DÉCOUVRIR LA PROGRAMMATION AVEC

# LE LANGAGE LINOTTE

VERSION 2.1.3

***Ronan Mounès***

***Wam***

# La programmation : qu'est-ce que c'est ?

La programmation, c'est la création de programmes.

Des programmes sont présents dans tous les appareils informatiques : ordinateur, console de jeu guichet automatique bancaire, ainsi que dans les composants de nombreux dispositifs électroniques : machine à laver, robot ménager, téléphone, appareil photo numérique, TV, ...

Un programme est une suite d'instructions compréhensibles par la machine, qui permettent d'offrir un service à l'utilisateur.
Ainsi, sur votre ordinateur, les programmes sont divers :

- navigateurs internet : Internet Explorer, Google Chrome, etc...
- traitements de texte : Open Office, Word, etc...
- jeux : Démineur, Solitaire, etc...

Les instructions de programme sont exécutées par le processeur.
Le processeur utilise un langage machine, qui est un langage constitué de 0 et de 1.
Ainsi, ce code binaire :

0100001001101111011011100110101001101111011101010111010

… représente, par exemple, une addition pour votre ordinateur.

Un tel langage est alors bien difficile à manipuler pour un programmeur.
Afin de rendre son travail moins pénible et moins sujet à de nombreuses erreurs, le programmeur utilise alors un langage de programmation.

Un langage de programmation permet de remplacer le code binaire par un langage plus compréhensible par l'homme.
Ainsi, un programmeur va pouvoir écrire ses instructions grâce à un vocabulaire et une ponctuation, inspirés d'une langue naturelle.

Voici un exemple de code écrit en Java :

```
public class HelloWorld {
    public static void main(String[] args) {
        // Display the greeting.
        System.out.println("Hello World!");
    }
}
```

Même si tout ceci semble du charabia, cela semble plus lisible qu'une suite de 0 et de 1, non ?

Ceci est rendu possible par l'utilisation d'un traducteur automatique.
Suivant le langage de programmation utilisé, il peut s'agir d'un **compilateur** ou d'un **interpréteur** :
- le **compilateur** lit le programme en entier puis le traduit en langage machine. Une fois la traduction achevée, le programme peut alors être exécuté par le processeur. Par exemple, le langage C++ utilise un compilateur.

- l'**interpréteur** lit le programme ligne par ligne et exécute immédiatement les instructions machines correspondantes. C'est le cas du langage Linotte.
L'avantage d'un interpréteur c'est que les erreurs peuvent être immédiatement corrigées. Le désavantage est que l'exécution du programme est moins rapide que s'il avait été préalablement traduit par un compilateur.

# Le langage Linotte

Linotte est un langage libre, cela signifie que n'importe qui peut l'utiliser, le distribuer et le modifier. Il a été créé par Ronan Mounès en 2005, et a pour but de permettre à tous, quelles que soit vos connaissances en informatique, aussi bien les enfants, les débutants désireux d'apprendre la programmation ou les programmeurs plus expérimentés voulant s'orienter vers un nouveau langage, de réaliser des programmes.

Cependant, il existe des centaines de langage de programmation. Alors, pourquoi choisir le Linotte ?

Un critère important dans le choix d'un langage de programmation est le niveau du langage. Il existe des langages de **haut niveau** et de **bas niveau**.
Un langage de **bas niveau** se veut très proche du fonctionnement de la machine : il permet d'optimiser ses programmes à l'instruction près, mais est beaucoup plus difficile à appréhender. De plus, les programmes ne pourront sans doute pas fonctionner d'un ordinateur à l'autre (on dit qu'ils ne sont pas **portables**).
Par exemple, le langage Assembleur est un langage de **bas niveau** :

```
.model tiny

.data
HelloMessage db 13, 10, 'Hello World !', 13, 10, '$'

.code
.486
org 100h
start:
    mov ax, @data
    mov ds,ax
    mov ax,3
    int 10h
    mov ah,9
    mov dx, offset HelloMessage
    int 21h
    xor ax,ax
    int 16h
    mov ax,3
    int 10h
    mov ah,4ch
    int 21h
end start
```

Un langage de **haut niveau** est assez éloigné du langage binaire et s'inspire des langues naturelles : il est plus souple et rapide à écrire. En revanche, il est plus lent à s'exécuter. Mais le programme peut être utilisé sur plusieurs types d'ordinateurs, même si toutefois, un programme écrit en langage de **haut niveau** est souvent conçu pour utiliser un système d'exploitation en particulier.
Le langage Java est un langage de **haut niveau**. L'interpréteur Linotte est d'ailleurs écrit en Java.

Voici le même exemple que précédemment, en langage Linotte :


Le Linotte est donc un langage de **haut niveau**.

De plus, la particularité du Linotte est qu'il dispose d'une syntaxe en français, contrairement aux autres langages, qui se rapprochent le plus souvent de l'anglais.
Comme l'indique ses devises « Tu sais lire un livre, alors tu peux écrire un programme informatique » et « Tu sais écrire une phrase, donc tu sais écrire un programme », le langage Linotte est très simple d'utilisation.

Mais derrière cette simplicité, le Linotte est un langage puissant et multi-paradigmes : on peut en effet commencer par adopter une logique de programmation impérative, ensuite évoluer vers la programmation fonctionnelle, ou encore aborder la programmation orientée objet !
Si tous ces termes vous sont inconnus, ne vous inquiétez pas : sachez seulement que ce sont différentes façons de programmer qui peuvent être utilisées en Linotte.

# Installation

Pour ce chapitre, il vous faut installer le logiciel Java fournit par SUN Microsystems. Vous pouvez le trouver sur le site http://www.java.com/fr/
Il est important d'installer une version égale ou supérieur à 6.0.

## Sous Microsoft Windows

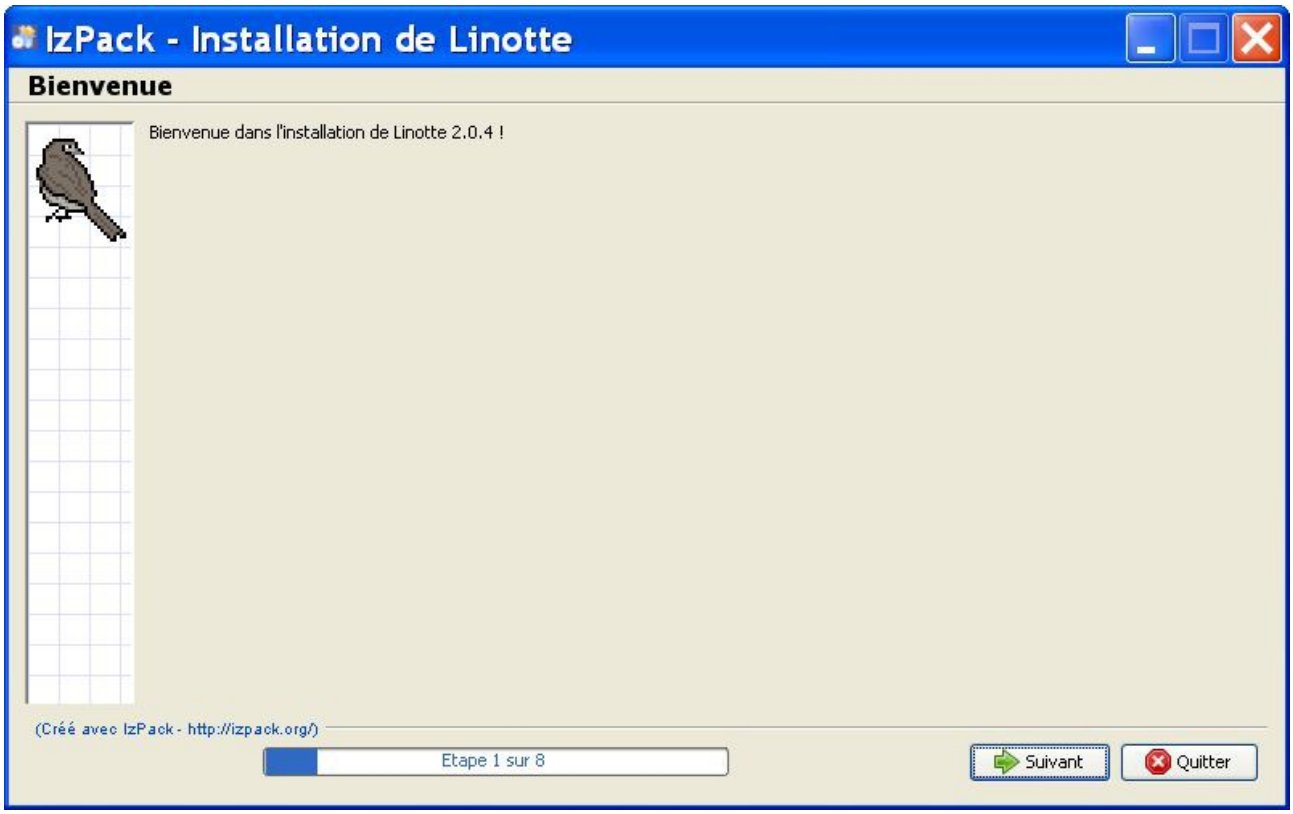
Avant de continuer, il vous faut un désarchiveur : http://www.7-zip.org/

Pour télécharger l'interprète Linotte, il vous faut aller sur le site officiel de Linotte : http://langagelinotte.free.fr/wordpress/

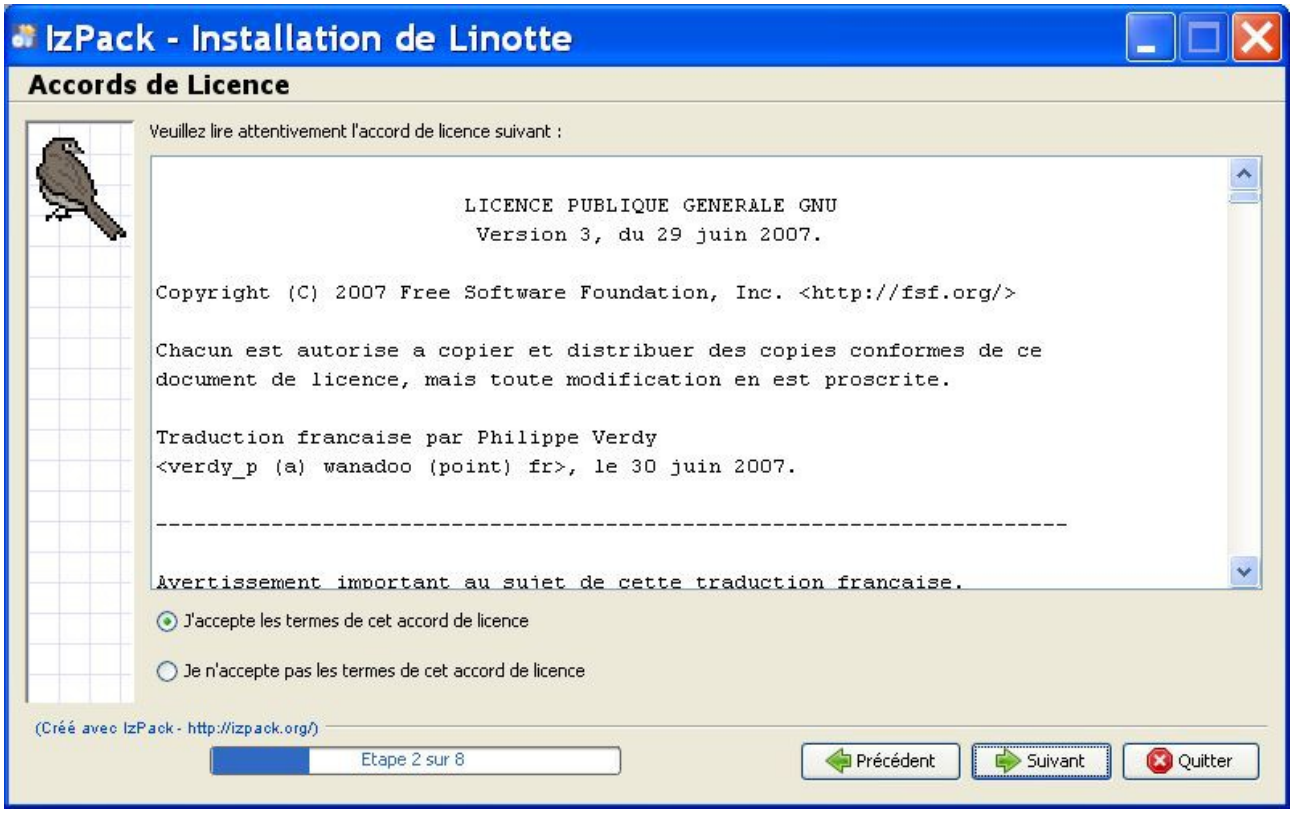
Enregistrez le fichier *Linotte-Setup_x.x.x.zip* sur votre disque dur.
Ce fichier contient l'exécutable pour l'interprète Linotte.

Si vous avez installé *7-zip*, lancez le désarchiveur et choisissez de l'extraire dans le répertoire de votre choix.

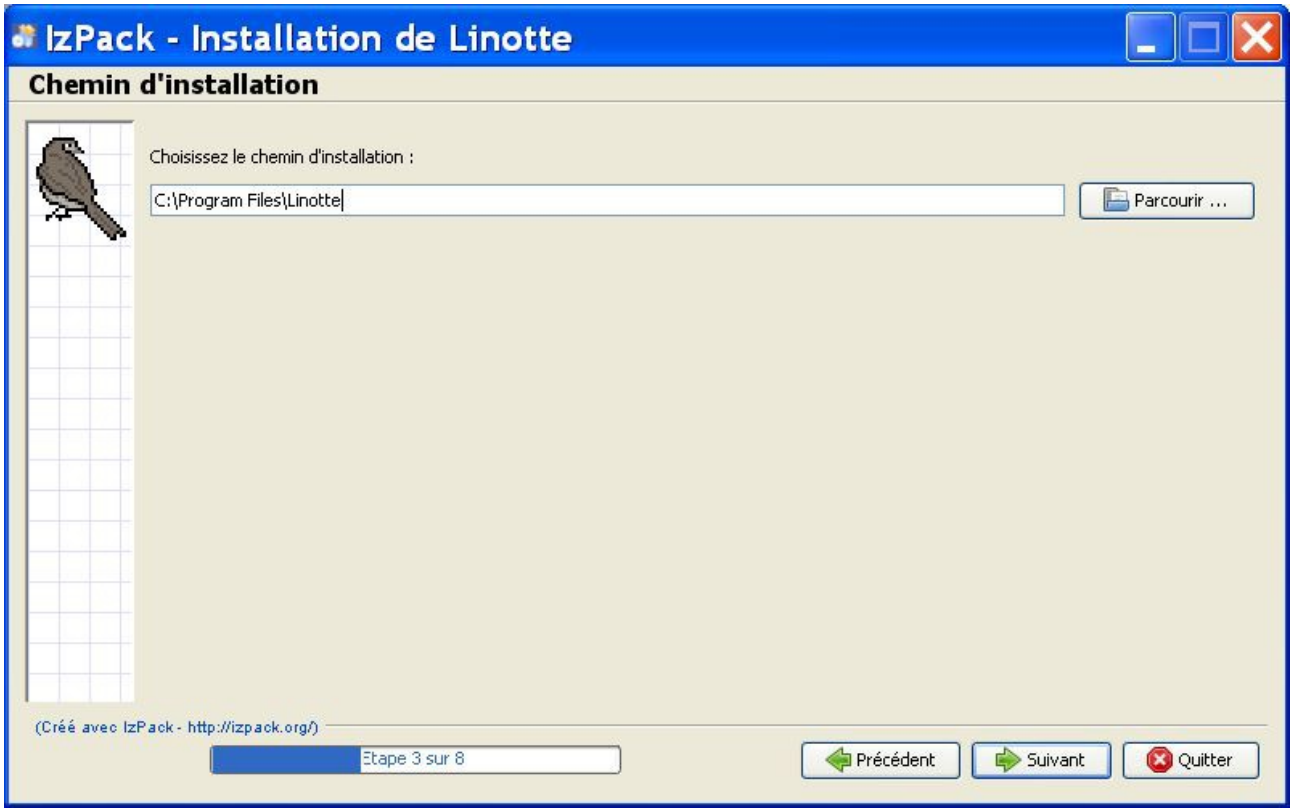
Dans le dossier *Setup-Linotte_x.x.x.* vous trouverez alors un fichier intitulé *Setup-Linotte*.
Double-cliquez dessus.
Vous devriez voir cette fenêtre apparaître :



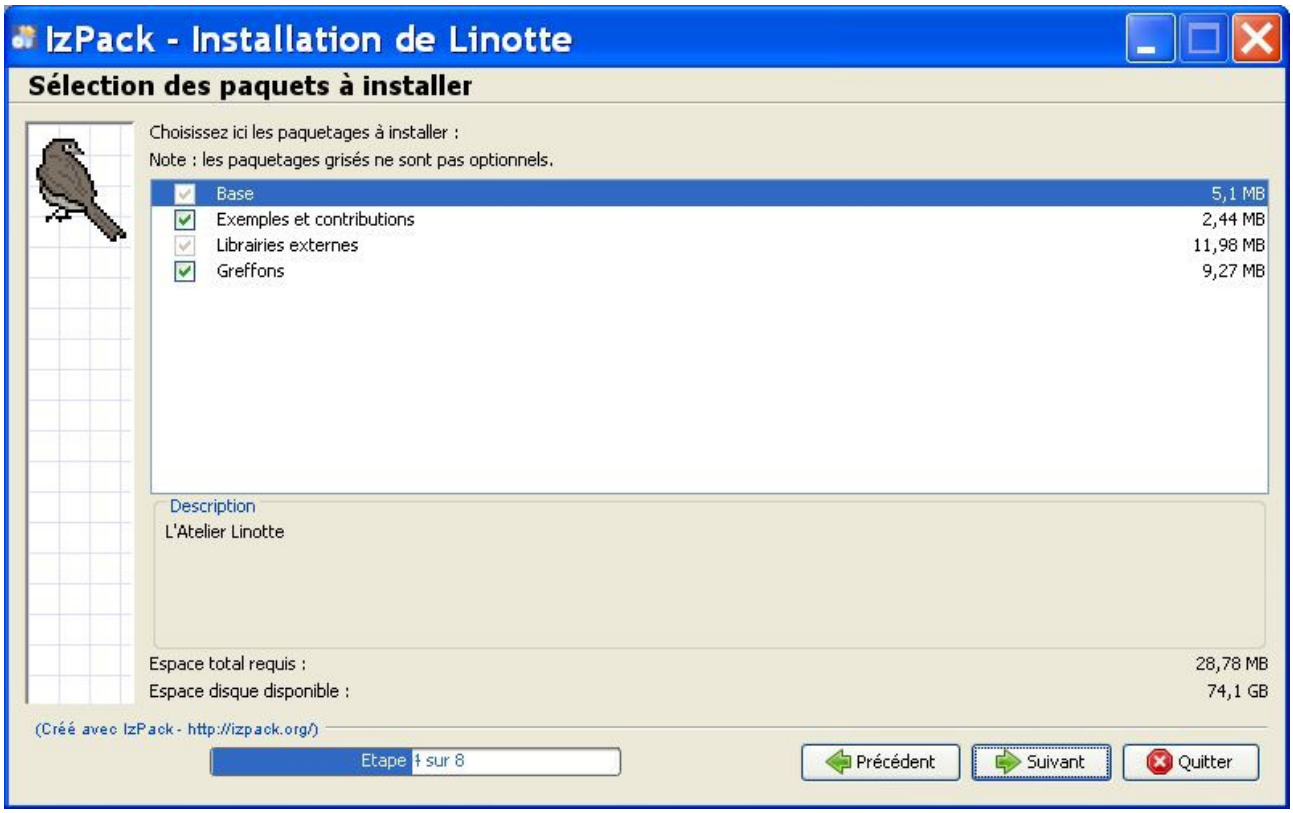
Cliquez sur **Suivant**.



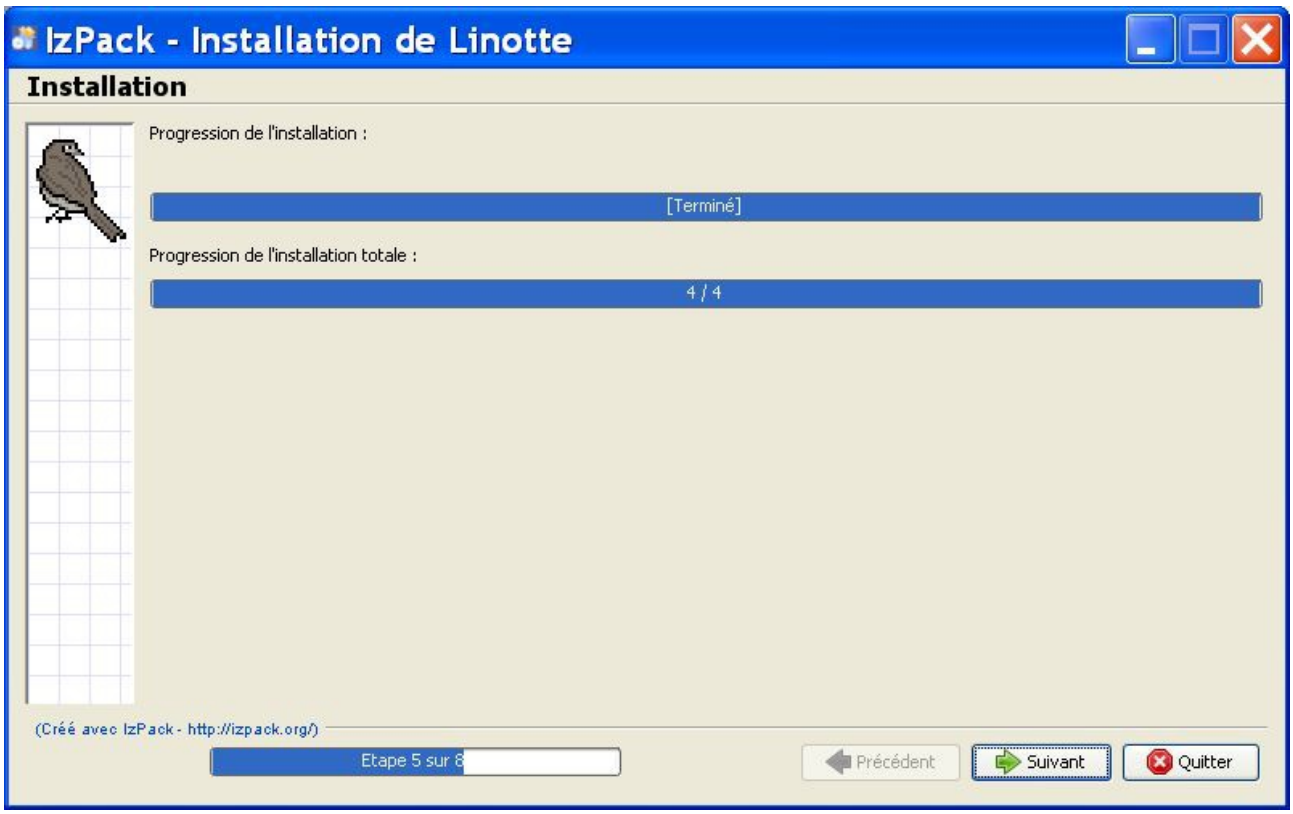

Acceptez les termes de l'accord de licence et cliquez sur **Suivant**.



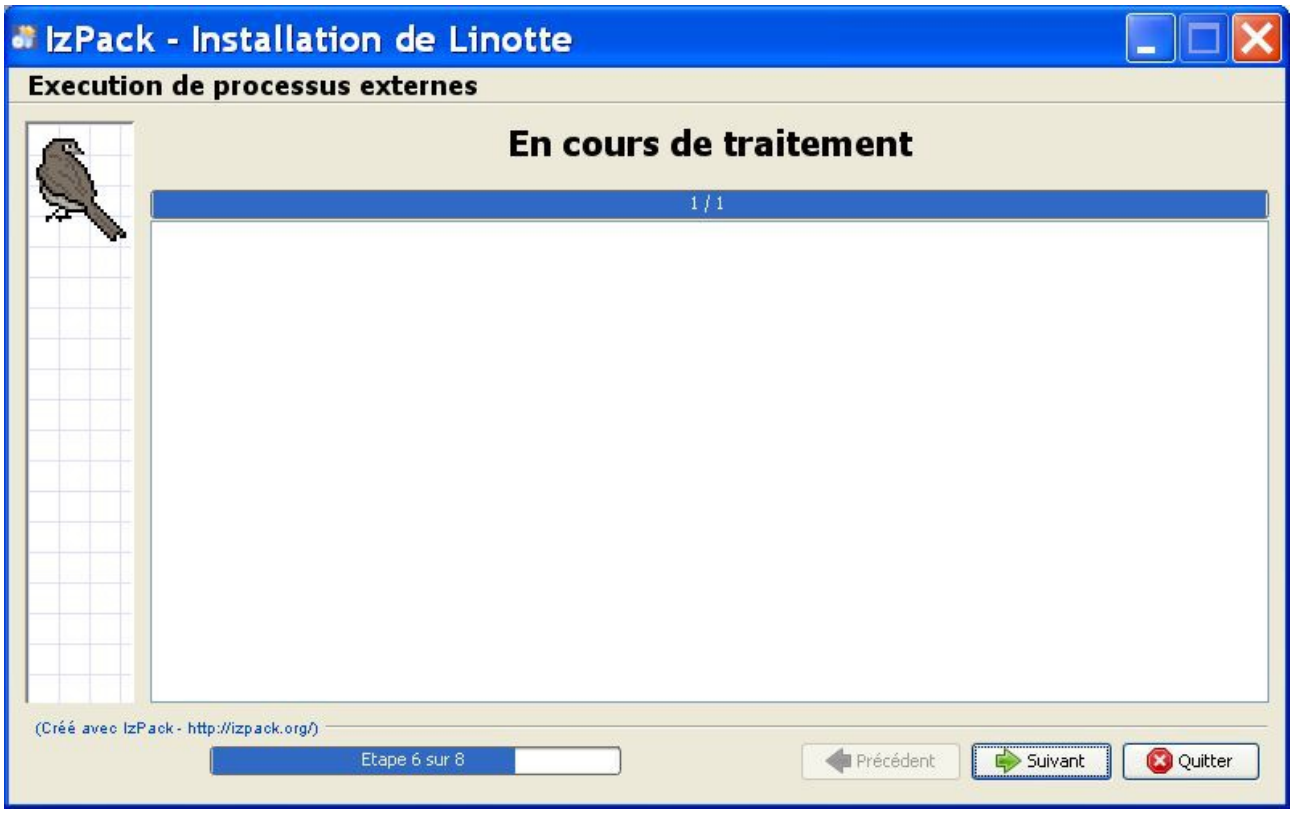
Choisissez le chemin d'installation, par exemple : C:\Program Files\Linotte.
Cliquez sur **Suivant**.




Laissez les options telles quelles, et cliquez sur **Suivant**.
Le programme s'installe alors, jusqu'à afficher ceci :

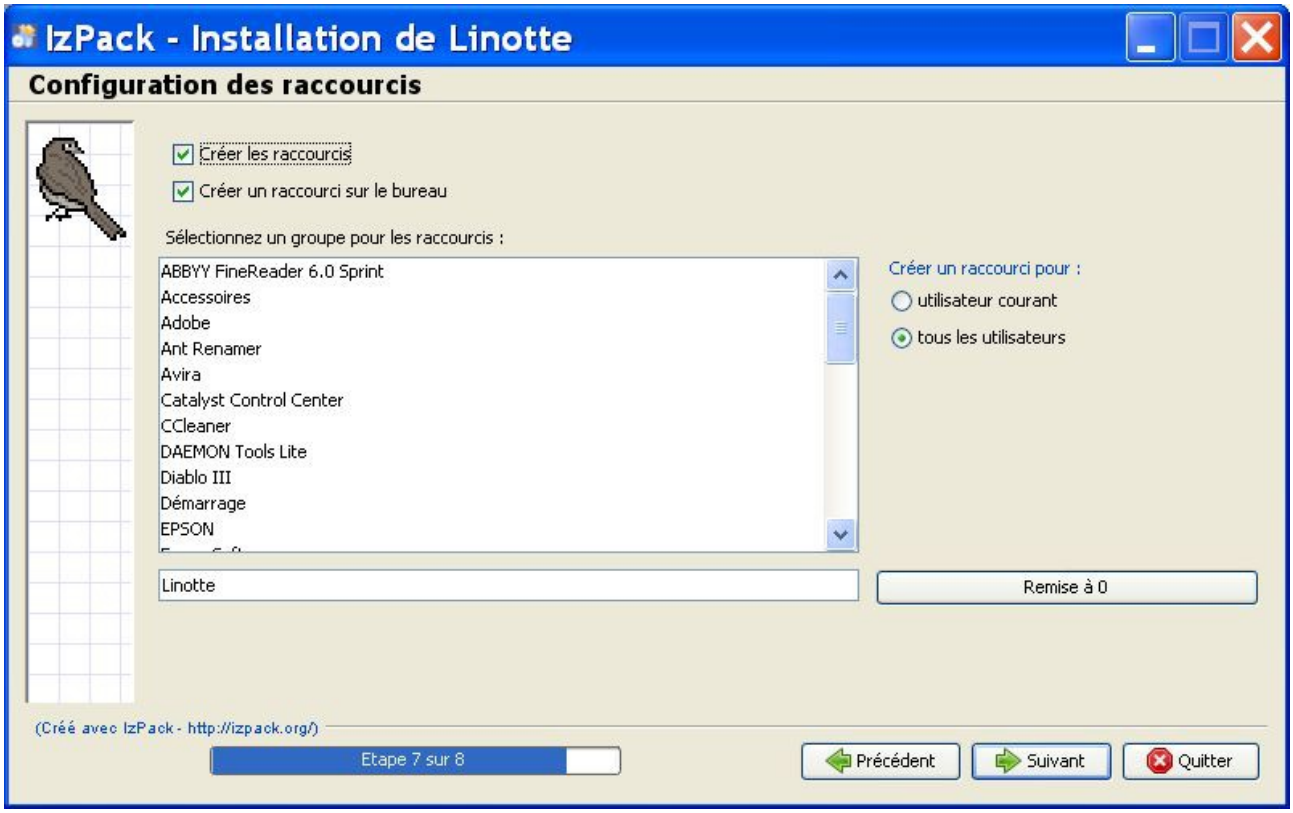



Cliquez sur **Suivant**.

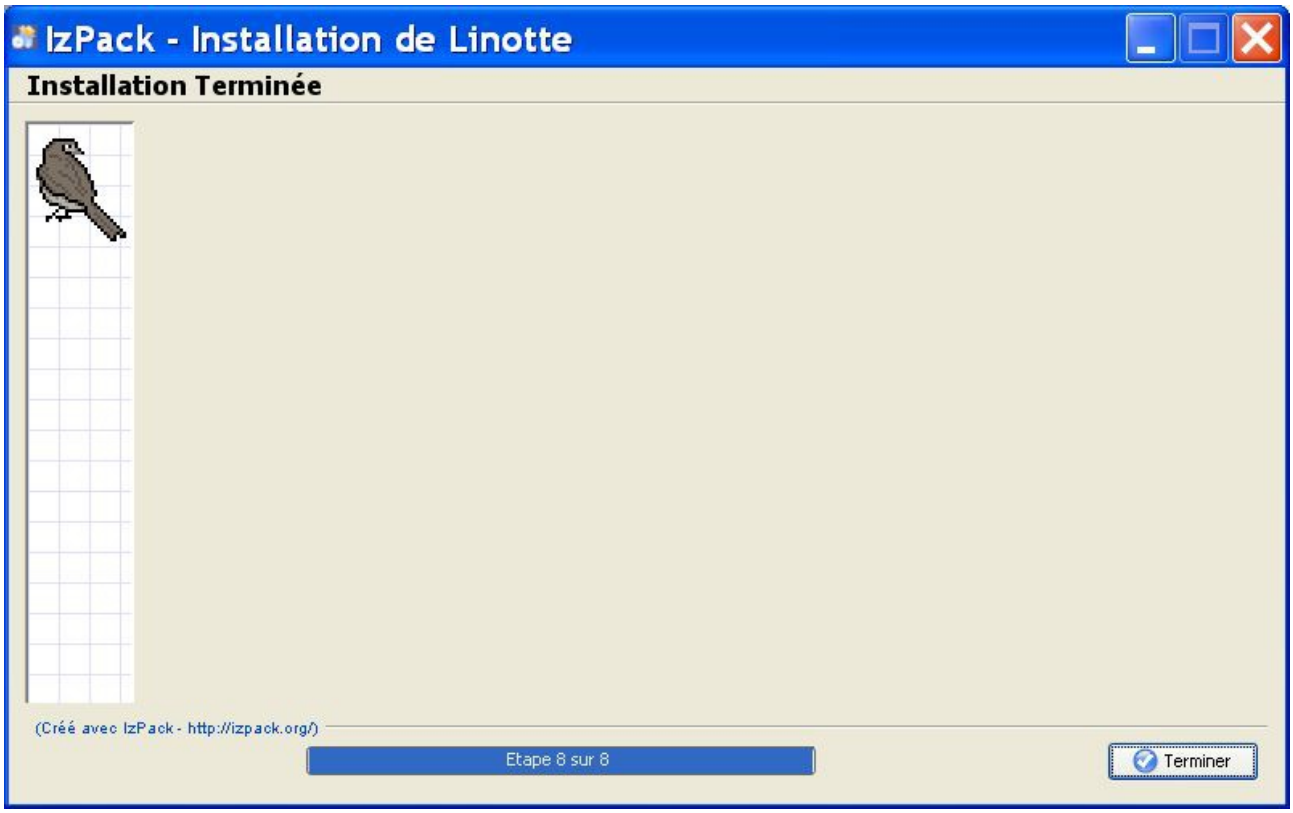



Linotte récupère les livres ouverts dans une précédente version de l'atelier.

Cliquez sur **Suivant**.




Si vous ne voulez pas créer de raccourcis, décocher les cases correspondantes.
Cliquez sur **Suivant**.



L’installation est terminée !
Cliquez sur **Terminer** pour fermer la fenêtre.

# Sous Linux

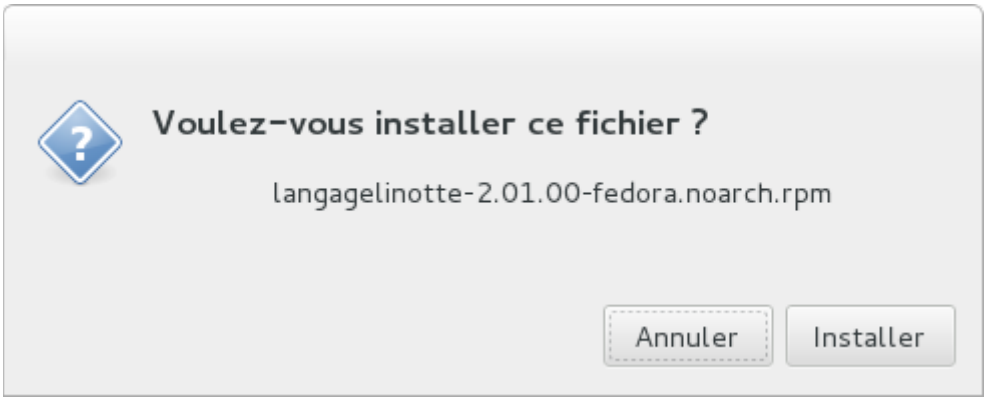
Pour télécharger l’interprète Linotte à l’aide de votre navigateur Internet, il vous faut aller sur le site officiel de Linotte : http://langagelinotte.free.fr/wordpress/

Enregistrez le fichier *Linotte-Setup_x.x.x.zip* sur votre disque dur.
Ce fichier contient l'exécutable pour l'interprète Linotte.

Toutes les distributions Linux savent décompresser les fichiers archives.
Vous pouvez utiliser l'environnement Gnome ou KDE pour extraire les fichiers.
Sinon, si vous être sur une console, tapez : *unzip Linotte-Setup_x.x.x.zip*
Les fichiers seront extraits dans le répertoire *Linotte-Setup_x.x.x*

**Sous Fedora :**

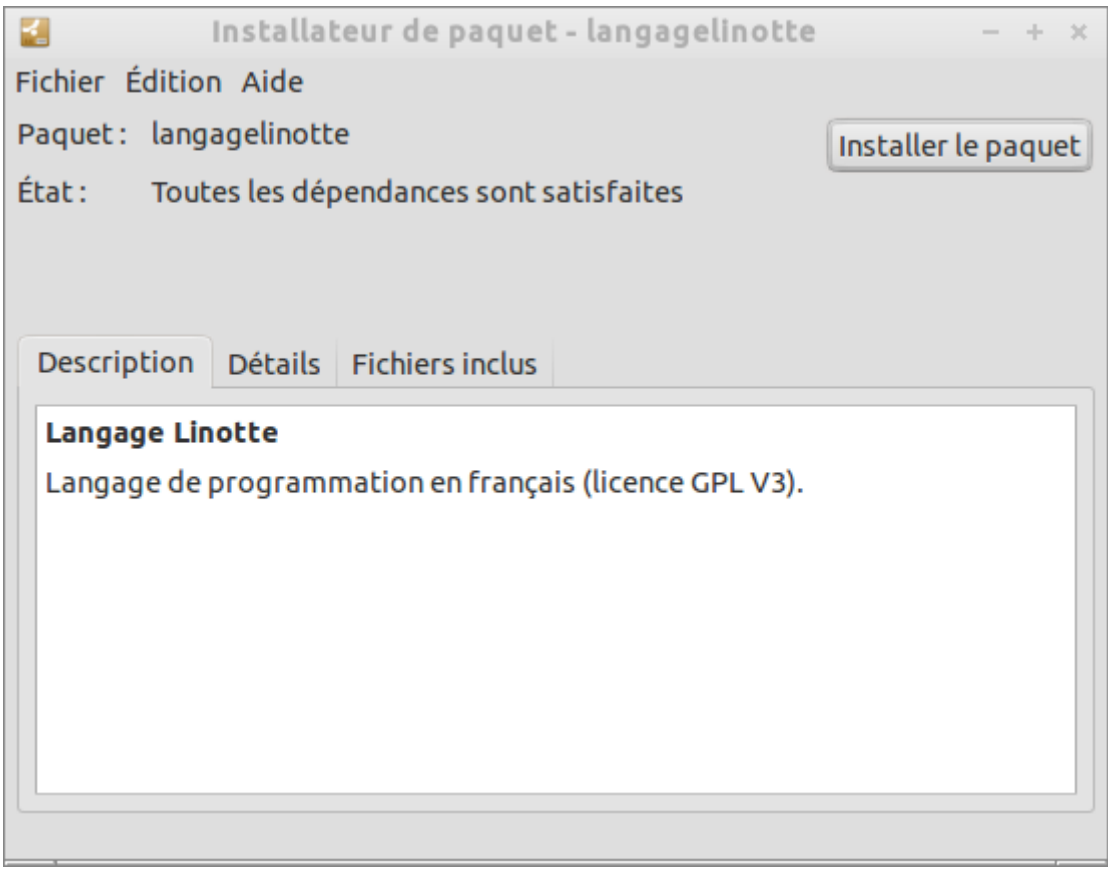
Vous devriez voir cette fenêtre apparaître :




Cliquez sur **Installer**.

**Sous Ubuntu :**

Vous devriez voir cette fenêtre apparaître :




Cliquez sur **Installer le paquet**.

# Sous MacOSX

Avant de continuer, il vous faut un désarchiveur : http://www.stuffit.com/mac-expander.html

Pour télécharger l'interprète Linotte, il vous faut aller sur le site officiel de Linotte : http://langagelinotte.free.fr/wordpress/

Enregistrez le fichier *Linotte-Setup_x.x.x.zip* sur votre disque dur.
Ce fichier contient l'exécutable pour l'interprète Linotte.

Pour décompresser les fichiers, il suffit de double-cliquer sur le fichier dans le Finder.

# L'atelier Linotte

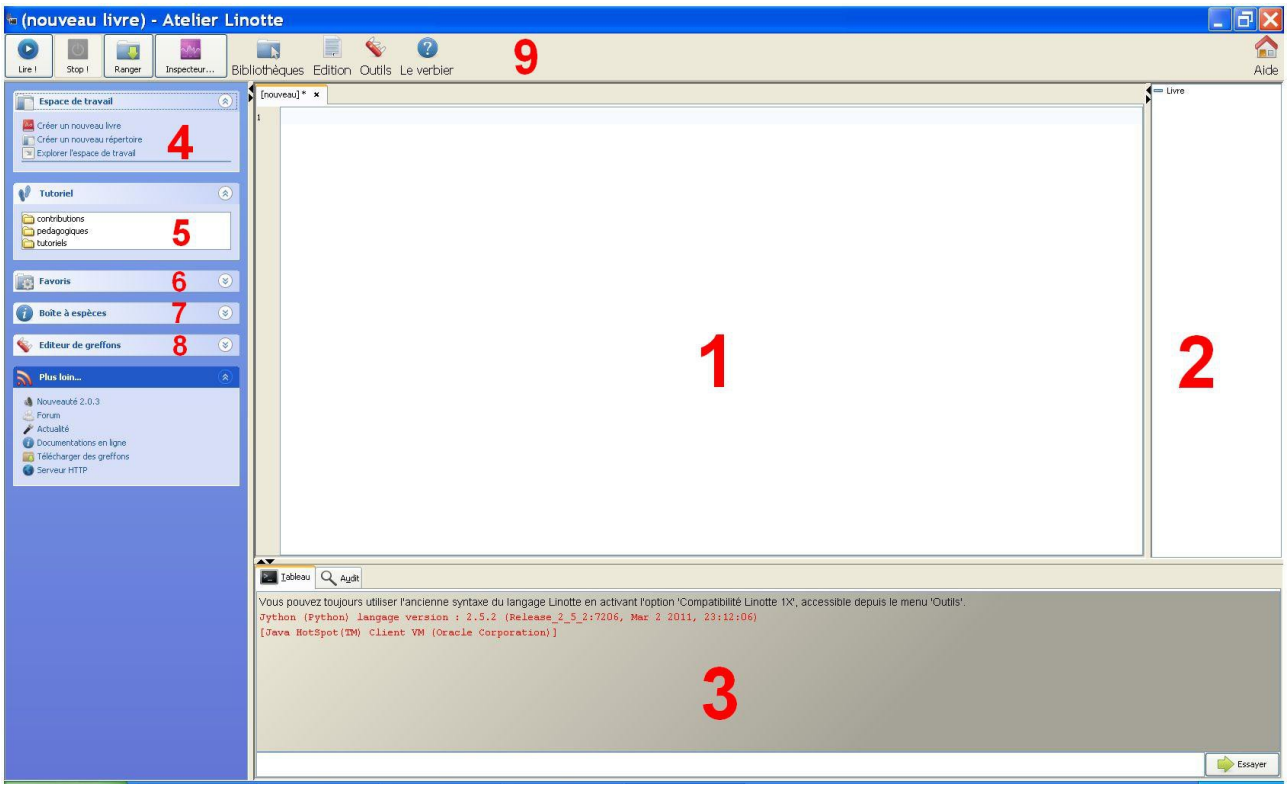
**Sous Microsoft Windows :**

Le lancement de Linotte se fait en cliquant sur l'icône **Linotte.exe** présent dans le répertoire d'installation.

**Sous Linux et MacOS :**

Dans le répertoire d'installation de Linotte, il faut cliquer sur le fichier **Linotte.sh**.

Voici comment se compose l'atelier Linotte :




1. Il s'agit du **cahier**, l'élément principal de l'atelier. C'est ici que vous allez écrire et modifier vos programmes. En fait, en Linotte, on appelle des programmes : des livres. Les livres se présentent sous forme d'onglets et représentent chacun un programme.

2. **Le sommaire**. Il permet de visualiser l'organisation de votre livre.

3. **Le tableau**. Il affiche les résultats du livre, ainsi que les messages d'erreur.

4. **L'espace de travail**. Il permet de créer un nouveau livre dans un dossier prédéfini. Sous Windows, le répertoire par défaut est : C:\Documents and Settings\"Votre nom d'ordinateur"\Mes documents\EspaceDeTravail. Il est également possible de créer un nouveau répertoire.

5. **Le tutoriel**. Il permet d'accéder à tous les exemples de programmation en Linotte disponibles.

6. **Les favoris**. A gauche de votre livre, se trouve une colonne indiquant le numéro de chaque ligne. En faisant un clique droit sur un numéro, cela vous permet de le mettre en favoris et donc, de pouvoir accéder directement à cette ligne.

7. **La boite à espèces**. Elle permet d'afficher toutes les caractéristiques et fonctions du vocabulaire utilisé en Linotte.

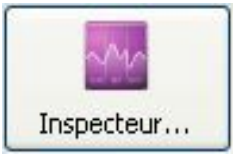
8. **L'éditeur de greffons**. Les greffons ont pour but d'enrichir le Linotte à partir de programmes écrits dans d'autres langages.

9. **La barre d'outils**. Elle se compose de plusieurs boutons et menus :

**lire !** : exécute votre livre. A noter que si vous avez plusieurs livres ouverts sur le cahier, l'atelier ne peut lire qu'un seul livre à la fois.

**Stop !** : stoppe l'exécution de votre livre.

**Ranger** : permet d'enregistrer votre livre dans le dossier de votre choix.



**Inspecteur** : permet de suivre le déroulement de votre livre.

**Bibliothèques** : ce menu se compose de cinq parties :

- Nouveau livre → Ouvre un livre vierge dans un nouvel onglet.
- Ouvrir un livre → Ouvre un livre déjà commencé dans un nouvel onglet.
- Ranger sous → Enregistre un livre déjà commencé dans un autre dossier (équivalent à Enregistrer sous.)
- Exporter au format PDF... → Enregistre un livre au format PDF
- Exporter au format PNG... → Enregistre un livre au format PNG
- Derniers livres ouverts → Contient l'historique des derniers livres utilisés

**Edition** : on y retrouve :

- Annuler → Permet d'annuler une action que l'on vient de faire.
- Rétablir → Permet de rétablir une action que l'on vient d'annuler.
- Copier, coller et couper → De grand classiques, je ne vous les présente pas.
- Rechercher → Permet de retrouver un mot dans le livre.

**Outils** : on y retrouve :

- Indenter le livre → Permet de mettre en forme "officielle" le livre que l'on vient d'écrire.
- Afficher la toile → La toile est une fenêtre permettant d'afficher des éléments graphiques.
- Manageur de greffons → Permet d'afficher tous les greffons disponibles et de les mettre à jour.
- Options → On y trouve les fonctions suivantes :
    - Popup message → Permet d'afficher les messages du tableau dans une fenêtre.
    - Mémoriser les livres → A l'ouverture de l'atelier, permet de réafficher les livres qui sont restés ouverts lors de la dernière fermeture de l'atelier.
    - Bonifier le cahier → Permet d'améliorer la lisibilité du code.
    - Manageur de styles → Permet de modifier toutes les couleurs de son livre.
    - Compatibilité Linotte 1.X → Permet d'écrire son livre en Linotte 1.X

**Le verbier** : contient toutes les instructions disponibles en Linotte. Si vous ne savez plus comment utiliser ou écrire l'une d'elles, le verbier vous sera très utile !

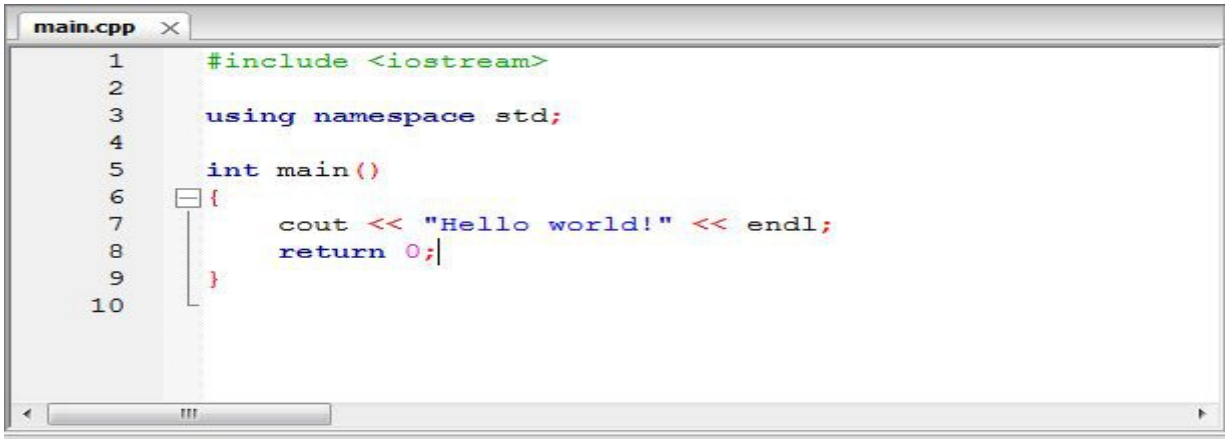
**Aide** : ce menu se compose de trois parties :

- Thèmes → Permet de changer de thème. Il y en a six par défaut.
- Remerciement → Ouvre la fenêtre des remerciements.
- A propos → Informations techniques et juridiques sur le langage Linotte.

# Votre premier livre !

Nous allons écrire notre premier programme en Linotte !
Celui-ci va consister à afficher un message de bienvenue sur le tableau.
En effet, à l'image des autres langages de programmation, le "*Hello world !*", que l'on traduira ici par "*Bonjour tout le monde !*", est un programme simple utilisé traditionnellement pour faire une démonstration rapide du langage.

Le "*Hello World !*" en C++ :



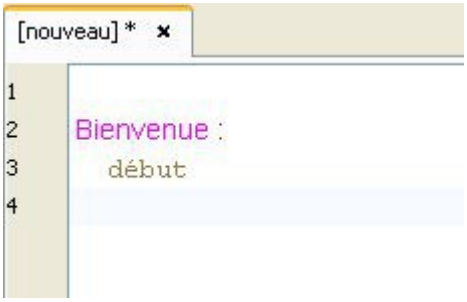

Voyons comment écrire cet exemple en Linotte !

## Les fonctions

Ouvrez un nouveau livre (**Bibliothèques** > **Nouveau livre**).
Allez dans **Le verbier** et sélectionnez **-Fonction**.
Vous devriez avoir ceci :


Ceci représente une **fonction**.
En effet, à l'image d'un livre, ceci va permettre de structurer votre code en paragraphes.

Commençons par donner un titre à notre fonction : remplacez le F par Bienvenue :




C'est dans cette fonction que nous allons écrire notre livre, qui affichera notre message.

# Les variables :

Et bien, si on déclarait notre message ?
Ajoutez ceci :


En écrivant ceci, vous venez de **déclarer une variable**.

Qu'est-ce qu'une **variable** ?

En voici deux types :
**- Les nombres**
**- Les chaînes de caractères**
Aussi appelé **texte** (Exemple : "*Marie a acheté des pommes*" est une chaîne de caractères)

Ici, la variable message de type texte représentera donc la phrase que l'on veut afficher sur le tableau.
Mais pour l'instant, notre variable est vide.
Ajoutons-y notre message :


Voila, notre message est prêt !

Vous remarquerez les guillemets **""** autour de la chaîne de caractères. Ils sont là pour différencier les textes et les nombres. **3** est un nombre et **"3"** est un texte.

Pour résumer, une variable se caractérise donc par un nom, un type et une **valeur** (représentée ici par le texte "*Bonjour tout le monde !*").

**Une petite précision s'impose :**

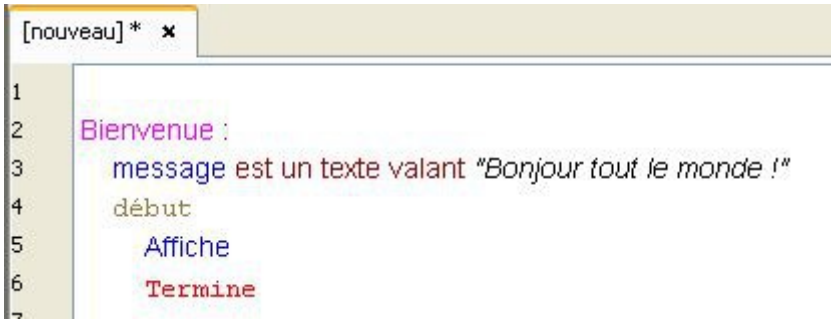
Les majuscules n'ont aucune incidence dans le nom des variables : MessAGE = Message = message
En revanche, les accents sont très importants : Méssage ≠ Mèssage ≠ Message

# Les verbes :

Pour le moment, notre message est prêt à l'emploi, mais notre livre ne fera rien.
Et oui, pour ça, **l'interprète** (l'atelier Linotte) a besoin de **verbes**.
Les **verbes** sont les actions que va effectuer notre livre.

Nous allons utiliser deux verbes : **Afficher** et **Terminer**.

Le verbe **Afficher** permet d'afficher quelque chose à l'écran.
Le verbe **Terminer** permet d'arrêter le programme.




Vous remarquerez qu'on a ajouté les verbes après le mot début.
En effet, le mot début sert à lancer les actions du programme. Il sépare les variables, situées au dessus, des verbes, en dessous.

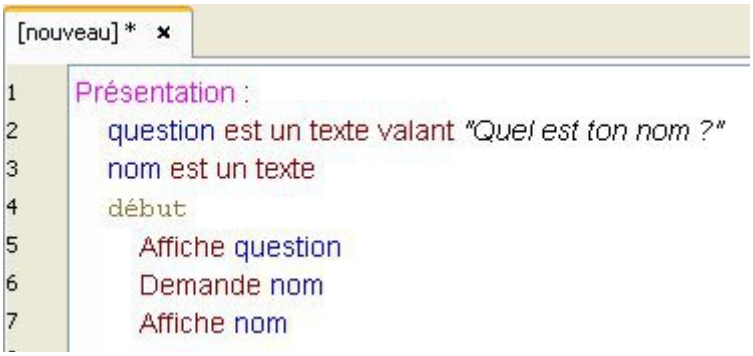
Cette organisation est obligatoire ! Si vous ne la respectez pas, votre programme ne fonctionnera pas.

Mais le verbe **Afficher**, utilisé seul, n'affiche rien. Il faut donc lui indiquer la chaîne de caractères à afficher, grâce à la variable message.

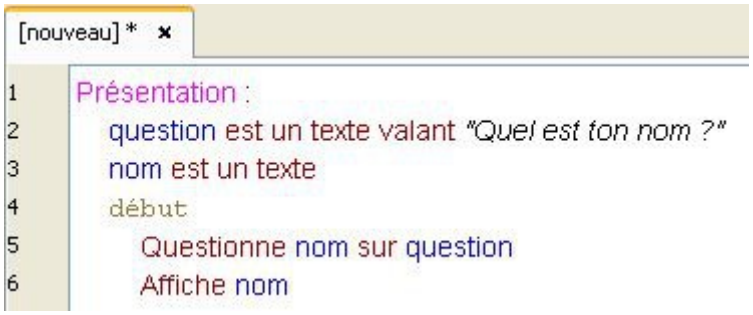
Félicitations ! Vous venez d'écrire votre premier programme en Linotte.
Si vous cliquez sur le bouton **Lire!**, vous verrez alors apparaître votre message sur le tableau de l'atelier.

# Interaction avec l'utilisateur

Le langage Linotte permet d'interagir directement avec l'utilisateur, à l'aide du verbe **Demander**. Écrivez ceci dans votre livre et lancez le programme :




Le verbe **Questionner** peut également être utilisé :




Le verbe **Questionner** à besoin de deux choses pour fonctionner :
- la variable question, de type texte, qui contient la question à poser.
- la variable nom, de type texte, qui contiendra votre réponse.

Les réponses aux verbes **Demander** et **Questionner** peuvent également être de type **nombre**.

# Les mathématiques

Le langage Linotte peut nous permettre d'effectuer toutes sortes d'opérations mathématiques.
Pour cela, il existe quatre verbes : **Ajouter**, **Soustraire**, **Multiplier** et **Diviser**.
Exécutez ce livre :

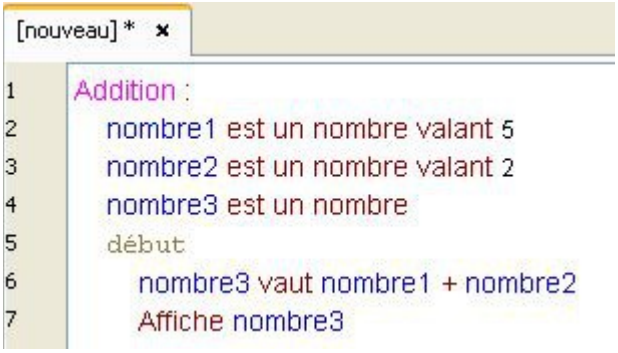

Dans cet exemple, nous avons déclaré deux variables de type nombre.
Notre programme consiste alors à additionner le nombre1 au nombre2 et à afficher le résultat.

Les trois autres verbes fonctionnent de la même façon.

Néanmoins, on préférera utiliser les symboles habituels : +, -, * et /

On écrira alors la même addition comme ceci :


Mais on peut également faire comme ceci :




Dans cet exemple, nous avons déclaré trois variables de type nombre.
Notre programme consiste alors à additionner les nombre1 et nombre2 et à afficher le résultat dans

le nombre3.
La différence avec l'exemple précédent c'est qu'ici nombre1 est toujours égal à **5** et nombre2 toujours égal à **2**.

**Une remarque s'impose :**

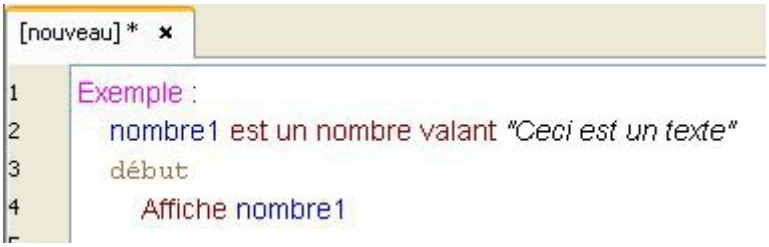
Vous aurez sûrement constaté l'absence de guillemets **""** dans les variables de type nombre.

```
nombre1 est un nombre valant 5
```

En effet, comme je vous l'ai expliqué plus haut, les guillemets **""** permettent de différencier les nombres des textes.
Pourtant, on peut très bien écrire ceci :


Ici le nombre **3** est considéré comme un texte, par la présence des guillemets **""**.

Dans certain cas, l'interprète peut donc transformer un nombre en texte.
L'inverse n'existe pas.
En effet, on ne peut pas écrire ceci :




Cet exemple ne fonctionne pas !

J'en profite pour signaler qu'en Linotte, pour écrire un nombre décimal, on utile le point et non la virgule, comme ceci :

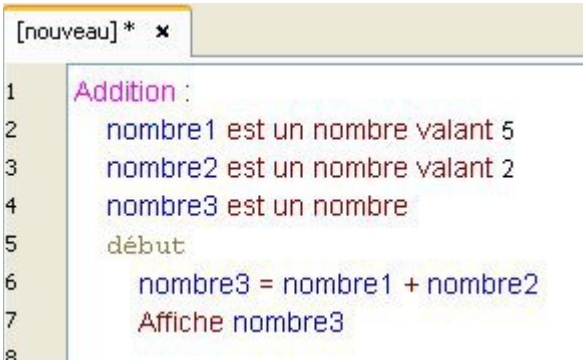
Et voici le résultat :


On constate qu'un nombre peut être négatif.

Et en Linotte, la taille des chaînes de caractères et des nombres est illimitée !

# Informatique contre Mathématiques :

Vous remarquerez que dans mes exemples précédents, je n'ai pas utilisé le signe = .
Pourtant, il existe bien en Linotte :




Cet exemple fonctionne.

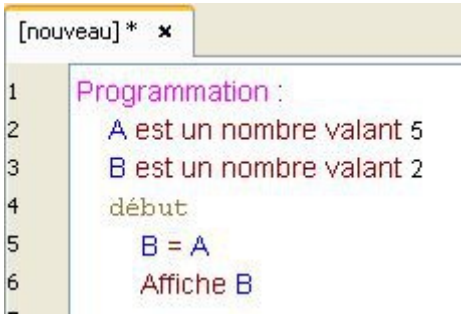
Mais j'ai préféré utiliser le verbe **Valoir** pour ne pas vous induire en erreur.
En effet, l'informatique et les mathématiques sont deux choses complètement différentes.

Si en mathématiques on écrit : A = B ou B = A
Les deux propositions sont équivalentes.

Mais en programmation, si vous écrivez ceci :


Ici A = **2**.

Et si vous écrivez l'inverse :




Ici B = **5**.

Les deux propositions ont un résultat différent !

De même, j'ai écrit :

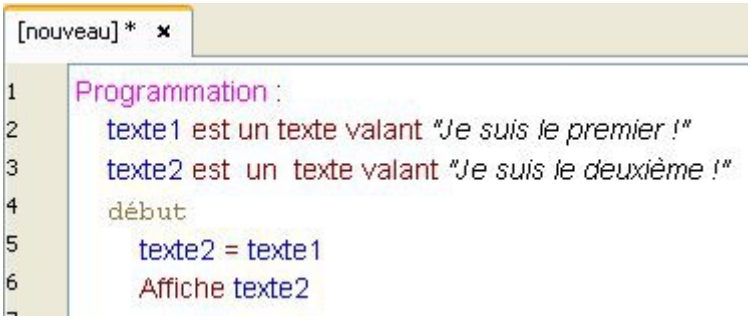
```
nombre2 vaut nombre2 + nombre1
```

Que l'on peut écrire :

```
nombre2 = nombre2 + nombre1
```

En mathématiques cette proposition constitue une équation sans solution.
Alors qu'en programmation, elle constitue une action extrêmement courante.

Et je finirais par cet exemple, qui n'a pas son équivalent en mathématiques, mais qui existe en Linotte :




# Voici les fonctions mathématiques disponibles :

a vaut b
Permet d'affecter la valeur b dans la valeur a.

Cette fonction est particulière car elle fonctionne également avec les variables qui ne sont pas des nombres.

a vaut b+c
Permet de calculer la somme de b et de c et de stocker la valeur dans a. On peut faire de même avec b-c, b*c ou b/c.

a vaut carré b
Permet de calculer le carré de b et de stocker la valeur dans a.

a vaut cube b
Permet de calculer le cube de b et de stocker la valeur dans a.

a vaut b puiss c
Permet de calculer b puissance c et de stocker la valeur dans a.

a vaut racine b
Permet de calculer la racine carrée de b et de stocker la valeur dans a.

a vaut abs b
Permet de calculer la valeur absolue de b et de stocker la valeur dans a.

a vaut arrondi b
Permet de calculer l'arrondi de b et de stocker la valeur dans a.

a vaut entier b
Permet de calculer la partie entière de b et de stocker la valeur dans a.

a vaut décimal b
Permet de calculer la partie décimale de b et de stocker la valeur dans a.

a vaut b mod c
Permet de calculer le reste de b dans la division par c et de stocker la valeur dans a.

a vaut exp b
Permet de calculer la fonction exponentielle de b et de stocker la valeur dans a.

a vaut log b
Permet de calculer le logarithme décimal de b et de stocker la valeur dans a.

a vaut logn b
Permet de calculer le logarithme naturel de b et de stocker la valeur dans a.

a vaut sin b
Permet de calculer le sinus de b et de stocker la valeur dans a.

a vaut cos b
Permet de calculer le cosinus de b et de stocker la valeur dans a.

a vaut asin b
Permet de calculer l'arc sinus de b et de stocker la valeur dans a.

a vaut acos b
Permet de calculer l'arc cosinus de b et de stocker la valeur dans a.

a vaut atan b
Permet de calculer l'arc tangente de b et de stocker la valeur dans a.

# Le verbe Afficher

Le verbe **Afficher** est particulier : il est capable d'afficher un nombre, sans utiliser de variable !
Comme ceci :


En réalité, on utilise bien une variable. Mais ici, on a pas besoin de la déclarer.
En effet, on n'a pas eu besoin d'écrire :

```
nombre1 est un nombre valant 5
```

On parle alors de **variable anonyme**.

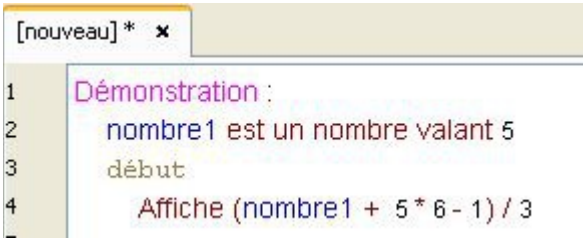
Et si le verbe **Afficher** peut afficher directement des nombres, il peut aussi afficher du texte !

Si on reprend notre tout premier programme, on aurait alors pu l'écrire comme ceci :


Vous l'aurez compris, en Linotte, comme dans les autres langages de programmation, il y a rarement qu'une seule façon d'écrire un programme. Il y en a généralement plusieurs : à vous de choisir celle qui vous semble la plus simple.

**Allons encore plus loin :**

Le verbe **Afficher** est capable d'effectuer directement des opérations mathématiques, par exemple :




Et le verbe **Afficher** peut même additionner des textes !

```
[nouveau] *
1  Bienvenue :
2      question est un texte valant "Comment allez-vous ?"
3      début
4          Affiche "Bonjour tout le monde ! " + question
```

Voilà le résultat :


**Une astuce :**

Pour effectuer un retour à la ligne avec le verbe **Afficher**, vous pouvez utiliser le symbole |, comme ceci :

```
[nouveau] *
1  Bienvenue :
2      question est un texte valant "Comment allez-vous ?"
3      début
4          Affiche "Bonjour tout le monde ! " + | + question
```

# Les commentaires

Les commentaires permettent d'annoter votre livre.
Ils n'ont aucun impact sur le fonctionnement de votre programme : ils sont simplement ignorés par l'interprète Linotte.
Pourtant, ils se révèlent souvent indispensables pour les développeurs : il leur permettent d'expliquer le fonctionnement de leur code.

Il existe deux façons d'écrire un commentaire :

Pour écrire un commentaire court, qui tient sur une seule ligne, il suffit de commencer par // puis d'écrire votre commentaire, comme ceci :

```
// Ceci est un commentaire court
```

Mieux, vous pouvez ajouter votre commentaire à la fin d'une ligne de code pour expliquer ce qu'elle fait :


Mais si votre commentaire tient sur plusieurs lignes, commencez votre annotation par /*, et finissez-la par */ :

```
/* Ceci est un long commentaire
qui tient sur
plusieurs lignes */
```

On peut également utiliser les commentaires pour désactiver temporairement une partie d'un livre :

# Les fonctions

Bien que je les ai évoquées un peu plus haut, les fonctions nécessitent d'avoir leur propre chapitre, afin d'expliquer leur fonctionnement.
Prenons cet exemple :


Une **fonction** est composée de deux choses :
- Un titre. Ici il s'appelle Exemple1.
- Le mot début.

Comme je l'ai déjà expliqué, le mot début sert à démarrer les actions du programme.
Au dessus, on indique les variables.
En dessous, on indique les actions que va effectuer le livre.

On a également vu qu'une variable était composée d'un nom, d'un type et d'une **valeur**.
On la déclare comme ceci :


Or, au dessus du mot début, on n'est pas obligé de déclarer la valeur de la variable.
On peut se contenter d'indiquer son nom et son type.


On remarque alors qu'on peut indiquer la valeur de la variable dans les actions du livre !

Mais quelle est la différence entre les deux exemples ?

Pour le comprendre, comparons-les directement dans le cahier :

```
[nouveau] *
1  Exemple1 :
2     nombre1 est un nombre valant 3
3     début
4        Affiche nombre1
5
6  Exemple2 :
7     nombre2 est un nombre
8     début
9        nombre2 vaut 5
10       Affiche nombre2
11
```

Si l'on exécute notre livre en cliquant sur le bouton **Lire !** de l'atelier, notre programme va s'arrêter à la ligne 5 !

En effet, pour que le programme puisse continuer, il faut lui indiquer de se rendre dans la fonction Exemple2.

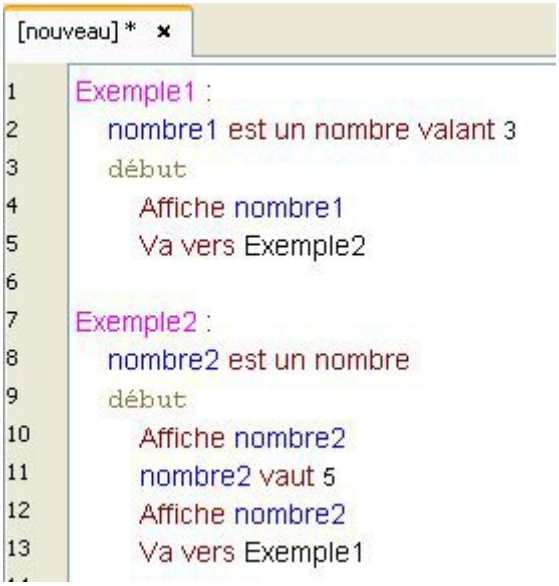
Pour cela, on utilise le verbe **Aller**, comme ceci :

```
[nouveau] *
1  Exemple1 :
2     nombre1 est un nombre valant 3
3     début
4        Affiche nombre1
5        Va vers Exemple2
6
7  Exemple2 :
8     nombre2 est un nombre
9     début
10       nombre2 vaut 5
11       Affiche nombre2
```

Et voici le résultat :

```
Tableau  Audit
3
5
```

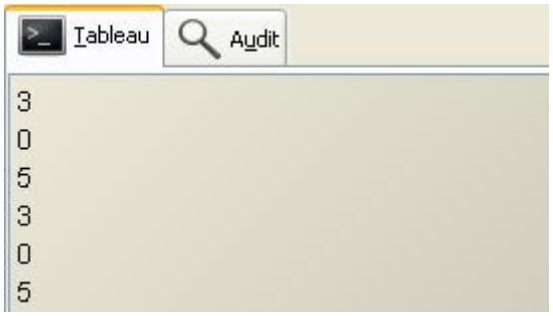
Ça fonctionne !
Mais ça ne répond toujours pas à notre question.
Modifions alors notre livre, comme ceci :




Attention ! Si vous exécutez ce programme, il va tourner en boucle !

En effet, comme à la fin de la fonction Exemple2, on lui dit de se rendre à la fonction Exemple1, ce programme n'a pas de fin !
Il faudra alors cliquer sur le bouton **Stop !** de l'atelier pour pouvoir l'arrêter.

Mais observons les premiers résultats :




Au premier passage, le livre va afficher ceci :
**3** : valeur de nombre1.
**0** : valeur de nombre2. Comme on ne lui a pas encore affecté une valeur, à ce moment là nombre2 est donc égal à **0**.
**5** : valeur de nombre2.

Puis le livre revient à la première fonction et fait donc un autre passage, qui donne ceci :
**3** : valeur de nombre1.
**0** : valeur de nombre2.
**5** : valeur de nombre2.

Avez-vous remarqué la différence ?

Lorsque l'on déclare la valeur de la variable comme ceci :

```
nombre1 est un nombre valant 3
```

Si l'on quitte la fonction, et que l'on y revient ensuite, la valeur de la variable ne change pas : elle est toujours égale à **3**.

Mais lorsque l'on indique la valeur de la variable dans les actions du livre :

```
Exemple2 :
   nombre2 est un nombre
   début
      nombre2 vaut 5
```

Lorsque l'on quitte la fonction et que l'on y revient, la valeur de la variable est alors égale à **0**.

Car, en vérité, lorsque l'on quitte une fonction, la variable est détruite.

Quand on reviens dans notre fonction, on recréer alors une nouvelle variable.

En déclarant la valeur d'une variable dès le départ (avant le mot début), on donne alors à notre variable une valeur initiale, ici **3**.

Si ce n'est pas le cas, comme dans Exemple2, celle-ci est alors réinitialisée à **0**.

Et si je fais ceci ? :

[nouveau] *

```
1  Exemple1 :
2     nombre1 est un nombre valant 3
3     début
4        nombre1 vaut 5
5        Affiche nombre1
```

Si l'on quitte la fonction, et que l'on y revient, la valeur de la variable nombre1 sera donc initialisée à **3**.

A noter que ce principe fonctionne avec tous les types de variable, comme les textes par exemple.

# La navigation dans le livre : point de vue programmation

Nous venons d'utiliser le verbe **Aller**.
Vous l'avez compris, il nous permet de se rendre à n'importe quelle fonction de notre livre.
On peut même faire ceci :

```
[nouveau] *
1  Exemple1 :
2     nombre1 est un nombre valant 3
3     début
4        Affiche nombre1
5        Va vers Exemple1
```

Attention, ce programme tourne en boucle !

Le verbe **Aller**, s'utilise à la fin d'une fonction.
En effet, si on fait ceci :

```
[nouveau] *
1  Exemple1 :
2     nombre1 est un nombre valant 3
3     début
4        Va vers Exemple2
5        nombre1 vaut 7
6        Affiche nombre1
7
8  Exemple2 :
9     nombre2 est un nombre
10    début
11       nombre2 vaut 5
12       Affiche nombre2
```

Voici le résultat :


**nombre1** ne s'affiche pas.

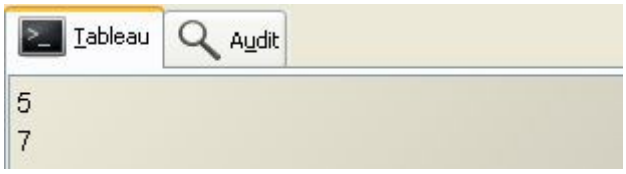
En effet, comme à la ligne 4 on se rend dans la fonction Exemple2, les ligne 5 et 6 ne sont pas exécutées !

Pour pouvoir se déplacer entre les fonctions sans avoir ce problème, on utilise alors deux autres verbes : **Parcourir** et **Revenir**.

Prenons cet exemple :

```
[nouveau] *
1  Exemple1 :
2     nombre1 est un nombre valant 3
3     début
4        nombre1 vaut 7
5        Parcours Exemple2
6        Affiche nombre1
7
8  Exemple2 :
9     nombre2 est un nombre
10    début
11       nombre2 vaut 5
12       Affiche nombre2
13       Reviens
```

Voici le résultat :




Si on analyse ces résultats, voici ce que sa donne :

Le mot Parcours dans Exemple1 permet de se rendre à la fonction Exemple2.

nombre2 s'affiche, soit **5**.

Le mot Reviens permet de retourner au point de départ : on revient donc à la ligne 5, et la ligne 6 peut s'exécuter !

On note que **nombre1** affiche **7**, et non **3**.

Et oui, car contrairement au verbe **Aller**, le verbe **Parcourir** ne détruit pas la variable.
En effet, utilisé avec le verbe **Revenir**, ce verbe permet de nous rendre dans la fonction Exemple2 sans quitter notre fonction principale Exemple1.
La variable nombre1 n'est alors pas détruite et conserve donc sa dernière valeur.

# La navigation dans le livre : point de vue utilisateur

Nous allons maintenant voir quelques fonctionnalités de l'atelier Linotte.

Si vous avez recopié l'exemple précédent sur le cahier de votre atelier Linotte, déplacez la fonction Exemple2 jusqu'à la ligne 31, en utilisant la touche **[entrée]**. (Il faut que Exemple2 ne soit plus visible à l'écran : descendez plus bas si nécessaire.)
Maintenant, retournez à la fonction Exemple1 et double-cliquez sur Exemple2.
Comme vous pouvez le voir, double-cliquer sur le nom d'une fonction vous y amène

automatiquement. Pratique !

Maintenant, si vous faites un clique droit dans le cahier, vous verrez un menu apparaître. Cliquez sur l'onglet **Retour** et vous reviendrez alors à la fonction précédente.

A noter que pour revenir en arrière, vous pouvez utiliser les touches **Alt + Flèche Gauche**.

Vous pouvez également voir le sommaire, à droite du cahier.
Vous constatez que les noms de vos fonctions y sont affichés.
Sélectionner l'un d'eux vous l'affichera automatiquement dans le cahier.

Pour l'instant, ces fonctionnalités ne sont pas très importantes, mais lorsque vous aurez un très grand livre avec des centaines de fonctions, croyez-moi, vous les trouverez bien utiles.

Enfin, vous pouvez même naviguer entre plusieurs livres.

Pour cela, ouvrez  un nouveau livre : **Bibliothèques > Nouveau livre**

En utilisant les touches **Ctrl + Page bas** vous affichez alors le livre précédent.

En utilisant les touche **Ctrl + Page haut** vous affichez le livre suivant.

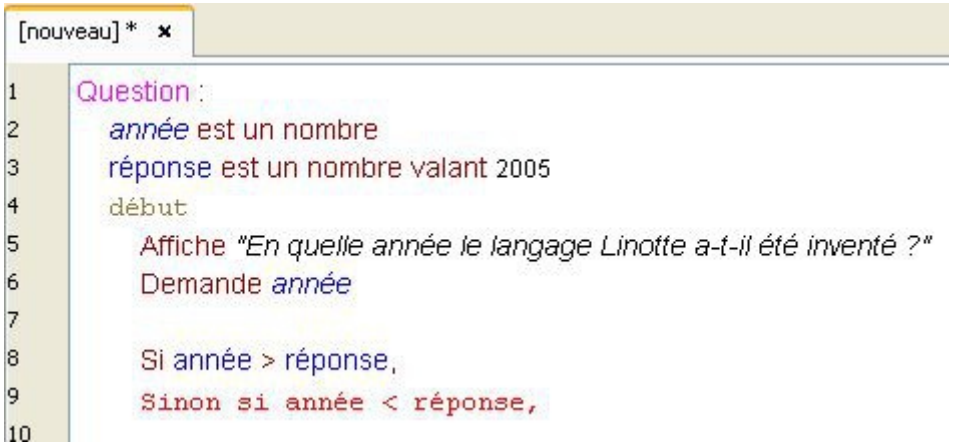
Et voici un dernier raccourci :

En utilisant les touches **Ctrl + M** vous pouvez agrandir ou réduire votre livre.

# Les conditions

Nous allons maintenant rendre nos programmes plus dynamiques.
Prenons cet exemple :


Suivant la réponse de l'utilisateur, on voudrait afficher des phrases différentes.
Pour cela, nous allons utiliser deux conditions, comme ceci :




A partir de la ligne 8, on analyse la valeur de la variable année, choisie par l'utilisateur, et on la teste en posant deux conditions :
- si la variable année est supérieure à la variable réponse.
- si la variable année est inférieure à la variable réponse.

Il nous reste plus qu'à attribuer des actions différentes à chacune de ces possibilités :


Voici ce qu'on a ajouté :
- Si la variable année est supérieure à la variable réponse, alors on va vers la fonction Avant.
- Sinon si la variable année est inférieure à la variable réponse, alors on va vers la fonction Après.

Chaque fonction nous permet alors d'afficher un message différent (ligne 12 et 17).
Puis les lignes 13 et 18 nous font revenir à la ligne 5, afin de reposer la même question.

Mais si l'utilisateur donne la bonne réponse en écrivant "*2005*" ?

Dans ce cas, la variable année ne sera ni inférieure, ni supérieure à la variable réponse. Elle lui sera égale. Les lignes 7 et 8 ne pourront donc pas s'exécuter.

Il nous faut alors conclure la condition en utilisant le mot **Sinon**, comme ceci :

```
[nouveau] *
1  Question :
2     année est un nombre
3     réponse est un nombre valant 2005
4     début
5        Affiche "En quelle année le langage Linotte a-t-il été inventé ?"
6        Demande année
7        Si année > réponse, va vers Avant
8        Sinon si année < réponse, va vers Après
9        Sinon affiche "Bravo, tu es trop fort !"
10
11 Avant :
12    début
13       Affiche "Non, non... c'est avant !"
14       Va vers Question
15
16 Après :
17    début
18       Affiche "Non, non... c'est après !"
19       Va vers Question
```

Ici, peu importe la réponse de l'utilisateur, notre programme pourra alors afficher une phrase correspondante.

# Les blocs

Dans notre exemple précédent, on constate qu'après l'utilisation d'une condition, nous pouvons effectuer une action.

```
Condition              Action
Si année > réponse, va vers Avant
```

Et nous ne pouvons en faire qu'une seule !

Pour pouvoir en faire plusieurs, nous utilisons les fonctions Avant et Après. En effet, dans chacune d'elles nous effectuons deux actions : afficher la phrase **et** revenir à la fonction Question.

Mais il existe un autre moyen pour pouvoir effectuer plusieurs actions après une condition.

Pour cela, il faut écrire nos actions dans un **bloc**.

Un **bloc** débute avec le verbe **Lire** et se termine avec le verbe **Fermer**.

Voyons voir ce que sa donne sur notre code :

[nouveau] *

```
1  Question :
2     année est un nombre
3     réponse est un nombre valant 2005
4     début
5        Affiche "En quelle année le langage Linotte a-t-il été inventé ?"
6        Demande année
7        Si année > réponse, lis
8           Affiche "Non, non... c'est avant !"
9           Va vers Question
10       ferme
11
12       Sinon si année < réponse, lis
13          Affiche "Non, non... c'est après !"
14          Va vers Question
15       ferme
16
17       Sinon affiche "Bravo, tu es trop fort !"
```

Comme vous pouvez le voir, les blocs, délimités par les mots **lis** et **ferme**, nous permettent d'effectuer plusieurs actions après l'utilisation d'une condition !

Nous n'avons alors plus besoin des fonctions Avant et Après.

**Détaillons l'écriture des blocs :**

Nous commençons par remplacer l'action à effectuer par l'action **Lire** :

```
   Condition          Action
Si année > réponse, lis
```

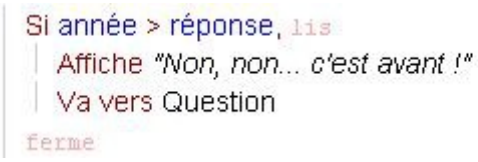
Attention à ne pas oublier la virgule entre la condition et l'action.

Ensuite, nous pouvons indiquer toutes les actions que nous voulons faire :

```
Si année > réponse, lis
   Affiche "Non, non... c'est avant !"
   Va vers Question
```

Nous pouvons en écrire autant qu'on veut, à raison d'une action par ligne.

Lorsque l'on à finit d'énumérer toutes les actions, nous pouvons refermer le bloc à l'aide du verbe **Fermer** :




Vous pouvez constater que l'on a avancé les actions d'une tabulation, par rapport à l'ouverture et à la fermeture du **bloc**.

Pensez à respecter cette organisation : elle facilitera la lecture de votre code.

Vous pouvez retrouver les blocs dans le verbier, dans l'onglet -**Bloc**.

**Détaillons maintenant les conditions :**

Voici le schéma que nous venons d'utiliser :

**Si**...,...
**Sinon si**...,...
**Sinon**

Vous pouvez y ajouter autant de conditions que vous le désirez :

**Si**...,...
**Sinon si**...,...
**Sinon si**...,...
**Sinon si**...,...
…
**Sinon**

Vous pouvez également se faire suivre des conditions indépendantes les unes des autres :

**Si**...,...
**Si**...,...
**Si**...,...

Quelle est la différence entre ce schéma et le précédent ?

Dans ce schéma :

| | |
|---|---|
| **Si**...,... | Si cette première condition n'est pas remplie... |
| **Sinon si**...,... | La seconde condition sera lue automatiquement par l'interprète. |

Dans ce schéma :

| | |
|---|---|
| **Si**...,... | Si cette première condition n'est pas remplie... |
| **Si**...,... | La seconde condition sera lue par l'interprète seulement si elle est remplie. |

Et enfin, vous pouvez même utiliser des conditions à l'intérieur d'autres conditions !

Voyez plutôt :

[nouveau] *

```
1  Question :
2     année est un nombre
3     réponse est un nombre valant 2005
4     début
5        Affiche "En quelle année le langage Linotte a-t-il été inventé ?"
6        Demande année
7        Si année > réponse, lis
8           Si année > 2010,  affiche "Non, non... c'est bien avant !"
9           Si année <= 2010, affiche "Non, non... c'est avant ! Tu n'es pas très loin..."
10          Va vers Question
11       ferme
12
13       Sinon si année < réponse, lis
14          Si année < 2000, affiche "Non, non... c'est bien après !"
15          Si année >= 2000, affiche "Non, non... c'est après ! Tu n'es pas très loin..."
16          Va vers Question
17       ferme
18
19       Sinon affiche "Bravo, tu es trop fort !"
```

Le signe <= signifie "inférieur ou égal". Le signe >= signifie "supérieur ou égal".

De la même façon on pourrait donc insérer des blocs dans d'autres blocs, comme par exemple, en effectuant plusieurs actions après les conditions Si année > 2010, et Si année <= 2010, :

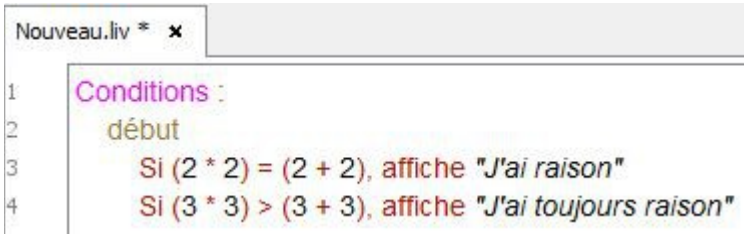
```
Si année > réponse, lis
   Si année > 2010, lis
      Affiche "Non, non... c'est bien avant !"
      Va vers Question
   ferme
   Si année <= 2010, lis
      Affiche "Non, non... c'est avant ! Tu n'es pas très loin..."
      Va vers Question
   ferme
ferme
```

L'exemple précédent fait la même chose mais son écriture est meilleure car l'instruction **Va vers Question** n'est alors employée qu'une fois au sein du bloc **Si année > réponse,** .

On remarque également que les conditions sont capables d'utiliser les **variables anonymes** :

```
Si année > 2010
```

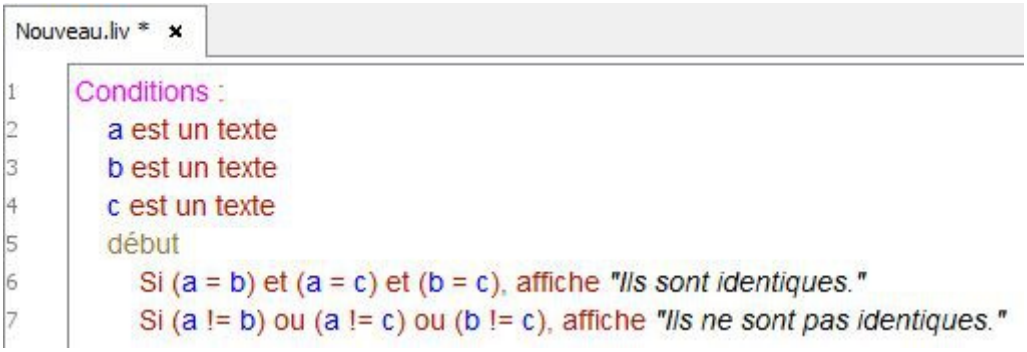
Et, comme le verbe **Afficher**, elles peuvent également utiliser les mathématiques !
Pour cela, il suffit d'utiliser des parenthèses, comme dans cet exemple :




**Une remarque :**

Dans beaucoup de langages de programmation, dans le cadre d'une condition le signe égal ne s'écrit pas = mais == .
En Linotte, les deux écritures sont possible.
Le contraire de égal s'écrit **!=** .

L'utilisation des parenthèses nous permet également de cumuler plusieurs conditions :




Pour pouvoir cumuler des conditions, on utilise alors les mots **et** et **ou**.

# Voici la liste des conditions disponibles :

| | |
|---|---|
| si ? = ? , … | S'écrit aussi si ? == ? , ... |
| si ? != ? , ... | |
| si ? > ? , ... | |
| si ? < ? , ... | |
| si ? >= ? , ... | |
| si ? <= ? , ... | |
| si ? est en collision avec ? , ... | Cette condition s'utilise seulement avec des objets graphiques. |
| si ? est vide , ... | |
| si ? est non vide , ... | |
| si ? est vrai/faux , ... | |
| si ? contient ? , ... | |
| si ? , ... | |

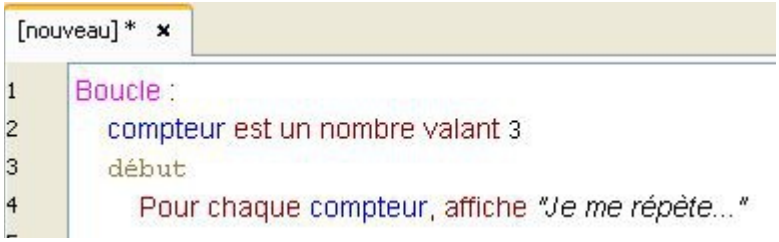
# Les boucles

Jusqu'à présent nous avons effectué des actions différentes les unes à la suite des autres.
Voyons maintenant comment répéter un certain nombre de fois une même action.

Nous nous baserons sur cet exemple :


## Pour chaque... ,...

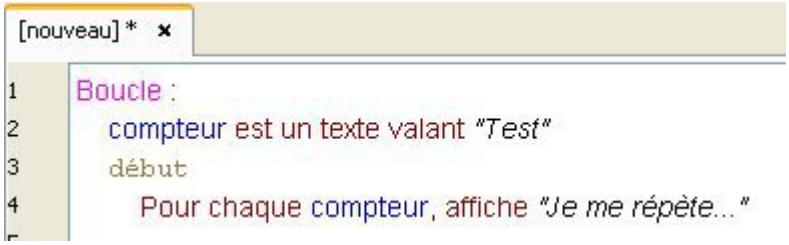
Voici le même exemple avec une boucle :




L'interprète va exécuter autant de fois l'action affiche que la **valeur** de la variable compteur.

Mais la boucle **Pour chaque...,...** est particulière.
En effet, son utilisation diffère suivant le type de variable qu'elle utilise.

Essayons avec une variable de type texte :




Si vous exécutez ce code, le message "*Je me répète...*" va s'afficher quatre fois sur le tableau.

Mais pourquoi ?

Pour comprendre ce qui se passe, servons-nous du **joker**.

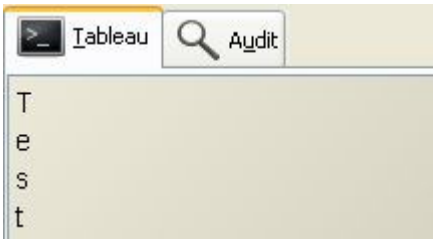
# Le joker :

Le **joker** est une variable que l'on utilise exclusivement dans les boucles.
On va s'en servir pour afficher ce que fait la boucle, lors de ses quatre passages.
Écrivez ce code :


Le joker est une variable particulière. En effet, son type et sa **valeur** dépendent du contenu de la boucle, à chacun de ses passages.

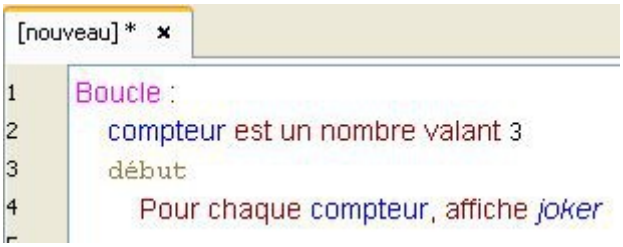
Dans une boucle, peu importe le type de la variable, le **joker** peut afficher directement son contenu, à chaque passage !

Voyons voir ce que ça donne :



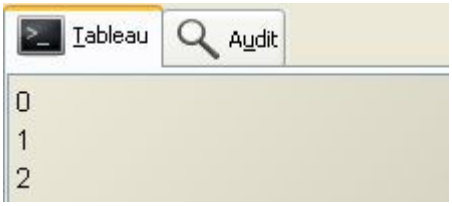

A chaque passage dans la boucle, l'interprète a alors affecté au joker une lettre de la variable compteur.

La boucle **Pour chaque...,...** , utilisé avec une variable de type texte, consiste donc à en décomposer toutes les lettres.
Utilisons maintenant le joker sur notre ancien exemple :




Que contient notre boucle ?

Voici le résultat :




La boucle ayant effectuée trois tours, l'interprète a alors affecté au joker une valeur correspondante au numéro de chaque passage.

La boucle **Pour chaque...,...** , utilisé avec une variable de type nombre, consiste donc à compter d'un nombre de fois correspondant à sa valeur.

C'est ainsi que notre phrase "*Je me répète...*" s'est affichée trois fois lors de notre premier exemple.

Pourtant on remarque que le résultat de l'exemple est : **0 1 2** et non **1 2 3**...

En effet, en programmation, la numérotation commencent à partir de **0**.
Pour commencer à compter à partir de 1, il faudrait alors utiliser une autre boucle.
Comme celle que nous allons voir dans le chapitre suivant.

**Une remarque :**

La boucle **Pour chaque...,...** , peut également s'écrire **Pour...,...**

Notre exemple précédent pourrait ainsi s'écrire comme ceci :


Les boucles peuvent également utiliser les **variables anonymes**.

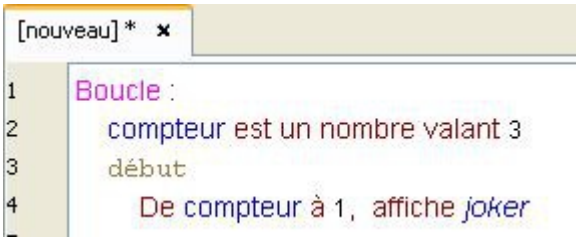
# De... à... ,...

Cette boucle ne fonctionne qu'avec des nombres.

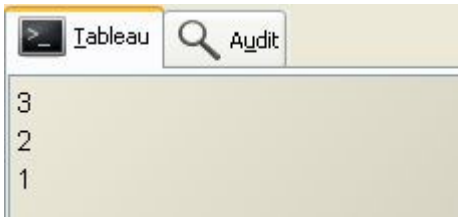
Prenons l'exemple précédent mais avec cette nouvelle boucle :

Et voici le résultat :


On peut aussi le faire en ordre décroissant :

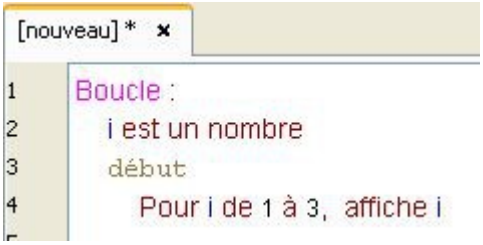



Ce qui donne :




# Pour... de... à... ,...

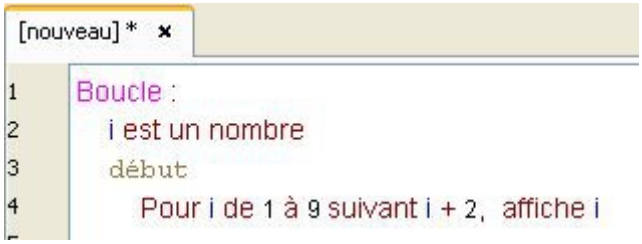
Cette boucle n'utilise pas le joker, nous permettant alors de choisir la variable que l'on va utiliser.
Voici un exemple :



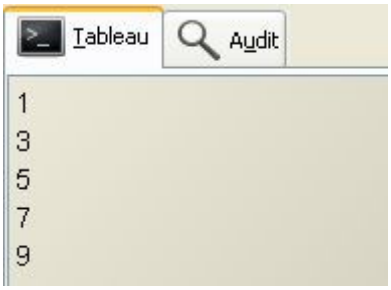

Et son résultat :


Ce programme affiche de nouveau les nombres **1**, **2**, **3**, le rôle du joker étant ici rempli par la variable **i**.

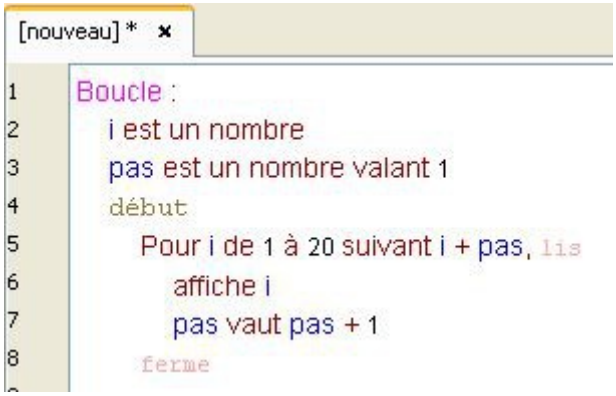
Un intérêt de cette boucle est qu'elle permet de fixer un pas différent de 1, autrement dit de faire en sorte que la variable augmente, par exemple, de deux en deux, à chaque passage dans la boucle :




La variable **i** étant initialisée à **0**, l'instruction suivant **i** + **2** signifie que **i** augmentera désormais de 2 en 2.
Ainsi, on obtient ceci :

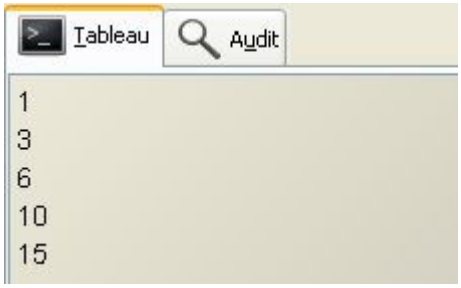



Allons encore plus loin :
Le pas lui même peut être une variable, ce qui permet d'obtenir une boucle avec un pas pouvant être modifié :




Les blocs s'utilisent aussi avec les boucles !

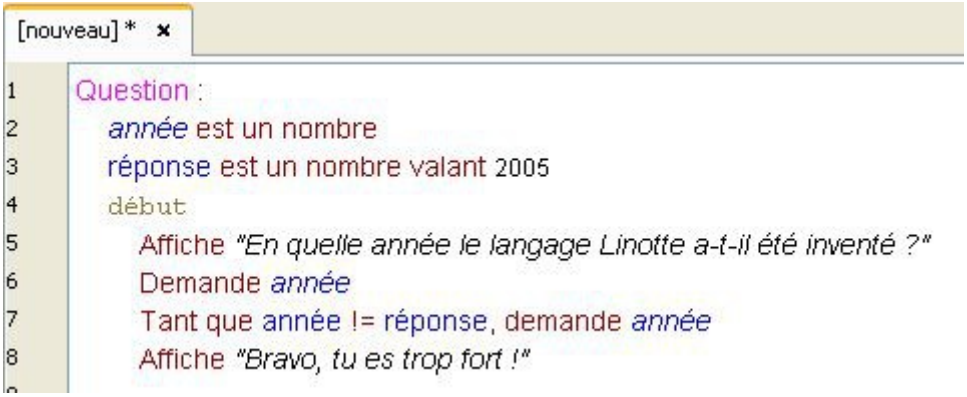
Au premier tour de la boucle, **i** étant initialisée à **0** et la variable pas valant **1**, le pas est alors fixé à 1.
Puis, à chaque tour, la variable pas étant incrémentée de **1**, le pas augmente alors de 2, puis 3, etc...
**i** augmente alors successivement de 1 en 1, puis de 2 en 2, puis de 3 en 3, etc...
La boucle s'arrête lorsque la valeur de **i** atteint ou dépasse **20**.
Ce qui donne donc :




# Tant que... ,...

Cette boucle utilise les conditions !
En effet, elle ne s'arrête que si la condition n'est plus remplie.

Reprenons alors notre exemple utilisé dans le chapitre sur les conditions :




Tant que la condition sera validée, la ligne 8 ne sera pas exécutée.

# Les casiers

Jusqu'à présent nous avons vu deux types de variables : les nombres et les textes.
Voyons maintenant un nouveau type : les casiers.

Le type casier nous permet de construire une liste de variables.
Par exemple :


Pour déclarer une variable de type casier, il faut donc écrire son nom et son type, mais également le type de variables qu'elle va contenir : ici, les variables utilisées sont de type texte.

L'utilisation du pluriel sur le type des variables n'a pas d'incidence. On peut écrire : **casier de texte** ou **casier de textes**.

On peut également construire une liste de nombres, comme ceci :

```
numéros est un casier de nombres valant 11, 21, 33, 44
```

Vous remarquerez que chaque variable est séparée par une virgule.

Attention : toutes les variables contenues dans un casier doivent être du même type.

Si on regarde le résultat de notre premier exemple, voici ce que ça donne :

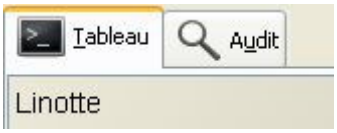

Mais imaginons que l'on veuille utiliser seulement la quatrième variable de la liste : "*Linotte*".
Pour cela, il faudra alors indiquer le nom du casier, suivit du numéro de la liste correspondant à notre variable, comme ceci :

```
Affiche langages{3}
```

On remarque la présence des accolades autour du numéro : elles sont obligatoires pour pouvoir accéder à notre variable.

Mais je croyais que tu voulais utiliser la quatrième variable, pas la troisième ?

Pourtant si on regarde le résultat :




C'est bien la quatrième variable de la liste qui s'affiche.

Et oui, souvenez-vous, en programmation la numérotation commence à partir de **0** !

Pour afficher la première variable de notre casier, il faut donc écrire :

```
Affiche langages{0}
```

Voyons maintenant comment insérer une nouvelle variable dans notre casier.
Pour cela, nous utilisons le verbe **Ajouter**.


A noter que le verbe **Ajouter** peut également utiliser les **variables anonymes**, comme ceci :


Voici le résultat :

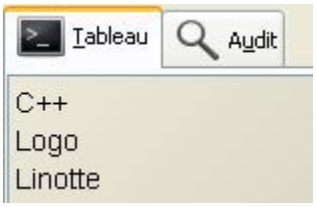

On remarque que la variable a été ajoutée à la fin de la liste.

Voyons maintenant comment retirer une variable d'un casier.

Pour cela, on utilise le verbe **Ôter**.

```
[nouveau] *
1  Casiers :
2     langages est un casier de textes valant "Java", "C++", "Logo", "Linotte"
3     début
4        ôte langages{0} de langages
5        Affiche langages
```

Et voici le résultat :




Désormais la variable "*C++*" se trouve à l'emplacement langages{0}, la variable "*Logo*" se trouve à l'emplacement langages{1}, etc...

Contrairement au verbe **Ajouter**, le verbe **Ôter** ne peut pas utiliser les **variables anonymes** :

```
[nouveau] *
1  Casiers :
2     langages est un casier de textes valant "Java", "C++", "Logo", "Linotte"
3     début
4        ôte "C++" de langages
5        Affiche langages
```

Cet exemple ne marche pas.
On ne peut donc pas accéder à une variable d'un casier si on ne connaît pas sa position.

**Heureusement, il existe plusieurs astuces :**

La première consiste à parcourir notre casier grâce à une boucle :

```
[nouveau] *
1  Casiers :
2     langages est un casier de textes valant "Java", "C++", "Logo", "Linotte"
3     début
4        Pour chaque langages, lis
5           Si joker contient "C++", ôte joker de langages
6        ferme
7        Affiche langages
```

En utilisant la boucle **Pour chaque...,...** le **joker** peut alors récupérer la valeur de chaque variable contenue dans le casier !

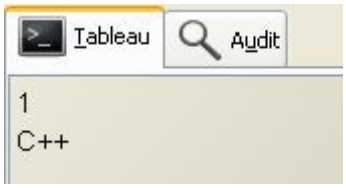
Il nous suffit ensuite d'utiliser une condition **Si... contient..., ...** et voilà le résultat :


La seconde astuce consiste à utiliser le verbe **Chercher** :

```
[nouveau] *
1  Casiers :
2    langages est un casier de textes valant "Java", "C++", "Logo", "Linotte"
3    pos est un nombre
4    début
5      Cherche pos, la position de "C++" dans langages
6      Si pos != 0, lis
7        pos vaut pos - 1
8        Affiche pos
9        Affiche langages{pos}
10     ferme
```

La variable **pos** utilisant le 0 pour indiquer qu'il n'a rien trouvé, la première variable du casier aura donc la position 1. Il faudra donc soustraire **pos** de **1** pour connaître sa véritable position.

Et voici le résultat :




Bien sûr, ces deux astuces nécessitent d'avoir des valeurs différentes pour chaque variable du casier.

**Autres astuces :**

Pour pouvoir ajouter des variables à un emplacement précis dans un casier, on peut utiliser le verbe **Insérer**.
Mais, comme pour le verbe **Chercher**, le verbe **Insérer** possède une exception : sa numérotation commence à partir de **1**.
Ainsi, pour ajouter une variable en première position d'un casier, il faut écrire ceci :

```
[nouveau] *
1  Casiers :
2     langages est un casier de textes valant "Java", "C++", "Logo", "Linotte"
3     début
4        insère "Python" à partir de 1 dans langages
5        Affiche langages
```

Et voici le résultat :


Et enfin, pour connaître le nombre de variables contenues dans un casier, vous pouvez utiliser le verbe **Mesurer**, comme ceci :

```
[nouveau] *
1  Casiers :
2     langages est  un  casier de textes valant "Java",  "C++", "Logo", "Linotte"
3     pos est un nombre
4     début
5        Mesure langages dans pos
6        Affiche pos
```

# Additionner des casiers

On peut additionner des casiers au sein d'un autre casier :

```
Nouveau.liv *
1  Casiers :
2     langages est un casier de textes valant "Java", "C++", "Logo", "Linotte"
3     langues est un casier de textes valant "Français", "Anglais", "Allemand"
4     addition est un casier de textes
5     début
6        addition = langages + langues
7        Affiche addition
```

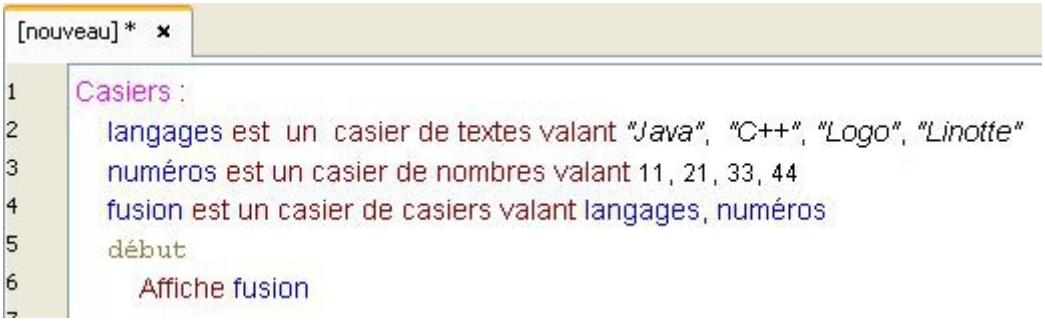
Attention : pour fonctionner, tous les casiers doivent être de même **type**.

Et voici le résultat :


Et on peut également additionner un casier avec des **variables anonymes** de même type :


# Les casiers de casiers

Un casier peut contenir... d'autres casiers !




On remarque que le casier **fusion** peut contenir des casiers de types différents.

Et voici le résultat :


Vous pouvez alors utiliser chaque casier séparément :

```
Nouveau.liv *
1  Casiers :
2     langages est un casier de textes valant "Java", "C++", "Logo", "Linotte"
3     numéros est un casier de nombres valant 11, 21, 33, 44
4     fusion est un casier de casiers valant langages, numéros
5     début
6        Ajoute "Python" dans fusion{0}
7        Affiche fusion
```

Ici on ajoute "*Python*" dans le casier langages, se trouvant à l'emplacement **0** du casier fusion.

Voici ce que sa donne :


Et vous pouvez même accéder à chaque variable de chaque casier, comme ceci :

```
[nouveau] *
1  Casiers :
2     langages est  un  casier de textes valant "Java",  "C++", "Logo", "Linotte"
3     numéros est un casier de nombres valant 11, 21, 33, 44
4     fusion est un casier de casiers valant langages, numéros
5     début
6        Affiche fusion{0 , 3}
```

Le premier chiffre de l'accolade correspond aux casiers, le second chiffre correspond aux variables.

On affiche donc :


# Les casiers anonymes

Nous pouvons créer des casiers sans avoir besoin de les déclarer, comme ceci :

```
[nouveau] *
1  Démonstration :
2        début
3           Affiche {"Bruno","Jean","Pierre"}
4
```

Ce qui donne :


Mais tout l'intérêt des casiers anonymes résident dans... les casiers de casiers anonymes :

```
[nouveau] *
1  Casiers :
2     fusion est un casier de casiers valant {"Java", "C++", "Logo", "Linotte"}, {11, 21, 33, 44}
3     début
4        Ajoute "Python" dans fusion{0}
5        ôte fusion{1 , 3} de fusion{1}
6        Affiche fusion
```

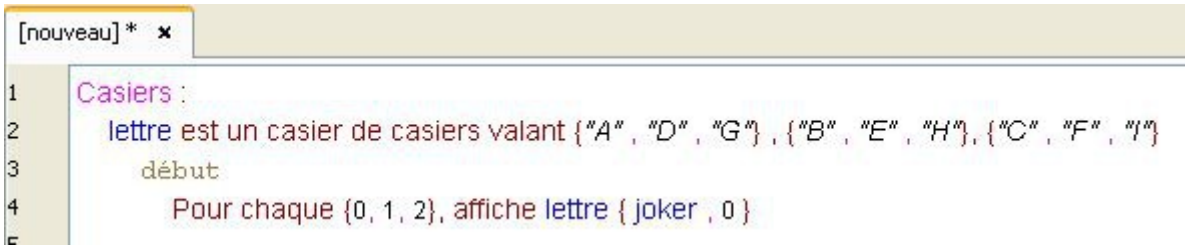
Ici, nous avons créé le casier de casiers **fusion**, sans avoir déclaré les casiers **langages** et **numéros** !

Voici ce que ça donne :


Comme vous pouvez le constater, on peut toujours utiliser chaque variable de chaque casier.

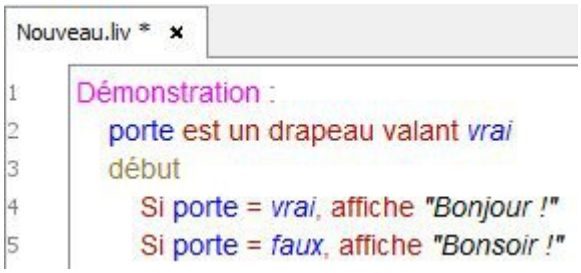
**Un petit exercice :**

Essayez de deviner ce qu'affiche ce programme :



# Les drapeaux

Voici encore un nouveau type de variables !
Rassurez-vous, les drapeaux sont très simple d'utilisation.
En effet, ils ne peuvent contenir que deux valeurs : **vrai** ou **faux**.




Même si leur intérêt semble limité, ils vous seront certainement bien utiles dans vos futurs programmes.

## Les variables particulières

Le Linotte met à votre disposition des variables que vous ne pouvez pas modifier. Vous pouvez seulement les lire. C'est l'interprète qui se charge de leur affecter une valeur : ce sont les **variables particulières**.

vrai et faux font parties de ces variables. En réalité :
- vrai est une variable de type nombre comprenant la valeur **1**.
- faux est une variable de type nombre comprenant la valeur **0**.

Dans le chapitre précédent, nous avons utiliser une autre variable particulière : le **joker**.

Les **variables particulières** sont donc des variables que nous n'avons pas besoin de déclarer et qui ont un rôle bien défini.
Au cours de ce tutoriel, nous allons en rencontrer d'autres.
La liste complète des variables particulières disponibles sera indiquée plus loin.

# Les espèces

Après les textes, les nombres, les casiers et les drapeaux, les espèces représentent le dernier type de variables existant en Linotte.

A l'instar d'autres langages de programmation, les espèces correspondent à de la **Programmation Orientée Objet**.
Derrière ce terme barbare se cache une fonctionnalité très intéressante :
les espèces vont nous permettre de créer nos propres types de variables !

Pour comprendre ce qu'est une espèce, voici un exemple :
Un chat est une espèce.
Les chats peuvent avoir les caractéristiques suivantes : une couleur, un âge, une taille...

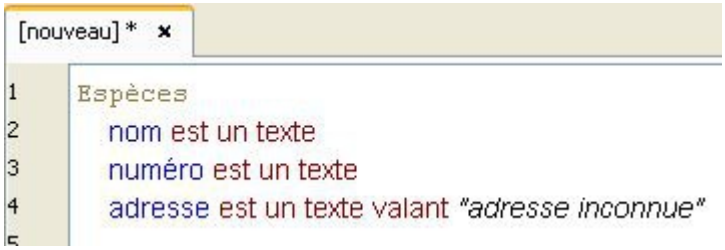
Chaque caractéristique (couleur, âge, taille) est une variable.
Chat est alors une espèce qui nous permet de regrouper toutes ces variables.

On peut ainsi créer des espèces contenant une infinité de variables.

Chaque variable d'une espèce est appelée **attribut**.

Mais prenons un exemple plus concret : nous allons créer un répertoire téléphonique.
Pour cela, nous allons créer une espèce **contact**.
Un **contact** sera constitué d'un **nom**, d'un **numéro** et d'une **adresse**.

Pour déclarer une espèce, commençons par créer une section Espèces : dans cette section, nous pourrons alors déclarer chaque attribut de notre espèce, comme ceci :




Déclarons ensuite notre espèce **contact** :


On notera la présence de virgules qui séparent chaque attribut de l'espèce, comme pour les casiers.

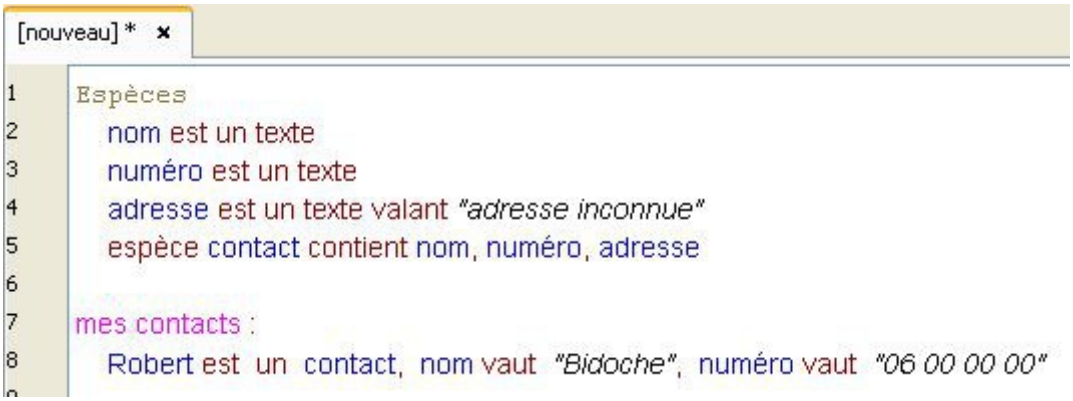
Ajoutons une fonction à notre livre :


Nous avons créé un nouveau type de variable : l'espèce contact !

On remarque que la section Espèces se trouve avant la fonction mes contacts.

En effet, la section **Espèces** doit se trouver avant la première fonction d'un livre. Cette organisation est obligatoire ! Si vous ne la respectez pas, votre programme ne fonctionnera pas.

Si on analyse la ligne 8, nous avons donc une variable de type contact qui a pour nom Robert.
Déterminons maintenant sa valeur :




On constate que la variable Robert n'a pas de valeur qui lui est propre.
En effet, on ne peut pas écrire :

```
Robert est un contact valant "Bidoche", "06 00 00 00"
```

Il faut alors affecter une valeur à chaque attribut de l'espèce, en utilisant le mot vaut.

Là encore, on notera la présence de virgules séparant chaque attribut.

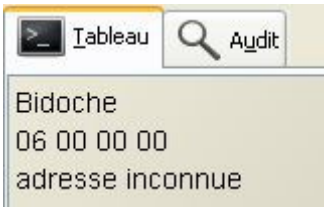
Néanmoins, il est possible de définir une valeur initiale pour chaque attribut d'une espèce, comme nous l'avons fait à la ligne 4 :

```
adresse est un texte valant "adresse inconnue"
```

Maintenant, si nous affichions ces valeurs ?

```
[nouveau] *
1  Espèces
2     nom est un texte
3     numéro est un texte
4     adresse est un texte valant "adresse inconnue"
5     espèce contact contient nom, numéro, adresse
6
7  mes contacts :
8     Robert est un contact, nom vaut "Bidoche", numéro vaut "06 00 00 00"
9     début
10       Affiche Robert
```

Et voici le résultat :




A la manière d'un casier, l'interprète affiche alors tous les attributs de la variable Robert.

Mais il est également possible de récupérer la valeur de chaque attribut séparément :

```
[nouveau] *
1  Espèces
2     nom est un texte
3     numéro est un texte
4     adresse est un texte valant "adresse inconnue"
5     espèce contact contient nom, numéro, adresse
6
7  mes contacts :
8     Robert est un contact, nom vaut "Bidoche", numéro vaut "06 00 00 00"
9     début
10       adresse de Robert vaut "45, avenue de la République"
11       Affiche nom de Robert
12       Affiche numéro de Robert
13       Affiche adresse de Robert
```

On peut bien sûr modifier chaque attribut à notre gré.

Et voici le résultat :


**Une astuce :**

Pour pouvoir utiliser un attribut dans une condition, il faut utiliser les parenthèses, comme ceci :

Nouveau.liv *

```
1  Espèces
2     nom est un texte
3     numéro est un texte
4     adresse est un texte valant "adresse inconnue"
5     espèce contact contient nom, numéro, adresse
6
7  mes contacts :
8     Robert est un contact, nom vaut "Bidoche", numéro vaut "06 00 00 00"
9     début
10       Si (adresse de Robert) = "adresse inconnue", adresse de Robert vaut "45, avenue de la République"
11       Affiche nom de Robert
12       Affiche numéro de Robert
13       Affiche adresse de Robert
```

**Une remarque :**

Si dans votre programme vous utilisez plusieurs espèces, ne créez qu'une seule section Espèces.
Comme ceci :

[nouveau] *

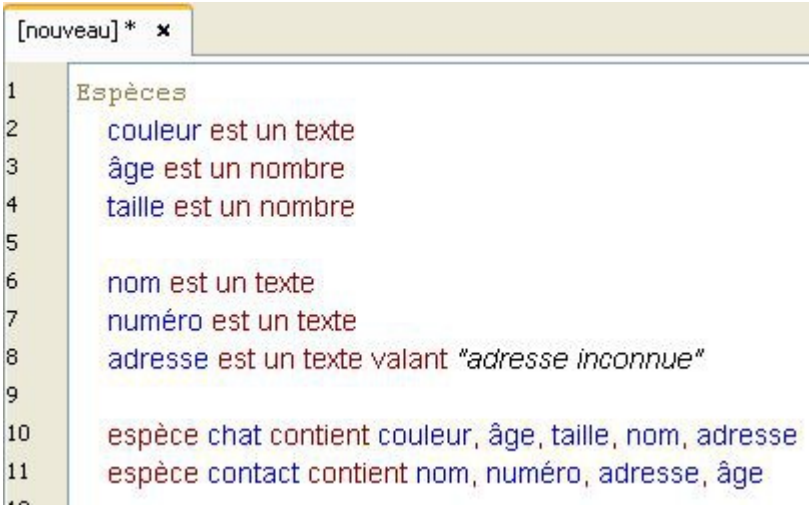
```
1  Espèces
2     couleur est un texte
3     âge est un nombre
4     taille est un nombre
5
6     nom est un texte
7     numéro est un texte
8     adresse est un texte valant "adresse inconnue"
9
10    espèce chat contient couleur, âge, taille
11    espèce contact contient nom, numéro, adresse
```

Attention : déclarez bien toutes les variables avant de créer vos espèces.

On constate qu'une espèce peut contenir des variables de types différents.

En effet, l'espèce chat contient alors une variable de type texte et deux variables de type nombre.

Vous pouvez même créer plusieurs espèces utilisant les mêmes variables :



# Les espèces graphiques

Nous allons maintenant étudier le graphisme.
Pour cela, nous allons utiliser la **toile**.
La **toile** est le support qui va nous permettre d'afficher du texte, des figures géométriques, etc... à l'écran.
Et tous ces objets (texte, figures géométriques, etc...) que l'on va afficher seront... Des espèces.
Mais rassurez-vous, vous n'avez pas besoin de les créer, elles sont déjà définies dans l'interprète.

Comme l'espèce chat que nous venons de voir, tous ces objets sont composés d'une multitude de variables.
En effet, pour afficher un objet sur la toile, nous avons besoin de trois caractéristiques essentielles :
- l'attribut **visible**, qui va déterminer si l'objet sera affiché ou non sur la toile.
- les coordonnées **x** et **y**, qui vont positionner l'objet sur la toile.

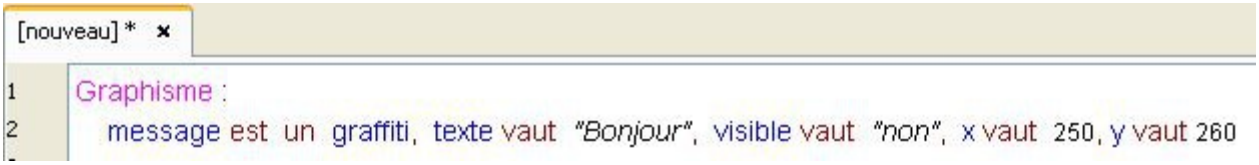
Toutes les espèces qui disposent alors de ces attributs sont des espèces graphiques.
Voici la première :

## Le graffiti

Le graffiti permet de projeter un texte sur la toile. Écrivez ce code :


Indiquons le texte à afficher :


Ajoutons enfin les trois caractéristiques essentielles de notre espèce graphique :




Maintenant que nous avons indiqué des coordonnées, nous pouvons afficher notre texte sur la toile.

Pour cela, nous allons utiliser le verbe **Projeter**.
On remarque que notre attribut visible à pour valeur **non ;** le verbe **Projeter** va alors lui affecter la valeur **oui**.

```
[nouveau] *
1  Graphisme :
2     message est un graffiti, texte vaut "Bonjour", visible vaut "non", x vaut 250, y vaut 260
3     début
4        Projette message
```

A la lecture de ce livre dans l'interprète, vous pouvez constater que "*Bonjour*" s'affiche au milieu de la toile.

Par défaut, l'attribut visible de chaque espèce graphique à pour valeur **non**. Vous n'êtes donc pas obliger de l'écrire :

```
[nouveau] *
1  Graphisme :
2     message est un graffiti, texte vaut "Bonjour", x vaut 250, y vaut 260
3     début
4        Projette message
```

Supprimez maintenant la ligne **2** de votre code et laisser le curseur sur cette ligne.
Aller dans le **verbier** de l'atelier, sélectionnez **Espèces**, puis cliquez sur **Graffiti**.
Vous devriez voir ceci apparaître sur le cahier :

```
[nouveau] *
1  Graphisme :
2  ? :: graffiti, x vaut ?, y vaut ?, texte vaut ?, couleur vaut ?, police vaut ?, taille vaut ?, transparence vaut ?, visible vaut ?, position vaut ?, angle vaut ?
3     début
4        Projette message
```

On ne peut pas bien le voir sur cette image mais vous pouvez alors visualiser sur le cahier tous les attributs de l'espèce graffiti.

Dans le verbier, le menu **Espèces** répertorie toutes les espèces disponibles en Linotte.

Vous pouvez alors indiquer une valeur à chaque attribut, modifiant ainsi son apparence sur la toile.

Modifions par exemple la couleur de notre graffiti :

```
[nouveau] *
1  Graphisme :
2     message :: graffiti, x vaut 250, y vaut 260, texte vaut "Bonjour", couleur vaut "jaune"
3     début
4        Projette message
```

On remarque que l'on a effacé tous les attributs qui n'étaient pas utiles.

Attention à bien indiquer les guillemets pour les attributs de type **texte**.

Mais que représente le symbole **::** ?

Vous l'aurez compris, **::** remplace les mots est un.

Le Linotte dispose ainsi de raccourcis pour certains termes, visant à augmenter la rapidité d'écriture de notre code.

La liste des raccourcis disponibles en Linotte sera indiquée plus loin dans le tutoriel.

Si vous exécutez ce livre, vous constatez alors que notre texte est devenu quasiment illisible.

Bien sûr, nous aurions pu choisir une autre couleur.

La liste de toutes les couleurs disponibles en Linotte se trouve dans le verbier.

Mais, si aucune couleurs ne vous convient, sachez que l'on peut également personnaliser notre couleur en indiquant les composantes **Rouge**, **Vert**, **Bleu** :

```
[nouveau] *
1  Graphisme :
2     message :: graffiti, x vaut 250, y vaut 260, texte vaut "Bonjour", couleur vaut "255 255 0"
3     début
4        Projette message
```

Même si l'on indique des chiffres à l'attribut **couleur**, cette variable étant de type **texte**, la présence des guillemets est quand même requise.

Pour découvrir les composantes des autres couleurs, vous pouvez visitez le lien suivant :

http://fr.wikipedia.org/wiki/Liste_de_couleurs

Le Linotte ne reconnaît que les composantes **RVB**.

Notre texte étant devenu quasiment illisible, corrigeons alors notre problème.

Pour cela, changeons la couleur de la toile :

```
[nouveau] *
1  Graphisme :
2     message :: graffiti, x vaut 250, y vaut 260, texte vaut "Bonjour", couleur vaut "255 255 0"
3     fond est une toile, couleur vaut "bleu"
4     début
5        Projette message
```

Et oui : même la toile est une espèce graphique !

Nous pouvons ainsi modifier sa couleur, mais également ses dimensions, sa bordure, etc...

**Une petite astuce :**

Vous pouvez personnaliser un peu plus votre graffiti en modifiant par exemple sa police.
Mais le choix des polices va dépendre de votre système.
Pour connaître alors toutes les polices disponibles, vous pouvez faire ceci :

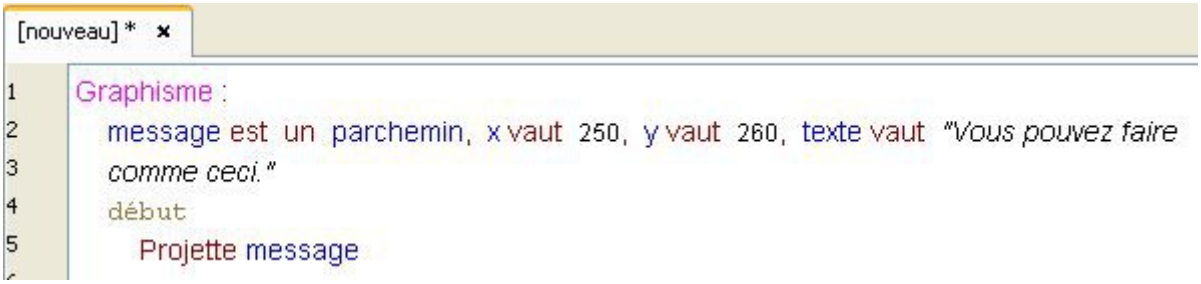

polices est une **variable particulière** du Linotte : vous ne pouvez pas la modifier.

Elle est de type casier et contient toutes les polices que vous pouvez utiliser.

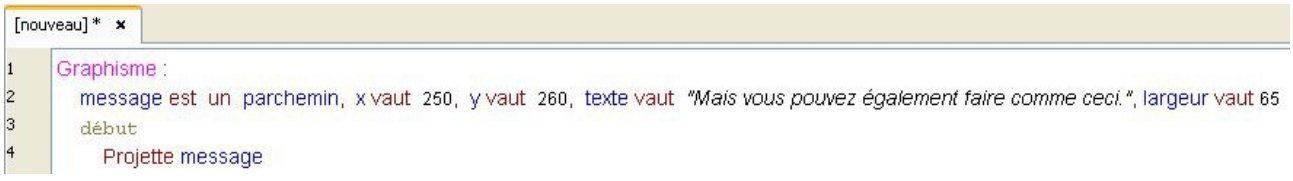
De la même façon, la **variable particulière** couleurs existe également.

# Le parchemin

Le parchemin est une espèce graphique comparable au graffiti, à la différence qu'il permet d'écrire sur plusieurs lignes.
Ainsi, vous pouvez utiliser le retour à la ligne lorsque vous déclarez votre message :




Mais vous pouvez également utiliser l'attribut largeur pour mettre en forme votre message :

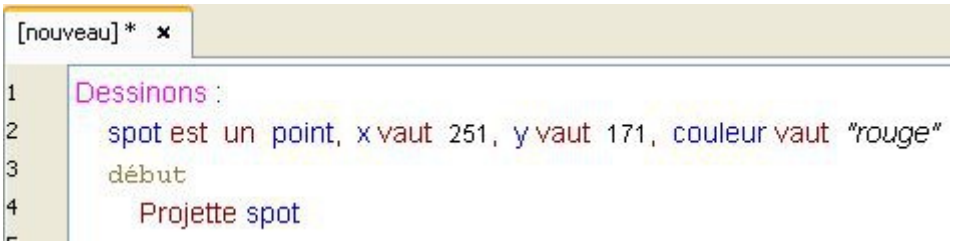



# Les figures géométriques

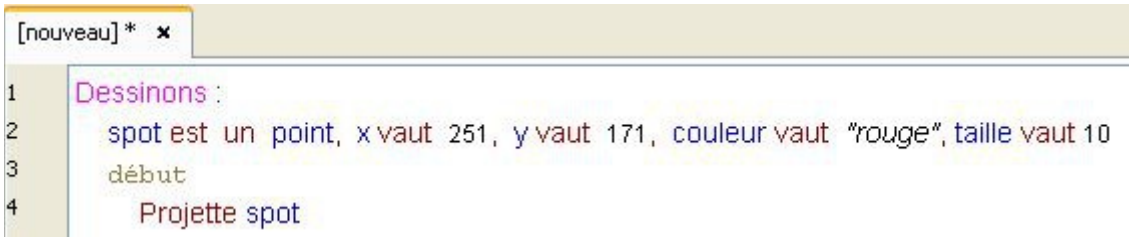
Voici la liste des figures géométriques disponibles en Linotte :
- Le point

- Le cercle
- Le rectangle
- La ligne

Dessinons un point :



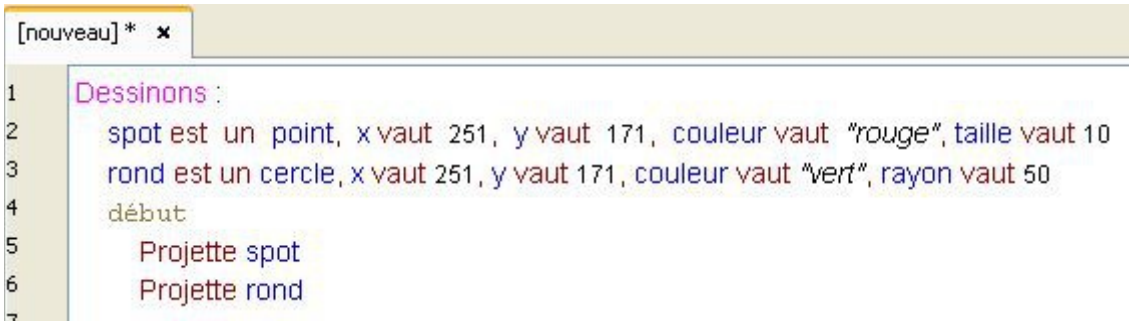

Le point paraît bien petit : agrandissons-le :




**Petite astuce :**

Si vous double-cliquez sur la toile, les coordonnées vont s'ajouter dans le cahier.

Maintenant, dessinons un cercle :




Le principe reste le même que le point, il faut juste définir le rayon en plus.

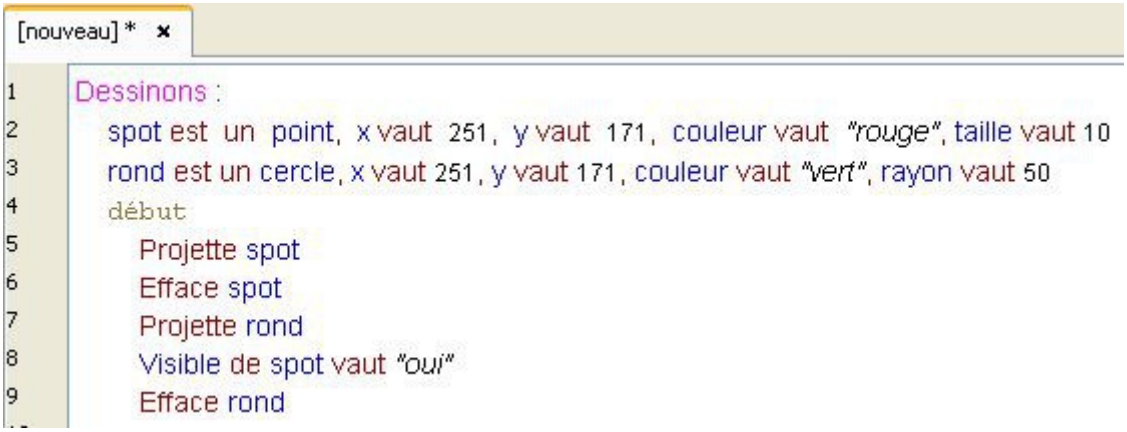
Pour le rectangle, le rayon est remplacé par une hauteur et une largeur.

Utilisé avec le cercle, le rectangle ou la ligne, l'attribut taille permet alors de modifier l'épaisseur du trait.

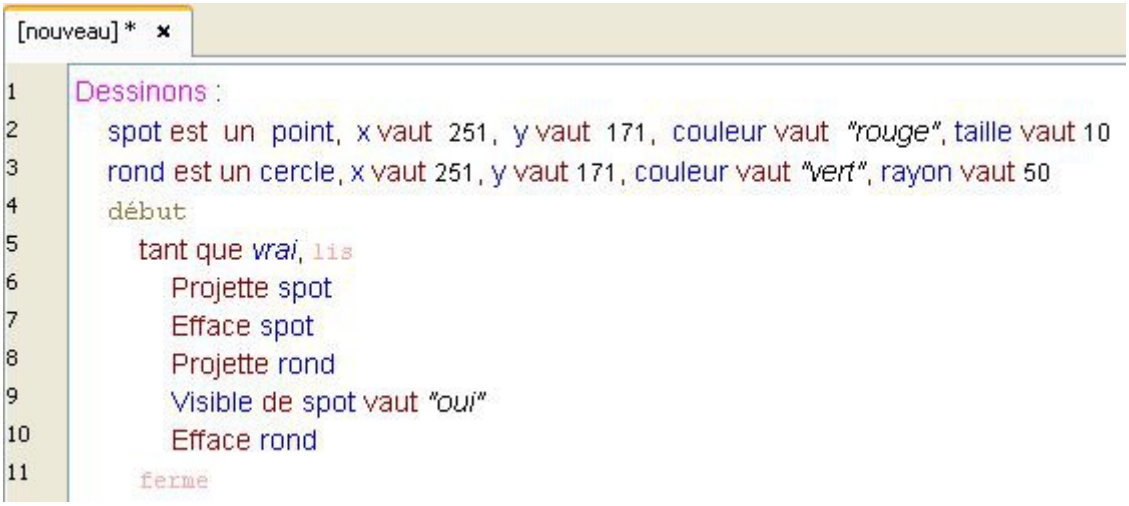
Dans le prochain exemple, nous allons utiliser le verbe **Effacer** pour alterner l'affichage entre le point et le cercle.

Le verbe **Effacer** va affecter la valeur **non** à l'attribut visible.

Pour réafficher notre objet, nous avons deux possibilités : soit utiliser le verbe **Projeter** ou alors remettre l'attribut visible à **oui** :




Si vous exécutez ce livre, vous n'allez probablement voir que le spot.
Rassurez-vous, c'est normal : l'exécution du livre est trop rapide.
Pour vous rendre compte de la vitesse d'exécution de votre code, nous allons réaliser une boucle infinie :




Ça va très vite n'est-ce pas ?

```
tant que vrai, lis
ferme
```

L'utilisation de cette condition permet de créer très simplement une boucle infinie. A utiliser avec prudence !

Heureusement, nous pouvons ralentir l'exécution de notre code.
Pour cela, utilisons le verbe **Attendre** :

```
[nouveau] *
1  Dessinons :
2     spot est un point, x vaut 251, y vaut 171, couleur vaut "rouge", taille vaut 10
3     rond est un cercle, x vaut 251, y vaut 171, couleur vaut "vert", rayon vaut 50
4     début
5        tant que vrai, lis
6           Projette spot
7           Attends 1 seconde
8           Efface spot
9           Projette rond
10          Attends 1 seconde
11          Visible de spot vaut "oui"
12          Attends 1 seconde
13          Efface rond
14       ferme
```

Le verbe **Attendre** peut s'utiliser avec des secondes ou des millisecondes.

Le cercle et le rectangle ont un attribut supplémentaire : plein. Si sa valeur est à **oui**, alors l'interprète va remplir la forme de sa couleur :

```
[nouveau] *
1  Dessinons :
2     spot est un point, x vaut 251, y vaut 171, couleur vaut "rouge", taille vaut 10
3     rond est un cercle, x vaut 251, y vaut 171, couleur vaut "vert", rayon vaut 50, plein vaut "oui"
4     début
5        tant que vrai, lis
6           Projette spot
7           Attends 1 seconde
8           Efface spot
9           Projette rond
10          Attends 1 seconde
11          Visible de spot vaut "oui"
12          Attends 1 seconde
13          Efface rond
14       ferme
```

Toutes les figures géométriques, ainsi que d'autres espèces graphiques, possèdent également un attribut permettant d'insérer une image dans une espèce graphique : l'attribut texture.
Pour cela, il suffit d'indiquer le chemin d'accès de votre image.
Voici un exemple avec l'espèce point :

```
[nouveau] *
1  Dessinons :
2     spot est un point, x vaut 251, y vaut 171, taille vaut 200, texture vaut "C:\Documents and Settings\Votre ordinateur\Mes documents\Herbe.jpg"
3     début
4        Projette spot
```

Et voici le résultat sur la toile :

Attention : les textures ne prennent pas en compte la rotation.

Ainsi, si vous modifiez l'attribut angle de votre espèce graphique, votre texture ne s'inclinera pas.

Enfin, il nous reste à voir la ligne. Celle-ci à une particularité : elle possède deux valeurs **x** et deux valeurs **y**. Chacune de ces valeurs correspondent à une extrémité :

```
[nouveau] *
1  Dessinons :
2     trait est une ligne, x1 vaut 50, y1 vaut 40, x2 vaut 350, y2 vaut 340
3     début
4        projette trait
```

# Le graphique et le patron

Le graphique vous permet d'afficher une image sur la toile.
Pour cela, il suffit de lui indiquer le chemin d'accès de votre image :

```
[nouveau] *
1  Image :
2     photo est un graphique, x vaut 250, y vaut 245, image vaut "C:\Documents and Settings\Votre ordinateur\Mes documents\Mer01.png"
3     début
4        Projette photo
```

L'attribut taille vous permet alors de zoomer ou dézoomer votre image.

Néanmoins, si vous voulez effectuer ce genre d'opération, préférez l'espèce patron.
En effet, le patron permet de charger des images au format SVG. Ce format vous permet d'agrandir vos images à l'infini.

```
[nouveau] *
1  Image :
2     photo est un  patron, x vaut 50, y vaut 40, modèle vaut "C:\Documents and Settings\Votre ordinateur\Plage01.svg",  hauteur vaut  500,  largeur vaut  500
3     début
4        Projette photo
```

Indiquez le chemin d'accès à votre image à l'attribut modèle.
Le zoom s'effectue avec les attributs hauteur et largeur.

**Une remarque :**

Les coordonnées **x** et **y** correspondent au coin haut gauche de votre image.
Ceci est valable pour toutes les espèces graphiques comme le rectangle ou le graffiti par exemple.

**Une petite astuce :**

Si votre image se trouve dans le même dossier que votre programme, vous n'avez pas besoin d'indiquer le chemin d'accès en entier. Vous pouvez vous contenter d'indiquer le nom de l'image :

```
Nouveau.liv *
1  Image :
2     photo est un graphique, x vaut 250, y vaut 245, image vaut "Mer01.png"
3     début
4        Projette photo
```

Ou si votre image se trouve dans un dossier :

```
Nouveau.liv *
1  Image :
2     photo est un graphique, x vaut 250, y vaut 245, image vaut "Mon dossier\Mer01.png"
3     début
4        Projette photo
```

Cette astuce est également valable pour l'attribut texture de certaines espèces graphiques.

# Le praxinoscope

Le praxinoscope est une espèce graphique qui permet de superposer rapidement plusieurs images sur la toile, afin de donner l'illusion du mouvement.
Pour pouvoir vous montrer un exemple, nous allons utiliser une série d'images fournies avec l'atelier Linotte.

Pour cela, rendez-vous dans le répertoire d'installation de l'atelier, puis dans le dossier **exemples > tutoriels > c_multimedia > images**.

Vous trouverez alors une série de douze images intitulées de **praxinoscope-0** à **praxinoscope-11**.

Sur votre cahier, recopiez alors ce code, en remplaçant les chemins d'accès aux images si cela est nécessaire :

[nouveau] *

```
Animation :
    écran est un praxinoscope, x vaut 200, y vaut 190, ...
        image0 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-0.png", ...
        image1 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-1.png", ...
        image2 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-2.png", ...
        image3 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-3.png", ...
        image4 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-4.png", ...
        image5 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-5.png", ...
        image6 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-6.png", ...
        image7 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-7.png", ...
        image8 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-8.png", ...
        image9 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-9.png", ...
        image10 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-10.png", ...
        image11 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-11.png"
    fond est une toile, couleur vaut "noir"
    début
        Projette fond
        Projette écran
```

L'utilisation des **...** permet de découper une ligne de code, afin de pouvoir l'écrire sur plusieurs lignes.

Si vous exécutez ce code, vous constaterez alors que image0 s'affiche sur la toile.

Afin de pouvoir faire défiler les images nous allons alors utiliser l'attribut trame du praxinoscope.
En effet, la valeur de l'attribut trame correspond au numéro de l'image que l'on veut afficher.
Ainsi, si vous écrivez ceci :

[nouveau] *

```
Animation :
    écran est un praxinoscope, x vaut 200, y vaut 190, ...
        image0 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-0.png", ...
        image1 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-1.png", ...
        image2 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-2.png", ...
        image3 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-3.png", ...
        image4 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-4.png", ...
        image5 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-5.png", ...
        image6 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-6.png", ...
        image7 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-7.png", ...
        image8 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-8.png", ...
        image9 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-9.png", ...
        image10 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-10.png", ...
        image11 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-11.png"
    fond est une toile, couleur vaut "noir"
    début
        Projette fond
        Projette écran
        trame de écran vaut 5
```

Vous constaterez que c'est la variable image5 qui s'affiche sur la toile.
C'est pourquoi, la valeur de l'attribut trame étant par défaut à **0**, c'était la variable image0 qui s'affichait à l'écran.
Pour pouvoir afficher les autres images, nous allons alors utiliser une boucle :

```
[nouveau] *
1  Animation :
2     écran est un praxinoscope, x vaut 200, y vaut 190, ...
3        image0 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-0.png", ...
4        image1 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-1.png", ...
5        image2 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-2.png", ...
6        image3 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-3.png", ...
7        image4 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-4.png", ...
8        image5 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-5.png", ...
9        image6 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-6.png", ...
10       image7 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-7.png", ...
11       image8 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-8.png", ...
12       image9 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-9.png", ...
13       image10 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-10.png", ...
14       image11 vaut "C:\Program Files\Linotte\exemples\tutoriels\c_multimedia\images\praxinoscope-11.png"
15    fond est une toile, couleur vaut "noir"
16    début
17       Projette fond
18       Projette écran
19
20       Pour chaque 12, lis   // Pour chacune des 12 images...
21          trame de écran vaut joker   // ...On affiche l'image correspondante au numéro
22          Attends 0.1 seconde
23       ferme
```

Grâce au praxinoscope, vous pouvez ainsi faire défiler jusqu'à 50 images.

# Le scribe

Le scribe est une espèce graphique interactive : il est capable d'intercepter les touches tapées sur le clavier et de les afficher aussitôt sur la toile.

```
[nouveau] *
1  Démonstration :
2     lettres est un scribe, x vaut 196, y vaut 253, couleur vaut "noir"
3     début
4        Projette lettres
```

Si vous exécutez ce livre, vous allez remarquer que rien ne se passe.
C'est normal. Le scribe est bien affiché mais il ne réagit pas.
Notre scribe dort, pour le réveiller il faut utiliser le verbe **Stimuler** :

```
[nouveau] *
1  Démonstration :
2      lettres est un scribe, x vaut 196, y vaut 253, couleur vaut "noir"
3      début
4          Projette lettres
5          Stimule lettres
```

Là encore, rien ne se passe !
En fait, si. Mais une fois de plus, l'exécution du livre est trop rapide : on a tout simplement pas le temps d'appuyer sur une touche du clavier que le programme est déjà terminé.

On pourrait alors utiliser le verbe **Attendre** pour retarder l'exécution du livre ?

Ça fonctionnerait. Mais sa nous permettrais d'utiliser notre programme que pour un temps défini. Comme par exemple :

```
[nouveau] *
1  Démonstration :
2      lettres est un scribe, x vaut 196, y vaut 253, couleur vaut "noir"
3      début
4          Projette lettres
5          Stimule lettres
6          Attends 5 secondes
```

C'est pourquoi nous allons plutôt utiliser un autre verbe : **Temporiser**.
Ce dernier permet de bloquer l'exécution d'un programme, en attendant qu'un événement se produise :

```
[nouveau] *
1  Démonstration :
2      lettres est un scribe, x vaut 196, y vaut 253, couleur vaut "noir"
3      début
4          Projette lettres
5          Stimule lettres
6          Temporise
```

Et voilà ! Maintenant, si vous tapez sur votre clavier, les lettres vont s'afficher sur la toile.

Mais notre programme ne s'arrête pas ?!

Et oui, le verbe **Temporiser** bloque notre livre jusqu'à ce qu'une touche soit enfoncée. Ainsi, le scribe, qui est déjà projeté et stimulé, intercepte la touche... Avant de se remettre en pause.

Pour le rendormir, il faut alors appuyer sur la touche **[entrée]** du clavier : ainsi, il n'intercepte plus les touches, le verbe **Temporiser** se débloque et notre programme peut continuer.

Dans notre exemple, cela termine donc notre livre.

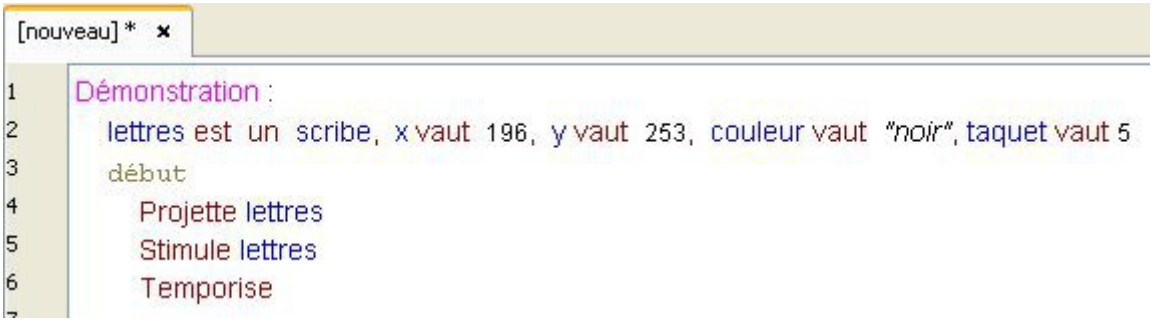
**Une astuce :**

Un clic droit sur la toile a le même effet que la touche **[entrée]**.

A noter que le verbe **Temporiser** peut également s'utiliser pour un temps défini :


Il est possible d'afficher plusieurs scribes sur la toile, mais un seul peut être actif. En utilisant le verbe **Stimuler**, celui-ci endort tous les scribes et réveille celui que vous souhaitez.

Le scribe limite la saisie d'un texte à 30 caractères. Pour changer cette limite, modifiez l'attribut taquet :




Le scribe a les même caractéristiques que le graffiti : vous pouvez modifier par exemple, sa couleur, sa police, son angle, sa position,…

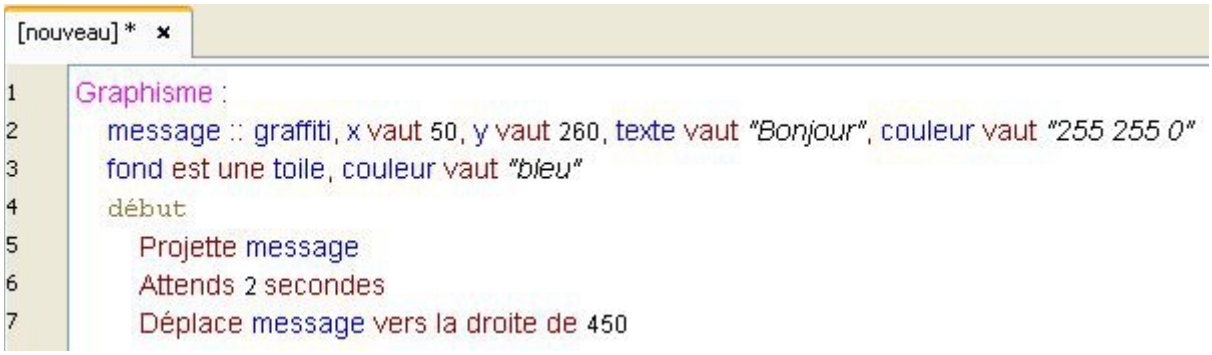
En fait, à quoi sert l'attribut **position** ?

Cette caractéristique détermine l'ordre d'affichage sur la toile.
Ainsi, si vous avez plusieurs images affichées sur la toile aux mêmes coordonnées, c'est l'image qui aura la plus haute position qui s'affichera par dessus les autres.

# Le verbe Déplacer

Ce verbe va nous permettre de déplacer nos espèces graphiques.

Reprenons un ancien exemple en y ajoutant le verbe **Déplacer** :



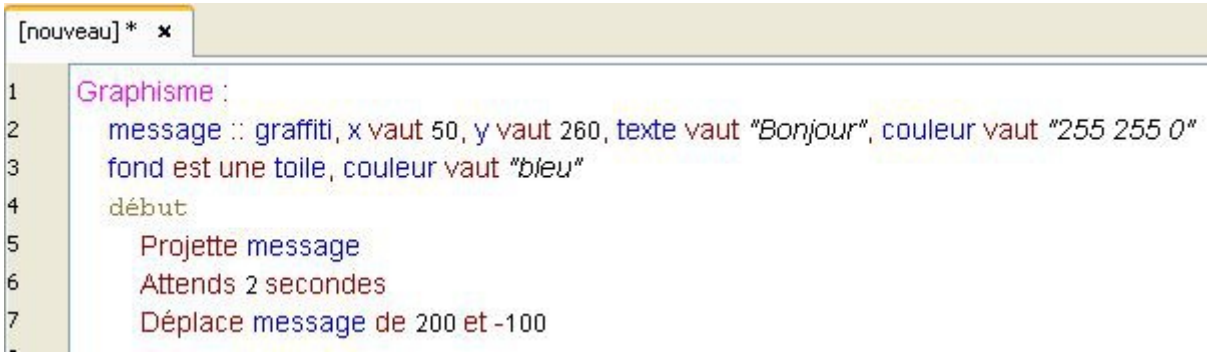

Comme vous pouvez le constater, notre graffiti se déplace alors de **450** pixels dans la direction choisie.

Les trois autres directions possibles sont les suivantes :

```
Déplace message vers la gauche de x
Déplace message vers le haut de x
Déplace message vers le bas de x
```

La valeur de **X** peut être positive ou négative.

Voici une forme plus évoluée qui permet de déplacer notre graffiti sur l'axe horizontal et sur l'axe vertical :




Cet exemple déplace notre graffiti de **200** pixels vers la droite et de **100** pixels vers le haut, le premier chiffre correspondant à l'axe horizontal et le second à l'axe vertical.

Voici le dernière forme du verbe **Déplacer**, qui permet de positionner notre graffiti vers une position précise :

```
[nouveau] *
1  Graphisme :
2     message :: graffiti, x vaut 50, y vaut 260, texte vaut "Bonjour", couleur vaut "255 255 0"
3     fond est une toile, couleur vaut "bleu"
4     début
5        Projette message
6        Attends 2 secondes
7        Déplace message vers 250 et 500
```

Cet exemple déplace notre graffiti aux coordonnées **x vaut 250**, **y vaut 500**.

# Les mégalithes

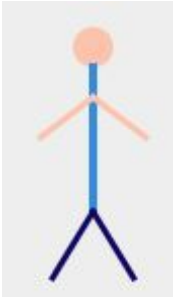
Un mégalithe permet de regrouper plusieurs espèces graphiques dans un même ensemble.
Prenons cet exemple :

```
[nouveau] *
1  Graphisme :
2     tête est un cercle , x vaut 440, y vaut 178, rayon vaut 5, taille vaut 10, couleur vaut "chair"
3     corps est une ligne, x1 vaut 440, y1 vaut 188, x2 vaut 440, y2 vaut 260, taille vaut 4, couleur vaut "bleu france"
4     bras 1 est une ligne, x1 vaut 440, y1 vaut 203, x2 vaut 414, y2 vaut 223, taille vaut 3, couleur vaut "chair"
5     bras 2 est une ligne, x1 vaut 440, y1 vaut 203, x2 vaut 466, y2 vaut 223, taille vaut 3, couleur vaut "chair"
6     jambe 1 est une ligne, x1 vaut 420, y1 vaut 293, x2 vaut 440, y2 vaut 260, taille vaut 3, couleur vaut "bleu nuit"
7     jambe 2 est une ligne, x1 vaut 460, y1 vaut 293, x2 vaut 440, y2 vaut 260, taille vaut 3, couleur vaut "bleu nuit"
8     début
9        Projette tête & corps & bras 1 & bras 2 & jambe 1 & jambe 2
```

A l'inverse des **…** , l'esperluette **&** permet de contracter le livre, en regroupant plusieurs actions identiques sur une même ligne.
Ce qui nous évite de devoir écrire :

```
Projette tête
Projette corps
Projette bras 1
Projette bras 2
Projette jambe 1
Projette jambe 2
```

Voici donc ce qui s'affiche sur la toile :



Imaginons que nous voulons déplacer notre personnage.
Il faudrait alors déplacer chaque élément qui le constitue.
C'est là qu'intervient le mégalithe !
Commençons par en créer un :

[nouveau] *

```
Graphisme :
   tête est un cercle , x vaut 440, y vaut 178, rayon vaut 5, taille vaut 10, couleur vaut "chair"
   corps est une ligne, x1 vaut 440, y1 vaut 188, x2 vaut 440, y2 vaut 260, taille vaut 4, couleur vaut "bleu france"
   bras 1 est une ligne, x1 vaut 440, y1 vaut 203, x2 vaut 414, y2 vaut 223, taille vaut 3, couleur vaut "chair"
   bras 2 est une ligne, x1 vaut 440, y1 vaut 203, x2 vaut 466, y2 vaut 223, taille vaut 3, couleur vaut "chair"
   jambe 1 est une ligne, x1 vaut 420, y1 vaut 293, x2 vaut 440, y2 vaut 260, taille vaut 3, couleur vaut "bleu nuit"
   jambe 2 est une ligne, x1 vaut 460, y1 vaut 293, x2 vaut 440, y2 vaut 260, taille vaut 3, couleur vaut "bleu nuit"
   personnage est un mégalithe
   début
      Projette tête & corps & bras 1 & bras 2 & jambe 1 & jambe 2
```

Il nous reste plus qu'à insérer chaque élément dans notre mégalithe.
Pour cela, nous utilisons le verbe **Fusionner** :

[nouveau] *

```
Graphisme :
   tête est un cercle , x vaut 440, y vaut 178, rayon vaut 5, taille vaut 10, couleur vaut "chair"
   corps est une ligne, x1 vaut 440, y1 vaut 188, x2 vaut 440, y2 vaut 260, taille vaut 4, couleur vaut "bleu france"
   bras 1 est une ligne, x1 vaut 440, y1 vaut 203, x2 vaut 414, y2 vaut 223, taille vaut 3, couleur vaut "chair"
   bras 2 est une ligne, x1 vaut 440, y1 vaut 203, x2 vaut 466, y2 vaut 223, taille vaut 3, couleur vaut "chair"
   jambe 1 est une ligne, x1 vaut 420, y1 vaut 293, x2 vaut 440, y2 vaut 260, taille vaut 3, couleur vaut "bleu nuit"
   jambe 2 est une ligne, x1 vaut 460, y1 vaut 293, x2 vaut 440, y2 vaut 260, taille vaut 3, couleur vaut "bleu nuit"
   personnage est un mégalithe
   début
      Fusionne tête & corps & bras 1 & bras 2 & jambe 1 & jambe 2 dans personnage
      Projette personnage
      Attends 2 secondes
      Déplace personnage de -100 et 0
```

Là encore, l'utilisation de l'esperluette **&** est bien pratique.

Et voilà le travail !

A noter que l'on peut indiquer des coordonnées initiales à notre mégalithe :

```
[nouveau] *
1  Graphisme :
2     tête est un cercle , x vaut 440, y vaut 178, rayon vaut 5, taille vaut 10, couleur vaut "chair"
3     corps est une ligne, x1 vaut 440, y1 vaut 188, x2 vaut 440, y2 vaut 260, taille vaut 4, couleur vaut "bleu france"
4     bras 1 est une ligne, x1 vaut 440, y1 vaut 203, x2 vaut 414, y2 vaut 223, taille vaut 3, couleur vaut "chair"
5     bras 2 est une ligne, x1 vaut 440, y1 vaut 203, x2 vaut 466, y2 vaut 223, taille vaut 3, couleur vaut "chair"
6     jambe 1 est une ligne, x1 vaut 420, y1 vaut 293, x2 vaut 440, y2 vaut 260, taille vaut 3, couleur vaut "bleu nuit"
7     jambe 2 est une ligne, x1 vaut 460, y1 vaut 293, x2 vaut 440, y2 vaut 260, taille vaut 3, couleur vaut "bleu nuit"
8     personnage est un mégalithe, x vaut 100, y vaut 150
9     début
10       Fusionne tête & corps & bras 1 & bras 2 & jambe 1 & jambe 2 dans personnage
11       Projette personnage
12       Attends 2 secondes
13       Déplace personnage de -100 et 0
```

Tous les éléments qui le constitue sont alors automatiquement déplacés vers ces coordonnées.

# Collision

Il est possible de détecter si deux espèces graphiques sont en collision.
Pour cela, il faut utiliser la condition suivante :

Si... est en collision avec..., ... .

Cette condition va détecter si les deux objets ont des pixels en commun :

```
[nouveau] *
1  Collision :
2     spot est un point, x vaut 50, y vaut 70, couleur vaut "rouge", taille vaut 10, position vaut 1
3     rond est un cercle, x vaut 290, y vaut 280, couleur vaut "vert", rayon vaut 50, plein vaut "oui"
4     début
5        Projette spot
6        Projette rond
7        Attends 1 seconde
8        Déplace spot de  240 et 210
9        Si spot est en collision avec rond, couleur de spot vaut "bleu"
```

On remarque que l'attribut **position** de **spot** vaut **1**.

En effet, si on le laisse à 0, lors de son déplacement, le spot disparaîtra sous le rond.

Pour que la collision soit possible, l'attribut **plein** de **rond** doit être à **oui**, sinon la collision ne concernera que la bordure du cercle.

**Une remarque :**

Une image ou un graffiti sont détecté par le système de collision comme des rectangles.

Exemple :

Sur cette image, la collision concerne toute la surface du rectangle rouge.

# Interaction avec l'utilisateur

Nous allons maintenant intercepter les événements liés au clavier et à la souris.

Dans un précédent exemple, nous avons déjà utilisé l'espèce graphique scribe pour intercepter les touches tapées au clavier et les afficher sur la toile.

Ici, nous allons apprendre à nous servir de la variable touche.

touche une **variable particulière** du Linotte qui retourne la dernière touche utilisée. Voyez plutôt :


Contrairement au **scribe**, ce code n'affiche pas toutes les lettres sur une ligne. Il les affiche successivement, au même endroit.

La variable particulière touche permet ainsi de connaître le nom de chaque touche du clavier.

Concernant les événements liés à la souris, nous avons deux autres **variables particulières** à notre disposition : sourisx et sourisy.

Celles-ci vont nous permettre de connaître la position du pointeur à tout moment, comme ceci :


**Une astuce :**

Vous pouvez changer la forme du curseur en modifiant la caractéristique pointeur de la toile :

```
[nouveau] *
1  Interaction avec la souris :
2     infox est un graffiti, x vaut 130, y vaut 260
3     infoy est un graffiti, x vaut 130, y vaut 290
4     fond est une toile, couleur vaut "blanc"
5     début
6        pointeur de fond vaut "main"
7        Projette infox & infoy
8        Tant que vrai, lis
9           texte de infox vaut sourisx
10          texte de infoy vaut sourisy
11       ferme
```

Les autres valeurs possibles de l'attribut pointeur sont : **flèche**, **texte** ou **normal**.

# Le crayon

Le crayon est une espèce graphique qui va nous permettre de dessiner sur la toile.

Le crayon possède deux attributs importants :
- l'attribut pointe, qui affiche un pinceau sur la toile.
- l'attribut posé, qui pose le crayon sur la toile pour commencer à dessiner.

Ces attributs possèdent deux valeurs : **oui** et **non**.

Voici une démonstration du crayon utilisant les variables particulières sourisx et sourisy :

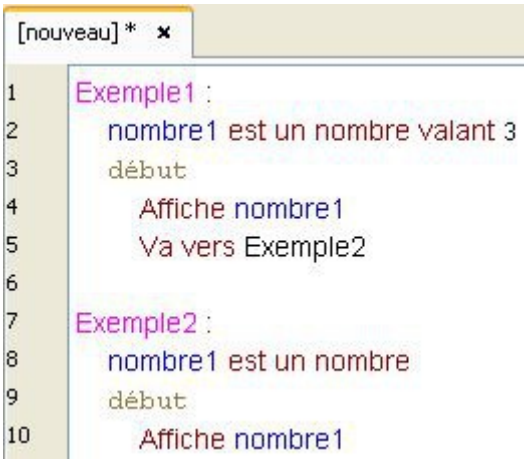
```
Nouveau.liv *
1  Dessinons :
2     fond est une toile, couleur vaut "blanc"
3     crayon est un crayon, couleur vaut "bleu", x vaut 100, y vaut 100, pointe vaut "non", posé vaut "non"
4     début
5        Projette fond & crayon
6        Tant que vrai, lis
7           Déplace crayon vers sourisx et sourisy   // Le crayon va alors suivre le pointeur de la souris
8           Si touche = "clique", lis
9              Si (posé de crayon) = "non", lis      // Si le crayon n'est pas posé sur la toile, on l'abaisse et on affiche le pinceau
10                posé de crayon vaut "oui"
11                pointe de crayon vaut "oui"
12             ferme
13             Sinon lis                             // Si le crayon est déjà posé sur la toile, on le relève et on efface le pinceau
14                posé de crayon vaut "non"
15                pointe de crayon vaut "non"
16             ferme
17          ferme
18       ferme
```

On remarque que la variable particulière **touche** peut également intercepter le clique de la souris.

# Les variables globales

Abordons maintenant les notions avancées du Linotte.

Pour comprendre ce qu'est une **variable globale**, prenons un exemple :




Dans notre cours sur les fonctions, nous avons établis que lorsque l'on quitte une fonction, la variable est détruite.

nombre1 est alors une **variable locale** : elle ne peut être utilisée que dans la fonction où elle est déclarée.

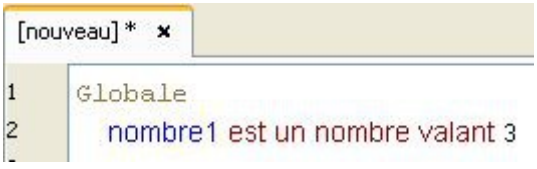
C'est pourquoi dans notre fonction Exemple2, nombre1 est égal à **0**, et non **3**. Car il s'agit en réalité d'une nouvelle variable.

En Linotte, il est ainsi possible de créer une infinité de variables ayant le même nom.

Mais imaginons que l'on veuille réutiliser nombre1 dans une notre fonction Exemple2, comment faire ?

Pour cela, la solution va consister à transformer nombre1 en une **variable globale**.

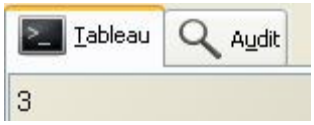
Commençons alors par créer une section Globale dans notre cahier : dans cette section nous allons ainsi déclarer notre variable :




Ajoutons ensuite notre première fonction :

```
[nouveau] *
1  Globale
2    nombre1 est un nombre valant 3
3
4  Exemple1 :
5    début
6      Affiche nombre1
```

Voici ce qu'affiche le tableau :




Exemple1 peut donc utiliser notre variable nombre1, sans avoir besoin de la déclarer.

Dans ce cas, allons plus loin :

```
[nouveau] *
1  Globale
2    nombre1 est un nombre valant 3
3
4  Exemple1 :
5    début
6      Affiche nombre1
7      nombre1 vaut 6
8      Va vers Exemple2
9
10 Exemple2 :
11   début
12     Affiche nombre1
13     nombre1 vaut 5
14     Affiche nombre1
15     Va vers Exemple1
```

Attention, ce programme tourne en boucle.

Mais il fonctionne ! nombre1 peut désormais être utilisé depuis n'importe quelle fonction du livre.

Observons les premiers résultats de notre exemple :


Au premier passage, le livre va afficher ceci :
**3** : valeur initiale de notre variable globale.
**6** : valeur affectée à notre variable dans Exemple1.
**5** : valeur affectée à notre variable dans Exemple2.

Puis le livre revient à la première fonction et fait donc un autre passage, qui donne ceci :
**5** : valeur affectée à notre variable dans Exemple2.
**6** : valeur affectée à notre variable dans Exemple1.
**5** : valeur affectée à notre variable dans Exemple2.

Que constate-t-on ?

Lorsque l'on quitte la fonction Exemple1 ou Exemple2, la variable n'est pas réinitialisée à **3**.

Étant une **variable globale**, celle-ci n'est pas détruite lorsque l'on quitte une fonction. Elle conserve alors sa dernière valeur connue.

On peut ainsi utiliser notre variable nombre1 dans toutes les fonctions de notre livre : sa valeur est sauvegardée.

Attention : la section **Globale** doit se trouver avant la première fonction d'un livre. Cette organisation est obligatoire ! Si vous ne la respectez pas, votre programme ne fonctionnera pas.

**Une petite remarque :**

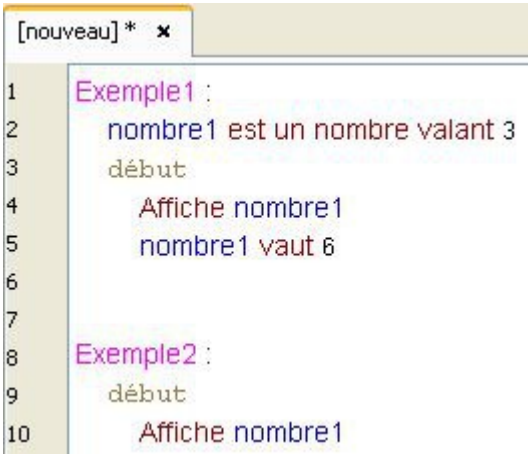
Vous n'êtes pas obligé d'affecter une valeur initiale à votre **variable globale** :


Ici, **nombre1** est donc égal à **0**.

# Les paramètres

Dans le chapitre précédent, nous avons appris que les variables étaient **locales** : on ne peut les utiliser que dans la fonction où elles sont déclarées.
Pourtant, il existe une autre façon de réutiliser une variable dans une autre fonction, sans utiliser les **variables globales**.
En utilisant les **paramètres.**

Prenons cet exemple :




Pour pouvoir utiliser notre variable nombre1 dans la fonction Exemple2, nous allons utiliser un **paramètre**.

En effet, un **paramètre** est une variable qui peut prendre la valeur d'une autre variable.

Il se déclare comme ceci :


Pour indiquer qu'une variable est un **paramètre**, nous la préfixons par un astérisque.

Il nous faut alors ordonner à l'interprète de se rendre dans la fonction Exemple2.

Pour cela, nous allons utiliser le verbe **Parcourir** :

```
[nouveau] *
1  Exemple1 :
2     nombre1 est un nombre valant 3
3     début
4        Affiche nombre1
5        nombre1 vaut 6
6        Parcours Exemple2
7
8  Exemple2 :
9     *nombre1 est un nombre
10    début
11       Affiche nombre1
```

Cet exemple ne fonctionne pas !

En effet, un **paramètre** prenant la valeur d'une autre variable, il faut alors indiquer la variable que nous allons transmettre à notre paramètre *nombre1 :

```
[nouveau] *
1  Exemple1 :
2     nombre1 est un nombre valant 3
3     début
4        Affiche nombre1
5        nombre1 vaut 6
6        Parcours Exemple2 avec nombre1
7
8  Exemple2 :
9     *nombre1 est un nombre
10    début
11       Affiche nombre1
```

En utilisant le mot avec, on détermine alors la variable à passer en paramètre.

Et voici le résultat :

```
Tableau  Audit
3
6
```

Ça fonctionne ! Nous pouvons alors désormais utiliser la variable nombre1 dans nos deux fonctions.

Mais faisons alors un petit test :

Modifions le nom de notre paramètre, comme ceci :


Cet exemple ne fonctionne pas !

En effet, voici le message d'erreur :

```
L'acteur est inconnu de la fonction et du livre : nombre1
```

Malgré l'utilisation du **paramètre**, on ne peut donc pas accéder à notre variable nombre1 dans la fonction Exemple2.

En effet, pour corriger notre exemple, il faut alors faire ceci :


Vous pouvez ainsi renommer votre paramètre comme bon vous semble.

Cet exemple fonctionne !

Mais alors, ce n'est pas véritablement la variable **nombre1** que l'on utilise dans notre fonction **Exemple2** ?

Effectivement, dans notre fonction Exemple2, nous utilisons le paramètre nombre2.

**nombre2** n'est alors qu'une copie de notre variable **nombre1** ?

Non. Car si nous modifions la valeur de nombre2, la valeur de notre variable nombre1 changera !

Pour le vérifier, nous allons utiliser un autre verbe.

Souvenez-vous, le verbe **Parcourir** s'utilise avec le verbe **Revenir**.

Complétons alors notre exemple comme ceci :


Voici le résultat :


Si on analyse ces résultats, voici ce que sa donne :

**3** : valeur initiale de nombre1.
**6** : valeur affectée à nombre1 dans Exemple1 et transmise au paramètre *nombre2.
**5** : valeur affectée à nombre2 dans Exemple2.

En modifiant la valeur du paramètre nombre2, nous avons alors modifié la valeur de notre variable nombre1 !

Un **paramètre** permet ainsi de rendre une variable accessible à d'autres fonctions.

À la ligne 7 on remarque que **nombre1** affiche **5**, et non **3**, sa valeur initiale.

Et oui, car les verbes **Parcourir** et **Revenir** permettent d'accéder à la fonction Exemple2 sans quitter notre fonction Exemple1.

La variable nombre1 n'est donc pas détruite et conserve ainsi sa dernière valeur connue.

Et ainsi nombre1 reste une **variable locale**.

Cela évite de devoir créer une **variable globale** qui deviendra alors accessible dans tout le livre.

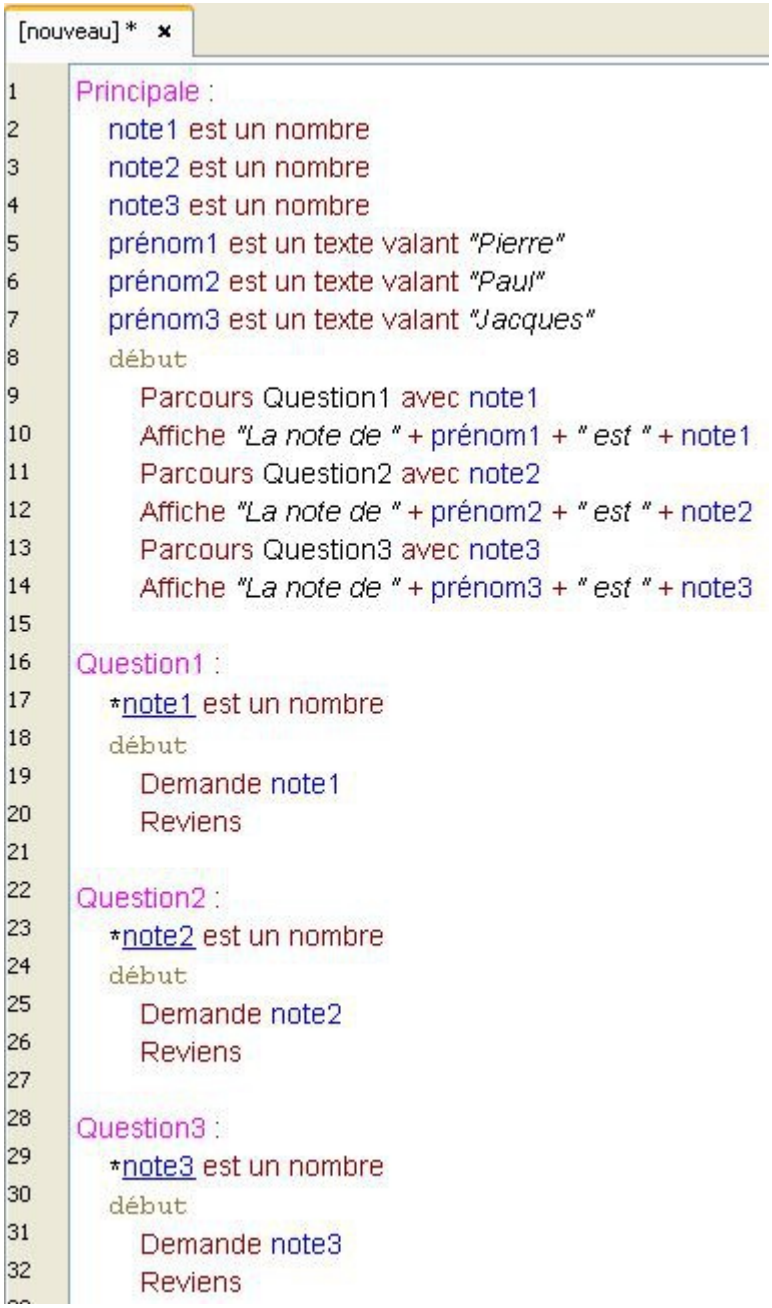
Mais justement, utiliser une **variable globale** paraît bien plus simple, non ?

Oui, mais l'utilisation de variables globales peut être une source d'erreurs.

En effet, lors de la recherche d'erreurs de programmation, lorsqu'une variable globale contient une information erronée, il est plus difficile de trouver la source de l'erreur, car la variable a pu être modifiée dans n'importe quelle partie du livre.
De plus, l'utilisation de variables globales rend la modification de notre programme difficile, car il faut alors comprendre le fonctionnement de tout le livre pour savoir comment une seule variable est utilisée, étant donné qu'elle peut être modifiée depuis n'importe quelle fonction.
En utilisant un **paramètre** ainsi que les verbes **Parcourir** et **Revenir**, on s'assure que la variable ne peut être utilisée que dans certaines fonctions précises, ce qui facilite la compréhension de notre livre.

Mais allons encore plus loin dans l'utilisation des **paramètres**.

Prenons cet exemple :

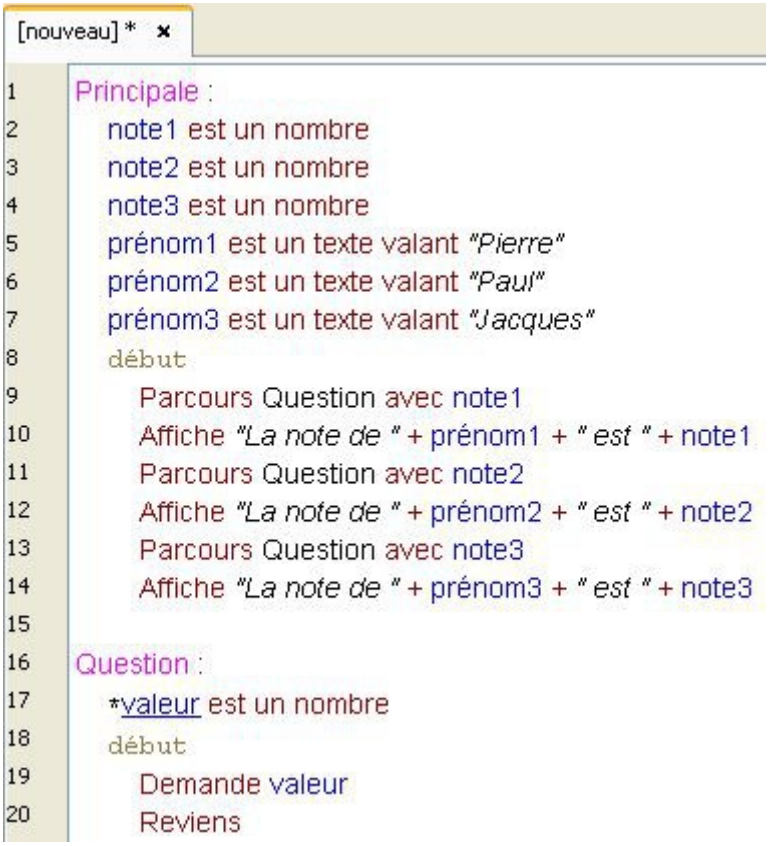



Dans cet exemple, on demande à l'utilisateur de donner une note à trois élèves. On affiche ensuite le résultat sur le tableau de l'atelier.

Et bien, sachez que l'on peut énormément simplifier ce code :

En effet, un seul **paramètre** peut prendre la valeur de plusieurs variables !

Pour cela, réécrivons notre code comme ceci :




Le paramètre *valeur prend alors successivement la valeur de note1, note2, puis note3 dans la fonction Question.

Pensez à utiliser des **paramètres** pour mieux organiser votre livre: vous allez ainsi réduire sa taille car tous les traitements identiques seront regroupés dans un même endroit.

D'ailleurs, on peut encore améliorer notre exemple :

```
[nouveau] *
1  Principale :
2     note1 est un nombre
3     note2 est un nombre
4     note3 est un nombre
5     prénom1 est un texte valant "Pierre"
6     prénom2 est un texte valant "Paul"
7     prénom3 est un texte valant "Jacques"
8     début
9        Parcours Question avec note1, prénom1
10       Parcours Question avec note2, prénom2
11       Parcours Question avec note3, prénom3
12
13 Question :
14    *valeur est un nombre
15    *prénom est un texte
16    début
17       Demande valeur
18       Affiche "La note de " + prénom + " est " + valeur
19       Reviens
```

Le verbe **Parcourir** peut utiliser plusieurs **paramètres**.

Pour cela, il suffit de séparer chaque variable d'une virgule et de déclarer un nombre de paramètres correspondants.

Attention : un paramètre ne peut remplacer qu'une variable ayant le même type.

Ainsi, respectez l'ordre d'énumération des variables.

Si vous écrivez ceci par exemple :

[nouveau] *

```
1  Principale :
2     note1 est un nombre
3     note2 est un nombre
4     note3 est un nombre
5     prénom1 est un texte valant "Pierre"
6     prénom2 est un texte valant "Paul"
7     prénom3 est un texte valant "Jacques"
8     début
9        Parcours Question avec note1, prénom1
10       Parcours Question avec note2, prénom2
11       Parcours Question avec note3, prénom3
12
13 Question :
14    *prénom est un texte
15    *valeur est un nombre
16    début
17       Demande valeur
18       Affiche "La note de " + prénom + " est " + valeur
19       Reviens
```

Ce code ne fonctionne pas.

En effet, la première variable, note1, est un nombre. Elle est donc incompatible avec le premier paramètre, *prénom, qui est un texte.

Il est également interdit d'affecter une valeur initiale à un paramètre. Il prend la valeur d'une autre variable, ainsi vous ne pouvez pas écrire :

```
*valeur est un nombre valant 10
```

Et enfin, sachez que l'on peut encore réduire la taille de notre code :

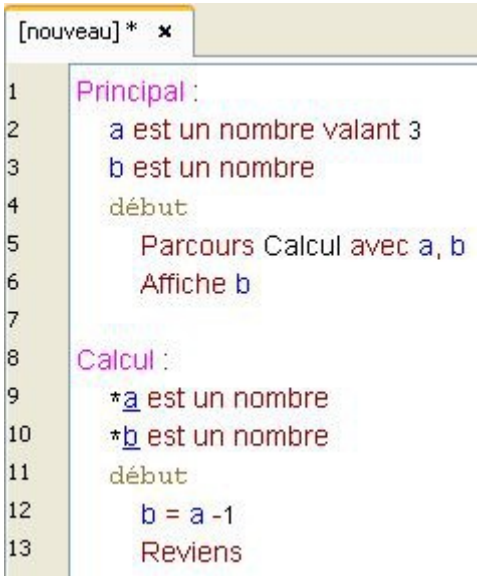
[nouveau] *

```
1   Principale :
2      note1 est un nombre
3      note2 est un nombre
4      note3 est un nombre
5      début
6         Parcours Question avec note1, "Pierre"
7         Parcours Question avec note2, "Paul"
8         Parcours Question avec note3, "Jacques"
9
10  Question :
11     *valeur est un nombre
12     *prénom est un texte
13     début
14        Demande valeur
15        Affiche "La note de " + prénom + " est " + valeur
16        Reviens
```

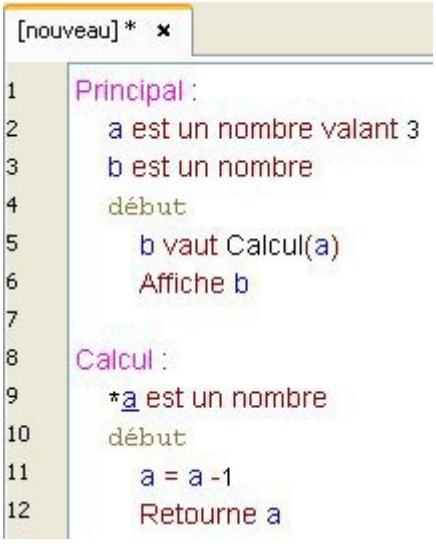
Le verbe **Parcourir** peut utiliser les **variables anonymes** !

# Le verbe Retourner

Le verbe **Retourner** permet à une fonction de renvoyer une valeur.
Prenons cet exemple :




Voici le même exemple en utilisant le verbe **Retourner** :




A la ligne 5, on appelle la fonction Calcul, en lui transmettant en paramètre le nombre a.
La fonction Calcul va alors soustraire a de **1** puis retourner le résultat dans la fonction Principal.

**Détaillons l'écriture de ce code :**

Le verbe **Retourner** fonctionne comme le verbe **Parcourir** :

Cette instruction :

```
Calcul(a)
```

Équivaut alors à celle-ci :


En utilisant le verbe **Retourner**, on remplace alors simplement le mot **avec** par des parenthèses.

Ensuite, dans la fonction appelée, on déclare le paramètre :

```
Calcul :
  *a est un nombre
  début
```

Et on termine la fonction par le verbe **Retourner**, suivit de la variable à renvoyer, ici notre paramètre :

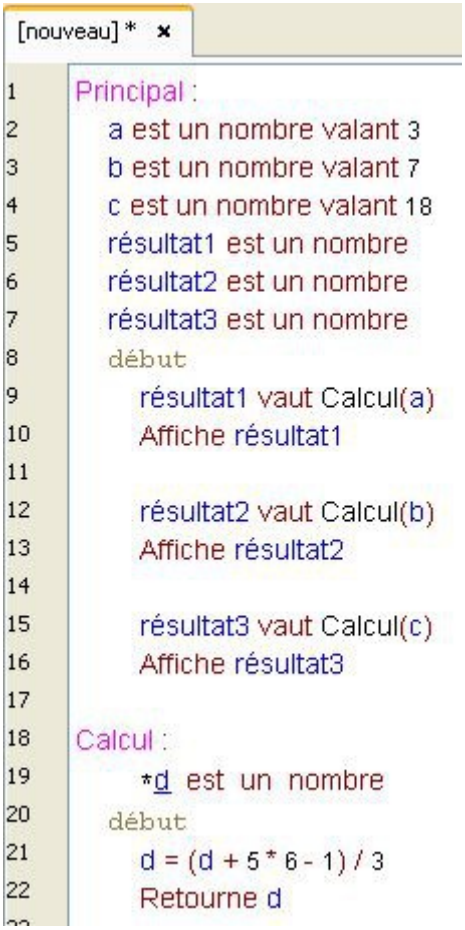
```
Calcul :
  *a est un nombre
  début
    a = a -1
    Retourne a
```

Ainsi, en écrivant simplement :

```
b vaut Calcul(a)
```

Notre variable b accède directement au résultat du calcul !

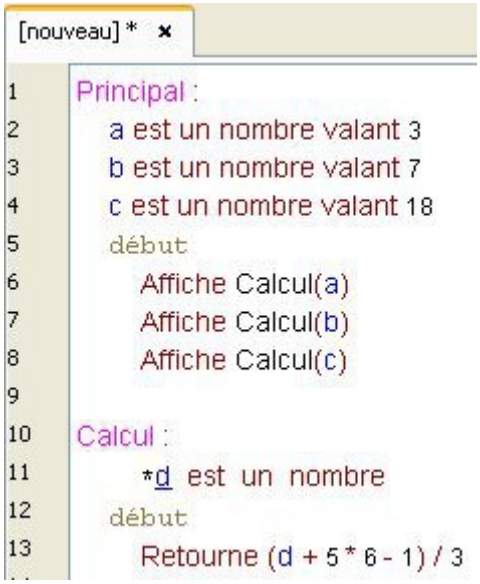
Et comme notre fonction Calcul emploie un paramètre, on peut l'utiliser avec différentes variables :



```
[nouveau] *
1  Principal :
2     a est un nombre valant 3
3     b est un nombre valant 7
4     c est un nombre valant 18
5     résultat1 est un nombre
6     résultat2 est un nombre
7     résultat3 est un nombre
8     début
9        résultat1 vaut Calcul(a)
10       Affiche résultat1
11
12       résultat2 vaut Calcul(b)
13       Affiche résultat2
14
15       résultat3 vaut Calcul(c)
16       Affiche résultat3
17
18 Calcul :
19       *d est un nombre
20    début
21       d = (d + 5 * 6 - 1) / 3
22       Retourne d
```

Comme pour le verbe **Parcourir**, inutile alors de réécrire notre fonction Calcul pour chaque variable a, b ou c : on peut se contenter de l'écrire une seule fois.

Mais l'intérêt du verbe **Retourner** par rapport au verbe **Parcourir** consiste à pouvoir appeler directement notre fonction Calcul en utilisant un verbe.
Par exemple, avec le verbe **Afficher** :



```
[nouveau] *
1  Principal :
2     a est un nombre valant 3
3     b est un nombre valant 7
4     c est un nombre valant 18
5     début
6        Affiche Calcul(a)
7        Affiche Calcul(b)
8        Affiche Calcul(c)
9
10 Calcul :
11       *d est un nombre
12    début
13       Retourne (d + 5 * 6 - 1) / 3
```

Le verbe **Retourner** peut également utiliser les **variables anonymes** et les opérations mathématiques.

Une fonction retournant un résultat peut ainsi être appelée par de nombreux verbes : **Afficher**, **Ajouter**, **Projeter**, …

**Une astuce :**

On peut également utiliser une fonction retournant un résultat, directement au sein d'une condition. Voici un exemple :

```
[nouveau] *
1   Principal :
2       a est un nombre valant 7
3       début
4           Si (Calcul(a)) = 12, affiche "Mon calcul est correct."
5           Sinon affiche "Il y a une erreur dans mon calcul."
6
7   Calcul :
8       *a est un nombre
9       début
10          a = (a + 5 * 6 - 1) / 3
11      Retourne a
```

Pour cela, il suffit alors de rajouter des parenthèses autour de la fonction à utiliser (ligne 4).

# La récursivité

En informatique, la récursivité est l'action de s'invoquer soi-même.
Voici un exemple de fonction récursive :

```
[nouveau] *
1   Globale
2       a est un nombre valant 3
3
4   Calcul :
5       début
6           a = a -1
7           Si a = 0, affiche a
8           Sinon lis
9               Affiche a
10              Va vers Calcul
11          ferme
```

Ainsi, la fonction Calcul va alors s'appeler trois fois, jusqu'à ce que la variable a atteigne **0**.

Et bien, le verbe **Retourner** permet aussi d'utiliser la récursivité.
Voici le même exemple avec le verbe **Retourner** :

Nouveau.liv *

```
1   Principal :
2      a est un nombre valant 3
3      début
4         Affiche Calcul(a)
5
6   Calcul :
7      *a est un nombre
8      début
9         a = a -1
10        Si a = 0, retourne a
11        Sinon lis
12           Affiche a
13           Retourne Calcul(a)
14        ferme
```

Cet exemple n'utilise pas de **variable globale** !

Et pour finir, voici la suite de Fibonacci :

[nouveau] *

```
1   Principal :
2      n est un nombre
3      début
4         Affiche "Entrez un nombre :"
5         Demande n
6         Affiche Fibo(n)
7
8   Fibo :
9      *n est un nombre
10     début
11        Si n < 2, retourne n
12        Sinon retourne Fibo(n-1) + Fibo(n-2)
```

# L'évolution des espèces

Développons un peu plus les possibilités offertes par l'utilisation des espèces.

## L'héritage

La notion d'héritage permet de créer une nouvelle espèce à partir d'une espèce déjà existante.
Afin de comprendre l'utilité de cette notion, reprenons notre ancien exemple sur les espèces :

```
[nouveau] *
1   Espèces
2      nom est un texte
3      numéro est un texte
4      adresse est un texte valant "adresse inconnue"
5      espèce contact contient nom, numéro, adresse
6
7   mes contacts :
8      Robert est un contact, nom vaut "Bidoche", numéro vaut "06 00 00 00"
9      début
10        Affiche nom de Robert
11        Affiche numéro de Robert
12        Affiche adresse de Robert
```

Imaginons que nous voulons afficher l'attribut nom de notre variable Robert sur la toile.
Pour cela, il faudrait alors utiliser l'espèce graphique graffiti, comme ceci :

```
[nouveau] *
1   Espèces
2      nom est un texte
3      numéro est un texte
4      adresse est un texte valant "adresse inconnue"
5      espèce contact contient nom, numéro, adresse
6
7   mes contacts :
8      Robert est un contact, nom vaut "Bidoche", numéro vaut "06 00 00 00"
9      présentation est un graffiti, couleur vaut "rouge", x vaut 250, y vaut 260
10     début
11        Affiche nom de Robert
12        Affiche numéro de Robert
13        Affiche adresse de Robert
14        texte de présentation vaut nom de Robert
15        Projette présentation
```

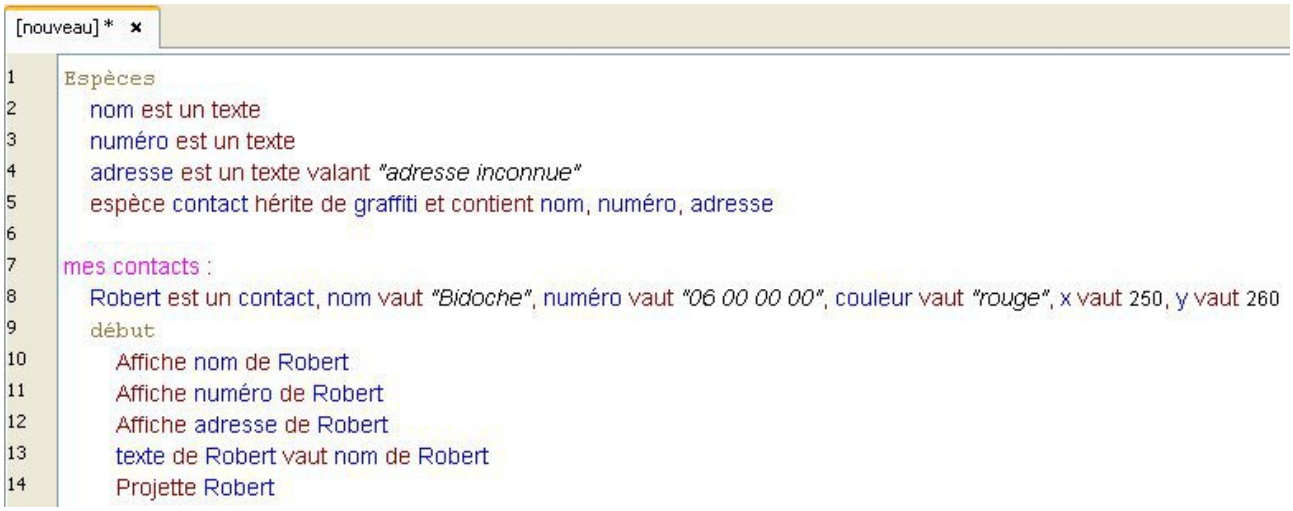
Nous utilisons alors deux espèces distinctes :

- l'espèce contact pour contenir les informations de notre répertoire.
- l'espèce graphique graffiti pour gérer l'affichage sur la toile.

Cette méthode deviendrait alors rapidement compliquée s'il fallait gérer une dizaine de contacts...
Pour simplifier notre exemple, nous allons alors faire hériter l'espèce contact de l'espèce graffiti :


Ainsi, l'espèce contact peut désormais utiliser les verbes liés à l'affichage comme : **Projeter**, **Déplacer**, ...

Et l'espèce contact hérite également de tous les attributs de l'espèce graphique graffiti :




# Les méthodes fonctionnelles

Jusqu'à présent, nous avons vu que les espèces contenait des variables, les **attributs**.
Et bien, les espèces peuvent également disposer de leurs propres fonctions !
Ce sont les **méthodes fonctionnelles**.

Si vous regardez dans le **verbier**, et que vous sélectionnez le menu **Espèces**, observez bien le nouveau menu qui s'affiche.
En effet, sur certaines espèces proposées, une petite flèche affichée à droite vous indiquera une liste de fonctions que vous pouvez sélectionner.
Ce sont là toutes les méthodes fonctionnelles dont dispose l'espèce choisie.
Ces fonctions sont alors utilisables uniquement par cette espèce.
Par exemple, l'espèce **Majordome** dispose de la méthode fonctionnelle **.présentation(**texte**)**
Voyons comment l'utiliser.

Pour cela, commençons par créer une variable de type majordome :

```
majordome.liv *
1  Démonstration :
2      messager est un majordome
3      début
```

Maintenant, appelons la méthode fonctionnelle de notre variable messager :

```
majordome.liv *
1  Démonstration :
2      messager est un majordome
3      début
4          messager.présentation(texte)
```

Ce code n'est pas complet.

**Détaillons la ligne 4 :**

Pour pouvoir appeler une méthode fonctionnelle, on commence donc par indiquer la variable de notre espèce à utiliser :

messager

Ensuite, on indique un point, qui marque l'appel à la méthode fonctionnelle :

messager**.**

Ensuite, on appelle la méthode fonctionnelle que l'on souhaite utiliser :

messager**.**présentation

Pour terminer, il faut savoir que les méthodes fonctionnelles sont des fonctions qui retourne un résultat : comme pour le verbe **Retourner**, elles utilisent donc les parenthèses, ainsi qu'un paramètre.
Le type du paramètre à employer est alors indiqué dans le **verbier.**
Ici, il s'agit d'un type texte :

messager**.**présentation(texte)

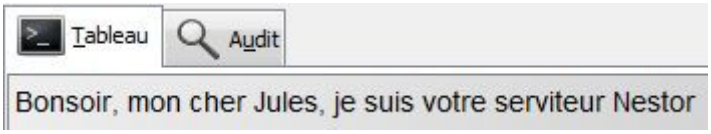
Ajoutons alors notre paramètre, comme ceci :

```
majordome.liv *
1  Démonstration :
2      messager est un majordome
3      début
4          messager.présentation("Jules")
```

Ce code n'est pas complet.

Enfin, comme pour le verbe **Retourner**, une méthode fonctionnelle s'utilise directement avec un verbe, par exemple :


Et voici le résultat :




La méthode fonctionnelle **.présentation(texte)** de l'espèce **majordome** sert donc à afficher une phrase à l'écran.

Mais étudions d'un peu plus près le fonctionnement d'une méthode fonctionnelle.

Pour cela, nous allons créer notre propre méthode fonctionnelle **.présentation(**texte**)**

Car, tout comme nous ajoutons des attributs à une espèce, nous pouvons également lui ajouter nos propres fonctions !

Commençons par créer notre propre espèce majordome :


Pour pouvoir créer une espèce, il nous faut obligatoirement un attribut.
L'attribut prénom désignera alors le nom de notre majordome, en remplacement du prénom **"*Nestor*"** utilisé précédemment.

Ajoutons ensuite à notre espèce majordome sa méthode fonctionnelle **.présentation()**

Pour cela, nous utilisons le mot propose, comme ceci :

```
[nouveau] * ×
1  Espèces
2     prénom est un texte valant "Albert"
3     espèce majordome contient prénom
4     majordome propose présentation
```

Créons alors notre fonction Présentation :

```
[nouveau] * ×
1  Espèces
2     prénom est un texte valant "Albert"
3     espèce majordome contient prénom
4     majordome propose présentation
5
6  Présentation :
7     début
```

Cette fonction étant une méthode fonctionnelle, il convient de respecter 2 règles :

Tout d'abord, il faut indiquer dans le nom de la fonction, l'espèce à laquelle elle appartient :

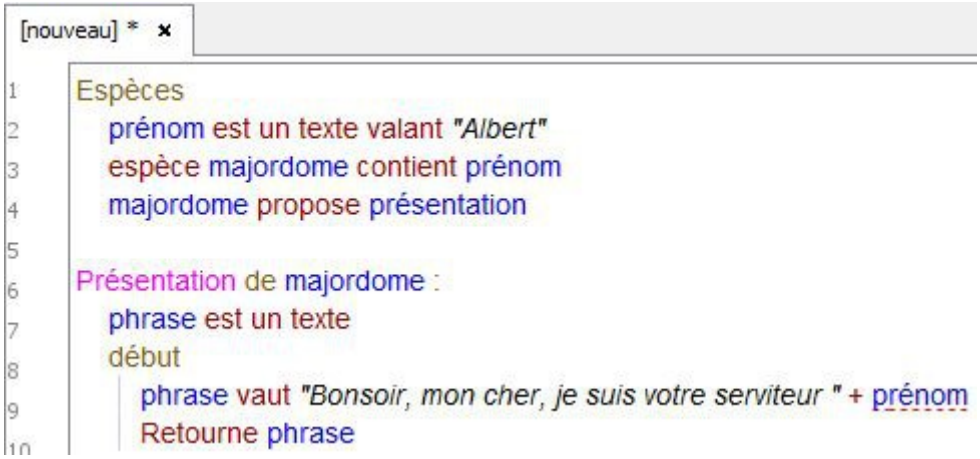
```
[nouveau] * ×
1  Espèces
2     prénom est un texte valant "Albert"
3     espèce majordome contient prénom
4     majordome propose présentation
5
6  Présentation de majordome :
7     début
```

Et ensuite, la méthode fonctionnelle doit utiliser le verbe **Retourner** :

```
[nouveau] * ×
1  Espèces
2     prénom est un texte valant "Albert"
3     espèce majordome contient prénom
4     majordome propose présentation
5
6  Présentation de majordome :
7     phrase est un texte
8     début
9
10       Retourne phrase
```

La variable phrase indiquera alors la phrase à afficher à l'écran.

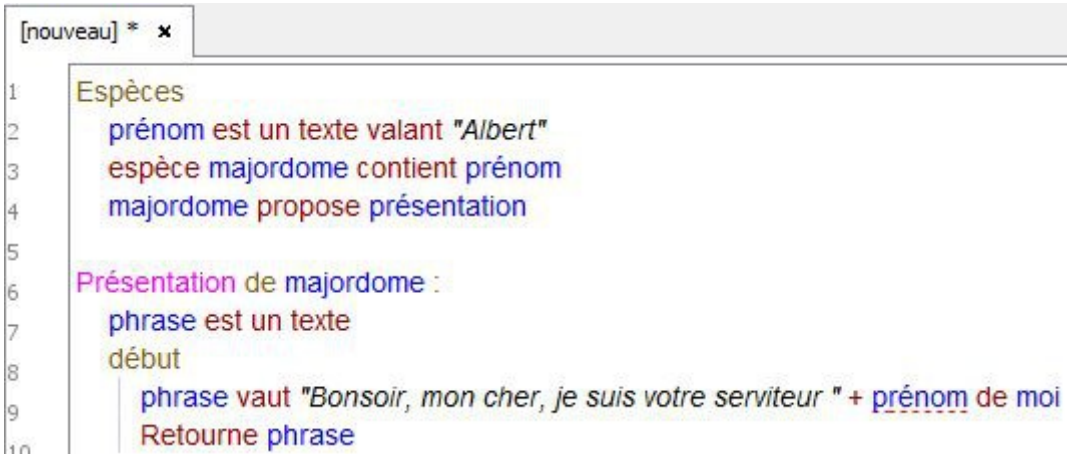
D'ailleurs, ajoutons-la :




Ce code contient une erreur.

En effet, on ne peut utiliser directement un attribut : il faut indiquer à notre attribut prénom à quelle espèce il appartient.

Pour cela, nous allons alors utiliser un **paramètre particulier**, qui ne s'utilise qu'avec les méthodes fonctionnelles, la variable moi :




On remarque que la paramètre particulier **moi** n'a pas besoin d'être déclaré.

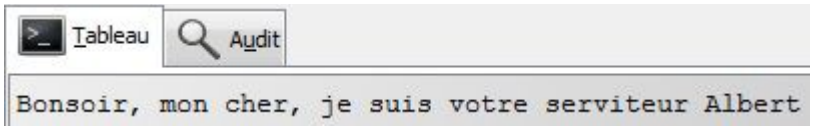
Il nous reste plus qu'à reprendre notre fonction Démonstration utilisé précédemment, qui se chargera d'appeler notre méthode fonctionnelle :

```
[nouveau] *
1  Espèces
2     prénom est un texte valant "Albert"
3     espèce majordome contient prénom
4     majordome propose présentation
5
6  Présentation de majordome :
7     phrase est un texte
8     début
9        phrase vaut "Bonsoir, mon cher, je suis votre serviteur " + prénom de moi
10       Retourne phrase
11
12 Démonstration :
13    messager est un majordome
14    début
15       Affiche messager.présentation()
```

Ce code fonctionne !

En effet, lors de son appel à la ligne 16, notre méthode fonctionnelle étant attachée à la variable messager, notre paramètre particulier moi va alors remplacer cette variable, nous permettant ainsi d'utiliser son attribut prénom.

Et d'afficher ceci :




En revanche, dans notre précédente fonction Démonstration, on pouvait personnaliser le message à afficher en indiquant le prénom "*Jules*".
Pour cela, rajoutons à notre méthode fonctionnelle un paramètre :

[nouveau] *

```
Espèces
    prénom est un texte valant "Albert"
    espèce majordome contient prénom
    majordome propose présentation

Présentation de majordome :
    *nom est un texte
    phrase est un texte
    début
        phrase vaut "Bonsoir, mon cher " + nom + ", je suis votre serviteur " + prénom de moi
        Retourne phrase

Démonstration :
    messager est un majordome
    début
        Affiche messager.présentation("Jules")
```

Et voilà le travail ! Nous venons ainsi de recréer la méthode fonctionnelle **.présentation(**texte**)**

**Une remarque :**

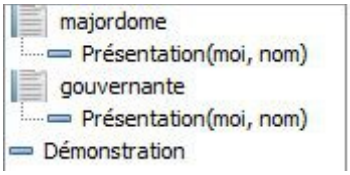
Lorsque vous avez plusieurs espèces, vous pouvez utiliser le même nom pour vos méthodes fonctionnelles.
Voici un exemple :

```
[nouveau] *
1  Espèces
2     prénom est un texte valant "Albert"
3     espèce majordome contient prénom
4     majordome propose présentation
5     prénom2 est un texte valant "Roberta"
6     espèce gouvernante contient prénom2
7     gouvernante propose présentation
8
9  Présentation de majordome :
10    *nom est un texte
11    phrase est un texte
12    début
13       phrase vaut "Bonsoir, mon cher " + nom + ", je suis votre serviteur " + prénom de moi
14       Retourne phrase
15
16 Présentation de gouvernante :
17    *nom est un texte
18    phrase est un texte
19    début
20       phrase vaut "Bonsoir, mon cher " + nom + ", je suis votre gouvernante " + prénom2 de moi
21       Retourne phrase
22
23 Démonstration :
24    messager est un majordome
25    messagère est une gouvernante
26    début
27       Affiche messager.présentation("Jules")
         Affiche messagère.présentation("Jules")
```

Ici, le paramètre particulier **moi** remplace alors les variables **messager** et **messagère**.

Ainsi, les deux espèces majordome et gouvernante disposent chacune d'une méthode fonctionnelle **.Présentation(**texte**)**.

Dans la partie **Sommaire** de l'atelier, vous pouvez alors constater que les méthodes fonctionnelles se rangent automatiquement sous l'espèce qui leur correspond :




Ceci vous permet alors de repérer facilement les fonctions de votre code qui appartiennent à une espèce.

De cette façon, si une erreur se glisse dans une méthode fonctionnelle, elle n'affectera que l'espèce

concernée : si la méthode fonctionnelle **.Présentation(**texte**)** de l'espèce majordome contient une erreur, elle n'affectera que cette espèce, et non l'espèce gouvernante.
De plus, les méthodes fonctionnelles étant indépendantes, contrairement aux fonctions accessibles depuis tout le livre, vous pourrez alors corriger votre méthode fonctionnelle sans devoir modifier les autres fonctions de votre code.

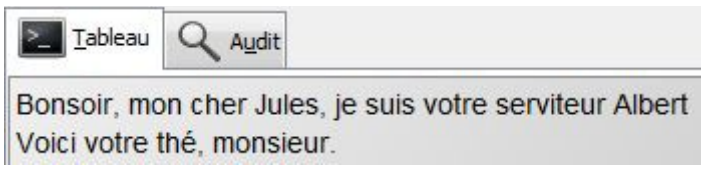
**Une autre remarque :**

Vous pouvez bien entendu ajouter plusieurs fonctions à une espèce.
Pour cela, il suffit d'utiliser la virgule, comme pour les attributs :

[nouveau] * ×

```
1  Espèces
2     prénom est un texte valant "Albert"
3     espèce majordome contient prénom
4     majordome propose présentation, service
5
6  Présentation de majordome :
7     *nom est un texte
8     phrase est un texte
9     début
10       phrase vaut "Bonsoir, mon cher " + nom + ", je suis votre serviteur " + prénom de moi
11       Retourne phrase
12
13 Service de majordome :
14    phrase est un texte valant "Voici votre thé, monsieur."
15    début
16       Retourne phrase
17
18 Démonstration :
19    messager est un majordome
20    début
21       Affiche messager.présentation("Jules")
22       Affiche messager.service()
```

Et voici le résultat :

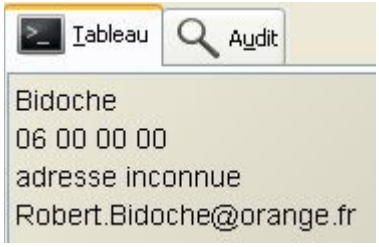



# Le verbe Attacher

Le verbe **Attacher** permet d'ajouter un nouvel attribut à une variable de type espèce.
Reprenons un ancien exemple, notre espèce contact, et ajoutons lui une adresse e-mail :

```
[nouveau] *
1  Espèces
2     nom est un texte
3     numéro est un texte
4     adresse est un texte valant "adresse inconnue"
5     espèce contact contient nom, numéro, adresse
6
7  Mes contacts :
8     Robert est un contact, nom vaut "Bidoche", numéro vaut "06 00 00 00"
9     mail est un texte
10    début
11       Attache mail à Robert
12       mail de Robert vaut "Robert.Bidoche@orange.fr"
13       Affiche Robert
```

Et voici le résultat :




Comme vous pouvez le constater, le verbe **Attacher** est très simple d'utilisation.

La variable mail est désormais un attribut de la variable Robert.

Attention cependant, la variable **mail** n'est pas une nouvelle caractéristique de l'espèce **contact**.

Cette variable ne se trouvant pas dans la section Espèces, si vous créez une nouvelle variable de type contact, elle ne disposera pas de l'attribut mail.

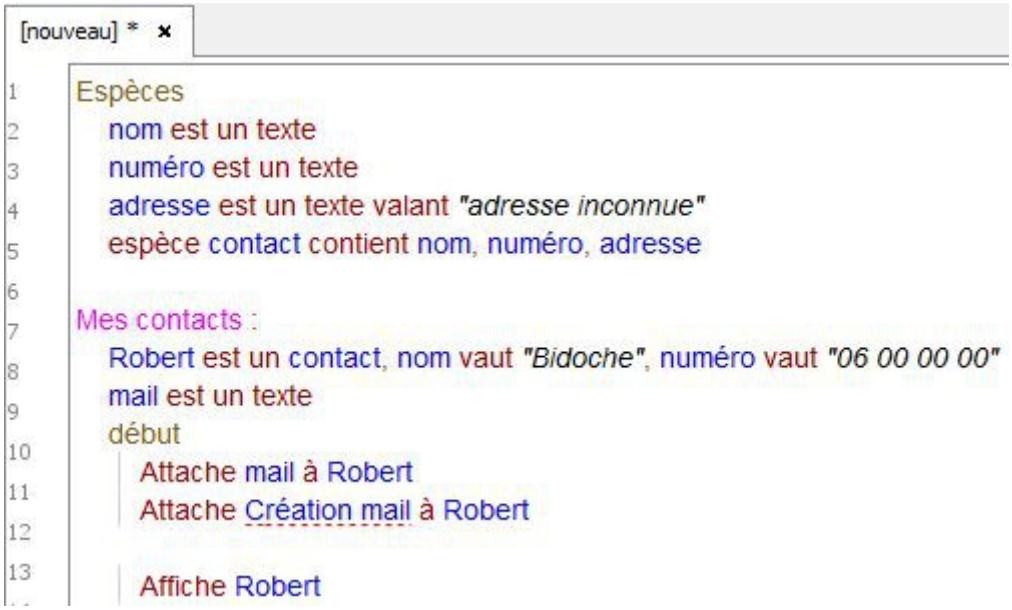
Le verbe **Attacher** permet également d'attacher une méthode fonctionnelle à une variable de type espèce.

Dans notre exemple précédent, nous avons écrit l'adresse e-mail de notre contact :

```
mail de Robert vaut "Robert.Bidoche@orange.fr"
```

Créons alors une méthode fonctionnelle **.Création mail()** qui se chargera elle-même d'écrire la valeur de l'attribut mail.

Commençons par attacher cette fonction à notre espèce :




L'utilisation du verbe **Attacher** est la même avec les fonctions qu'avec les attributs.

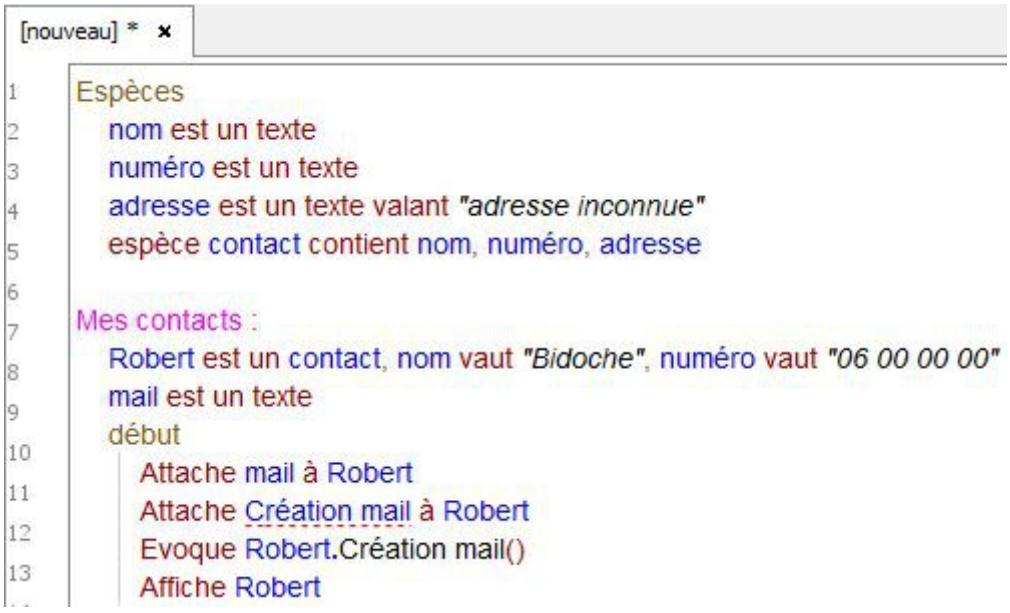
Ensuite, appelons notre méthode fonctionnelle en utilisant un verbe, par exemple, le verbe **Afficher**.

Mais l'intégralité de la variable Robert sera déjà afficher à la ligne 14 :

```
Affiche Robert
```

Pour effectuer l'appel de notre méthode fonctionnelle, nous allons alors utiliser un autre verbe : le verbe **Évoquer**.

Ce dernier permet d'appeler une fonction retournant un résultat ou une méthode fonctionnelle, sans effectuer aucune autre action :

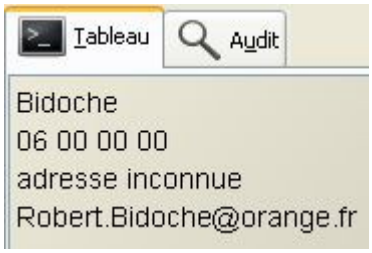


Enfin, il nous reste plus qu'à créer notre fonction Création mail :

```
[nouveau] *
1  Espèces
2     nom est un texte
3     numéro est un texte
4     adresse est un texte valant "adresse inconnue"
5     espèce contact contient nom, numéro, adresse
6
7  Mes contacts :
8     Robert est un contact, nom vaut "Bidoche", numéro vaut "06 00 00 00"
9     mail est un texte
10    début
11       Attache mail à Robert
12       Attache Création mail à Robert
13       Evoque Robert.Création mail()
14       Affiche Robert
15 Création mail :
16    début
17       mail de moi vaut "Robert." + nom de moi + "@orange.fr"
18       Retourne mail de moi
```

**Création mail** étant une méthode fonctionnelle, elle peut utiliser le paramètre particulier **moi**.

Et voici le résultat :




Attention cependant, la méthode fonctionnelle **.Création mail()** n'est pas une nouvelle caractéristique de l'espèce **contact**.

C'est pourquoi nous n'avons pas nommé notre méthode fonctionnelle comme ceci :

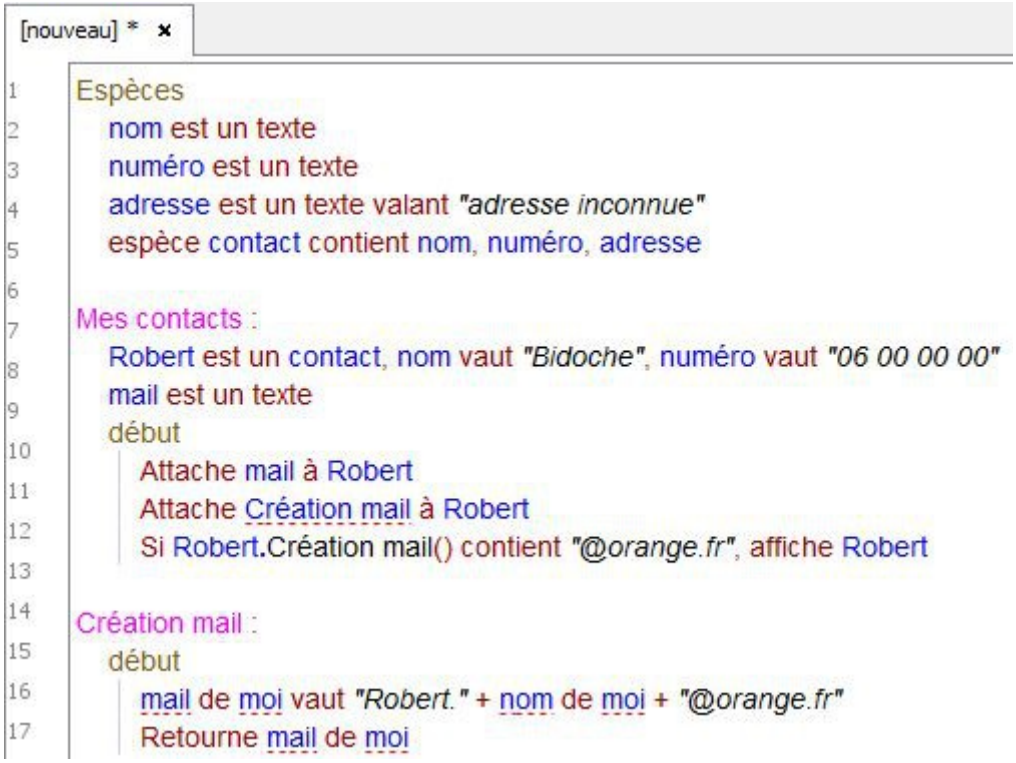
```
Création mail de contact :
   début
```

En effet, cette fonction ne se trouvant pas dans la section Espèces, cette méthode fonctionnelle n'appartient qu'à notre variable Robert.

Ainsi, si vous créez une nouvelle variable de type contact, elle ne disposera pas de la méthode fonctionnelle **.Création mail()**.

**Une astuce :**

On peut même utiliser une méthode fonctionnelle au sein d'une condition.
Vérifions par exemple que notre méthode fonctionnelle **.Création mail()** utilise le bon service de messagerie :




A la ligne 13, la condition va donc vérifier que l'attribut mail, retourné par notre méthode fonctionnelle, contienne bien la valeur "*@orange.fr*".

# Affichage dynamique du nom des variables

Dans les chapitres précédents, nous avons appris à créer des méthodes fonctionnelles.
Comme elles utilisent des paramètres, elles peuvent donc être utilisées par différentes variables.
Ainsi, si on ajoute dans un de nos exemples précédents une variable Germaine de type contact, elle pourra avoir sa propre adresse e-mail :

```
[nouveau] *
1  Espèces
2     nom est un texte
3     numéro est un texte
4     adresse est un texte valant "adresse inconnue"
5     espèce contact contient nom, numéro, adresse
6
7  Mes contacts :
8     Robert est un contact, nom vaut "Bidoche", numéro vaut "06 00 00 00"
9     Germaine est un contact, nom vaut "Dupont", numéro vaut "06 00 00 01"
10    mail est un texte
11    début
12       Attache mail à Robert
13       Attache Création mail à Robert
14       Evoque Robert.Création mail()
15       Affiche Robert
16       Attache mail à Germaine
17       Attache Création mail à Germaine
18       Evoque Germaine.Création mail()
19       Affiche Germaine
20
21 Création mail :
22    début
23       mail de moi vaut "Robert." + nom de moi + "@orange.fr"
         Retourne mail de moi
```

Et voici le résultat :

```
Tableau  Audit
Bidoche
06 00 00 00
adresse inconnue
Robert.Bidoche@orange.fr
Dupont
06 00 00 01
adresse inconnue
Robert.Dupont@orange.fr
```

Le contact **Germaine** dispose alors de son propre attribut **mail** et de sa propre méthode

fonctionnelle **.Création mail()**.

Mais ce livre possède une anomalie : l'adresse e-mail de Germaine.

```
Robert.Dupont@orange.fr
```

Son prénom ne correspond pas.

Dans cette ligne :

```
mail de moi vaut "Robert." + nom de moi + "@orange.fr"
```

Il faudrait alors remplacer ***"Robert"*** par ***"Germaine"***.

Mais pour que notre méthode fonctionnelle puisse utiliser les deux prénoms, on devrait alors utiliser le paramètre particulier moi :

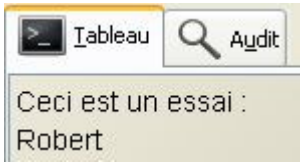
```
mail de moi vaut prénom de moi +"." + nom de moi + "@orange.fr"
```

Ce code ne fonctionne pas.

En effet, l'espèce contact ne dispose pas d'un attribut prénom. Robert et Germaine sont simplement les noms de nos deux variables.

Heureusement, il existe un moyen d'afficher directement le nom d'une variable.
Il suffit d'utiliser des crochets, comme nous le montre cet exemple très simple :


Voici le résultat :




L'utilisation de crochets **[ ]** nous permet donc d'afficher directement le nom d'une variable.

Ainsi, le paramètre particulier moi renvoyant, soit sur la variable Robert, soit sur la variable Germaine, on peut alors modifier notre exemple comme ceci :

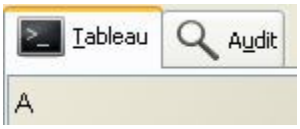
```
[nouveau] *
1  Espèces
2     nom est un texte
3     numéro est un texte
4     adresse est un texte valant "adresse inconnue"
5     espèce contact contient nom, numéro, adresse
6
7  Mes contacts :
8     Robert est un contact, nom vaut "Bidoche", numéro vaut "06 00 00 00"
9     Germaine est un contact, nom vaut "Dupont", numéro vaut "06 00 00 01"
10    mail est un texte
11    début
12       Attache mail à Robert
13       Attache Création mail à Robert
14       Evoque Robert.Création mail()
15       Affiche Robert
16       Attache mail à Germaine
17       Attache Création mail à Germaine
18       Evoque Germaine.Création mail()
19       Affiche Germaine
20
21 Création mail :
22    début
23       mail de moi vaut [moi] +"." + nom de moi + "@orange.fr"
         Retourne mail de moi
```

Et voici le résultat :

# Chargement dynamique des variables

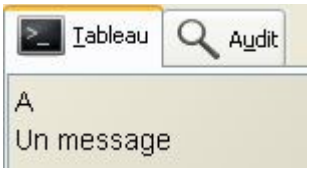
Voici une nouvelle notion qui ne fonctionne qu'avec les variables de type texte.
Prenons cet exemple :


Voici son résultat :




Utilisons maintenant le chargement dynamique :


Voici le résultat :




En utilisant les signes << et >> autour de la variable, vous pouvez alors accéder à la variable A, nommée par la valeur de C.

Ceci fonctionne également :

```
[nouveau] *
1  Chargement dynamique :
2     A est un texte valant "Un message"
3     B est un texte valant "Un deuxième message"
4     C est un texte valant "A"
5     début
6        Affiche C
7        Affiche <<C>>
8        C vaut "B"
9        Affiche <<C>>
```

Et voici le résultat :


Et enfin, vous pouvez combiner le chargement dynamique avec l'affichage dynamique :

```
[nouveau] *
1  Chargement dynamique :
2     A est un texte valant "Un message"
3     B est un texte valant "Un deuxième message"
4     C est un texte valant "A"
5     début
6        Affiche C
7        Affiche <<C>>
8        C vaut "B"
9        Affiche <<C>>
10       Affiche [<<C>>]
```

Et voici le résultat :

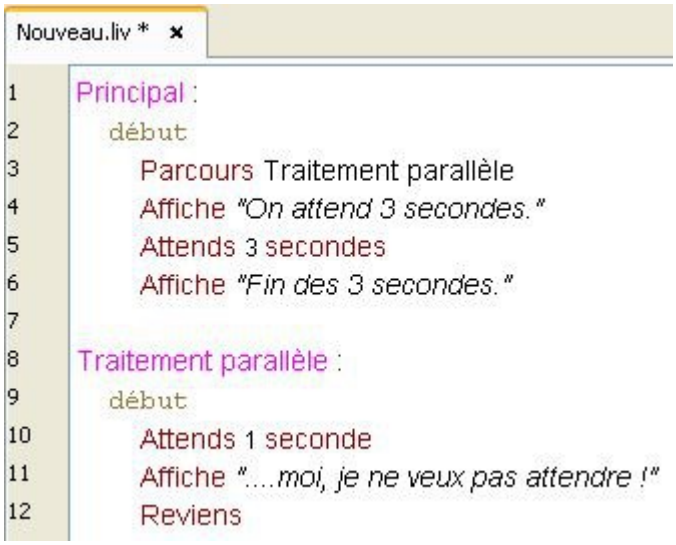

À la ligne 9, en utilisant les signes << et >> autour de la variable C, vous accédez alors à la valeur de la variable B.
À la ligne 10, l'utilisation des crochets **[ ]** permet alors d'afficher le nom de la variable qui est utilisée, soit B.

# Parallélisation des traitements

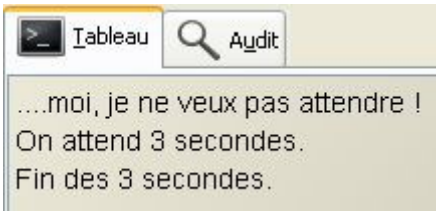
Après l'instruction Va vers..., les verbes **Parcourir** et **Revenir**, le verbe **Retourner**, et enfin le verbe **Évoquer**, il nous reste à voir un dernier verbe permettant de se déplacer dans un livre : le verbe **Appeler**.

La particularité de ce dernier est qu'il permet d'exécuter plusieurs fonctions en même temps.

Voici un premier exemple utilisant le verbe **Parcourir** :




Et son résultat :

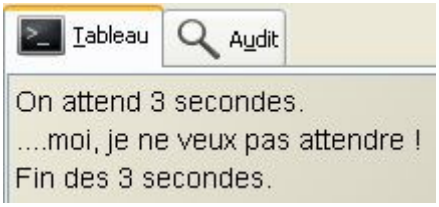



A la ligne 3, on va dans la fonction Traitement parallèle.
On arrive donc à la ligne 10, où le programme se met en pause pendant **1** seconde, avant d'exécuter la ligne 11 et d'afficher son message.
On reviens ensuite à la ligne 4, où l'on exécute la suite de la fonction Principal, affichant ainsi les deux derniers messages.

Voici le même exemple en utilisant le verbe **Appeler** :

```
Nouveau.liv *
1  Principal :
2    début
3      Appelle Traitement parallèle
4      Affiche "On attend 3 secondes."
5      Attends 3 secondes
6      Affiche "Fin des 3 secondes."
7
8  Traitement parallèle :
9    début
10     Attends 1 seconde
11     Affiche "....moi, je ne veux pas attendre !"
12     Reviens
```

Et son résultat :



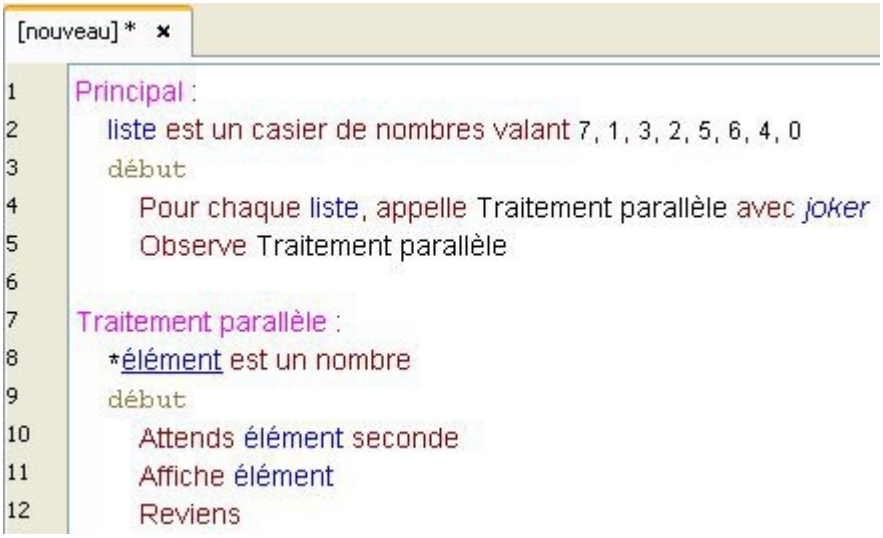

Les deux résultats sont différents !

Dans la fonction Principal, on commence par appeler la fonction Traitement parallèle.
On arrive donc à la ligne 10, où le programme se met en pause pendant **1** seconde.
Pourtant, le message de la ligne 4 s'affiche directement sur le tableau, avant de se mettre en pause pendant **3** secondes.
Dans le fonction Traitement parallèle, la seconde étant écoulée, la ligne 11 alors s'exécute, affichant le deuxième message.
Puis, dans la fonction Principal, à la fin des **3** secondes, la ligne 6 s'exécute, affichant alors le dernier message.

Verdict : les deux fonctions se sont exécutées en même temps !

À la différence des autres verbes, qui exécutent chaque fonction les unes après les autres, le verbe **Appeler** permet donc d'utiliser plusieurs fonctions simultanément.

Et voici un dernier exemple que vous pouvez exécuter :

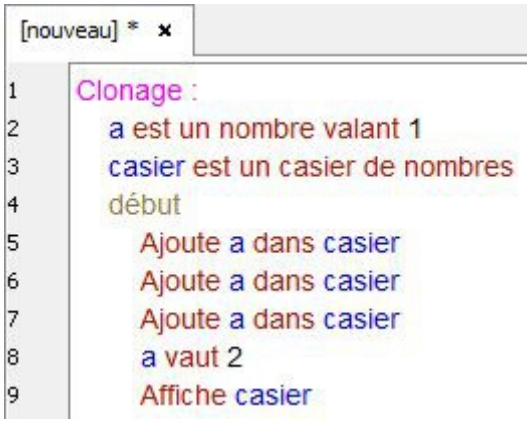



Le verbe **Appeler** peut utiliser des paramètres !

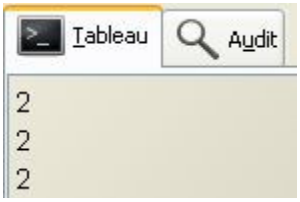
Le verbe **Observer** permet d'attendre que tous les traitements parallèles soient terminés avant de poursuivre l'exécution du programme, c'est-à-dire pour cet exemple, d'atteindre la ligne 6 et donc de terminer le livre.

# Le clonage

Linotte supporte le clonage, c'est-à-dire la duplication d'un objet à l'identique.
Prenons un exemple simple :

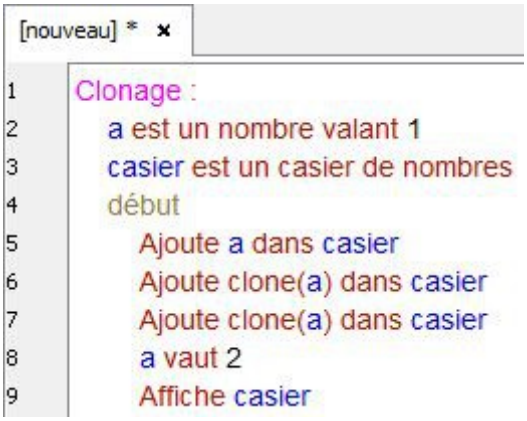



Voici le résultat :




Comme nous avons ajouté trois fois le nombre a dans le casier, la modification de sa valeur est effective avec les trois variables du casier.

Reprenons le même exemple mais en utilisant le clonage :

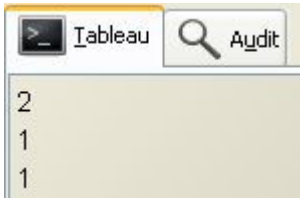



Pour cloner un objet nous écrivons alors ceci :

clone()

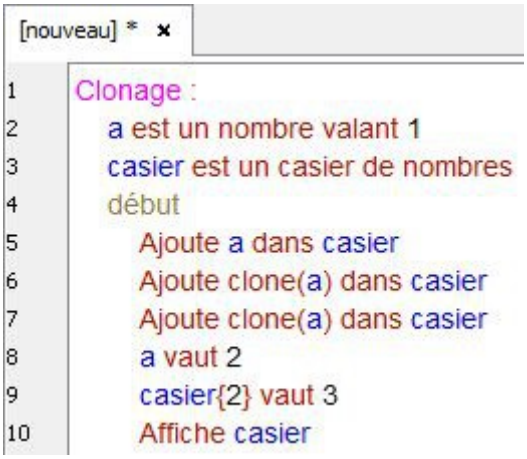
En insérant dans les parenthèses la variable que l'on veut dupliquer.

Et voici le résultat :




Les deux dernières variables du casier ne sont plus a. Ce sont des copies de a : leur valeur est donc identique à la valeur de a au moment de la duplication.
Modifier la valeur de a ne change alors pas la valeur de ses clones.

Pour pouvoir modifier la valeur d'un clone, il faut alors faire ceci :

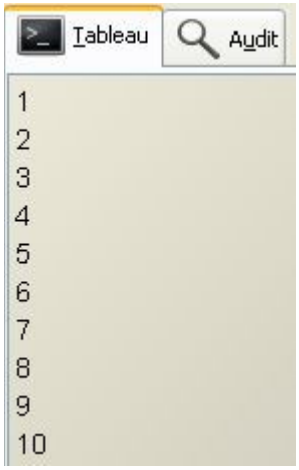



Et voici le résultat :


Voici un exemple où nous insérons dix nombres dans un casier :

```
[nouveau] *
1  Clonage :
2      a est un nombre valant 1
3      casier est un casier de nombres
4      début
5          Pour chaque 10, lis
6              Parcours Ajouter avec clone(a), casier
7              a vaut a + 1
8          ferme
9          Affiche casier
10
11 Ajouter :
12     *copie_a est un nombre
13     *casier est un casier de nombres
14     début
15         Ajoute copie_a dans casier
16         Reviens
```

A chaque passage dans la boucle, la valeur de a est incrémentée de **1**.
A chaque passage, chaque clone de a copie alors sa nouvelle valeur.
Ce qui nous donne le résultat suivant :




Observez maintenant le résultat en supprimant clone() de notre exemple.

Pour conclure ce chapitre, voici un dernier exemple utilisant le graphisme :

[nouveau] *

```
1  Clonage :
2     rond est un cercle, couleur vaut "rose", plein vaut "oui", rayon vaut 30, x vaut 289, y vaut 290
3     groupe est un casier de cercle
4     début
5        Projette rond
6        Ajoute clone(rond) dans groupe
7        Ajoute clone(rond) dans groupe
8        Ajoute clone(rond) dans groupe
9        De 1 à 100, lis
10          Déplace rond vers la gauche de 1
11          Déplace groupe{0} vers la droite de 1
12          Déplace groupe{1} vers le haut de 1
13          Déplace groupe{2} vers le bas de 1
14          Attends 20 millisecondes
15       ferme
```

Le clonage peut s'utiliser avec n'importe quel type de variable !

# Les souffleurs

Les **souffleurs** permettent d'utiliser l'instruction **Dès que…, …**

Ils se déclarent dans la section souffleurs, que l'on ajoute dans une fonction, avant le mot début.

Voici un exemple de programme qui tournerait à l'infini si il n'y avait pas de souffleur :


Un **souffleur** équivaut donc à une condition.
Dès que la condition est remplie, vous pouvez alors effectuer n'importe quelle action.


En revanche, vous ne pouvez pas utiliser de bloc :


Ceci est interdit.

Vous ne pouvez donc utiliser qu'une seule action par souffleur.

**Une remarque importante :**

Un souffleur se déclenche **après** que la condition soit remplie.

Dans notre exemple, la variable nombre atteint d'abord **10** et, après seulement, le souffleur s'active.

Pour y voir plus clair, complétons notre exemple comme ceci :


Si vous exécutez ce livre, vous pouvez alors constater que, dès que notre variable nombre atteint **10** :
- nombre s'affiche sur le tableau,
- nombre2 s'affiche à son tour,
- l'instruction Va vers **Boucle** s'exécute,
- et enfin, le souffleur se déclenche...

Il convient de ne pas l'oublier lorsque l'on utilise un souffleur...

# Les événements

Un événement est une action effectuée par l'utilisateur sur la toile.
La gestion des événements va alors nous permettre de faire réagir une espèce graphique à cette action.
Par exemple, cliquer sur la toile est un événement :


**Détaillons l'écriture d'un événement :**

A la ligne 5, on fais réagir la variable soleil à l'événement **clic souris** :

```
Fais réagir soleil à "clic souris"
```

On remarque la présence de guillemets **" "** autour de l'événement.

Lorsque cet événement est détecté par l'interprète Linotte, on déclenche la fonction Changer la couleur, à l'aide du mot pour :

```
Fais réagir soleil à "clic souris" pour Changer la couleur
```

La fonction Changer la couleur utilise alors un paramètre :

```
Changer la couleur :
  *objet est un cercle
  début
```

Ainsi, la variable réagissant à l'événement est automatiquement envoyée à la fonction appelée.

Le type du paramètre doit alors correspondre au type de la variable déclenchant l'événement.

Malgré cela, la fonction Changer la couleur ne peut prendre aucun autre paramètre ; c'est pourquoi on indique pas de parenthèses après le nom de la fonction :

```
Fais réagir soleil à "clic souris" pour Changer la couleur()
```

Cet exemple ne fonctionne pas.

Enfin, on termine la fonction par le verbe **Revenir** :

```
Changer la couleur :
   *objet est un cercle
   début
      couleur de objet vaut "rouge"
      Reviens
```

De plus, sachez que lors d'un événement, la fonction est appelée en parallèle : l'exécution de la fonction Décor n'est donc pas arrêtée lors du déclenchement de l'événement.

**Une remarque :**

Un événement se déclenche seulement si vous cliquez sur l'objet qui lui est associé. Si vous cliquez ailleurs sur la toile, l'événement ne se déclenchera pas.

# Voici la liste des événements disponibles :

**double clic souris**
Fais réagir l'espèce graphique au double-clique de la souris.

**souris entrante**
Fais réagir l'espèce graphique à l'entrée du pointeur de la souris dans sa surface.

**souris sortante**
Fais réagir l'espèce graphique à la sortie du pointeur de la souris de sa surface.

**début glisser-déposer**
Fais réagir l'espèce graphique au maintient du clique, suivit du déplacement de la souris.

**glisser-déposer**
Fais réagir l'espèce graphique au maintient du clique, accompagné du déplacement de la souris et enfin, de son relâchement.

**Une remarque :**

Il est possible de faire réagir une même variable à plusieurs événements.

# Les bibliothèques

Jusqu'à présent, nous ne pouvions exécuter qu'un seul livre à la fois.
Et bien, les bibliothèques vont nous permettre d'utiliser plusieurs livres simultanément.

Nous allons prendre comme exemple la suite de Fibonacci. Recopiez ceci dans votre cahier :


Imaginons que nous voulons connaître le temps que met l'interprète à calculer le résultat.
Au lieu de modifier notre exemple, nous allons créer un nouveau programme.

Ouvrez un nouveau livre (**Bibliothèques** > **Nouveau livre**).
Recopiez-y ce code :

**Expliquons ce livre :**

Nous commençons par déclarer trois variables globales :

```
Globale
   _seconde est un nombre
   _minute est un nombre
   _heure est un nombre
```

Ensuite, nous créons la fonction Start :

Dans cette fonction, nous donnons à nos trois variables globales les valeurs des variables seconde, minute et heure.

```
Start :
   début
      _seconde vaut seconde
      _minute vaut minute
      _heure vaut heure
      Reviens
```

**seconde**, **minute** et **heure** sont des **variables particulières** du Linotte : elles indiquent l'heure, la minute et la seconde en cours.

Enfin, nous créons la fonction Stop :

Dans cette fonction, nous récupérons l'heure, la minute et la seconde en cours dans les variables tmp_heure, tmp_minute et tmp_seconde.

```
Stop :
   * temps est un texte
   tmp_seconde est un nombre valant seconde
   tmp_minute est un nombre valant minute
   tmp_heure est un nombre valant heure
```

Puis, nous écrivons cette ligne :

```
temps vaut tmp_heure * 3600 + tmp_minute * 60 + tmp_seconde - (_heure * 3600 + _minute * 60 + _seconde)
```

**Détaillons-la :**

Nous soustrayons les valeurs des variables tmp_heure, tmp_minute et tmp_seconde à l'heure de départ, contenue dans les variables globales, pour connaître le temps qui s'est écoulé.

```
tmp_heure + tmp_minute + tmp_seconde - (_heure + _minute + _seconde)
```

Nous convertissons le tout en secondes :

```
tmp_heure * 3600 + tmp_minute * 60 + tmp_seconde - (_heure * 3600 + _minute * 60 + _seconde)
```

Et nous indiquons le résultat dans le paramètre temps :

```
temps vaut tmp_heure * 3600 + tmp_minute * 60 + tmp_seconde - (_heure * 3600 + _minute * 60 + _seconde)
```

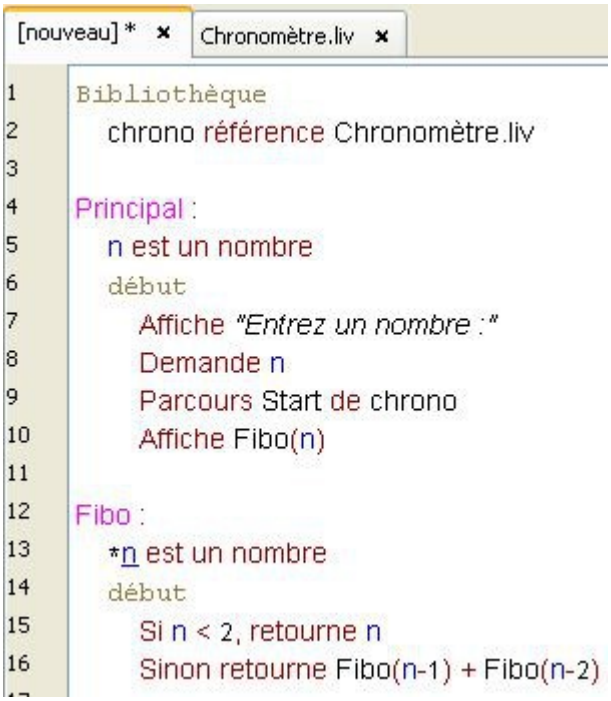
Notre programme pour calculer le temps écoulé est alors terminé.

Vous pouvez désormais l'enregistrer sur votre disque dur (**Bibliothèques** > **Ranger sous**) : sauvegardez-le sous le nom **Chronomètre.liv**.

Il nous reste plus qu'à ajouter notre programme dans notre suite de Fibonacci.

Pour cela, revenons à notre premier livre et commençons par créer une section Bibliothèque, avant notre fonction Principal.
Dans cette section, nous allons créer la bibliothèque **chrono** qui fera référence à notre livre **Chronomètre.liv** :

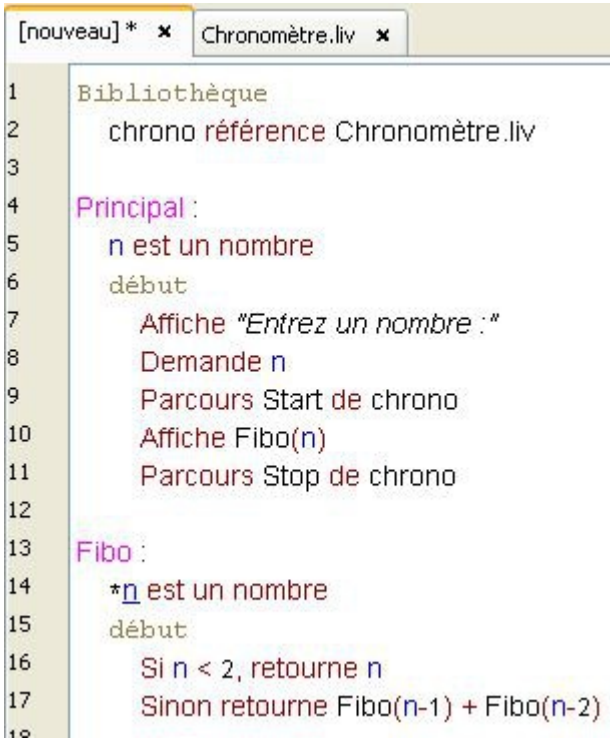

Ensuite, dans notre fonction Principal, avant de commencer le calcul, ajoutons un appel à la fonction Start de notre livre **Chronomètre.liv** :



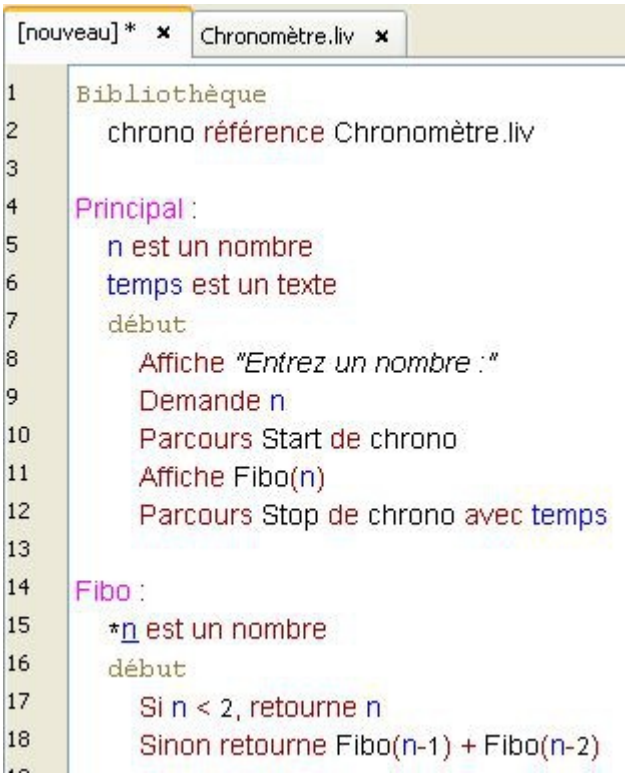

Lors de l'utilisation d'une bibliothèque, nous utilisons alors le verbe **Parcourir** pour appeler notre fonction.

Notre fonction Start n'utilisant pas de paramètre, nous n'avons alors pas besoin de lui transmettre une variable.

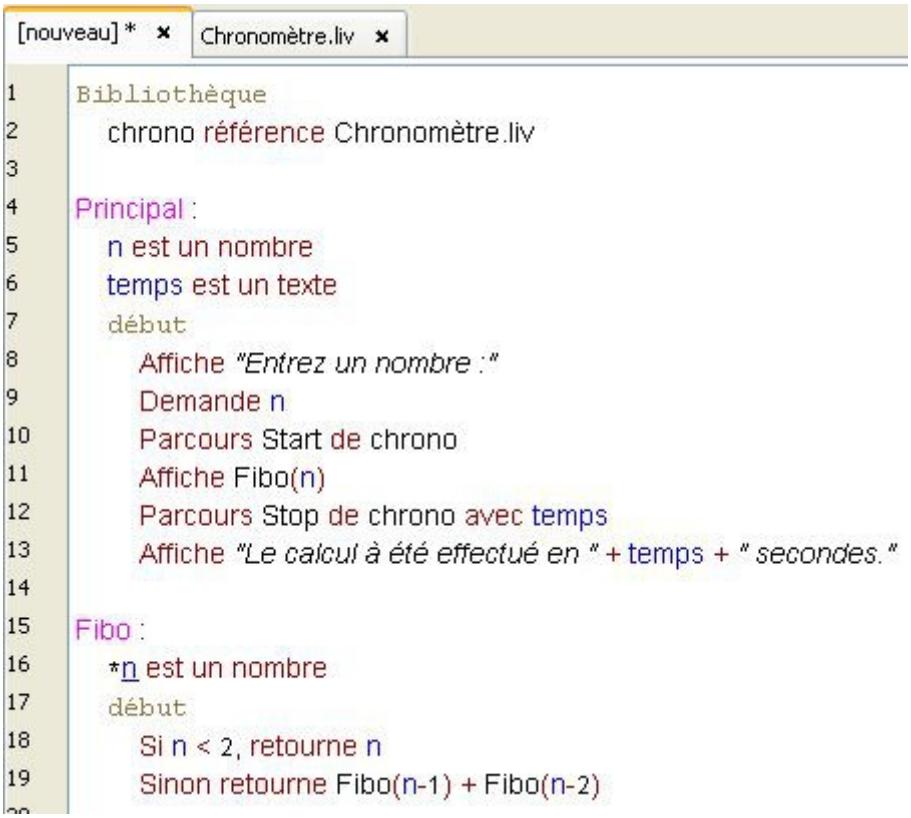
Une fois le calcul effectué, nous pouvons faire un appel à la fonction Stop de notre livre **Chronomètre.liv** :




Mais ici, notre fonction Stop prend un paramètre : il faut donc l'ajouter.



Enfin, il ne reste plus qu'a afficher le résultat :




Sauvegardez alors votre livre :

En effet, pour pouvoir utiliser les bibliothèques, il faut que chaque livre se trouve dans le même répertoire.

Vous pouvez alors tester votre programme.

**Une remarque :**

Votre livre **Chronomètre.liv** n'est pas fait pour être exécuté seul : il se doit d'être utilisé uniquement comme bibliothèque.

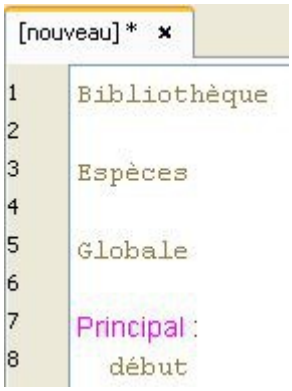
Par sécurité, vous pouvez alors modifier votre programme comme ceci :

[nouveau] * | Chronomètre.liv *

```
Globale
   _seconde est un nombre
   _minute est un nombre
   _heure est un nombre

Principal :
   début
      affiche "Ne pas exécuter directement ce livre !"

Start :
   début
      _seconde vaut seconde
      _minute vaut minute
      _heure vaut heure
      Reviens

Stop :
   * temps est un texte
   tmp_seconde est un nombre valant seconde
   tmp_minute est un nombre valant minute
   tmp_heure est un nombre valant heure
   début
      temps vaut tmp_heure * 3600 + tmp_minute * 60 + tmp_seconde - (_heure * 3600 + _minute * 60 + _seconde)
      Reviens
```

**Une remarque importante :**

Veuillez respecter l'ordre d'écriture des différentes sections :




Si vous ne respectez pas cet ordre, votre programme ne fonctionnera pas.

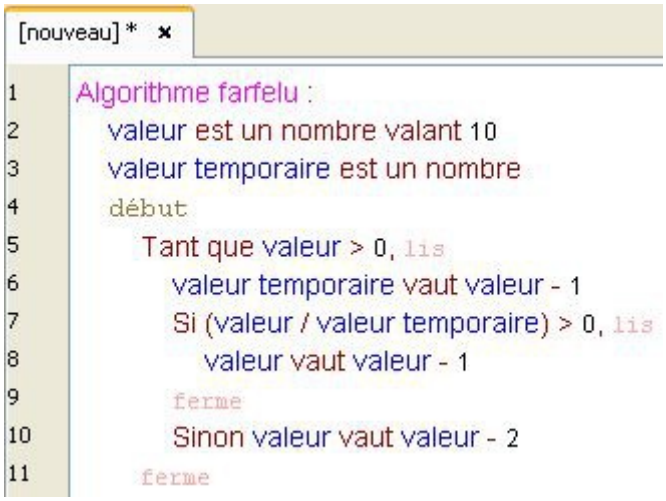
Si vous ne vous en souvenez pas, ouvrez un nouveau livre et allez dans **Le verbier > +Livre**.

# Le débogage

Nous sommes tous un peu des têtes de linotte... Une erreur de programmation est vite arrivée.
Pour éviter que notre programme "plante" subitement et surtout, pour mieux comprendre l'origine du problème, le Linotte met à notre disposition plusieurs solutions de dépannage :

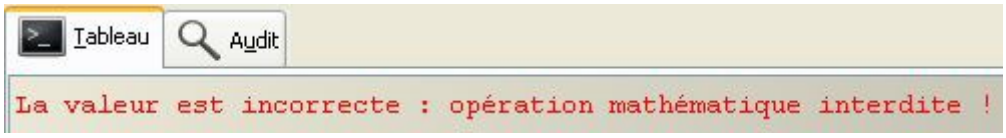
## Le verbe Essayer

Prenons cet exemple :




Ce programme ne fonctionne pas.

En effet, voici le message d'erreur qui s'affiche sur le tableau :




Pour comprendre ce qui se passe, entourons alors notre algorithme du bloc **Essayer** :

```
[nouveau] *
1  Algorithme farfelu :
2     valeur est un nombre valant 10
3     valeur temporaire est un nombre
4     début
5        Essaie lis
6           Tant que valeur > 0, lis
7              valeur temporaire vaut valeur - 1
8              Si (valeur / valeur temporaire) > 0, lis
9                 valeur vaut valeur - 1
10             ferme
11             Sinon valeur vaut valeur - 2
12          ferme
13       ferme
```

En cas d'erreur, le verbe **Essayer** nous permet alors de compléter notre code par une condition :

```
[nouveau] *
1  Algorithme farfelu :
2     valeur est un nombre valant 10
3     valeur temporaire est un nombre
4     début
5        Essaie lis
6           Tant que valeur > 0, lis
7              valeur temporaire vaut valeur - 1
8              Si (valeur / valeur temporaire) > 0, lis
9                 valeur vaut valeur - 1
10             ferme
11             Sinon valeur vaut valeur - 2
12          ferme
13       ferme
14       Sinon affiche "Pfff, une erreur dans ton algorithme lorsque valeur vaut " + valeur
```

Et voici le résultat :

```
Tableau | Audit
Pfff, une erreur dans ton algorithme lorsque valeur vaut 1
```

Dans cet exemple, notre programme plante donc lorsque la variable valeur atteint **1**, car notre condition :

```
Si (valeur / valeur temporaire) > 0, lis
```

…consiste alors à diviser 1 par 0, ce qui est impossible.
Ainsi, l'utilisation du verbe **Essayer** permet de trouver plus facilement l'origine d'une erreur.
De plus, si une erreur survient dans le bloc, le programme ne "plante" pas et peux continuer son exécution normalement.

# Le verbe Déboguer

Reprenons notre exemple précédent, en utilisant le verbe **Déboguer** :

```
[nouveau] *
1  Algorithme farfelu :
2     valeur est un nombre valant 10
3     valeur temporaire est un nombre
4     début
5        Débogue
6           Tant que valeur > 0, lis
7              valeur temporaire vaut valeur - 1
8              Si (valeur / valeur temporaire) > 0, lis
9                 valeur vaut valeur - 1
10             ferme
11             Sinon valeur vaut valeur - 2
12          ferme
```

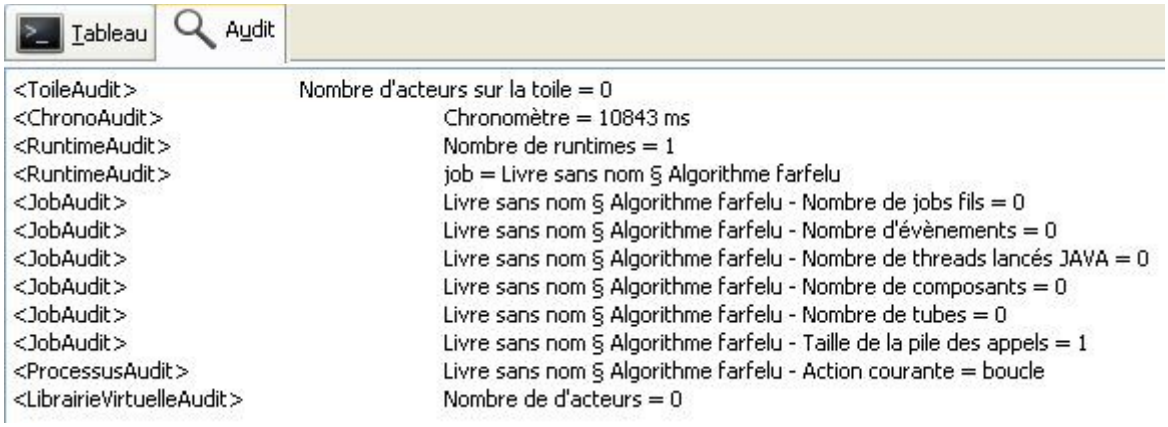
En cliquant sur le bouton **Lire !** de l'atelier, l'interprète détecte alors la présence du verbe **Déboguer** dans le programme et ouvre **l'inspecteur** :


L'atelier propose alors deux boutons pour poursuivre l'exécution du programme :
- **Continuer** : en cliquant sur ce bouton, le programme s'exécute normalement.
- **Pas à pas** : ce bouton permet d'avancer dans l'algorithme, instruction par instruction, et de suivre son évolution dans l'inspecteur afin de déterminer précisément l'origine de l'erreur.

# L'audit

Vous aurez certainement remarqué l'onglet **Audit**, se trouvant au dessus du tableau.
A l'instar de **l'inspecteur** qui permet de suivre le déroulement de votre livre, **l'audit** va afficher encore plus d'informations sur le fonctionnement de votre programme.

Si ce n'est pas déjà fait, recopiez le code précédent dans votre cahier et rendez-vous dans l'onglet **Audit**. Cliquez alors sur le bouton **Activer l'audit** puis sur le bouton **Lire !** de l'atelier.
Voici alors ce qui s'affiche à la place du tableau :




L'audit affiche alors en temps réel bien plus d'informations que l'inspecteur.

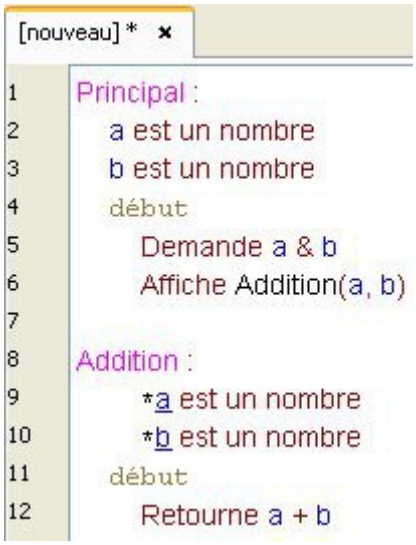
Mais l'onglet **Audit** propose également un autre bouton : **Activer le fichier de trace .linotte/trace.log**

Ce dernier permet d'enregistrer toutes les informations affichées par **l'audit** dans un fichier texte se trouvant à l'adresse suivante : C:\Documents and Settings\"Votre nom d'ordinateur"\.linotte.

Le fichier **trace.log** contenant énormément d'informations, il est également possible d'activer ou de désactiver les traces, afin de sélectionner uniquement celles que l'on veut conserver, comme ceci :

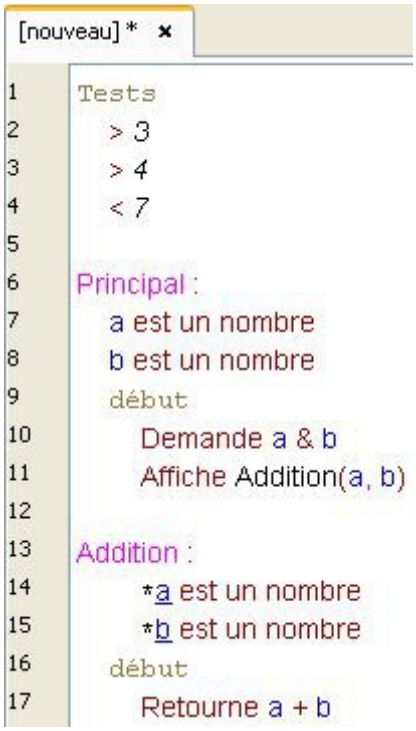
# Les tests

Prenons cet exemple :




Ici, on demande à l'utilisateur d'entrer deux nombres. Puis, on les additionne et on les affiche sur le tableau.
Afin de pouvoir tester si notre programme fonctionne correctement, nous allons utiliser les **tests**.
Ces tests automatiques permettent de simuler les réponses de l'utilisateur.

Pour cela, il suffit d'ajouter un bloc Tests au début du livre, comme ceci :




À la ligne 2, on signale qu'en entrée (le symbole >), le programme recevra le nombre **3**.
Puis, à la ligne 3, le programme recevra le nombre **4**.
À la ligne 4, on signale qu'en sortie (le symbole <), on attendra le nombre **7**.

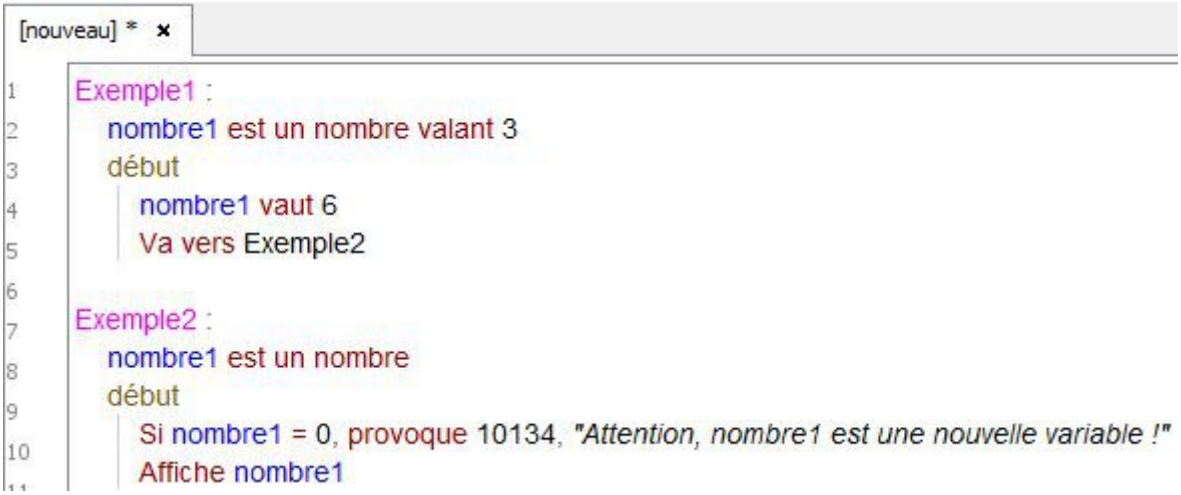
L'atelier détecte alors la présence de tests dans le programme et propose deux boutons pour l'exécuter : **Lire !** ou **Tester !**.
Ainsi, si vous cliquez sur le bouton **Tester !**, voici ce qui s'affiche sur le tableau :


Notre programme fonctionne !

Vous pouvez alors exécuter le programme en cliquant sur le bouton **Lire !** en toute sérénité.

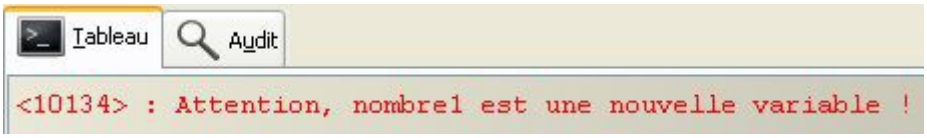
# Le verbe Provoquer

À l'inverse des exemples précédents, le verbe **Provoquer** permet d'arrêter le programme en affichant un numéro d'erreur et un message que vous aurez choisit :




Attention : le numéro de l'erreur doit être supérieur à 10100 pour fonctionner.

Et voici le résultat :



# Les paradigmes de programmation

Intéressons-nous maintenant aux différents paradigmes utilisables en Linotte.

Qu'est-ce qu'un paradigme de programmation ?

Un paradigme est une façon de penser, une manière de voir les choses.
Au cours de ce tutoriel, nous avons constaté qu'il existait plusieurs façons d'écrire son code.
En programmation, un paradigme définit donc notre manière de concevoir son programme.

Le Linotte étant un langage multi-paradigmes, faisons un rapide tour d'horizon des principaux paradigmes de programmation que vous pouvez développer :

## La programmation impérative

Il s'agit du paradigme originel et le plus couramment utilisé.

Le langage impératif comprend l'utilisation de plusieurs types d'instructions principales :
- l'assignation (c'est-à-dire la déclaration d'une variable)
- l'instruction Va vers
- les conditions
- les boucles
- les variables globales

Voici un exemple :

[nouveau] *

```
1  Globale
2     nombre1 est un nombre
3     nombre2 est un nombre
4
5  Principal :
6     début
7        nombre1 vaut 5
8        nombre2 vaut 2
9        Va vers Addition
10
11 Addition :
12    début
13       nombre2 vaut nombre2 + nombre1
14       Va vers Affichage
15
16 Affichage :
17    début
18       Affiche nombre2
```

La programmation impérative est certainement la plus simple à utiliser.
Néanmoins, elle rend la modification de notre programme et la recherche d'erreurs de programmation difficiles.
En effet, considérons que la valeur de notre variable nombre1 soit erronée.
Le résultat affiché dans notre fonction Affichage ne serait donc pas le résultat attendu.
Imaginons alors que notre code contiennent des milliers de lignes, impliquant des centaines de fonctions.
La variable nombre1 étant une variable globale, sa valeur pourrait avoir été modifiée depuis n'importe quelle fonction du livre !
Il faudrait alors analyser notre code en entier et comprendre son fonctionnement dans les moindres détails pour pouvoir trouver l'origine de l'erreur...

# La programmation fonctionnelle

La programmation fonctionnelle permet de décrire son programme comme un emboîtement de fonctions, que l'on peut imbriquer les unes dans les autres.
Ainsi, la programmation fonctionnelle n'utilise pas de variables globales mais permet l'utilisation de variables locales entre plusieurs fonctions :


La fonction **Affichage** est alors imbriquée dans la fonction **Addition**, elle même imbriquée dans la fonction **Principal**.

La programmation fonctionnelle apporte alors :
- les verbes **Parcourir** et **Revenir**

- les paramètres
- la récursivité

Si la valeur de notre variable nombre1 est erronée, comme elle n'est transmise qu'en paramètre, nous pouvons suivre son évolution à travers chaque sous-fonction de notre code.
Ainsi, même si notre livre contient des centaines de fonctions, il n'est pas nécessaire de toutes les analyser pour pouvoir trouver l'origine de l'erreur.
La programmation fonctionnelle diminue ainsi le risque d'erreurs, induits par l'utilisation des variables globales, et facilite la modification du programme.

# La programmation orientée objet

En Programmation Orientée Objet (POO), le programme est considéré comme une collection d'objets, qui interagissent entre eux.
Une des particularités de cette approche est qu'elle permet de regrouper les variables et les fonctions au sein d'un même objet (on parle alors d'**encapsulation**).

La programmation orientée objet apporte ainsi :
- les espèces et les attributs
- les méthodes fonctionnelles
- l'héritage

Les attributs **nombre1** et **nombre2**, ainsi que les fonctions **Addition** et **Affichage**, appartiennent alors à l'objet **test**.

Simulons là encore que la valeur de notre variable nombre1 soit erronée et donc que le résultat affiché s'avère faux.
La variable nombre1 appartenant à l'espèce résultat, il nous suffit alors de regarder le bloc Espèces en haut de notre livre pour avoir la liste de toutes les méthodes fonctionnelles que contient notre espèce (ligne 5).
Car seules les méthodes fonctionnelles de notre objet test, et la fonction Principal où est déclaré notre objet, sont concernées.
Ainsi, même si notre livre contient des centaines de fonctions, celles susceptibles d'héberger notre erreur sont facilement identifiables.
De plus, chaque méthode fonctionnelle étant indépendante, une fois l'erreur localisée, on pourra alors la corriger sans devoir modifier les autres fonctions.
Contrairement à la programmation fonctionnelle qui, de part la structure de son code, imbriquant les fonctions les unes dans les autres, la modification d'une fonction implique souvent la modification de chacune de ses sous-fonctions. Cela peut alors concerner un nombre important de fonctions éparpillées dans le code...

# Les variables particulières disponibles

Voici la liste des variables particulières reconnus par le Linotte :

**auteur** : contient l'auteur de l'interprète Linotte.

**version** : contient la version de l'interprète Linotte.

**spécification** : contient la version des spécifications de l'interprète Linotte.

**ecranv** : taille verticale de l'écran.

**ecranh** : taille horizontale de l'écran.

**polices** : casier contenant les polices disponibles sur le système.

**couleurs** : casier contenant les couleurs reconnues par le Linotte.

**livre** : nom du livre en cours d'utilisation.

**joker** : permet d'afficher le contenu d'une boucle.

| : permet d'effectuer un retour à la ligne.

**vrai** : contient la valeur 1.

**faux** : contient la valeur 0.

**sourisx** : position horizontale de la souris.

**sourisy** : position verticale de la souris.

**touche** : tampon des touches.

**année** : contient l'année en cours.

**mois** : contient le numéro du mois en cours.

**jour** : contient le numéro du jour en cours.

**heure** : contient l'heure en cours (sur 24 heures).

**minute** : contient la minute en cours.

**seconde** : contient la seconde en cours.

**pi** : contient la valeur PI.

**e** : contient la valeur de la constante d'Euler.

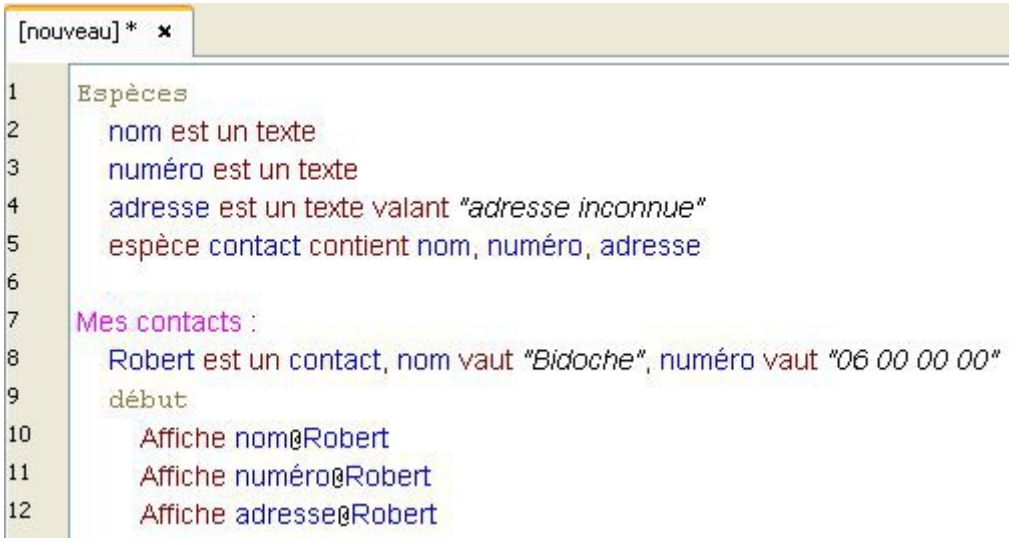
# Les raccourcis disponibles

Le Linotte dispose de raccourcis, visant à augmenter la rapidité d'écriture de notre code.

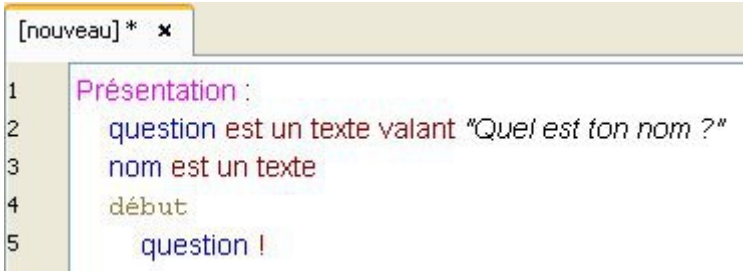
Le raccourci **::** remplace les mots est un :


Le raccourci **<-** remplace le mot valant :


Le raccourci **@** remplace le mot de :

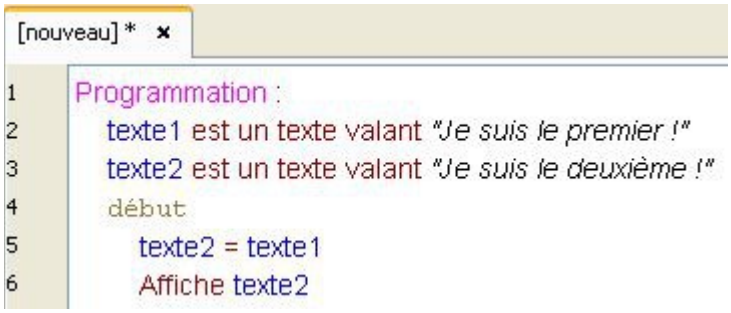



Le raccourci **!** remplace le verbe **Afficher** :



Le raccourci **?** remplace le verbe **Demander** :


Le raccourci **=** remplace le verbe **Valoir** :




Le raccourci # remplace clone() :


Le Linotte offre également plusieurs possibilités d'écriture :

L'utilisation de l'esperluette **&** permet de regrouper plusieurs actions identiques sur une même ligne :

```
[nouveau] *
1  Présentation :
2      nom & prénom est un texte
3      début
4          Demande nom & prénom
5          Affiche nom & prénom
```

L'utilisation du point-virgule **;** permet de regrouper plusieurs actions différentes sur une même ligne :

```
[nouveau] *
1  Présentation :
2      nom est un texte
3      prénom est un texte
4      début
5          Demande nom ; Affiche nom
6          Demande prénom ; Affiche prénom
```

A l'inverse, l'utilisation du symbole **…** permet de découper une ligne :

```
[nouveau] *
1  Casiers :
2      langages est un casier de textes valant "Java", "C++",…
3          "Logo", "Linotte"
4      début
5          Affiche langages
```

**Interpolation de chaîne :**

Grâce à l'utilisation du symbole **${}** autour d'une variable :

On peut remplacer ce code :

```
[nouveau] *
1  Bienvenue :
2      nom est un texte valant "Nicolas"
3      début
4          Affiche "Bonjour " + nom + " !"
```

Par ceci :


**Inférence de types simples :**

Pour les types texte et nombre, l'interprète peut déterminer automatiquement le type :