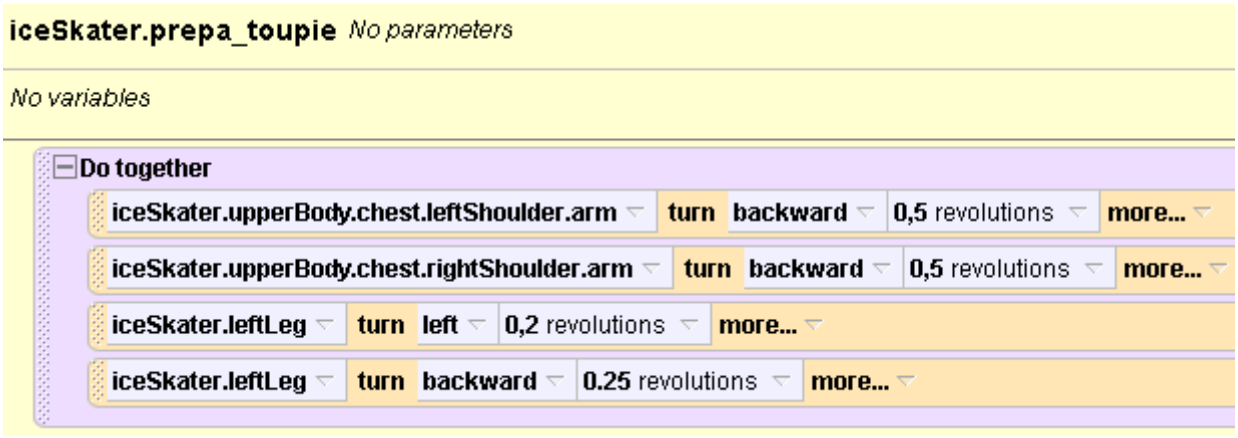


Les Class-Level method et les paramètres

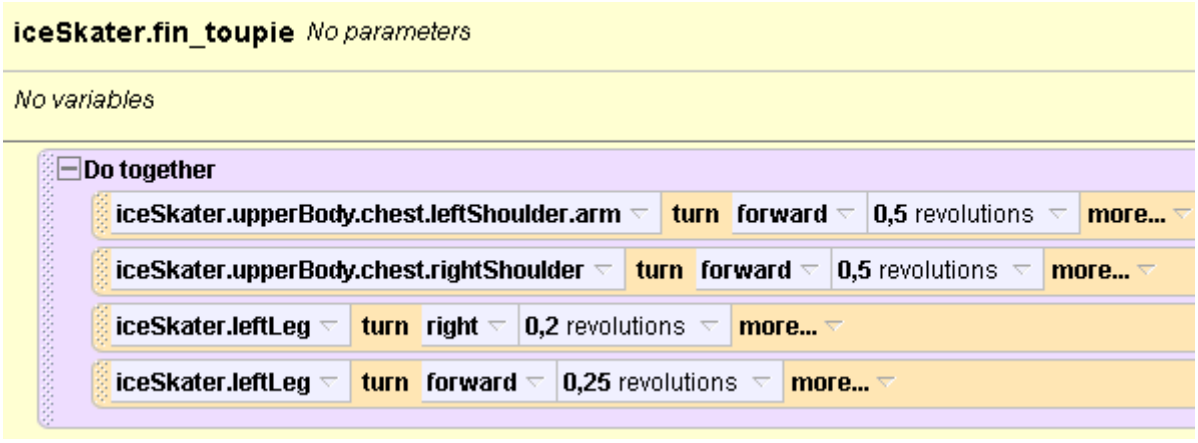
Comme pour les World-Class method, les Class-Level method peuvent admettre des paramètres. Écrivons une autre Class-Level method pour la classe IceSkater.

Notre patineuse sait désormais patiner, nous allons maintenant lui apprendre à faire la toupie. Nous allons utiliser 3 méthodes : une méthode pour préparer la toupie (prepa_toupie), une méthode pour effectuer la toupie (toupie) et une méthode pour terminer la toupie (fin_toupie). Les méthodes prepa_toupie et fin_toupie seront des Class-Level method sans paramètre. La méthode toupie sera une Class-Level method avec un paramètre (le nombre de tours à effectuer). De plus, cette méthode appellera la méthode prepa_toupie et la méthode fin_toupie.

La méthode prepa_toupie :

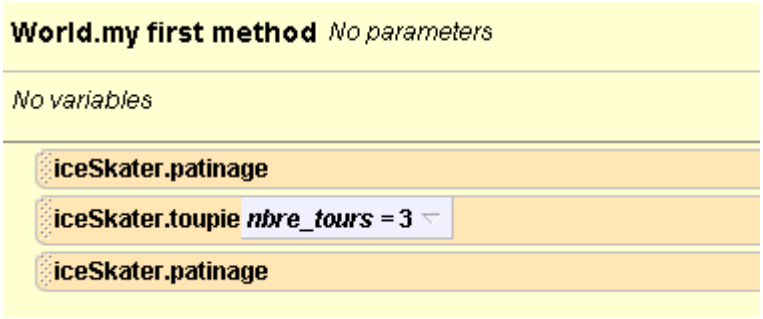


La méthode fin_toupie :



La méthode toupie avec l'appel aux autres méthodes et le paramètre nbre_tours :

et enfin la méthode World.my first method (patine-toupie-patine) :



création d'objet et héritage

Notre instance iceSkater (attention je parle bien de l'instance, pas de la classe IceSkater, j'espère que maintenant, vous faites bien la différence !) possède maintenant 2 nouvelles méthodes (nous avons créé plus de 2 méthodes, mais il n'y en a que 2 directement utilisables : patinage et toupie). Si nous créons un nouveau monde et que nous utilisons de nouveau la class IceSkater pour créer une nouvelle instance, nos 2 nouvelles méthodes auront disparu.

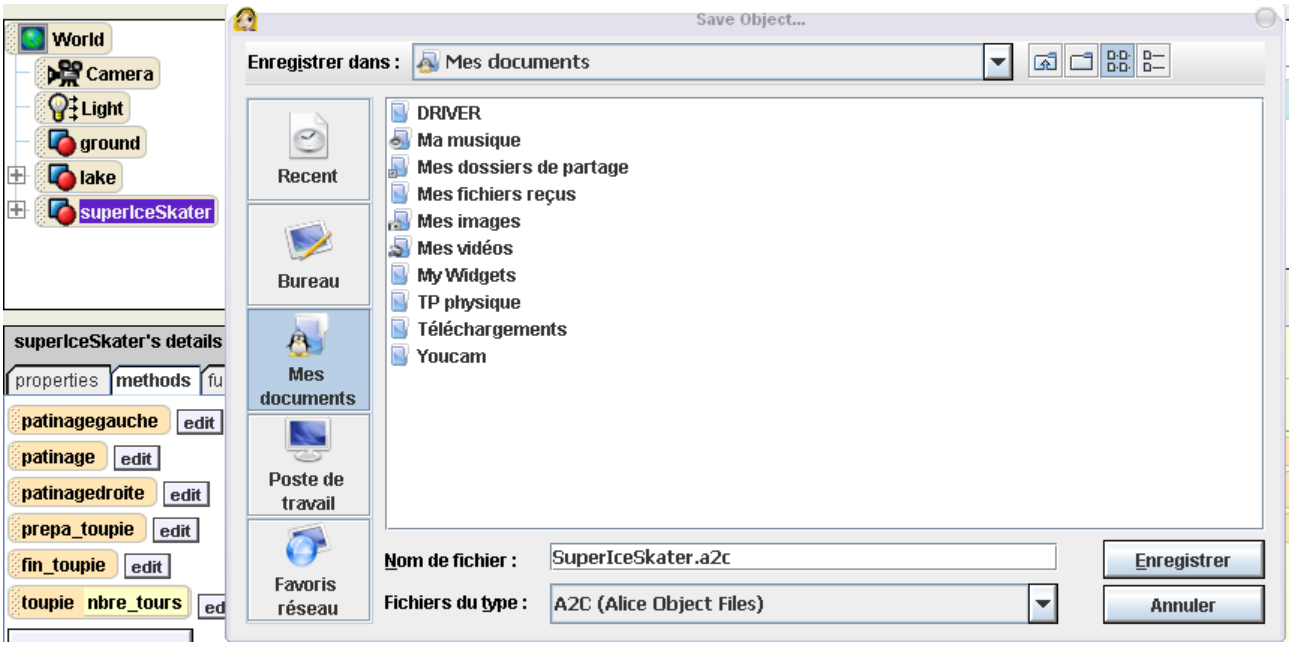
Faut-il "s'amuser" à réécrire nos méthodes à chaque fois que nous créons une instance à partir de la classe IceSkater ? Bien sûr que non !

Nous allons utiliser notre instance iceskater (avec ces métodes supplémentaires), pour créer une nouvelle classe SuperIceSkateur (nous aurions pu lui donner n'importe quel nom !). Cette nouvelle classe bénéficiera de toutes les méthodes de la classe IceSkater (puisque notre instance iceskater est issue de la classe IceSkater) mais aussi des nouvelles méthodes que nous venons de créer (patinage et toupie). Nous dirons que la classe SuperIceSkater descend de la classe IceSkater (classe parent : Iceskater classe enfant : SuperIceSkater). Nous dirons aussi que la classe SuperIceSkater hérite des méthodes, des attributs(properties) et des fonctions de la classe IceSkater. Cette notion d'héritage est très importante en POO .

La personne qui utilisera la nouvelle classe SuperIceSkater pourra utiliser les méthodes patinage et toupie sans se soucier le moins du monde du code qu'il a fallu écrire pour les programmer. Encore une fois, nous sommes ici au cœur de la POO. Un programme écrit avec un langage du type C++, Java ou Python (qui accepte la POO) fonctionnera exactement sur ce principe.

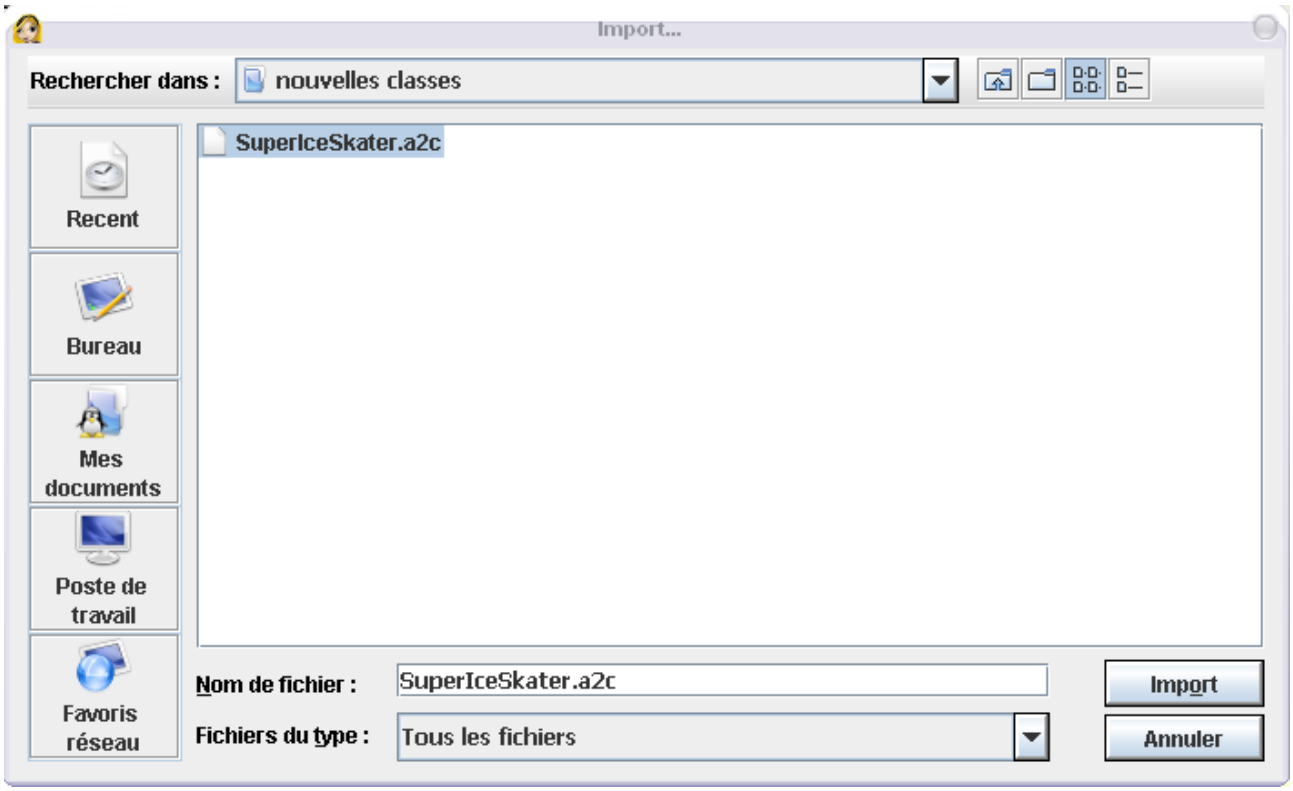
Assez parlé théorie, passons à la pratique :

Pour créer notre nouvelle classe SuperIceSkater, il faut renommer notre instance iceskater en superIceSkater (encore une fois pas de majuscule au début du nom, pour l'instant, c'est toujours une instance pas encore une classe (oui, je sais la différence est un peu difficile à comprendre, mais faites un effort, c'est très important !).
Ensuite, cliquer droit sur superIceSkater et choisir "save object...". Il ne vous reste plus qu'à sauvegarder votre objet (votre nouvelle classe) dans un endroit approprié.

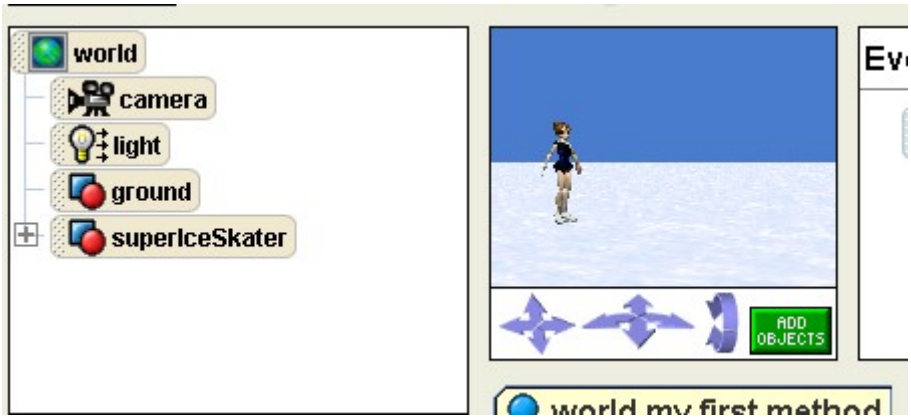


Noter qu'Alice a mis un S à la place du s (nous sauvegardons bien une classe !)

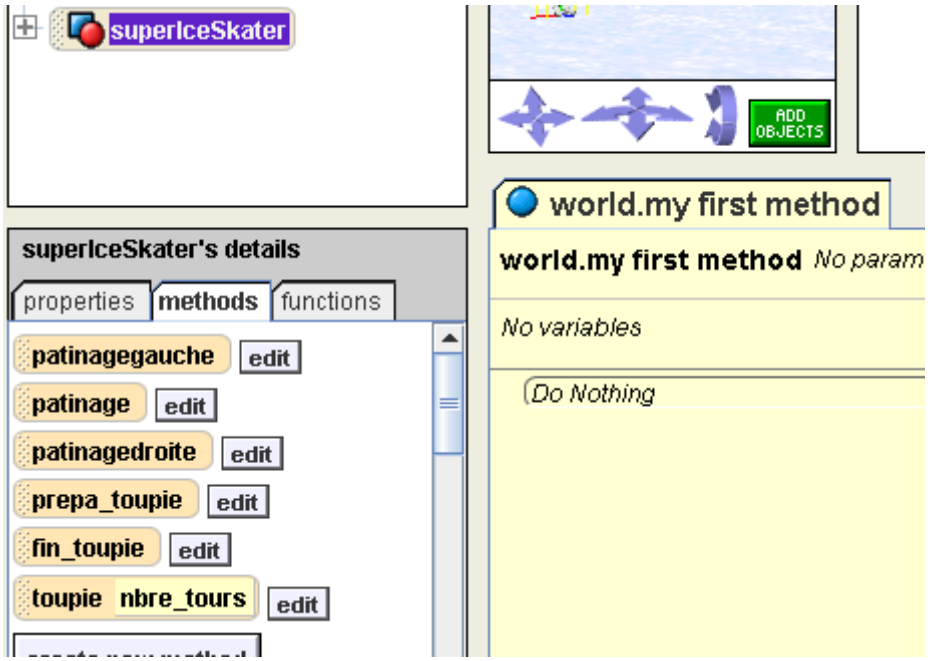
Si maintenant vous (ou un autre) désire utiliser notre nouvelle classe SuperIceSkater, il suffira de cliquer sur "File" (menu du haut), et de choisir "Import" dans le menu déroulant :



Et voici un nouveau monde avec une instance superIceSkater issue de notre nouvelle classe SuperIceSkater.

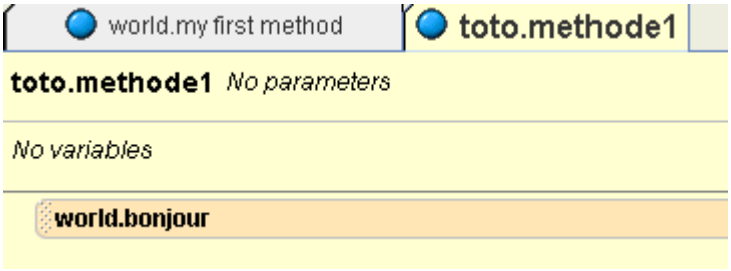


Et comme vous pouvez le constater, notre instance iceSkater possède bien toutes les nouvelles méthodes :



Quelques règles à respecter

Il ne faut jamais appeller de World-level-method depuis une Class-level-method.



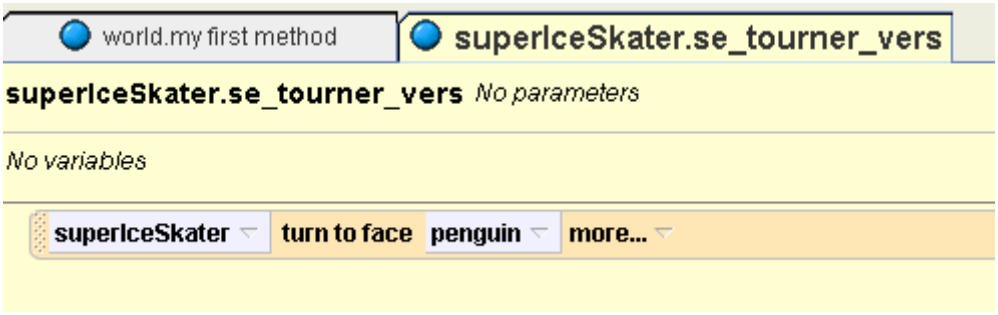
A ne surtout pas faire !

Prenons un exemple simple : nous programmons une nouvelle Class-level-method pour une instance toto (issue de la classe Toto) : toto.methode1.
La méthode toto.methode1 fait appel à une World-class-method : World.bonjour (programmée par nos soins).
Créons une nouvelle classe SuperToto à partir de l'instance toto.

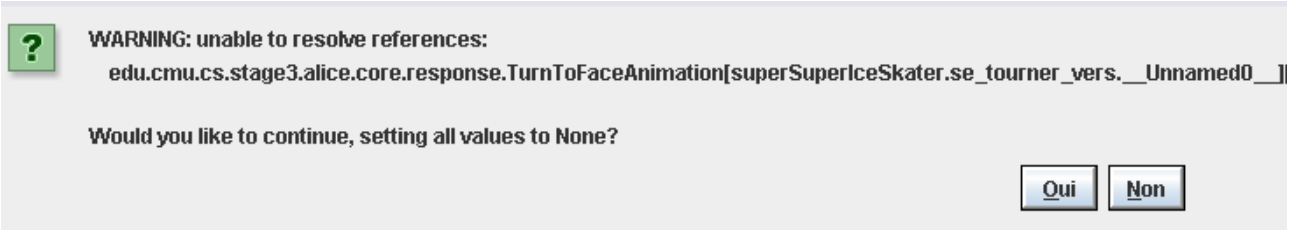
Si nous essayons d'utiliser notre nouvelle classe SuperToto dans un autre monde (un nouveau programme), Alice nous renverra un méchant message d'erreur : "World.bonjour non définie". En effet dans notre nouveau programme la méthode World.bonjour n'existe pas , d'où le message d'erreur.

Dans le même genre, il ne faut jamais utiliser une instance dans une Class-level-method :
Retrouvons notre superIceSkater. Nous allons écrire une nouvelle Class-level-method pour superIceSkater extrémement (ridiculement ?) simple qui aura pour seul interêt, de vous montrer, ce qu'il ne faut surtout pas faire.

Avant cela, ajoutons un pingouin à notre scène. Notre nouvelle Class-level-method (se_tourner_vers) ne comportera qu'une seule instruction :



Imaginons maintenant qu'extrêmement fier de notre nouvelle méthode, nous décidions de créer une nouvelle classe SuperSuperIceSkater pour pouvoir profiter de cette nouvelle méthode !!
Et là, catastrophe :



En bref, Alice vous dit :"penguin, connaît pas !". En effet dans votre nouveau programme l'instance penguin n'existe pas. Voilà pourquoi il ne faut surtout pas utiliser une instance dans une Class-level method.
Comment faire alors ?

Il faut tout simplement utiliser un paramètre de type objet à la place de l'instance penguin.

Comme ça, pas d'erreur en cas de création et d'utilisation d'une nouvelle.
Ensuite, rien de nouveau !

Exercices chapitre IV
2e partie

Exercice 4.4

Créer un monde avec un cadenas à combinaisons (à rechercher dans objects)
Coder 6 Class-level-method :

- un_cran_a_gauche (tourne la roue du cadenas d'un cran vers la gauche)
- un_cran_a_droite (tourne la roue du cadenas d'un cran vers la gauche)
- un_tour_a_droite (tourne la roue du cadenas d'un tour vers la droite)
- un_tour_a_gauche (tourne la roue du cadenas d'un tour vers la gauche)
- ouvrir (ouvre le cadenas)
- fermer (ferme le cadenas)

Une fois les méthodes codées, créer une nouvelle classe qui héritera de la classe Combolock (par exemple CombolockPlus)

Exercice 4.5

Exercice libre : choisir un objet, créer pour lui une nouvelle class-level-method (avec au moins 1 paramètre), à partir de l'instance précédente, créer une nouvelle classe. Imaginer une scène permettant d'illustrer l'utilisation de votre nouvelle classe (et surtout de votre nouvelle méthode).

Chapitre V
Interaction utilisateur - machine

Nos programmes commencent à être complexes, mais jusqu'à présent, il manque l'essentiel :

l'I-N-T-E-R-A-C-T-I-V-I-T-E. L'utilisateur est pour l'instant uniquement spectateur, or, dans la plupart des programmes, il interagit avec la machine par l'intermédiaire de la souris et du clavier. Pour gérer cette interaction homme-machine, tous les programmes modernes sont pourvus d'un observateur d'événements. L'observateur d'événements surveille en permanence les périphériques d'entrées (clavier et souris dans la plupart des cas) et réagit en conséquence lors de l'utilisation de ces périphériques.

et Alice

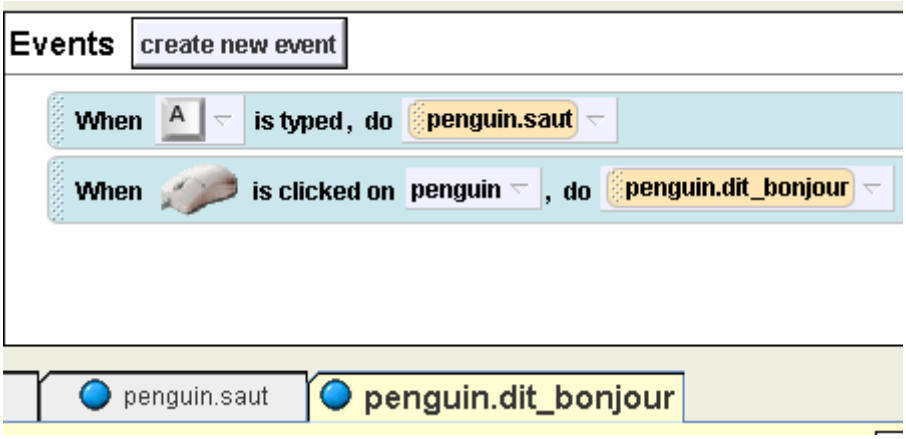
Bien évidemment, Alice possède un observateur d'événements. On "programme" l'observateur d'événements d'Alice grâce à la fenêtre "Events" (la fenêtre "Events" est un peu plus qu'un "simple" observateur d'événements, car c'est dans cette fenêtre que l'on trouve le fameux "When the world starts, do world.my first method"

Dans la fenêtre vous allez pouvoir choisir les actions à entreprendre (souvent l'appel d'une méthode) en cas :

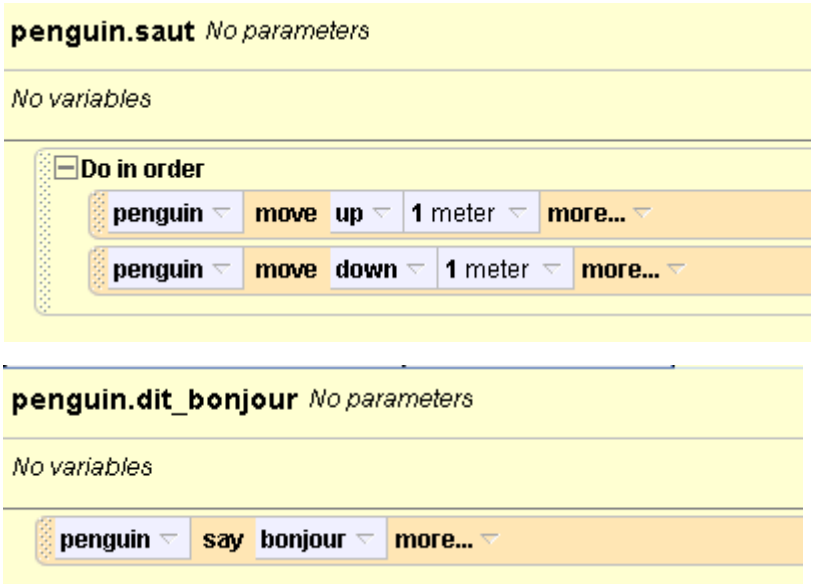
- d'appui sur une touche du clavier
- d'action avec la souris (clique sur un objet, déplacement,.....)

Prenons un exemple très simple :
Nous allons créer 2 événements : "cliquer sur A" et "clic de souris sur l'instance penguin".
L'événement "cliquer sur A" appellera une méthode "penguin.saut" et l'événement "clic de souris sur l'instance penguin " appellera la méthode "penguin.dit_bonjour".

Pour créer un événement, il faut cliquer sur....."create new event", ensuite, il ne reste plus qu'à compléter :

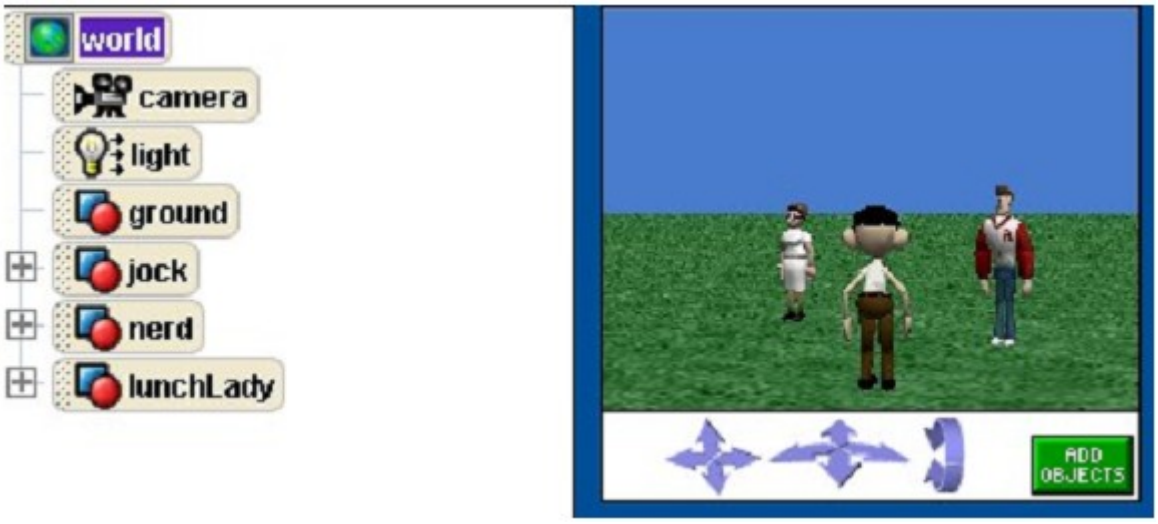


Pour obtenir ceci, il faut avoir au préalable créé les Class-level method saut et dit_bonjour.
Il nous reste justement à compléter ces 2 méthodes :



Et voilà, enfin un programme avec de l'I-N-T-E-R-A-C-T-I-V-I-T-E !!

Méthodes, paramètres, Events et if/else
L'étude de l'exemple suivant va nous permettre d'utiliser toutes les notions que nous venons d'étudier.
Créons une scène avec une instance de la classe Nerd, une instance de la classe Jock et une instance de la classe Lunchlady

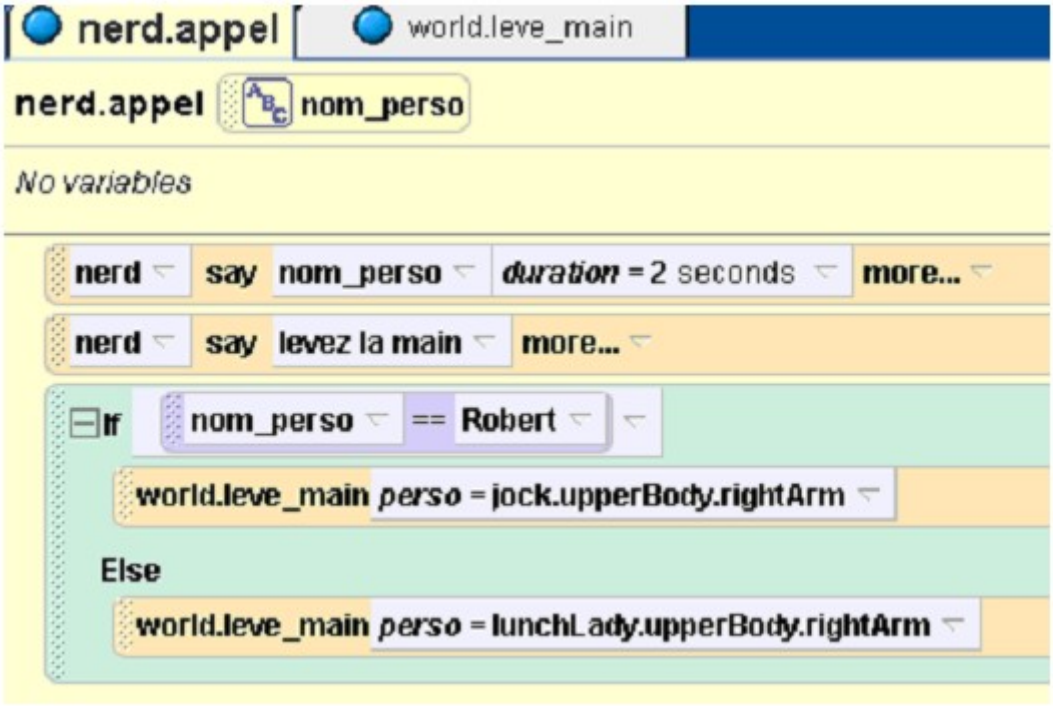


L'idée est très simple : si l'utilisateur clique sur lunchLady (que nous appellerons par la suite Gertrude), nerd devra dire : « Gertrude », une pause, « levez la main ». Gertrude devra alors lever puis baisser la main.
Si l'utilisateur clique sur jock (que nous appellerons par la suite Robert), nerd devra dire : « Robert », une pause, « levez la main ». Robert devra alors lever puis baisser la main.
Tout cela semble relativement simple à réaliser, mais nous allons essayer d'être « le plus propre possible », notamment en utilisant au maximum les paramètres des méthodes.

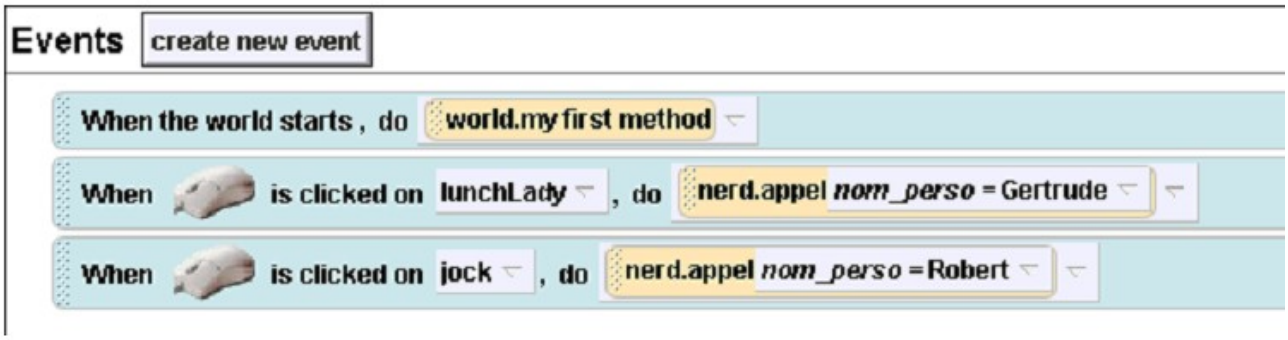
Nous allons écrire uniquement 2 méthodes :

- appel, qui sera une class level method de la classe Nerd
- leve_main qui sera une world level method

Voici la méthode appel :



Vous remarquerez la présence d'un paramètre de type String, nom_perso. Ce paramètre est utilisé dès la première ligne de la méthode.
Cette méthode est appelée par l'observateur d'événements « Events »



En cas de clic sur lunchLady, la méthode nerd.appel est appelée avec le paramètre nom_perso égal à « Gertrude ».
En cas de clic sur jock, la méthode nerd.appel est appelée avec le paramètre nom_perso égal à « Robert».

Voici maintenant la méthode leve_main :

Cette méthode possède un paramètre perso, de type « Object »
L'appel de cette méthode (et le choix de la valeur à attribuer au paramètre perso) se trouve au niveau du if de la méthode nerd.appel.

Exercices chapitre V

Exercice 5.1

Créer une scène avec un scientifique fou, une table (Furniture), un mixeur (Kitchen/Blender) et une tasse (Kitchen/Mug). Placer le scientifique fou derrière la table, le mixeur et la tasse sur la table.
Écrire un programme permettant qu'un clic sur le mixeur entraîne les événements suivants :

- le scientifique fou se tourne vers le mixeur
- le bras du scientifique se tend vers le mixeur
- le mixeur se met à tourner sur lui même.

De plus, un clic sur la tasse entraîne :

- le scientifique fou se tourne vers la tasse
- le bras du scientifique se tend vers la tasse
- le liquide présent dans la tasse disparaît

Vous devrez écrire le moins de codes possible (faites preuves d'astuces lors de l'écriture des méthodes), n'hésitez pas à utiliser des paramètres.

Exercice 5.2

En "hommage" à Lewis Carroll (auteur d'Alice aux pays des merveilles), vous allez créer une scène avec un arbre, un chat (cheshireCat) et un interrupteur (Controls/TwoButtonSwitch).
Le chat, qui est perché dans l'arbre, devient invisible (à part son sourire) en cas d'appui sur le bouton rouge et doit redevenir visible après l'utilisation du bouton vert. Vous devez écrire une seule méthode.

Exercice 5.3

Créer une scène avec un bonhomme de neige et 4 sphères (shapes/sphere) de 4 couleurs différentes (rouge, vert, bleu et jaune). Écrire un programme permettant au bonhomme de neige de prendre la couleur de la sphère choisie (on choisira la sphère en cliquant dessus). Un clic sur le bonhomme de neige doit le faire redevenir blanc. A chaque changement de couleur, le bonhomme de neige doit "dire" sa couleur.

Exercice 5.4
Créer une scène d'hiver avec un lac gelé, un joueur de hockey, une crosse de hockey, un palet de hockey et un but de hockey. Placer 3 sphères colorées (vert, rouge, jaune). Le joueur s'entraîne à frapper au but. L'utilisateur contrôle le déclenchement et la puissance du tir en cliquant sur une des sphères (vert : faible, rouge : moyen, jaune : fort). L'appui sur la barre d'espace doit ramener le palet à sa position d'origine.

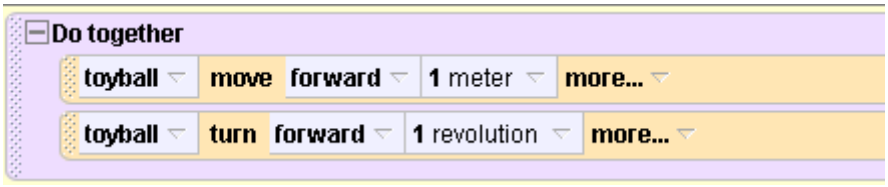
Chapitre VI
Retour sur les fonctions et le couple if/else

Nous avons à plusieurs reprises utilisé les fonctions proposées par Alice. Il est peut-être bon de rappeler qu'une fonction renvoie une valeur (au contraire d'une méthode). Il est bien évidemment possible de créer ses propres fonctions dans Alice et comme les méthodes, ces fonctions peuvent utiliser des paramètres.

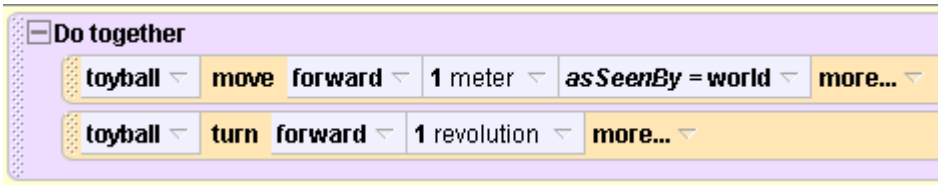
Les fonctions dans Alice ressemblent vraiment aux fonctions en mathématiques (y = 3x+2 si x = 3 alors y = 11, on a bien une valeur d'entrée (3), qui correspond à un paramètre dans Alice, et une valeur de sortie (11)).

Créer ses propres fonctions dans Alice

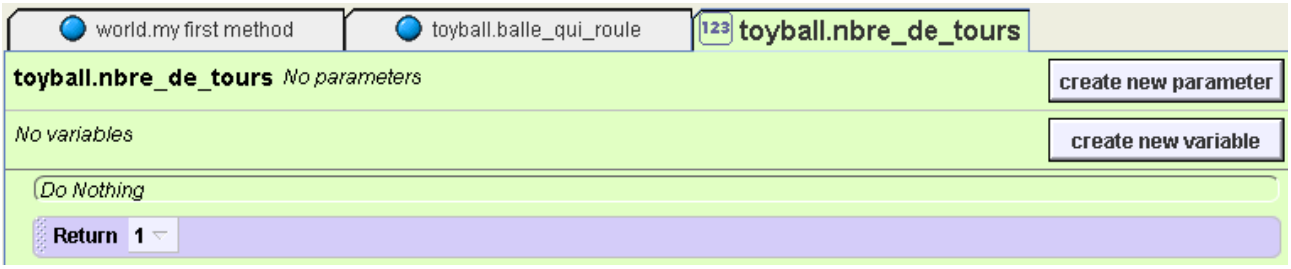
Nous allons chercher à simuler un ballon qui roule. La première idée qui nous vient est d'utiliser la méthode move et la méthode turn dans un do together :



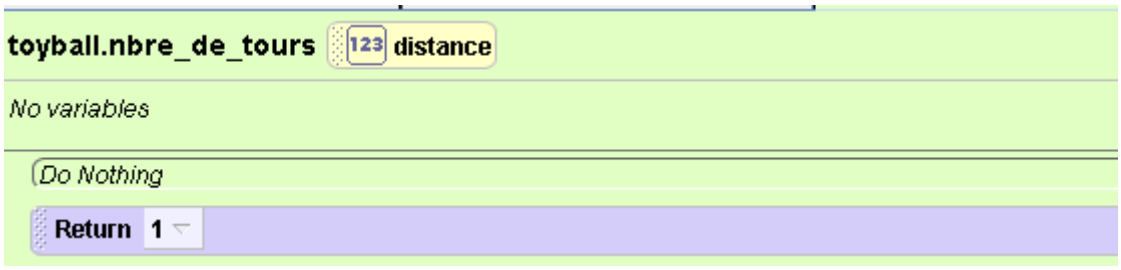
Mais le résultat n'est pas à la hauteur de nos espérances !
En effet, il ne faut pas oublier que le move se fait par rapport aux axes portés par le ballon, or, les axes du ballon tournent (méthode turn). Il nous faut donc utiliser AsSeenBy :



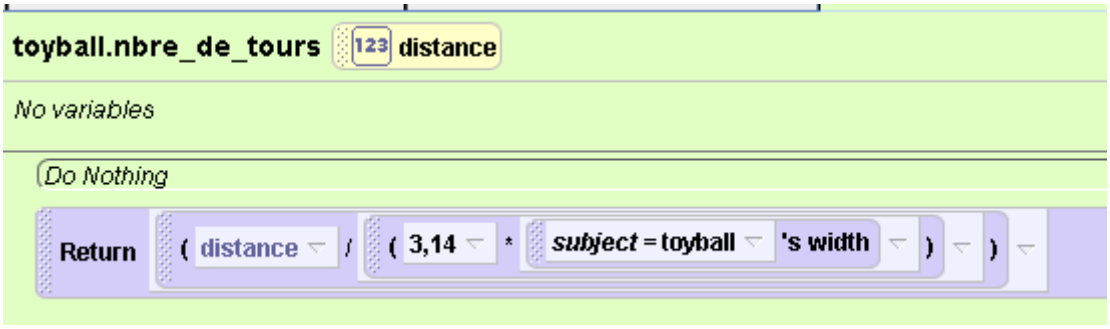
Le résultat est correct, mais si nous décidons de changer la distance parcourue par le ballon, nous devrons aussi modifier le nombre de tours effectués par le ballon : tout cela n'est pas très propre !
Faisons un peu de maths (ou de physique !) : le ballon ne glissant pas sur le sol, quelle est la distance parcourue par le ballon quand il effectue un tour ? Alors ?
La réponse est : la circonférence du ballon, c'est à dire $2\pi R$ (R étant le rayon du ballon).
Si le ballon parcourt une distance d, combien de tours doit-il effectuer ? (De plus en plus dur !).
Une petit produit en croix nous permet de trouver : nbre de tour = $d/2\pi R$.
Revenons à Alice, nous allons créer une fonction nbre_de_tours :
Cette fonction sera utilisée par la classe toyball, on sélectionne donc l'instance toyball, l'onglet functions et enfin on clique sur create new function.On obtient alors :



Remarquez que la fonction créée apparaît à côté des méthodes (nous avons créé une class-level-method balle_qui_roule), il faut aussi remarquer le Return à la fin de la fonction. Pour l'instant la fonction renvoie comme valeur 1, nous allons maintenant remplacer le 1. Mais avant ça, nous devons créer un paramètre distance :

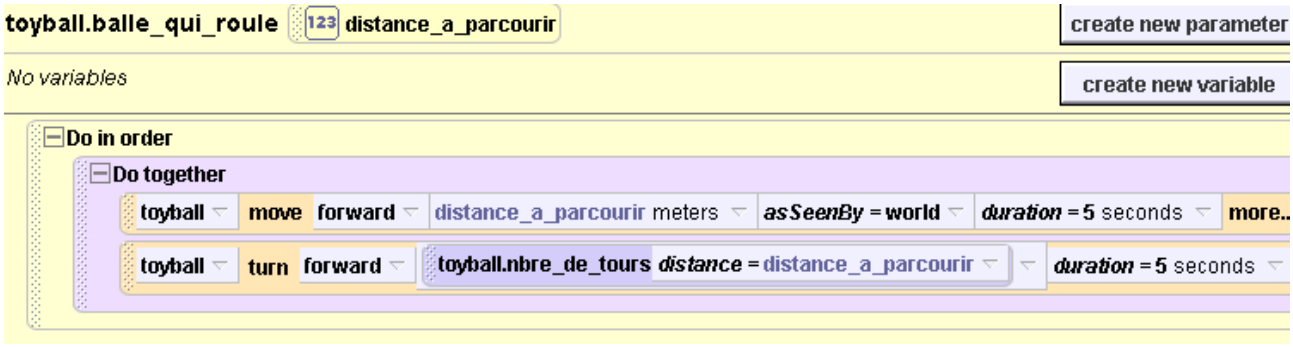


Nous allons maintenant remplacer le 1 par distance/(2x3,14x(largeur de la balle/2)) car largeur de la balle/2 est tout simplement le rayon de la balle. On a donc distance/(3,14xlargeur de la balle) :

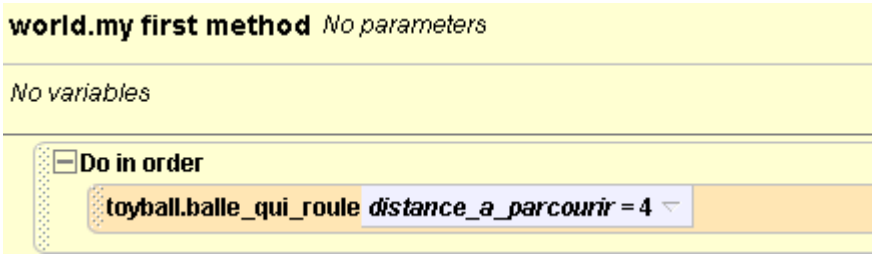


Remarquez que nous avons uniquement complété la partie return de la fonction (ce qui est renvoyé), nous aurions pu aussi écrire des instructions juste au-dessus (nous verrons des exemples plus tard).
Nous allons maintenant utiliser la fonction nbre_de_tours dans la class level method balle qui roule :

Nous pouvons même utiliser un paramètre pour la méthode balle_qui_roule qui nous permettra de choisir la distance à parcourir lors de l'appel de la méthode dans world.my first method :



Voilà l'appel de la fonction :



Voilà, cela fonctionne.

Retour sur le couple if/else

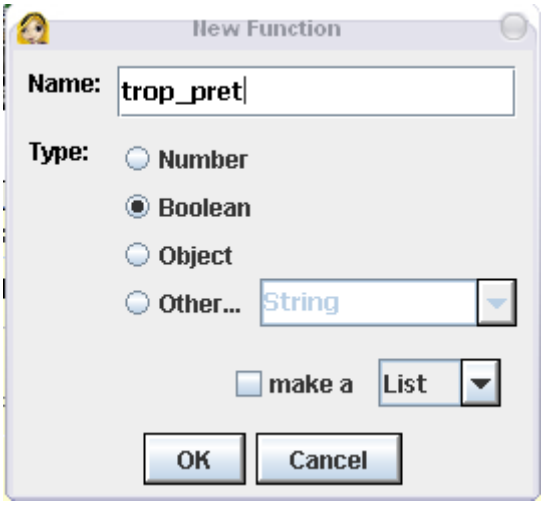
Nous allons créer une simulation de contrôle de trafic aérien :
Créons une scène avec un aéroport (éventuellement une tour de contrôle), un avion et un hélicoptère.

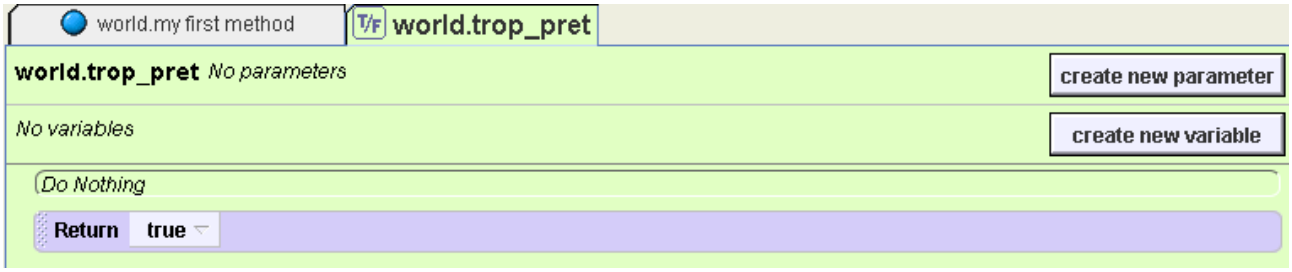
L'idée est de créer un programme permettant d'éviter les collisions entre l'hélicoptère et l'avion.
Si la différence d'altitude entre l'avion et l'hélicoptère est trop faible, le programme devra modifier l'altitude d'un dès deux engins. Voici l'idée :

Création d'une fonction "trop_pret" qui renvoie vrai si les 2 engins sont trop prêt et faux si la distance de sécurité est respectée. Une méthode "modif_alt" qui modifie l'altitude. La méthode world my first method appelle la méthode "modif_alt" (si besoin est !) puis fait avancer les 2 engins.

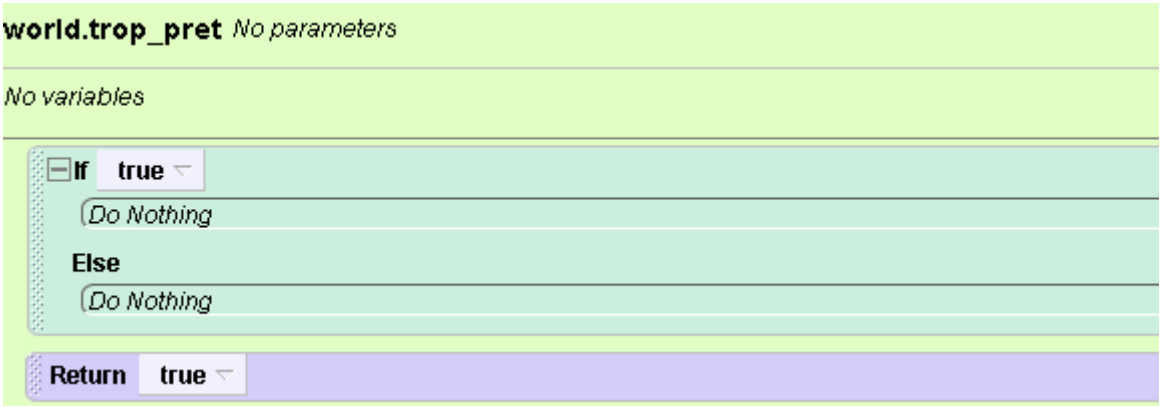
Commençons par l'écriture de la fonction "trop_pret" :

Notre fonction devra renvoyer un booléen

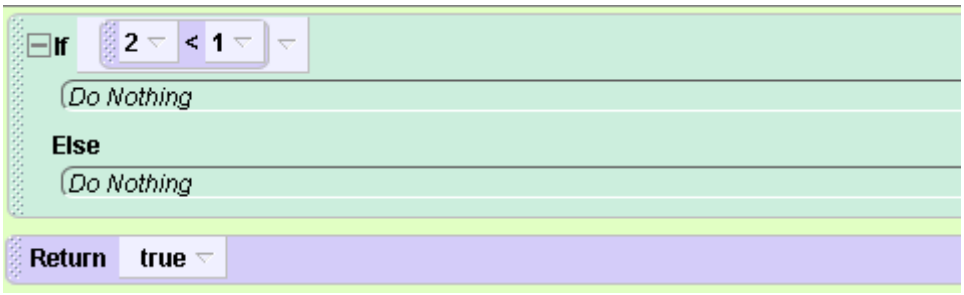




Si la différence d'altitude entre les 2 engins est inférieure à une certaine valeur, la fonction devra renvoyer "true" sinon elle renverra "false". Cette différence d'altitude sera donnée par la fonction appartenant à l'instance helicopter : "helicopter distance above". Nous devons respecter un certain ordre pour arriver au résultat attendu:

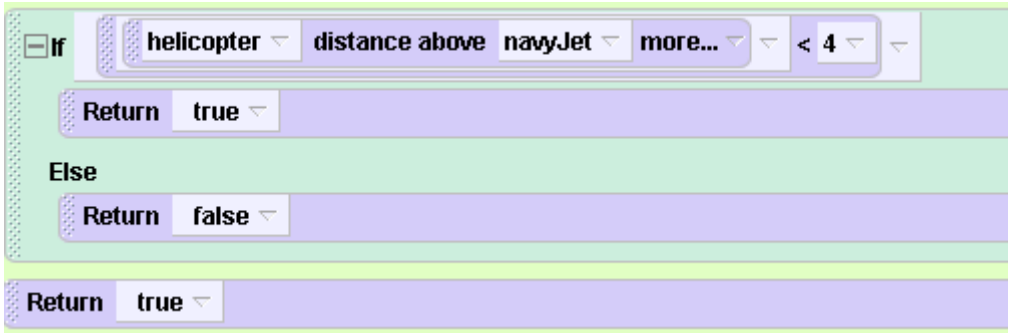


Utilisons la fonction a<b (appartient à world), les valeurs (ici 2 et 1 n'ont aucune importance)



Nous pouvons maintenant utiliser la fonction "helicopter distance above"

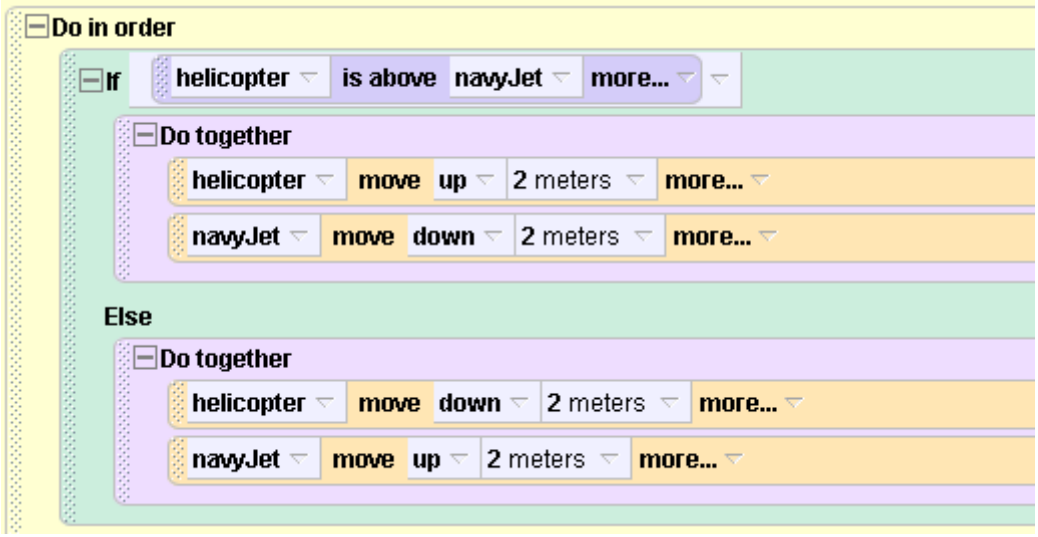
Il n'y a plus qu'à choisir la distance de sécurité (ici par exemple 4 m)



Le dernier Return ne peut pas être supprimé, mais cela n'a aucune importance, car il n'est jamais exécuté.

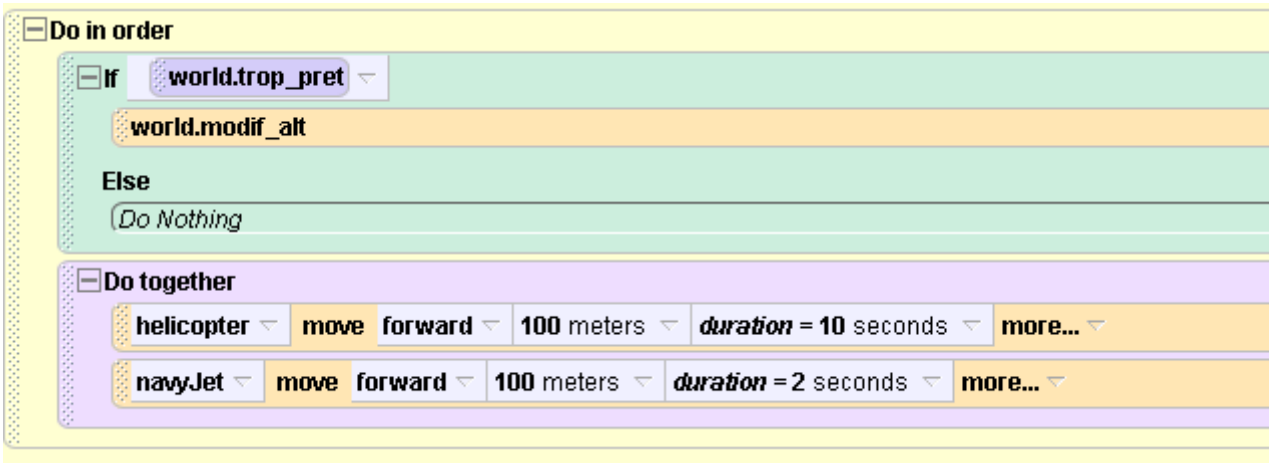
Passons maintenant à l'écriture de la méthode "modif_alt":
Créons une world level method

Nous devons faire des tests pour savoir qui est au-dessus et qui est en dessous :



Si l'hélicoptère est au-dessus, l'hélicoptère monte de 2 m et l'avion descend de 2 m sinon l'avion monte de 2 m et l'hélicoptère descend de 2 m.

Il nous reste à écrire la world.my first method :

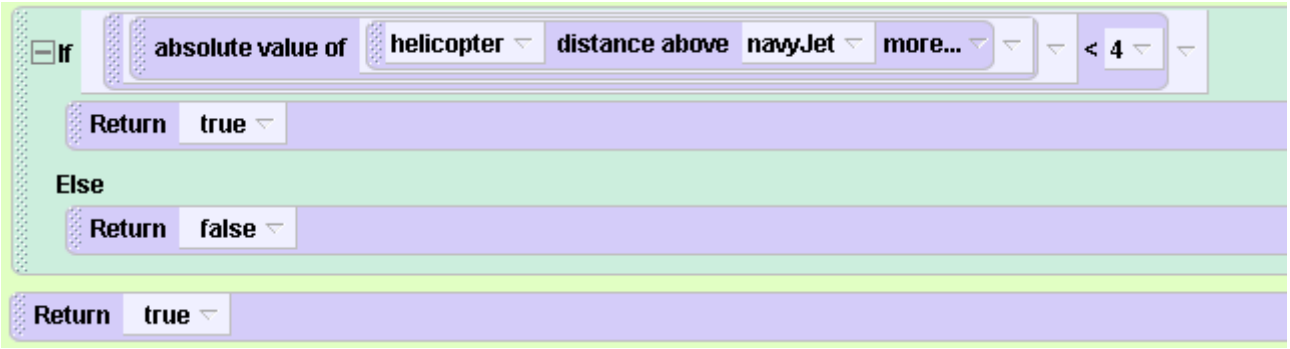


Visiblement le programme fonctionne très bien quand l'hélicoptère est au-dessus de l'avion. En revanche quand l'hélicoptère est en-dessous de l'avion, la fonction "modif_alt" est systématiquement appelée (quelque soit la différence d'altitude entre les 2 engins). Nous avons donc affaire à un bug. Essayons de comprendre pourquoi :

Le problème vient de la fonction "helicopter distance above navyjet" utilisée dans notre fonction "trop_pret". En effet, si l'hélicoptère est 10 m au-dessus de l'avion, la fonction "helicopter distance above navyjet" renverra la valeur 10.
En revanche si l'hélicoptère est 10 m en dessous de l'avion, la fonction "helicopter distance above navyjet" renverra la valeur – 10. Si vous réfléchissez un peu notre problème vient de là.
En effet - 10 est bien inférieur à 4, il est donc logique que notre fonction "trop_pret" retourne "true" alors que la différence d'altitude est supérieure au 4 m "réglementaire".
Pour que cela fonctionne, il faudrait que la fonction "helicopter distance above navyjet" renvoie 10 au lieu de -10, dit autrement, il faudrait supprimer le signe moins.

Les mathématiques vont venir à notre secours. Il existe en effet une fonction mathématique qui supprime justement les signes moins : la valeur absolue. Valeur absolue de -10 est égale à 10.

Il faut donc demander à Alice de prendre la valeur absolue de "helicopter distance above navyjet". Pour cela il suffit d'ajouter "absolute value of a" (fonction de world) devant "helicopter distance above navyjet".



Notre programme fonctionne maintenant correctement (il reste un petit bug dans le calcul de la différence d'altitude, parfois l'hélicoptère semble au-dessus, et pourtant, c'est lui qui descend, mais là, on ne peut pas y faire grand-chose, c'est un bug dans Alice !)

Voilà, maintenant, on peut éventuellement mettre des paramètres (de type objet) à la place de navyJet et d'helicopter, on peut aussi remplacer la distance de sécurité par un paramètre de type nombre, mais bon, tout cela, on l'a déjà vu auparavant et je vous laisse vous débrouiller tout seul.

Exercices chapitre VI

Exercice 6.1

Créer un monde avec une voiture et une route. Vous devez écrire un programme permettant à la voiture d'avancer de façon réaliste sur la route (en faisant tourner les roues correctement).

Exercice 6.2

Créer un monde avec un pingouin. Ecrire une fonction permettant au pingouin de se déplacer (en glissant) aléatoirement. Pour cela, il faudra écrire une méthode "deplacement". A chaque appel de cette méthode le pingouin devra se déplacer (ce déplacement ne devra pas se limiter à un simple avant-arrière ou un simple gauche-droite, il devra être plus complexe). Pour vous aider, je vous signale qu'un move foreward avec un nombre négatif comme argument entraine un mouvement vers l'arrière (même principe pour les mouvements gauche-droite).

Exercice 6.3

Créer un monde avec une mare ronde (choisir "Class circle" dans "Shapes", puis colorer le cercle en bleu). Placer une abeille (Class Bee, ne pas hésiter à la grossir) au-dessus de la mare. Ecrire une méthode permettant à l'abeille de décrire une trajectoire circulaire au dessus de la mare (interdiction d'utiliser "asSeenBy"), la circonférence du cercle décrit par l'abeille devra être identique à la circonférence de la mare.

Exercice 6.4

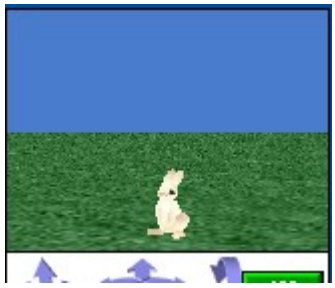
Nous allons créer un programme permettant de travailler son vocabulaire en Anglais. Créer un monde avec un chat (Class Sitting_cat). Placer devant le chat 3 mots en 3D (les mots en 3D sont des objects comme les autres) : cat, dog et bird. Au cas où l'élève clique sur la bonne réponse (le bon mot en 3D), le chat devra dire :"Bravo tu es le meilleur". Si l'élève clique sur n'importe quoi d'autres, le chat devra hocher la tête et dire "Non, essaye encore une fois". Pour arriver à vos fins, vous devrez créer une fonction "cestbon" qui renverra true en cas de bonne réponse et false en cas de mauvaise.

Chapitre VII
L'instruction while

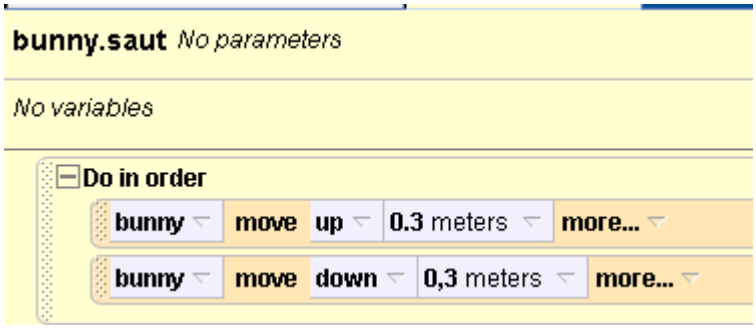
Retour sur l'instruction loop

Nous avons déjà entre-aperçu l'instruction loop dans le chapitre 3, commençons donc par quelques rappels :
Si vous avez besoin de répéter plusieurs fois la même instruction (ou le même groupe d'instructions), vous devez utiliser la fonction loop.

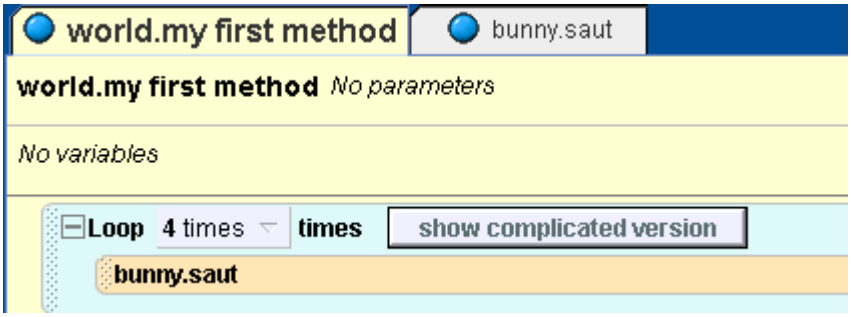
Créons un monde avec un lapin.



Programmons une méthode (très simple) pour faire sauter le lapin :



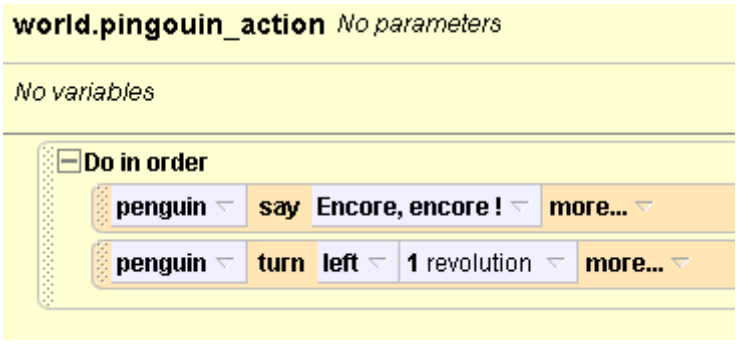
Pour faire sauter le lapin plusieurs fois, il suffit d'utiliser loop :



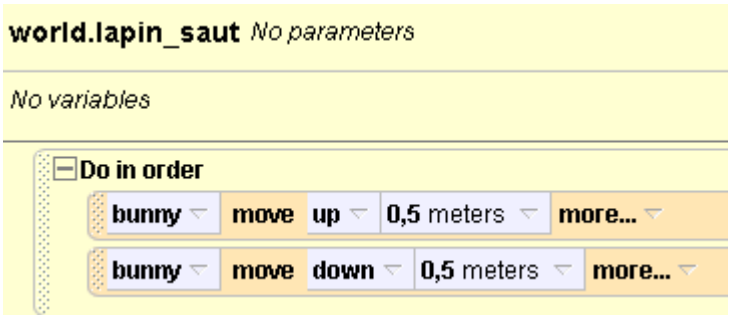
le lapin sautera 4 fois...

Nous pouvons aussi placer une boucle (loop) dans une boucle :
Rajoutons un pingouin à notre monde et écrivons un programme pour que le lapin accomplisse 3 sauts et qu'après chaque série de 3 sauts, le pingouin dise : « Encore, encore !» et effectue ensuite un tour sur lui même.

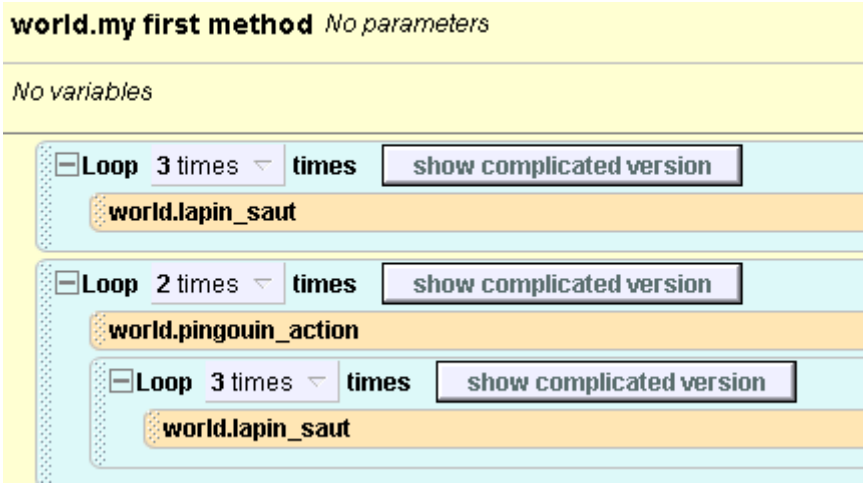
Créons méthode « pingouin_action» (très très simple) :



Créons la méthode « lapin_saut » (toujours très simple) :



et maintenant les appels des méthodes :

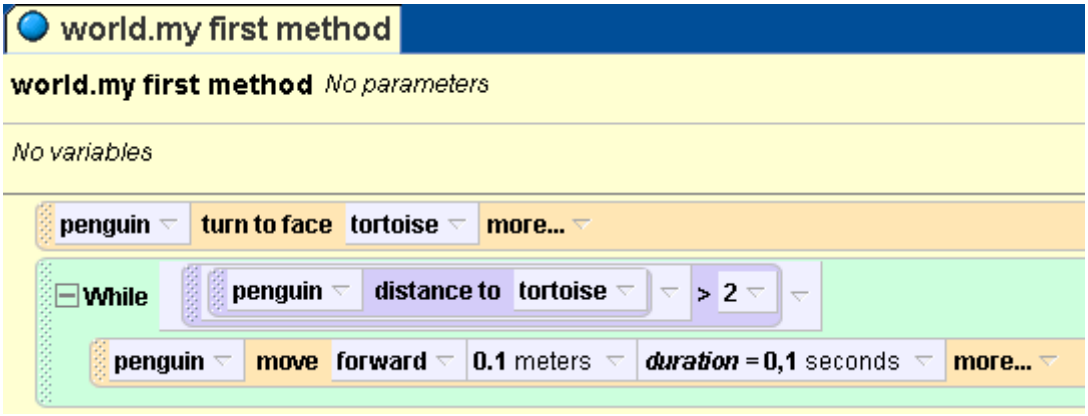


Pourquoi le premier « Loop 3 times » est « en dehors » du Loop global (2 times) ?

Noter aussi que « 3 times » peut-être remplacé par une fonction qui renverra le nombre de fois que la boucle devra s'exécuter :

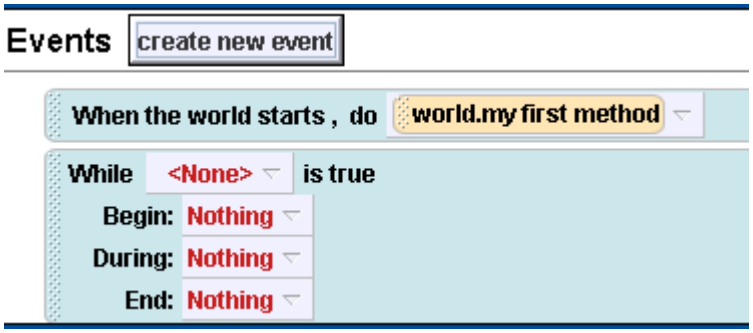
Un loop un peu particulier : l'instruction while

En anglais while veut dire « tant que ». While est une boucle, mais une boucle un peu spéciale : while est toujours associé à une fonction qui renvoie un booléen. Tant que cette fonction renverra « vrai », les instructions présentes dans le while seront exécutées.
Créons un monde avec un pingouin et une tortue. Nous pouvons alors écrire le programme suivant :



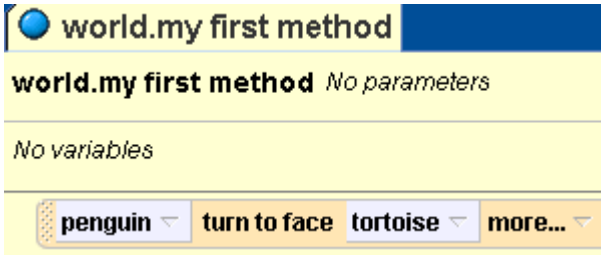
Attention, une erreur de logique peut faire « tomber » le programme dans une boucle infinie (une fonction qui renvoie toujours vrai).

While peut aussi être utilisé dans le gestionnaire d'évènements avec par exemple « while something is true » :
Appuyer sur « create new event » puis sélectionner «while something is true » :

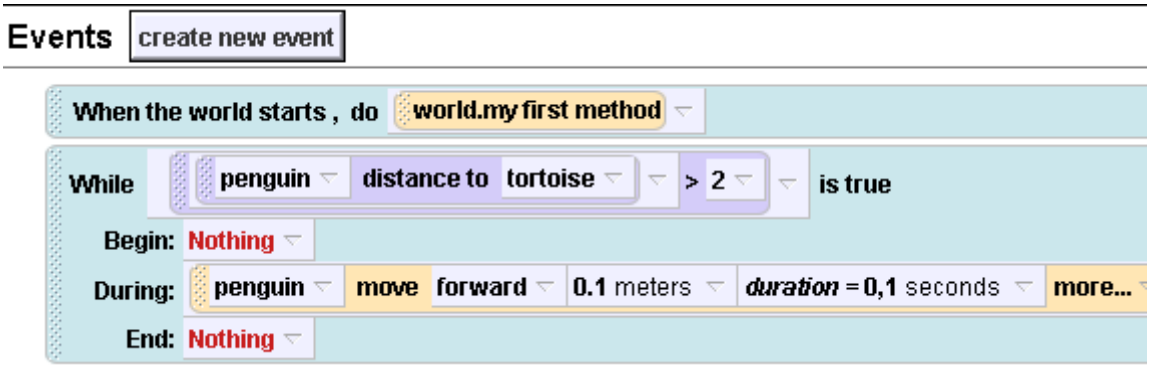


Je reviendrais plus tard sur Begin, During et End. Pour l'instant nous allons réécrire le programme du pingouin et de la tortue en utilisant le gestionnaire d'événements :

world.myfirstmethod sera uniquement utilisé pour tourner le pingouin vers la tortue

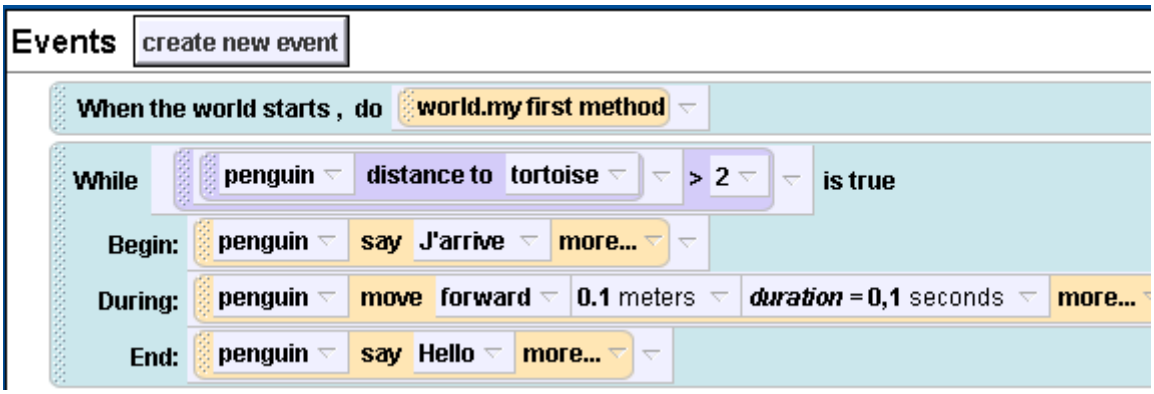


Pour le reste, en utilisera le gestionnaire d'évènements



Voilà, le résultat est presque le même ! A vous de trouver la différence.

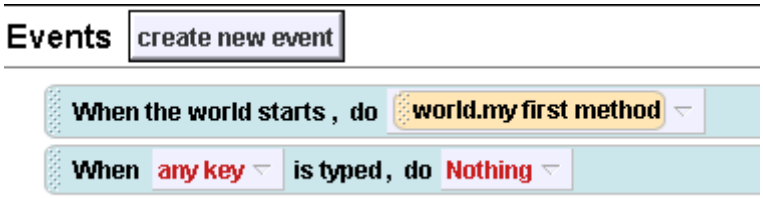
Revenons maintenant sur Begin et End :
Quand la fonction du while devient « vraie » avant d'exécuter ou d'appeler une méthode (avec « During »), il est possible d'appeler (d'exécuter) une autre méthode. De la même manière pour « terminer » le while (la fonction du while vient de devenir fausse), nous pouvons appeler une autre méthode grâce à « End ». Un petit exemple :



Je pense que l'exemple parle de lui même, analysez-le !

NB : Il sera dans la plupart des cas beaucoup plus « propre » de créer des méthodes (ici c'est inutile, le programme est trop simple) et de les appeler avec Begin, During et End.

Autre exemple qui peut s'avérer très utile (notamment pour faire des jeux !) : « while any key is pressed » (tant qu'une touche est pressée)
Pour « accéder » à cette instruction, il faut d'abord choisir « when any key is typed..... »



Il faut ensuite faire clic droit sur « When any key..... », choisir « change to » et sélectionner « While any key is …...

Vous pouvez alors compléter « Begin », « During » et « End » comme précédemment

Voici un petit exemple tout simple

Exercice chapitre VII

Exercice 7.1

Ecrire un programme permettant de faire "tomber" le brouillard sur une scène par un simple appui sur la touche espace (le brouillard se lève dès que la touche espace est relâchée). Un personnage (de votre choix) devra annoncer l'arrivée du brouillard (juste avant que le brouillard "tombe").

Exercice 7.2

Créer une scène avec un bonhomme de neige et une bonne femme de neige. L'utilisateur doit pouvoir contrôler le bonhomme de neige (avant, arrière, tourner à gauche, tourner à droite). Le bonhomme de neige est extrêmement timide, il devient rouge quand il s'approche un peu trop près de la bonne femme de neige et redevient blanc quand il s'en éloigne.

Chapitre VIII
Les variables

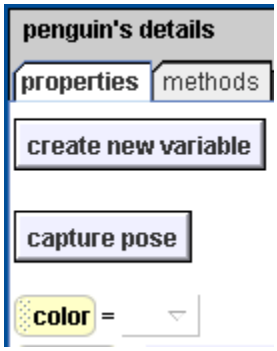
Dans ce dernier chapitre, nous allons aborder le sujet des variables. La notion de variable est très importante en programmation, voilà pourquoi, il me parait important de préciser quelques notions généralistes avant d'entrer dans le vif du sujet :

Un programme « passe son temps » à traiter des données. Pour pouvoir traiter ces données, l'ordinateur doit les ranger dans sa mémoire (la RAM).
La RAM se compose de cases dans lesquelles nous allons ranger ces données (une donnée dans une case). Chaque case a une adresse (ce qui permet au processeur de savoir où sont rangées les données).
Alors, qu'est-ce qu'une variable ?
Eh bien c'est une petite information (une donnée) temporaire que l'on stocke dans une case de la RAM. On dit qu'elle est "variable" car c'est une valeur qui peut changer pendant le déroulement du programme.
Une variable est constituée de 2 choses :
– Elle a une valeur : c'est la donnée qu'elle stocke (par exemple le nombre 5 ou la suite de caractères (chaine de caractères) « bonjour »)
– Elle a un nom : c'est ce qui permet de la reconnaître.

Les variables dans Alice

Sans le savoir, vous avez déjà à de nombreuses reprises utilisés des variables dans Alice. Avez-vous déjà utilisé l'onglet "Properties" ? Oui, donc vous avez déjà utilisé des variables !
En POO chaque instance va posséder des méthodes (vu et revu.....) et des variables (que l'on appelle aussi des attributs, mais par souci de simplicité, je n'utiliserai que le terme de variable). Par exemple, dans Alice, le pingouin possède la méthode wing_flap

et la variable "color"

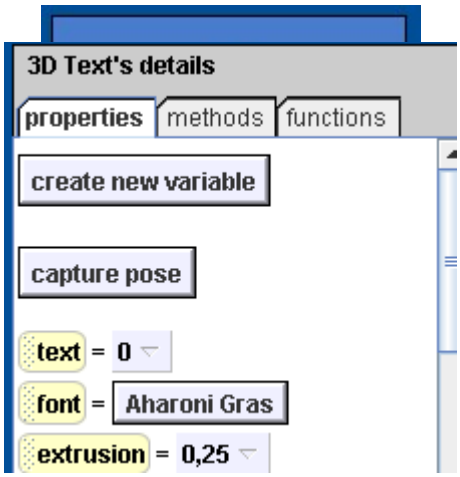


Comme déjà dit plus haut: instance d'un objet = des variables + des méthodes (ou fonctions) dans Alice, mais cette idée est vraie en POO plus généralement.
Enfin, il faut savoir que la valeur d'une variable peut être modifiée en utilisant une méthode (vous avez déjà eu à faire ce genre de choses à de nombreuses reprises).

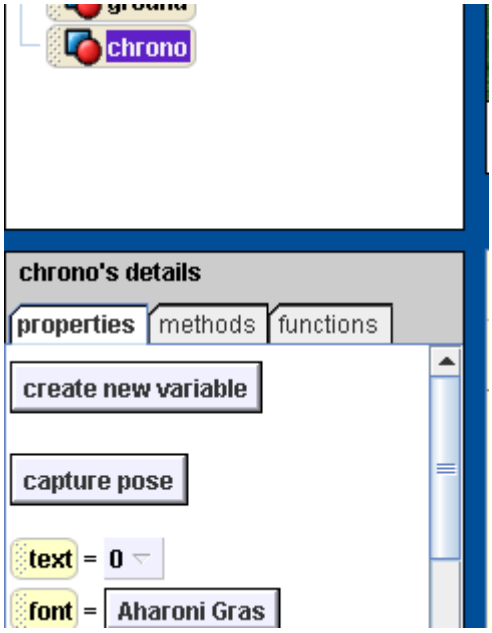
Créer ses propres variables d'instance

Création d'un chrono

Chaque instance possède des variables, mais vous pouvez aussi créer les vôtres. Nous allons prendre un exemple : la création d'un chrono.
Nous allons utiliser un texte en 3D, ce texte est une instance (un objet) comme un autre il possède donc des méthodes et des variables (de toute façon, en POO, tout est objet !
Au départ la variable "text" de "3D text" est égale à zéro.

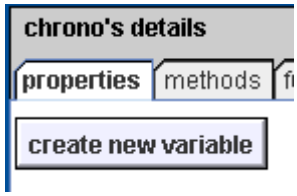


Puisque nous voulons créer un chrono, nous allons renommer l'instance 3D text en "chrono" (nous aurions très bien pu laisser 3D Text, mais bon, je trouve cela plus logique de changer le nom)

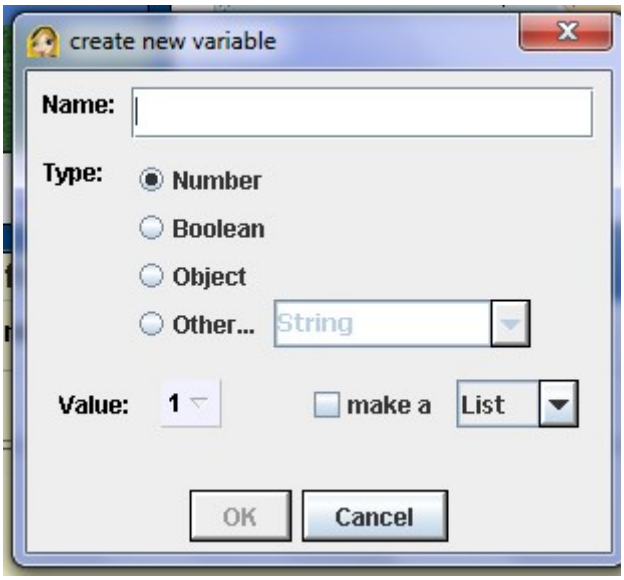


Il est maintenant temps de créer une nouvelle variable pour chrono : "temps"

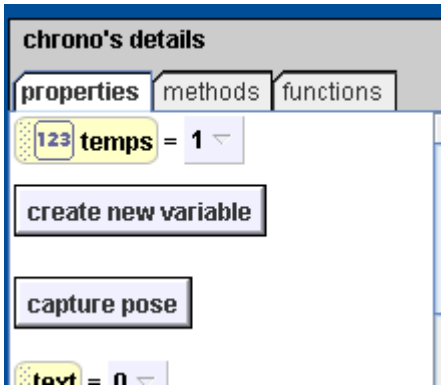
Appuyer sur "create new varialbe"



et entrer le nom de la nouvelle variable (temps) dans le champ prévu à cet effet (Name). On laissera "Number" sélectionné.

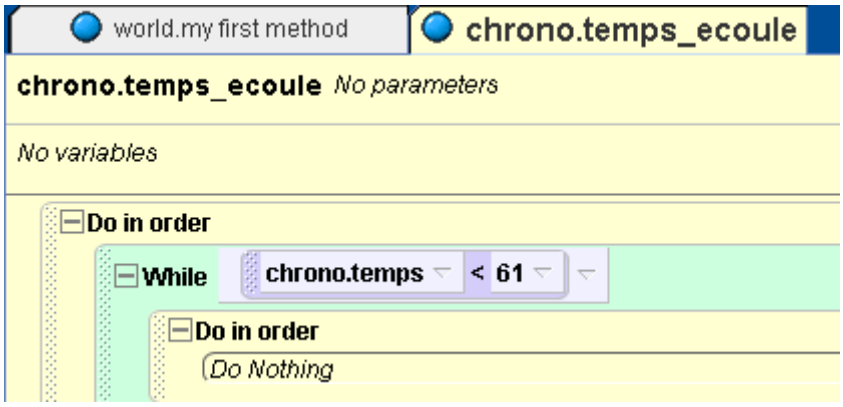


Voilà nous avons créé une nouvelle variable de type nombre :



Vous remarquerez que par défaut temps = 1

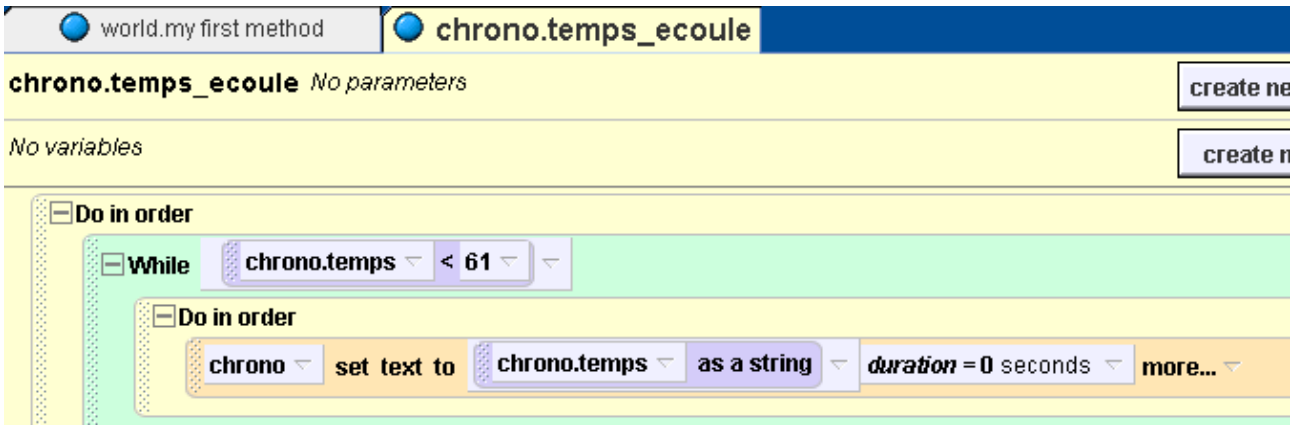
Nous allons maintenant créer une nouvelle méthode (une classe level method) pour "chrono", on l'appellera "temps_ecoule".
Nous allons utiliser un while dans notre méthode : "tant que "temps" est inférieur à 61" suivit d'un "do in order"



Pourquoi chrono.temps < 61 ? Tout simplement parce que notre chrono devra fonctionner pendant 1 minute (alors pourquoi pas 60 ? Réfléchissez bien à la question, vous allez trouver vous même la réponse !!). Par la suite, nous remplacerons 61 par un paramètre. Par ailleurs vous remarquerez que comme temps est une variable de l'instance chrono on écrira chrono.temps.

Qu'allons nous mettre dans le "Do in order" ?

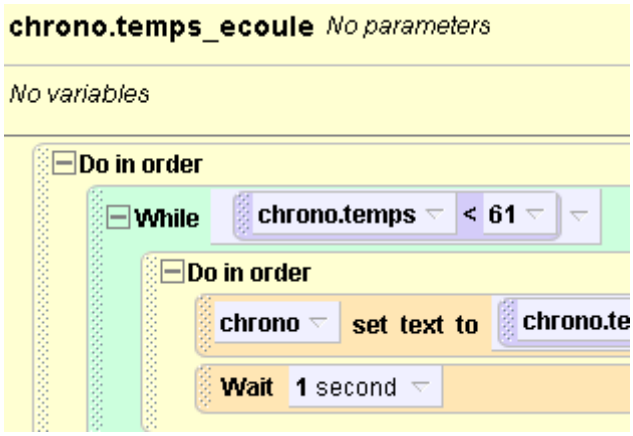
Première chose, notre variable temps est de type "number" alors que la variable "text" est de type "string" (chaine de caractère). Or, la première chose que nous voulons faire c'est d'utiliser la valeur contenue dans la variable "temps" et de l'afficher à l'écran grâce à la variable "text", il faut donc, transformer le nombre contenu dans la variable "temps" en chaine de caractère, puis d'associer cette chaine de caractère à la variable "text".
Tout ceci a l'air compliqué, mais dans Alice cela se fait en une seule ligne :



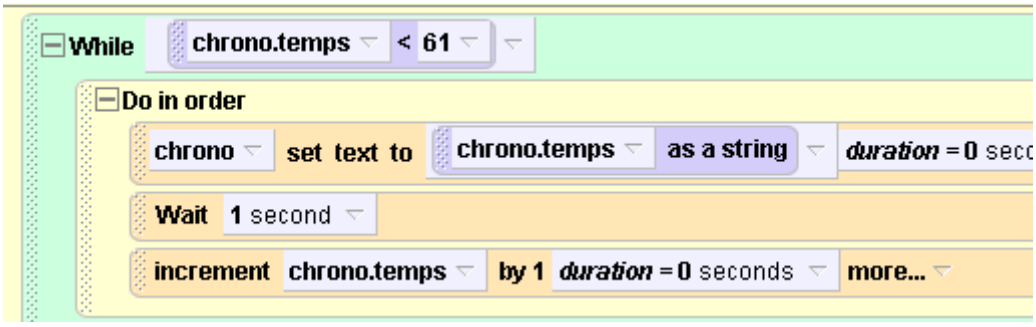
Pour avoir cette ligne, il faut :

"Enmener" la variable "text" dans le "Do in order", choisir "default string" dans le menu.
"Déposer" la fonction "what a string" (vous la trouverez dans l'onglet "function" de "world") à la place de "default string" choisir "chrono" puis "chrono.temps" dans le menu déroulant. N'oubliez pas le "duration=0".

Ensuite nous allons ajouter un "wait 1 second"



Enfin, nous allons terminer notre boucle en augmentant la valeur de l'attribut "temps" d'une unité.



Pour obtenir cette dernière ligne, il faut "emmener" l'attribut "temps" dans le "Do in order" et choisir "increment chrono.temps by 1" dans le menu déroulant. Là aussi, ne pas oublier le "duration = 0"

Je pense que tout cela est limpide et ne demande aucune explication supplémentaire !!

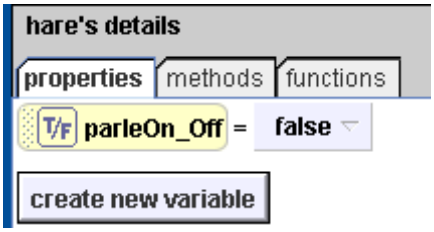
Il ne reste plus qu'à appeler la méthode "temps_ecoule" depuis la méthode "world.my first method"

Je vous laisse le soin de modifier la méthode "temps_ecoule" pour que la durée chronométrée soit définie dans un paramètre au moment de l'appel de la méthode.

Utilisation des booléens

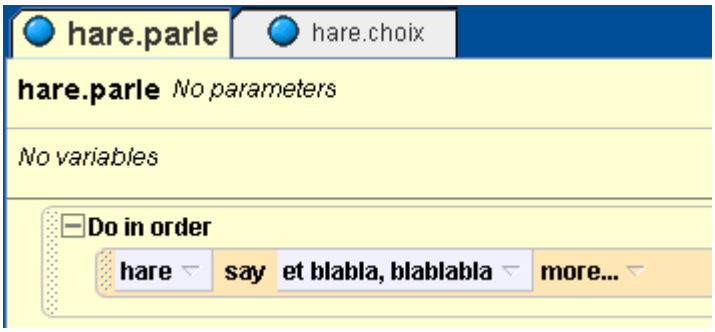
Un petit exemple pour montrer l'utilité des variables de type booléens :
Créons un monde avec un lièvre. L'idée de base est très simple : en appuyant sur la barre espace le lièvre doit se mettre à parler (blablabla), si vous appuyez de nouveau sur la barre espace, le lièvre devra arrêter de parler.
Le programme n'est pas si simple à écrire, car l'appui sur la barre espace devra déclencher ou arrêter les paroles. Alors, comment faire ?
Nous allons créer une variable de type booléen qui sera vraie si le lièvre est déjà en train de parler et fausse si le lièvre est silencieux.

Nous allons créer une variable de l'objet "Hare" de type booléen : parle On_Off (qui sera false au départ)



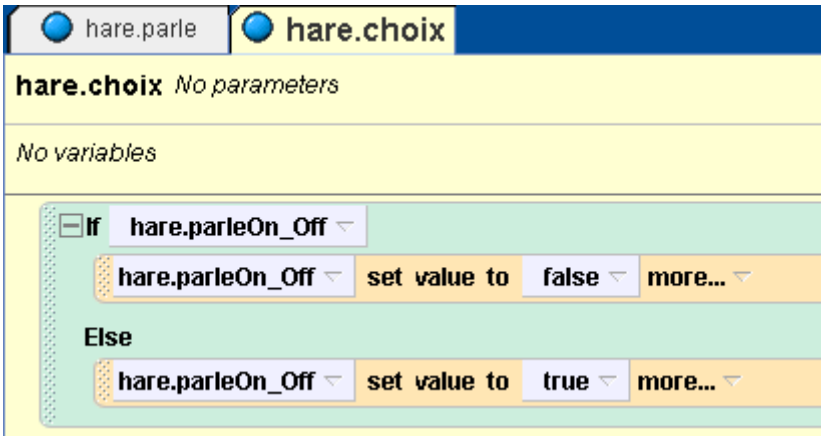
Cette variable sera utilisée dans "Events" :

Tant que la variable parle On_Off est false, la méthode parle n'est pas appelée, tant que parle On_Off est true la méthode parle est appelée en permanence (utilisation du while).
Voici d'ailleurs la méthode parle :



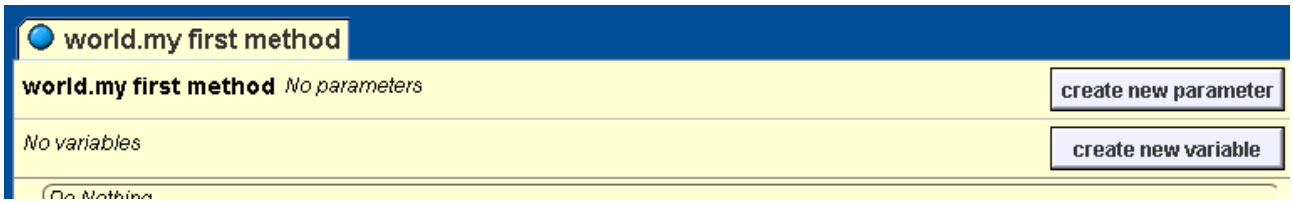
On peut difficilement faire plus simple.

Il nous reste maintenant à étudier la méthode choix. C'est cette méthode qui fera basculer l'attribut parle On_Off de false à true et vice versa. C'est la méthode la plus complexe, mais aussi la plus intéressante :

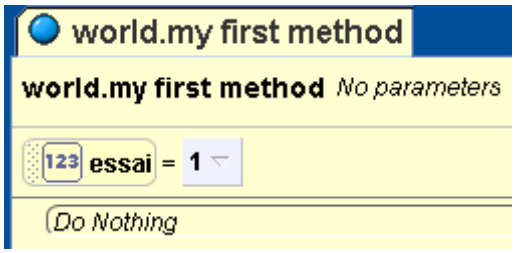


Les variables dans les méthodes ou les fonctions : les variables locales

Pour l'instant nos variables sont associées à une instance de classe. Il existe dans Alice d'autres types de variables : les variables locales.
Les variables sont associées à des méthodes, créons une variable locale dans la méthode "world my first method" en cliquant sur "create new variable" :

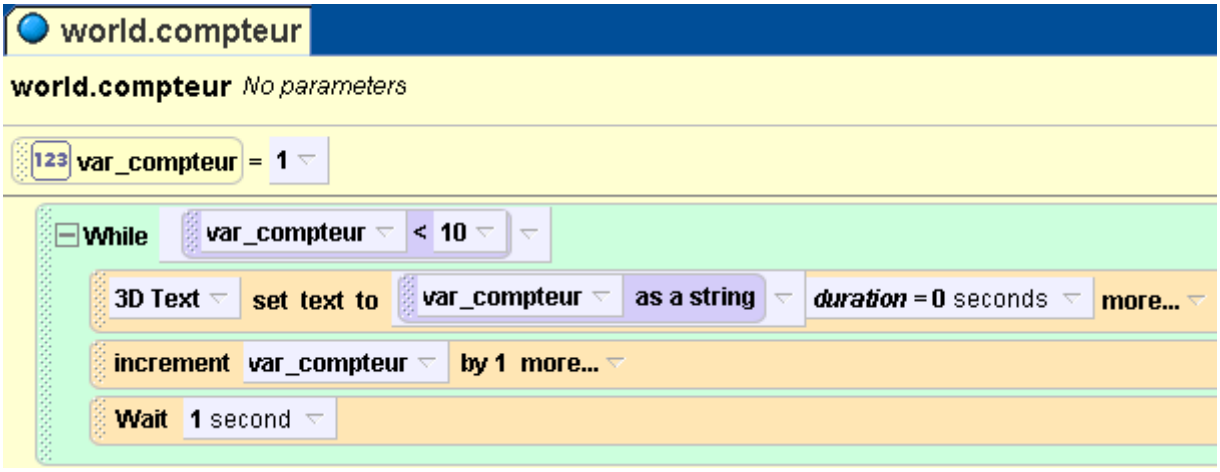


Nous appellerons cette variable : essai



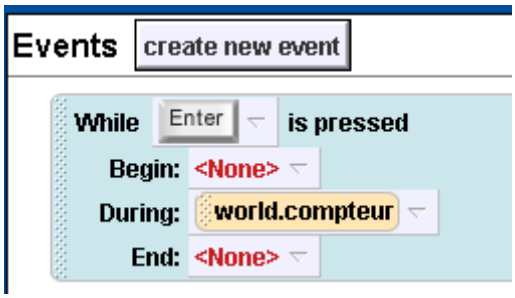
Comme vous pouvez le constater, notre variable est de type "number", mais elle aurait pu être de type "Boolean", "Object" ou autre.
Les variables locales s'utilisent de la même façon que les variables d'instance, mais, chose très importante, elles n'existent que dans la méthode où elles ont été créé (d'où leur nom de locale !).
A partir du moment où vous "quittez" une méthode, la variable locale est "détruite".

Ecrivons un petit programme : Créons une méthode "compteur"



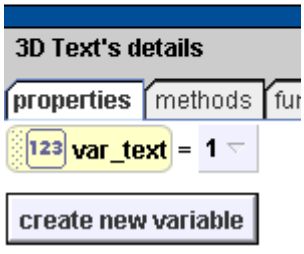
Nous avons créé une variable locale "var_compteur", sinon le programme ne pose pas de problème.

Cette méthode est appelée dans le gestionnaire d'évènements avec un while :



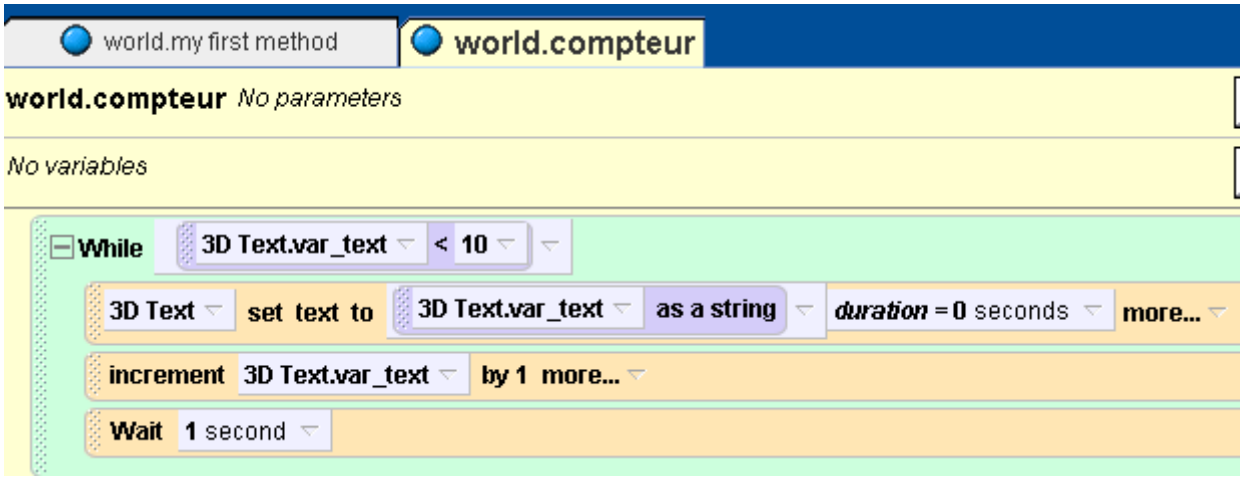
Tant que la touche "entrée" reste appuyée, nous "restons" dans la méthode "compteur". Si nous relâchons la touche "entrée", nous sortons de la méthode "compteur", la variable "var_compteur" est "détruite". Si nous appuyons de nouveau sur la touche "entrée" la variable "var_compteur" est re-créé avec la valeur de départ, pratiquement, le compteur est remis à zéro.

Ecrivons le même programme avec une variable d'instance à la place de la variable locale :



Nous avons créé une variable d'instance : "var_text"

Le reste du programme n'est quasiment pas modifié



Pourtant cela change tout : la variable d'instance "var_text" n'est pas une variable locale, elle n'est donc pas "détruite" en cas de "sortie" de la méthode "compteur", le compteur ne redémarre donc pas à zéro.

Exercices chapitre VIII

Exercice 8.1

Créer une scène avec un lapin et un pingouin. Un clic sur le lapin fait augmenter le score du lapin d'une unité (le score sera un texte 3D). Même chose pour le pingouin (cet exercice pourra par la suite vous servir pour développer des jeux plus compexes)

Exercice 8.2
Créer une scène avec un manège (Amusement Park/Carousel) et un bouton (Controls/Button). Au départ le manège est arrêté. Une pression sur le bouton devra mettre le manège en rotation, une nouvelle pression sur ce même bouton devra l'arrêter. L'animation du démarrage et de l'arrêt du manège devra être réaliste.

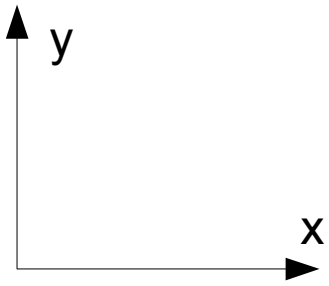
Exercice 8.3
Vous allez créer un "jeu" pédagogique destiné aux enfants de l'école primaire :
L'enfant verra apparaître sur l'écran un animal (un chameau, une vache ou un pingouin), au-dessus de l'animal, l'enfant aura 3 choix possibles (texte en 3D) : "Camel", "cow" ou "penguin"

Si l'enfant clique sur le bon mot (celui qui correspond à l'animal affiché), on verra apparaître "Bravo" et un nouvel animal s'affichera à la place du précédent. Dans le cas contraire on verra apparaître "Faux essaye encore une fois".
Attention le choix de l'animal à afficher (vache, chameau ou pingouin) devra être aléatoire (géré par le hasard)
Cet exercice est vraiment complexe (malgré les apparences) soyez donc très méthodique dans votre démarche.

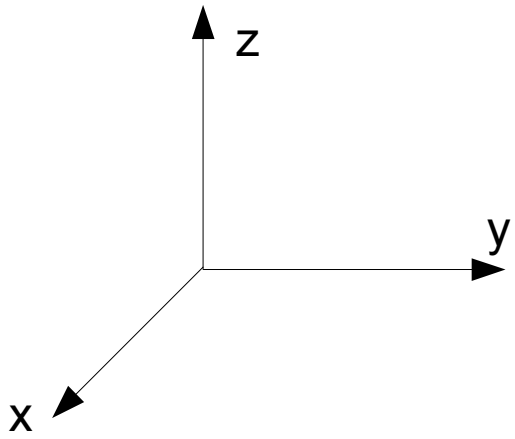
Exercice 8.4
Compléter l'exercice précédent avec un affichage de score (bonne réponse 1, mauvaise réponse – 1, le score ne pouvant pas être négatif.)

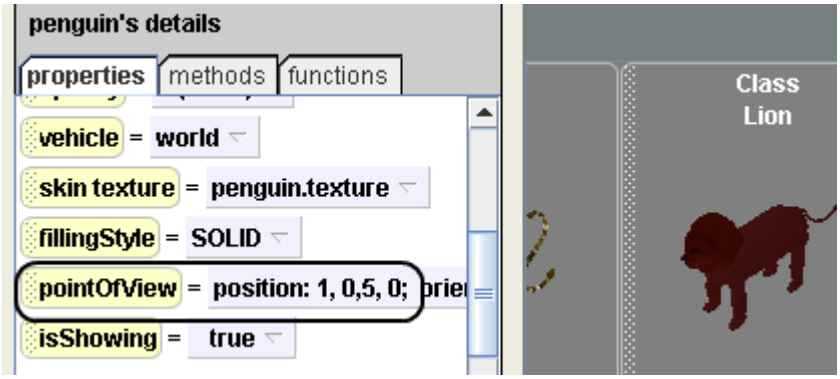
Annexe 1 : Les coordonnées dans Alice

Petit intermède de mathématiques :
Vous savez sans doute qu'un point en mathématique possède 2 coordonnées, une abscisse (x) et une ordonnée (y)



Dans Alice, il y a 3 dimensions, donc 3 coordonnées : x, y et z





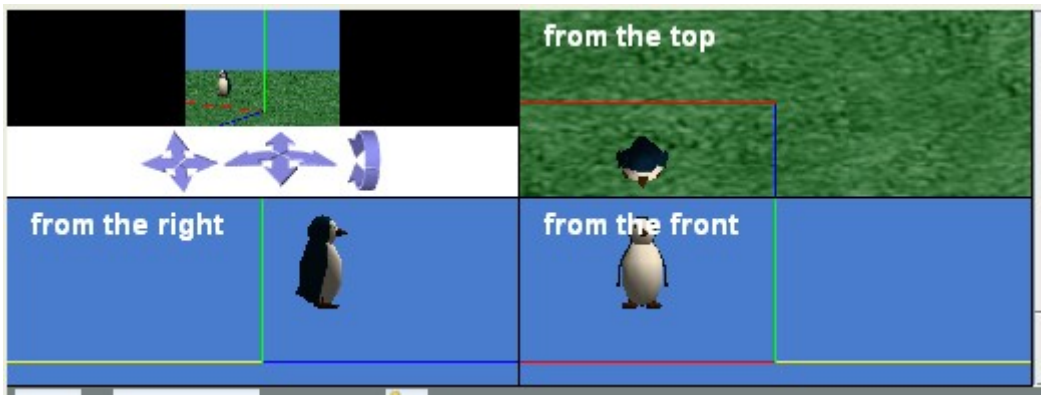
coordonnées de l'objet (1,0.5,0)

Les coordonnées sont visibles en sélectionnant l'onglet " properties " (dans l'onglet properties, on va trouver tous les attributs de l'objet concerné).
Qui dit coordonnées, dit origine (point de coordonnée 0,0,0). L'origine est située au centre du monde. Les coordonnées des objets sont donc toujours données par rapport au centre du monde.
Mais où se trouve le centre du monde ?

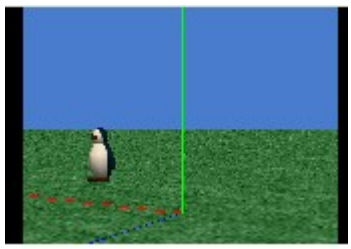
Pour le savoir, il faut cliquer sur " ground ", on voit apparaître 3 axes (un rouge, un vert et un bleu), le centre du monde se trouve à l'intersection de ces 3 axes.

Sur cette image on voit les 3 axes (le bleu, le rouge et le vert) (on voit aussi un pingouin !).

Nous pouvons maintenant déterminer les coordonnées d'un objet :



" quad view " du pingouin



" single view " du pingouin

Nous pouvons lire les coordonnées du pingouin :

ntOfView = position: 1,44, 0,58, 0,62;

Ici le pingouin a pour coordonnées : 1,44 mètres par rapport à l'axe rouge, 0,58 mètre par rapport à l'axe vert et 0,62 mètre par rapport à l'axe bleu.
La première coordonnée correspond donc à l'axe rouge, la deuxième coordonnée correspond à l'axe vert et la troisième coordonnée correspond à l'axe bleu. En maths on a (x,y,z) pour les coordonnées d'un point, dans Alice on aura donc (rouge,vert,bleu).
Petit problème (cela ne vous aura sans doute pas échappé !) :
Alice nous donne : " position: 1,44, 0,58, 0,62 " que de virgules !!!

Si notre ordinateur avait été anglo-saxon, on aurait eu : 1.44,0.58,0.62 (le point correspond à la séparation entre la partie entière et la partie décimale, la virgule correspondant à la séparation entre les coordonnées). Mais dans notre système : des virgules, que des virgules, il faut faire avec !!

Annexe 2 : Les méthodes move, turn, roll

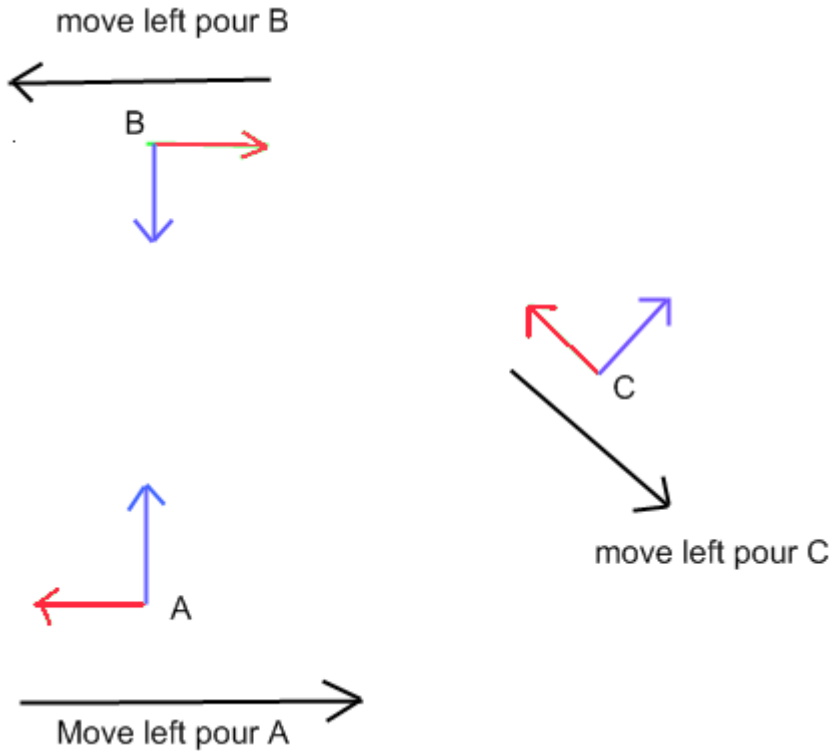
La méthode “move” et le système d'axes

Nous avons déjà abordé le problème des coordonnées d'un objet, mais développons un peu le sujet : comme déjà dit auparavant chaque objet possède 3 axes, un rouge, un vert et un bleu. Le bleu correspond à avant arrière (la flèche de l'axe est vers l'avant), le rouge correspond à gauche-droite (la flèche est vers la droite) et le vert correspond à haut-bas (fléchée vers le haut).

La méthode “move” permet de se déplacer :

- vers la gauche (left)
- vers la droite (right)
- vers le haut (up)
- vers le bas (down)
- vers l'avant (forward)
- vers l'arrière (backward)

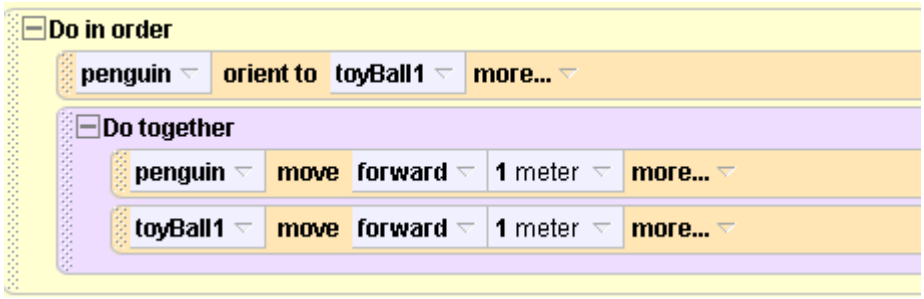
Il faut bien savoir que chaque objet possède son propre système d'axes, et que la gauche pour un objet n'est pas forcement la gauche pour un autre objet.
Voici 3 personnages A,B et C (vus du dessus), on leur applique à chacun la méthode “MOVE LEFT”



Vous pouvez constater que les objets ne vont pas du tout se déplacer dans la même direction alors qu'on leur a appliqué la même méthode (move left).
Cela complique donc énormément les choses si l'on veut déplacer 2 objets dans la même direction !
En fait, pas vraiment, car les concepteurs d'Alice ont pensé au problème !

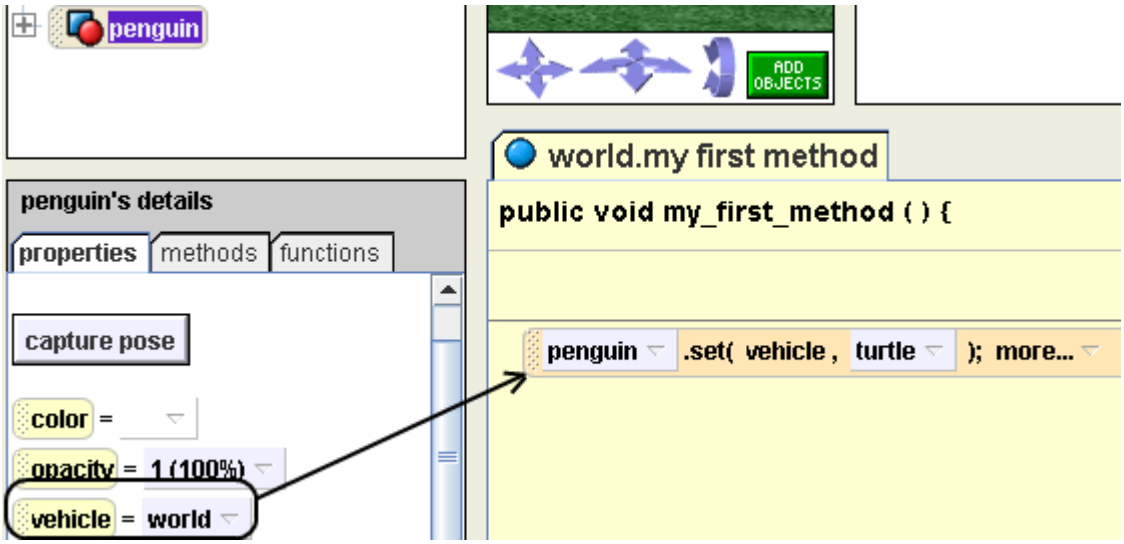
Prenons un exemple :
Créer une scène avec un pingouin et une balle (“toyball1” dans “sports”), mettre le pingouin sur la balle.
Faire avancer le pingouin et la balle de 1 m vers l'avant (MOVE FOREWARD). Comme vous pouvez le constater, la balle et le pingouin ne prennent pas la même direction.

Nous allons “synchroniser” les systèmes d'axes du ballon et du pingouin, pour ce faire nous utiliserons la méthode “orient to”: toyball.orient to(penguin)



Désormais les axes du pingouin et de la balle sont alignés.

Il existe une deuxième manière pour associer 2 objets au cours d'un déplacement, la “properties” “vehicle” :



Pour illustrer l'utilisation de cette méthode, créons une scène avec une tortue et un pingouin (le pingouin est sur le dos de la tortue)

La méthode “MOVE” appliquée à “turtle” déplacera aussi le pingouin.

La méthode “TURN”

la méthode turn peut être appliquée :

- à gauche
- à droite
- vers l'avant
- vers l'arrière

La méthode “TURN” “LEFT” entraînera une rotation de l'objet autour de l'axe vert, vers la gauche
La méthode “TURN” “RIGHT” entraînera une rotation de l'objet autour de l'axe vert, vers la droite
La méthode “TURN” “FOREWARD” entraînera une rotation de l'objet autour de l'axe rouge, vers l'avant.
La méthode “TURN” “BACKWARD” entraînera une rotation de l'objet autour de l'axe rouge, vers l'arrière.

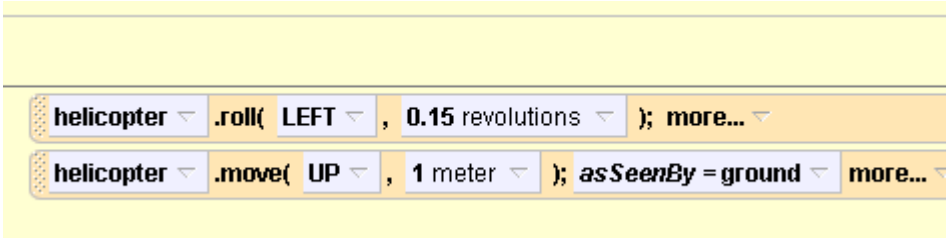
La méthode “ROLL”

la méthode roll peut (seulement) être appliquée :

- à gauche
- à droite

La méthode “ROLL” “LEFT” entraînera une rotation de l'objet autour de l'axe bleu, vers la gauche
La méthode “ROLL” “RIGHT” entraînera une rotation de l'objet autour de l'axe bleu, vers la droite

Attention : Il ne faut jamais oublier que la rotation d'un objet entraînera une rotation des axes associés à l'objet. Il faut donc être attentif en cas de rotation successive ou en cas de rotation suivie d'une translation, voici un exemple :
Créer un monde avec un hélicoptère. Appliquer la méthode “ROLL” “LEFT” “0.15 revolutions” de ce dernier. À la suite, appliquer la méthode “MOVE” “UP” “1 meters”. L'hélicoptère part sur le côté au lieu de s'élever, encore un bug !!?
Non, cela est tout à fait normal, “MOVE” “UP”, se fait selon l'axe vert de l'hélico, or cet axe à tourné en même temps que l'hélico (avec la méthode “ROLL”), donc l'hélico, “monte” en suivant l'axe vert, c'est-à-dire sur le côté !
Le paramètre “asSeenby” suivi de “world” demande à Alice d'utiliser les axes liés à “world” (à la place des axes de l'hélico) seulement pour la méthode associée à “asSeenby” (ici “MOVE” “UP”)



Vous pouvez constater que maintenant l'hélico monte à la verticale (au lieu de partir sur le côté).

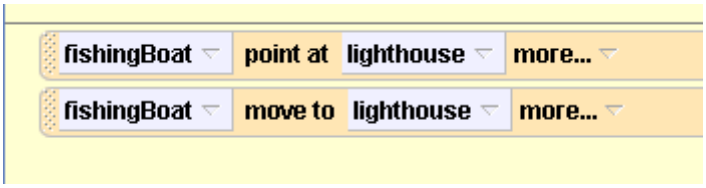
La méthode “point at”
La méthode “point at” ressemble un peu à la méthode “turn to face” vu précédemment. Elle permet de faire pointer l'axe bleu (forward) d'un objet 1 vers le centre d'un objet 2.

La méthode “move to”

La méthode “move to” permet de déplacer un objet 1 de telle façon à ce que le centre de l'objet 1 coïncide avec le centre de l'objet 2.

Pour illustrer ces 2 méthodes, un petit exemple s'impose :

Créer un environnement marin avec un phare et un bateau. Aligner le bateau sur le phare (méthode “point at”) et déplacer le bateau jusqu'au centre du phare (méthode “move to”).

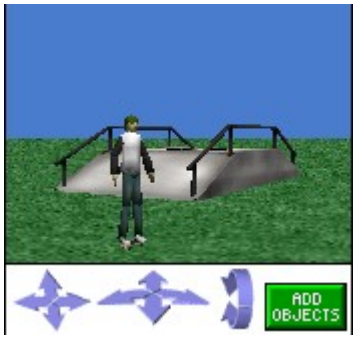


Voilà, admirez le résultat (le bateau est rentré dans le phare).

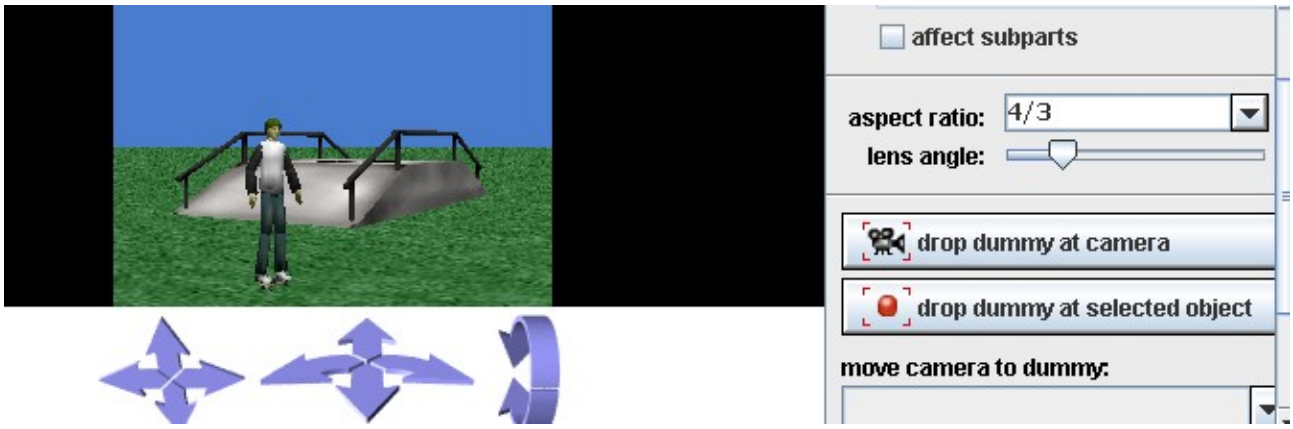
Annexe 3 : La caméra

Pendant la mise en place d'une scène, vous pouvez déplacer la caméra à l'aide des touches :

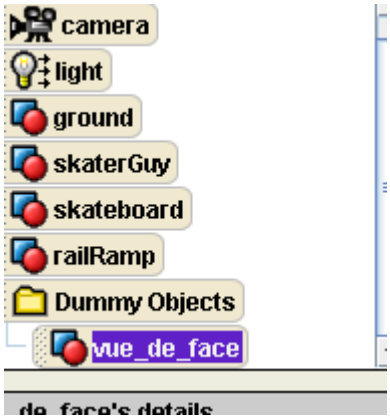
Ces touches ne sont pas utilisables pendant l'exécution d'un programme. Alors, comment faire des effets de caméra ?
Nous allons utiliser des objets factices (de faux objets, des objets invisibles). Ces objets seront placés lors de la mise en place de la scène. Il suffira ensuite d'associer la caméra à l'un de ces objets grâce à la méthode "set point of view to", pour que la caméra "filme" la scène depuis l'objet factice choisi. Bien évidemment, au cours d'une même scène, on pourra changer de point de vue en changeant d'objet factice. L'objet factice peut être considéré comme le trépied d'une caméra. Avec plusieurs objets factices, vous aurez à votre disposition plusieurs trépieds et donc plusieurs possibilités pour placer votre caméra.
Encore une fois un bon exemple vaut mieux qu'un long discours :
Créer une scène avec un skateboarder et une rampe.



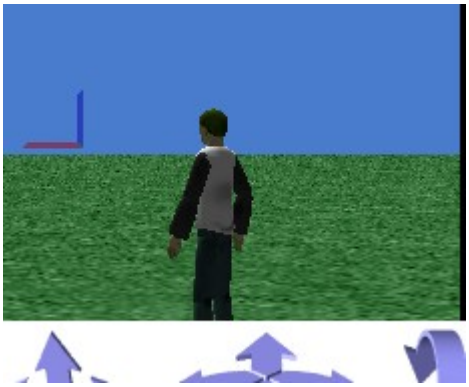
Nous allons placer 2 trépieds (invisibles), 1 qui nous permettra de voir le skateboarder de face et l'autre qui nous permettra de le voir de dos.
Sélectionner "ADD OBJECTS", puis "more controls >>", vous devez alors avoir à l'écran :



Cliquer sur "drop dummy at camera" pour déposer votre 1er objet factice à l'endroit où se trouve la caméra. Un dossier Dummy Objects apparaît dans la fenêtre des objets. Ouvrir ce dossier, vous constatez qu'un nouvel objet nommé "dummy" est apparu. Renommer cet objet en "vue_de_face".



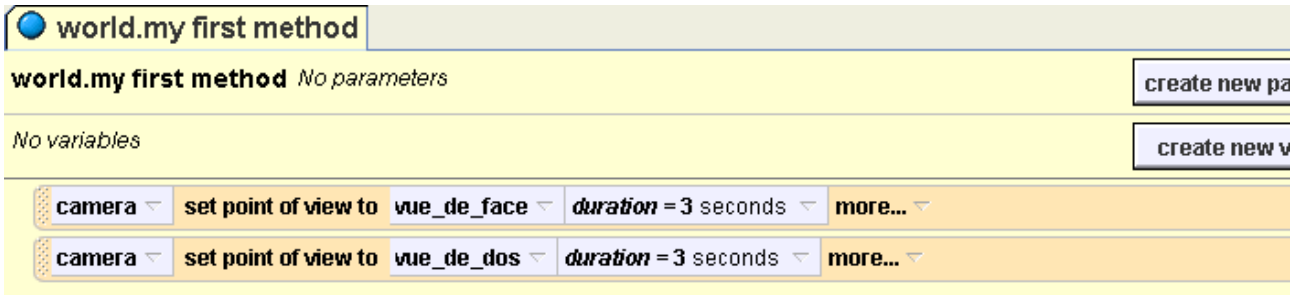
Déplacer votre caméra (à l'aide des touches de direction) de façon à voir le skateboarder de dos.



Cliquer de nouveau sur "drop dummy at camera", un 2e "dummy" apparaît dans la liste renommer le en "vue_de_dos"

Passons maintenant à la programmation :

Utilisons la fonction "set point of view to" pour regarder la scène de face pendant 3 secondes puis de dos aussi pendant 3 secondes



Voilà comment obtenir des effets de caméra. Je vous laisse découvrir par vous-même l'utilisation de "drop dummy at selected objet" car le principe est le même.

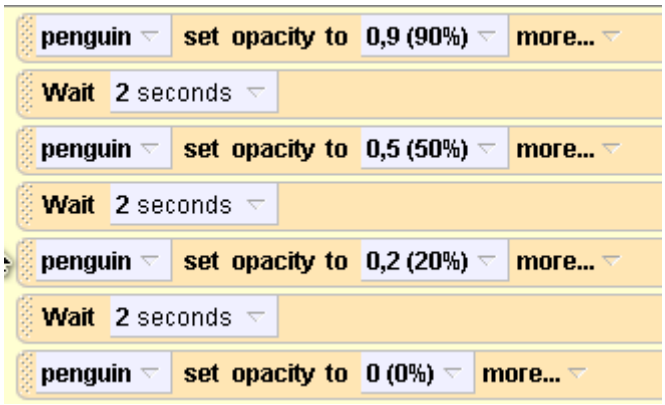
Il existe une autre possibilité pour "jouer" avec la caméra :
Vous pouvez aussi obliger la caméra à se déplacer en même temps qu'un objet en modifiant le "vehicle property" de la caméra. Reprenons notre scène Robot-ET et arrangeons-nous pour que la caméra suive le robot lors de son déplacement vers le rocher (voir le fichier act 3-5c.a2w).

Annexe 4 : Rendre les objets invisibles

Nous allons ici aborder 2 attributs (properties) qui, même s'ils ne sont pas fondamentaux, peuvent être intéressants à utiliser : "opacity" et "isShowing"

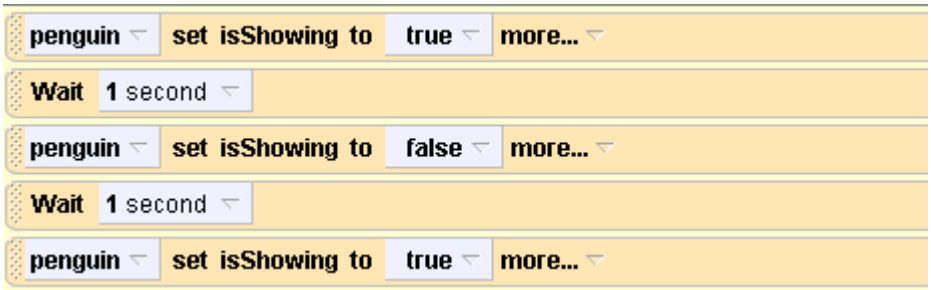
Opacity

Opacity va nous permettre de modifier la transparence d'un objet de 0% (complètement transparent) à 100 % (complètement opaque), toutes les valeurs intermédiaires étant possibles. Choisir une valeur intermédiaire peut vous permettre de "simuler" un fantôme (un spectre).Comme tous les attributs, opacity peut-être modifiée dans un programme (glisser-déposer) :



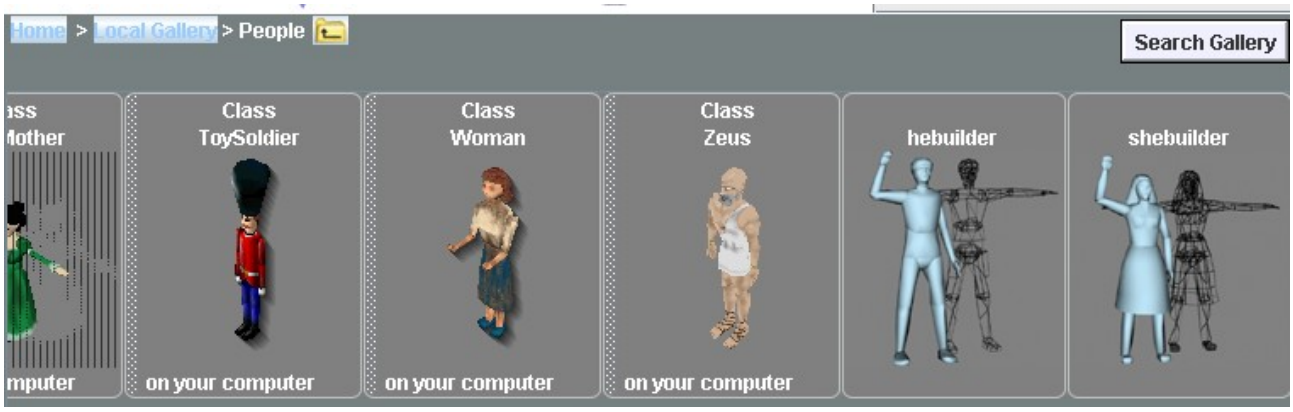
isShowing

Nous pouvons aussi utiliser isShowing à la place de opacity (opacity ne fonctionne pas sur mac !) isShowing fonctionne en tout ou rien, il peut prendre uniquement 2 valeurs : true et false. Si isShowing = true, l'objet est complètement opaque, si isShowing=false, l'objet est invisible. Bien évidemment comme pour opacity, isShowing peut-être utilisé dans un programme (combiné à un if, cela peut donner des choses intéressantes).

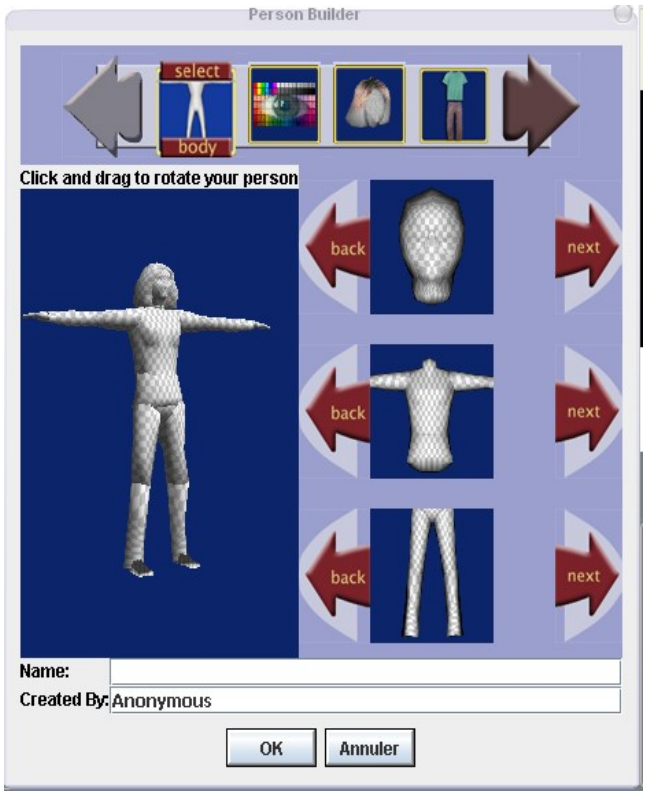


Annexe 5 : Créer ses propres personnages

Vous pouvez créer vos propres personnages en allant dans la galerie People



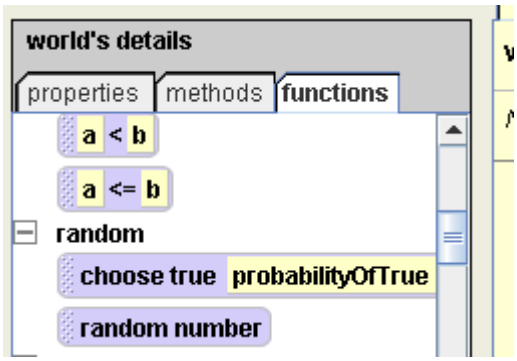
Pour "fabriquer" un homme choisir hebuilder et pour fabriquer une femme choisir shebuilder.
Vous vous retrouvez alors avec des milliers de combinaisons possibles, faites votre choix :



Enfin pour terminer, Alice propose une importante collection de méthodes pour chaque personnage créé, n'hésitez pas à les utiliser.

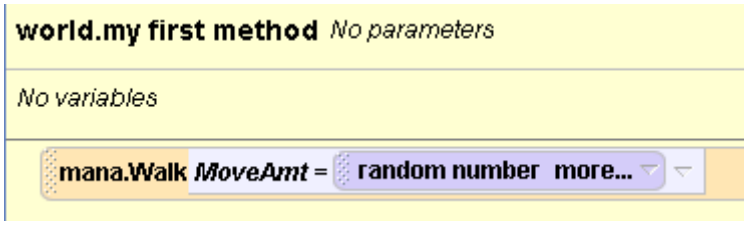
Annexe 6 : Nombres aléatoires

En programmation, il est relativement courant d'avoir besoin de nombres tirés au hasard par l'ordinateur (un exemple très simple, un programme qui simulera un lancé de dés devra avoir une instruction qui tirera au hasard un nombre entre 1 et 6).
Dans Alice, il existe bien évidemment une fonction qui vous permet de tirer au hasard un nombre.
Vous trouverez cette fonction dans les fonctions de "world"

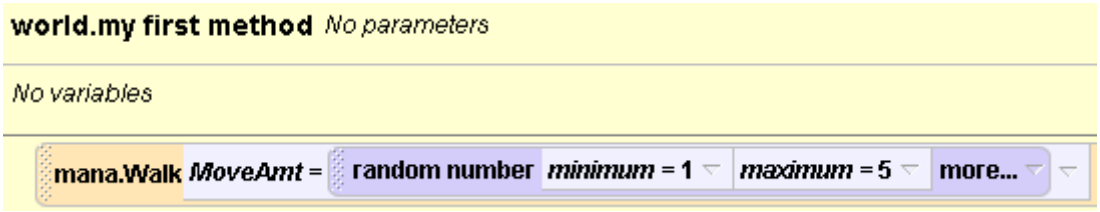


C'est la fonction "random number"

Un petit exemple pour illustrer cela :
Placer le personnage Mana. Utilisons-la class level method "mana.walk", choisir un nombre (par exemple 3) comme paramètre (MoveAmt). Remplacer alors 3 par la fonction "random number"



Choisir "more..." pour donner l'encadrement de la valeur aléatoire (minimum et maximum):



Ici la fonction random number renverra un nombre compris entre 1 et 5
Lancer le programme plusieurs fois, vous pouvez alors constater que Mana parcourt une distance différente à chaque fois.

Ici, la fonction random number renvoie un nombre qui ne sera pas forcément un entier (on pourrait avoir 1,2 ou 4,8). Si vous avez besoin d'un entier (par exemple pour une simulation de lancé de dés) vous devez utiliser encore une fois "more..." de la fonction "random number" et choisir "true" pour "integerOnly"