

LANGAGE DE PROGRAMMATION SECCIA

Auteur : Sylvain Seccia
Copyright © 2001-2009 Sylvain Seccia

SECCIA est un langage orienté objet assisté incluant l'héritage et le polymorphisme. C'est un langage très peu typé (entier, flottant et chaîne de caractères uniquement), sensible à la casse et possédant une syntaxe simple. Les fonctions peuvent être appelées récursivement.

Tout est basé sur l'utilisation d'objets, même pour la gestion des nombres entiers, des nombres à virgule et les chaînes de caractères. Les variables ne sont que des références d'objets. La déclaration de variables se présente sous la forme suivante :

```
File text
```

Ici, **text** est une référence à un objet **File** (pour lire et écrire des fichiers). Écrit de cette sorte, la variable ne fait référence à aucun objet. Elle est donc nulle.

Il est nécessaire de créer un objet et de sauvegarder sa référence dans une variable comme le montre la ligne suivante :

```
text = new File
```

Ainsi la variable **text** pointe bel et bien sur un nouvel objet **File**. On appelle également un objet créé : une instance d'objet.

Si l'on écrit une seconde fois la même instruction à la suite, deux objets **File** seront alors créés au lieu d'un seul mais la variable **text** contiendra la référence du second objet. La référence du premier objet sera perdue, ce qui rend l'accès à l'objet impossible jusqu'à la fin du programme. Plus grave en mode non test, cela causerait des fuites de mémoire.

Il va de soi que si un objet a été créé avec la commande **new**, il doit être également supprimé avec la commande **delete**.

```
delete text  
text = null
```

La première ligne détruit l'objet. Cependant la variable **text** contient toujours la référence de l'objet alors qu'il n'existe plus. Un message d'erreur sera généré si l'on tente d'utiliser cette variable. Pour éviter toute erreur, il est fortement conseillé de remettre la référence à 0, c'est-à-dire nulle. Certes l'accès provoquera toujours une erreur mais l'on saura que la variable ne pointe sur aucun objet.

Cette manipulation n'est pas évidente pour les plus novices mais elle est nécessaire. Cela dit, la création et la destruction des objets qui viennent d'être décrites à l'instant peuvent être gérées automatiquement par le langage. Dans ce cas, il suffit de déclarer la variable à l'aide du mot clé **new** et de ne plus s'en occuper :

```
File text new
```

Cette déclaration rend l'emploi des instructions précédentes inutiles. En fait lorsque la variable est gérée par le langage, sa référence est fixe. L'assignation de référence par le caractère '=' ne peut être appliquée sur ce type de variable contrairement à cette exemple :

```
File a  
File b  
a = new File  
b = a
```

L'assignation n'a pour but que de copier la référence. Il existe deux variables **File** dans ce code mais qu'un seul objet **File**. Et les deux variables peuvent accéder à ce même objet.

Le choix entre ces deux méthodes dépend surtout des objets à créer et de l'application à développer. Il est tout fait envisageable de développer une application entière sans passer par la méthode complexe, tout en laissant au langage la responsabilité des instances.

Avant de poursuivre, il est important de bien comprendre les mots employés. Nous allons donc utiliser maintenant exclusivement les termes suivants :

Objet : pour les noms des types (**File, Integer, String...**)

Variable : pour la déclaration (**text, a, b...**)

Instance : pour les objets créés (**new File**)

Référence : La référence d'une instance contenue dans une variable (c'est une adresse numérique codée sur 4 ou 8 octets en fonction du système d'exploitation (32 bits ou 64 bits).

SOMMAIRE

1. LES OBJETS NOMBRE ET TEXTE.....	4
2. L'UTILISATION D'OBJETS	5
3. LA CREATION D'OBJETS.....	7
4. LES PARAMETRES DE FONCTIONS	10
5. LES VALEURS DE RETOUR.....	11
6. LES CONDITIONS ET BOUCLES DU LANGAGE.....	12
7. LES OPERATEURS	14
8. LA STRUCTURE D'UNE APPLICATION	15
9. LES EVENEMENTS.....	16
10. L'HERITAGE	18
11. LE POLYMORPHISME	21
12. LES CONSTANTES	22
13. LE MOT CLE HIDDEN	25
14. LES SUB-APPLICATIONS.....	26
15. LES EXTENSIONS C++.....	27
16. LES MOTS CLES DE LA COMPILATION	28
17. LA CONVENTION SECCIA.....	29
18. LES MOTS CLES DU LANGAGE	30

1. LES OBJETS NOMBRE ET TEXTE

L'objet **Integer** permet de stocker des nombres entiers signés codés sur 8 octets (64 bits).

L'objet **Float** permet de stocker des nombres flottants codés sur 8 octets (64 bits).

L'objet **String** permet de stocker des chaînes de caractères (1 octet par caractère).

L'objet **UniString** permet de stocker des chaînes de caractères Unicode (2 octets par caractère).

Il existe donc trois types de constantes comme le montre le tableau suivant :

	Décimal	Hexadécimal
Integer	182 255 1000	0xB6 0xff 0x3E8
Float	15.6 -1000.0	
String	"Hello World"	

Un nombre entier peut être écrit sous sa forme hexadécimale. Pour cela, vous devez ajouter le préfixe **0x** en minuscule ou majuscule. Les lettres (A, B, C, D, E, F) peuvent, elles aussi, être écrites en minuscule ou majuscule.

Le langage est capable d'effectuer des conversions automatiques. Voici les cas possibles de conversion :

	Integer	Float	String	UniString
Integer	Integer	Float	String	UniString
Float	Float	Float	String	UniString
String	String	String	String	UniString
UniString	UniString	UniString	UniString	UniString

Remarque : Une division entre deux **Integer** retournera toujours un **Float**.

L'objet **Value** regroupe les trois types sous forme d'un seul objet. Les conversions sont automatiquement gérées par le langage.

Enfin une dernière précision est à noter pour les textes du code : le caractère **%** permet d'insérer n'importe quel caractère ASCII en spécifiant sa valeur hexadécimale alors que le caractère **\$** permet d'insérer la valeur d'un **Integer**, **Float**, **String** ou d'une constante.

```
text = "my name is %22mike%22!" // my name is "mike"!  
  
String name  
name = "mike"  
text = "my name is $name" // my name is mike
```

Cette forme est essentiellement employée pour les requêtes des bases de données et s'avère en l'occurrence très pratique.

Important : Le caractère ``` sera systématiquement remplacé par le caractère `'` pour être compatible avec les requêtes.

```
question = "Are you `mike`?"  
query = "Table.Field=`$question`" // no conflict problem
```

2. L'UTILISATION D'OBJETS

Un objet possède ses propres fonctions, ses propres variables et ses propres constantes. Il existe de nombreux objets dans le logiciel **SECCIA**, en voici une liste non exhaustive :

Integer, Float, String, UniString, Color, Math, Disk, File, Timer, Button, Edit, Slider.

Il vous sera même possible de créer vos propres objets.

En introduction nous avons abordé rapidement la création d'instances en expliquant qu'il existe deux façons différentes pour y parvenir. En voici un exemple illustrant les deux méthodes :

```
File text new
text.OpenFile("c:\myFile.txt", true, true) // write mode
text.Write("Here my data")
text.Close()
```

```
File text
text = new File
text.OpenFile("c:\myFile.txt", true, true) // write mode
text.Write("Here my data")
text.Close()
delete text
```

Pour les objets **Integer**, **Float**, **String** et **UniString** la manipulation est légèrement différente car il n'est pas possible de les instancier en dehors de la déclaration. Pour une raison simple : la variable ne retourne pas la référence de l'instance mais directement sa valeur sous la forme d'un entier, flottant ou chaînes de caractères en fonction de l'objet.

```
Integer a new
Integer b new
a = 10
b = a + 5
b++
```

Nous avons deux variables **Integer** gérées par le langage, donc deux instances **Integer** distinctes.

Remarque : Un **Integer** est toujours initialisé avec la valeur 0.

Nous changeons la valeur de **a**, puis la valeur de **b** en additionnant la valeur de **a** avec 5. Enfin nous ajoutons une unité à la valeur de **b**, c'est-à-dire 10+5+1 soit 16.

Cette syntaxe est très répandue mais une des priorités du langage a été de conserver une logique de base. C'est pourquoi ces trois objets possèdent des fonctions **Set** et **Get** pour modifier et récupérer leur valeur.

```
Integer a new
Integer b new
a.Set(10)
b.Set(a.Get()+5)
b.AddOne()
```

Du fait que les objets **Integer**, **Float** et **String** ne puissent être instanciés, le mot clé **new** devient facultatif dans la déclaration.

```
Integer a
Integer b new
a = 10
b = 20
```

Ainsi le seul moyen d'obtenir la référence de l'instance est d'appeler la fonction **GetObjectRef**.

```
Integer i
Float f
String s
UniString u
i.GetObjectRef()
f.GetObjectRef()
s.GetObjectRef()
u.GetObjectRef()
```

La fonction globale **GetObjectRef** est à utiliser pour tous les autres objets.

```
File file
GetObjectRef(file)
```

Comme leur nom l'indique, les fonctions globales peuvent être accédées depuis n'importe quelle fonction. Cela dit en cas de conflit, si un objet possède une fonction portant le même que la fonction globale, cette dernière devient inaccessible. Pour remédier à ce problème, il est également possible d'inclure le nom de l'objet car il n'existe qu'une seule instance de l'objet **Global**. Il va de même pour l'objet **Application**.

```
File file
Global.GetObjectRef(file)
```

Pour les constantes, si l'objet en contient, il suffit d'écrire :

```
var.NAME_OF_THE_CONSTANT
```

Il va de même pour les variables membres :

```
var.m_age
var.m_age = 18
```

Les mots clés **true** et **false** sont des nombres entiers. Ils ont l'avantage d'être plus explicites que des valeurs 0 et 1 dans les paramètres.

```
var.Show(true)
var.Show(1)
```

```
var.Show(false)
var.Show(0)
```

Enfin le mot clé **null** correspond à la valeur 0 et a pour but d'indiquer qu'une variable ne pointe sur aucune instance.

```
var1 = null
var2.LoadFile(null)
```

3. LA CREATION D'OBJETS

La création de nouveaux types d'objets est un outil extrêmement puissant qui vous permettra de mieux structurer vos applications et de les rendre plus évolutives.

Prenons un cas de figure simple et concret dont nous allons étudier pas à pas son avancement. Nous allons créer un nouvel objet nommé **Box** (Boîte). Concrètement on pourrait le comparer à une boîte en carton de largeur, hauteur et profondeur différentes et personnalisables.

Le mot clé **object** permet de définir un nouvel objet.

```
object Box
{
    constant MINSIZE 2 // min size in cm
    Integer m_width // width in cm
    Integer m_height // height in cm
    Integer m_depth // depth in cm
}
```

Les accolades sont obligatoires pour délimiter le contenu de l'objet. Dans cette structure, seules les constantes et les variables peuvent y figurer. Une constante, comme son nom l'indique, est une valeur fixe. C'est pour cette raison que sa valeur est définie en même temps que la déclaration de la constante. Les constantes sont toujours membres d'un objet. Par convention les constantes sont écrites en majuscule et les variables membres en minuscule précédées par 'm_'. Alors que les noms des objets commencent par une majuscule. Cela évite tout éventuel conflit.

Il est possible d'accéder aux variables membres d'un objet directement ou indirectement via une fonction. Si la lecture directe est plus rapide que l'appel d'une fonction, il est préférable de passer par celle-ci lors d'une quelconque modification d'un membre.

Après avoir défini notre objet et ses variables membres, nous allons définir ses fonctions. Sachez que chaque fonction possède ses propres variables, ce sont des variables locales contrairement aux variables membres (sous entendu membre d'un objet). Une fonction (appelée aussi méthode dans certains langages) est définie par le mot clé **function** :

```
function Box.SetSize(Integer w, Integer h, Integer d)
{
    m_width = w // new width
    m_height = h // new height
    m_depth = d // new depth
}
```

Pour indiquer au langage l'objet parent d'une fonction, il est nécessaire d'inclure le nom de l'objet séparé par un point.

Notre fonction **SetSize** nous sert à changer la taille de la boîte. Elle nécessite trois paramètres de type **Integer** qui seront considérés comme des variables locales.

Important : Ce ne sont que les références des instances qui sont transmises par les paramètres à l'exception des objets **Integer**, **Float**, **String** et **UniString**. Uniquement pour ces derniers, une nouvelle copie d'instance est effectuée et devient locale à la fonction.

En plus des paramètres, une fonction a la possibilité de retourner une donnée. Nous allons définir une nouvelle fonction pour calculer le volume de notre boîte en carton. Elle n'aura besoin d'aucun paramètre mais d'une valeur retournée représentant le volume en cm³ ou en ml.

```
function Integer Box.GetVolume()
{
    return m_width * m_height * m_depth
}
```

La définition est légèrement différente. Comme elle ne contient aucun paramètre, les parenthèses sont vides ce qui reste tout à fait logique. En revanche comme la fonction retourne une donnée, il est indispensable d'indiquer quel est son type avant les paramètres.

Dans le code, nous multiplions la largeur, la hauteur et la profondeur et nous retournons le résultat via le mot clé **return**. Ce mot clé n'a pas forcément besoin d'une valeur, à condition que la définition de la fonction ne spécifie aucun type de retour. L'intérêt d'un tel cas de figure serait de quitter une fonction prématurément.

Important : Lorsque le type de retour défini par la fonction est un **Integer**, la valeur fournie est automatiquement convertie en un nombre entier. Pour l'objet **Float**, elle est convertie en un nombre flottant et pour les objets **String/UniString**, en une chaîne de caractères. En revanche pour les autres, seule la référence de l'instance de l'objet est retournée.

Attention : Il ne faut jamais retourner la référence d'une variable locale gérée automatiquement par le langage. Car l'instance créée à l'appel de la fonction sera automatiquement détruite et la fonction retournera une référence d'une instance inexistante.

```
function Color MyObj.Get()
{
    Color color new
    return color           // ERROR!!!

    Color color
    color = new Color
    return color           // OK
}
```

Pour finir sur ce sujet, notez qu'il existe deux fonctions réservées. Lorsque vous créez une nouvelle instance d'un objet, vous créez également ses variables membres. Il est souvent utile d'initialiser certaines valeurs.

```
function Box.Constructor()
{
    m_width = 10
    m_height = 10
    m_depth = 10
}
function Box.Destructor()
{
}
```

Ces deux fonctions sont bien évidemment appelées automatiquement par le langage lors de la création et de la destruction d'une instance. Elles ne possèdent aucun paramètre et aucune valeur de retour. Elles sont facultatives et ne peuvent être appelées par le programmeur.

Nous n'avons pas utilisé la constante **MINSIZE** volontairement afin d'alléger le code. Cette constante pourrait nous servir pour vérifier si la taille de la boîte est conforme aux caractéristiques souhaitées.

Il est temps de créer notre première instance de la même façon qu'un objet **Integer**.

```
Integer volume new
Box myBox new
myBox.SetSize(20, 5, 10)
volume = myBox.GetVolume()
```

Nous changeons la taille de la boîte **myBox** avec les valeurs : 20 cm, 5 cm et 10 cm. Ensuite nous calculons le volume et nous stockons le résultat dans la variable **volume**, c'est-à-dire 1000 cm³ ou 1000 ml ou bien encore 1 litre.

Remarque : La déclaration des variables est normalement située avant le code.

Une dernière précision sur ce chapitre : pour identifier l'instance actuelle de l'objet dans une fonction, il suffit d'utiliser le mot clé **this**. Ce mot clé retourne l'instance qui a appelé la fonction. Elle est forcément de même type que l'objet. Un de ses intérêts est de pouvoir transmettre la référence de l'instance par l'intermédiaire d'un paramètre d'une fonction.

```
function MyObject.MyFirstFunction()  
{  
    MySecondFunction(this)    // transmit instance  
}
```

Au premier abord vous pourriez penser que ce code n'a pas beaucoup d'intérêt car la fonction **MySecondFunction** est également une fonction de l'objet **MyObject**, et peut donc elle-même utiliser le mot clé **this**. Dans la pratique, la seconde fonction pourrait être une fonction d'un autre objet comme le montre le code suivant :

```
function MyFirstObject.MyFunction(MySecondObject obj)  
{  
    obj.MyFunction(this)  
}
```

Ou l'objet peut tout à fait être le même, mais d'une instance différente :

```
function MyObject.MyFunction(MyObject obj)  
{  
    obj.MyFunction(this)  
}
```

4. LES PARAMETRES DE FONCTIONS

Nous avons déjà parlé des paramètres, et pourtant nous n'avons pas encore fait le tour sur ce sujet. Vous le savez, les fonctions et les événements peuvent contenir des paramètres entre parenthèses. Lorsqu'un de ces éléments doit être appelé, il est nécessaire de fournir une valeur pour chaque paramètre. Or cette tâche peut devenir fastidieuse si les paramètres sont nombreux.

Il est donc possible de rendre certains paramètres optionnels. Dans ce cas, c'est la valeur définie par défaut qui sera utilisée.

```
function Obj.Show(Integer visible = true)
{
}
```

```
function Obj.Set(Integer a = 5, String s = "", Float f = 1.5)
{
}
```

```
function Obj.Load(File file = null)
{
}
```

```
function Obj.SaveImage(Integer format = Image.BMP)
{
}
```

```
Show(false)           // the default value is ignored
Set(10)                // only the first value is ignored
Load()                 // file has no object
SaveImage()            // use the default value
SaveImage(Image.BMP)  // the default value is ignored
```

Seules les constantes et les mots clés suivants sont autorisés : **null**, **true** et **false**.

Lorsqu'un paramètre possède une valeur par défaut, tous les autres paramètres situant à droite de celui-ci doivent être obligatoirement des paramètres optionnels. C'est à dire qu'ils doivent, eux aussi, posséder des valeurs par défaut.

Le code suivant est erroné car **z** n'est pas un paramètre optionnel :

```
function Obj.Set(Integer x, Integer y = 0, Integer z) // ERROR!!!
{
}
```

5. LES VALEURS DE RETOUR

Vous avez peut être déjà remarqué qu'il n'est pas possible de retourner la référence d'un **Integer**, **Float**, **String** ou **UniString**.

```
function Integer Obj.Get()
{
    Integer day new
    day = 25
    return day
}
```

Premièrement l'instance de l'objet **Integer** est locale à la fonction et sera détruite à la sortie. Deuxièmement la dernière ligne ne retourne pas la référence de l'instance mais sa valeur (25).

Utilisez plutôt l'objet **Value** de cette façon :

```
function Value Obj.Get()
{
    Value day
    day = new Value
    day.SetInt(25)
    return day
}
```

6. LES CONDITIONS ET BOUCLES DU LANGAGE

Vous avez déjà pu observer une partie de la syntaxe du langage mais il reste encore les conditions et boucles entre autre.

```
if [expression]
    ...
else
    ...
end

if [expression]
    ...
end

if [expression] or [expression] and [expression]
    ...
end

if [expression] || [expression] && [expression]
    ...
end

if [expression] and ([expression] or [expression])
    ...
end
```

```
while [expression]
    ...
end

while [expression]
    ...
    continue           // to go to the next loop
                       // without reading the following code
    ...
end

while [expression]
    ...
    break              // to quit
                       // without reading the following code
    ...
end
```

```
for [init expression] ; [condition expression] ; [increment expression]
    ...
    continue
    ...
    break
    ...
end
```

```
switch [expression]
    case [constant]
        ...
        break
    case 2
        ...
        break
    case 0x05
    case 10.6
    case "text"
        ...
```

```
        break
    default
        ...
        break
end
```

S'il existe plusieurs **case** spécifiant la même valeur, seul le premier **case** est pris en compte.

7. LES OPERATEURS

Nous connaissons tous, les quatre opérateurs de base pour effectuer des calculs ('+', '-', '/', '*'). Les calculs plus complexes sont effectués par l'intermédiaire de l'objet **Math**.

Remarque : Pour les chaînes de caractères, le signe '+' permet de lier deux textes pour n'en former plus qu'un.

Le modulo est représenté par le caractère '%', également disponible depuis l'objet **Math** sous forme de fonction.

Signe	Définition	Exemple
+	effectue une addition	a + b
-	effectue une soustraction	a - b
*	effectue une multiplication	a * b
/	effectue une division	a / b
%	retourne le reste de la division	a % b

En programmation, des opérateurs permettent de tester deux expressions entre elles. Ce sont les opérateurs d'égalité :

Signe	Définition	Exemple
==	est égal à	a == b
!=	est différent de	a != b
>	est plus grand que	a > b
<	est plus petit que	a < b
>=	est plus grand ou égal à	a >= b
<=	est plus petit ou égal à	a <= b
?=	est similaire à (voir String.IsSimilar)	a ?= b

SECCIA fournit également des opérateurs bit à bit pour les nombres entiers.

Signe	Définition	Exemple
	opérateur OU inclusif (OR)	a b
&	opérateur ET (AND)	a & b
^	opérateur OU exclusif (XOR)	a ^ b
<<	décale les bits à gauche	a << b
>>	décale les bits à droite	a >> b

Les opérateurs unaires sont les suivants :

Signe	Définition	Exemple
~	inverse tous les bits	~a
!	négation logique	!a
++	ajoute immédiatement 1 à un Integer/Float	a++
--	enlève immédiatement 1 à un Integer/Float	a--

Les opérateurs d'affection sont les suivants :

Signe	Définition	Exemple
+=	effectue une addition	a += 2
-=	effectue une soustraction	a -= b
*=	effectue une multiplication	a *= b
/=	effectue une division	a /= 2
%=	retourne le reste de la division	a %= 10

Les opérateurs bit à bit suivent les règles suivantes :

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

8. LA STRUCTURE D'UNE APPLICATION

Une application n'est ni plus ni moins un objet, nommé **Application**. Pour cela vous devez créer votre propre objet et le nommer **Application** pour qu'il soit considéré comme objet principal.

```
object Application
{
}
function Application.Constructor()
{
}
function Application.Destructor()
{
}
```

Comme vous l'aurez compris, **Constructor** sera la première fonction appelée de votre application. **SECCIA** se charge de créer une instance de l'objet **Application**.

Contrairement aux objets classiques, l'objet **Application** peut comporter en plus de ces deux fonctions, une troisième fonction.

```
function Application.Constructor()
{
    EnableApplicationLoop(true)
}
function Application.Loop()
{
}
```

La fonction **Loop** est uniquement appelée si l'application a activé la méthode de boucle infinie à l'aide de la fonction **EnableApplicationLoop**. Par défaut, cette méthode n'est pas activée. Lorsque le programme est en mode boucle, l'application consomme continuellement les ressources de l'ordinateur. Cela correspond à une boucle **while** interne.

Il n'est donc pas nécessaire et fortement déconseillé d'activer cette méthode si vous n'en avez pas l'utilité. En revanche elle est fort utile pour développer, par exemple, un jeu qui nécessite une boucle infinie.

*Remarque : L'objet **DxEngine** en mode "auto refresh" nécessite l'activation du mode boucle.*

9. LES EVENEMENTS

La programmation **Windows**[®] est basée sur le principe d'événements et de messages en fil d'attente. Ce système est très utilisé pour les contrôles. Lorsqu'un utilisateur d'une application clique sur un bouton, l'application doit être capable d'intercepter le message du clic.

Dans le logiciel **SECCIA**, toujours par convention, les événements sont précédés par 'On'. Le bouton aura donc un événement **OnClick**.

Il faut différencier les événements et les messages. Les événements sont situés à l'intérieur des objets comme pour les fonctions. Alors que les messages sont définis dans d'autres objets, car ils servent à intercepter les événements.

La définition d'un événement est similaire à la définition d'une fonction. Un événement peut contenir des paramètres et une valeur de retour. La seule différence réside dans le mot clé **event**.

```
event Box.OnChange( )
{
}
```

Cela ne suffit pas, nous devons appeler cet événement dans notre objet **Box** sinon il ne sera jamais appelé. Contrairement aux fonctions, il est totalement impossible d'appeler un événement à l'extérieur de l'objet. C'est l'objet qui doit appeler lui-même ses événements.

Important : Les événements ne contiennent généralement aucun code à l'exception d'un **return** si la définition possède un type de retour.

Reprenons notre exemple de départ en ajoutant un appel d'événement lorsque les dimensions de la boîte changent de valeurs. Nous avons écrit précédemment une fonction pour changer les dimensions, nous allons donc maintenant ajouter l'appel dans cette dernière.

```
function Box.SetSize(Integer w, Integer h, Integer d)
{
    m_width = w
    m_height = h
    m_depth = d
    OnChange()           // event call
}
```

Pour l'instant nous nous concentrons uniquement sur l'objet en question mais n'oubliez pas que les événements ne sont pas utiles pour l'objet lui-même mais pour l'objet parent.

Lorsque l'on clique sur un bouton, ce n'est pas le bouton lui-même qui doit recevoir le message mais l'objet qui a créé le bouton. C'est pourquoi les événements restent vides et qu'il n'y a aucun message dans notre objet.

Revenons dans l'objet **Application** et ajoutons l'événement de notre objet **Box** sous la forme d'un message.

```
object Application
{
    Box m_box new
}
function Application.Constructor()
{
    m_box.SetSize(20, 5, 10)    // width, height and depth
}
message Application.Box.OnChange( )
{
}
```

A la différence des fonctions et des événements, les messages ont besoin d'une autre donnée : le nom de l'objet où se trouve l'événement.

A présent si l'on devait résumer : il existe un événement **OnChange** dans l'objet **Box** et un message **OnChange** dans l'objet **Application**. La question qui se pose naturellement, est de savoir lequel des deux est appelé. L'événement ou le message ? Cela dépend de l'instance et non de l'objet. Comme **m_box** est créé dans l'objet **Application**, c'est donc l'objet **Application** qui va recevoir le message. L'événement **OnChange** de **m_box** ne sera jamais appelé. En revanche si le message **OnChange** n'est pas défini dans l'objet **Application**, alors cette fois-ci l'événement **OnChange** de **m_box** sera appelé.

Nous avons parlé seulement des variables gérées par le langage à l'aide du mot clé **new**. En ce qui concerne la gestion manuelle des instances, un nouveau mot clé fait son apparition : **intercept**.

```
object Application
{
    Box m_box
}
function Application.Constructor()
{
    m_box = new Box
    intercept m_box
}
function Application.Destructor()
{
    delete m_box
}
```

Le mot clé **intercept** permet d'intercepter tous les événements d'une instance. Si l'on retire cette ligne, l'instance **Box** n'enverra aucun message à l'objet **Application**. Le mot clé ne peut pas être appliqué aux variables gérées par le langage, car dans ce cas les interceptions sont également gérées automatiquement.

Un point reste à clarifier concernant les messages. Lors de la définition d'un message, nous indiquons le nom de l'objet contenant l'événement. Or si nous créons deux instances **Box**, il est difficile de savoir de quelle instance provient le message. Evidemment le langage fournit un mot clé spécial pour identifier l'appelant : **caller**.

```
message Application.Box.OnChange()
{
    Integer volume new
    volume = caller.GetVolume()
}
```

Le mot clé **caller** contient la référence de l'instance qui a envoyé le message et ne peut donc être employé que dans les messages.

10. L'HERITAGE

L'héritage est un concept de programmation orientée objet plus complexe. Il permet de créer un objet dérivé d'un autre objet. Ainsi l'objet dérivé possède un objet base, et hérite de tous les membres de celui-ci (fonctions, variables et constantes).

Important : Ni les événements ni les messages ne sont hérités. De même pour les fonctions **Constructor** et **Destructor**.

Toujours grâce à notre exemple mais en repartant de zéro, vous allez mieux vous rendre compte de l'intérêt de l'héritage. Nous avons un objet **Box** qui représente une boîte en carton. Nous souhaitons à présent ranger dans cette boîte, deux types de jouets : des balles et des cubes en plastique. Ces balles et ces cubes ne sont pas tous de la même taille. Il existe un point commun entre ces deux objets, ce sont des jouets qui seront rangés dans la boîte en carton. Nous pouvons donc créer une base **Toy** et deux objets dérivés **Ball** et **Cube**.

En programmation, cela donne le code suivant :

```
object Box
{
    Array m_toys new      // array containing all toys
}
object Toy
{
    Integer m_weight new // weight in grammes
}
object Ball from Toy
{
    Integer m_radius new // radius in cm
}
object Cube from Toy
{
    Integer m_size new    // width in cm
}
```

Nous avons bien quatre objets différents dont deux, dérivés de l'objet **Toy**. Le mot clé **from** permet d'indiquer au langage que c'est un objet dérivé.

Vous constaterez que l'objet **Toy** possède un **Integer** dont héritent les objets **Ball** et **Cube**. Cette variable représente le poids du jouet. Cela évite de devoir créer deux variables : une dans **Ball** et une autre dans **Cube**.

Pour simplifier, nous avons enlevé les dimensions de l'objet **Box**.

Important : Si un objet dérivé possède une variable membre déjà présente dans l'objet base, alors deux variables distinctes portant le même nom coexisteront. Il sera en revanche impossible d'accéder de l'extérieur à la variable de la base via l'objet dérivé.

Un nouveau type d'objet dont on n'a jamais parlé fait son apparition dans **Box**. Il s'agit de **Array**, un objet pour gérer les tableaux. Les jouets qui figureront dans ce tableau seront les jouets présents dans la boîte en carton.

Important : Les tableaux stockent uniquement les références des instances. Il n'est pas possible d'utiliser les objets **Integer**, **Float**, **String** et **UniString** car ces objets ne retournent pas de références. C'est pourquoi il est nécessaire de faire appel aux objets **IntegerArray**, **FloatArray**, **StringArray** et **UniStringArray**.

L'étape suivante consiste à définir les fonctions dont nous avons besoin :

- **Toy** : **SetWeight**
- **Ball** : **SetSize**, **GetVolume**
- **Cube** : **SetSize**, **GetVolume**
- **Box** : **AddToy**, **GetToyWeight**

La fonction **AddToy** ajoutera un jouet dans la boîte en carton.

```
function Toy.SetWeight(Integer weight)
{
    m_weight = weight
}
```

```
function Ball.SetSize(Integer radius)
{
    m_radius = radius
}
function Integer Ball.GetVolume()
{
    return m_radius * m_radius * m_radius * PI * 4/3
}
```

```
function Cube.SetSize(Integer size)
{
    m_size = size
}
function Integer Cube.GetVolume()
{
    return m_size * m_size * m_size
}
```

```
function Box.AddToy(Toy toy)
{
    m_toys.Add(toy)
}
function Box.Destructor()
{
    m_toys.DeleteAll()
}
```

La fonction **GetToyWeight** nous retournera le poids total des jouets présents dans la boîte en carton. Elle nécessite un code plus important car nous devons faire une boucle pour récupérer le poids de chaque jouet.

```
function Integer Box.GetToyWeight()
{
    Integer i new
    Integer weight new
    Toy toy

    weight = 0
    for i=0 ; i<m_toys.GetCount() ; i++
        toy = m_toys.Get(i)
        weight.Add(toy.m_weight)
    end
    return weight
}
```

La fonction **Get** de l'objet **Array** retourne la référence de l'instance stockée à l'index spécifié.

Dans le code de l'application, nous allons créer deux jouets : une balle et un cube. La fonction **AddToy** est définie par un paramètre **Toy**. Cela ne pose aucun problème car un objet **Ball** est aussi un objet **Toy**.

```
Ball myBall
myBall = new Ball
myBall.SetWeight(50)
myBall.SetSize(2)

Cube myCube
myCube = new Cube
myCube.SetWeight(100)
```

```
myCube.SetSize(3)

Box myBox new
myBox.AddToy(myBall)
myBox.AddToy(myCube)

Integer weight new
weight = myBox.GetToyWeight()
```

Le poids total des jouets sera de 150 grammes (50+100).

L'héritage peut encore aller plus loin. Le langage est capable de gérer des objets dérivés sur plusieurs niveaux. Rien ne nous empêche de définir un autre objet, dérivé de l'objet **Ball**. Ce nouvel objet héritera à la fois des membres de l'objet **Ball**, mais également de l'objet **Toy**.

Profitions-en pour comprendre comment est-il possible d'accéder à une variable ou une constante définie à deux endroits. Pour cela, ajoutons un objet **Truck** (Camion) dérivé de l'objet **Toy** avec une variable **m_weight**.

```
object Truck from Toy
{
    Integer m_weight new
}
function Truck.Constructor()
{
    m_weight          // access to the variable of the Truck object
    Toy.m_weight      // access to the variable of the Toy object
}
```

Pour accéder à la variable **m_weight** de l'objet **Toy**, il est nécessaire d'ajouter son nom suivi d'un point. Pour une raison simple : la version **m_weight** de l'objet **Truck** cache la version de l'objet **Toy**.

11. LE POLYMORPHISME

Le polymorphisme est la continuité de l'héritage pour apporter à vos applications encore plus de souplesse et de puissance grâce à une programmation plus générique. Le but du polymorphisme d'héritage est de pouvoir appeler une fonction d'un objet sans se soucier de son type intrinsèque.

Nous avons vu qu'il était possible d'avoir un variable membre déjà présente dans une base. Dans ce cas, deux variables coexistent. Or le résultat n'est pas le même pour les fonctions, car une fonction déjà présente dans la base devient une fonction virtuelle si elle présente également dans l'objet dérivé.

Continuons dans notre démarche en ajoutant une nouvelle fonction dans l'objet **Toy**.

```
function Integer Toy.GetVolume()  
{  
    return 0  
}
```

Désormais la fonction **GetVolume** existe à la fois dans l'objet de base **Toy** et à la fois dans les objets dérivés **Ball** et **Cube**.

Examinons le code suivant :

```
Toy myToy1 new // variable containing a Toy instance  
Ball myBall new // variable containing a Ball instance  
myToy1.GetVolume() // call the function of the Toy object  
myBall.GetVolume() // call the function of the Ball object  
  
Toy myToy2 // Toy variable without instance  
myToy2 = myBall // copy the reference  
myToy2.GetVolume() // call the function of the Ball object
```

On pourrait croire que la variable **myToy2** contient la référence d'une instance d'un objet **Toy**, alors qu'elle contient en réalité la référence d'une instance d'un objet **Ball**. De ce fait, la fonction virtuelle **GetVolume** est liée avec la fonction **GetVolume** de l'objet **Ball** et non avec l'objet de base **Toy**.

Remarque : Avec l'héritage il n'y a plus de certitude concernant le type de l'instance. Il faut donc différencier le type de la variable et le type de l'instance.

Une autre question se pose lorsque l'on se trouve dans une fonction virtuelle. Est-il possible d'appeler statiquement une fonction située dans une base ? Bien sûr. Sinon le code suivant poserait un sérieux problème.

```
function Integer Derived.GetAge()  
{  
    GetAge()  
}
```

Effectivement, la fonction **GetAge** s'appelle elle-même indéfiniment et nous ne pouvons jamais appeler la fonction **GetAge** de l'objet de base.

Pour avoir accès uniquement à la fonction de l'objet de base, il suffit de rajouter son nom. L'instance de l'objet concerné est l'instance actuelle.

```
function Integer Derived.GetAge()  
{  
    Base.GetAge()  
}
```

On peut ainsi remonter de base en base jusqu'à la base principale si l'on souhaite que toutes les fonctions soient appelées.

12. LES CONSTANTES

L'utilisation des constantes est essentielle dans un programme. Ne laissez jamais des valeurs dans vos instructions :

```
dayCount = curMonth * 30 + curDay
```

Nous calculons le nombre de jours d'une date sur une base de 30 jours par mois. Le nombre 30 n'est pas très explicite. Une constante serait beaucoup plus adaptée.

```
constant DAYSPERMONTH 30
dayCount = curMonth * DAYSPERMONTH + curDay
```

Cela facilite également un éventuel changement de valeur par la suite.

Grâce à l'hexadécimal, les constantes apportent une fonctionnalité supplémentaire très intéressante. Prenons un cas précis, totalement nouveau. Nous voulons créer un objet **Pizza** qui contiendrait des informations au sujet des ingrédients. Nous pourrions présenter l'objet de cette façon :

```
object Pizza
{
  Integer m_cheese new           // true, false
  Integer m_tomato new          // true, false
  Integer m_anchovy new         // true, false
}
```

Chaque variable aurait soit la valeur **false** soit la valeur **true** et serait éventuellement éditable par l'intermédiaire de trois fonctions. Plus trois autres fonctions pour la lecture.

```
function Pizza.SetCheese(Integer has)
{
  m_cheese = has!=0           // convert integer to bool (true or false)
}
function Integer Pizza.GetCheese()
{
  return m_cheese
}
```

Si notre pizza contient les trois ingrédients, il faudrait écrire le code suivant.

```
Pizza pizza new
pizza.SetCheese(true)
pizza.SetTomato(true)
pizza.SetAnchovy(true)
```

Ensuite il faut pouvoir tester les ingrédients. Admettons que nous voulions savoir si notre pizza contient des anchois. Dans ce cas, l'opération est simple.

```
if pizza.GetAnchovy()==true
end
```

Pour savoir si notre pizza contient les trois ingrédients, trois conditions s'imposent.

```
if pizza.GetCheese() && pizza.GetTomato() && pizza.GetAnchovy()
end
```

Prenons maintenant en compte les constantes et recommençons notre exemple. Au lieu d'avoir trois variables différentes pour chaque ingrédient, nous n'en avons plus qu'une seule.

```
object Pizza
```

```

{
    constant CHEESE 0x01
    constant TOMATO 0x02
    constant ANCHOVY 0x04
    Integer m_type new
}

```

La quatrième constante aurait la valeur 0x08, la cinquième 0x10, la sixième 0x20 et ainsi de suite...

```

function Pizza.Set(Integer type)
{
    m_type = type
}
function Integer Pizza.Get()
{
    return m_type
}

```

Avec les constantes, la pizza aux trois ingrédients nécessite les instructions suivantes en reprenant le code de notre exemple précédent.

```

Pizza pizza new
pizza.Set(Pizza.CHEESE | Pizza.TOMATO | Pizza.ANCHOVY)

if pizza.Get() & Pizza.ANCHOVY
end

if pizza.Get() & (Pizza.CHEESE | Pizza.TOMATO | Pizza.ANCHOVY)
end

```

Pour simplifier d'avantage, nous pouvons rajouter deux constantes définissant la spécialité de la pizza en cumulant les ingrédients.

```

object Pizza
{
    constant CHEESE          0x01
    constant TOMATO          0x02
    constant ANCHOVY         0x04
    constant MARGUERITE      CHEESE | TOMATO
    constant NAPOLITAN      MARGUERITE | ANCHOVY
    Integer m_type new
}

```

Ainsi pour savoir si la pizza est une napolitaine, il suffit de tester la variable `m_type`.

```

if pizza.Get() == Pizza.NAPOLITAN
end

```

Notez la subtilité avec l'instruction suivante qui ne permet pas de savoir si la pizza est belle et bien une napolitaine mais de savoir si la pizza contient tous les ingrédients de la napolitaine avec éventuellement des ingrédients supplémentaires.

```

if pizza.Get() & Pizza.NAPOLITAN
end

```

Enfin comme les constantes sont utilisables dans les `switch`, il devient alors possible de tester les spécialités de cette sorte :

```

switch pizza.Get()
case Pizza.MARGUERITE
    break
case Pizza.NAPOLITAN

```

end

break

13. LE MOT CLE HIDDEN

L'éditeur de code est capable de lister automatiquement les membres d'un objet dans un menu déroulant lors de l'écriture de vos instructions. Ce qui facilite et rend le développement plus agréable et plus convivial.

Cet éditeur affiche à la fois les membres de l'objet en question mais également les membres de ses objets bases. Ainsi pour mieux les différencier, une icône différente est représentée à gauche du nom.

Uniquement depuis l'extérieur, il est possible de ne pas inclure certaines variables, constantes et fonctions dans cette liste en spécifiant le mot clé **hidden** dans la déclaration.

```
object MyObject
{
    constant PUBLIC          1
    constant PRIVATE        2    hidden
    Disk m_public1
    Disk m_public2 new
    Disk m_private1 hidden
    Disk m_private2 new hidden
    Disk m_private3 hidden new
}
```

```
function MyObject.MyPrivateFunction() hidden
{
}
```

Important : Malgré que les éléments soient cachés, ils resteront toujours accessibles ; excepté pour l'objet **SubApplication** où seuls les constantes, fonctions et événements non cachés sont exportés.

14. LES SUB-APPLICATIONS

Une application peut également être compilée sous la forme d'une Sub-Application : un fichier DLL portant l'extension **.sub**. De même pour le fichier exécutable, la DLL contient les ressources et le code source de l'application à l'exception de l'objet **Application** qui n'est jamais inclus.

Les Sub-Applications sont indépendantes et réutilisables dans d'autres applications, comme une extension C++.

Pour utiliser une Sub-Application au sein du logiciel, il est d'abord nécessaire de copier le fichier en question dans le dossier **SubApplications** avant de démarrer **SECCIA**. La Sub-Application sera ainsi accessible dans la catégorie **Plugins** de la liste des objets.

Quelques restrictions sont à prendre en compte par rapport aux autres objets. Leur nom commence toujours par le préfixe **Sub_** pour les différencier. Les fonctions et événements à exporter ne peuvent utiliser que des objets **Integer**, **Float**, **String** et **UniString**. Car il n'est pas possible d'accéder aux objets de l'application, ni en écriture ni en lecture. L'héritage n'est pas autorisé. Enfin, une Sub-Application ne peut pas être un contrôle accessible depuis l'éditeur de boîtes de dialogue.

Ainsi seules les fonctions et les événements de l'objet **Sub-Application** seront exportés à condition qu'ils respectent les règles suivantes :

- Les fonctions doivent être déclarées sans le mot **hidden**
- Les paramètres doivent être de type **Integer**, **Float**, **String** ou **UniString**.
- La valeur de retour, si présente, doit être de type **Integer**, **Float**, **String** ou **UniString**.

15. LES EXTENSIONS C++

Le SDK (*Software Development Kit*) permet de développer des modules externes indépendants à l'aide du langage C++. Ces extensions sont des fichiers DLL conçus exclusivement pour s'intégrer au logiciel et à vos applications comme de simples objets. L'avantage des extensions est de pouvoir apporter de nouvelles fonctionnalités à l'environnement en distribuant ou non le code source en C++.

Pour utiliser une extension au sein du logiciel, il est d'abord nécessaire de copier le fichier en question dans le dossier **Extensions** avant de démarrer **SECCIA**. L'extension sera ainsi accessible dans la catégorie **Plugins** de la liste des objets.

Quelques restrictions sont à prendre en compte par rapport aux autres objets. Leur nom commence toujours par le préfixe **Ext_** pour les différencier. Les fonctions et événements des extensions ne peuvent utiliser que des objets **Integer**, **Float**, **String** et **UniString**. Car il n'est pas possible d'accéder aux objets de l'application, ni en écriture ni en lecture. L'héritage n'est pas autorisé.

Contrairement aux Sub-Applications, les extensions peuvent être des contrôles accessibles depuis l'éditeur de boîtes de dialogue. Dans ce cas précis, l'extension est dérivée de l'objet **Control**.

Pour plus d'information, voir les fichiers fournis avec le SDK.

16. LES MOTS CLES DE LA COMPILATION

Lors de la construction du fichier exécutable ou lors du test de l'application au sein de l'interface **SECCIA**, le langage est capable d'exclure des lignes de code à l'aide de mots clés particuliers.

`_ifdef, _ifndef, _else, _end`

Ces mots clés ne font pas partie du code de votre application au moment de son exécution.

```
object Application
{
}
function Application.Constructor()
{
#ifdef VERSION_DEMO
    Demo()
#endif

#ifdef VERSION_DEMO
function Application.Demo()
{
    MessageBox("DEMO")
}
#endif
}
```

Si **VERSION_DEMO** a été préalablement défini, le code sera intégré au programme. Dans le cas contraire la fonction **Demo** n'existera pas et ne pourra donc pas être appelée. C'est pourquoi il est nécessaire d'exclure également l'appel de la fonction dans **Constructor**.

Le résultat final dépendra donc des définitions, dans notre cas précis il peut y avoir deux cas possibles :

```
object Application
{
}
function Application.Constructor()
{
}
```

```
object Application
{
}
function Application.Constructor()
{
    Demo()
}
function Application.Demo()
{
    MessageBox("DEMO")
}
```

17. LA CONVENTION SECCIA

Le langage est sensible à la casse, c'est-à-dire qu'il fait la distinction entre les majuscules et les minuscules. Cette caractéristique est essentielle pour obtenir un code performant à condition de respecter une certaine rigueur. Soyons clair, il n'est pas question ici d'appeler deux variables locales d'une fonction à l'aide du même nom. Cela deviendrait une catastrophe pour la compréhension du code. Les variables doivent porter des noms très explicites. En revanche, on s'en servira pour distinguer les différents types d'éléments.

Remarque : Vous êtes libre de nommer vos variables comme bon vous semble, ce ne sont que des conseils pratiques qui vous aideront à mieux structurer votre code.

Commençons par lister tous les types d'éléments existants : les constantes, les objets, les fonctions, les variables membres et les variables locales. Les mots en bleu et en gris sont les mots clés du langage, ils sont réservés.

Dans la plupart des langages, les constantes sont en majuscule. Nous utiliserons la même convention.

```
constant DIFFICULTY_EASY 0
constant DIFFICULTY_HARD 1
```

Les objets **SECCIA** commencent tous par une majuscule, nous ferons de même pour nos propres objets.

```
object Integer
object Box
object SuperBox
```

Les fonctions, les événements et les messages ont toujours des parenthèses, il n'y a donc aucune restriction à nommer une fonction : **Integer**. Cependant, il est plus clair de nommer une fonction en commençant par une majuscule comme pour les objets et d'éviter de choisir un nom d'objet.

```
function Box.Select()
function Box.RemoveColor()
```

Pour identifier plus rapidement un événement, nous adopterons le préfixe 'On'.

```
event Box.OnOpen()
```

Une variable locale peut avoir le même nom qu'une variable membre sans provoquer de conflit. La variable membre devient alors inaccessible sans l'utilisation de fonctions. Une convention très employée consiste à écrire 'm_' devant chaque variable membre.

```
object Box
{
    Integer m_toyCount           // member
}
function Box.Constructor()
{
    Integer toyCount           // local variable
}
```

18. LES MOTS CLES DU LANGAGE

La liste complète des mots clés du langage :

object
function
event
message

_ifdef
_ifndef
_else
_end

and
bool
break
caller
case
constant
continue
default
delete
else
end
false
for
from
hidden
if
intercept
new
null
or
return
switch
this
true
while