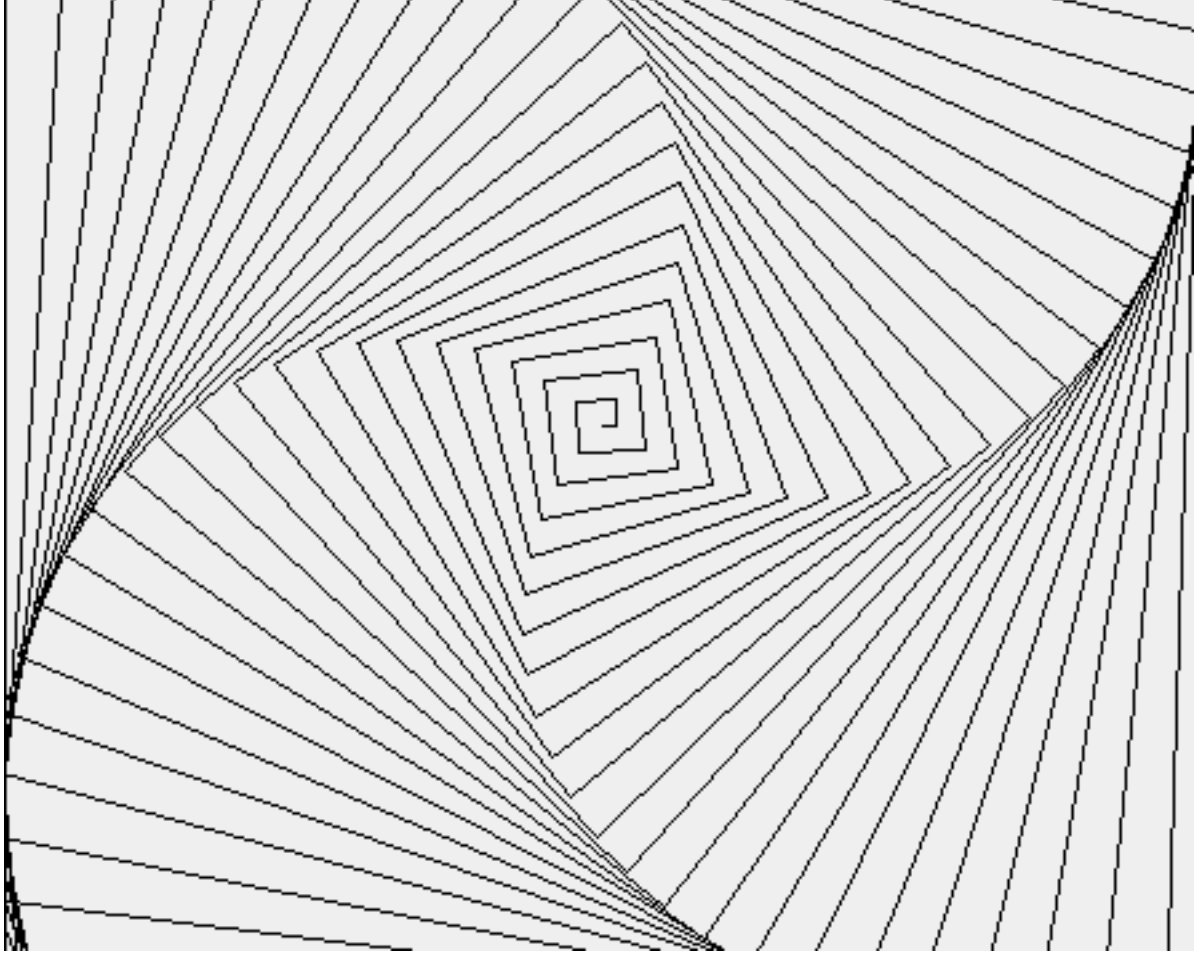


Dis-moi caro, c'est quoi programmer ?

Cours de Technologie Année 1999-2000

Ecole française de Berne
Stéphane et Florence Ducasse
ducasse@iam.unibe.ch

August 19, 2000

Plan détaillé

Stéphane & Florence Ducasse (ducasse@iam.unibe.ch)

Notes aux lecteurs

Le document que vous êtes en train de lire représente le résultat d'un cours donné à l'Ecole Française de Berne en classe de Technologie pour les classes de 6 ème, 5ème, 4ème et 3ème commencé en Novembre 1999.

Le présent cours est en constante évolution. Les chapitres contenus dans ce document sont les chapitres qui ont déjà été expérimentés par les élèves et que nous considérons comme stables. La forme et le fond seront peut-être sujets à des changements. Ce cours et cet environnement représentent beaucoup d'énergie aussi s'il vous arrivait de le copier, copiez toujours ce document car nous voulons que les lecteurs aient conscience de nos intentions.

D'autre part, s'il vous arrivait d'utiliser ce cours, nous aimerions être tenus au courant de vos résultats.

Acquérir des concepts informatiques

Nous présentons rapidement ici quels sont nos objectifs d'acquisition. Les tortues sont des animaux informatiques qui ont longtemps été utilisés pour l'enseignement de concepts mathématiques. Ceci n'est pas notre objectif principal bien que nous ayons besoin de certains concepts mathématiques. Notre objectif principal est l'enseignement de *concepts informatiques*.

Voici les objectifs d'acquisition que nous nous sommes fixés:

- Notion d'état
- Messages (en première approximation des instructions)
- Séquence de messages
- Boucles
- Variables
- Notion d'abstraction
- Définition de méthodes (en première approximation des procédures)
- Réutilisation de code
- Elaboration de nouvelles instructions par composition d'autres instructions préalablement définies
- Notion d'objet (non couvert par ce support)

Ce cours n'est pas un cours de LOGO même si nous utilisons une tortue graphique semblable à celle de LOGO. Certains concepts n'existent pas en LOGO comme la notion d'objets, de classes, de messages... Bien sûr on pourra utiliser l'environnement en mode LOGO si l'on extrait les éléments nécessaires et en fait une relecture. Pour cela, la définition de la commande POUR de LOGO pourrait être simulée en créant une méthode au niveau de la classe `Turtle` qui appelerait le compilateur directement. Ceci est simple à faire mais si vous ne savez pas comment faire contactez-nous.

Nous utilisons et introduisons les concepts mathématiques suivants lorsqu'ils sont nécessaires :

- Direction
- Angles
- Mouvement relatif/absolu
- Repère dans le plan

Contraiment à certaines implantations de tortues ou équivalent, nous avons choisi de suivre les conventions mathématiques pour les angles afin que les élèves n'aient pas à apprendre deux conventions.

Stéphane & Florence Ducasse (ducasse@iam.unibe.ch)

Votre avis nous intéresse!

Nous sommes intéressés par vos remarques, vos expériences, exercices et suggestions pour améliorations. Plus particulièrement , nous sommes intéressés par les points de vues suivants:

- Comment utiliser ce cours pour des enfants du primaire?
- Comment utiliser ce cours comme support de cours de mathématiques (primaire ou secondaire) ?
- Comment vous, adulte, comprenez les concepts présentés? Est-ce que les concepts sont introduits de manière compréhensible? Les exercices sont-ils adaptés?
- Comment utiliser ce cours comme base pour la compréhension de la programmation?

Améliorations possibles

L'environnement actuel est le résultat d'une première implantation développée très rapidement. La liste de possibles améliorations est extrémement longue, voici les idées les plus importantes que nous avons :

- Un véritable environnement en Squeak (debugger, exécution lente de la tortue, aide à la syntaxe peut être basée sur les SyntaxMorphs).
- Une version française des méthodes et de l'environnement de manière propre et automatisée.

Où trouver les plus récentes versions ?

`http://www.iam.unibe.ch/~ducasse/Teaching/` contient des pointeurs sur la dernière version du cours et de l'environnement. Vous pouvez aussi nous contacter à `ducasse@iam.unibe.ch`. D'autre part, nous sommes en train de développer un Wiki (un ensemble de pages html éditables de manière collaborative) qui regroupe les meilleurs graphiques des éléves de l'Ecole Française de Berne : `http://kilana.unibe.ch:8080/TurtleGallery/`. Allez-y jeter un oeil et ajouter vos propres graphiques....parce que tout le mondre peut éditer ce site.

Stéphane et Florence Ducasse

Installation

Dans ce chapitre, vous allez apprendre à charger le code des tortues dans Squeak et vous familiariser avec l'environnement que vous allez utiliser. Dans une version ultérieure, nous proposerons un environnement dans lequel le code sera déjà chargé.

1 Squeak

Nous nous servons de Squeak. Squeak qui est un environement développé par Disney Lab et précédemment Apple est gratuit et plus d'informations peuvent être trouvées à `http://www.squeak.org/`. Si vous prenez Squeak sur Mac n'oubliez pas de prendre le fichier SqueakV2Sources et de le placer dans le dossier Squeak. Ce fichier est inclus automatiquement dans la version PC.

1.1 Commencer avec Squeak

Les informations suivantes vous sont présentées afin que vous puissiez recréer le même environnement de travail sur votre ordinateur.

Ouvrir Squeak Pour ouvrir Squeak, double cliquez sur le fichier Squeak2.7.image. Vous devez obtenir une fenêtre identique à celle reproduite dans la figure 1.1.

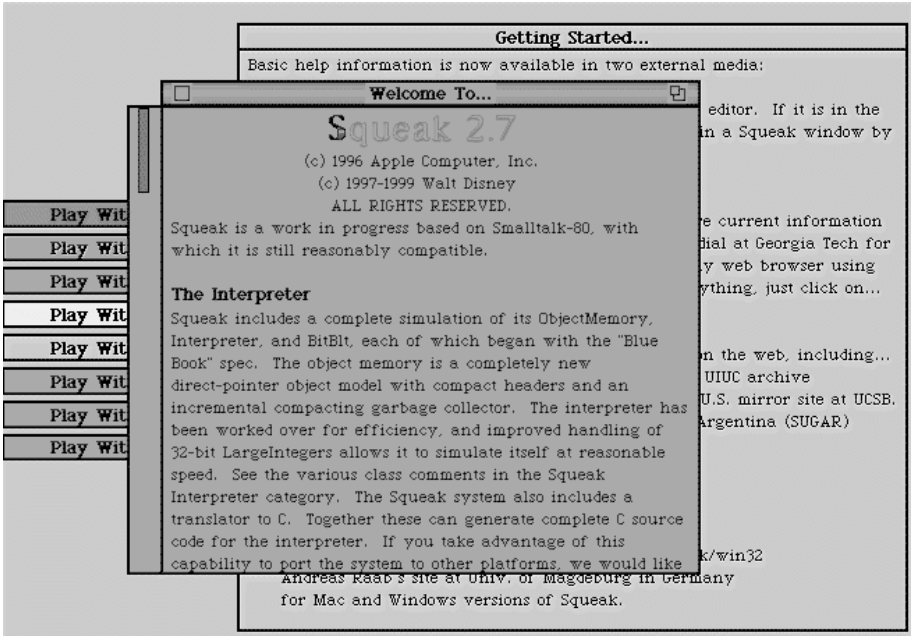


Figure 1.1: Bienvenue dans Squeak.

1.2 Charger l'environnement

L'environnement définissant les tortues n'est pas contenu automatiquement dans Squeak. Pour cela il faut que vous chargiez dans Squeak le fichier `EnvTurtle5xs.2.cs` Pour cela vous devez ouvrir une liste de fichiers (file list).
Voici la procédure à suivre étape par étape:

Etape 1: Ouvrir une liste de fichiers

Tout d'abord en cliquant faites apparaître le menu et sélectionner **open** puis sélectionnez le choix **file list...**

La fenêtre représentée par la figure 1.2 doit apparaître :

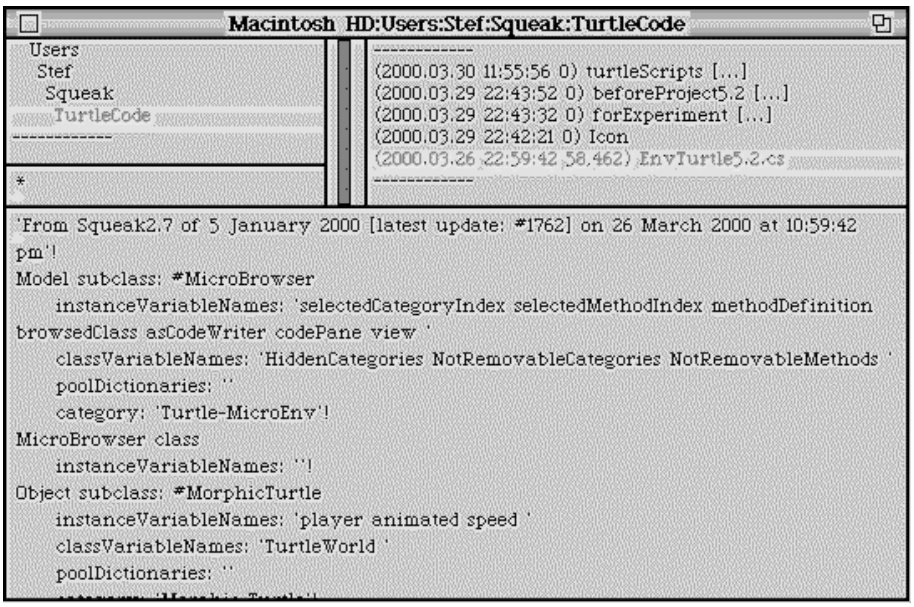


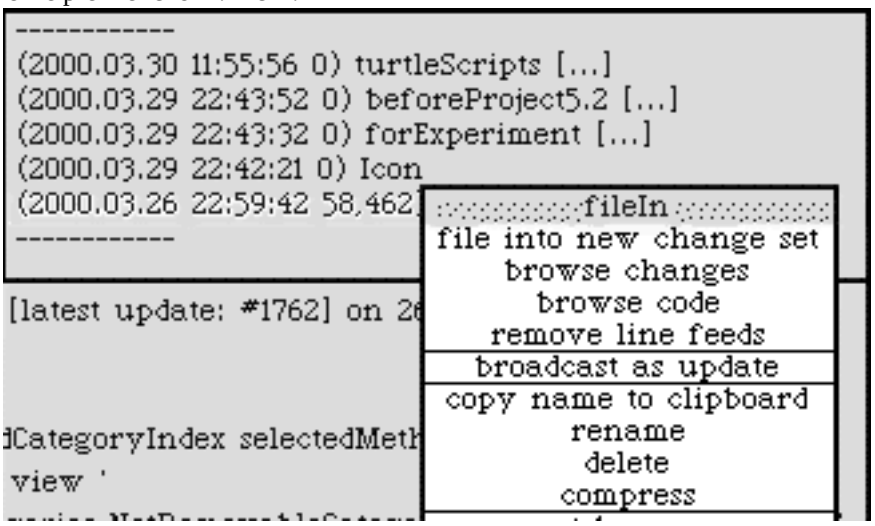
Figure 1.2: Le sélecteur de fichier.

Etape 2: Charger `EnvTurtle5xs.2.cs`

Pour charger dans Squeak le fichier `EnvTurtle5xs.2.cs` vous devez sélectionner dans la partie en haut à gauche le disque sur lequel se trouve `EnvTurtle5xs.2.cs` Ici, il se trouve dans sur le disque dur (non

visible) puis dans Squeak, SqueakTurtle. Lorsque vous cliquez sur un dossier, son contenu s'affiche dans la partie en haut à droite.

Dans cette partie, sélectionnez le fichier `EnvTurtle5xs.2.cs` uis faites apparaître le menu suivant et sélectionnez le premier choix : **fileIn**.



Un indicateur de chargement doit apparaître afin de vous montrer l'état d'avancement du chargement du fichier. Lorsque celui-ci disparaît vous pouvez détruire le sélecteur de fichiers en cliquant sur le coin de la fenêtre marqué d'un X.

Etape 3: Sauvez votre environnement

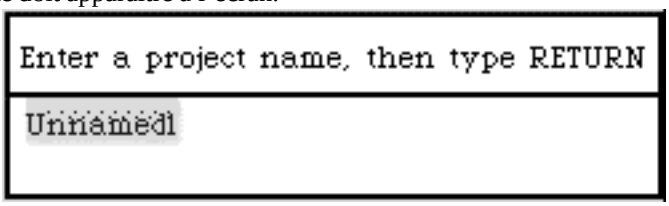
Maintenant vous pouvez sauver votre environnement comme cela vous n'aurez plus à recharger le fichier définissant les tortues. Notez que vous pouvez sauver le système à d'autres moments afin de pouvoir revenir dans le même état ultérieurement.

1.3 Créer un espace de travail

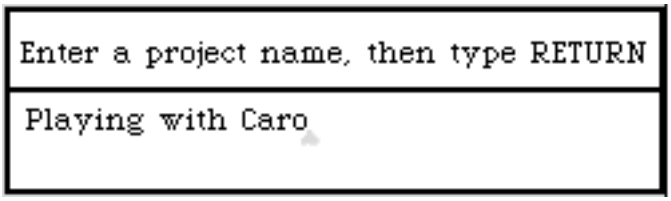
Suivez les étapes suivantes pour créer un espace de travail.

Créer un projet morphique (morphic-turtle project)) Faites apparaître le menu suivant et sélectionnez le choix **open...** vous devez obtenir le second menu. Dans ce second menu sélectionnez le choix morphic-turtle project.

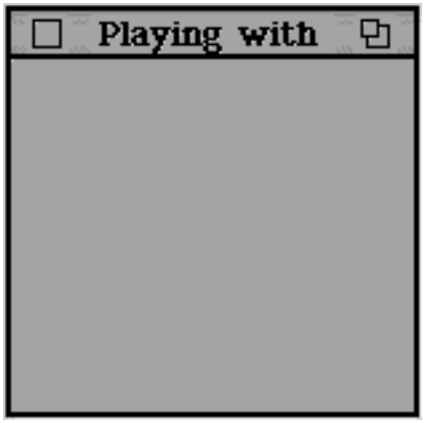
La fenêtre suivante doit apparaître à l'écran.



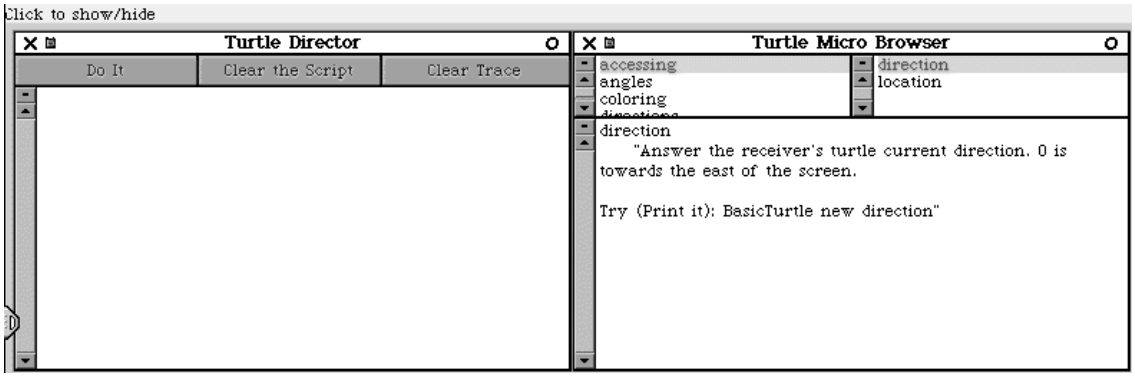
Entrez un nom.



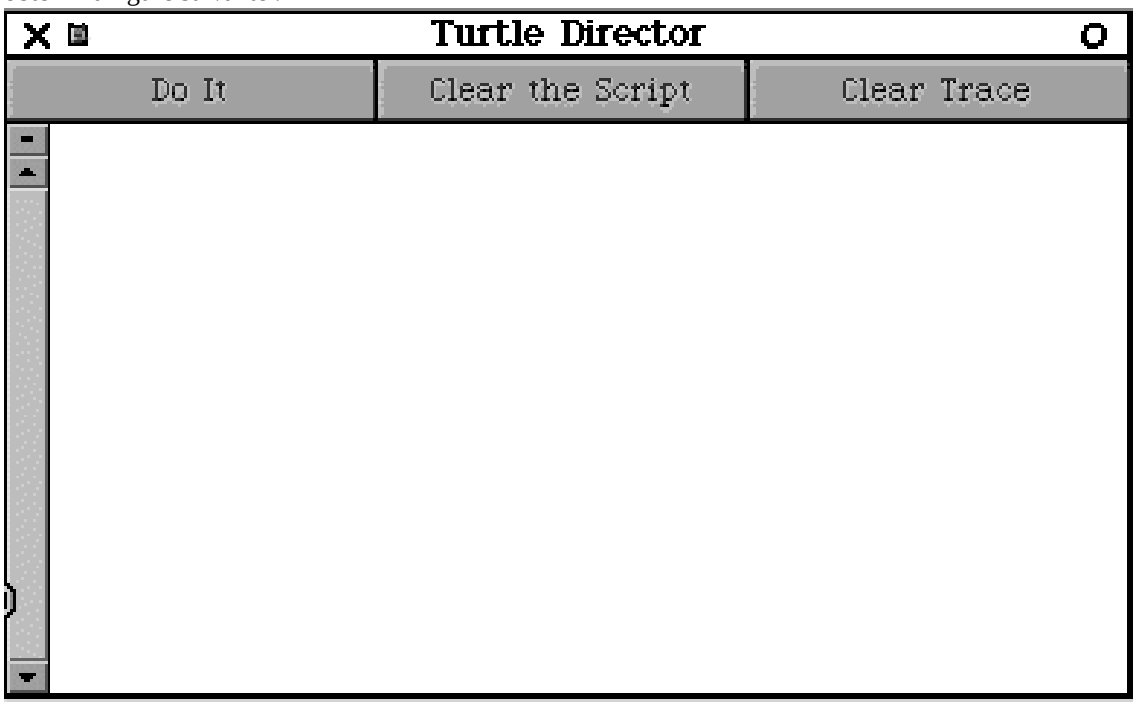
Une nouvelle fenêtre doit apparaître. Cliquez ensuite sur cette nouvelle fenêtre et sélectionnez le choix **enter** comme présenté par l'image suivante.



Vous devez être maintenant dans un projet morphique et voir l'environnement suivant :



La partie gauche est l'évaluateur dans lequel vous allez écrire vos scripts. La partie droite est l'éditeur que vous utiliserez pour enseigner de nouvelles méthodes aux tortues à partir du chapitre **??**. Pour le moment, vous pouvez détruire l'éditeur en cliquant sur la croix en haut à gauche de l'éditeur. Vous devez obtenir la figure suivante :



Astuce... Notez que c'est un bon moment pour sauver votre image.

2 Soulevons le couvercle

Squeak comme les autres environnements de programmation du langage Smalltalk fonctionne un peu comme une grosse, très grosse et très puissante calculatrice. La calculatrice s'appelle une *machine virtuelle*, c'est elle qui calcule et exécute le code. C'est le fichier executable (.exe sur pc). Cette machine virtuelle a besoin d'information qu'elle exécute. Cette information se trouve dans trois fichiers, le fichier .image, le fichier .changes et le fichier sources. Le fichier source représente le code de tout Squeak tel que vous l'obtenez si vous prenez Squeak sur le réseau.

En fait, la machine virtuelle n'a besoin que du fichier .image qui représente le code d'une manière compréhensible pour la machine virtuelle. On appelle cela du byte-code, il s'agit non plus de texte mais de code comme 13 37 qui représente des instructions de la machine virtuelle. Le fichier .image contient donc l'équivalent du fichier source mais représenté en byte-codes.

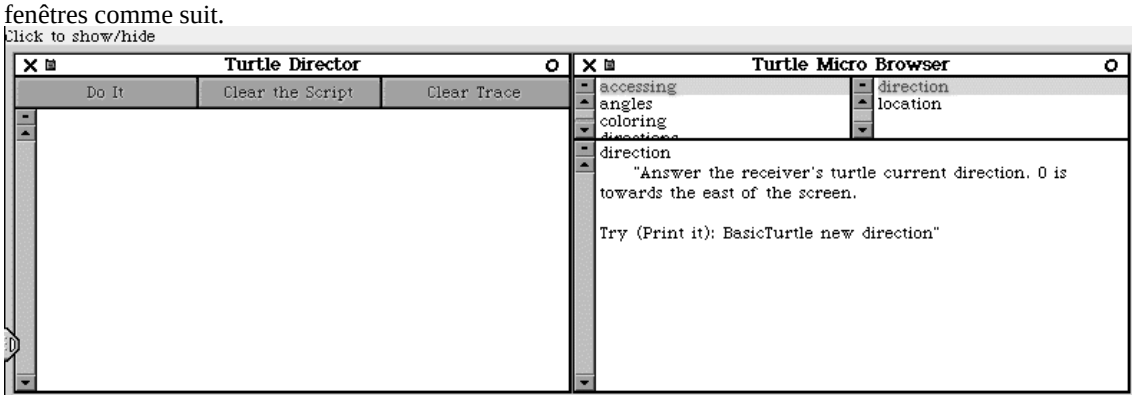
Quand un utilisateur définit de nouvelles opérations en Squeak sous forme de classes et méthodes, le texte est sauvé dans le fichier .changes. En effet, le fichier source peut être partagé par plusieurs Squeak par contre un seul fichier .change est associé à chaque fichier .image. Ces deux fichiers doivent toujours être synchronisés.

Quand un utilisateur veut regarder le code de Squeak, Squeak cherche ses informations non pas dans le fichier .image mais dans les fichiers .source et .change.

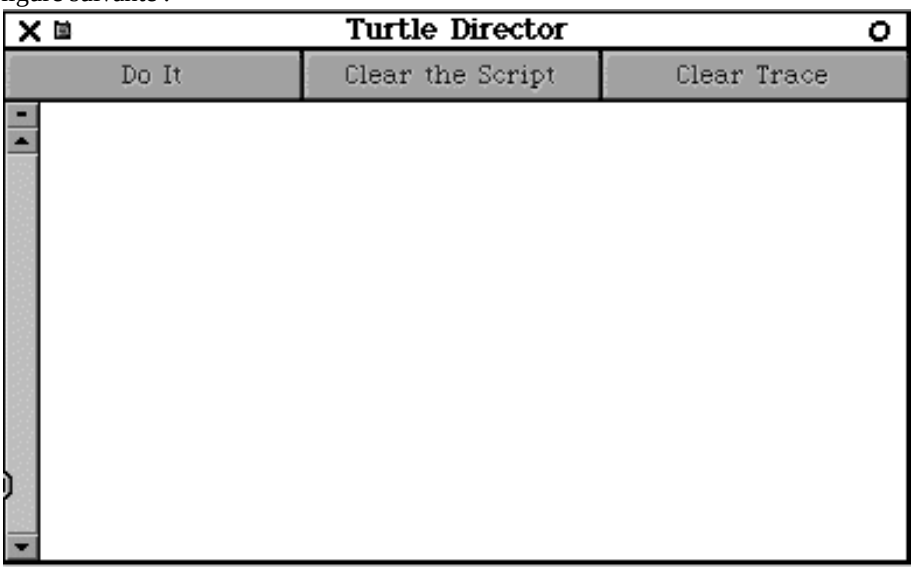
On peut bien sûr sauvez le code que l'on tape dans des fichiers séparés.

Premier Contact

Dans ce chapitre, vous allez vous familiariser avec l'environnement de la tortue. Dans une version ultérieure, nous proposerons un environnement dans lequel le code sera déjà chargé. Lorsque vous avez suivi la procédure de chargement du code de la tortue dans squeak comme montré au chapitre 1, vous obtenez deux fenêtres comme suit.



La partie gauche est l'évaluateur dans lequel vous allez écrire vos scripts. La partie droite est l'éditeur que vous utiliserez pour enseigner de nouvelles méthodes aux tortues à partir du chapitre **??**. Pour le moment, vous pouvez détruire l'éditeur en cliquant sur la croix en haut à gauche de l'éditeur. Vous devez obtenir la figure suivante :



Astuce... Notez que c'est un bon moment pour sauver votre image.

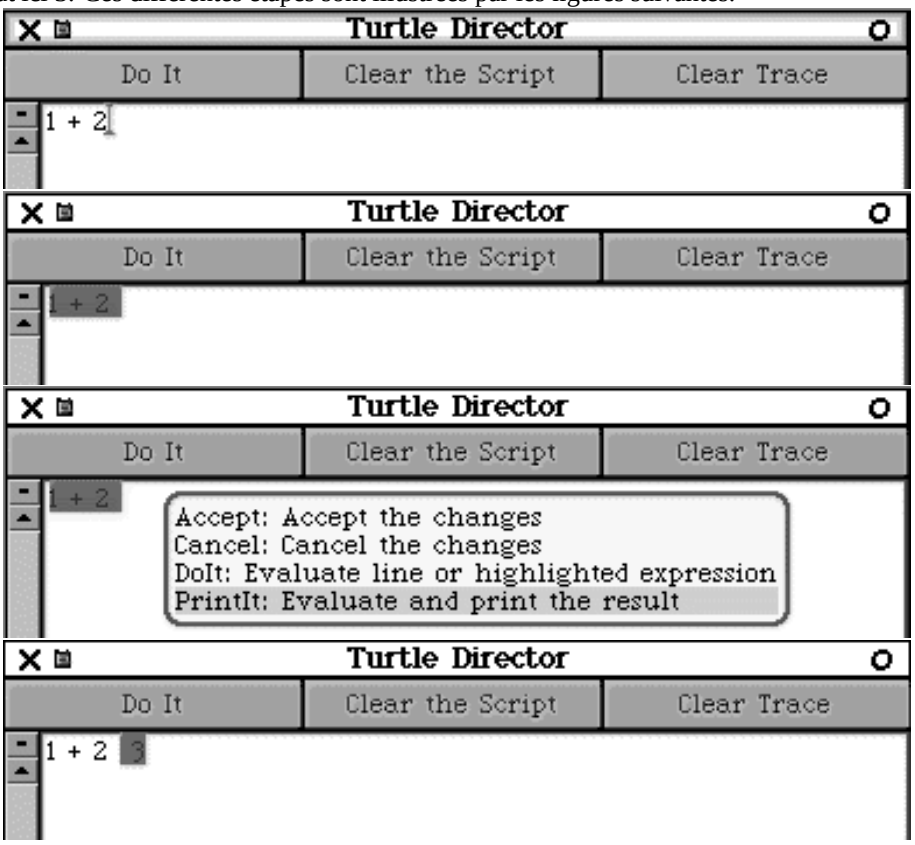
Stéphane & Florence Ducasse (ducasse@iam.unibe.ch)

1 Utiliser l'évaluateur

L'évaluateur est un peu comme une grosse machine à calculer, vous tapez des expressions, vous les sélectionnez et le système fait les actions correspondantes et lorsque cela vous intéresse affiche le résultat de l'action.

1.1 Un premier exemple dont le résultat nous intéresse

Tapez `1 + 2`, sélectionnez l'expression `1 + 2`, ensuite faites apparaître le menu suivant et choisissez *print it*. print It a pour effet de demander au système de faire une action : ici additionner 1 et 2, et *d'afficher* son résultat ici 3. Ces différentes étapes sont illustrées par les figures suivantes.



1.2 Un exemple avec une tortue.

La figure suivante montre la création d'une tortue, une séquence de messages qui lui sont envoyés et leurs effets : ici l'affichage de la tortue, sa progression et un second affichage.

Nous expliquerons dans les chapitres suivants ce qu'est une tortue et les messages que l'on peut lui envoyer. Pour obtenir le même résultat, tapez le code du script suivant dans l'évaluateur et cliquez sur le bouton **Do It**. Ici, le résultat du message `show` ne nous intéresse pas, seul son effet sur la tortue nous intéresse donc nous ne demandons pas à l'évaluateur d'afficher le résultat mais juste d'exécuter le message `show`.

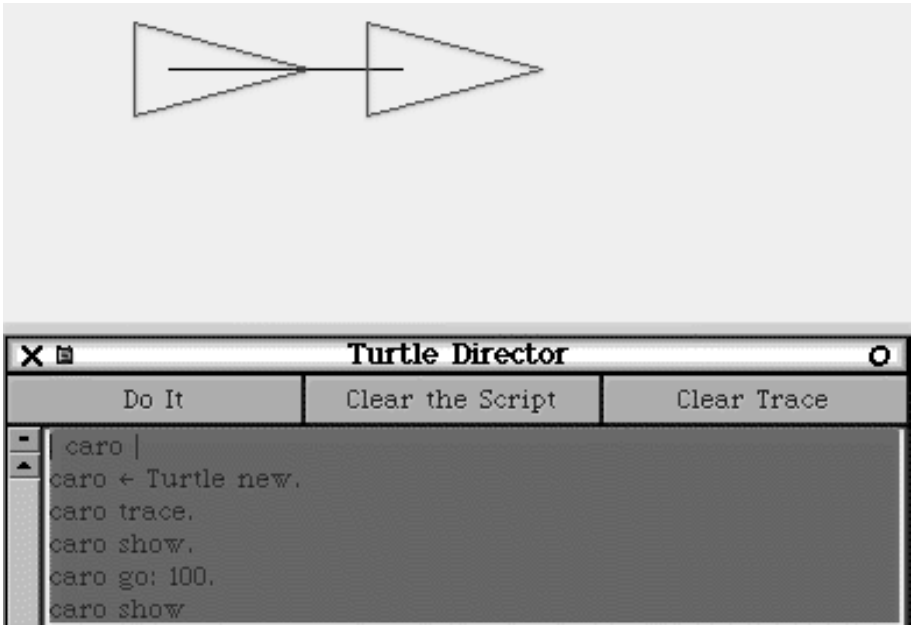


Figure 2.1: Un script (permettant la création d'une tortue et l'envoi des messages) et son résultat.

Script 1 : Mon premier script

```
| caro |
caro ← Turtle new.
caro trace.
caro show.
caro go: 100.
caro show
```

Important! Point de vocabulaire. Nous appelons un *script* un ensemble de lignes de codes qui :

1. crée une tortue et
2. lui envoie une série de messages.

2 Comment créer et envoyer des messages à une tortue?

Nous allons maintenant regarder en détail votre premier script. Pour créer une tortue, nommée Caroline (`caro` pour les intimes), et lui envoyer des messages, suivez les étapes suivantes.

2.1 Etape 1 : Tapez le script suivant dans un évaluateur.

La première ligne `|caro|` déclare que nous allons utiliser un nom, ici `caro`. La seconde ligne `caro ←Turtle new` crée une tortue et l'associe au nom caro, la troisième fait apparaître la tortue (Voir chapitre 7 pour plus d'explications sur les deux premières lignes).

Script 2 : Création d'une tortue et premier message

```
|caro|
caro ← Turtle new.
caro show.
```

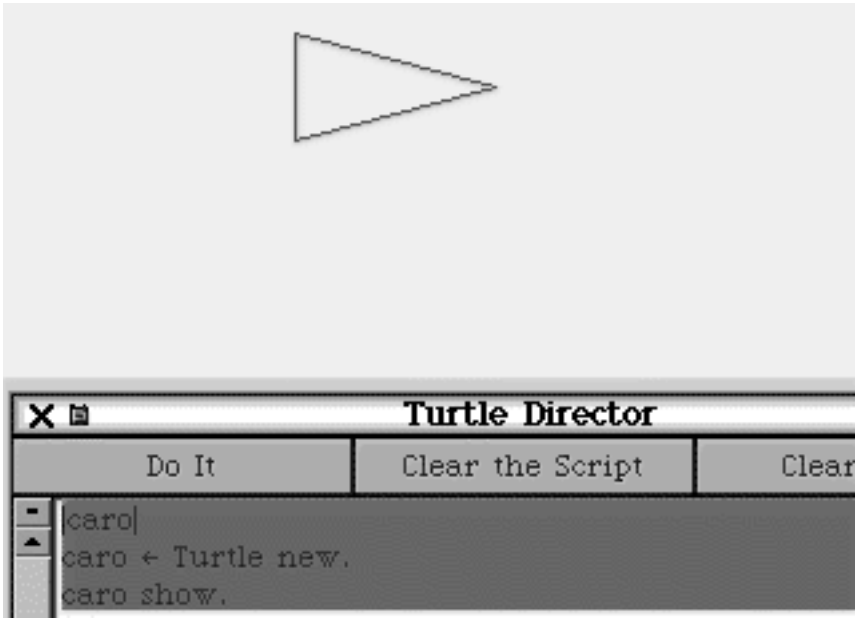


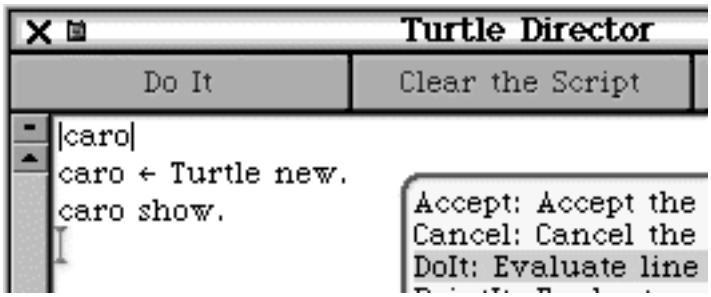
Figure 2.2: Le script crée une tortue et lui envoie un premier message: `show`. Donc la tortue l'exécute et s'affiche.

Remarque. Lorsque vous tapez la touche _ vous obtenez une petite flèche comme le montre la figure 2.2. Les scripts utilisent _ pour représenter la flèche car ce symbole graphique n'existe pas sur les claviers d'ordinateurs.

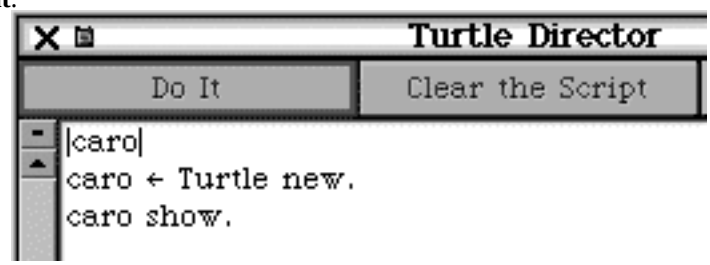
Important! Attention `Turtle` commence par une majuscule car c'est une fabrique de tortues.

2.2 Etape 2 : Evaluer le script

Sélectionnez le texte comme le montre la figure suivante et en cliquant sur le bouton de droite sur PC et Option click sur Mac et choisissez le choix **do it** dans le menu qui apparait.



Comme le montre la figure suivante, presser sur le bouton **Do It**, a le même effet que sélectionner tout le texte et faire **do it**.



2.3 Séquence de messages

Une tortue peut recevoir plusieurs messages afin d'exécuter des figures complexes. Vous allez maintenant demander à une tortue de s'afficher et d'avancer un peu.

Script 3 : Tortue show

```
|caro|
caro ← Turtle new.
caro show.
caro go: 100
```

Ici dernière ligne `caro go: 100` demande à la tortue nommée `caro` d'avancer de 100 pixels. Comme nous ne demandons pas à caro d'apparaître nous ne voyons pas qu'elle a effectivement avancé.

> **Important!** Un message qui nécessite de donner une valeur (comme avancer de 100) finit toujours par un ":" comme `go:`.

Par défaut, une tortue ne s'affiche pas mais on peut lui demander de s'afficher (commande `show`). Envoyez-lui donc le message `show` comme dans le script suivant.

Script 4 : Tortue show-show

```
| caro |
caro ← Turtle new.
caro show.
caro go: 100.
caro show
```

> **Important!** Attention pour qu'une tortue comprenne les messages, on doit les séparer par des points.

Astuce... Deux messages doivent être séparés par un point donc le dernier point n'est pas nécessaire car il ne sépare rien. Mais vous pouvez aussi le mettre cela n'a pas d'importance.

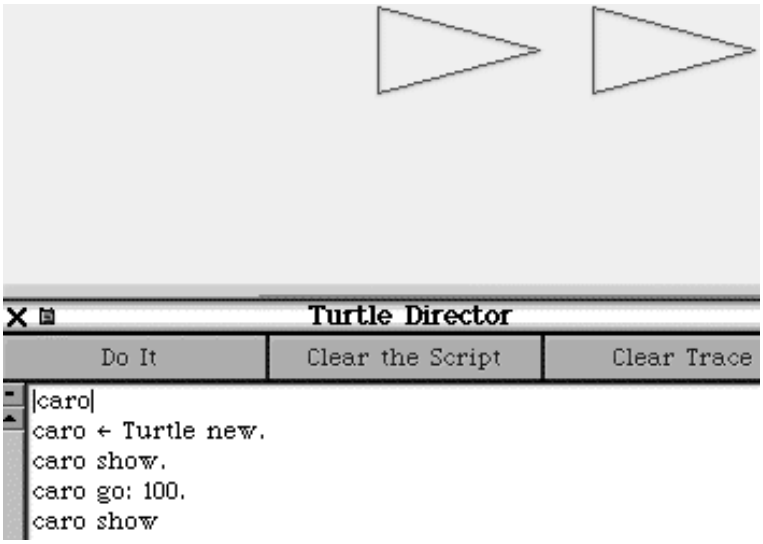


Figure 2.3: Script et résultat de l'évaluation.

3 A retenir

Important! Nous nommons *évaluer une expression* le fait de la taper, de la sélectionner et sélectionner le choix **do it** ou **print it**.

Important! Un message qui nécessite de donner une valeur (comme avancer de 100) finit toujours par un ":" comme `go:`.

Important! Point de vocabulaire. Nous appelons un *script* un ensemble de lignes de codes qui :

1. crée une tortue et
2. lui envoye une série de messages.

Important! Attention `Turtle` commence par une majuscule car c'est une fabrique de tortues.

4 Soulevons le couvercle

4.1 Do it ou Print It

La différence entre évaluer une expression et afficher le résultat de cette évaluation est importante. Afficher le résultat d'une expression (print it) nécessite donc d'évaluer l'expression et d'afficher ce résultat. L'évaluation d'une expression se limite à exécuter une expression et à rendre le résultat qu'il soit intéressant ou pas sans l'afficher.

Dans le cadre de la tortue: la différence entre faire faire une action à une tortue (do it) et imprimer le résultat de l'action ne sera vraiment importante que lorsque l'on commencera à utiliser les résultats des messages envoyés comme savoir la position courante de la tortue ou sa direction.

4.2 Une tortue ou la tortue

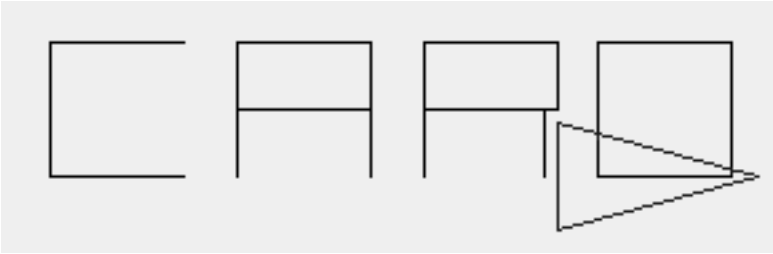
Comme vous l'avez peut-être remarqué nous employons toujours le terme "une tortue" et non "la tortue". En effet, notre but est de vous faire comprendre les concepts de la programmation dans le contexte plus précis de la programmation par objets. Dans notre approche, contrairement à celle de Logo dans lequel il n'existe qu'une tortue, nous pouvons créer autant de tortues que nous le souhaitons.

C'est pour cela que nous disons une tortue car nous ne savons pas exactement de quelle tortue il s'agit, nous savons juste qu'il s'agit d'une tortue.

Une fabrique de tortues. En Squeak, nous avons implanté une *classe* qui représente les tortues. Nous avons construit une classe, une fabrique de tortues, qui permet de créer autant de tortues que l'on veut.

Explorons l'analogie un peu plus loin! Une machine fabriquant des boites de conserves crée des boites ayant les mêmes propriétés (poids, tailles, date de péremption) et même comportements (résistance à l'écrasement). Cependant si on vide une boite donnée son poids va changer, on écrase une boite donnée les autres ne sont pas affectées. De la même manière, une classe décrit les propriétés des tortues : chaque tortue a une position, une couleur, une direction, sait si elle est en en mode écrit, et les comportements d'une tortue (avancer, tourner...). Toutes les tortues ont le même comportement mais sont potentiellement dans des états différents (couleur, position, direction).

Des tortues et des hommes



Dans ce cours nous allons utiliser des *tortues* informatiques pour créer des figures géometriques et vous aider à apprendre à programmer. Pour cela, vous allez *créer des tortues* et interagir avec elles en leur envoyant des *messages*. Dans ce chapitre, vous allez expérimenter afin de découvrir ce que sont les tortues et voir comment on peut communiquer avec elles.

1 Mais au fait qu'est-ce qu'une tortue?

Une tortue est un objet graphique qui a une *position*, une *direction* et une *couleur*. Evaluez les expressions suivantes pour vous en rendre compte. Vous n'êtes, bien sûr, pas obligé de tout retaper à chaque fois mais vous pouvez simplement ajouter ce qu'il manque.

1.1 Expérimentons avec la position

Script 5 : Position au départ

```
|caro|
caro ← Turtle new.
caro show
```

Script 6 : Position et en avant

```
|caro|
caro ← Turtle new.
caro show.
caro go: 100.
caro show
```

Script 7 : Go go

```
|caro|
caro ← Turtle new.
caro show.
caro go: 100.
caro show.
caro go: 200.
caro show
```

1.2 Expérimentons avec la direction

Script 8 : Direction west

```
|caro|
caro ← Turtle new.
caro show.
caro west.
caro show
```

Script 9 : Far West

```
|caro|
caro ← Turtle new.
caro show.
caro west.
caro go: 100.
caro show
```

Script 10 : West North

```
|caro|
caro ← Turtle new.
caro show.
caro west.
caro go: 100.
caro show.
caro north.
caro go: 300.
caro show
```

Exercice 1: Autres expériences

Expérimentez de la même manière les commandes `south` et `east` .

1.3 Expérimentons avec la couleur de la trace

Script 11 : Trace

```
|caro|
caro ← Turtle new.
caro trace.
caro west.
caro go: 100.
caro show
```

Script 12 : Angle droit

```
|caro|
caro ← Turtle new.
caro trace.
caro west.
caro go: 100.
caro show.
caro north.
caro go: 300.
caro show
```

Script 13 : Des couleurs

```
|caro|
caro ← Turtle new.
caro trace.
caro west.
caro go: 100.
caro north.
caro go: 100.
caro color: Color red.
caro go: 100.
caro color: Color blue.
caro east.
caro go: 100
```

Remarque. `Color` nécessite aussi une majuscule car c'est aussi une fabrique de couleurs. Ici `Color blue` fabrique la couleur bleue.

1.4 Déductions

Que déduisez-vous quant à la position d'une tortue nouvellement créée :

- sa position est :
- sa direction est :
- sa couleur :

1.5 Messages et état

Comme vous l'ont montré les expériences précédentes, une tortue peut changer de couleur, avancer, laisser des traces sur le sol, ne pas en faire, changer de direction, s'afficher, se cacher graphiquement.... Pour cela, on lui envoie des *messages* qu'elle exécute.

En fait, une tortue est *caractérisée* par les données suivantes qui représentent son *état* à un moment donné.

- **Direction.** La direction d'une tortue est un angle qui indique dans quelle direction la tortue va avancer.
- **Couleur.** La couleur est la couleur avec laquelle une tortue peut dessiner.
- **Ecrire ou pas.** Une tortue peut laisser une trace ou non lorsqu'elle bouge.
- **Place à l'ecran.** Lorsqu'elle bouge une tortue, change de place à l'écran.

Létat d'une tortue est modifié par les messages que nous envoyons à une tortue. Ainsi on peut dire à une tortue de changer de direction, de laisser une trace.... nous allons revenir en détails sur chacun de ces aspects lors des chapitres suivants.

2 Pratiquons

Exercice 2: Mystère

Comme nous l'avons déjà vu avec les scripts précédents, on peut demander à une tortue de laisser une trace (`trace`). Devinez ce que fait le script suivant et vérifiez en l'évaluant, si aviez deviné juste.

Script 14 : Mystère

```
|caro|
caro ← Turtle new.
caro trace.
caro west.
caro go: 100.
caro noTrace.
caro north.
caro go: 100.
caro color: Color red.
caro go: 100.
caro trace.
caro color: Color blue.
caro east.
caro go: 100
```

> **Important!** Lorsque les messsages sont composés de plusieurs mots, on utilise des majuscules au début des mots pour permettre de les identifier plus facilement. Exemple, on écrit `trace` et `noTrace`.

Exercice 3: SOS

En morse, SOS est composé de trois signaux courts suivis de trois signaux longs et de trois signaux courts. Ecrivez un script qui dessine un SOS graphique comme le montre la figure 3.1

Figure 3.1: SOS.

Exercice 4: `trace` and `show`

Pour bien comprendre la différence entre les commandes `trace` and `show`. Ecriez les scripts qui produisent les figures 3.2, 3.3, 3.4 et 3.5.

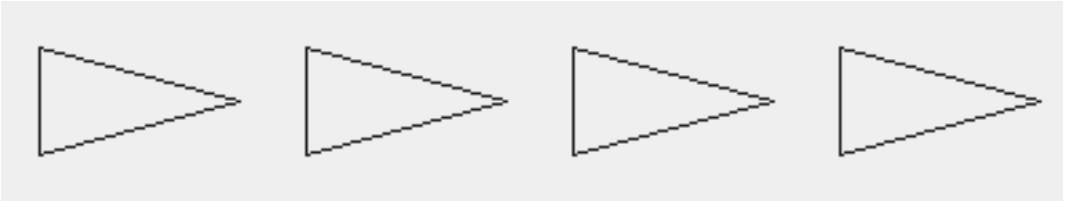


Figure 3.2: 4 shows

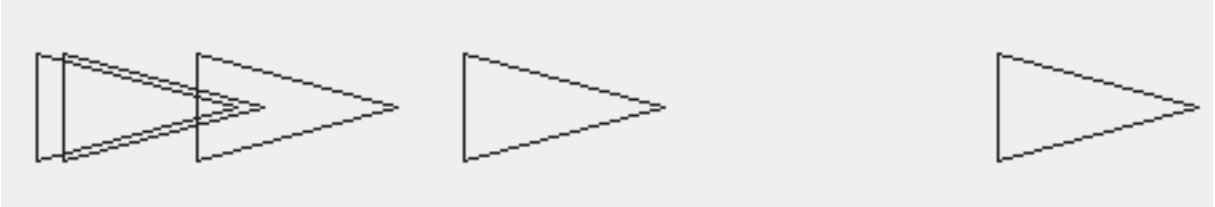


Figure 3.3: Accél’eratons

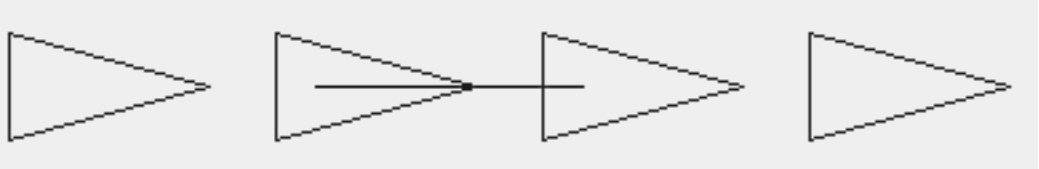


Figure 3.4: Show trace variations numéro 2

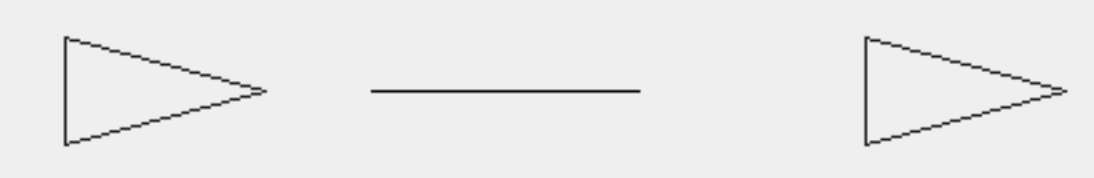


Figure 3.5: Show trace variations numéro 3

3 A retenir

1: Une tortue?!

Une tortue est un objet graphique qui a une *position*, une *direction* et une *couleur*.

2: Créer une tortue

Script 15 : Création d'une tortue

```
|caro|
caro ← Turtle new.
```

3: Messages

Nom	Description	Exemple

`show`	Une tortue s'affiche à l'écran, elle est repérée par un triangle qui par défaut pointe à l'est. **Par défaut une tortue ne s'affiche pas.**	`|caro|`
		`caro ← Turtle new.`
		`caro show.`

`hide`	Efface le triangle qui représente une tortue.
	C'est surtout intéressant pour trouver ses erreurs.
		`|caro|`
		`caro ← Turtle new.`
		`caro show.`
		`caro hide`

Nom	Description	Exemple

`trace`	Une tortue abaisse son stylo pour écrire, alors elle peut se déplacer en laissant une trace. On peut changer la couleur de sa trace (`color:`).	`|caro|`
		`caro ← Turtle new.`
		`caro trace.`
		`caro go: 100`

`noTrace`	Une tortue lève son stylo de l'écran, elle n'écrit plus.	`|caro|`
		`caro ← Turtle new.`
		`caro trace.`
		`caro go: 100.`
		`caro noTrace.`
		`caro go: 100.`
		`caro show`

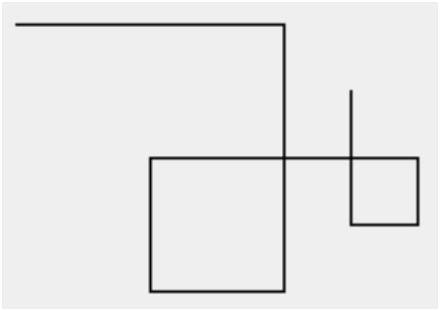
Nom	Description	Exemple

`color:`	Change la couleur de la trace laissée par une tortue	`|caro|`
		`caro ← Turtle new.`
		`caro trace.`
		`caro color: Color red.`
		`caro go: 100`

Nom	**Description**	**Exemple**

`go:`	Une tortue avance d'un nombre de pixels.
	Si l'on veut tracer une droite, avant d'envoyer le message `go:` à une tortue, il faut lui envoyer d'abord le message `trace` et éventuellement le message qui définit la couleur de sa trace. Sinon elle partira sans écrire ou en écrivant en noir!
		`|caro|`
		`caro ← Turtle new.`
		`caro trace.`
		`caro go: 100`

Ne perdons pas le nord



Lors du chapitre précédent, vous avez vu qu'une tortue pointe dans une direction dans laquelle elle avance. Dans ce chapitre, nous allons voir comment on peut changer cette direction.

1 La rose des vents

Exercice 5: Repères

Reproduisez la figure 4.1 et écrivez les directions correspondant aux messages `north`, `south`, `east` et `west` sur le cercle.

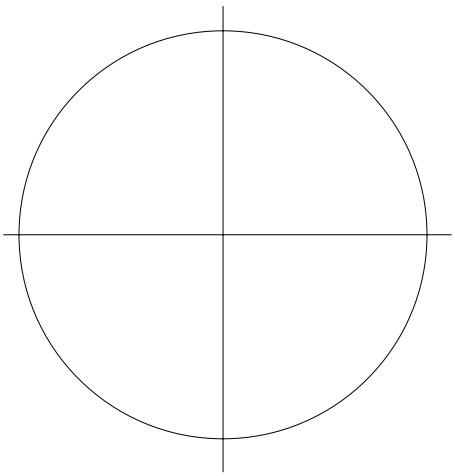


Figure 4.1: Une rose des vents.

2 Tournons!

Une tortue comprend les messages suivants : `north`, `south`, `east` et finalement `west`. Ces messages font pointer une tortue dans des directions prédéfinies.

Stéphane & Florence Ducasse (ducasse@iam.unibe.ch)

Exercice 6: Mystères & Co

Pour les scripts suivants, dessinez ce que fait la tortue selon vous et ensuite tapez le script pour comparer. Notez que vous pouvez bien sûr modifier un script ou en écrire un selon votre envie.

Script 16 : Quels sont ces serpents qui sifflent sur...

```
| caro |
caro ← Turtle new.
caro trace.
caro show.
caro go: 100.
caro north.
caro go: 100.
caro west.
caro go: 100.
caro north.
caro go: 100.
caro east.
caro go: 100
```

Script 17 : Un nombre? Non! Si!

```
| caro |
caro ← Turtle new.
caro trace.
caro west.
caro go: 100.
caro north.
caro go: 100.
caro east.
caro go: 100.
caro north.
caro go: 100.
caro west.
caro go: 100.
caro south.
caro go: 30
```

Script 18 : Nous irons au bois

```
| caro |
caro ← Turtle new.
caro trace.
caro east.
caro go: 100.
caro north.
caro go: 100.
caro west.
caro go: 100.
caro east.
caro go: 100.
caro north.
caro go: 100.
caro west.
caro go: 100
```

Exercice 7: Lettres

Ecrivez un script qui dessine la lettre C.

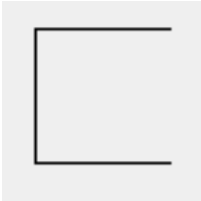


Figure 4.2: Caro et l'alphabet: la lettre C.

Exercice 8: Carré

Ecrivez un script qui fait faire un carré de 100 pixels de côté à une tortue en utilisant uniquement `north`, `south`, `east` et `west`. Modifier ensuite votre script pour faire un carré avec un côté bleu (Color Blue), blanc (Color white), rouge (Color red) et vert (Color green).

Exercice 9: Une Croix

Ecrivez un script qui dessine une croix comme le montre la figure 4.3.

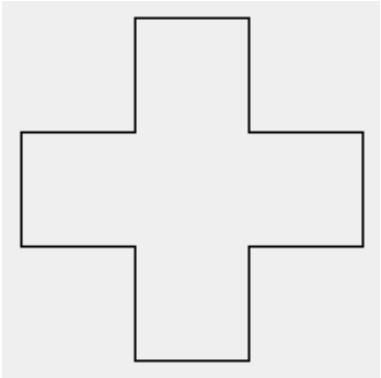


Figure 4.3: Une croix

Exercice 10: `show` n'est pas `trace`

Ecrivez les scripts qui produisent les figures **??** en utilisant les méthodes de `north`, `south`, `east` et `west`.

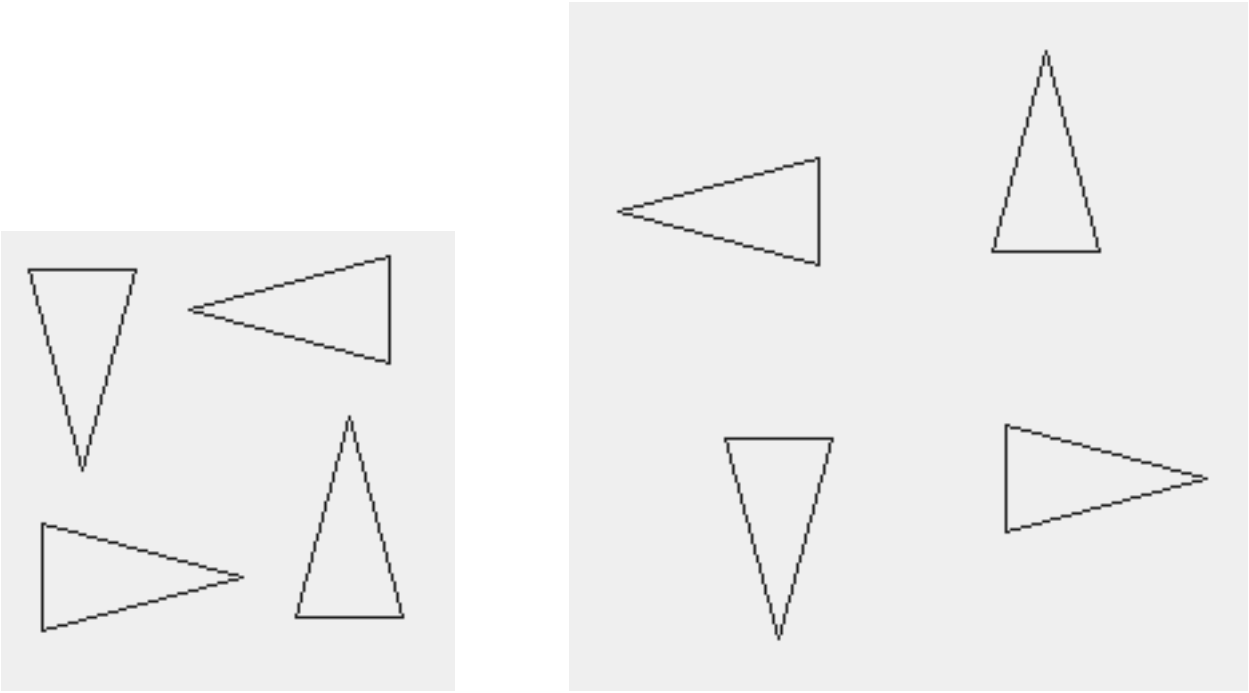


Figure 4.4: Sélection du texte en vue d'une définition de méthode et définition de la méthode `square100`.

3 A retenir

Nom	Description	Exemple

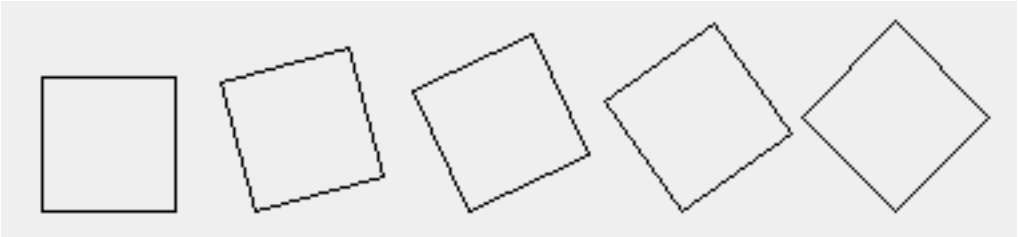
`north`	Ordonne à la tortue de pointer au nord.	`|caro|`
		`caro ← Turtle new.`
		`caro show.`
		`caro north.`
		`caro show.`

`south`	Ordonne à la tortue de pointer au sud.	`|caro|`
		`caro ← Turtle new.`
		`caro show.`
		`caro south.`
		`caro show.`

`west`	Ordonne à la tortue de pointer à l'ouest.	`|caro|`
		`caro ← Turtle new.`
		`caro show.`
		`caro west.`
		`caro show.`

`east`	Ordonne à la tortue de pointer à l'est.	`|caro|`
		`caro ← Turtle new.`
		`caro north.`
		`caro show.`
		`caro east.`
		`caro show.`

Tournons un peu, beaucoup,...



Lors du chapitre précédent, vous avez vu qu'une tortue peut changer de direction. Une tortue comprend les messages suivants : `north`, `south`, `east` et finalement `west`. Ces messages font pointer une tortue dans des directions prédéfinies. Cependant, avoir seulement la possibilité de tourner dans des directions prédéfinies nous limite beaucoup. Par exemple, il est impossible de dessiner les figures ci-dessus d'un carré tombant. En fait on aimerait avoir la possibilité de dire à une tortue de tourner d'un certain angle *à partir de sa direction actuelle*. C'est justement ce que nous allons voir dans ce chapitre.

1 La rose des vents

Exercice 11: Repères

Reproduisez la figure 5.1 et écrivez les directions correspondant aux messages `north`, `south`, `east` et `west` sur le cercle.

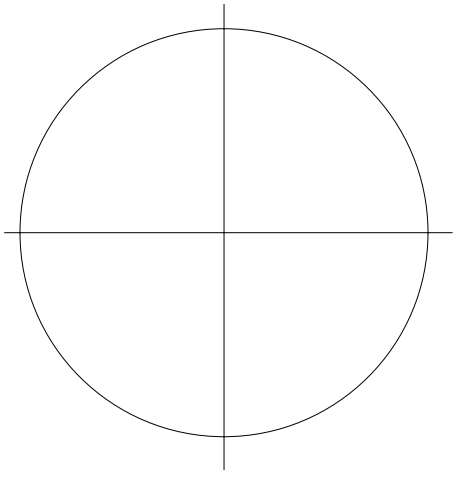


Figure 5.1: Une rose des vents.

2 Tournons!

Une tortue comprend aussi les messages `turnLeft:` et `turnRight:` qui changent la direction de la tortue d'un certain nombre de degrés vers la gauche ou la droite.

Exercice 12: Mystères & Co

Pour les scripts suivants, dessinez ce que fait la tortue selon vous et ensuite tapez le script pour comparer. Notez que vous pouvez bien sûr modifier un script ou en écrire un selon votre envie.

Script 19 : Mystère 1

```
|caro|
caro ← Turtle new.
caro trace.
caro show.
caro go: 100.
caro turnLeft: 45.
caro show.
caro go: 100.
caro turnLeft: 45.
caro show.
caro go: 100
```

Script 20 : Mystère 2

```
|caro|
caro ← Turtle new.
caro trace.
caro show.
caro go: 100.
caro turnRight: 45.
caro show.
caro go: 100.
caro turnRight: 45.
caro show.
caro go: 100
```

Script 21 : Mystère 3

```
|caro|
caro ← Turtle new.
caro trace.
caro show.
caro go: 100.
caro turnRight: 45.
caro show.
caro go: 100.
caro turnLeft: 45.
caro show.
caro go: 100
```

Script 22 : Mystère 4

```
|caro|
caro ← Turtle new.
caro trace.
caro show.
caro go: 100.
caro turnLeft: 30.
caro show.
caro go: 100.
caro turnLeft: 30.
caro show.
caro go: 100.
caro turnLeft: 30.
caro show.
caro go: 100
```

Exercice 13: Elémentaire, mon cher Watson!

Pouvez-vous déduire ce que les commandes `turnLeft:` et `turnRight:` font ? Que pouvez-vous déduire pour les séquences de messages suivantes :

```
caro turnLeft: a.  caro turnLeft: b = caro turnLeft: a ? ?  b

caro turnRight: a.
caro turnLeft: a   =  caro turnLeft: a   ? ?   a

caro turnRight: a.
caro turnLeft: a   =  caro turnLeft:   ??
```

3 Zéro or not Zéro ?

Reproduisez le cercle de la figure 5.2 et repérez sur le cercle en degrés les angles 0, 45, 90, 180, 270 à partir de *l'angle zéro* en utilisant l'instruction `turnLeft:`.

> **Astuce...** L'angle zéro degré correspond à la direction `east` donc fait pointer la tortue vers l'est. Comme la tortue pointe à l'est par défaut il suffit de lui envoyer le message `turnLeft:`.

Faites de même et repérez sur le cercle en degrés les angles 0, 45, 90, 180, 270 à partir de *l'angle 90* en utilisant l'instruction `turnLeft:`.

> **Astuce...** L'angle 90 à partir de l'angle zéro correspond à la direction nord. Vous pouvez donc faire pointer une tortue au nord avant de lui envoyant le message `turnLeft:`.

4 Absolu versus relatif

Nous allons expérimenter ensemble pour bien comprendre la différence entre un changement de direction absolu et un relatif à la direction courante.

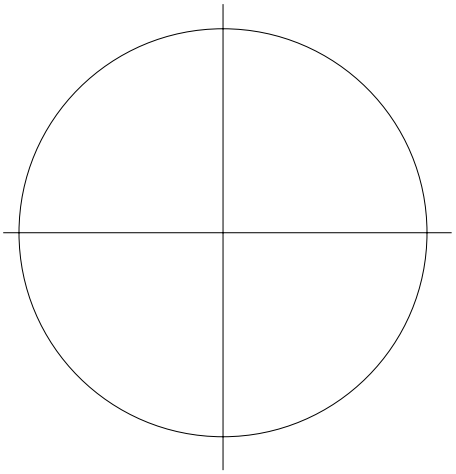


Figure 5.2: Une rose des vents.

Exercice 14: Analyse

En analysant les résultats des scripts suivants, expliquez pourquoi `caro north` n'est pas égal à caro `turnRight: 90` ou à `caro turnLeft: 270`.

Script 23 : North

```
|caro|
caro ← Turtle new.
caro trace.
caro show.
caro go: 100.
caro turnLeft: 90.
caro show.
caro go: 100.
caro color: Color red.
caro north.
caro go: 100.
caro show
```

Script 24 : South

```
|caro|
caro ← Turtle new.
caro trace.
caro show.
caro south.
caro go: 100.
caro turnLeft: 90.
caro show.
caro go: 100.
caro color: Color red.
caro north.
caro go: 100.
caro show
```

Que déduisez-vous quant à la différence entre les instructions `north`, `south`, `east`, `west` et les instructions `turnLeft:` et `turnRight:`?

4.1 Conclusion

- Les messages `north`, `south`, `east` et `west` font pointer une tortue dans une des directions quelle que soit la direction précédente de celle-ci. Ce sont des changements de directions absolus.
- Les messages `turnLeft:` et `turnRight:` font tourner la tortue d'un certain angle par rapport à sa direction courante. Ce sont des changements de directions *relatifs*.

Astuce... L'astuce pour maîtriser les changements de directions relatifs est de se mettre à la place de la tortue en regardant dans la direction vers laquelle elle pointe comme lorsqu'on lit un plan. Si vous voulez tourner à droite utiliser `turnRight:` ou à gauche `turnLeft:`

Important! La direction dans laquelle une tortue pointe après avoir effectué l'instruction `turnLeft:` ou `turnRight:` dépend de sa direction de départ. On dit que ces instructions sont *relatives* à la direction de départ.

5 Exercices

Exercice 15: Carré Absolu

Ecrivez un script qui fait faire un carré de 100 pixels de côté à une tortue en utilisant uniquement `north`, `south`, `east` et `west`.

Exercice 16: Carré Relatif

Faites de même en utilisant exclusivement `turnLeft:` ou `turnRight:`.

Exercice 17: Tournons Maintenant

Pour bien comprendre la différence et son importance, ajoutez la ligne suivante dans vos scripts et analysez les résultats obtenus :

Script 25 : Ajoutez cela

```
|caro|
caro ← Turtle new.
caro trace.
caro turnLeft: 33.  ....
```

Essayez maintenant de reproduire les étapes montrant un carré tombant comme le montre la première figure de ce chapitre.

Exercice 18: Pace

Ecrivez un script qui fait faire le signe de la paix à une tortue.

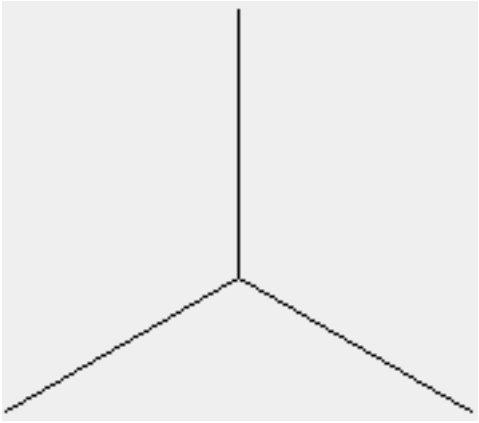


Figure 5.3: Un symbole de paix.

6 A retenir

Nom	Description	Exemple

`turnLeft:`, `turnRight:`	Ordonne à la tortue de tourner à droite ou à gauche d'un certain angle (en degrés) par rapport à la direction courante.	`|caro|`
		`caro ← Turtle new.`
		`caro show.`
		`caro turnLeft: 90.`
		`caro go: 100.`
		`caro show.`
		`caro turnRight: 45`
		`caro go: 300.`
		`caro show`

Bouclons

Dans les chapitres précédents, vous avez appris que vous pouviez envoyer des séquences de messages à une tortue. Vous avez aussi vu que de telles séquences peuvent être très longues et répétitives. Dans ce chapitre, vous allez apprendre comment on peut écrire des séquences de messages à l'aide de boucles ce qui permet d'éviter des répétitions et des erreurs.

1 Une étoile

Nous voulons que caro décrive une étoile c'est-à-dire qu'elle pointe dans les directions: 0, 30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 330, 360. Bref un tour complet en s'affichant après avoir changé de direction c'est-à-dire tous les 30 degres. Les figures suivantes illustrent les premières étapes ainsi que le résultat final. Proposez une solution.

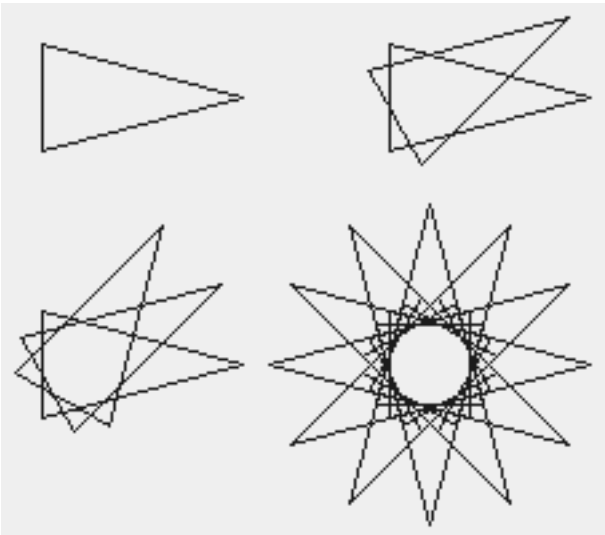


Figure 6.1: Etapes

Votre solution doit ressembler à ceci:

Stéphane & Florence Ducasse (ducasse@iam.unibe.ch)

Script 26 : Etoile sans boucle

```
|caro|
caro ← Turtle new.
caro show.
caro turnLeft: 30.
caro show.
caro turnLeft: 30.
caro show.
caro turnLeft: 30.
caro show.
caro turnLeft: 30.
caro show.
caro turnLeft: 30.
caro show.
caro turnLeft: 30.
caro show.
caro turnLeft: 30.
caro show.
caro turnLeft: 30.
caro show.
caro turnLeft: 30.
caro show.
caro turnLeft: 30
```

C'est un peu laborieux car on doit répéter 12 fois les mêmes instructions. Imaginez si l'on avait voulu faire une centaine de positions!!!

Nous allons donc introduire des *boucles* avec le message `timesRepeat:`. Une *boucle* permet de répéter un certain nombre de fois une séquence de messages.

Dessiner une étoile en utilisant une boucle est maintenant réellement plus court comme le montre le script **Etoile avec boucle** équivalent au script **Etoile sans boucle**.

Script 27 : Etoile avec une boucle

```
|caro|
caro ← Turtle new.
12 timesRepeat: [ caro show.  caro
                turnLeft: 30]
```

Tapez le code précédent pour vérifier et faites d'autres expériences en changeant par exemple le nombre d'itérations ou le code répété.

`12 timesRepeat:` indique que la séquence de messages `caro show.  caro turn:  30` doit être faite 12 fois. Les crochets `[` et `]` servent à délimiter la séquence à répéter.

Important! `n timesRepeat:  [ blabla ]` répète n fois `blabla`

Exercice 19: Carré

Convertissez le script suivant qui n'utilise pas de boucle en un script qui en utilise une.

Script 28 : Carré sans boucle

```
|caro|
caro ← Turtle new.
caro trace.
caro go: 100.
caro turnLeft: 90.
caro go: 100.
caro turnLeft: 90.
caro go: 100.
caro turnLeft: 90.
caro go: 100
```

Exercice 20: Escalier

Ecrivez le script qui produit un escalier comme le montre la figure 6.2.

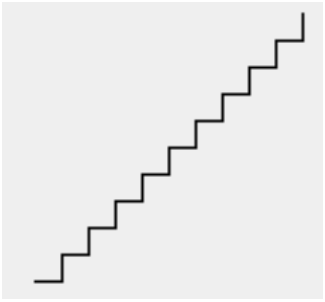


Figure 6.2: Escalier

Vous pouvez commencer sans boucle et ensuite en utiliser.

Exercice 21: Escalier stylisé

En modifiant légèrement le script escalier, générez l'escalier montré dans la figure 6.3.

Figure 6.3: L'escalier stylisé.

Exercice 22: Pyramide

En vous inspirant des exercices précédents écrivez les scripts qui dessinent les pyramides mayas comme le montre la figure 6.4. Notez que la seconde pyramide de la figure 6.4 est symétrique.

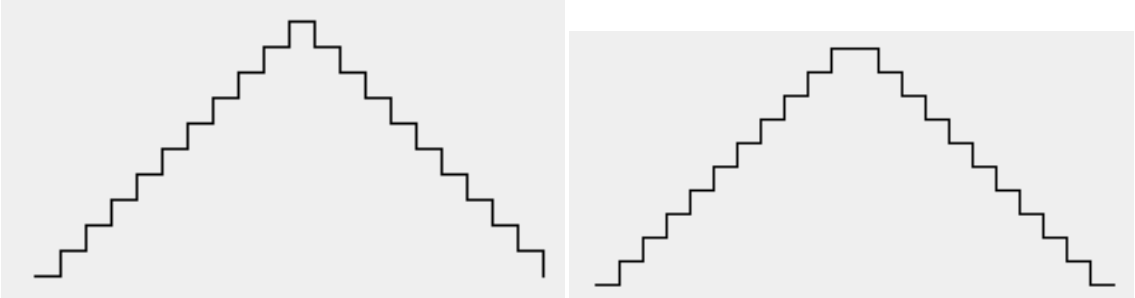


Figure 6.4: Pyramides assymétrique et symétrique.

Exercice 23: Croix

A l'aide d'une boucle écrivez le script qui dessine la croix.

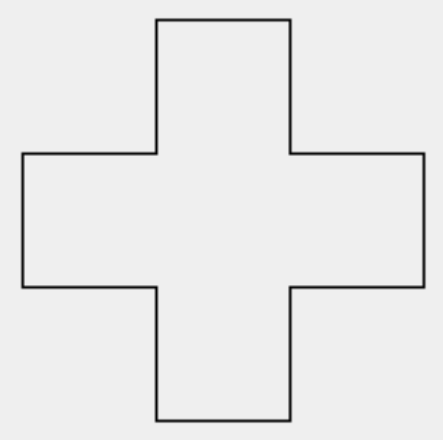


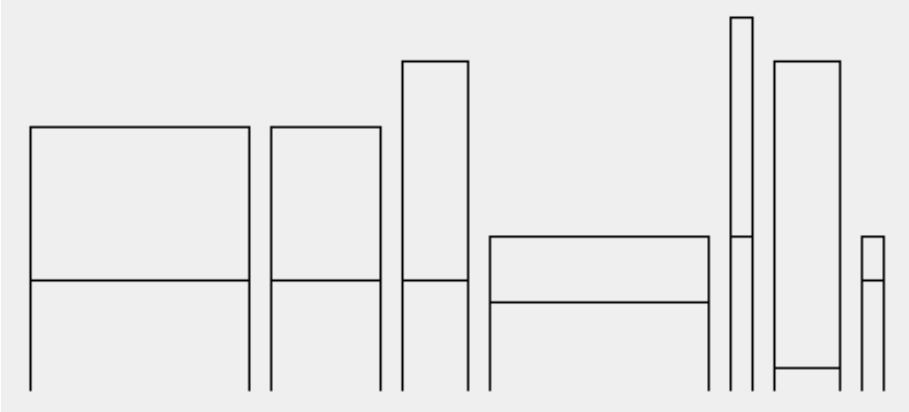
Figure 6.5: Croix.

2 A retenir

Nom	Description	Exemple

`n timesRepeat: []`	Répète n fois une séquence de messages	`|caro|`
		`caro ← Turtle new.`
		`caro trace.`
		`4 timesRepeat: [caro go: 10.`
		`caro turnLeft: 90]`

Mais qu'est-ce qu'une variable?



Nous sommes constamment entrain de donner des noms aux concepts, aux choses ou aux êtres qui nous entourent. Par exemple, nous donnons des noms aux animaux, aux voitures. Ainsi nous associons un mot ou symbole à une chose ou être. Il est alors possible de faire *référence* à cette chose ou d'interagir avec cet être en utilisant ce nom. D'autre part, les noms peuvent temporaires comme lorsque des acteurs jouent un rôle au cinéma ou au théatre. En effet, l'association entre le rôle et l'acteur ne dure que le temps du film.

Dans ce chapitre, nous allons montrer que cette nécessité donner des noms aux choses existe aussi dans la programmation. Nous allons montrer pourquoi on a besoin de variables, ce qu'est une variable et comment on peut écrire des scripts utilisant des variables.

1 Caro écrit

Imaginons que nous voulions que la tortue écrive des mots. Pour cela il faut déjà qu'elle sache écrire des lettres comme A, B,... Commençons donc par là !

1.1 La lettre A

Imaginez que l'on veuille dessiner une lettre A comme le montre la figure 7.1. Une lettre A est caractérisée par une *hauteur*, une *largeur* et la *distance* depuis la base à partir de laquelle on déssine la barre verticale.

Exercice 24: A

Proposez un script qui dessine une lettre A de 100 pixels de hauteur, 70 pixels de large et une barre située à 60 pixels de la base du A.

Vous devez obtenir un script ayant à peu près la même forme que le script suivant :

Script 29 : A de 100 pixels de haut

```
| caro |
caro ← Turtle new.
caro trace.
caro north.
caro go: 100.
caro east.
caro go: 70.
caro south.
caro go: 100.
caro noTrace.
caro west.
caro go: 70.
caro north.
caro go: 60.
caro trace.
caro east.
caro go: 70.
```

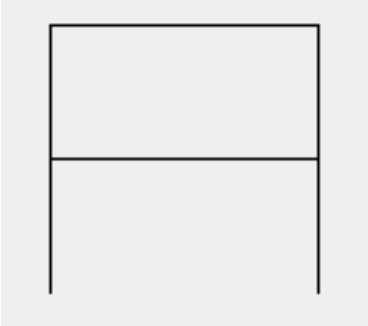


Figure 7.1: La lettre A de 100 pixels de hauteur, 70 pixels de largeur et une barre à 50 pixels.

1.2 Le Monde des As

Maintenant si l'on souhaite dessiner des lettres A de tailles différentes. Nous allons devoir changer *partout* dans le script les valeurs qui représentent la hauteur, la largeur et la place de la barre verticale de la lettre, ici 100, 70 et 60, par d'autres valeurs.

Exercice 25: frAnkenstein.

Changez votre script pour dessiner une lettre A de 200 pixels de hauteur, 90 pixels de largeur et une barre verticale à 70 pixels de la base.

Vous devez obtenir un script semblable au script suivant :

Script 30 : A de 200 pixels de haut

```
| caro |
caro ← Turtle new.
caro trace.
caro north.
caro go: 200.
caro east.
caro go: 90.
caro south.
caro go: 200.
caro noTrace.
caro west.
caro go: 90.
caro north.
caro go: 70.
caro trace.
caro east.
caro go: 90.
```

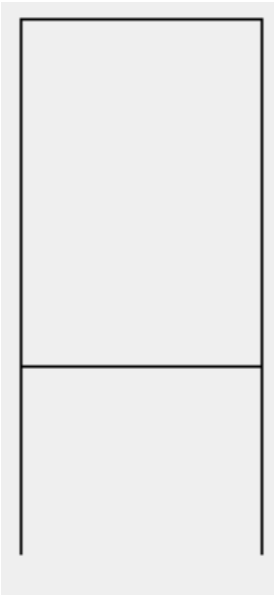


Figure 7.2: La lettre A de 200 pixels de hauteur.

2 Des variables

Comme vous l'avez vu dans l'exercice précédent, changer la taille de la lettre A est assez fastidieux et vous devez faire attention de bien identifier quels nombres correspondent aux caractéristiques de la lettre pour ne pas introduire d'erreurs. En fait, on aimerait

- *définir* la hauteur, la largeur et la distance de la barre de la lettre une fois pour toutes,
- pouvoir y faire *référence*,
- et éventuellement en modifier la valeur.

C'est exactement ce qu'une variable permet de faire! Etrange non?

> **Important!** Une variable est un *nom* auquel on *associe une valeur*. On peut ensuite y faire *référence et obtenir* la valeur associée à cette variable. Il est aussi possible de *modifier* la valeur associée à cette variable pour lui associe une nouvelle valeur.

2.1 Déclaration d'une variable

Pour utiliser une variable, il faut d'abord la *déclarer*, c'est-à-dire donner le nom de la variable dont on va se servir. En Squeak, une variable se déclare en mettant un nom entre des barres verticales `||`. Le script 31 montre la déclaration de trois variables `hauteur`, `largeur` et `barre`. En Squeak, les noms de variables peuvent aussi contenir des nombres.

Script 31 : Déclarations de variables

```
| hauteur largeur barre |
```

2.2 Associer une valeur à une variable

Avant de pouvoir se servir d'une variable c'est-à-dire de pouvoir faire référence à la valeur que l'on lui a associée, il faut bien évidemment lui *associer une valeur*.

En Squeak, pour associer une valeur à une variable on utilise la flêche ←ou `:=`. Le script 32 montre qu'après avoir défini des variables `hauteur`, `largeur` et `barre`) on leur associe des valeurs.

Ici on associe 100 à la variable `hauteur`, 70 à la variable `largeur` et 60 à la variable `barre`.

Script 32 : Associer des valeurs aux variables

```
| hauteur largeur barre |
hauteur ← 100.
largeur ← 70.
barre ← 60.
```

Vocabulaire Associer une valeur à une variable s'appelle aussi *affecter une valeur à une variable*. On dit aussi *initialiser une variable* quand c'est la première fois que l'on lui associe une valeur.

2.3 Utiliser des variables

Pour utiliser la valeur d'une variable, il suffit d'écrire la variable dans un script. Par exemple, le script 33 montre la déclaration de la variable `malongueur`, l'affectation du nombre 100 à la variable `malongueur` et son utilisation. En effet, la ligne `caro go: longueur` signifie que l'on envoie le message `go:` à la tortue nommée `caro` avec comme longueur la *valeur* associée à la variable `malongueur` ici 100. La tortue avance donc de 100 pixels.

Script 33 : Utilisation de la variable `longueur`

```
1   | caro malongueur |
2   caro ← Turtle new.
3   malongueur ← 100.
4   caro trace.
5   caro go: malongueur
```

2.4 A propos de caro.

Vous avez deviné juste, `| caro |` définit bien une variable de nom `caro` qui a pour valeur une tortue parmi d'autres.

Script 34 : Et caro!

```
1   | caro |
2   caro ← Turtle new.
3   caro trace.
```

En fait ligne 1, on déclare que l'on va utiliser une variable nommée `caro`. Ensuite ligne 2 on associe à la variable `caro` une tortue qui est créée par l'expression `Turtle new`. Ensuite ligne 3 on envoie des messages à la tortue en accédant à la valeur (la tortue) associée à la variable `caro`.

2.5 XYZ! Qu'est-il dit?

Il est important de bien comprendre que l'on est libre de donner le nom que l'on veut à une variable. Nous verrons plus loin qu'il exixte des exceptions à cette règle. Ainsi l'exécution des deux scripts suivants produits le même effet. Dans chacun des deux scripts les variables sont affectées avec les mêmes valeurs et utilisées de la même manière.

Script 35 : C'est simple

```
| caro malongueur |
caro ← Turtle new.
malongueur ← 100.
caro trace.
caro go: malongueur
```

Script 36 : Qu'est-ce qu'il dit?

```
| x z |
x ← Turtle new.
z ← 100.
x trace.
x go: z
```

La très grande différence est que le premier script est facilement compréhensible, le second ne l'est pas. Il est donc très important de donner des noms aux variables qui décrivent à quoi elles vont servir. Et ceci même pour vos propres scripts. Il est fort probable que vous deviez les relires dans six mois et vous aimerez les lire facilement. Imaginez si nous avions écrit tous les scripts de ce livre en utilisant `x` et `y` pour toutes les variables. Cette notion de nommage est *extrêmement* importante en informatiques.

3 De retour à nos écrits

Nous allons utiliser des variables dans le script de la lettre A et voir comment cela fonctionne. Reprenez le script 29 et introduisez la variable `hauteur`.

3.1 Analyse d'un oubli

Dans le script suivant nous avons introduit la variable `hauteur` et nous lui avons affectée la valeur 120. Mais nous n'obtenu pas une lettre de 120 pixels de hauteur. Pourquoi?

Script 37 : Une hauteur non utilisée

```
| caro hauteur |
caro ← Turtle new.
hauteur ← 120.
caro trace.
caro north.
caro go: 100.
caro east.
caro go: 70.
caro south.
caro go: 100.
caro noTrace.
caro west.
caro go: 70.
caro north.
caro go: 50.
caro trace.
caro east.
caro go: 70.
```

Parce que nous n'avons pas utilisé la variable, le script utilise les anciennes valeurs, donc le A obtenu n'est pas une lettre A de 120 pixels mais un A de 100 pixels.

3.2 Utiles variables

Le script suivant montre une bonne utilisation de la variable `hauteur`. Après avoir déclaré, donné une valeur à la variable *hauteur*, nous l'utilisons lors des messages envoyés à une tortue.

Script 38 : Une hauteur bien utilisée

```
| caro hauteur |
caro ← Turtle new.
hauteur ← 120.
caro trace.
caro north.
caro go: hauteur.
caro east.
caro go: 70.
caro south.
caro go: hauteur.
caro noTrace.
caro west.
caro go: 70.
caro north.
caro go: 50.
caro trace.
caro east.
caro go: 70.
```

Exercice 26: Varions

Faites varier les variables que peuvent prendre la variable `hauteur`. Il suffit pour cela de changer la valeur affectée à la variable.

Dans le script suivant nous avons introduit toutes les variables qui décrivent la lettre A.

Script 39 : A avec toutes les variables

```
| caro hauteur largeur barre|
caro ← Turtle new.
hauteur ← 120.
largeur ← 70.
barre ← 50.
caro trace.
caro north.
caro go: hauteur.
caro east.
caro go: largeur.
caro south.
caro go: hauteur.
caro noTrace.
caro west.
caro go: largeur.
caro north.
caro go: barre.
caro trace.
caro east.
caro go: largeur
```

Exercice 27: Varions II. The return

Amusez-vous à donner des valeurs différentes aux variables `hauteur`, `largeur` et `barre`.

4 Lions des variables

4.1 De belles lettres

Une lettre pour être reconnaissable et agréable à lire doit respecter certaines proportions, c'est-à-dire que les longueurs qui la décrivent ne sont pas données au hasard mais ont des relations entre elles. Par exemple décidons que la largeur doit 2/3 de la hauteur et que la barre doit être placée à la moitié de la hauteur.

Nous pouvons exprimer ces relations de la manière suivante :

Script 40 : Lettre A avec valeurs des variables liées

```
| caro hauteur largeur barre|
caro ← Turtle new.
hauteur ← 120.
largeur ← 120 * 2 / 3.
barre ← 120 / 2.
caro trace.
caro north.
caro go: hauteur.
caro east.
caro go: largeur.
caro south.
caro go: hauteur.
caro noTrace.
caro west.
caro go: largeur.
caro north.
caro go: barre.
caro trace.
caro east.
caro go: largeur
```

En effet, la valeur d'une variable peut être déterminée par une expression plus ou moins complexe. Ici la valeur de la variable `largeur` est le résultat rendu par l'expression 120 / 2 * 3 c'est-à-dire 80. De la même façon, la valeur de la variable `caro` est le résultat de l'expression `Turtle new` qui a pour valeur une tortue.

Exercice 28: Gulliver

Changez la hauteur de la lettre, essayez de dessiner une lettre de 50 pixels et une de 120.

Comme vous le voyez cette solution n'est pas encore satisfaisante car il faut encore à chaque fois changer la valeur de la hauteur et mettre à jour cette valeur dans les expressions qui calculent les valeurs des variables `largeur` et `barre`.

Quelle est votre solution ? Essayez de trouver une solution à ce problème. Elle est basée sur l'utilisation de la variable `hauteur` dans les expressions qui calculent les valeurs des deux autres variables.

4.2 Utilisons et lions des variables

En fait, la valeur d'une variable peut dépendre de la valeur d'une autre variable. Ainsi le script peut être réécrit comme suit :

Script 41 : Lettre A avec variables liées

```
| caro hauteur largeur barre|
caro ← Turtle new.
hauteur ← 120.
largeur ← hauteur * 2 / 3.
barre ← hauteur / 2.
caro trace.
caro north.
caro go: hauteur.
caro east.
caro go: largeur.
caro south.
caro go: hauteur.
caro noTrace.
caro west.
caro go: largeur.
caro north.
caro go: barre.
caro trace.
caro east.
caro go: largeur
```

> **Important!** La seule contrainte que l'on a est que la valeur d'une variable entrant dans la définition de la valeur d'une autre variable doit être connue lorsque la valeur de la nouvelle variable est calculée.

Ici, la variable `hauteur` doit avoir une valeur lorsque l'on définit la valeur de la variable `largueur` ou `barre`. Voir en 6 pour plus de détail.

5 Avez-vous vraiment compris?

Vous avez vu comment on utilise des variables, vous pouvez tester vous connaissances avec les exercices suivants.

Exercice 29: Expériences avec des variables

Evaluez les scripts suivants pour vous rendre compte que l'on affecte une valeur à une variable, que l'on peut manipuler cette valeur (comme faire des calculs) et changer cette valeur. Essayez de deviner de quelle valeur la tortue va avancer.

Script 42 : Mystères variables

```
|caro size|
caro ← Turtle new.
caro trace.
size ← 100.
caro go: size
```

Script 43 : Mystères variables

```
|caro size|
caro ← Turtle new.
caro trace.
size ← 100.
caro go: size + 100.
```

Script 44 : Mystères variables

```
|size caro |
caro ← Turtle new.
caro trace.
size ← 100.
size + 100.
caro go: size
```

Script 45 : Mystères variables

```
|size caro |
caro ← Turtle new.
caro trace.
size ← 100.
size ←  200.
caro go: size
```

Script 46 : Mystères variables

```
| size caro |
caro ← Turtle new.
caro trace.
size ← 100.
size ← 200.
size ← 100.
caro go: size
```

Script 47 : Mystères variables

```
| size caro |
caro ← Turtle new.
caro trace.
size ← 100.
size ← size + 100.
caro go: size
```

Script 48 : Mystères variables

```
| size caro |
caro ← Turtle new.
caro trace.
size ← 100.
size ← size + 100.
size + 200.
caro go: size
```

Script 49 : Mystères variables

```
| size caro |
caro ← Turtle new.
caro trace.
size ← 100.
size ← size + size.
caro go: size
```

6 Vos possibles erreurs

Utiliser des variables donne beaucoup de puissance mais demande un peu d'attention. En effet, il possible d'obtenir des erreurs.

6.1 Valeur par défaut d'une variable

Si l'on n'associe pas une valeur à une variable, les programmes risquent de ne pas fonctionner car on risque d'utiliser des variables ayant de mauvaises valeurs. Par défaut, Squeak associe la valeur `nil` à une variable. `nil` représente les valeurs indéfinies avec lesquelles on ne peut rien faire. `nil` est juste une valeur pour que l'on sache si l'on a donné une valeur à une variable ou pas. Le script suivant ne fonctionne pas. Pourquoi ?

Script 50 : Un script ne fonctionnant pas

```
| caro maLongueur|
caro ← Turtle new.
caro go: maLongueur.
```

Dans le script 50, on n'associe pas de valeur à la variable `maLongueur`, donc Squeak associe `nil`. Ensuite, on envoye le message `go:` à une tortue mais la variable `maLongueur` n'a pas pour valeur un nombre mais `nil`. Donc le système fait une erreur.

6.2 Retour sur l'association d'une valeur à une variable

Notez que la valeur par défaut d'une variable est `nil`. En Squeak, c'est une valeur qui nous permet de savoir si une variable à déjà eu une valeur. Il est donc très important que vous associez toujours une valeur à une variable. On dit que l'on *initialise* la variable.

Par exemple, si vous tapez et évaluez :

Script 51 : Où size n'est pas initialisée

```
| size caro |
caro ← Turtle new.
caro go: size + 10
```

Vous obtenez une erreur. En effet `size` n'a pas été initialisée donc sa valeur vaut `nil` et ajoutez 10 à `nil` n'a aucun sens. Il faut donc au minimum que size ait une valeur qui soit un nombre si l'on veut lui ajouter un autre nombre ou si l'on veut que la variable puisse être utilisée avec la méthode `go:`. En effet, `go:` fait avancer une tortue un certain *nombre* de fois.

Astuce... Vérifiez toujours que les variables sont initialisées avant d'en utiliser la valeur.

6.3 Variables liées

> **Important!** La seule contrainte que l'on a est que la valeur d'une variable entrant dans la définition de la valeur d'une autre variable doit être connue lorsque la valeur de la nouvelle variable est calculée.

Par exemple, dans le script suivant, la variable `hauteur` doit avoir une valeur lorsque l'on définit la valeur de la variable `largueur`.

Script 52 : Hauteur et largeur

```
| caro hauteur largeur |
caro ← Turtle new.
hauteur ← 120.
largeur ← hauteur * 2 / 3.
caro trace.
caro go: largeur.
```

Le script suivant est incorrect car la valeur de la variable `hauteur` n'est pas connue lorsque l'on définit la valeur de la variable `largeur`. Lorsque la valeur de la variable `largeur` est calculée, `hauteur` n'est pas connue donc Squeak lui associe `nil`. Ensuite, Squeak essaye de multiplier `hauteur` donc `nil` par 2 et cela n'a pas de sens donc produit une erreur.

Script 53 :

```
|hauteur largeur barre|
largeur ← hauteur * 2 / 3.
hauteur ← 100
```

6.4 A retenir

> **Important!** Une variable est un *nom* auquel on *associe une valeur*. On peut ensuite y faire *référence et obtenir* la valeur associée à cette variable. Il est aussi possible de *modifier* la valeur associée à cette variable pour lui associe une nouvelle valeur.

Nom	Description	Exemple
| hauteur |	Déclare que l'on va utiliser la variable `hauteur`	
hauteur ←100	Associe une valeur à la variable `hauteur`	

caro go: hauteur + 100	Accède la valeur de la variable `hauteur`	`|caro hauteur|`
		`caro ← Turtle new.`
		`hauteur ← 100 + 100.`
		`caro go: hauteur`

Bouclons dans l'inconnu

Au chapitre **??**, nous avons montré que l'utilisation de boucles est utile car elle permet de ne pas répéter du code. Cela est vrai mais nous avions toujours la possibilité de copier le code plusieurs fois. Maintenant nous allons montrer pourquoi nous avons *vraiment* besoin de boucles.

Nous avons besoin de boucles car nous avons besoin de répéter des actions un certain nombre de fois *sans connaître ce nombre lors de l'écriture du code*. En effet, il arrive que l'on ait besoin de demander à l'utilisateur de choisir un nombre ou bien de tirer un nombre au hasard. Remarquez que sans boucle il est impossible d'avoir une solution car copier du code ne fonctionne pas.

1 Un peu d'interactivité

Imaginons que l'on veuille dessiner un escalier dont le nombre de marches soit donné par l'utilisateur du programme comme le montre le script Escalier Interactif. Sans boucle, il est impossible d'écrire un script ayant le bon nombre de marches car nous ne savons lorsque nous sommes entrain de le définir ce que l'utilisateur choisira.

Figure 8.1: On demande le nombre de marches de notre future escalier avec comme valeur initiale 10

Script 54 : Escalier Interactif

```
|caro nombre|
caro ← Turtle new.
caro trace.
nombre ← (FillInTheBlank
            request: 'Combien de marches'
            initialAnswer: '10') asNumber.
nombre timesRepeat: [ caro go: 10.
                      caro north.
                      caro go: 10.
                      caro east]
```

L'exécution du script produit les figures 8.2 et 8.3.

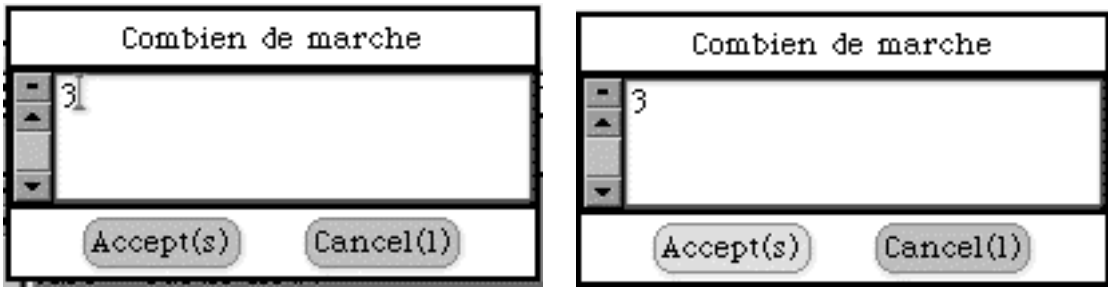


Figure 8.2: L'utilisateur demande 3 marches et termine son choix

Figure 8.3: Le script dessine trois marches

1.1 Expliquons FillInTheBlank

L'expression `FillInTheBlank...` crée une petite fenêtre qui demande des valeurs à l'utilisateur et les rend au programme comme le montrent les figures 8.1 et 8.2.

Elle est créée par le message `request:initialAnswer:` envoyé à `FillInTheBlank` [1]. Le message nécessite de donner après `request:` un texte (une chaine de caractères) indiquant à l'utilisateur ce que l'on attend de lui et après `initialAnswer:` une valeur par défaut qui est aussi une chaine de caractères. Si l'utilisateur est satisfait et ne change pas la valeur par défaut elle sera rendue comme réponse.

```
(FillInTheBlank request: 'Combien de marche' initialAnswer: '10')
```

De chaine à nombre. Le résultat rendu par la fenêtre est une chaine de caractères qui représente la valeur donnée par l'utilisateur. Ici comme nous avons besoin d'un nombre pour la boucle nous convertissons la chaine en nombre à l'aide de la commande `asNumber`.

Référence En Squeak, une *chaine de caractères*, une suite de caractères s'écrit à l'aide des quotes simples. Les simples quotes délimitent la chaine de caractères.

Par exemple, les expressions suivantes sont des chaines de caractères.

`'a'` est une chaine composée d'un seul caractère.

`'blabla'` est une chaine composée de plusieurs caractères.

`'combien de marches'` est une chaine composée de plusieurs caractères avec des espaces.

`'ne fais pas l"idiot'` si l'on veut inclure dans une chaine de caractères des quotes simples, il faut les doubler.

`'10'` est une chaine composée du caractère `1` suivi du caractère `0`.

---

[1] `FillInTheBlank` est une fabrique de ces petites fenêtres.

Enseigner aux tortues

Jusqu'à présent vous avez défini des scripts dans lesquels vous *avez créé une tortue* à laquelle vous avez envoyé des messages. Utiliser des scripts est une approche simple et pratique mais *extrêmement* limitée. En effet, on ne peut pas appeler un script comme les autres messages. La seule solution est de recopier un script à l'intérieur d'un autre. De plus, on ne peut pas invoquer le même script sur différentes tortues sans le recopier. La solution à ces problèmes existe. Il s'agit de définir des *méthodes*. Une méthode représente une *séquence de messages à laquelle on donne un nom* et que l'on peut utiliser comme n'importe quelles méthodes que les tortues connaissent déjà (`go:`, `turnLeft:`...). Ainsi une méthode peut être utilisée par différentes tortues.

Dans cette partie, vous allez donc apprendre comment définir de nouvelles méthodes pour les tortues. Pour cela vous allez aussi apprendre à manipuler un nouvel éditeur.

1 Le problème

Lors des exercices précédents, vous avez vu que l'on peut envoyer des messages à des tortues afin de créer des figures complexes. Par exemple, vous avez écrit un script qui doit ressembler au script suivant (script 55) ordonnant à une tortue de tracer un carré de 100 pixels de côté :

Script 55 : square100

```
| caro |
caro ← Turtle new.
caro trace.
4 timesRepeat:[caro turnLeft: 90.
               caro go: 100].
caro noTrace
```

Cependant, à chaque fois que nous voulons dessiner un carré nous avons à écrire la même séquence de messages, encore et toujours. Cela pose les problèmes suivants :

- il est *pénible de réécrire de longues* scripts.
- Plus important, certaines scripts peuvent être *complexes* et on peut introduire des *erreurs* lors de leur recopie.
- Comme le montre le script ci-dessous (script 56), il est aussi possible de créer plusieurs tortues en même temps et de leur faire executer la même séquence de messages. Mais à ce moment là il faut *répéter* les scripts.

Script 56 : square100 pour plusieurs tortues: Répétitions

```
|caro maur|
caro ← Turtle new.
maur ← Turtle new.

maur color: Color red.
maur go: 300.

maur trace.
4 timesRepeat: [maur turnLeft: 90.
                maur go: 100].
maur noTrace.

caro trace.
4 timesRepeat: [caro turnLeft: 90.
                caro go: 100].
caro noTrace
```

Ce que l'on voudrait pouvoir faire ! En fait, nous aimerions pouvoir *définir une fois pour toutes* une séquence de messages, lui *donner un nom* et que *toutes* les tortues puissent l'exécuter.

Le script suivant (script 57) montre le même code que précédemment avec cette approche, c'est-à-dire en imaginant que `square100` représente la séquence de messages qui dessine un carré de 100 pixels de côté.

Script 57 : Réutilisation de la méthode `square100`

```
| caro maur |
caro ← Turtle new.
maur ← Turtle new.

maur color: Color red.
maur go: 300.

maur square100.
caro square10
```

Dans ce script on crée deux tortues mais on ne répéte donc pas la définition de la méthode, on appelle la méthode `square100` comme les autres méthodes que les tortues connaissent.

Notez que le nom de la méthode est `square100` mais on aurait pu l'appeler `square`. Ici `100` fait partie du nom de la méthode.

2 Définir vos propres méthodes

Nous allons maintenant expliquer en détail comment vous allez définir de nouvelles méthodes comme `square100`. Pour cela vous allez vous servir d'un nouvel outil : un éditeur de code. Dans un prochain chapitre, nous vous expliquerons l'éditeur en détail.

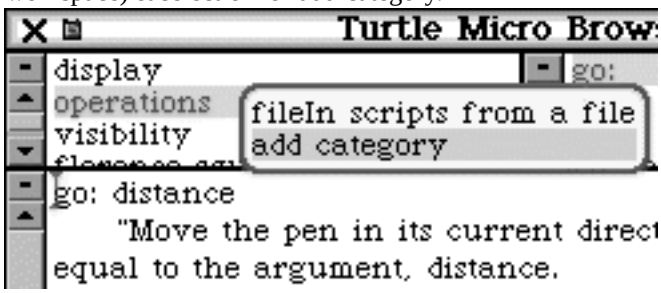
Comme vous allez maintenant définir de nouvelles méthodes, nous allons tout d'abord créer une *catégorie*. Une catégorie est comme un dossier dans lequel vous allez mettre les méthodes que vous définissez ceci vous permettra d'accéder et de sauver facilement vos méthodes.

2.1 Utiliser l'éditeur pour créer une nouvelle catégorie

Nous allons créer une catégorie dans laquelle vous allez ensuite définir plusieurs méthodes. Nous allons ajouter une nouvelle catégorie nommée `'square'` (donnez lui le nom que vous voulez mais que le nom indique clairement les méthodes qu'elle va contenir). Suivez les étapes suivantes :

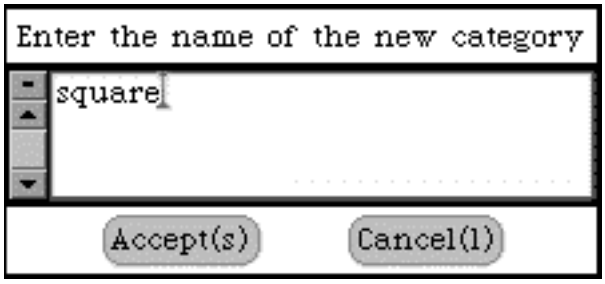
Etape 1: Menu

Faites apparaître le menu en selectionnant une catégorie (et en pressant sur la touche commande comme vous le faisiez dans le workspace) et sélectionnez add-category.



Etape 2: Nommer la catégorie

L'éditeur vous demande alors le nom de la catégorie que vous voulez créer. Entrez le nom de votre catégorie.



L'éditeur crée alors la catégorie et la sélectionne automatiquement, il est prêt pour définir de nouvelles méthodes. Le texte qu'il vous présente est un aide-mémoire qui vous rappelle la structure d'une méthode.

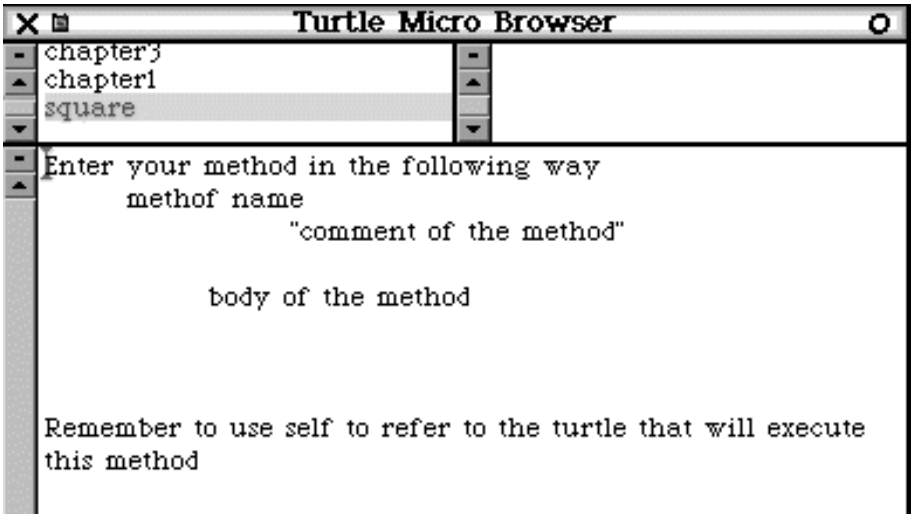


Figure 9.1: Nouvelle catégorie créée.

3 Utiliser l'éditeur pour définir une méthode

La méthode `square100` que nous avons présentée en début de leçon se définit de la manière suivante. Nous allons expliquer en détail chacune des parties du code et ensuite vous la définirez à l'aide de l'éditeur (figure 9.2).

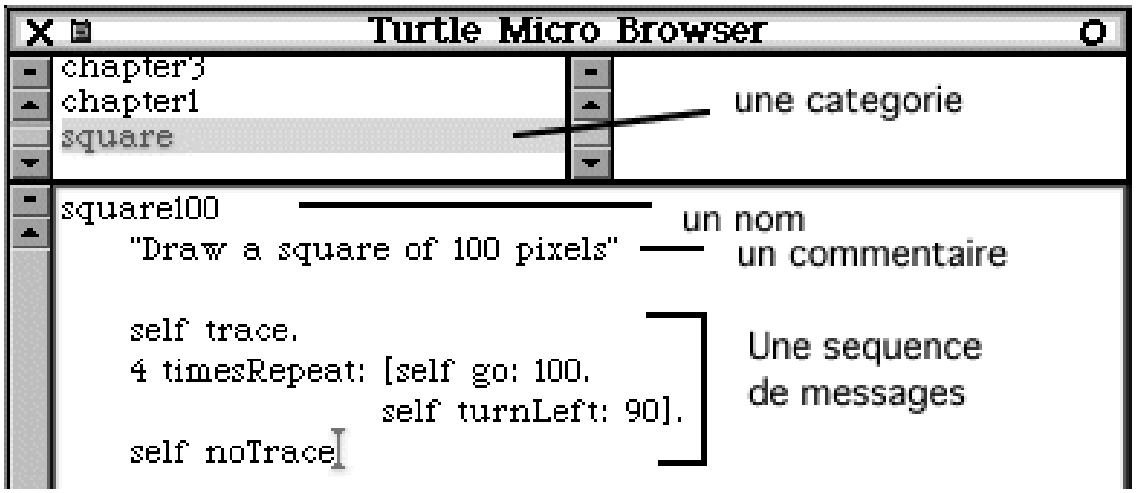


Figure 9.2: Une méthode : un nom, un commentaire et une séquence de messages.

Méthode

```
square100
   "Draws a square of 100"

   self trace.
   4 timesRepeat: [ self go: 100.
                    self turnLeft: 90].
   self noTrace
```

3.1 Structure d'une méthode.

Comme le montre la figure 9.1, le texte dans l'éditeur nous indique comment se structure une méthode. Une méthode se décompose en trois parties :

Method name (obligatoire) : le nom complet de la méthode, ici `square100`.

> **Important!** Le nom d'une méthode doit représenter ce que fait la méthode. Donnez toujours des noms intelligents à vos méthodes.

Comment of the method (facultatif) : un commentaire décrivant ce que fait la méthode. En général, un commentaire est très utile pour comprendre ce que fait une méthode sans avoir à comprendre sa définition (séquences de méthodes). Les commentaires s'écrivent en anglais courant et sont délimités par des `""`. Pour `square100` nous avons:

```
"Draws a square of 100 pixels"
```

> **Important!** Un commentaire ne dit pas comment est dessiné le carré mais ce que fait la méthode: c'est-à-dire dessine un carré! Pour savoir comment une méthode fonctionne il est plus simple de lire le code!

Body of the method (obligatoire) : il s'agit de la séquence de messages que l'on veut associer au nom de la méthode. Ici la définition de la méthode est la séquence de messages suivante :

```
self trace.
4 timesRepeat: [self turnLeft: 90.
                self go: 100].
self noTrace
```

Noter que la séquence de messages n'inclut pas la création de la méthode. En effet, la méthode peut être invoquée sur différentes tortues. Nous expliquons plus loin ce qu'est la variable `self` et pourquoi il n'est pas nécessaire de la définir comme les autres variables. Noter aussi que l'on décale les commentaires et le code de la méthode un peu vers la droite afin de pouvoir la voir clairement.

3.2 Création de la méthode

Pour créer la méthode `square100` suivez les étapes suivantes :

Etape 1: Tapez la méthode

Sélectionnez le texte (figure 9.3) dans la partie basse de l'éditeur (figure **??**) et tapez le texte suivant :

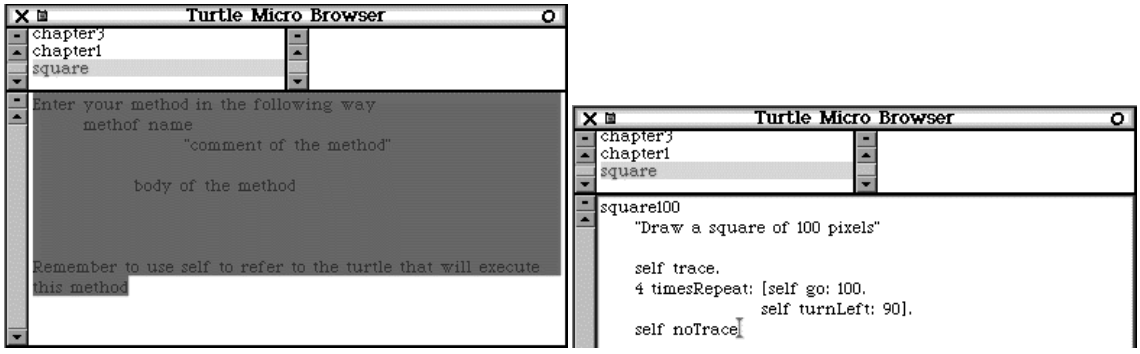


Figure 9.3: Sélection du texte en vue d'une définition de méthode et définition de la méthode `square100`.

Etape 2: Compiler la méthode

Faites apparaître le menu et sélectionnez le choix **Accept : Compile the Method** (figure 9.4). Cela a pour effet de définir `square100` comme une nouvelle méthode que toutes les tortues peuvent comprendre. C'est lors de la compilation d'une méthode que Squeak vérifie si vous n'avez pas tapé des erreurs syntaxiques, comme oublié un point, oublié de définir une variable ou de donner un argument dans un message. Ce n'est que lorsqu'une méthode est compilée qu'elle peut être utilisée.

Etape 3: Confirmation de l'éditeur

L'éditeur vous montre que la méthode est bien définie en l'affichant dans la liste de méthodes contenues dans la catégorie courante ici `'square'`. Ici celle-ci est ne contient que la méthode `square100`.

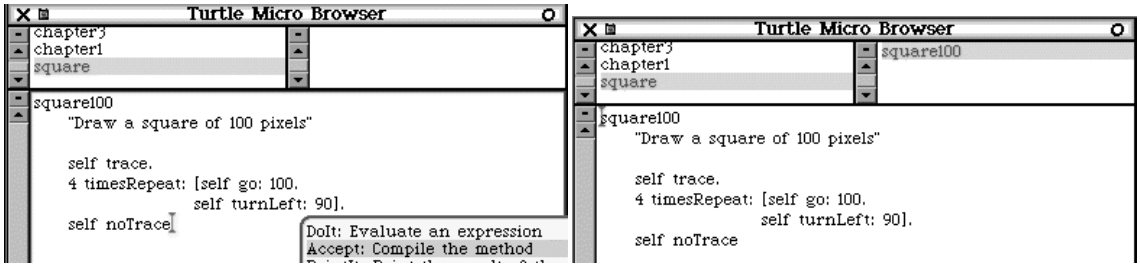


Figure 9.4: Compilation de la méthode `square100` et l'éditeur confirme que la éthode `square100` est définie.

Etape 4: Vérification du fonctionnement de la méthode.

Une fois la méthode `square100` définie, vérifiez qu'une tortue l'exécute correctement en évaluant par exemple le script 57 ou script ci-dessous :

Script 58 : Appel de square100.

```
|caro|
caro ← Turtle new.
caro square100
```

Tester la méthode est très important. Squeak permet d'intégrer des exemples dans la définition même des méthodes et nous reviendrons sur cet aspect dans un prochain chapitre.

4 Précisions sur la définition d'une méthode

4.1 La variable `self`

Dans le code de la méthode `square100`, nous avons utilisé la variable `self`. En effet, quand nous définissons une nouvelle méthode, celle-ci pourra être exécutée par différentes tortues comme `caro` et `maur` dans notre exemple ci-dessus. Il nous faut donc un moyen de désigner la tortue qui exécutera effectivement la méthode que nous sommes en train de définir et les méthodes qui la composent.

La variable `self` est définie en Squeak à cet effet. La variable `self` est une variable spéciale qui représente toujours la *tortue recevant le message* et nous n'avons pas à la définir à l'aide des `||`.

Méthode

```
square100
   "Draws a square of 100 pixels"

   self trace.
   4 timesRepeat: [self turnLeft: 90.
                   self go: 100].
   self noTrace
```

Par exemple dans `square100` ci-dessus, nous définissons que la tortue qui exécutera la méthode `square100`, devra en fait :

- tout d'abord laisser une trace,
- répéter 4 fois touner de 90 degrés et avancer de 100 pixels et
- ne plus laisser de trace.

Notez que pour dessiner un carré il s'agira de la même tortue. Cependant cette méthode pourra être exécutée par différentes tortues.

Remarque. Dans tous les scripts que vous avez définis lors des leçons précédentes vous avez toujours (1) créé une tortue et (2) envoyé à cette tortue des messages afin qu'elle les exécute. Il est donc normal de ne pas inclure la création d'une tortue dans la définition d'une nouvelle méthode puisque celle-ci sera envoyée à une tortue déjà existante.

Important! `self` représente la tortue qui recevra le message dans lequel `self` apparaît. `self` est une variable spéciale donc vous n'avez pas besoin de la définir à l'aide de `||`.

Astuce... Pensez que la tortue qui exécutera la méthode que vous définissez s'appelle `self`.

5 Votre première méthode

Exercice 30: `thing`

Définissez la *méthode* `thing` dont nous vous donnons la définition sous forme de script et dont voici une utilisation.

Script 59 : `thing`

```
| caro |
caro ← Turtle new.
caro trace.
caro go: 100.
caro turnRight: 90.
caro go: 100.
caro turnLeft: 90.
caro go: 50.
caro turnRight: 90.
caro go: 50.
caro turnRight: 90.
caro go: 100.
caro turnRight: 90.
caro go: 25.
caro turnRight: 90.
caro go: 25.
caro turnRight: 90.
caro go: 50
```

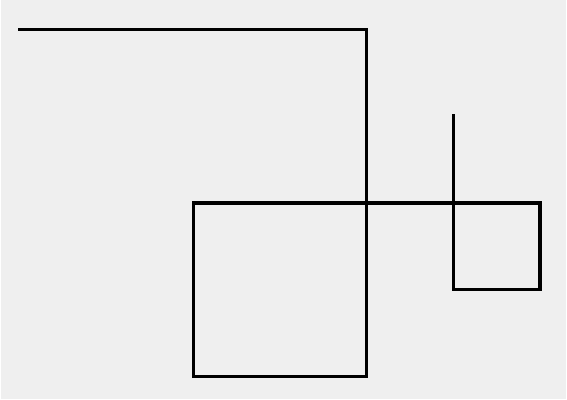


Figure 9.5: `thing` : une drôle de chose.

Script 60 : Utilisation de la méthode `thing`

```
| caro |
caro ← Turtle new.
caro trace.
caro thing
```

6 A Retenir

Important! Dans une méthode, `self` représente la tortue qui recevra le message dans lequel self apparaît. Il n'est pas nécessaire de la définir à l'aide des `||`.

6.1 Vocabulaire

Nous nommons :

un envoi de message : lorsque l'on demande à une tortue de faire une action.

```
caro go: 100
```

Ici on envoie le message `go: 100` à caro. Elle avance de 100 pixels.

un script : une séquence d'envois de messages incluant la définition d'une tortue. Le script crée une tortue puis lui fait exécuter un carré de 100 de côté. Seule la tortue créée par ce script peut l'exécuter.

```
|caro|
caro ← Turtle new.
caro trace.
4 timesRepeat: [caro turnLeft: 90.
                caro go: 100]
```

une méthode : un nom donné à un ensemble de messages que toutes les tortues sont capables d'exécuter. Exemple : `go:`, `color:`, `square100` sont des méthodes que toutes les tortues peuvent exécuter. Une méthode est composée d'un nom, d'un commentaire et d'une définition.

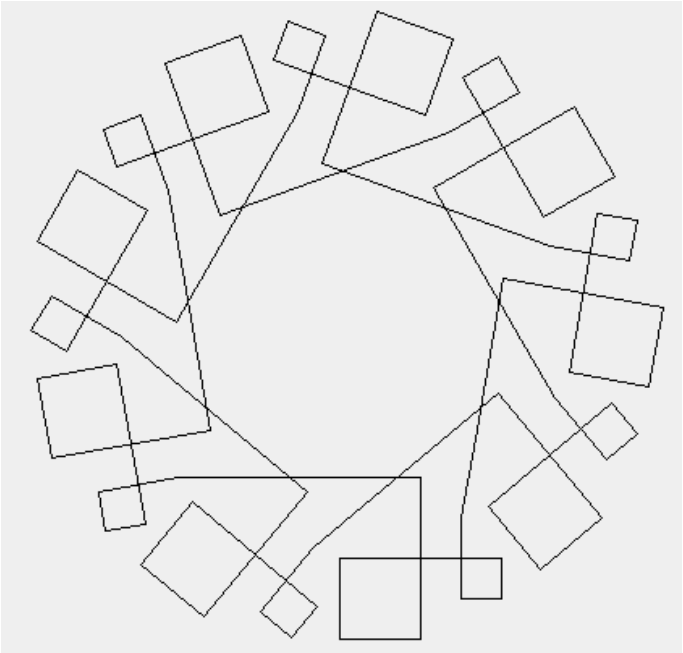
Méthode

```
square100 "Draws a square of 100 pixels"

    self trace.
    4 timesRepeat: [self turn:90.
                    self go: 100].
    self noTrace
```

une catégorie : un nom pour un ensemble de méthodes.

Composons!



Lors du précédent chapitre, nous avons montré comment nous définissons de nouvelles méthodes que toutes les tortues peuvent exécuter ensuite. Nous avons montré que définir de nouvelles méthodes est intéressant car :

- cela évite de devoir réécrire des scripts et d'introduire des erreurs,
- et que les méthodes peuvent être utilisées par différentes tortues.

L'autre avantage que nous allons explorer dans ce chapitre et dans la suite du cours est la possibilité de composer des méthodes, c'est-à-dire qu'une méthode est définie en utilisant d'autres méthodes. Pouvoir composer des méthodes est extrêmement important car cela permet de définir une méthode en fonction d'autres méthodes sans avoir à connaître comment ces autres méthodes sont définies.

1 Exemple: la méthode `square100`

Composer des méthodes est assez naturel et n'est pas nouveau. C'est ce que nous avons fait lors du chapitre précédent lorsque nous avons défini des méthodes ! Par exemple, la méthode `square100` est définie en faisant appel aux méthodes `turnLeft:`, `trace`, `noTrace`... Elle est donc composée d'autres messages et nous ne faisons pas attention de savoir comment ces messages étaient eux-mêmes définis.

Stéphane & Florence Ducasse (ducasse@iam.unibe.ch)

Méthode
```
square100

    self trace.
    4 timesRepeat: [self turnLeft:90.
                    self go: 100].
    self noTrace
```

2 Des choses et d'autres: `thing`

Lors du chapitre 9, vous avez défini la méthode `thing` qui dessinait la figure 10.1 dont nous vous rappelons la définition de la méthode `thing`. Si vous avez oublié de la sauver vous devez la redéfinir à l'aide de l'éditeur.

Méthode
```
thing
    "draws a thing"

    self go: 100.
    self turnRight: 90.
    self go: 100.
    self turnRight: 90.
    self go: 50.
    self turnRight: 90.
    self go: 50.
    self turnRight: 90.
    self go: 100.
    self turnRight: 90.
    self go: 25.
    self turnRight: 90.
    self go: 25.
    self turnRight: 90.
    self go: 50
```

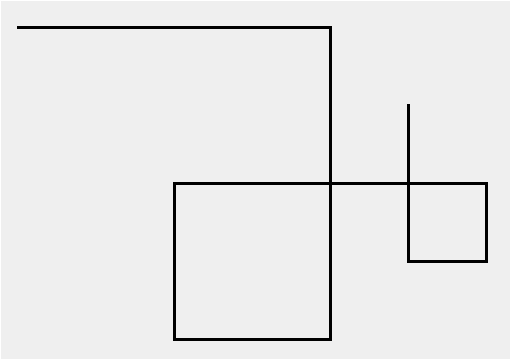


Figure 10.1: `thing` : une drôle de chose.

Exercice 31: Motifs

`completeThing`. Définissez la méthode `completeThing` qui appelle quatre fois `thing` et réalise la figure complète et dont on montre un appel dans la figure 10.2.

`completeThing2`. Définissez la méthode `completeThing2` qui dessine la figure 10.3 et dont on donne la définition.

`multipleThings`. Définissez la méthode `multipleThings` qui dessine la figure 10.4.

2.1 Qu'avons-nous mis en évidence ? la réutilisation!

Comme vous le voyez avec les méthodes `completeThing`, `completeThing2` et `multipleThings`, la méthode `thing` n'est définie qu'une seule fois, mais est réutilisée plusieurs fois par les méthodes `completeThing`, `completeThing2` et `multipleThings` pour produire des dessins différents. La déf-

Script 61 : Utilisation de la méthode completeThing

```
| caro |
caro ← Turtle new.
caro trace.
caro completeThing
```

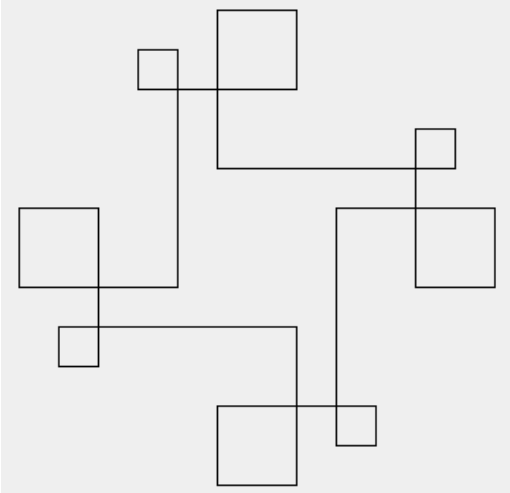


Figure 10.2: `completeThing` : un motif complet.

Méthode

```
completeThing2 "Draws another com-
plete thing"

    9 timesRepeat: [self thing.
                    self turnRight: 10.
                    self go: 50]
```

Script 62 : Utilisation de la méthode completeThing2

```
| caro |
caro ← Turtle new.
caro trace.
caro completeThing2
```

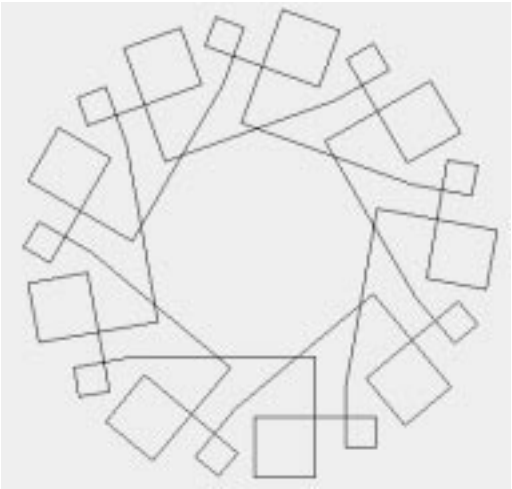


Figure 10.3: `completeThing2` : un autre motif complet.

Méthode

```
multipleThings
    "Draws another thing based picture

    8 timesRepeat: [self thing.
                    self turnLeft: 45.
                    self go: 100]
```

Script 63 : Utilisation de la méthode `multipleThings`

```
| caro |
caro ← Turtle new.
caro trace.
caro multipleThings
```

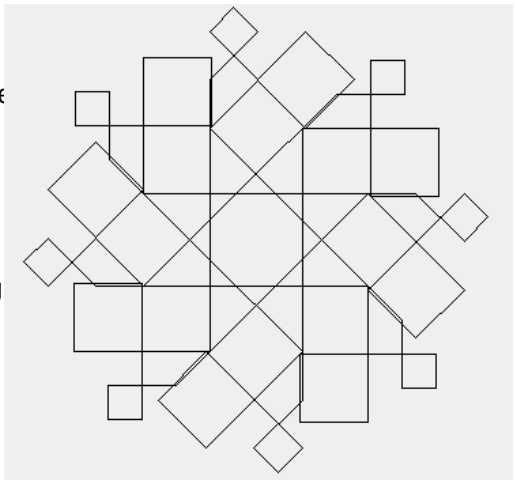


Figure 10.4: `multipleThings` : un autre motif encore plus fou.

inition de `thing` comme une méthode nous a donc permis de : (1) la définir une seule fois, (2) réutiliser cette méthode dans différents contextes et (3) de ne pas introduire d'erreurs lors des redéfinitions.

2.2 Qu'avons-nous mis en évidence ? l'abstraction

Si vous regardez attentivement la définition de la méthode `multipleThings`, vous constatez qu'elle est définie en termes de la méthode `thing` qui elle est aussi définie en termes d'autres méthodes comme `go:`, `turnLeft:`...

Méthode

```
multipleThings
   "Draws another thing based picture"

   8 timesRepeat: [self thing.
                   self turnLeft: 45.
                   self go: 100]
```

En fait, une méthode plus complexe est souvent définie en faisant appel à des méthodes plus simples et ainsi de suite. Il est essentiel de comprendre que lorsque l'on définit la méthode `multipleThings` nous ne voulons pas savoir comment la méthode `thing` est définie, nous voulons juste l'utiliser ! De cette façon, il est plus simple pour nous de définir des méthodes en fonction d'autres sans avoir à retenir leur propre définition. Nous avons juste à nous rappeler ce qu'elles font et non comment elles le font. On dit que nous nous abstrayons de leur définition, c'est pourquoi on dit aussi qu'une méthode définit une abstraction.

Pour bien vous montrer cela nous avons réecrit la méthode `multipleThings` sans appel à la méthode `thing`. Au lieu de cela nous avons recopié directement la définition de `thing` (en italique) à l'intérieur de `multipleThings`.

Méthode

```
multipleThingsSansUtiliserThing
   "Draws another thing based picture"

   8 timesRepeat: [self go: 100.
                   self turnRight: 90.
                   self go: 100.
                   self turnRight: 90.
                   self go: 50.
                   self turnRight: 90.
                   self go: 50.
                   self turnRight: 90.
                   self go: 100.
                   self turnRight: 90.
                   self go: 25.
                   self turnRight: 90.
                   self go: 25.
                   self turnRight: 90.
                   self go: 50thing.
                   self turnLeft: 45.
                   self go: 100]
```

Comparer la méthode `multipleThings` et `thingsSansUtiliserThing`. `multipleThings` apparaît comme bien plus simple. Imaginez maintenant ce que pourrait être la méthode `thingsSansU-`

`tiliserThing` si l'on montrait le code des méthodes `go:`, `turnLeft:` et `turnRight:`.

> **Important!** On peut définir une méthode en fonction d'autres méthodes et ceci sur plusieurs niveaux sans avoir à comprendre comment ces méthodes ont elles-mêmes été construites.

3 Des carrés partout

Vous allez maintenant créer des figures nouvelles à l'aide de la méthode `square100`.

3.1 Carrés et variations

Exercice 32: Un gros carré.

Définissez les méthodes `box` et `separatedBox` qui produisent les dessins de la figure 10.5.

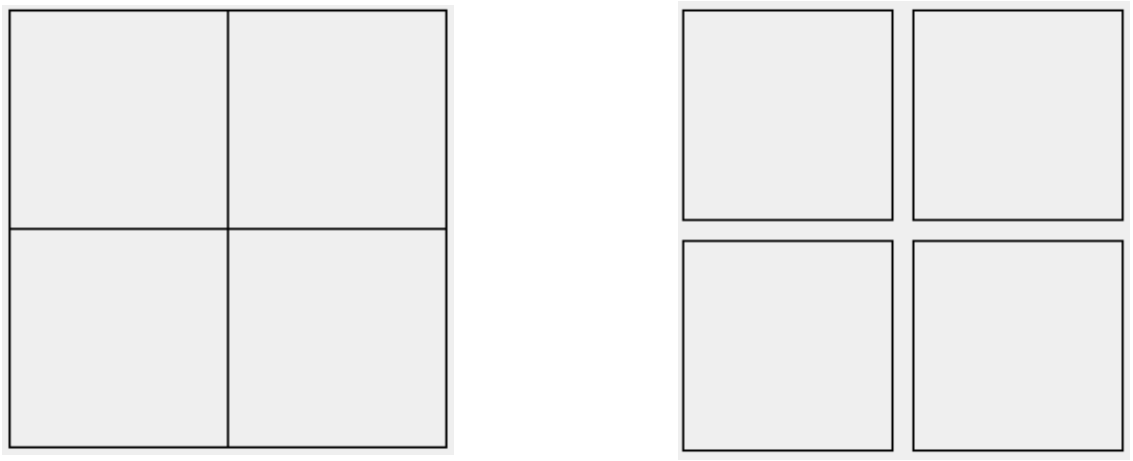


Figure 10.5: Un carré et un autre.

Exercice 33: Variantes

Changez le script ci-dessus pour obtenir différentes figures. Vous pouvez changer l'angle, faire avancer la tortue d'une certaine longueur....Essayer de reproduire la figure 10.6.

Exercice 34: `star`.

En utilisant la méthode `box`, expérimentez et définissez le script qui produit l'étoile figure 10.6.

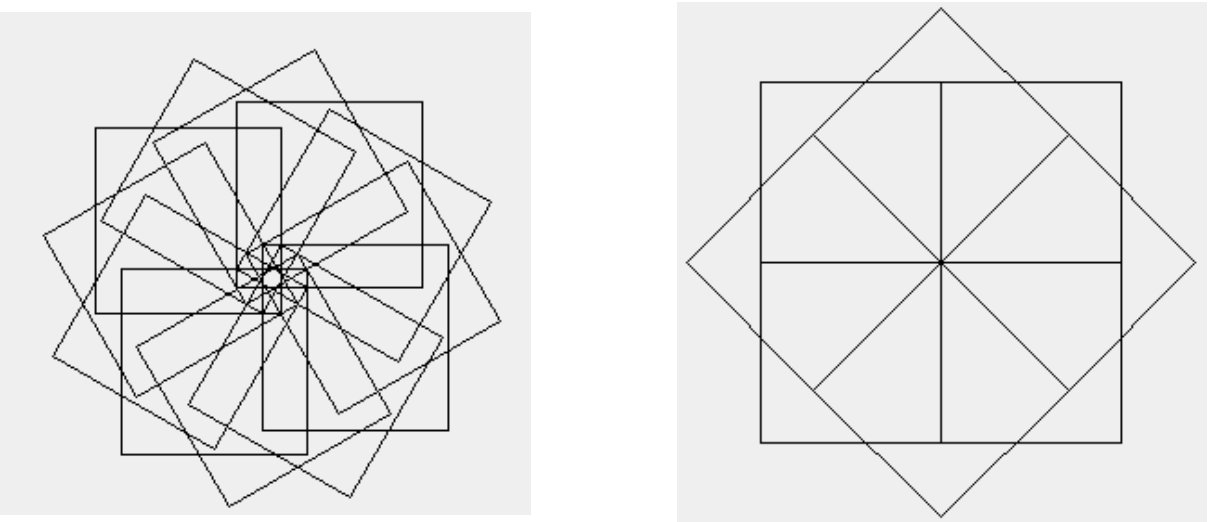


Figure 10.6: Variations autour d’un carré et une étoile.

Pour mieux composer graphiquement

1 A propos de `square100` et de l'utilisation de `trace`/`noTrace`

Comme le montre la définition de la méthode `square100`, nous avons choisi d'inclure les opérations `trace` et `noTrace` à l'intérieur de celle-ci. Une des raisons est que cela nous permet ensuite de répéter plusieurs fois de suite un carré sans avoir à se soucier si la tortue laisse une trace après l'exécution de la méthode `square100`.

Méthode

```
square100

    self trace.
    4 timesRepeat: [self turnLeft:90.
                    self go: 100].
    self noTrace
```

Ainsi si nous définissons la méthode `square100WithoutTrace` et que nous l'utilisons dans le script suivant, nous obtenons la figure 11.1 qui montre que nous avons oublié de mettre un `noTrace`. Par contre, si nous invoquons la méthode `square100`, nous n'avons pas ce problème comme le montre la figure 11.2. C'est pourquoi inclure dans la définition de la méthode les appels aux méthodes `trace` et `noTrace` nous évite de répéter ces instructions lors de l'utilisation des méthodes.

Figure 11.1: Résultat de la mauvaise utilisation de la méthode squareWithoutTrace : il y a des traits entre les carrés.

Méthode

```
square100WithoutTrace

      4 timesRepeat: [self turnLeft: 90.
                      self go: 100]
```

Script 64 : Utilisation de la méthode square100WithoutTrace avec oubli d'appel à `noTrace`

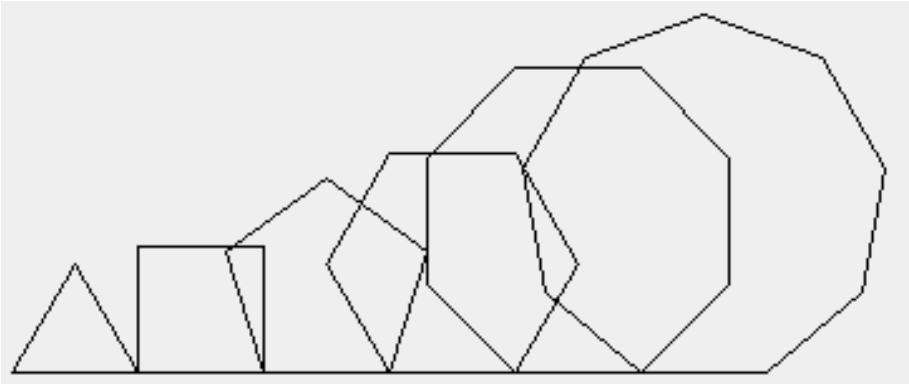
```
|caro|
caro ← Turtle new.
caro trace.
4 timesRepeat: [caro square100WithoutTrace.
                caro go: 120]
```

Script 65 : `square100` menant à un dessin correct

```
|caro|
caro ← Turtle new.
4 timesRepeat: [caro square100.
                caro go: 120]
```

Figure 11.2: Résultat de l'utilisation de la méthode `square100`.

Argumentons!



1 Problème

La méthode `square100` est limitée car la taille du carré créé est fixe. Bien sûr, nous pourrions définir les méthodes : `square25`, `square15`, `square10`, `square65`, `square77`, `square100`, `square115` et `square12`, mais cela n'est pas une approche satisfaisante car nous ne pouvons appeler une méthode que lorsque nous l'avons préalablement définie. Nous voulons maintenant définir une méthode qui permet de spécifier la taille du carré à dessiner.

Comme vous l'avez vu dans la leçon sur les variables, le script permettant de définir un carré ainsi que sa taille est le suivant :

Script 66 : squareWithSize

```
|caro size|
caro ← Turtle new.
size ← 10.
caro trace.
4 timesRepeat: [caro turnLeft: 90.
                caro go: size].
caro noTrace
```

Nous allons nous servir de ce script pour définir la méthode `square:` qui dessine un carré d'une taille donnée. Tout d'abord le script suivant montre comment la méthode `square:` peut être utilisée une fois définie.

Script 67 : Utilisation de la méthode `square:`

```
| caro |
caro ← Turtle new.
caro square: 10.
caro go: 300.
caro square: 20
```

Le script ci-dessus dessine deux carrés : un carré avec une taille de 10 pixels et un autre avec une taille de 20 pixels de côté.

Notez que comme la méthode `go:` la méthode `square:` a besoin d'un argument pour s'exécuter et donc elle se termine par un double-point. En fait, une méthode a besoin d'un double-point pour chaque argument, ainsi `square:` *`size`* `withColor:` *`aColor`* a deux arguments qui sont spécifiés par des double-points.

2 Méthode et arguments

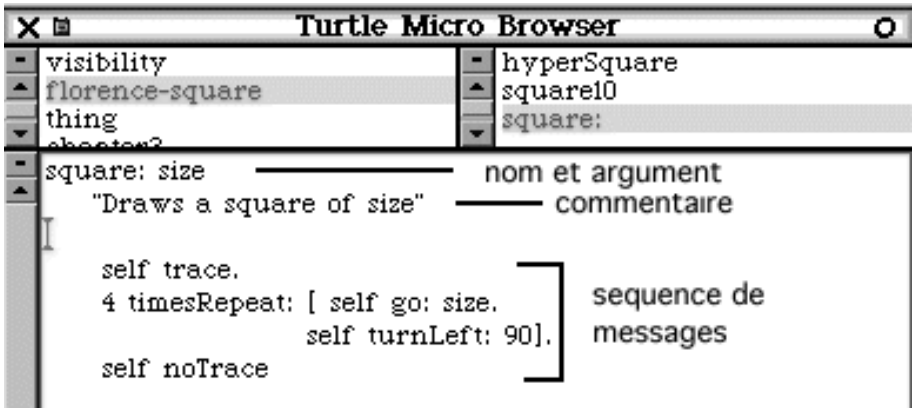


Figure 12.1: La méthode `square:`.

La méthode `square:` est définie, comme le montre la figure 12.1 et la définition ci-après, en utilisant un argument nommé `size` qui représente la taille du carré que l'on dessine.

Méthode

```
square: size
   "Draws a square of size"

   self trace.
   4 timesRepeat:[self go: size.
                  self turnLeft: 90].
   self noTrace
```

2.1 Mais qu'est-ce qu'un argument ?

Dans la méthode `square:`,

- `size` est une variable un peu spéciale appelée un *argument*,
- `size` fait partie du nom de la méthode. En fait le nom de la méthode est `square:` et le nom de l'argument est `size`. C'est pourquoi `size` n'a pas à être déclarée à l'aide de `||`.
- comme une variable, un argument représente une valeur. Cependant un argument représente une valeur qui *sera donnée lorsque le message sera envoyé*. Par exemple, `size` représente la valeur 10 et 20 dans le premier script 67.
- Un argument est défini implicitement dans le nom de la méthode, il n'est pas nécessaire de le déclarer avec `||`. Ici `square: size` définit l'argument `size`.
- Contrairement aux variables que l'on doit initialiser à une certaine valeur avant de les utiliser, un argument prend automatiquement la valeur spécifiée par l'utilisation de la méthode (nommée aussi appel de la méthode), donc il n'a pas à être initialisé. Dans le script suivant, l'argument `size` de la méthode `square:` va prendre respectivement les valeurs 10 et 20.

Script 68 : Utilisation de la méthode `square:`

```
| caro |
caro ← Turtle new.
caro square: 10.
caro go: 300.
caro square: 20
```

2.2 Une explication

On peut imaginer que l'appel d'une méthode revient à donner de nouvelles valeurs aux variables et arguments comme le montre la *simulation* ci-dessous.

Si l'on prend la définition de méthode `square:` comme nous l'avons déjà montrée, l'utilisation ou appel de la méthode revient à associer de nouvelles valeurs à la variable `self` et à l'argument `size`. Ici lors des appels `caro square: 10` et `caro square: 20`, les variables et arguments de `square:` vont prendre les valeurs : *self = caro, size = 10* et *self = caro, size = 20*. On obtient en remplaçant les variables et arguments le code suivant:

Illustration pour le premier appel de la méthode `square:`.

```
caro square: 10.
          self = caro
          size = 10

        caro trace.
        4 timesRepeat: [ caro go: 10.
caro turnLeft: 90].
        caro noTrace
```

Illustration pour le second appel de la méthode `square:`.

```
caro square: 20.
          self = caro
          size = 20

        caro trace.
        4 timesRepeat: [caro go: 20.
                        caro turnLeft: 90].
        caro noTrace
```

3 Pratiquons!

Exercice 35: Hexagone

Le script **hexagone** dessine un hexagone. Définissez la méthode `hexagone:` qui permet de spécifier la taille des côtés de l'hexagone.

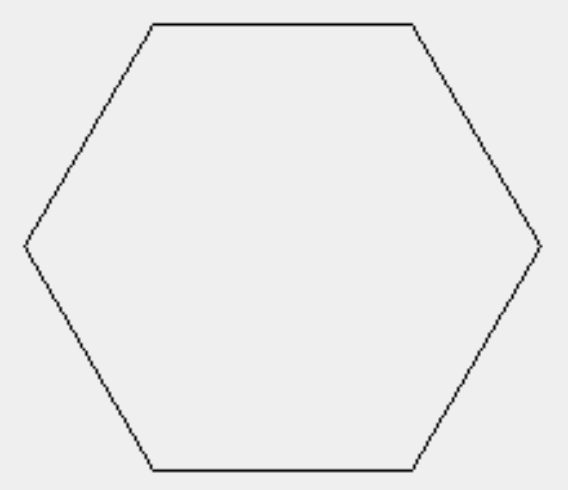


Figure 12.2: Un hexagone de taille 100

Script 69 : Hexagone

```
|caro taille|
caro ← Turtle new.
taille ← 100.
caro trace.
6 timesRepeat: [caro turn: 60.
                caro go: taille].
caro noTrace
```

Script 70 : Utilisation de la méthode `hexagone:`

```
|caro |
caro ← Turtle new.
caro hexagone: 100
```

4 Plusieurs arguments

Le nombre d'arguments d'une méthode n'est pas limité. En effet, une méthode peut avoir zéro argument (comme `east`, `noTrace`, `square100`), un argument (comme `square:`, `go:`, `turnLeft:`...) mais aussi deux, trois ou quatre arguments.

4.1 Conventions

Comme le montre le nom de la méthode `square:` ou `go:`, le nom d'une méthode ayant un argument se termine par un `:`. En fait, il faut un caractère `:` avant chaque argument. Exemple :

- `square: size` indique que la méthode `square:` nécessite une valeur pour s'exécuter.
- `square: size withColor: aColor` indique la méthode `square:withColor:` nécessite deux arguments, le premier étant la taille du carré et le second sa couleur.

Exercice 36: Un carré de couleur

Définissez la méthode `square: aNumber withColor: aColor` qui dessine un carré de taille et de couleur données.

Script 71 : Utilisation de la méthode square:withColor:

```
| caro |
caro ← Turtle new.
caro square: 150 withColor: Color red.
caro go: 200.
caro square: 50 withColor: Color blue
```

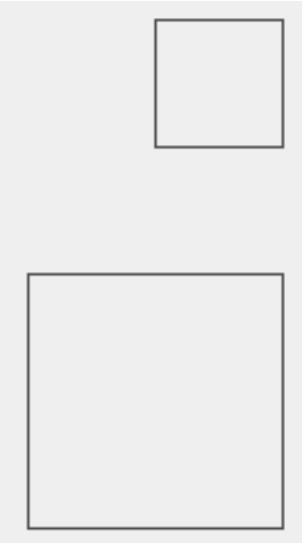


Figure 12.3: Deux carrés de couleurs.

Exercice 37: Polygones....

Définissez la méthode `polygon:size:angle` qui dessine un polygone dont on donnera le nombres de faces, la taille et l'angle duquel il faut tourner (voir figure 12.4). Définissez une méthode qui a seulement besoin du nombre de faces et de la taille.

Script 72 : Quelques polygones

```
| caro |
caro ← Turtle new.
caro polygon: 3 size: 50 angle: 120.
caro go: 50.
caro polygon: 4 size: 50 angle: 90.
caro go: 50.
caro polygon: 5 size: 50 angle: 72.
caro go: 50.
caro polygon: 6 size: 50 angle: 60.
```

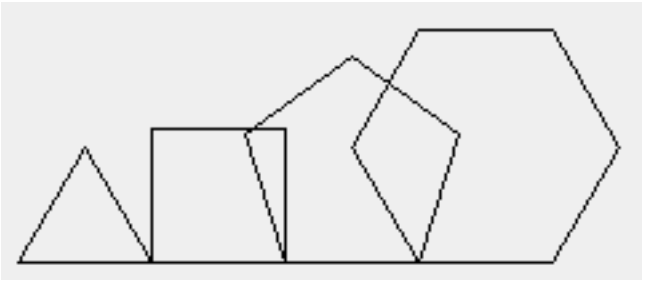


Figure 12.4: Un triangle, un carré, un pentagone, un hexagone... bref des polygones.

4.2 Croix

En partant du script de la croix donné plus ci-après, définissez une méthode nommée `croix:` qui trace une croix d'une taille donnée. Astuce dans le script notez que 100 / 2 = 50.

Script 73 : Croix

```
|caro|
caro ← Turtle new.
caro trace.
4 timesRepeat: [caro go: 50.
                caro turnLeft: 90.
                caro go: 100.
                caro turnRight: 90.
                caro go: 100.
                caro turnRight: 90.
                caro go: 50].
caro noTrace
```

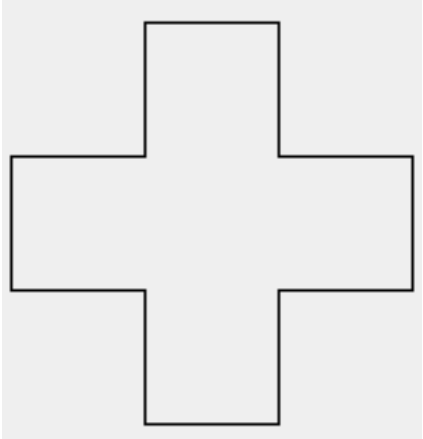


Figure 12.5: Une croix de 50 pixels.

Script 74 : Utilisation de la méthode `croix:`

```
|caro|
caro ← Turtle new.
caro croix: 30
```

4.3 Des croix étranges

Exercice 38: Etrange croix

En modifiant légérement la méthode `croix:`, définissez la méthode `etrangeCroixlargeur:profondeur:` qui permet de donner deux dimensions différentes et dessine les croix suivantes. La croix correspond à : `caro etrangeCroixLargeur: 30 profondeur: 60`

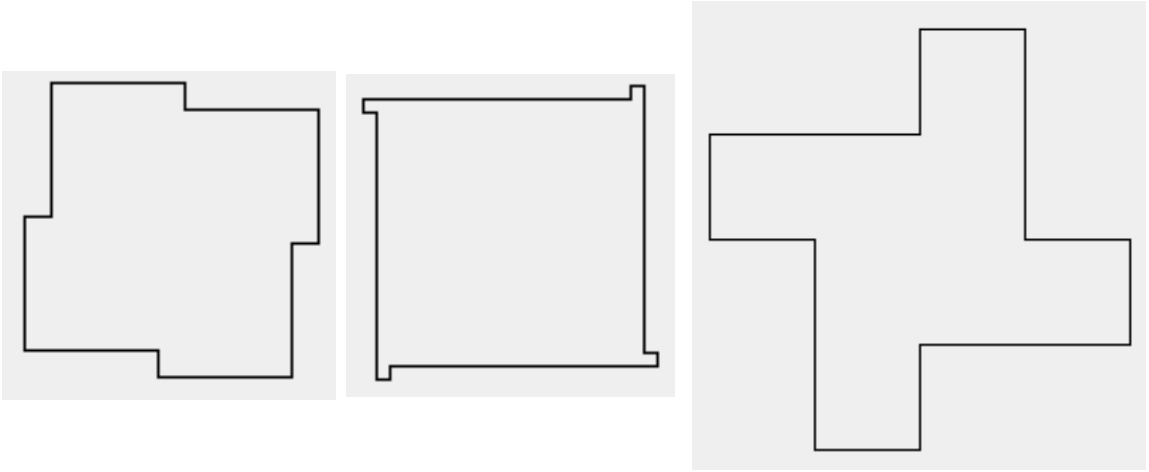
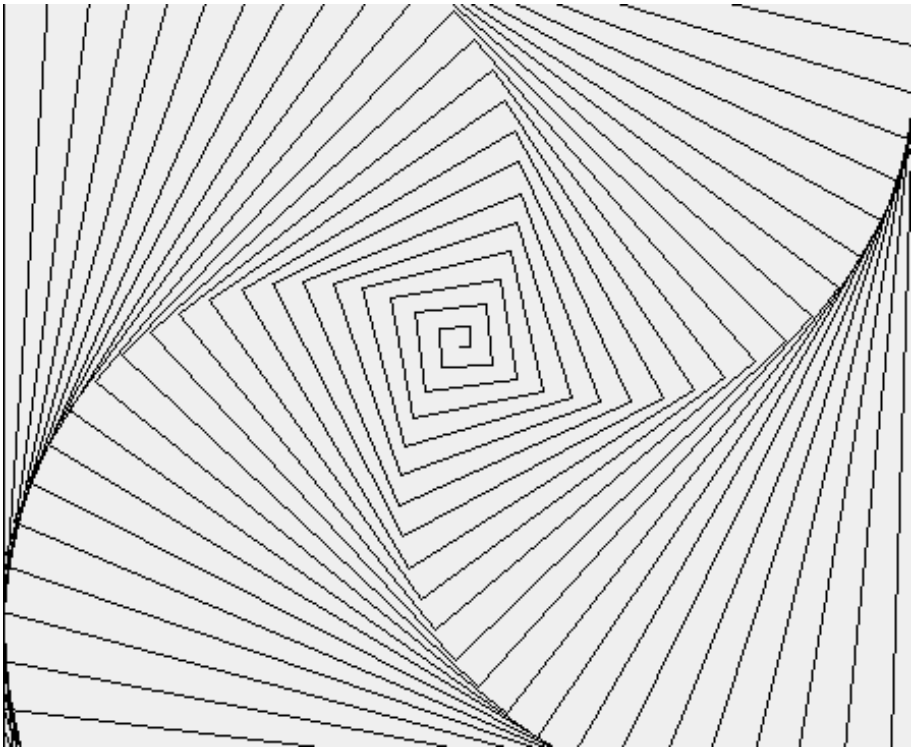


Figure 12.6: `caro etrangeCroixLargeur: 5 profondeur: 50`, `caro etrangeCroixLargeur: 50 profondeur: 50` et `caro etrangeCroixLargeur: 50 profondeur: 5`

Variables et Boucles



Dans ce chapitre, vous allez apprendre comment on peut utiliser des variables et changer leurs variables à l'aide de boucle.

1 Utilisons des variables

Lorsque l'on écrit certains scripts on voudrait pouvoir envoyer des messages aux tortues qui dépendent de *l'étape qui est en train d'être exécutée* ou du *nombre* d'étapes ayant déjà été exécutées. Par exemple, on voudrait qu'une tortue trace un escalier dont les marches grandissent comme dans la figure 13.1.

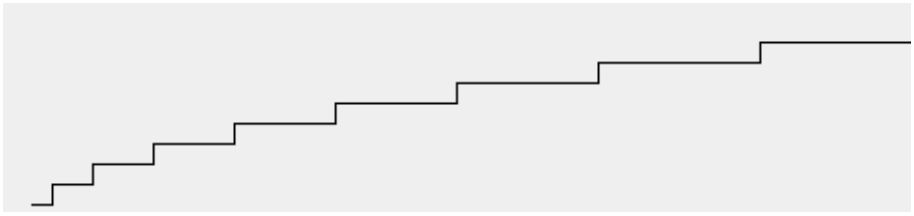


Figure 13.1: Escalier tordu.

Stéphane & Florence Ducasse (ducasse@iam.unibe.ch)

Pour cela nous avons besoin de *pouvoir changer la valeur des arguments des messages* en fonction de l'étape dans laquelle la tortue se trouve. Pour cela, nous utilisons des *variables*.

Le début de cet étrange escalier sans utiliser de boucles pourrait ressembler au script suivant :

Script 75 : Etrange escalier sans boucles

```
|caro|
caro ← Turtle new.
caro trace.
caro east.
caro go: 10.
caro north.
caro go: 10.
caro east.
caro go: 20.
caro north.
caro go: 10.
caro east.
caro go: 30.
caro north.
caro go: 10.
caro east.
caro go: 40.
caro north.
caro go: 10.
caro east.
caro go: 50.
...
```

Ce qu'il faut voir ici est que la longueur de chaque marche est simplement la longueur de la précédente marche à laquelle on ajoute 10. Ici 20 = 10 + 10, 30 = 20 + 10, 40 = 30 + 10, 50 = 40 + 10

Si on représente la longueur d'une marche à l'aide de la variable longueur on obtient:

longueur seconde étape = longueur après première étape + 10
longueur troisième étape = longueur seconde étape + 10
longueur quatrième étape = longueur troisième étape + 10 ...

Avec Squeak on écrit simplement `longueur ←longueur + 10` cela veut dire que la valeur de la variable longueur prend pour valeur l'ancienne valeur de la variable longueur plus 10. Si vous avez des doutes, relisez le chapitre 7.

1.1 Retour sur l'escalier simple

Regardons comment on peut modifier la méthode dessinant un escalier simple afin de produire l'escalier tordu.

La méthode `escalierSimple` est la suivante:

Méthode

```
escalierSimple

   self trace.
   10 timesRepeat: [self east.
                    self go: 10.
                    self north
                    self go: 10]
```

Elle est invoquée de la manière suivante :

Script 76 : Utilisation de `escalierSimple`

```
| caro |
caro ← Turtle new.
caro escalierSimple
```

La méthode `escalierSimple` est équivalente à la méthode suivante dans laquelle on introduit la variable `longueur` pour représenter la longueur d'une marche :

Méthode

```
escalierSimpleAvecLongueur
  |longueur|

  longueur ← 10.
  10 timesRepeat: [self east.
                   self go: longueur.
                   self north.
                   self go: 10]
```

Cette méthode est équivalente à la méthode précédente car la valeur de la variable `longueur` ne change pas et représente uniquement 10.

Pour produire un escalier tordu comme celui de la figure 13.2, il faut, en fait, changer la valeur de la longueur à l'intérieur de la boucle comme dans la méthode `escalierTordu` :

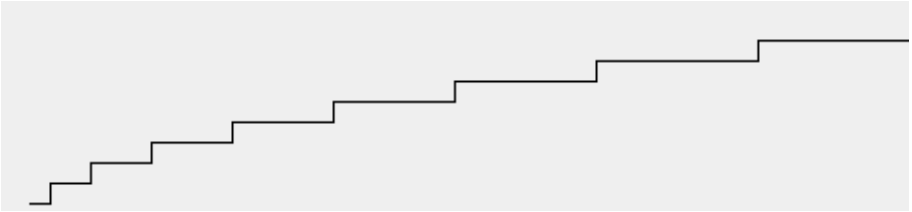


Figure 13.2: Escalier tordu.

Méthode

```
escalierTordu

  |longueur|
  longueur ← 10.
   10 timesRepeat: [self east.
                    self go: longueur.
                    self north.
                    self go: 10.
                    longueur ← longueur + 10]
```

Expliquons Tout d'abord `self go: longueur` fait avancer la tortue de la valeur de la variable `longueur`, puis tourner au nord et avancer de 10. Ensuite une nouvelle valeur pour la variable `longueur` est calculée en ajoutant 10 à la valeur précédente de la variable `longueur`.

Notez que suivant ce que l'on veut calculer la nouvelle valeur de la variable `longueur` pourrait être calculer en fonction d'autres variables. En effet, une variable peut dépendre d'autres variables et pas seulement de sa valeur (voir 7).

Exercice 39: Escalier réellement tordu

En modifiant la précédente méthode obtenez la figure 13.3.

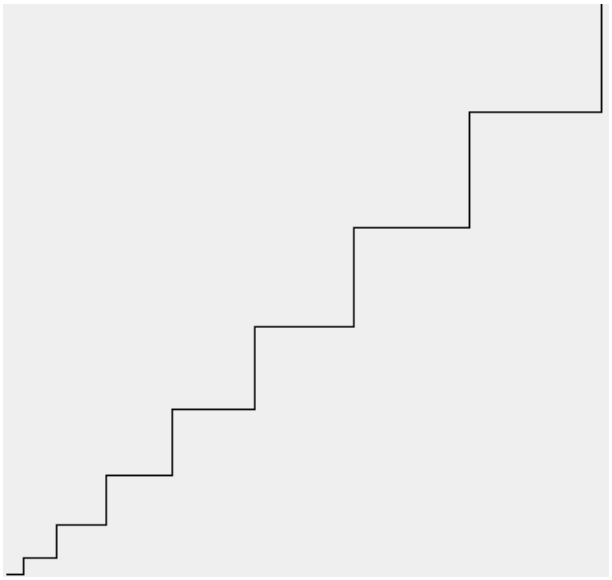


Figure 13.3: Escalier réellement tordu.

2 Labyrinthes

Exercice 40: Labyrinthe

Ecrivez la méthode `labyrinthe` qui produit le labyrinthe représenté par la figure 13.4. Aide 1 : Utilisez une boucle dans laquelle chaque élément de boucle consiste à avancer la tortue d'une certaine distance puis à lui faire faire un angle droit. Aide 2 : La longueur d'un morceau de labyrinthe est la longueur précédente plus une certaine valeur donc il faut penser à augmenter sa valeur dans la boucle et bien sûr ne pas oublier d'initialiser la variable.

Script 77 : Utilisation de la méthode `labyrinthe`

```
| caro |
caro ← Turtle new.
caro labyrinthe
```

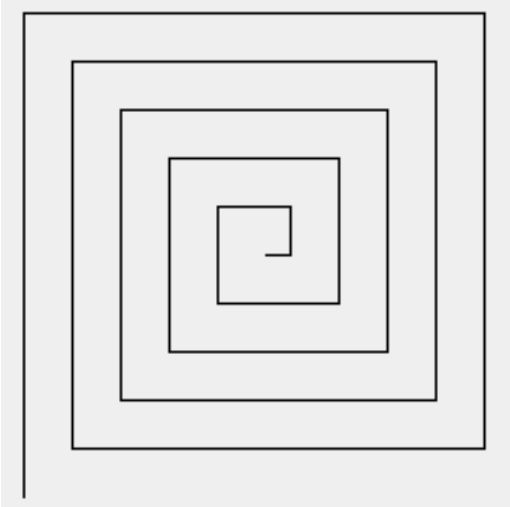


Figure 13.4: Labyrinthe.

Essayez de faire plus d'itérations (nombre de fois que la boucle tourne). Essayez avec 100 ou 200.

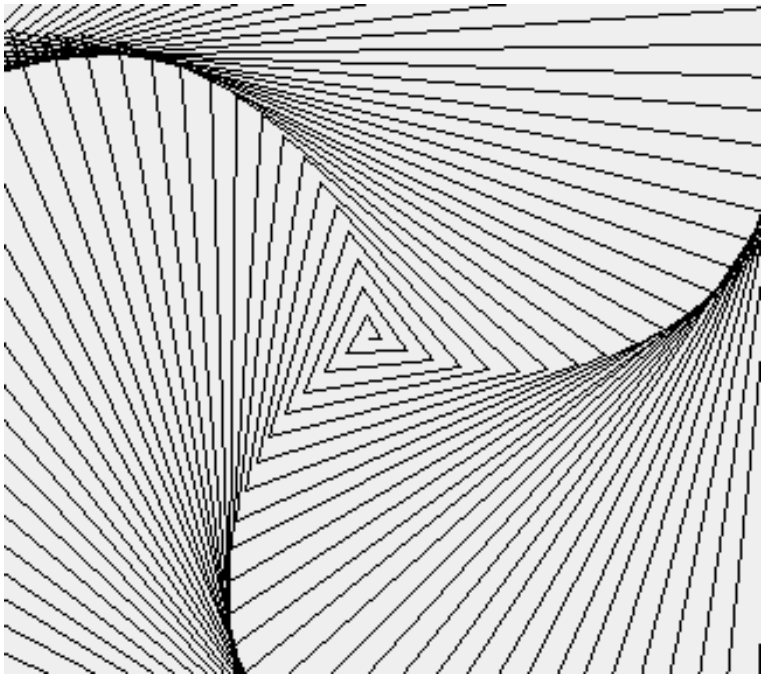


Figure 13.5: Spirale ave un angle de 121 degrés.

Exercice 41: Spirale

Reprenez le code de la méthode `labyrinthe` et définissez la méthode `spiralAngle: anAngle` qui définit une spirale dont on peut donner la valeur de l'angle.

Script 78 : Utilisation de `spiralAngle:`

```
|caro|
caro ← Turtle new.
caro spiralAngle: 55.
caro reset.
caro color: Color blue.
caro spiralAngle: 90
```

Cette méthode `spiralAngle:` qui produit les différentes figures 13.5 et 13.6.

L'idée est que vous devez utiliser une variable pour l'angle. Aide 1: introduisez tout d'abord une variable angle supplémentaire mais sans changer sa valeur (comme nous l'avons fait pour escalier simple). C'est-à-dire que vous devez obtenir le labyrinthe (figure 13.4). Aide 2 : Puis vous changer la valeur de l'angle.

Essayez avec de nombreuses itérations et en faisant varier la valeur donnée aux angles, essayez par exemple, 121, 144, 160. Notez que le labyrinthe est une spirale avec un angle de 90. La figure de début de chapitre est générée avec un angle de 91 degrés.

2.1 Multiplions au lieu d'ajouter

Exercice 42: Facteur

Nous avons toujours augmenté la longueur de deux segments consécutifs d'un même nombre. Il serait amusant de définir la nouvelle longueur comme un coefficient de l'ancienne. Définissez la méthode `spiralFactor:angle:`.

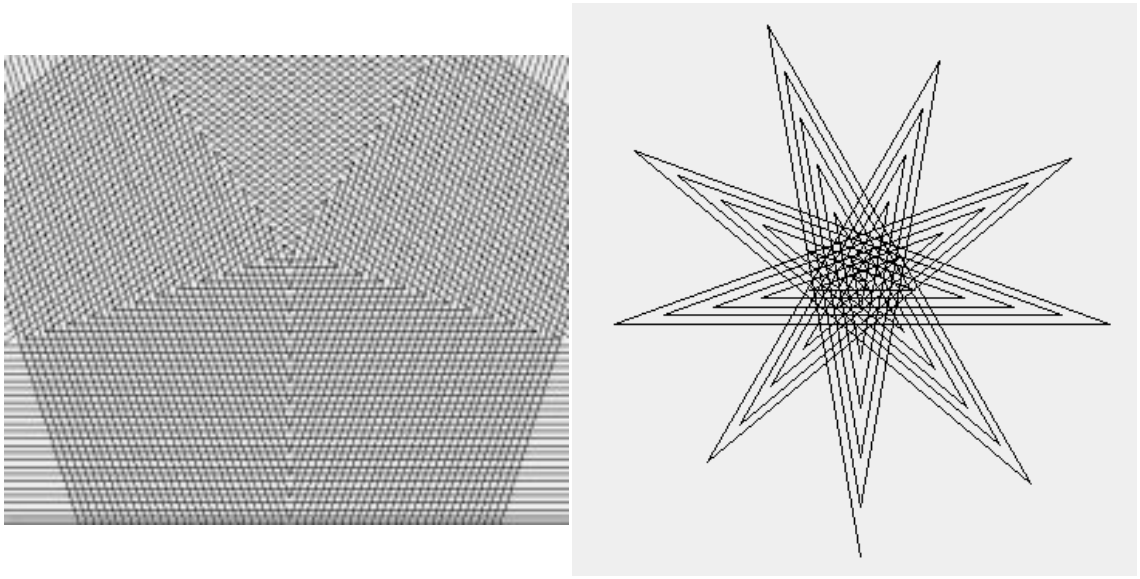


Figure 13.6: Spirale avec un angle de 144 puis un angle 160 degrés.

Script 79 : Utilisation de `spiralFactor:angle:`

```
| caro |
caro ← Turtle new.
caro spiralFactor: 1.1 angle: 90
```

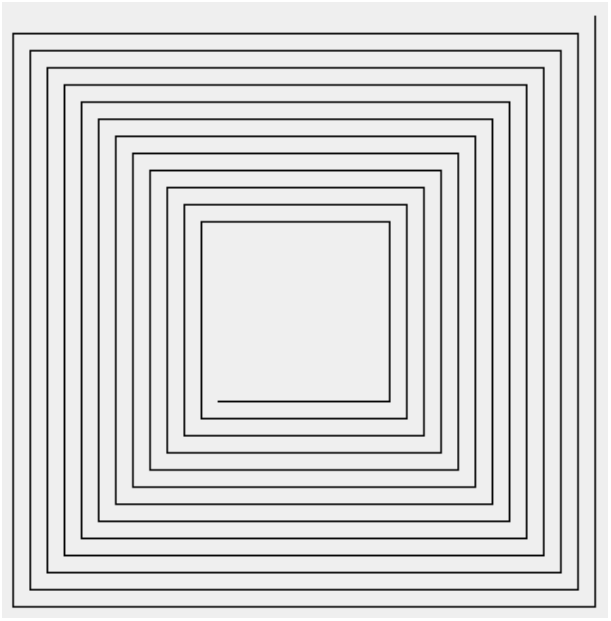


Figure 13.7: `caro spiralSegments: 50 size: 100 increment: 5 angle: 90`

2.2 Un générateur de labyrinthes

Lorsque l'on regarde attentivement la méthode `labyrinthe` on s'aperçoit qu'un labyrinthe dépend de 4 facteurs : la *taille* de sa première cloison (ici par exemple taille ←5), le *nombre de segments* (ici 100), l'*angle* entre deux segments (ici 90) et finalement l'*augmentation* de la longueur entre chaque segment.

Exercice 43: Un générateur

Définissez la méthode `spiralSegments:size:increment:angle:` qui permet de préciser le nombre de segments (premier argument), la taille de la première cloison (second argument), l'augmentation en longueur (troisième argument) et l'angle entre deux segments (dernier argument).

Script 80 : Utilisons la méthode `spiralSegments:size:increment:angle:`

```
|caro|
caro ← Turtle new.
caro spiralSegments: 500 size: 5 increment: 5 angle: 92.
```

Les figures 13.7 et 13.8 montrent quelques résultats que vous pouvez obtenir en faisant varier les différents facteurs.

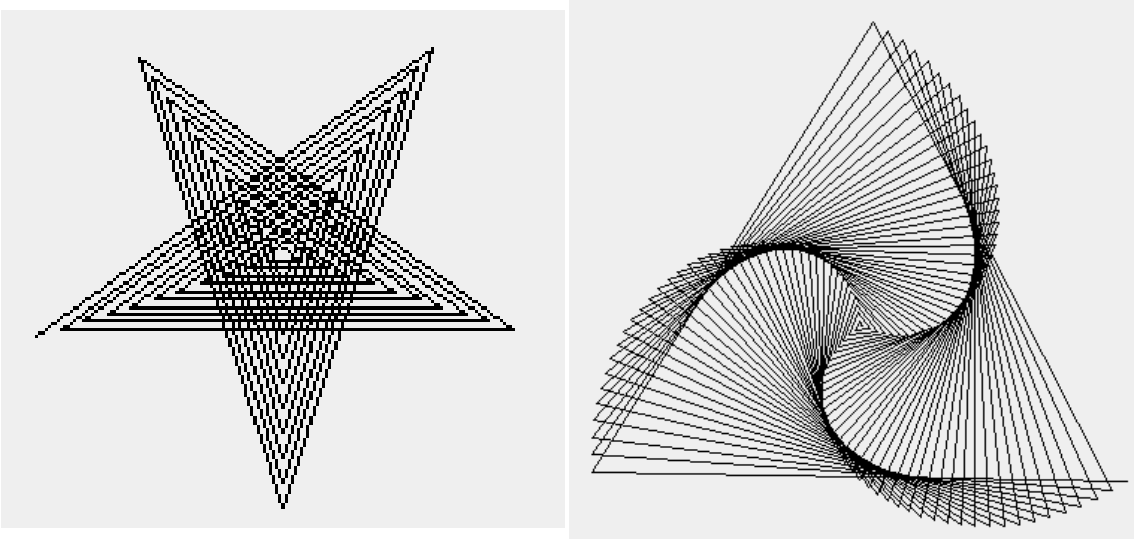
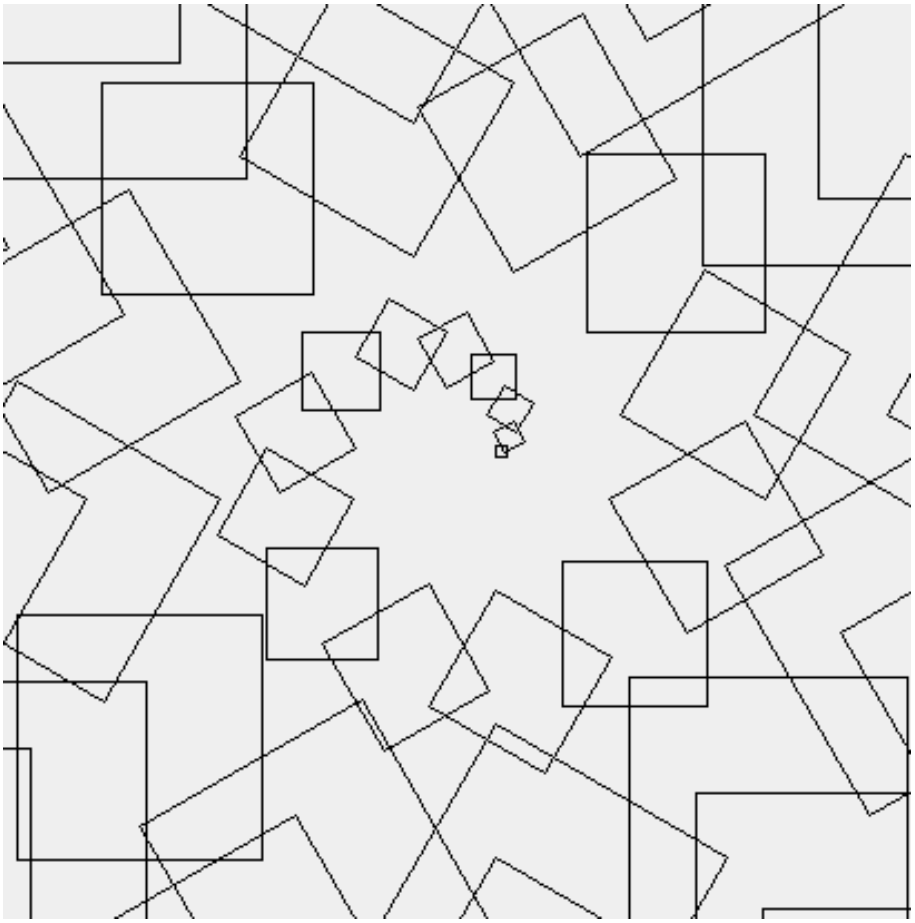


Figure 13.8: `caro spiralSegments: 50 size: 10 increment: 3 angle: 144` et `caro spiralSegments: 120 size: 1 increment: 3 angle: 12`

xk

Composons un peu plus loin!



Vous avez appris à définir des méthodes au chapitre 9, à composer ces méthodes au chapitre 10 et à utiliser des arguments au chapitre 12. Maintenant, vous allez composer des méthodes avec des arguments. Notez que ce n'est pas nouveau : lorsque vous avez défini la méthode `square:` vous avez utilisé un argument et appelez la méthode `go:` avec cet argument.

1 Carrés

Dans ce chapitre, vous allez avoir besoin de la méthode `square:` qui dessine un carré d'une taille donnée.

Définissez la méthode `squares:size:space:` qui dessine une rangée de carrés contenant un nombre donné de carrés, d'une taille donnée et avec une distance donnée entre chaque carré.

Stéphane & Florence Ducasse (ducasse@iam.unibe.ch)

Script 81 : Utilisation de la méthode `squares:size:space:`

```
| caro |
caro ← Turtle new.
caro squares: 5 size: 20 space: 30
```

2 Hexagones et Carrés

Lors du chapitre précédent nous avons vu la définition de la méthode `hexagone:` que nous rappelons ci-après :

Méthode

```
hexagon: size
     "Draws an hexagon"

     self trace.
     6 timesRepeat: [self turnRight: 60.
                     self go: size].
     self noTrace
```

Exercice 44: Hexacarré

Définissez la méthode `hexagoneSquare:` qui produit la figure

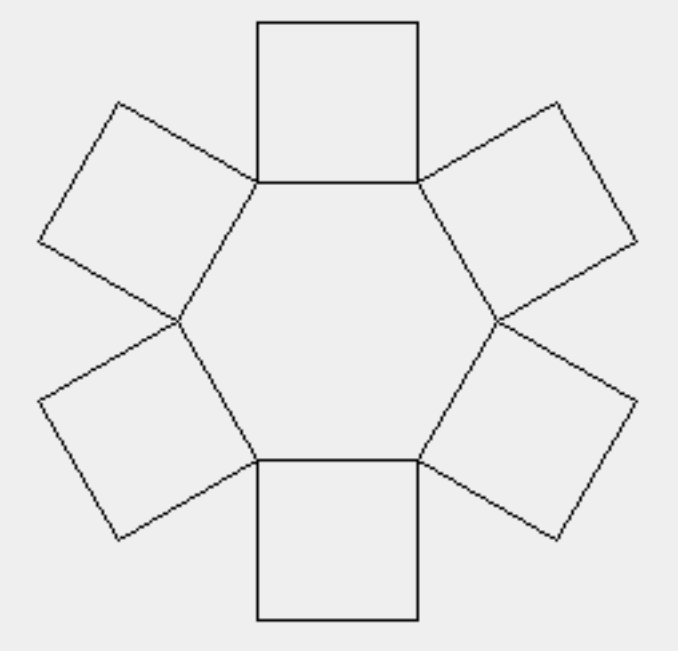


Figure 14.1: Une hexacarré.

Script 82 : Utilisation de la méthode `hexagoneSquare`

```
|caro|
caro ← Turtle new.
caro hexagoneSquare: 40
```

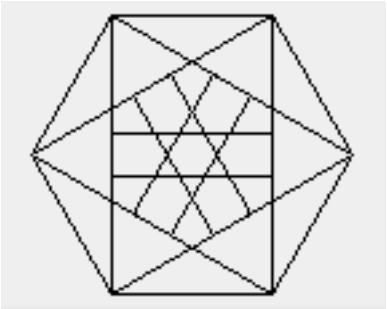


Figure 14.2: Un possible résultat de `hexagonSquare:`

Attention : Cet exercice peut être difficile à réaliser car les carrés peuvent tourner dans différents sens. Il est possible que vous obteniez la figure précédente au lieu de celle de l'hexagone avec des carrés. Cela est dû au sens dans lequel un carré est dessiné. En effet, regardez la figure 14.3 créée par le script suivant :

Script 83 : Deux carrés tournant dans des sens différents

```
|caro taille|
caro ← Turtle new.
caro trace.
4 timesRepeat: [caro turnLeft: 90.
                caro go: 100].
caro color: Color blue.
4 timesRepeat: [caro turnRight: 90.
                caro go: 100]
```

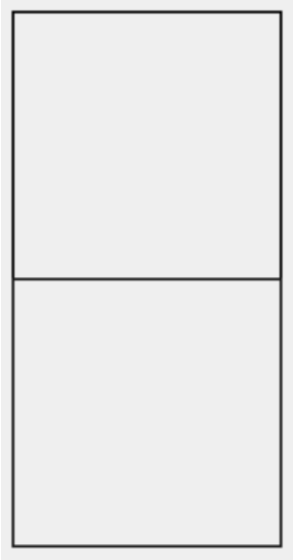


Figure 14.3: Deux carrés.

3 HyperSquare

En utilisant la méthode `square:` essayez de reproduire la figure hyperSquare qui illustre le début de ce chapitre.

Script 84 : Utilisation de la méthode `hyperSquare`

```
|caro |
caro← Turtle new.
caro north.
caro go: 100.
caro east.
caro hyperSquare
```

Exercice 45: `pentagone`.

En utilisant la méthode `spiral:angle:` définie au chapitre 13 qui créé une petite étoile comme montré la figure 14.4 définissez la méthode pentagone qui dessine la figure 14.5.

Script 85 : Utilisation de `spiral:angle:`

```
| caro |
caro ← Turtle new.
caro spiral: 20 angle: 144
```

Figure 14.4: Une petite étoile.

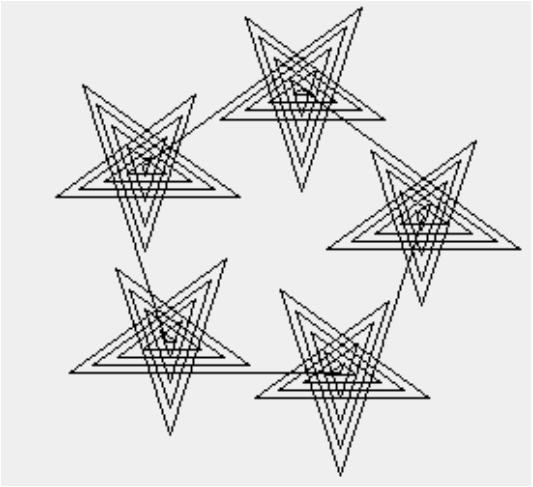


Figure 14.5: Un bien étrange pentagone.