

LINGUAGEM DE PROGRAMAÇÃO SECCIA

Autor: Sylvain Seccia

Copyright © 2001-2009 Sylvain Seccia

SECCIA é uma linguagem orientada a objetos, que inclui herança e polimorfismo. É uma linguagem muito pouco tipada (apenas com inteiros, decimais e texto), sensível e com uma sintaxe clara e simples. As funções podem ser chamadas recursivamente.

Tudo é baseado em objetos, até mesmo a gestão de números inteiros, decimais e texto. As variáveis são apenas referências a objetos. A declaração de variáveis é a seguinte:

```
File ficheiro
```

Neste caso **ficheiro** é uma referência do objecto **File** (para ler e escrever em ficheiros). Escrito desta forma, a variável não se refere a qualquer objeto. É no entanto apenas uma referência. É então necessário criar um objeto e gravar a sua referência numa variável como mostra a linha a seguir:

```
ficheiro = new File
```

Assim, a variável **ficheiro** torna-se parte do objecto **File**. É então criada uma instância desse mesmo objeto. Se escrever uma segunda vez a mesma instrução, terá criado como resultado, dois objetos do tipo **File**, em vez de apenas um, mas a variável passa a estar referenciada apenas ao segundo objeto ficando a referência ao primeiro perdida. Desta forma é impossível aceder ao primeiro objecto até ao fim do programa e a tentativa de aceder a ele pode provocar falhas graves no programa.

Escusado será dizer que se um objeto foi criado com o comando **new** deve ser terminado com o comando **delete**.

```
delete ficheiro  
ficheiro = Null
```

A primeira linha destrói o objeto. No entanto, a variável **ficheiro** contém ainda uma referência ao objeto quando ele já não existe. Uma mensagem de erro será gerada quando alguém tentar usar essa variável. Para evitar qualquer erro, é aconselhável assignar a mesma a nulo (0), ou seja, **null** (zero). Embora o seu acesso cause um erro, temos a garantia que a variável não aponta para um objeto.

Esta manipulação não é óbvia para todos, mas é necessária. Entretanto, a criação e destruição dos objetos que foram descritas anteriormente podem ser manuseadas automaticamente pela linguagem. Nesse caso, basta declarar uma variável usando directamente a palavra-chave **new** na sua declaração:

```
file ficheiro new
```

Esta declaração torna desnecessária a utilização das instruções acima descritas. De fato, quando a variável é gerada pela linguagem, a sua referência é fixa.

A atribuição da referência pelo sinal '=' não pode ser aplicada a este tipo de variável ao inverso do exemplo seguinte:

```
file a
file b
a = new file
b = a
```

A última linha de código é destinada apenas para copiar a referência. Existem duas variáveis no código, mas apenas uma do tipo **file**. E as duas variáveis podem aceder ao mesmo objeto.

A escolha entre estes dois métodos depende principalmente do tipo de aplicação a desenvolver.

Antes de prosseguir, é importante compreender os termos a serem usados a partir daqui:

Objeto: Para os nomes do tipo (**File, Integer, String ...**)

Variável: Para as declarações (**texto, a, b ...**)

Instância: Para os objetos criados (**new File**)

Referência: A referência a uma instância numa variável (um endereço numérico é sempre escrito em 4 ou 8 *bytes*, dependendo do sistema operacional (32 bits ou 64 bits)).

1. OS OBJETOS NÚMERICOS E DE TEXTO

O objeto **Integer** é utilizado para armazenar inteiros codificados até 8 *bytes* (64 bits).

O objeto **Float** pode armazenar decimais até 8 *bytes* (64 bits).

O objeto **String** pode armazenar texto com tamanho indefinido.

Existem então três tipos de categorias como nos é mostrado na tabela a seguir:

	Decimal	Hexadecimal
Integer	182 255 1000	0xB6 0xff 0x3E8
Float	15,6 -1000,0	
String	"Olá Mundo"	

Um inteiro pode ser escrito na forma hexadecimal. Para isso, você deve acrescentar um zero e um X seguidos (sem espaços) em letras maiúsculas ou minúsculas. As letras (A, B, C, D, E, F) também poderão ser escritas em letras minúsculas.

A linguagem é capaz de efectuar conversão automática. Assim são possíveis as seguinte conversões:

	Integer	Float	String
Integer	Integer	Float	String
Float	Float	Float	String
String	String	String	String

Nota: A divisão entre os dois inteiros (**Integer**) retorna sempre um decimal (**float**).

O objeto **Value** inclui todas as três categorias num único objeto. As conversões são assim processadas automaticamente pela linguagem.

Finalmente, um último esclarecimento acerca da notação no texto do código: o sinal de % pode inserir qualquer carater ASCII especificando o seu valor hexadecimal e o sinal \$ é utilizado para indicar o valor de um **integer**, **float**, ou uma **string** de texto.

```
texto = "o meu nome é 22%miguel22%" // o meu nome é "miguel"!
```

```
string nome
nome = "miguel"
texto = "o meu nome é $nome" // o meu nome é miguel
```

Esta forma é usada principalmente para consultas de dados nas bases de dados, e neste caso, é muito prática.

Importante: O carater ` será automaticamente substituído pelo carater ' para ser compatível nas consultas de dados nas bases de dados.

```
pergunta = "És o `miguel`?"
textoSQL = "Table.Field=`$pergunta`" // não existe conflito
```

2. OS OBJETOS EM GERAL

Um objeto tem as suas próprias funções, os seus próprios valores e as suas próprias constantes. Existem muitos objetos no software **SECCIA**, mas aqui fica uma lista não exaustiva dos mesmos:

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

Irá ainda ser possível criar e manipular os seus próprios objetos.

Na introdução, discutimos que existem duas maneiras de conseguir isso. Aqui está um exemplo de ambos os métodos:

```
file ficheiro new
ficheiro.OpenFile( "c:\myFile.txt" , true , true ) // modo de escrita
ficheiro.Write( "Os meus dados" )
ficheiro.Close()
```

```
file ficheiro
ficheiro = new file
ficheiro.OpenFile( "c:\myFile.txt" , true , true ) // modo de escrita
ficheiro.Write( "Os meus dados" )
ficheiro.Close()
delete ficheiro
```

Para os objetos **Integer**, **Float** e **String** a sua manipulação é um pouco diferente porque não é possível para criar uma instância fora da respectiva declaração. Isto existe por uma simples razão: A variável não retorna a referência do objeto diretamente, mas o seu valor como um inteiro, decimal ou texto, dependendo da ação.

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

Neste caso temos duas variáveis inteiras (**Integer**) geridas pela linguagem e, portanto, duas instâncias inteiras (**Integer**) diferentes.

Nota: Um inteiro (*integer*) é sempre inicializado com o valor 0.

Mudamos o valor de **a** e **b** assignado ao **a** o valor 10 e adicionado ao valor **b** o **a** mais 5. Finalmente acrescentamos uma unidade ao valor de **b**, ou seja, $10 + 5 + 1 = 16$.

Esta sintaxe tornou-se generalizada, mas uma linguagem como esta deve manter uma base lógica. Portanto, estes três objetos (**Integer**, **Float** e **String**) têm as propriedades **Get** e **Set**.

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

Porque os objetos **Integer**, **Float** e **String** não podem ser instanciados, a palavra-chave **new** é opcional aquando da sua declaração.

```
integer a
integer b new
a = 10
b = 20
```

Assim, a única maneira de obter a sua referência é através da chamada à função **GetObjectRef**.

Para objetos que contêm constantes, basta escrever:

```
var.NAME_OF_THE_CONSTANT
```

O mesmo é verdadeiro para as variáveis:

```
var.m_idade
var.m_idade = 18
```

As palavras-chave **true** e **false** são numéricos inteiros. Têm a vantagem de serem mais explícitos que os valores 0 e 1.

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

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

Finalmente, a palavra-chave **null** corresponde ao valor 0 e destina-se a indicar que uma variável não aponta para qualquer instância .

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

3. A CRIAÇÃO DE OBJETOS

A criação de novos tipos de objetos é uma ferramenta extremamente poderosa que irá ajudá-lo a organizar as suas aplicações e torná-las mais escaláveis.

Considere o seguinte cenário.

A criação de um novo objeto chamado **Caixa** (Box). Especificamente poderíamos compará-la como uma caixa de papelão com largura, altura e profundidade com diferentes valores.

A palavra-chave **object** permite definir um novo objeto.

```
object Caixa
{
    constant TAMANHO_MINIMO 2 // Tamanho minimo em cm
    integer m_largura          // Largura em cm
    integer m_altura           // Altura em cm
    integer m_profundidade     // Profundidade em cm
}
```

As chavetas servem para definir os limites do objeto. Nesta estrutura, apenas constantes e variáveis membro (*propriedades*) podem ser incluídas. Uma constante, como o próprio nome indica, é um valor fixo. É por esta razão que o seu valor é definido ao mesmo tempo em que a constante é criada. Por convenção as constantes são escritas em letras maiúsculas e as variáveis em minúsculas precedidas por "m_". Os nomes dos objetos começam com uma letra maiúscula. Assim são evitados eventuais conflitos.

É possível aceder às variáveis membro de um objecto, directa ou indirectamente, através de funções (*métodos*). Se a sua leitura directa é mais rápida do que a chamada de dentro de uma função, o melhor é utilizar esse procedimento.

Após a definição do objeto e as suas variáveis membro, vamos definir as suas funções. Note que cada método tem as suas próprias variáveis locais, que são contrárias às variáveis membro (no ciclo de vida de uma instância do objeto). A função (também chamada em algumas línguas de método) é definida pela palavra-chave **function**:

```
function Caixa.AtribuiTamanho( integer l, integer a, integer p)
{
    m_largura = l           // Nova largura
    m_altura = a           // Nova altura
    m_profundidade = p     // Nova profundidade
}
```

Para especificar o tipo do objeto pai numa função, é necessário incluir o nome do objeto separado por um ponto. A nossa função **AtribuiTamanho** é utilizada para alterar o tamanho da caixa. A mesma exige três parâmetros do tipo inteiro (**Integer**) e que são considerados como variáveis locais.

Importante: Estas são referências transmitidas pelos parâmetros, com exceção dos objetos **Integer**, **Float** e **String**. Para este último, uma nova cópia do processo é feita tornando-se local à função.

Na maioria das vezes uma função retorna dados. Vamos agora definir uma nova função para o cálculo do volume de nossa caixa (*Box*). Esta não vai precisar de passar argumentos, mas precisa retornar um valor que representa o volume em *cc* ou *ml*.

```
function Integer Caixa.CalcularVolume()
{
    return m_largura * m_altura * m_profundidade
}
```

Aqui a definição é ligeiramente diferente. Como a função não contém qualquer parâmetro, os parênteses estão vazios o que é inteiramente lógico. No entanto, como a função retorna um valor, é fundamental indicar qual o seu tipo.

No código, multiplicamos a largura pela altura e pela profundidade e retornamos o resultado através da palavra-chave **return**. Esta palavra-chave não precisa necessariamente de um valor associado, desde que na definição da função não especifiquemos um tipo de dados a retornar. O interesse de um tal cenário seria por exemplo a necessidade de uma saída prematura da função.

Importante: Quando o tipo de retorno definido pela função é um inteiro (**integer**), o valor fornecido é automaticamente convertido para um inteiro. Para um decimal (**Float**), ele é convertido em decimal e o objeto **String** em uma cadeia de texto. Contrastando com os outros, apenas a referência para a instância do objeto é retornada.

Atenção: Nunca retorne uma referência de uma variável local manipulada automaticamente pela linguagem, porque o corpo criado na função será automaticamente destruído e a função retorna uma referência não existente.

```
function color MeuObjeto.Get()
{
    color cor new
    return cor // ERRO!

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

Finalmente e sobre este assunto, observe que existem duas funções reservadas, **constructor** e **destructor**. Estas servem por exemplo para inicializar e destruir as variáveis membro. Quando cria uma nova instância de um objeto, são também criadas as suas variáveis membro.

```
function Caixa.Constructor()
{
    m_largura = 10
    m_altura = 10
    m_profundidade = 10
}
```

```
function Caixa.Destructor()
{
}
```

Estas duas funções são naturalmente chamadas automaticamente pela linguagem na criação e destruição de uma instância do objeto. Elas são opcionais e podem ser chamadas pelo programador.

Nós não usamos a constante **TAMANHO_MINIMO** com o propósito de reduzir o código. Esta constante poderá ser utilizada para verificar se o tamanho da caixa está de acordo com características desejadas. É agora tempo de criar a nossa primeira instância da mesma forma que criamos um objeto numerico inteiro (**Integer**).

```
Integer volume new
Caixa caixaQuadrada new

caixaQuadrada.AtribuiTamanho(20, 5, 10)
volume = caixaQuadrada.CalcularVolume()
```

Primeiro mudamos o tamanho do caixa com os valores: 20 cm, 5 cm e 10 cm. Depois mandamos calcular o volume e armazenar o resultado na variável **volume**, que é 1000 cc, 1000 ml ou mesmo 1 litro.

Nota: A declaração de variáveis é normalmente efectuada antes do restante código.

Um último esclarecimento sobre este capítulo. Para identificar a atual instância do objeto numa função, basta usar a palavra-chave **this**. Esta retorna a instância que chamou a função e é do mesmo tipo do objeto. Um dos seus interesses é enviar a referência do objeto através de um parâmetro da função.

```
function MeuObjeto.PrimeiraFuncao()
{
    SegundaFuncao(this) // Passa a instância
}
```

À primeira vista você poderia pensar que isto não tem muito interesse, porque a função **SegundaFuncao** também é uma função do objeto **MeuObjecto**, e pode até recuperar a palavra-chave **this**. Mas isso dependerá da configuração, e se a função precisa de outra referência ou não. Na prática, a segunda função poderia ser também uma função de outro objeto como mostrado no exemplo seguinte:

```
function MeuPrimeiroObjeto.MinhaFuncao(MeuSegundoObjecto obj)
{
    obj.MinhaFuncao(this)
}
```

Ou o objeto pode ser exactamente o mesmo, mas de uma instância diferente:

```
function MeuObjeto.MinhaFuncao(MeuObjeto obj)
{
    obj.MinhaFuncao(this)
}
```

4. OS PARÂMETROS DAS FUNÇÕES

Nós já conversamos sobre os parâmetros, mas ainda não falamos tudo sobre o assunto. Já sabe que as funções e os eventos podem conter parâmetros entre parênteses. Quando estes parâmetros devem ser conhecidos, é necessário para fornecer um valor para cada elemento. Esta tarefa pode tornar-se pesada, se os parâmetros forem muitos.

Assim, é possível colocar alguns parâmetros como opcionais. Nesse caso, um valor defeito será usado.

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

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

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

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

```
Show(false)           // O valor padrão é ignorado
Set(10)                // Apenas o primeiro valor é ignorado
Load()                 // O ficheiro não tem nenhum objeto
SaveImage()            // Usar o valor padrão "BMP"
SaveImage(Image.JPG)  // O valor padrão "BMP" é ignorado
```

Apenas as constantes e as seguintes palavras-chave são permitidas: **null** , **true** e **false**.

Quando um parâmetro tem um valor padrão, todos os outros parâmetros á direita deste devem ser também opcionais. Isto significa que eles também devem ter valores padrão.

O código seguinte não está correcto, porque z não é um parâmetro opcional:

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

5. O RETORNO DE VALORES

Talvez já tenha percebido que não é possível retornar a referência de um **integer**, **float** ou **string**.

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

A primeira instância do numerico inteiro (**Integer**) é local à função e vai ser destruída ao sair da função. Depois, a última linha não retorna a referência do tipo de dado, mas o seu valor (25).

Utilize então o objecto **Value**:

```
function value Obj.Get()
{
    value dia
    dia = new value
    dia.SetInt(25)
    return dia
}
```

6. LOOPS E CONDIÇÕES

Você já viu uma parte da sintaxe da linguagem, mas ainda existe muito mais como o caso das condições e loops.

```
if [expressão]
    ...
else
    ...
end

if [expressão]
    ...
end

if [expressão] or [expressão] and [expressão]
    ...
end

if [expressão] || [expressão] && [expressão]
    ...
end

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

```
while [expressão]
    ...
end

while [expressão]
    ...
    continue // Ir para o próximo ciclo sem a leitura do restante código
    ...
end

while [expressão]
    ...
    Break // Sair do ciclo sem a leitura do restante código
    ...
end
```

```

for [início] ; [condição] ; [incremento]
...
continue
...
break
...
end

```

```

switch [expressão]
  case 2
    ...
    break
  case 0x05
  case 10.6
  case "texto"
    ...
    break
  default
    ...
    break
end

```

Se existem vários **case** especificando o mesmo valor, apenas o primeiro **case** é levado em conta.

7. OS OPERADORES MATEMÁTICOS

Todos nós sabemos, os quatro operadores básicos para executar cálculos ('+', '-', '/', '*'). O objeto **Math** permite efectuar os calculos mais complexos.

Nota: Para strings, o sinal "+" serve para concatenar dois textos num.

O operador de modulo é representado pelo simbolo '%', também disponível no objeto **Math** através de uma função.

Operador	Definição	Exemplo
+	Realiza uma adição	a.Get() + b.Get()
-	Realiza uma subtracção	a.Get() - b.Get()
*	Realiza uma multiplicação	a.Get() * b.Get()
/	Realiza uma divisão	a.Get() / b.Get()
%	Retorna o resto da divisão	a.Get() % b.Get()

Em programação, os seguintes operadores lógicos podem testar as duas expressões do exemplo anterior:

Operador	Definição	Exemplo
==	É igual a	a.Get() == b.Get()
!=	É diferente de	a.Get() != b.Get()
>	É maior do que	a.Get() > b.Get()
<	É menor do que	a.Get() < b.Get()
>=	É maior ou igual a	a.Get() >= b.Get()
<=	É menor ou igual a	a.Get() <= b.Get()
?=	É semelhante a (<i>String.IsSimilar</i>)	a.Get() ?= b.Get()

A linguagem **SECCIA** também fornece operadores bit a bit para os números.

Operador	Definição	Exemplo
	Operador OU	a.Get() b.Get()
&	Operador E	a.Get() & b.Get()
^	OU exclusivo	a.Get() ^ b.Get()
<<	Desloca bits à esquerda	a.Get() << b.Get()
>>	Desloca bits à direita	a.Get() >> b.Get()

Os operadores unários são:

Operador	Definição	Exemplo
~	Inverte todos os bits	~a.Get()
!	Negação lógica	!a
++	Acrescenta um Integer ou Float	a++
--	Remove um Integer ou Float	a--

Os operadores de afetação são:

Operador	Definição	Exemplo
+=	Realiza uma adição e iguala a	a += 2
-=	Realiza uma subtração e iguala a	a -= b.Get()
*=	Realiza uma multiplicação e iguala a	a *= b
/=	Realiza uma divisão e iguala a	a /= 2
%=	Retorna o resto da divisão e iguala a	a %= 10

Os operadores bitwise seguem as seguintes regras:

AND	0000	1111
0000	0000	0000
1111	0000	1111

OR	0000	1111
0000	0000	1111
1111	1111	1111

XOR	0000	1111
0000	0000	1111
1111	1111	0000

8. A ESTRUTURA DE UMA APLICAÇÃO

Uma aplicação não é nada mais do que um objeto chamado **Application**. Para fazer isso deve criar o seu próprio objeto, e apontar **Application** para ser considerado como objeto principal.

```
object Application
{
}

function Application.Constructor()
{
}

function Application.Destructor()
{
}
```

Como irá perceber, a primeira função a ser chamada na sua aplicação será o construtor (**Constructor**). A linguagem **SECCIA** cria assim uma instância do objeto **Application**.

O objeto **Application** pode conter, para além destas duas funções, uma terceira função.

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

function Application.Loop()
{
}
```

A função **Loop** da aplicação só é chamada através da sua ativação pela função **EnableApplicationLoop**. Por defeito, esse método não é ativado. Quando o programa estiver na função **Loop**, a aplicação consome de forma contínua os recursos do computador. Isto corresponde ao mesmo que um loop interno **while**.

É altamente recomendável ativar este método apenas quando precisar dele. Mas é muito útil para desenvolver, por exemplo, um jogo que exija um loop infinito.

*Nota: O objeto **DxEngine** no modo "auto refresh" requer a ativação da função **Loop**.*

9. OS EVENTOS

A programação **Windows**® é baseada no princípio de eventos e mensagens. Este sistema é utilizado pelos objetos. Quando um usuário clicar num botão, a aplicação deve ser capaz de interceptar a mensagem do clique. No software **SECCIA** e sempre por convenção, os eventos são precedidos por 'On'. O botão terá por exemplo um evento **OnClick**.

Devemos diferenciar entre os eventos e as mensagens. Os eventos estão localizados dentro dos objetos como funções, enquanto que as mensagens estão definidas noutros objetos, que as usam para interceptar os eventos.

A definição de um evento é similar à definição de uma função. Um evento pode conter parâmetros e retornar um valor. A única diferença está na palavra-chave **event**.

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

Isto no entanto não é o suficiente, temos de ativar este evento no objeto **Caixa** senão ele nunca será conhecido. Ao contrário das funções, é absolutamente impossível para chamar um evento fora do objeto. É o objeto que necessita de chamar os seus próprios eventos.

Importante: Os eventos geralmente não contêm qualquer código, à exceção de um **return** se a sua definição tem um tipo de retorno.

Voltando ao nosso exemplo, comecemos por adicionar uma chamada ao evento, sempre que as dimensões da caixa mudam os seus valores. Tendo previamente escrito uma função para alterar as dimensões, vamos agora adicionar a chamada ao evento.

```
function Caixa.AtribuiTamanho( integer l, integer a, integer p)
{
    m_largura = l           // Nova largura
    m_altura = a           // Nova altura
    m_profundidade = p    // Nova profundidade
    OnChange()           // Chamada ao evento
}
```

Por agora, concentremo-nos apenas sobre o objeto em questão, mas não se esqueça de que os eventos não são relevantes para o objeto em si, mas para o objeto pai.

Quando se clica num botão, não é ele mesmo quem deve receber a mensagem, mas o objeto que criou o botão. Essa é a razão pela qual os eventos estão vazios e não há nenhuma mensagem associada.

Vamos voltar ao objeto **Application** e adicionar o evento para o nosso objeto **Caixa** na forma de uma mensagem.

```
object Application
{
    Caixa m_caixa new
}

function Application.Constructor()
{
    m_caixa.AtribuiTamanho(20, 5, 10) // Largura, altura e profundidade
}

message Application.Caixa.OnChange()
{
}
```

Em contraste com as funções e os eventos, as mensagens necessitam de mais dados, nomeadamente o nome do objeto onde está o evento.

Resumindo: Existe um evento **OnChange** no objeto **Caixa** e uma mensagem **OnChange** no objeto **Application**. A pergunta que se coloca é qual dos dois é chamado. O evento ou a mensagem? Depende da codificação e não do objeto. Como a variável **m_caixa** é criada no objeto **Application**, é este que irá receber a mensagem. O evento **OnChange** de **m_caixa** nunca será desta forma conhecido. Contudo, se a mensagem **OnChange** não é definida no objeto **Application**, o evento **OnChange** de **m_caixa** será então chamado.

Até agora referimos que a criação de uma instância de um objeto é efectuada pela palavra-chave **new**. No entanto apareceu uma nova palavra-chave: **intercept**.

```
object Application
{
    Caixa m_caixa
}

function Application.Constructor()
{
    m_caixa = new Caixa
    intercept m_caixa
}
```

```
function Application.Destructor()
{
    delete m_caixa
}
```

A palavra-chave **intercept** intercepta todos os eventos numa instância. Se retirar essa linha, o objeto **Caixa** é impedido de enviar qualquer mensagem para o objeto **Application**. A palavra-chave não pode ser aplicada às variáveis geridas pela linguagem, porque, neste caso, as intercepções são tratadas automaticamente.

Permanece um ponto a ser esclarecido relativamente às mensagens. Aquando da definição de uma mensagem, damos o nome do objeto contendo o evento. Mas se criar duas instâncias **Caixa**, é difícil saber de onde a mensagem surgiu. Obviamente que a linguagem oferece uma palavra-chave especial para identificar que chamou a mensagem: **caller**.

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

A palavra-chave **caller** contém a referência para a instância chamada e só pode ser utilizada em mensagens.

10. HERANÇA

Herança é uma forma de programação orientada para objetos que vai mais longe. Ela permite criar um objeto derivado de outro objeto herdando por sua vez todos os membros do mesmo (funções, variáveis e constantes).

Importante: Nem os eventos, nem as mensagens são herdados. O mesmo se passa com as funções **Construtor** e **Destructor**.

Graças ao nosso exemplo, e partindo do início, vamos refletir melhor sobre os interesses da herança. Temos um objeto **Caixa** que representa uma caixa de papelão. Estamos agora empenhado em colocar nessa caixa, dois tipos de brinquedos: bolas e cubos de plástico. Essas bolas e cubos não são todas do mesmo tamanho. Existe no entanto um ponto comum entre esses dois objetos, que são o facto de serem brinquedos devem ser colocados na caixa de papelão. Assim podemos criar um objeto **Brinquedo** mais dois objetos derivados deste : **Bola** e **Cubo**.

Em termos de programação temos então o seguinte código:

```
object Caixa
{
    Array m_brinquedos new // Array com todos os brinquedos
}

object Brinquedo
{
    Integer m_peso // Peso em gramas
}
```

```

object Bola from Brinquedo
{
    Integer m_raio           // Raio em centímetros
}

object Cubo from Brinquedo
{
    Integer m_largura       // Largura em cm
}

```

Temos quatro itens, incluindo duas diferentes derivações do objeto **Brinquedo**. A palavra-chave **from** é utilizada para indicar de quem um objeto é derivado.

Os objectos **Bola** e **Cubo** possuem uma variável inteira (**Integer**) do objeto **Brinquedo** por derivarem deste. Essa variável é o peso do brinquedo (**m_peso**). Isto evita ter que criar duas variáveis. Para simplificar, removemos as dimensões do objeto **Caixa**.

Importante: Se um objeto já tem uma variável membro derivada do objeto base e, em seguida, criar uma local com o mesmo nome estas duas podem coexistir no objeto. No entanto, é impossível o acesso exterior para a variável base derivada através do objeto.

Apareceu no objeto **Caixa** um novo tipo de objeto que nunca se falou até então. Trata-se do **Array**, um objeto para gerir tabelas de diferentes dimensões. Neste exemplo, os brinquedos incluídos na tabela são os brinquedos da caixa de papelão.

Importante: Os **Arrays** armazenam apenas as referências das instâncias. Por isso não é possível usar os objetos **Integer**, **float** e **String** porque eles não retornam referências. Neste caso, é necessário utilizar os objetos **IntegerArray**, **FloatArray** e **StringArray**.

O próximo passo é definir as funções que precisamos:

- **Brinquedo:** **AtribuirPeso()**
- **Bola:** **AtribuirTamanho()** e **CalcularVolume()**
- **Cubo:** **AtribuirTamanho()** e **CalcularVolume()**
- **Caixa:** **AdicionarBrinquedo()** e **RetornarPesoBrinquedos()**

```

function Brinquedo.AtribuirPeso (Integer peso)
{
    m_peso = peso
}

```

```

function Bola.AtribuirTamanho(Integer raio)
{
    m_raio = raio
}

function Integer Bola.CalcularVolume()
{
    return m_raio * m_raio * m_raio * PI * 4/3
}

```

```

function Cubo AtribuirTamanho(Integer tamanho)
{
    m_tamanho = tamanho
}

function Integer Cubo. CalcularVolume()
{
    return m_tamanho * m_tamanho * m_tamanho
}

```

```

function Caixa.AdicionarBrinquedo(Brinquedo brinquedo)
{
    m_brinquedos.Add(brinquedo)
}

function Caixa.Destructor()
{
    m_brinquedos.DeleteAll()
}

```

A função **RetornarPesoBrinquedos** vai retornar o peso total dos brinquedos na caixa de papelão. Exige uma codificação mais elaborada, porque precisamos fazer um loop para obter o peso de cada brinquedo.

```

function Integer Caixa.RetornarPesoBrinquedos()
{
    Integer i new
    Integer peso new
    Brinquedo brinquedo

    peso = 0
    for i = 0; i < m_brinquedos.GetCount(); i++
        brinquedo = m_brinquedos.Get(i)
        peso.Add(brinquedo.m_peso)
    end

    return peso
}

```

A função **Get** do objeto **Array** retorna a referência para a instância armazenada no índice especificado.

Na aplicação vamos criar dois **Brinquedos**: Uma **Bola** e um **Cubo**. A função **AdicionarBrinquedo** é definida por um parâmetro do tipo **Brinquedo**. Isto não coloca qualquer problema, porque um objeto **Bola** também é um **Brinquedo**.

```

Bola minhaBola
minhaBola = new Bola
minhaBola.AtribuirPeso(50)
minhaBola.AtribuirTamanho(2)

Cubo meuCubo
meuCubo = new Cubo
meuCubo.AtribuirPeso(100)

```

```
meuCubo.AtribuirTamanho(3)

Caixa minhaCaixa new
minhaCaixa.AdicionarBrinquedo(minhaBola)
minhaCaixa.AdicionarBrinquedo(meuCubo)

Integer peso new
peso = minhaCaixa.RetornarPesoBrinquedos()
```

O peso total dos brinquedos será de 150 gramas (50 + 100).

A herança ainda pode ir mais longe. A linguagem é capaz de derivar objetos em vários níveis. Nada impede-nos de definir por exemplo um outro objeto, que deriva do objeto **Bola**. Este novo objeto vai herdar ambos os membros dos objetos **Bola** e **Brinquedo**.

Vamos agora dar a conhecer como é que é possível aceder a uma variável ou constante definida em dois lugares. Para fazer isso, adicionamos um objeto **Camiao** (Truck) derivado do objeto base **Brinquedo** com uma variável também chamada de **m_peso**.

```
object Camiao from Brinquedo
{
    Integer m_peso new
}

function Camiao.Constructor()
{
    m_peso // Acesso à variável do objeto Camiao
    Brinquedo.m_peso // Acesso à variável do objeto Brinquedo
}
```

Para aceder a **m_peso** do objeto **Brinquedo**, é necessário adicionar o nome dele (refira-se objeto) seguido por um ponto. Isto é por uma razão simples: A versão **m_peso** do objeto **Camiao** coloca em cache a do objeto **Brinquedo**.

11. POLIMORFISMO

O polimorfismo é a continuidade da herança para proporcionar ainda mais flexibilidade e poder à codificação.

Vimos anteriormente que era possível ter uma variável membro já existente no objeto base. Neste caso, duas variáveis coexistem. Mas o resultado não é o mesmo se estivermos a falar de funções.

Vamos continuar a nossa abordagem, acrescentando uma nova função ao objeto **Brinquedo**.

```
function Integer Brinquedo.RetornarVolume()
{
    return 0
}
```

Agora, passa a existir a função **RetornarVolume** tanto no objeto base **Brinquedo** como em ambos os objetos derivados: **Bola** e **Cubo**.

Considere o seguinte código:

```
Brinquedo meuBrinquedo1 new // Variável instanciada do objeto Brinquedo
Bola minhaBola new // Variável que contém uma instância Bola
meuBrinquedo1.RetornarVolume() // Chamada à função do objeto Brinquedo
minhaBola.RetornarVolume() // Chamada à função do objeto Bola

Brinquedo meuBrinquedo2 // Variável não instanciada
meuBrinquedo2 = minhaBola // Cópia da referência
meuBrinquedo2.RetornarVolume() // Chamada à função do objeto Bola
```

Podemos pensar que a variável **meuBrinquedo2** contém uma referência da instância do objeto **Brinquedo**, mas ela realmente contém é uma referência a uma instância do objeto **Bola**. Como resultado, a função virtual **RetornarVolume** está relacionada com a função **RetornarVolume** do objeto **Bola**, e não com a do objeto base **Brinquedo**.

Nota: Com a herança há mais certeza sobre o tipo de instânciação. Devemos, portanto, diferenciar o tipo da variável e o tipo da instância.

Outra questão surge quando estivermos numa função virtual. É possível chamar uma função que está no objeto base estaticamente? Claro. Caso contrário o código seguinte poderia representar um grave problema.

```
function Integer Derivado.RetornarIdade()
{
    RetornarIdade()
}
```

Aliás, a função **RetornarIdade** é chamada por ela própria indefinidamente e nunca poderemos chamar a função **RetornarIdade** do objeto base.

Para ter acesso apenas à função base do objeto, basta adicionar a palavra-chave **Base** que se refere ao seu nome (refira-se objeto). O exemplo seguinte ilustra isso mesmo.

```
function Integer Derivado.RetornarIdade()
{
    Base.RetornarIdade()
}
```

Podemos usar a palavra-chave **Base** quantas vezes forem necessárias até chegarmos à função do objeto que pretendemos utilizar.

12. AS EXTENSÕES DA LINGUAGEM SECCIA

13. OS COMANDO DE COMPILAÇÃO

Durante a construção do ficheiro executável ou aquando de testes na aplicação, a linguagem **SECCIA** é capaz de excluir linhas de código, utilizando as palavras-chave :

`_ifdef, _ifndef, _else, _end`

Essas palavras-chave não fazem parte do código da sua aplicação no momento da sua execução.

```
object Application
{
}

function Application.Constructor()
{
    _ifdef VERSION_DEMO
        Demo()
    _end
}

_ifdef VERSION_DEMO
    function Application.Demo()
    {
        MessageBox("DEMO")
    }
_end
```

Se **VERSION_DEMO** tiver sido definido, o código será integrado no programa. Caso contrário a função **Demo** não existe e portanto não pode ser chamada. Por isso, é necessário também excluir a chamada à função no **Constructor**.

O resultado final vai depender das definições, mas no nosso caso, pode haver dois possíveis cenários:

```
object Application
{
}

function Application.Constructor()
{
}
```

```

object Application
{
}

function Application.Constructor()
{
    Demo()
}

function Application.Demo()
{
    MessageBox("DEMO")
}

```

14. A NOMENCLATURA DE DESENVOLVIMENTO NA LINGUAGEM SECCIA

A linguagem é sensível, ou seja, faz a distinção entre maiúsculas e minúsculas. Esta característica é essencial para manter o código com um certo rigor. Sejam claros, não é para criar duas variáveis locais numa função com o mesmo nome. Seria uma catástrofe para a compreensão do código. As variáveis devem conter nomes muito explícitos. No entanto, servirá para distinguir os diferentes tipos de elementos.

Nota: Vocês são livres para colocarem os nomes das variáveis como bem entenderem, isto são apenas conselhos práticos.

Começamos por listar todos os tipos de elementos existentes: as constantes, os objetos, as funções, as variáveis membro e as variáveis locais. As palavras a azul e cinzento são palavras-chave reservadas pela linguagem.

Na maior parte das linguagens, as constantes são escritas em maiúsculas. Nós usamos o mesmo formato.

```

constant DIFICULDADE_FACIL 0
constant DIFICULDADE_DIFICIL 1

```

Como os objetos de **SECCIA** começam com a primeira letra em maiúscula, nós faremos o mesmo com os nossos objetos.

```

object Integer
object Caixa
object CaixaGrande

```

As funções, eventos e mensagens possuem sempre parênteses e não possuem restrições quanto ao seu nome. No entanto devemos tal como nos objetos começar o nome com a primeira letra em maiúsculas.

```

function Caixa.Selecionar()
function Caixa.RemoveCor()

```

Para ser mais fácil a identificação dos eventos foi adicionada a palavra 'On'.

```

event Caixa.OnOpen()

```

Uma variável local pode ter o mesmo nome de uma variável membro pois não existe nenhum conflito. A variável membro torna-se inacessível se não utilizarmos as funções. Uma simples convenção refere que devemos escrever 'm_' antes do nome de cada variável membro.

```
object Caixa
{
    Integer m_TotalBrinquedos // Membro
}

function Caixa.Constructor()
{
    Integer totalBrinquedos // Variável local
}
```

15. CASO PRÁTICO : AS CONSTANTES

As utilização de constantes é essencial num programa porque nunca deixa instruções sem valor associado:

```
numeroDias = mesCorrente * 30 + diaCorrente
```

Podemos calcular o número de dias de uma data com base em 30 dias por mês. No entanto o número 30 não é muito explícito. Uma constante seria aqui muito mais apropriada.

```
constant DIAS_DO_MES 30
numeroDias = mesCorrente * DIAS_DO_MES + diaCorrente
```

Esta forma também facilita uma posterior alteração do valor.

As constantes hexadecimais proporcionam uma funcionalidade muito interessante. Peguemos num exemplo completamente novo. Vamos criar um objeto **Pizza** contendo informação acerca dos seus ingredientes. Desta forma temos:

```
object Pizza
{
    Integer m_queijo new // true, false
    Integer m_tomate new // true, false
    Integer m_anchovas new // true, false
}
```

Todas as variáveis têm o valor **false** podendo passar posteriormente a **true** e podem ser editadas por três funções existindo também três funções de leitura.

```
function Pizza.AdicionarQueijo(Integer possui)
{
    m_queijo = possui != 0 // Converte um inteiro para verdadeiro ou falso
}

function Integer Pizza.VerificarPossuiQueijo()
{
    return m_queijo
}
```

Se a nossa pizza possui três ingredientes podemos escrever o código seguinte.

```
Pizza pizza new
pizza.AdicionarQueijo(true)
pizza.AdicionarTomate(true)
pizza.AdicionarAnchovas(true)
```

A seguir é necessário validar os diversos ingredientes existentes na pizza. Digamos por exemplo que necessitamos de saber se a pizza possui anchovas. Neste caso a operação é simples.

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

Para verificar se a nossa pizza possui os três ingrediente são necessárias três condições.

```
if pizza.VerificarPossuiQueijo() && pizza.VerificarPossuiTomate() && pizza.VerificarPossuiAnchovas()
end
```

Agora tenhamos em conta as constantes no nosso exemplo. Em vez de termos três variáveis diferentes para os nossos ingredientes, nós temos apenas uma.

```
object Pizza
{
    constant QUEIJO 0x01
    constant TOMATE 0x02
    constant ANCHOVAS 0x04
    Integer m_tipo_ingrediente new
}
```

Uma quarta constante teria o valor 0x08, uma quinta o valor 0x10, uma sexta o valor 0x20 e assim por diante...

```
function Pizza.Set(Integer tipoIngrediente)
{
    m_tipo_ingrediente = tipoIngrediente
}

function Integer Pizza.Get()
{
    return m_tipo_ingrediente
}
```

Como a nossa pizza necessita de três ingredientes refaça o código do exemplo acima mencionado.

```
Pizza pizza new
pizza.Set(Pizza.QUEIJO | Pizza.TOMATE | Pizza.ANCHOVAS)

if pizza.Get() & Pizza.ANCHOVAS
end
```

```
if pizza.Get() & (Pizza.QUEIJO | Pizza.TOMATE | Pizza.ANCHOVAS)
end
```

Para simplificar ainda mais, podemos adicionar duas constantes para definir a especialidade da pizza ao combinar os ingredientes.

```
object Pizza
{
    constant QUEIJO 0x01
    constant TOMATE 0x02
    constant ANCHOVAS 0x04
    constant MARGARIDA QUEIJO | TOMATE
    constant NAPOLITANA MARGARIDA | ANCHOVAS
    Integer m_tipo_ingrediente new
}
```

Agora podemos verificar se a pizza é do tipo napolitana através da variável `m_tipo_ingrediente`.

```
if pizza.Get() == Pizza.NAPOLITANA
end
```

De notar a sutileza da instrução seguinte que nos permite saber se a pizza é realmente uma pizza napolitana com os seus respectivos ingredientes e ainda se possui ingredientes adicionais.

```
if pizza.Get() & Pizza.NAPOLITANA
end
```

Finalmente o uso das constantes no comando `switch`, torna possível o respectivo teste das diversas especialidades da pizza:

```
switch pizza.Get()
case Pizza.MARGARIDA
    break
case Pizza.NAPOLITANA
    break
end
```

16. O EDITOR DE CÓDIGO

O editor de código é capaz de mostrar automaticamente os membros de um objeto (variáveis e funções) enquanto o programador escreve o seu código. Esta forma facilita o desenvolvimento tornando-o mais agradável e fácil de utilizar.

Este editor mostra não só o que foi referido no parágrafo anterior mas também os membros (variáveis e funções) do objeto base. Para diferenciá-los é mostrada uma imagem diferente antes do nome do membro.

É possível incluir no objeto constantes, variáveis e funções que não são visíveis exteriormente através da palavra-chave `hidden`.

```
object MeuObjecto
```

```
{  
    constant PUBLICO 1  
    constant PRIVADO 2 hidden  
  
    Disk m_publico1  
    Disk m_publico2 new  
    Disk m_privado1 hidden  
    Disk m_privado2 new hidden  
    Disk m_privado3 hidden new  
}
```

```
function MeuObjecto.MinhaFuncaoPrivada() hidden  
{  
}  
}
```

Importante: Os elementos escondidos continuam a poder ser acedidos dentro do objeto.

17. AS PALAVRAS-CHAVE DA LINGUAGEM SECCIA

A lista completa das palavras-chave da linguagem:

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