

# THE SECCIA PROGRAMMING LANGUAGE

Author: Sylvain Seccia  
Copyright © 2001-2009 Sylvain Seccia

**SECCIA** is an assisted object-oriented programming language that includes inheritance and polymorphism. The language has a very light typology (integer, floating numbers and character strings only), is case-sensitive and endowed with a simple syntax. Functions can be called recursively.

In **SECCIA**, everything is based on the use of objects, including the management of integers, floating numbers and strings. Variables are references to objects. Declaring variables takes the following shape:

```
File text
```

Here, **text** is a reference to a **File** object (which reads from and writes to files). In this form, the variable does not reference an object: it is, therefore, a *null* variable.

It is necessary to create an object and save its reference in a variable, as shown in the following code:

```
text = new File
```

Now, the **text** variable does point to a new **File** object. A created object is also called an object *instance*.

If you were to write the above line of code twice, two **File** objects would then be created but the **text** variable will only contain the reference to the second **File** object. The reference to the first **File** object is lost, meaning that this object cannot be accessed for the remainder of the program. In test mode, this would cause memory leaks.

It should be obvious that if an object is created using the **new** command, it must also be removed using the **delete** command.

```
delete text  
text = null
```

The first line destroys the object. But the **text** variable still contains a reference to an object which has been deleted and therefore does not exist anymore. If you attempt to use this variable, **SECCIA** will generate an error message. In order to avoid such errors, it is recommended that the reference be reset to null. Any attempt to use this variable will still create an error, but one will know that the variable is not pointing to any object anymore.

This process is not obvious for novices but it is essential. Thankfully, the creation and deletion of objects as described above can be managed automatically in **SECCIA**. One only needs to declare the variable using the **new** keywords and then to forget about it:

```
File text new
```

This way of declaring variables makes our earlier instructions redundant. In effect, when the variable is managed directly by the language, its reference is fixed. For this type of variable, it is not possible to assign a reference using the '=' character, contrary to the example below:

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

The assignment's only purpose is to copy the reference. There are two **File** variables but only one **File** object. Both variables can access the same object.

The choice between the two methods described above will be driven mostly on the kind of objects that need to be created and the type of application being developed. It is perfectly possible to develop a full application without recourse to the more complex method, letting the **SECCIA** language take care of instances.

Before pressing on, let's define terms so that we have a clear understanding of the words and expressions used in the rest of this guide:

Object: for naming types (**File, Integer, String...**)

Variable: for declaration (**text, a, b...**)

Instance: for created objects (**new File**)

Reference: The reference to an instance contained in a variable (a digital address coded in 4 or 8 bytes depending on the operating system (32 or 64 bits)).

## TABLE OF CONTENTS

1. NUMBER AND TEXT OBJECTS.....	4
2. USING OBJECTS.....	5
3. OBJECT CREATION.....	7
4. FUNCTION PARAMETRES.....	10
5. RETURN VALUES.....	11
6. CONDITIONS AND LOOPS.....	12
7. OPERATORS.....	14
8. STRUCTURE OF AN APPLICATION.....	15
9. EVENTS.....	16
10. INHERITANCE.....	18
11. POLYMORPHISM.....	21
12. CONSTANTS.....	22
13. THE HIDDEN KEYWORD.....	24
14. SUB-APPLICATIONS.....	25
15. C++ EXTENSIONS.....	26
16. COMPILATION KEYWORDS.....	27
17. SECCIA CONVENTIONS.....	28
18. LANGUAGE KEYWORDS.....	29

## 1. NUMBER AND TEXT OBJECTS

The **Integer** object stores signed integers coded in 8 bytes (64 bits).

The **Float** object stores floating numbers coded in 8 bytes (64 bits).

The **String** object stores character strings (1 byte per character).

The **UniString** object stores Unicode strings (2 bytes per character).

There are, therefore, three types of constants as shown in the table below.

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

An integer can be written in its hexadecimal form. In this case, you must add the prefix **0x** in lower or upper case. Letters (A, B, C, D, E, F) can also be written in lower or upper case.

The language is capable of automatic conversion. The following conversions are possible:

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

**Note:** Dividing one **Integer** by another will always return a **Float**.

The **Value** object groups all three types in a single object. Type conversion is then automatically handled by the language.

Finally, please note that the **%** character enables the insertion of any ASCII character by appending its hexadecimal value and that the **\$** character allows the insertion of the value of an **Integer**, **Float**, **String** or that of a constant.

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

This for is particularly useful for database requests.

**Important:** The ``` character will be replaced by the `'` character by default to ensure compatibility with database requests.

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

## 2. USING OBJECTS

An object has its own functions, variables and constants. There are many objects in the **SECCIA** software: here is a non-exhaustive list.

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

In addition, you will be able to create your own objects.

In the introduction, we briefly covered the creation of instances, explaining that there are two methods that can be used. Here is an example using each method in turn:

```
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
```

For the **Integer, Float, String** and **UniString** objects, the process is slightly different as it is not possible to create an instance of such objects outside their declaration, for a simple reason: the variable does not return the reference of the instance but its value, directly, in the shape of an integer, a floating number or a character string depending on the object type.

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

In the above, we have two **Integer** variables handled by the language and therefore two distinct **Integer** instances.

*Note: An **Integer** is always initialized with a value of 0.*

Here, we first change the value of **a**, then the value of **b** by adding 5 to the value of **a**. Finally, we add one unit to the value of **b**, so  $10+5+1$  i.e. 16.

This kind of syntax is quite common but one of the priorities of the language was to maintain basic logic. This is why all three objects possess **Set** and **Get** functions that allow the setting and retrieval of their value.

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

As the **Integer, Float** and **String** objects cannot be instantiated, the **new** keyword becomes optional.

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

Thus the only way to get at the instance reference is to call the `GetObjectRef` function.

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

The global function `GetObjectRef` will be used for all objects.

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

As their name implies, global functions can be accessed from any other function. This said, in case of conflict, if an object possesses a function whose name is the same as a global function, the latter becomes inaccessible. This can be overcome: it is possible to include the object name as only one instance of the `Global` object exists. This is also true of the `Application` object.

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

As for constants, if any are contained in an object, one simply writes:

```
var.NAME_OF_THE_CONSTANT
```

The same applies to member variables:

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

The keywords `true` and `false` are integers. They have the advantage of being more explicit than 0 and 1 values in parameters.

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

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

Finally, the `null` keyword corresponds to a value of 0. Its goal is to indicate that a variable does not point to any instance.

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

### 3. OBJECT CREATION

The creation of new object types is a very powerful tool. It will allow you to structure your applications more efficiently and make them future-proof.

Let's take a simple and concrete example and develop it step by step. We are going to create a new object called **Box**. We could describe it as a cardboard box with customizable width, height and depth.

The **object** keyword defines the new object.

```
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
}
```

The squiggly brackets are compulsory to delineate the content of the object. In this structure, only constants and variables can feature. A constant, as its name implies, has a fixed value. That is why its value is defined when it is declared. Constants are always members of an object. By convention, the names of constants are written in upper case and variables members in lower case preceded by 'm\_'. The name of an object starts with upper case. This avoids possible conflicts.

It is possible to access the member variables of an object directly or indirectly using a function. The direct reading of a variable is quicker than calling a function, but the latter will be used by default when a member variable is being modified.

After defining our new object and its member variables, we are going to define its functions. Note that each function has its own variables and that these are local variables, unlike member variables (shorthand for variables that are members of an object). A function (also called a method in certain languages) is defined using the **function** keyword:

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

To indicate to the **SECCIA** language the parent object of a function, we write the name of the object after a full stop.

The **SetSize** function is used to change the size of the box. It requires three parameters of the **Integer** type which will be considered as local variables.

**Important:** Only references to instances are passed on through parameters, except for **Integer**, **Float**, **String** and **UniString** objects. For the latter, a new copy of the instance is created which then becomes local to the function.

In addition to parameters, functions can return data. We are now going to define a new function designed to calculate the volume of our cardboard box. This function will not require parameters but it will return a value of the volume expressed in cm<sup>3</sup> or in ml.

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

The definition of the function is somewhat different from our first function. As this function has no parameters, the brackets are empty, which is logical. But as the function returns a value, it is imperative that the type of value returned be indicated before the parameters.

In the code, we multiply the width, height and depth and return the result using the **return** keyword. This keyword does not require a value by default as long as the function does not specify a return type. This may be useful to exit a function prematurely, for example.

**Important:** When the returned type is an **Integer**, the returned value is automatically converted to an integer. For the **Float** object, the returned value is converted to a floating number and to a character string for the **String/UniString** objects. But for all other objects, only the reference of the instance is returned.

**Warning:** One must never attempt to return the reference to a local variable handled automatically by the language. The instance created when the function is called will be destroyed automatically and the function will then return the reference to a non-existent instance.

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

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

Note that there are two reserved functions. When you create a new instance of an object, you also create its member variables. It is often useful to initialize some of the values.

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

Those two functions are called automatically by the language at the moment of creation and destruction of an instance. They have no parameters and no return values. They are optional and cannot be called by the coder.

We did not use the **MINSIZE** constant on purpose to keep the code light in our example. This constant could be used to check whether the size of the box conforms to some desired specification.

It is now time to create our first instance, in the same way as one creates an **Integer** instance.

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

We change the size of the **myBox** box using the values: 20 cm, 5 cm and 10 cm. Then, we calculate the volume of the box and store the result in the **volume** variable: 1000 cm<sup>3</sup> or 1000 ml or 1 liter.

**Note:** *The declaration of variables is generally located before the code.*

A last remark on this topic: to identify the current instance of an object within a function, one uses **this** keyword. This keyword returns the instance that called the function. It has the same type as the object by definition. One of its uses is allowing the transmission of the instance reference via a function parameter.

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

At first sight, one might think the above code is not very useful as the **MySecondFunction** function is also a function of the **MyObject** object, and is therefore capable of using the **this** keyword itself. But in practice, the second function could be a function of another object, as shown in the following code:

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

Or the object can be the same but in the shape of a different instance:

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

#### 4. FUNCTION PARAMETRES

We have already used parameters in this guide but we haven't exhausted the subject. As you know, functions and events can contain parameters within brackets. When one of these components is to be called, it is necessary to specify a value for each parameter. If parameters are numerous, this easily becomes a fastidious task

In order to avoid unnecessary specification, parameters can be made optional. In that case, the defined default value will be used.

```
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
```

Only constants and the following keywords are allowed: **null**, **true** and **false**.

When a parameter has a default value, all parameters to the right have to be optional (i.e. have a default value specified).

By way of example, the following code is wrong as **z** is not an optional parameter:

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

## 5. RETURN VALUES

You will have noted that it is not possible to return the reference of **Integer**, **Float**, **String** or **UniString** objects.

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

Firstly, the instance of the **Integer** object is local to the function and will be destroyed on exit. Secondly, the second line of code does not return the instance reference but its value (25).

Use the **Value** object in the following way instead:

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

You will have to delete the **Value** instance yourself.

## 6. CONDITIONS AND LOOPS

You have seen some of the language syntax but we still have to review, among other things, conditions and loops.

```
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
```

If several **case** exist specifying the same value, only the first **case** is taken into consideration.

## 7. OPERATORS

We are all familiar with the four basic operators for calculation ('+', '-', '/', '\*'). More complex calculations are performed using the **Math** object.

*Note: For character strings, the '+' sign is used for concatenation.*

The remainder is depicted by the '%' character, available in the **Math** object as a function.

Sign	Definition	Example
+	performs an addition	a + b
-	performs a subtraction	a - b
*	performs a multiplication	a * b
/	performs a division	a / b
%	returns the remainder of a division	a % b

In programming, certain operators compare two expressions. They are the equality operators:

Sign	Definition	Example
==	is equal to	a == b
!=	is different from	a != b
>	is greater than	a > b
<	is smaller than	a < b
>=	is greater than or equal to	a >= b
<=	is smaller than or equal to	a <= b
?=	is similar to (see String.IsSimilar)	a ?= b

**SECCIA** also allows bit-to-bit operators for integers.

Sign	Definition	Example
	operator OR inclusive (OR)	a   b
&	operator AND	a & b
^	operator OR exclusive (XOR)	a ^ b
<<	moves bits to the left	a << b
>>	moves bits to the right	a >> b

Unitary operators are as follows:

Sign	Definition	Example
~	invert all bits	~a
!	logical negation	!a
++	adds 1 immediately to an Integer/Float	a++
--	subtracts 1 immediately from an Integer/Float	a--

Assignment operators are as follows:

Sign	Definition	Example
+=	performs an addition	a += 2
-=	performs a subtraction	a -= b
*=	performs a multiplication	a *= b
/=	performs a division	a /= 2
%=	returns the remainder of a division	a %= 10

Bit-to-bit operators obey the following rules:

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. STRUCTURE OF AN APPLICATION

An application is simply an object named **Application**. You will need to create the object and name it **Application** before it can be considered the main object.

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

You will have understood that **Constructor** will be the first function called by your application. **SECCIA** will handle the creation of an instance of the **Application** object.

Unlike conventional objects, the **Application** object can, in addition to the above functions, a third function, as follows.

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

The **Loop** function is only called if the application has activated the infinite loop method using **EnableApplicationLoop**. By definition, this method is not active. When the program is in loop mode, the application calls on computer resources continuous. This is similar to an internal **while** loop.

It is unnecessary and unadvisable, therefore, to activate this method unless absolutely required. However, it is particularly useful in game development, for example, when the game requires an infinite loop.

*Note: The **DxEngine** object in "auto refresh" mode requires the loop mode to be active.*

## 9. EVENTS

**Windows**<sup>®</sup> programming is based on the principle of events and messages in a queue. This is very much how controls operate. When a user clicks on a button, the application must be capable of intercepting the message of the click.

In **SECCIA**, by convention, events are preceded by the word 'On'. The button will therefore have an **OnClick** event.

One must differentiate between events and messages. Events are located within objects, like functions. Messages are defined in other objects as they are used to intercept events.

The declaration of an event is similar to that of a function. An event can contain parameters and a return value. The only difference is the existence of the **event** keyword in the declaration.

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

The above is not sufficient, however. We need to call this event from our **Box** object or it will never be called. Unlike functions, events cannot be called externally. It is the object itself that has to call its events.

**Important:** Events generally contain no code except a **return** if the declaration has a return type.

Let's revisit our original example, adding an event call when the dimensions of the box change. We have already written a function to change the dimensions. We now add a call within the function

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

For the time being, we concentrate on the object itself but bear in mind that events are of no use to the object, only to its parent object

When one clicks a button, it is not for the button to receive the message, but for the object that created the button. That is why events remain empty and our object does not contain a message.

Let's get back to the **Application** object and add the event belonging to our **Box** object in the guise of a message.

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

Contrary to functions and events, messages need an additional piece of information: the name of the object in which the event is located.

In summary: there is an **OnChange** event in the **Box** object and an **OnChange** message in the **Application** object. The obvious question is: which one is called? The event or the message? This depends not on the object, but on the instance. As **m\_box** is created in the **Application** object, it is the **Application** object that will receive the

message. The **OnChange** event of **m\_box** will never be called. But if the **OnChange** message is not specified in the **Application** object, then the **OnChange** event of **m\_box** will be called.

So far, we have only covered the variables handled by the language with the help of the **new** keyword. When it comes to the manual handling of instances, a new keyword is required: **intercept**.

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

The **intercept** keyword allows the interception of all the events of an instance. If the line of code is removed, the **Box** instance will not send messages to the **Application** object. This keyword cannot be applied to variables handles automatically by the language because in that case, intercepts are also handled automatically.

There is one point we still have to clarify at this stage, on the subject of messages. When we define a message, we specify the name of the object containing the event. If we create two instances of **Box**, it becomes difficult to know which instance is sending the message. Of course, the language provides a special keyword to identify the calling instance: **caller**.

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

The **caller** keyword contains the reference to the instance generating the message. As a result, it can only be used in messages.

## 10. INHERITANCE

Inheritance is a more complex concept in Object-Oriented Programming. It allows for the creation of an object derived from another object. The derived object has a base object and inherits all the members of that base objects (functions, variables and constants).

**Important:** Neither events nor messages are inherited. Neither are the functions of **Constructor** and **Destructor**.

Starting from scratch but using our original example for illustration, we will demonstrate the value of inheritance. We have a **Box** object that represents a cardboard box. Now we intend to put away in that box two types of toys: plastic balls and cubes. These balls and cubes are not all of the same size. Nevertheless, they have a common denominator in that they will be the toys that are put away in the cardboard box. So we can create a base object **Toy** and two derived objects, **Ball** and **Cube**.

In programming terms, we write the following code:

```
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
}
```

We now have four distinct objects, two of which are derived from the **Toy** object. The **from** keyword indicates that the object is derived.

You will notice that the **Toy** object possesses an **Integer** which is inherited by the **Ball** and **Cube** objects. This variable represents the weight of the toy. This way, we do not have to create two variables (one in **Ball** and one in **Cube**).

To simplify, we have removed the dimensions from the **Box** object.

**Important:** If a derived object possesses a member variable that is already present in the base object, then two distinct variables with the same name will co-exist. But it will then be impossible to access the base objects variable externally from the derived object.

A new type of object we have not yet introduced appears in **Box**. That is **Array**, and object that provides (no surprises there) array management. The toys featuring in that array will be those present in the cardboard box.

**Important:** Arrays only store references to instances. **Integer**, **Float**, **String** and **UniString** cannot be used in conjunction with an array because they do not return references. This is why the language provides you with dedicated array objects such as **IntegerArray**, **FloatArray**, **StringArray** and **UniStringArray**.

The next step is defining the functions we need:

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

The **AddToy** function will add a single toy to the cardboard box.

```
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()
}
```

The **GetToyWeight** function will return the total weight of the toys inside the box. It requires a loop to cycle through all the toys whose weight we need to add up.

```
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
}
```

The **Get** function of the **Array** object returns the reference of the instance stored in the specified index.

In the application code, we are going to create two toys: a ball and a cube. The **AddToy** function is defined by the parameter **Toy**. This cause no conflict as a **Ball** object is also a **Toy** object.

```
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()
```

The total weight of the toys will be 150 grams (50+100).

Inheritance can go further. The language is capable of handling objects derived at several levels. Nothing prevents us from defining another object, derived from **Ball**. This new object will inherit not only the member variables of the **Ball** object but also those of the **Toy** object.

Let's take this opportunity to explain how it is possible to access a variable or a constant defined in two distinct places. To do this, we add a new object called **Truck** derived from **Toy**, with a 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
}
```

To access the variable **m\_weight** of the **Toy** object, you need to write the name of the variable followed by a full stop. This is because the **m\_weight** version of the **Truck** object hides the version in **Toy**.

## 11. POLYMORPHISM

Polymorphism can be described as the logical progression from basic inheritance. It will bring to your application additional flexibility and power by way of a more generic form of programming. The goal of polymorphic inheritance is to enable the calling of an object function regardless of its intrinsic type.

We have seen that it is possible to have a member variable already featuring in a base object. In this case, two variables co-exist. But the same is not true of functions because a function that is already present in a base object becomes virtual if it also features in the derived object.

Let's proceed by adding a new function to the **Toy** object.

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

From now on, the **GetVolume** function exists in both the **Toy** object and in the derived **Ball** and **Cube** objects.

Let's examine the following code:

```
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
```

One would think that **myToy2** contains the reference of an instance of the **Toy** object but, in reality, it contains a reference to an instance of the **Ball** object. Thus, the virtual function **GetVolume** is linked to the function **GetVolume** of the **Ball** object, not with that of the base object **Toy**.

*Note: With inheritance, there is no more certainty concerning the type of an instance. One must differentiate between the type of a variable and the type of an instance.*

Another question arises regarding the virtual function. Is it possible to make a static call to a function located in a base object? Of course, otherwise our code would face some serious issues.

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

As you can see, the **GetAge** function calls itself indefinitely and we can never call the **GetAge** function of the base object.

To get access only to the base object function, all we need to do is to add its name. The object instance concerned is the current instance.

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

One can thus move up from base to base if one wants to call all the functions.

## 12. CONSTANTS

The use of constants is essential in programming. Never leave values in your instructions:

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

Here, we calculate the number of days in a date based on a 30 day month. The number 30 is not explicit. A constant would be better suited.

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

This facilitates a change in value which may be required at a later stage.

With the use of hexadecimals, constants bring us important additional functionality. Let's take a specific example. We want to create a **Pizza** object which would hold information about its ingredients. We could construct the object as follows:

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

Each variable can be either **false** or **true** and could be further edited using three functions (and three more to read the values)

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

If our pizza contains all three ingredients, we have to write the following code.

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

Then, we must ensure that we can test which ingredients are present. Let's say we want to know if our pizza contains anchovies. In that case, the operation is simple.

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

To find out if the pizza contains all three ingredients, we need three conditions.

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

Now, let's take constants into account and revisit our example from scratch. Instead of having three variables – one for each ingredient –, we only need the one.

```
object Pizza
{
```

```

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

```

The fourth constant would take the value 0x08, the fifth 0x10, the sixth 0x20 and so on.

```

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

```

Revisiting the code in our previous version of this example, we make the following modifications to accommodate constants.

```

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

if Flag(pizza.Get(), Pizza.ANCHOVY)
end

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

```

To simplify further, we can add two constants that define the specialty of the pizza based on the accumulation of ingredients.

```

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

```

So, for example, to know if the pizza is a Neapolitan, we just need to test the `m_type` variable.

```

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

```

Now for a subtle difference: the code below does not enable you specifically to know if the pizza is a Neapolitan but it will tell you if the pizza contains all the ingredients of the Neapolitan and perhaps some extra ingredients.

```

if Flag(pizza.Get(), Pizza.NEAPOLITAN)
end

```

Finally, as constants can be used in a `switch`, it is possible to test the pizza type as follows:

```

switch pizza.Get()
case Pizza.MARGUERITE
    break
case Pizza.NEAPOLITAN
    break
end

```

### 13. THE HIDDEN KEYWORD

The code editor is capable of listing all the members of an object automatically inside a scrolling menu as you write your code. This facilitates the coding process.

The editor displays not only the members of the chosen object but also the members of its base objects. To make them visually distinct, a different icon is visible to the left of the name.

It is possible to exclude certain variables, constant and functions displayed in the list by adding the keyword **hidden** in the declaration.

```
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:** Those components may be hidden but they remain accessible. The exception is the **SubApplication** object for which only non-hidden constants, functions and events are exported.

## 14. SUB-APPLICATIONS

An application can be compiled in the shape of a Sub-Application: a DLL file with a **.sub** extension is produced. As for executable files produced by **SECCIA**, the DLL contains resources and the application's source code with the obvious exception of the **Application** object which is never included.

Sub-Applications are independent and reusable in other applications, and are similar to C++ extensions.

To use a Sub-Application within the software, you must first copy the relevant file to the **SubApplications** folder before starting **SECCIA**. The Sub-Application can then be accessed from the **Plugins** category of the object list.

Some restrictions apply to Sub-Applications. Their name always starts with the **Sub\_** prefix. Functions and events to be exported can only use the **Integer**, **Float**, **String** and **UniString** objects as it is not possible to access the application's objects (read or write) externally. Inheritance is not possible. Finally, a Sub-Application cannot be a control accessible from the dialog editor.

Only the functions and events of the **Sub-Application** object will be exported, as long as they conform to the following rules:

- Functions must be declared without the **hidden** keyword.
- Parameters must be of one of the following types: **Integer**, **Float**, **String** or **UniString**.
- The return value, if it exists must be one of the following types: **Integer**, **Float**, **String** or **UniString**.

## 15. C++ EXTENSIONS

The SDK (*Software Development Kit*) enables the development of independent external modules using the C++ language. These extensions are DLL files created exclusively for use within **SECCIA** and applications developed within it, as simple objects. The advantage of extensions is that they can bring new functionality to the **SECCIA** environment whether the author decides to distribute the source code or not.

To use an extension within the software, first copy the relevant file to the **Extensions** folder before starting **SECCIA**. The extension will then be available in the **Plugins** category of the object list.

Bear in mind that restrictions apply to extensions. Their name always starts of the **Ext\_** prefix. An extension's functions and events can only use **Integer**, **Float**, **String** and **UniString** objects as it is not possible to access an application's objects externally (read or write). Inheritance is not possible.

Unlike Sub-Applications, extensions can be controls accessible from the dialog editor. In this precise case, the extension is derived from the **Control** object.

For more information, please refer to the SDK documentation.

## 16. COMPILATION KEYWORDS

During compilation of the executable file or in test mode within **SECCIA**, the language is capable of excluding selected lines of code simply by using one of the following special keywords.

`_ifdef`, `_ifndef`, `_else`, `_end`

Those keywords do not form part of your application's code at run time.

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

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

If `VERSION_DEMO` has been defined, the code will be integrated into the program. Otherwise, the `Demo` function does not exist and cannot be called by definition. This is why it is also necessary to exclude the call to the `Constructor` function.

The final result will depend on definitions. In our case, there are two cases:

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

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

## 17. SECCIA CONVENTIONS

The language is case-sensitive. That is to say that it differentiates between upper-case and lower-case. This is an essential requirement for efficient code but it requires a certain rigor. Let's be clear: best practice dictates that two variables local to a function must not have the same name. This would render code incomprehensible. Variables need explicit names. But this is not what case-sensitivity is about. Case-sensitivity is used to differentiate element types.

*Note: You are free to name your variables as you see fit. The below are only practical guidelines that will help structure your code more efficiently.*

Let's first list all element types: constants, objects, functions, member variables and local variables. Words in blue and grey are reserved keywords.

In most languages, constants are written in upper-case. We use the same convention.

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

**SECCIA** objects will all start with an upper-case letter and we shall use the same convention for our own objects.

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

Functions, events and messages always have brackets and no other constraint applies to the naming of functions as in: **Integer**. Nevertheless it is clearer to use a first upper-case letter – as for objects – and to refrain from using a name for a function that is the same as the name of an object.

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

To identify an event more quickly, we shall adopt the prefix 'On'.

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

A local variable can have the same name as a member variable without creating a conflict. The member variable then becomes inaccessible except when using functions. A standard convention is to write 'm\_' before each member variable.

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

## 18. LANGUAGE KEYWORDS

Here is a complete list of keywords in **SECCIA**:

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