Introducción a la Programación Orientada a Objetos con FiveWin
Parte I
Por James Bott

Yo sé que la mayoría de ustedes temen a muerte a la OOP (Object-Oriented Programming, es decir, Programación Orientada a Objetos), pero el concepto básico de la OOP realmente no es para nada complicado, aunque el auténtico "ah ha" puede no llegar antes de 6 meses o más. ¿ Pero, primero, por qué necesitamos la OOP? La respuesta simplemente es que el software se está volviendo demasiado complejo para escribirse en la antigua forma procedural. Se hace muy difícil interpretar procedimientos de miles de líneas. Claro que puede dividirlos en llamadas de funciones, pero entonces termina teniendo que pasar muchas variables, y puede tener que pasar un array con todos los resultados, aumentando la complejidad.

Empecemos con algo que sabemos y entonces lo trabajamos con la OOP. Como programadores de Clipper tenemos las llamadas "variables estáticas". Éstas pueden declararse al principio de un archivo PRG y pueden usarse a lo largo de todo el PRG. Esto alivia el pasaje de variables y evita el uso del array antes mencionado.
Nosotros podríamos declarar 50 estáticas y usarlas y cambiarlas en cualquier UDF en el mismo PRG. Esto ayuda, pero no nos da todas las ventajas de la OOP.

Necesitamos definir algunas condiciones de la OOP. Primero nosotros tenemos: “clase” y “objeto”. Una clase es como un plano y un objeto es una instancia de una clase, del mismo modo que una casa es una instancia de un plano. Hay sólo una clase (definiendo un objeto particular) pero usted puede tener muchas instancias de esa clase (objetos).
Usando la OOP podemos crear variables de instancia que son similares a las estáticas. Así, podemos crear una clase TPerson (persona), y podemos declarar las variables de instancia para el nombre, la fecha de nacimiento (DOB, Date of birth en inglés) etc.
Lo hacemos de esta forma:

```
create class TPerson
   var cName
   var dDOB
endclass
```

Usando la OOP también podemos crear métodos que son similar a las funciones estáticas pero con más flexibilidad (esto lo veremos más tarde). Agreguemos un método edad a nuestra clase TPerson. Para simplificar, calculémosla en días:

```
create class TPerson
   cName
   dDOB
   method age
endclass

method age
return date()-::dDOB
```

[Nota que sólo una clase puede definirse por PRG]

ENCAPSULADO

Un objeto debe ser como una "Caja Negra". En otros términos debe ser auto-contenido y el usuario tendrá necesidad de conocer la interfaz del objeto. Note que aquí el usuario será otro programador, no un usuario final. Así, para nuestra clase TPerson, los usuarios sólo necesitan saber llamar el método edad para obtener la edad de la persona. No necesitan saber cómo el método calcula la edad. Este concepto se llama encapsulado.

Siguiendo esta filosofía, nosotros necesitamos hacer nuestro código como piedra, sólido y tan "a prueba de bombas" como sea posible. Nosotros debemos usar solo variables locales y estáticas y cualquier cambio hecho al entorno de Clipper debe restaurarse a su estado original. Cada clase tiene un método llamado "New" (nuevo) qué inicializa el objeto. Allí nosotros asignamos los valores predeterminados a todas las variables de instancia y posiblemente pasemos algunos parámetros de arranque.

```
method new(cName,dDOB)
   default cName:="",dDOB:=ctod(" / / ")
   ::cName:=cName
   ::dDOB:=dDOB
return self
```

El comando DEFAULT es un práctico comando agregado por FiveWin. Lo usamos para asignar los valores predeterminados a todos los parámetros pasados disponibles. Noten que cName no es el mismo que ::cName. Los dos puntos dobles (::) es solo un atajo para Self, Así ::cName preprocesa self:cName. Self refiere al objeto TPerson así ::cName es la variable perteneciente al objeto TPerson, en tanto que cName es el parámetro pasado.

Bien , la primera cosa que hacemos cuando queremos usar un objeto TPerson, es crear un objeto TPerson vacío llamando el método new() (nuevo). Nosotros asignamos este objeto a una variable, oPerson en este caso,:

```
oPerson:=TPerson():new()
```

O podemos pasar los parámetros al mismo tiempo:

```
oPerson:=TPerson():new("John Smith", ctod("01/01/50"))
```

Note arriba, que al definir la nueva clase devolvemos SELF. Esto es para que podamos usar la sintaxis de una sola línea y ni siquiera asignar el objeto a una variable.
Por ejemplo:

```
TPerson():new(,ctod("01/01/50")):age()
```

Esto devolverá la edad de una persona nacida el 1 de enero de 1950. Estamos devolviendo SELF desde el nuevo método que pasa entonces al método de edad. Es buena idea devolver SELF de cualquier método que no necesite devolver nada más,
por otro lado, podemos tener múltiples instancias de la clase TPerson al mismo tiempo.

```
oFather:=TPerson():new("John Smith",ctod("01/01/50"))
oMother:=TPerson():new("Mary Smith",ctod("05/02/55"))
```

Ahora nosotros podemos encontrar el nombre de cada uno de ellos:

```
msgInfo( oFather:name )

msgInfo( oMother:name )
```

Y aún más interesante, podemos saber la edad:

```
msgInfo( oFather:age() )

msgInfo( oMother:age() )
```

Si ayuda, usted puede pensar en un objeto como un recipiente que no sólo contiene datos (variables de instancia), sino código (los métodos). Ahora, aquí está una característica muy poderosa. Podemos pasar los objetos enteros a funciones u a otros objetos. Por ejemplo, nosotros podríamos pasar oFather a un objeto informe y podríamos imprimir su nombre, DOB, y edad desde dentro del objeto informe. Ésta es simplemente una prueba del poder de OOP.

HERENCIA

La herencia es uno de las características más útiles y poderosas de la OOP. Una clase puede heredar todas las variables y métodos de otra. Así, creemos una nueva clase TCust (cliente, Customer en inglés), que herede de nuestra clase TPerson (asumamos que todos los clientes son personas y no compañías). Aquí queremos agregar un par de nuevas variables de instancia, cCustNo (número de cliente), y nBalance (el saldo actual adeudado). También necesitaremos inicializarlos.

```
class TCust from TPerson
   var cCustNo
   var nBalance
   method new
endclass

method new(cName,dDOB,cCustNo,nBalance)
   default cCustNo:="",nBalance:=0
return super:new(cName,dDOB)
```

Nota que referimos al método New() de la clase madre como el super:new(), así solo pasamos las variables necesarias al método de la clase madre para inicializarlos.

Ahora nosotros podemos inicializar un objeto cliente:

```
oCust:=TCust():new("John Smith",ctod("01/01/50"),"1111","500.00")
```

Entonces nosotros podemos encontrar su nombre, edad, y balance:

```
msgInfo( oCust:name )
msgInfo( oCust:age() )
msgInfo( oCust:nBalance )
```

Aquí, otra característica a considerar. Los cambios a las clases madres se reflejan en las hijas. Así si usted descubriera un error en el método edad de la clase TPerson, usted podría arreglarlo y también se reflejaría en la clase TCust o usted podría agregar nuevas variables o métodos a la clase TPerson y ellas se heredarían automáticamente a la clase TCust.

POLIFORMISMO

Note que hemos usado la sintaxis oPerson:age() y oCust:age(). Tenemos un método edad en ambas clases. En este caso una clase heredó el método de la otra, pero también podríamos tener un método edad en cualquier otra clase no relacionada, digamos una clase automóvil por ejemplo. La habilidad de usar la misma sintaxis en clases diferentes se llama el poliformismo (significando muchas formas). No sólo ayuda a simplificar la comprensión del código sino que además le permite crear código más genérico, como pasando cualquier tipo de objeto a un objeto Reporte y llamando el método de impresión simplemente, por ejemplo oObject:print().

MODULARIDAD

El concepto de modularidad refiere al poder descomponer el código en piezas de código mejores (módulos) a fin de ganar en legibilidad y facilidad de revisión. Así en lugar de sólo un método impresión en una clase informe, usted podría dividirlo en título, cuerpo, y métodos pie de página. Igualmente, el método cuerpo podría dividirse en los títulos de la columnas, línea de ítems, y método pie de página de columna.

SINTAXIS DE LAS CLASES DE FIVEWIN

Algunos de ustedes pueden estar preguntándose como el lenguaje de FiveWin ha evitado toda esta sintaxis de la OOP. FW usa el preprocesador simplemente para convertir la sintaxis de OOP a la de estilo de comando. Para nuestro objeto TPerson, nosotros podríamos escribir:

```
DEFINE PERSON oFather NAME "John Smith" DOB ctod("01/01/59")
```

Y el preprocesador lo traduciría simplemente a:

```
oFather:=TPerson():new("John Smith",ctod("01/01/50"))
```

Uno puede ver inmediatamente que la sintaxis de comando es mucho más fácil de recordar y escribir. Nota que el preprocesador es normalmente lo suficientemente listo para manejar las cláusulas puestas en diferente orden así que para este caso podrías escribir:

```
DEFINE PERSON oFather DOB ctod("01/01/59") NAME "John Smith"
```

Y el preprocesador aún lo interpretaría correctamente.

RESUMEN

En parte I hemos discutido los elementos esenciales de sintaxis de OOP creando clases simples. En parte II discutiremos problemas de OOP más avanzados que incluyen más sintaxis y algún análisis y conceptos de diseño.

 Escrito por James Bott, jbott@compuserve.com http://ourworld.compuserve.com/homepages/jbott,