ABAP orientado a objetos

Versión 1.0
22 de Marzo del 2002
Manuel Fernández Arguelles AG

LENGUAJE ABAP ORIENTADO A OBJETOS

¿QUÉ ES LA ORIENTACIÓN A OBJETOS?

La programación orientada a objetos es un método de desarrollo de software basado en el comportamiento real de los objetos en el mundo real. Se pretende desarrollar componentes de software que se comporten como los objetos reales a los que representan.
La orientación a objetos es una técnica usada en muchos lenguajes de programación los cuales comparten una terminología usada universalmente.
En esta sección se realizará una primera visión de conjunto de estos términos para en secciones posteriores adentrarse en la implementación de estos conceptos en el lenguaje ABAP.

Objetos

Un objeto es únicamente una porción de código fuente que contiene datos y proporciona servicios. Los datos constituyen los *atributos* del objeto. Los servicios que proporciona el objeto se conocen como *métodos* y se asemejan en su funcionamiento a las funciones. Normalmente los métodos operan con los datos *privados* del objeto, esto es, con datos que son sólo *visibles* para los métodos del objeto. De esta manera, los atributos de un objeto no pueden ser cambiados directamente por el usuario del objeto, sólo pueden ser cambiados por los métodos de ese objeto. Así se garantiza la consistencia interna del objeto.

Clases

Una clase es una entidad teórica que describe el comportamiento de un objeto. Desde un punto de vista meramente técnico, un objeto es una *instancia* en tiempo de ejecución de una clase. En principio se pueden crear cualquier número de objetos basados en una única clase. Cada instancia de una clase (objeto) tiene su propia identidad y su propio conjunto de valores para sus atributos. Dentro de un programa un objeto es identificado por su referencia, la cual le proporciona un nombre que define inequivocamente al objeto y permite acceder a sus métodos y atributos.

Propiedades de los objetos

En la programación orientada a objetos, los objetos tienen normalmente las siguientes propiedades:

- Encapsulación – Los objetos restringen la visibilidad de sus recursos (atributos y métodos) al resto de usuarios. Cada objeto posee una *interface* que determina la manera de interactuar con él. La implementación del objeto (su interior) es encapsulada, lo que quiere decir que desde fuera el objeto es invisible, simplemente se usa.
- Polimorfismo – El polimorfismo quiere decir que métodos que se llaman exactamente igual pueden comportarse de manera distinta en clases diferentes. La orientación a objetos tiene unas estructuras llamadas interfaces que permiten acceder a métodos con el mismo nombre en diferentes clases. Dentro de cada clase particular se puede redifinir el método obteniendo distintos métodos con el mismo nombre. Asi es que un método no se define exactamente con su nombre, si no con su nombre y el nombre de la clase a la que pertenece.
- Herencia – Se pueden utilizar clases existentes para originar nuevas clases. Las nuevas clases originadas heredan los datos y los métodos de la *superclase*. De cualquier manera, se pueden sobreescibir los métodos existentes, incluso añadir métodos nuevos.

Usos de la orientación a objetos

Las principales ventajas de la programación orientada a objetos son:

- Sistemas de software muy complejos se vuelven mucho más simples de comprender debido a que la orientación a objetos proporciona una representación mucho más cercana a la realidad que otras técnicas de programación.
- En un sistema correctamente diseñado con orientación a objetos es posible realizar cambios al nivel de la clases, sin tener que realizar cambios en ningún otro punto del sistema. Esto reduce significativamente el costo total del mantenimiento necesario.
- A través del polimorfismo y la herencia es posible la reutilización de componentes individuales.
- La cantidad de trabajo en revisión y mantenimiento del sistema se reduce debido a que muchos problemas pueden ser detectados y corregidos en la fase de diseño.

Para conseguir estos objetivos necesitamos:

- Lenguajes de programación orientados a objetos – Las técnicas de programación orientadas a objetos no utilizan necesariamente un lenguaje orientado a objetos, aunque la eficiencia del desarrollo depende directamente de la utilización o no de un lenguaje orientado a objetos.
- Herramientas orientadas a objetos – Estas herramientas permiten crear programas orientados a objetos en lenguajes orientados a objetos. Permiten además diseñar y almacenar los objetos desarrollados y las relaciones entre ellos.
- Diseño orientado a objetos – El diseño orientado a objetos de un sistema de software es, de todos los requerimientos para conseguir las ventajas arriba enumeradas, el más importante, el que más tiempo consume y el más dificil de llevar a cabo.

¿QUÉ SON LOS OBJETOS ABAP (ABAP OBJECTS)?

Este es un nuevo concepto introducido en el release 4.0 que tiene dos significados, por un lado se refiere al entorno de ejecución ABAP y por otro a la extensión orientada a objetos del lenguaje ABAP

El entorno de ejecución

El nombre de *ABAP objects* para todo el entorno de ejecución ABAP quiere indicar el objetivo aún no totalmente desarrollado de adaptar SAP completamente a la orientación a objetos. El *ABAP Workbench* permite crear objetos del *R/3 Repository* tales como programas, objetos de autorización, objetos de bloqueo, etcetera. El *Business Object Repository* (BOR) permite crear SAP Business objects. Hasta ahora las técnicas de orientación a objetos se habían usado exclusivamente en el diseño, pero desde el release 4.0 el lenguaje ABAP es ya un lenguaje orientado a objetos.

Extensión orientada a objetos de ABAP

ABAP Objects es asimismo un conjunto de sentencias orientadas a objetos que han sido introducidas dentro del lenguaje ABAP. Esta extensión se cimenta en el lenguaje ya existente, siendo compatible con él. Se pueden usar objetos en programas existentes, de la misma manera que se pueden usar sentencias ABAP convencionales en programas ABAP orientados a objetos.
El resto del lenguaje ABAP está creado desde un principio orientado a una programación estructurada, en la cual los datos se almacenan de manera estructurada en tablas en la base de datos y los programas mediante funciones acceden a estos datos y trabajan con ellos.

PASO DE LAS FUNCIONES A LOS OBJETOS

Los objetos son el centro de cualquier modelo orientado a objetos. Los objetos contienen atributos (datos) y métodos (funciones). Uno de sus principales objetivos es el suministar al desarrollador de software una forma de trabajo en la cual poder examinar un problema real y poder proporcionar una solución individualizada al problema. En el entorno de los negocios podrían ser objetos las entidades *Cliente*, *Factura*, etcetera.
Desde el Release 3.1 en adelante, el Business Object Repository (BOR) contiene ejemplos de tales objetos (transacción SW02).
Lo más parecido a los objetos que tenía ABAP eran los módulos de funciones y los grupos de funciones. Supongamos que tenemos un grupo de funciones para procesar pedidos. Los atributos de un pedido son los datos globales del grupo de funciones, mientras que los módulos de funciones son las acciones que manipulan los datos, o sea los métodos. Esto quiere decir que los datos reales del pedido están encapsulados en el grupo de funciones y no se puede acceder directamente a ellos, sólo mediante los módulos de funciones. De esta manera se garantiza la consistencia de los datos.
Cuando se ejecuta un programa ABAP, el sistema inicia una nueva sesión interna que tiene una zona de memoria en la cual reside el programa ABAP y sus datos asociados. Cuando el programa llama a un módulo de funciones, una instancia del grupo de funciones más sus datos es cargado en el área de memoria de la sesión interna. Un programa puede cargar distintas instancias llamando a módulos de funciones de diferentes grupos de funciones.
La instancia de un grupo de funciones en el área de memoria de la sesión interna representa prácticamente el concepto de objeto. Cuando se llama al módulo de funciones, el programa que llama usa la instancia del grupo de funciones basada en su descripción en la biblioteca de funciones. El programa no puede acceder a los datos en el grupo de funciones directamente pero si a través del módulo de funciones. El módulo de funciones y sus parámetros son la interface ente el grupo de funciones y el usuario.
La principal diferencia ente la verdadera orientación a objetos y los grupos de funciones es que mientras que un programa puede trabajar simultaneamente con varios grupos de funciones, no puede hacerlo con varias instancias de un mismo grupo. Si un programa quiere procesar varios pedidos a la vez tendría que

adaptar el grupo de funciones para incluir una administración de instancias, usando por ejemplo, números que diferencien las instancias. En la práctica, esto es muy complicado de realizar. Por esto, los datos son almacenados en el programa y los módulos de funciones son llamados para trabajar con ellos (programación estructurada). Un problema es por ejemplo que todos los usuarios de un módulo de funciones deben usar las mismas estructuras de datos así como el propio grupo de funciones. El cambiar la estructura de interna de datos de un grupo de funciones afecta a muchos usuarios, sin poder predecir las posibles implicaciones del cambio. El único modo de evitar esto es mediante las interfaces, con una técnica que garantice que las estructuras internas de las instancias permanecerán ocultas, permitiendo cambiarlas mas tarde sin causar ningún problema
Esto se consigue con la orientación a objetos. ABAP Objects permite definir datos y funciones en clases en lugar de en grupos de funciones. Usando clases, un programa ABAP puede trabajar con cualquier número de instancias (objetos) basados en la misma plantilla. En lugar de cargar en la memoria una única instancia de un grupo de un grupo de funciones implícitamente cuando se llama al módulo de funciones, el programa ABAP ahora puede generar las instancias de la clase explícitamente usando la nueva sentencia ABAP **CREATE OBJECT**. Cada instancia representa a un único objeto, y se puede acceder a cada una mediante su referencia. La referencia del objeto es lo que permite a un programa ABAP acceder a la interface de la instancia

EJEMPLO

En el siguiente ejemplo vemos la orientación a objetos de un sencillo grupo de funciones como es el caso de un contador:
Supongamos que tenemos el grupo de funciones COUNTER.

```
FUNCTION-POOL contador.
  DATA: cont TYPE i.

  FUNCTION fijar_contador.
* Interface local → importing value (fijar_valor)
    cont = fijar_valor.
  ENDFUNCTION.

  FUNCTION incrementar_contador.
    ADD 1 TO cont.
  ENDFUNCTION.

  FUNCTION obtener_contador.
* Interface local → exporting value (obtener_valor)
    obtener_valor = cont.
  ENDFUNCTION.
```

El grupo de funciones tiene un campo entero llamado cont y tres módulos de funciones, fijar_contador, incrementar_contador y obtener_contador que trabajan con este campo. Dos de los módulos de funciones tienen parámetro input y output. Son los módulos que conforman la interface del grupo de funciones. Cualquier programa ABAP puede trabajar con este grupo de funciones, por ejemplo:

```
DATA numero TYPE i VALUE 5.

CALL FUNCTION 'FIJAR_CONTADOR'
     EXPORTING fijar_valor = numero.
DO 3 TIMES.
  CALL FUNCTION 'INCREMENTAR_CONTADOR'.
ENDDO.
CALL FUNCTION 'OBTENER_CONTADOR'
     IMPORTING obtener_valor = numero.
```

Después de que esta sección del programa haya sido ejecutada, la variable numero tendrá el valor 8. El programa no puede acceder por sí mismo al campo cont del grupo de funciones. Las operaciones sobre este campo están encapsuladas en el módulo de funciones. El programa sólo puede comunicarse con el grupo de funciones mediante la llamada a los módulos de funciones.

CLASES

Las clases son las plantillas de los objetos. A la inversa, podemos decir que el tipo de un objeto es el mismo que el de su clase. Una clase es la descripción abstracta de un objeto. También podemos decir que una clase es un conjunto de instrucciones que tienen como objetivo construir un objeto. Los atributos de los objetos están definidos por los componentes de la clase (atributos, métodos y eventos), que son los que describen y controlan el comportamiento de los objetos.

Clases locales y globales

Las clases en ABAP Objects se pueden declarar bien globalmente o bien localmente. Las clases globales se definen en el generador de clases (transacción SE24) en el ABAP Workbench. Estas clases son almacenadas en class pools en la librería de clases en el R/3 Repository. Todos los programas ABAP en un sistema R/3 pueden acceder a las clases globales. Las clases locales se definen en un programa ABAP. Las clases locales y sus interfaces sólo pueden ser invocadas desde el programa en el que se han definido. Cuando se usa una clase en un programa ABAP el sistema busca primero una clase local con el nombre especificado. Si no encuentra ninguna entonces busca una clase global. A parte de la cuestión de la visibilidad, no hay ninguna diferencia entre usar una clase global o una clase local. Lo que si cambia sensiblemente es la manera en la que una clase local y una clase global son creadas.
Si se define una clase que se va a usar en un único programa, normalmente es suficiente con definir aparentemente los componentes visibles de manera que la clase se ajuste a nuestro programa. Por otro lado, las clases globales deben estar preparadas para ser usadas en cualquier parte. Esto quiere decir que se tienen que aplicar ciertas restricciones cuando se define la interface de una clase global, ya que la clase debe estar preparada para garantizar que cualquier programa que use un objeto de esa clase reconozca el tipo de datos de cada parámetro de la interface.
Veremos por un lado como crear clases e interfaces locales en un programa ABAP, para después ver como utilizar el generador de clases para crear clases e interfaces globales.

DEFINICIÓN DE CLASES LOCALES

Las clases locales son el conjunto de sentencias que están entre las sentencias CLASS… y ENDCLASS. Una definición completa de una clase constará de una parte declarativa en la que se definen los componentes, y si es necesario una parte de implementación en la que se implementan estos componentes.
La parte declarativa de una clase está comprendida entre las sentencias:

```
CLASS <class> DEFINITION.
…
ENDCLASS.
```

La parte declarativa contiene la declaración de todos los componentes de la clase (atributos, métodos y eventos). Cuando se definen clases locales, la parte declarativa pertenece a los datos globales del programa, por tanto se habrá de situar al principio del programa.
Si se declaran métodos en la parte declarativa de una clase, se deberá escribir también su parte de implementación. Ésta es la que va incluída entre las siguientes sentencias:

```
CLASS <class> IMPLEMENTATION.
…
ENDCLASS.
```

La parte de implementación contiene la implementación de todos los métodos de la clase. Esta parte actúa como un bloque, esto quiere decir que cualquier sección de código que no forme parte del bloque no será accesible.

ESTRUCTURA DE UNA CLASE

La estructura de una clase se define principalmente basándose en:
- Una clase contiene componentes.
- Cada componente se asigna a una sección de visibilidad (público, protegido o privado).
- Las clases implementan métodos.

Componentes de las clases

Los componentes de una clase confeccionan sus contenidos. Todos los componentes son declarados en la parte declarativa de la clase. Los componentes definen los atributos de los objetos en una clase. Cuando se define una clase, cada componente es asignado a una de las tres distintas secciones de visibilidad que definen la interface externa de la clase. Todos los componentes de una clase son visibles dentro de la

clase. Además todos comparten el mismo espacio por lo que sus nombres deben ser únicos dentro de la clase.
Hay dos tipos de componentes en una clase, aquellos que existen separadamente para cada objeto de una clase, y aquellos que existen sólo una vez para la clase entera, independientemente del número de instancias. Estos componentes son conocidos como dependientes de instancia o independientes de instancia (o estáticos) respectivamente.
En ABAP Obejcts, las clases pueden definir los siguientes componentes (debido a que todos los componentes que se pueden declarar en las clases también pueden ser declarados en las interfaces, las siguientes descripciones se aplican de la misma manera a las interfaces):

Atributos

Los atributos son los campos de datos internos de una clase y pueden tener cualquier tipo de datos ABAP. El estado de un objeto viene determinado por el contenido de sus atributos. Un tipo de atributos son las variables referenciadas. Estas variables permiten crear y acceder a los objetos, de manera que si se definen en una clase permiten acceder a otros objetos desde dentro de la clase.

- Atributos dependientes de instancia. – El contenido de estos atributos es específico de cada objeto. Se declaran usando la sentencia **DATA**.
- Atributos estáticos – El contenido de los atributos estáticos define el estado de la clase y es válido para todas las instancias la clase. Los atributos estáticos existen sólo una vez para la clase. Se declaran usando la sentencia **CLASS-DATA**. Son accesibles desde todo el entorno de ejecución de la clase. Todos los objetos de una clase pueden acceder a sus atributos estáticos. Si se cambia un atributo estático en un objeto, el cambio es visible en todos los demás objetos de la clase.

Métodos

Los métodos son procedimientos internos de una clase que definen el comportamiento de un objeto. Los métodos pueden acceder a todos los atributos de una clase. Esto les permite cambiar el contenido de los atributos de un objeto. Los métodos poseen también una interface con parámetros que les permite recibir valores cuando son invocados y devolver valores después de la llamada. Los atributos privados de una clase sólo pueden ser cambiados por métodos de la misma clase.
La definición y la interface de un método son similares a las de los módulos de funciones. Un método se define en la parte declarativa de la clase y se implementa en la parte de implementación usando las sentencias:

```
METHOD <meth>.
…
ENDMETHOD.
```

Se pueden declarar tipos de datos locales y objetos en los métodos de la misma manera que en cualquier otro procedimiento ABAP (subrutinas y módulos de funciones). Los métodos se pueden llamar mediante la sentencia **CALL METHOD**.

- Métodos dependientes de instancia – Estos métodos se declaran usando la sentencia **METHODS**. Pueden acceder a todos los atributos de una clase, y pueden desencadenar todos los eventos de una clase.
- Métodos estáticos o independientes de instancia – Estos métodos se declaran usando la sentencia **CLASS-METHODS**. Sólo pueden acceder a los atributos estáticos y desencadenar eventos estáticos.
- Métodos especiales – Además de los métodos normales que se pueden llamar con la sentencia **CALL METHOD**, hay dos métodos especiales llamados **CONSTRUCTOR** y **CLASS_CONSTRUCTOR** que son automáticamente llamados cuando se crea un objeto (CONSTRUCTOR) o cuando se accede por primera vez a los componentes de la clase (CLASS_CONSTRUCTOR).

Eventos

Los objetos o las clases pueden usar eventos para desencadenar un tipo de métodos en otros objetos o clases. Estos métodos se llaman métodos que manejan eventos (event handler methods) En una llamada normal a un método, el método puede ser llamado por cualquier número de usuarios. Cuando un evento es desencadenado, cualquier número de estos métodos puede ser llamado. La unión ente el disparador del evento (trigger) y el manejador del evento (handler) no es establecida de antemano, si no en el entorno de ejecución. En las llamadas normales a métodos, el programa que llama determina los métodos a los que quiere llamar. Estos métodos tienen que existir. El manejador de eventos determina los eventos a los cuales tiene que reaccionar. No tiene porque existir un método manejador de eventos registrado para cada evento. Los eventos de una clase pueden ser desencadenados en los métodos de la misma clase usando la

sentencia **RAISE EVENT**. Un método de la misma o de diferente clase, se declara como método manejador de eventos utilizando la adición **FOR EVENT <evt> OF <class>**.
Los eventos tienen una interface de parámetros similar a la de los métodos, pero sólo tienen parámetros de salida. Los parámetros son pasados por el disparador (sentencia **RAISE EVENT**) al método manejador de eventos el cual los recibe como parámetros de entrada.
El vínculo de unión entre el disparador y el manejador (trigger y handler) es establecido dinámicamente en el programa usando la sentencia **SET HANDLER**. El disparador y el manejador pueden ser objetos o clases, dependiendo de si tenemos eventos dependientes de instancia o eventos estáticos y métodos manejadores de eventos. Cuando un evento es disparado, el correspondiente método manejador de eventos es ejecutado en todas las clases registradas para ese manejador.
Existen dos tipos de eventos:

- Eventos dependientes de instancia – Se declaran con la sentencia **EVENTS**. Sólo pueden ser desencadenados en un método dependiente de instancia.
- Eventos estáticos o independientes de instancia – Se declaran con la sentencia **CLASS-EVENTS**. Todos los métodos (dependientes de instancia y estáticos) pueden desencadenar eventos estáticos. Los eventos estáticos son el único tipo de eventos que puede ser desencadenado por un método estático.

Tipos

Se pueden definir tipos de datos ABAP dentro de una clase con la sentencia **TYPES**. Los tipos de datos no son específicos de cada instancia y existen una sóla vez para todos los objetos de la clase.

Constantes

Las constantes son un tipo especial de atributos estáticos. Su valor se fija cuando son declaradas y no puede ser cambiado. Se declaran usando la sentencia **CONSTANTS**. Las constantes existen sólo una vez para todos los objetos de la clase.

Visibilidad

La parte declarativa de una clase se divide en tres áreas de distinta visibilidad:

```
CLASS <class> DEFINITION.
  PUBLIC SECTION.
  …
  PROTECTED SECTION.
  …
  PRIVATE SECTION.
  …
ENDCLASS.
```

Estas tres áreas definen la visibilidad externa de los componentes de la clase, esto es, la interface entre la clase y el usuario. Cada componente de una clase ha de ser asignado a una de estas tres secciones:

- Public section – Todos los componentes declarados en la sección pública son accesibles para todos los usuarios de la clase y para todos los métodos de la clase y de cualquier clase que herede de ella. Los componentes públicos conforman la interface entre la clase y el usuario.
- Protected section – Todos los componentes declarados en la sección protegida son accesibles para todos los métodos de la clase y de las clases que heredan de ella. Los componentes protegidos conforman la interface entre una clase y todas sus subclases. Debido a que la herencia aún no está activa en el release 4.5B la sección protegida tiene actualmente el mismo efecto que la sección privada.
- Private section – Los componentes declarados en la sección privada son sólo visibles en los métodos de la misma clase. Los componentes privados no forman parte de la interface externa de la clase.

Encapsulación

Las tres áreas de visibilidad son la base de una de las más importantes características de la orientación a objetos, la encapsulación. Cuando se define una clase hay que tener mucho cuidado en el diseño de los componentes públicos, intentando declarar tan pocos como sea posible. Los componentes públicos de las clases globales no pueden ser cambiados una vez que se ha liberado la clase.
Por ejemplo, los atributos públicos son visibles externamente, y forman parte de la interface entre un objeto y sus usuarios. Si se quiere encapsular el estado de un objeto completamente no se tiene que

declarar ningún atributo público. Además de definir la visibilidad de un atributo, se puede proteger también de los cambios usando la adición **READ-ONLY**.

Visión de conjunto

```
CLASS c1 DEFINITION.
  PUBLIC SECTION.
    DATA:     a1 …
    METHODS:  m1 …
    EVENTS:   e1 …
  PROTECTED SECTION.
    DATA:     a2 …
    METHODS:  m2 …
    EVENTS:   e2 …
  PRIVATE SECTION.
    DATA:     a3 …
    METHODS:  m3 …
    EVENTS:   e3 …
ENDCLASS.


CLASS c1 IMPLEMENTATION.
  METHOD m1 … ENDMETHOD.
  METHOD m2 … ENDMETHOD.
  METHOD m3 … ENDMETHOD.
ENDCLASS.
```

Esta sería la estructura de la parte de declaración y la parte de implementación de una clase local c1.

Componentes públicos	Interface entre la clase y sus usuarios
Componentes protegidos	Interface con las subclases de la clase c1
Componentes privados	No visibles externamente, completamente encapsulados en la clase
Implementación de los métodos	Tienen acceso completo a todos los componentes de la clase

EJEMPLO DE UNA CLASE LOCAL

De la misma manera que vimos el ejemplo del contador para el grupo de funciones se puede realizar lo mismo con una clase:

```
CLASS C_CONTADOR DEFINITION.
  PUBLIC SECTION.
    METHODS: FIJAR_CONTADOR IMPORTING VALUE(FIJAR_VALOR) TYPE I,
             INCREMENTAR_CONTADOR,
             OBTENER_CONTADOR EXPORTING VALUE(OBTENER_VALOR) TYPE I.
  PRIVATE SECTION.
    DATA CONT TYPE I.
ENDCLASS.

CLASS C_CONTADOR IMPLEMENTATION.
  METHOD FIJAR_CONTADOR.
    CONT = FIJAR_VALOR.
  ENDMETHOD.
  METHOD INCREMENTAR_CONTADOR.
    ADD 1 TO CONT.
  ENDMETHOD.
```

```
  METHOD OBTENER_CONTADOR.
    OBTENER_VALOR = CONT.
  ENDMETHOD.
ENDCLASS.
```

La clase c_contador contiene tres métodos públicos, fijar_contador, incrementar_contador y obtener_contador. Cada uno de ellos trabaja con el atributo privado cont. Dos de estos métodos tienen parámetros de entrada y de salida. Estos son los que forman la interface de la clase. El atributo cont no es visible directamente.
Más adelante se verá como crear instancias de una clase para poder así utilizar la clase en el programa.

UTILIZACIÓN DE OBJETOS

Objetos

Los objetos son instancias de las clases. Cada objeto tiene una identidad propia y tiene sus propios atributos. Todos los objetos transitorios residen en el contexto de una sesión interna (área de memoria de un programa ABAP). Los objetos permanentes en la base de datos aún no están disponibles (documentación del release 4.6B). Una clase puede tener un número indefinido de objetos (instancias).

Referencias a objeto

Las referencias a objeto se usan para acceder a un objeto desde un programa ABAP. Las referencias a objeto son punteros a los objetos. En ABAP los objetos están siempre contenidos en variables referenciadas.
Las variables referenciadas o bien contienen el valor 'initial' o bien contienen la referencia a un objeto ya existente. La identidad de un objeto depende de su referencia. Una variable referenciada que apunta a un objeto es la que conoce la identidad del objeto. Los usuarios no pueden acceder a la identidad del objeto directamente. Las variables referenciadas en ABAP son tratadas como cualquier otro objeto de datos elemental. Esto quiere decir que una variable referenciada puede contener una tabla interna o una estructura.
ABAP contiene un tipo de datos predefinido para las referencias, comparable a los tipos de datos para las estructuras o para las tablas internas. El tipo de datos completo no está definido hasta la declaración en el programa ABAP. El tipo de datos de la variable referenciada determina como el programa actúa con su valor, o sea, con la referencia al objeto. Hay dos tipos principales de referencias, la referencia a clases y la referencia a interfaces (se verá mas adelante).
Las referencias a clases se definen usando la siguiente adición:
… TYPE REF TO <class>.
Esta adición se usa en las sentencias **TYPES** o **DATA**. Una variable referenciada de este tipo se llama variable referenciada a clase o referencia a clase simplemente.
Una referencia a clase <cref> permite al usuario crear una instancia, o sea un objeto, de la clase y acceder a un componente visible de la siguiente manera:
cref->comp

¿Cómo crear objetos?

Antes de crear un objeto de una clase es necesario declarar una variable referenciada con la referencia a la clase. Una vez que se ha declarado la referencia <obj> a la clase <class>, se puede crear el objeto usando la sentencia **CREATE OBJECT <cref>**. Esta sentencia crea una instancia de la clase <class>, y la variable referenciada <cref> contiene la referencia al objeto.

Acceder a los componentes de un objeto

Los programas sólo pueden acceder a los componentes de las instancias usando las referencias de las variables referenciadas. La sintaxis es la siguiente, siendo ref la variable referenciada:

- Para acceder al atributo attr: **ref->attr.**
- Para llamar al método meth: **CALL METHOD ref->meth.**

Para los componentes estáticos (independientes de instancia, sólo dependientes de clase) se puede usar tanto el nombre de la clase como la variable referenciada. También es posible acceder a los componentes estáticos de una clase antes de que un objeto de la clase haya sido creado. La sintaxis, siendo class la clase es la siguiente:

- Para acceder al atributo estático attr: **`class->attr.`**
- Para llamar al método estático meth: **`CALL METHOD class->meth.`**

Dentro de una clase se puede acceder también a los componentes individuales mediante la referencia a sí mismo ME:

- Para acceder al atributo attr en la propia clase: **`me->attr.`**
- Para llamar al método meth en la propia clase: **`CALL METHOD me->meth.`**

CREAR MAS DE UNA INSTANCIA DE UNA CLASE

En un programa se pueden crear cualquier número de objetos de una misma clase. Estos objetos son completamente independientes unos de otros. Cada uno tiene su propia identidad dentro del programa y sus propios atributos. Cada sentencia **`CREATE OBJECT`** genera un nuevo objeto, cuya identidad está completamente definida por su referencia al objeto.

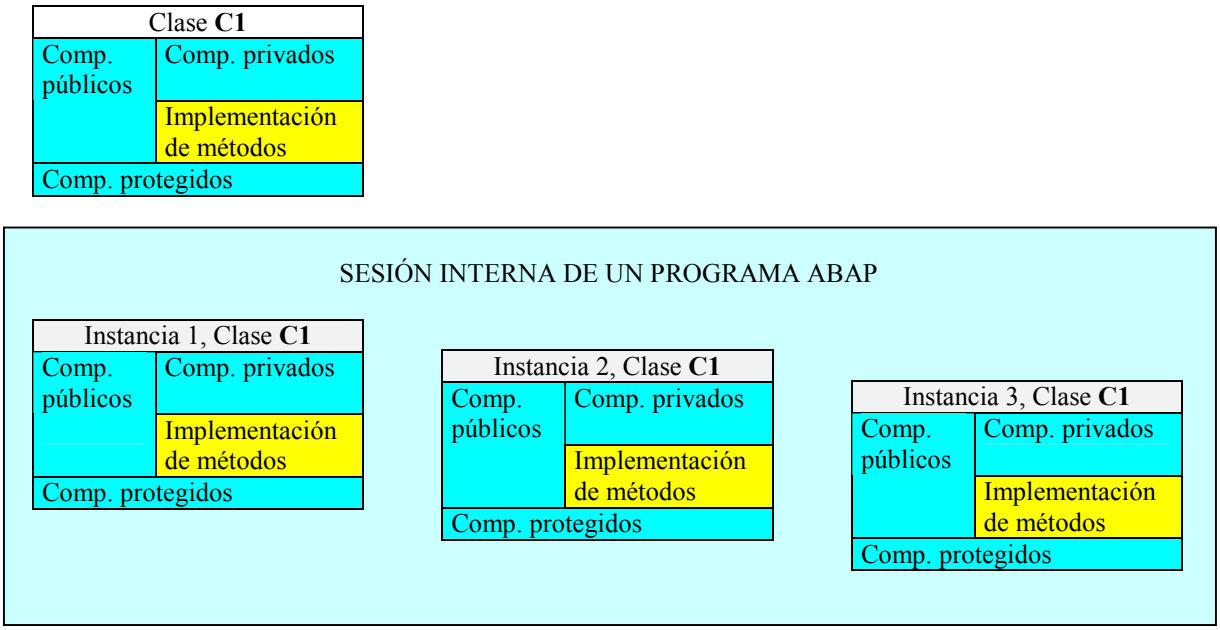
Asignar referencias

Se pueden asignar referencias a distintas variables referenciadas usando la sentencia **`MOVE`**. De esta manera se puede tener las referencias en varias variables referenciadas apuntando al mismo objeto.
Cuando se asigna una referencia a una variable referenciada distinta, sus tipos deben ser compatibles.
Cuando se usa la sentencia **`MOVE`** o el operador de asignación **`=`** para asignar variables referenciadas, el sistema debe ser capaz de reconocer en el chequeo de la sintaxis si la asignación va a ser posible. Esto mismo se aplica cuando se pasan variables referenciadas como parámetros a procedimientos.
Si escribimos la sentencia **`cref1 = cref2`**, las dos referencias tienen que tener el mismo tipo, esto es, tienen que referirse a la misma clase, o bien la clase de **`cref1`** tiene que ser la clase predefinida como vacía, **`OBJECT`**.
La clase **`OBJECT`** no tiene componentes y tiene la misma función para las variables referenciadas que el tipo de datos **`ANY`** para las variables normales.
Las variables referenciadas con el tipo **`OBJECT`** pueden funcionar como contenedoras para pasar referencias. De cualquier manera, nunca pueden ser usadas para acceder a objetos.

Tiempo de vida de un objeto

Un objeto existe mientras se esté usando en el programa, lo que quiere decir que existe siempre que al menos una referencia apunte hacia él, o al menos un método del objeto esté registrado como método manejador de eventos.
Desde el momento en que deja de haber referencias a un objeto y ninguno de sus método es un método manejador de eventos, el objeto es borrado de la memoria automáticamente. El ID del objeto queda libre para ser usado por un nuevo objeto.

OBJETOS COMO INSTANCIAS DE UNA CLASE



Las ilustraciones de arriba muestran una clase y sus instancias dentro de una sesión interna de un programa ABAP.

EJEMPLO: CÓMO CREAR Y USAR UNA CLASE.

En este ejemplo veremos cómo crear y usar una instancia de la clase c_counter que creamos en la sección anterior.

```
DATA cref1 TYPE REF TO c_contador.
```

Creamos una variable **cref1** que es referenciada a la clase **c_contador**. Esta variable puede contener referencias a todas las instancias de la clase **c_contador**. La clase **c_contador** debe ser conocida para el programa en el momento en que la sentencia data tiene lugar. Por tanto la clase **c_contador** debe estar o bien declarada localmente antes de la sentencia data o bien globalmente con el constructor de clases.
Después de esta sentencia el contenido de **cref1** es **initial**, o sea la referencia no apunta a ninguna instancia.

```
DATA cref1 TYPE REF TO c_contador.
CREATE OBJECT cref1.
```

La sentencia **CREATE OBJECT** crea un objeto (instancia) de la clase **c_contador**. La referencia en la variable referenciada **cref1** apunta a este objeto.
La instancia de la clase **c_contador** se llama **c_contador<1>** debido a que así es como se visualizan los contenidos de la variable de objeto en el debugger después de que la sentencia **CREATE OBJECT** haya sido ejecutada. Este nombre es sólo usado internamente por el programa y no aparece nunca en el propio programa ABAP.

```
DATA cref1 TYPE REF TO c_contador.
DATA numero TYPE i VALUE 5.
CREATE OBJECT cref1.
CALL METHOD cref1->fijar_contador
     EXPORTING fijar_valor = numero.
DO 3 TIMES.
 CALL METHOD cref1->incrementar_contador.
ENDDO.
CALL METHOD cref1->obtener_contador
     IMPORTING obtener_valor = numero.
```

EL programa ABAP puede acceder a los componentes públicos de los objetos usando la variable referenciada **cref1**, lo cual en este caso se corresponde a llamar a los métodos públicos de la clase **c_contador**. Después que el programa haya sido ejecutado la variable **numero** y el atributo privado del objeto **cont** tienen ambos el valor 8.

Podemos también manejar varias instancias de la misma clase.

```
DATA cref1 TYPE REF TO c_contador.
DATA cref2 TYPE REF TO c_contador.
DATA cref3 LIKE cref1.
```

Así creamos tres variables referenciadas a la clase **c_contador**. Todas ellas contienen el valor **initial**.

```
DATA cref1 TYPE REF TO c_contador.
DATA cref2 TYPE REF TO c_contador.
DATA cref3 LIKE cref1.
CREATE OBJECT cref1, cref2, cref3.
```

El sistema crea tres objetos de la clase a partir de las tres variables referenciadas a la clase. Las referencias en las tres variables apuntan a cada uno de los objetos. Internamente las instancias se llaman **c_contador <1>**, **c_contador <2>**, y **c_contador <3>**. El número se asigna en el orden en que son creadas.

```
DATA cref1 TYPE REF TO c_contador.
DATA cref2 TYPE REF TO c_contador.
```

```
DATA cref3 LIKE cref1.
DATA numero1 TYPE i VALUE 5.
DATA numero2 TYPE i VALUE 0.
DATA numero3 TYPE i VALUE 2.
CREATE OBJECT cref1, cref2, cref3.
CALL METHOD: cref1->fijar_contador
     EXPORTING fijar_valor = numero1,
             cref2->fijar_contador
     EXPORTING fijar_valor = numero2,
…
CALL METHOD cref2->incrementar_contador.
…
CALL METHOD cref1->incrementar_contador.
…
CALL METHOD: cref1->obtener_contador
     IMPORTING obtener_valor = numero1,
             cref2->obtener_contador
     IMPORTING obtener_valor = numero2,
…
```

El programa ABAP usa las variables referenciadas para acceder a los objetos, en este caso a los métodos publicos de la clase **c_contador**.
Cada objeto tiene su propia identidad y su propio estado, ya que el atributo privado dependiende de instancia **cont** tiene distintos valores en cada objeto. El programa administra varios contadores.

```
DATA cref1 TYPE REF TO c_contador.
DATA cref2 TYPE REF TO c_contador.
DATA cref3 LIKE cref1.
CREATE OBJECT cref1, cref2.
```

Ahora declaramos tres variables referenciadas para la clase **c_contador** y se crean dos objetos para la clase. Las referencias en las variables referenciadas **cref1** y **cref2** apuntan a cada uno de los objetos. La referencia **cref3** se mantiene **initial**.

```
DATA cref1 TYPE REF TO c_contador.
DATA cref2 TYPE REF TO c_contador.
DATA cref3 LIKE cref1.
CREATE OBJECT cref1, cref2.
MOVE cref2 TO cref3.
```

Después de la sentencia **MOVE**, **cref3** contiene la misma referencia que **cref2** y ambas referencias apuntan al objeto **c_contador<2>**. Un usuario puede usar cualquiera de ellas para acceder al objeto.

```
DATA cref1 TYPE REF TO c_contador.
DATA cref2 TYPE REF TO c_contador.
DATA cref3 LIKE cref1.
CREATE OBJECT cref1, cref2.
MOVE cref2 TO cref3.
CLEAR cref2.
```

La sentencia **CLEAR** reinicializa la referencia de **cref2** al valor **initial**. En este momento la variable referenciada **cref2** contiene el mismo valor que inmediatamente después de su declaración y ya no apunta a ningún objeto.

```
DATA cref1 TYPE REF TO c_contador.
DATA cref2 TYPE REF TO c_contador.
DATA cref3 LIKE cref1.
CREATE OBJECT cref1, cref2.
MOVE cref2 TO cref3.
CLEAR cref2.
cref3 = cref1.
```

La referencia en **cref3** ahora apunta al objeto **c_contador<1>**. Ya no hay referencias apuntando al objeto **c_contador<2>** el cual es automáticamente borrado, con lo cual el nombre interno **c_contador<2>** está libre de nuevo.

DECLARACIÓN Y LLAMADA DE MÉTODOS

En esta sección se enseñará como declarar, implementar y llamar métodos en el lenguaje ABAP.

Declaración de métodos

Los métodos se pueden declarar bien en la parte declarativa de una clase o bien en una interface. Para declarar métodos dependientes de instancia se usa la siguiente sentencia:

```
METHODS <meth>
    IMPORTING.. [VALUE()<i_i>[])] TYPE type [OPTIONAL]..
    EXPORTING.. [VALUE()<e_i>[])] TYPE type [OPTIONAL]..
    CHANGING.. [VALUE()<c_i>[])] TYPE type [OPTIONAL]..
    RETURNING VALUE(<r>)
    EXCEPTIONS.. <e_i>..
```

Para declarar métodos estáticos se usa se usa la siguiente sentencia:

```
CLASS-METHODS <meth>..
```

Ambas sentencias tienen la misma sintaxis.

Cuando se declara un método se puede definir su interface de parámetros usando las adiciones **IMPORTING**, **EXPORTING**, **CHANGING**, y **RETURNING**. Estas adiciones definen los parámetros de entrada, de salida, de entrada/salida y el código que devuelve el método. También definen si los parámetros se pasan por referencia o por valor **(VALUE)**, su tipo **(TYPE)**, o si es opcional o por defecto **(OPTIONAL, DEFAULT)**. Al contrario que en los módulos de funciones, el modo por defecto de pasar parámetros a un método es por referencia. Para pasar un parámetro por valor es necesario especificar explícitamente la adición **VALUE**. El valor de retorno **(RETURNING)** debe pasarse siempre explícitamente como por valor **(RETURNING VALUE)**. Es apropiado para métodos que devuelven un solo valor. Si se usa esta adición no se pueden usar los parámetros **EXPORTING** o **CHANGING**.

Al igual que en los módulos de funciones, se pueden usar las excepciones **(EXCEPTIONS)** para permitir al usuario reaccionar ante las situaciones de error cuando el método se está ejecutando.

Implementación de métodos

Los métodos se implementan en la parte de implementación de la clase con las sentencias:

```
METHOD <meth>.
...
ENDMETHOD.
```

Al implementar un método no hay que especificar los parámetros de la interface del método, ya que están definidos en la declaración del método. La interface de parámetros de un método se comporta dentro de la implementación del método como variables locales. Se pueden definir variables locales adicionales dentro del método usando la sentencia **DATA**.

Al igual que en los módulos de funciones, se puede usar las sentencias **RAISE <exception>** y **MESSAGE RAISING** para controlar los errores.

Cuando se implementa un método estático hay que tener en cuenta que sólo puede trabajar con los atributos estáticos de la clase. Los métodos dependientes de instancia pueden trabajar tanto con atributos estáticos como con atributos dependientes de instancia.

Llamada a métodos

Para llamar a un método se usa la siguiente sentencia:

```
CALL METHOD <meth>
    EXPORTING... <i_i> =.<f _i>...
    IMPORTING... <e_i> =.<g _i>...
    CHANGING ... <c_i> =.<f _i>...
    RECEIVING r = h
    EXCEPTIONS... <e_i> = rc _i...
```

La manera en que se llama al método depende del método del que se trate y de desde donde se llame. En la parte de implementación de una clase se pueden llamar directamente métodos de la misma clase simplemente usando su nombe.

```
CALL METHOD <meth>...
```

Desde fuera de la clase, podremos llamar sólo a los métodos cuya visibilidad nos permite hacerlo. Los métodos visibles dependientes de instancia pueden ser llamados desde fuera de la clase usando la sentencia:

```
CALL METHOD <ref>-><meth>...
```

donde **<ref>** es una variable referenciada cuyo valor apunta a una instancia de la clase.
Los métodos visibles estáticos pueden ser llamados desde fuera de la clase con la sentencia:
CALL METHOD <class>=><meth>...
donde class es el nombre de la clase.
Cuando se llama a un método se deben pasar todos los parámetros de entrada no opcionales en las adiciones **EXPORTING** o **CHANGING** en la sentencia **CALL METHOD**. En cambio con los parámetros de salida es optativo pasarselos al método usando las adiciones **IMPORTING** o **RECEIVING**. De la misma manera no es obligatorio controlar las excepciones desencadenadas con la adición **EXCEPTIONS**, aún así se recomienda controlar los errores.
La manera de pasar y recibir valores con los métodos es la misma que con las funciones.
..<parámetro formal> = <parámetro real> después de la correspondiente adición.
Los parámetros de la interface (formales) van siempre a la izquierda del signo igual. El signo igual sirve para asignar las variables del programa a los parámetros de la interface del método.
Si la interface de un método tiene sólo un parámetro **IMPORTING**, se puede usar la siguiente llamada abreviada:
CALL METHOD <method>(f).
El parámetro real <f> es pasado al parámetro de entrada del método.
Si la interface de un método tiene sólo parámetros **IMPORTING**, se puede usar la siguiente llamada abreviada:
CALL METHOD <method>(....<i_i> =.<f $_i$>...).
Cada parámetro real <f $_i$ > es pasado al correspondiente parámetro de entrada del método.

Métodos manejadores de eventos

Los métodos manejadores de eventos son una clase especial de métodos que no pueden ser llamados usando la sentencia **CALL METHOD**. En su lugar estos métodos son desencadenados mediante eventos.
Un método se define como manejador de eventos mediante la adición:
... FOR EVENT <evt> OF <cif>...
en la sentencia **METHODS** o **CLASS-METHODS**.
Las interfaces de los métodos manejadores de eventos tienen que cumplir una serie de reglas:

- La interface sólo puede tener parametros de entrada **(IMPORTING)**.
- Cada parámetro de entrada al método **(IMPORTING)** debe tener su correspondiente parámetro de salida **(EXPORTING)** en el evento.
- Los atributos de los parámetros están definidos en la declaración del evento (sentencia **EVENTS**) y son tomados por el método manejador de eventos.

Constructores

Los constructores son un tipo especial de métodos que no pueden ser llamados con la sentencia **CALL METHOD**. Estos métodos son llamados automáticamente por el sistema para fijar el estado inicial de un nuevo objeto o clase. Hay dos tipos de constructores, los dependientes de instancia y los estáticos o independientes de instancia. Los constructores son métodos con un nombre predifinido. Para usarlos deben ser declarados explícitamente en la clase.
El constructor dependiente de instancia de una clase es un método que se llama **CONSTRUCTOR**. Se declara en la sección pública de la siguiente manera:
METHODS CONSTRUCTOR
IMPORTING.. [VALUE(]<i_i>[)] TYPE type [OPTIONAL]..
EXCEPTIONS.. <e_i>.
Se implementa en la parte de implementación de la misma manera que cualquier otro método. El sistema llama al constructor dependiente de instancia una vez para cada instancia de la clase, justo después de que el objeto haya sido creado mediante la sentencia **CREATE OBJECT**.
Se le pueden pasar parámetros de entrada y controlar sus errores usando las adiciones **EXPORTING** y **EXCEPTIONS** en la sentencia **CREATE OBJECT**.
El constructor estático de una clase es el método estático predefinido **CLASS_CONSTRUCTOR**. Se declara en la sección pública de la siguiente manera:
CLASS-METHODS CLASS_CONSTRUCTOR.
Se implementa como cualquier otro método. El constructor estático no tiene parámetros. El sistema llama al constructor estático una vez para cada clase, justo antes de la clase se utiliza por primera vez. Debido a esto el constructor estático no puede acceder a los componentes de la propia clase.

EJEMPLO DEL USO DE MÉTODOS

El siguiente ejemplo muestra como declarar, implementar y usar métodos en ABAP Objects.

Introducción

Este ejemplo usa tres clases llamadas **c_team**, **c_biker** y **c_bicycle**. Un usario (un programa) puede crear objetos de la clase **c_team**. En la pantalla de selección la clase **c_team** pide el número de miembros de cada equipo.
Cada objeto en la clase **c_team** puede crear tantas instancias de la clase **c_biker** como miembros haya en el equipo. Cada instancia de la clase **c_biker** crea ua instancia de la clase **c_bicycle**.
Cada instancia de la clase **c_team** puede comunicarse con el programa a través de una lista interactiva. El programa puede elegir miembros de los equipos para trabajar con ellos. Las instancias de la clase **c_biker** permiten al programa elegir la acción a realizar en una pantalla posterior.

Restricciones

Hay sentencias ABAP usadas en el proceso de listas que aún no están completamente disponibles en ABAP Objects. Para producir un output de test se pueden usar las siguientes sentencias:

- **WRITE [AT] /<offset>(<length>) <f>**
- **ULINE**
- **SKIP**
- **NEW-LINE**

Declaración

Este ejemplo usa clases locales debido a que las pantallas de selección pertenecen a un programa ABAP y no pueden ser definidas o llamadas en clases globales. Se definen dos pantallas de selección y tres clases.

```
*********************************************************************
* Global Selection Screens
*********************************************************************

SELECTION-SCREEN BEGIN OF: SCREEN 100 TITLE TIT1, LINE.
  PARAMETERS MEMBERS TYPE I DEFAULT 10.
SELECTION-SCREEN END OF: LINE, SCREEN 100.
*---------------------------------------------------------------------
SELECTION-SCREEN BEGIN OF SCREEN 200 TITLE TIT2.
  PARAMETERS: DRIVE      RADIOBUTTON GROUP ACTN,
              STOP       RADIOBUTTON GROUP ACTN,
              GEARUP     RADIOBUTTON GROUP ACTN,
              GEARDOWN   RADIOBUTTON GROUP ACTN.
SELECTION-SCREEN END OF SCREEN 200.

*********************************************************************
* Class Definitions
*********************************************************************

CLASS: C_BIKER   DEFINITION DEFERRED,
       C_BICYCLE DEFINITION DEFERRED.
*---------------------------------------------------------------------
CLASS C_TEAM DEFINITION.
  PUBLIC SECTION.
    TYPES: BIKER_REF     TYPE REF TO C_BIKER,
           BIKER_REF_TAB TYPE STANDARD TABLE OF BIKER_REF
                         WITH DEFAULT KEY,
           BEGIN OF STATUS_LINE_TYPE,
             FLAG(1) TYPE C,
             TEXT1(5) TYPE C,
             ID TYPE I,
             TEXT2(7) TYPE C,
             TEXT3(6) TYPE C,
```

```
            GEAR TYPE I,
            TEXT4(7) TYPE C,
            SPEED TYPE I,
          END OF STATUS_LINE_TYPE.
    CLASS-METHODS: CLASS_CONSTRUCTOR.
    METHODS: CONSTRUCTOR,
             CREATE_TEAM,
             SELECTION,
             EXECUTION.
PRIVATE SECTION.
  CLASS-DATA: TEAM_MEMBERS TYPE I,
              COUNTER TYPE I.
  DATA: ID TYPE I,
        STATUS_LINE TYPE STATUS_LINE_TYPE,
        STATUS_LIST TYPE SORTED TABLE OF STATUS_LINE_TYPE
                    WITH UNIQUE KEY ID,
        BIKER_TAB TYPE BIKER_REF_TAB,
        BIKER_SELECTION LIKE BIKER_TAB,
        BIKER LIKE LINE OF BIKER_TAB.
  METHODS: WRITE_LIST.
ENDCLASS.
*---------------------------------------------------------------------
CLASS C_BIKER DEFINITION.
  PUBLIC SECTION.
    METHODS: CONSTRUCTOR IMPORTING TEAM_ID TYPE I MEMBERS TYPE I,
             SELECT_ACTION,
             STATUS_LINE EXPORTING LINE
                         TYPE C_TEAM=>STATUS_LINE_TYPE.
  PRIVATE SECTION.
    CLASS-DATA COUNTER TYPE I.
    DATA: ID TYPE I,
          BIKE TYPE REF TO C_BICYCLE,
          GEAR_STATUS TYPE I VALUE 1,
          SPEED_STATUS TYPE I VALUE 0.
    METHODS BIKER_ACTION IMPORTING ACTION TYPE I.
ENDCLASS.
*---------------------------------------------------------------------
CLASS C_BICYCLE DEFINITION.
  PUBLIC SECTION.
    METHODS: DRIVE EXPORTING VELOCITY TYPE I,
             STOP EXPORTING VELOCITY TYPE I,
             CHANGE_GEAR IMPORTING CHANGE TYPE I
                         RETURNING VALUE(GEAR) TYPE I
                         EXCEPTIONS GEAR_MIN GEAR_MAX.
  PRIVATE SECTION.
    DATA: SPEED TYPE I,
          GEAR TYPE I VALUE 1.
    CONSTANTS: MAX_GEAR TYPE I VALUE 18,
               MIN_GEAR TYPE I VALUE 1.
ENDCLASS.
*********************************************************************
```

Ninguna de las clases tiene atributos públicos. El estado de las clases sólo puede ser cambiado por sus métodos. La clase **c_team** tiene un constructor estático. Las clases **c_team** y **c_biker** tienen constructores dependientes de instancia.

Implementación

La parte de implementación de las clases contiene la implementación de los métodos declarados en la parte de declaración. Las interfaces de los métodos ya han sido definidas en la declaración. En la implementación los parámetros de las interfaces se comportan como variables locales.
Los siguientes métodos están implementados en el bloque:

```
CLASS C_TEAM IMPLEMENTATION.
*********************************************************************
METHOD CLASS_CONSTRUCTOR.
    TIT1 = 'Team members ?'.
    CALL SELECTION-SCREEN 100 STARTING AT 5 3.
    IF SY-SUBRC NE 0.
      LEAVE PROGRAM.
    ELSE.
      TEAM_MEMBERS = MEMBERS.
    ENDIF.
ENDMETHOD.
*********************************************************************
METHOD CONSTRUCTOR.
  COUNTER = COUNTER + 1.
  ID = COUNTER.
ENDMETHOD.
*********************************************************************
METHOD CREATE_TEAM.
  DO TEAM_MEMBERS TIMES.
    CREATE OBJECT BIKER EXPORTING TEAM_ID = ID
                                  MEMBERS = TEAM_MEMBERS.
    APPEND BIKER TO BIKER_TAB.
    CALL METHOD BIKER->STATUS_LINE IMPORTING LINE = STATUS_LINE.
    APPEND STATUS_LINE TO STATUS_LIST.
  ENDDO.
ENDMETHOD.
*********************************************************************
METHOD SELECTION.
  CLEAR BIKER_SELECTION.
  DO.
    READ LINE SY-INDEX.
    IF SY-SUBRC <> 0. EXIT. ENDIF.
    IF SY-LISEL+0(1) = 'X'.
      READ TABLE BIKER_TAB INTO BIKER INDEX SY-INDEX.
      APPEND BIKER TO BIKER_SELECTION.
    ENDIF.
  ENDDO.
  CALL METHOD WRITE_LIST.
ENDMETHOD.
*********************************************************************
METHOD EXECUTION.
  CHECK NOT BIKER_SELECTION IS INITIAL.
  LOOP AT BIKER_SELECTION INTO BIKER.
    CALL METHOD BIKER->SELECT_ACTION.
    CALL METHOD BIKER->STATUS_LINE IMPORTING LINE = STATUS_LINE.
    MODIFY TABLE STATUS_LIST FROM STATUS_LINE.
  ENDLOOP.
  CALL METHOD WRITE_LIST.
ENDMETHOD.
*********************************************************************
METHOD WRITE_LIST.
  SET TITLEBAR 'TIT'.
  SY-LSIND = 0.
  SKIP TO LINE 1.
  POSITION 1.
  LOOP AT STATUS_LIST INTO STATUS_LINE.
    WRITE: / STATUS_LINE-FLAG AS CHECKBOX,
             STATUS_LINE-TEXT1,
             STATUS_LINE-ID,
             STATUS_LINE-TEXT2,
```

```
              STATUS_LINE-TEXT3,
              STATUS_LINE-GEAR,
              STATUS_LINE-TEXT4,
              STATUS_LINE-SPEED.
  ENDLOOP.
ENDMETHOD.
*****************************************************************
ENDCLASS.
```

El constructor estático es ejecutado antes de que la clase **c_team** se use por primera vez en el programa. El constructor estático llama a la dynpro 100 y fija el atributo estático **team_members** al valor metido por el usuario en el programa. Este atributo tiene el mismo valor para todas las instancias de la clase **c_team**.

El constructor dependiente de instancia es ejecutado justo después de que cada instancia de la clase **c_team** es creada. Para contar el número de instancias creadas de c_team se usa el atributo estático **counter**. El atributo dependiente de instancia **ID** guarda el número de la instancia de la clase.

El método público dependiente de instancia **create_team** puede ser llamado por cualquier usuario de la clase con una variable referenciada a una instancia de la clase. Este método se usa para crear instancias de la clase **c_biker** usando la variable referenciada privada **biker** en la clase **c_team**. Se tienen que pasar los dos parámetros de entrada **team_id** y **members** al constructor dependiente de instancia de la clase **c_biker** en la sentencia **CREATE OBJECT**. Las referencias a las instancias recién creadas son insertadas en la tabla interna privada **biker_tab**. Después de que el método haya sido ejecutado cada línea de la tabla interna contiene una referencia a una instancia de la clase **c_biker**. Estas referencias son sólo visibles dentro de la clase **c_team**. Los usuarios externos no pueden acceder a los objetos de la clase **c_biker**. El método **create_team** también llama al método **status_line** de cada nuevo objeto creado y usa el área de trabajo **status_line** para rellenar la tabla interna privada **status_list** con el parámetro de salida **line**.

El método público dependiente de instancia **selection** puede ser llamado por cualquier usuario de la clase que tenga una variable referenciada con una referencia a ua instancia de la clase. El método selecciona todas las lineas en la lista actual en las cuales el checkbox en la primera columna está seleccionado. Para esas lineas se copian las correspondientes variables referenciadas desde la tabla **biker_tab** en una tabla interna privada adicional llamada **biker_selection**. **Selection** llama entonces al método privado **write_list** el cual despliega la lista.

El método público dependiente de instancia **execution** puede ser llamado por cualquiera de los usuarios de la clase que tenga una variable referenciada con una referencia a una instancia de la clase. El método llama a los dos métodos **select_action** y **status_line** para cada instancia de la clase **c_biker** para la cual hay una referencia en la tabla **biker_selection**. La linea de la tabla **status_list** con la misma clave que el componente **id** en el área de trabajo **status_line** es sobreescrita y desplegada por el método privado **write_list**.

El método privado dependiente de instancia **write_list** sólo puede ser llamado desde los métodos de la clase **c_team**. Este método se usa para desplegar la tabla interna privada **status_list** en la lista básica (sy-lsind = 0, tratamiento listas, indice lista bifurcación) del programa.

Los métodos de la clase **c_biker** van en otro bloque:

```
CLASS C_BIKER IMPLEMENTATION.
*****************************************************************
METHOD CONSTRUCTOR.
  COUNTER = COUNTER + 1.
  ID = COUNTER - MEMBERS * ( TEAM_ID - 1).
  CREATE OBJECT BIKE.
ENDMETHOD.
*****************************************************************
```

```
METHOD SELECT_ACTION.
  DATA ACTIVITY TYPE I.
  TIT2 = 'Select action for BIKE'.
  TIT2+24(3) = ID.
  CALL SELECTION-SCREEN 200 STARTING AT 5 15.
  CHECK NOT SY-SUBRC GT 0.
    IF GEARUP = 'X' OR GEARDOWN = 'X'.
      IF GEARUP = 'X'.
        ACTIVITY = 1.
      ELSEIF GEARDOWN = 'X'.
        ACTIVITY = -1.
      ENDIF.
    ELSEIF DRIVE = 'X'.
      ACTIVITY = 2.
    ELSEIF STOP = 'X'.
      ACTIVITY = 3.
    ENDIF.
  CALL METHOD BIKER_ACTION( ACTIVITY).
ENDMETHOD.
*********************************************************************
METHOD BIKER_ACTION.
  CASE ACTION.
    WHEN -1 OR 1.
      CALL METHOD BIKE->CHANGE_GEAR
        EXPORTING CHANGE = ACTION
          RECEIVING GEAR = GEAR_STATUS
        EXCEPTIONS GEAR_MAX = 1
                   GEAR_MIN = 2.
      CASE SY-SUBRC.
        WHEN 1.
          MESSAGE I315(AT) WITH 'BIKE' ID
          ' is already at maximal gear!'.
        WHEN 2.
          MESSAGE I315(AT) WITH 'BIKE' ID
          ' is already at minimal gear!'.
      ENDCASE.
    WHEN 2.
      CALL METHOD BIKE->DRIVE IMPORTING VELOCITY = SPEED_STATUS.
    WHEN 3.
      CALL METHOD BIKE->STOP IMPORTING VELOCITY = SPEED_STATUS.
  ENDCASE.
ENDMETHOD.
*********************************************************************
METHOD STATUS_LINE.
  LINE-FLAG = SPACE.
  LINE-TEXT1 = 'Biker'.
  LINE-ID = ID.
  LINE-TEXT2 = 'Status:'.
  LINE-TEXT3 = 'Gear = '.
  LINE-GEAR = GEAR_STATUS.
  LINE-TEXT4 = 'Speed = '.
  LINE-SPEED = SPEED_STATUS.
ENDMETHOD.
*********************************************************************
ENDCLASS.
```

El constructor dependiente de instancia es ejecutado justo después de que se cree cada instancia de la clase **c_biker**. Se usa para contar el número de instancias de la clase **c_biker** en el contador **counter** y para asignar el correspondiente número al atributo dependiente de instancia **id** de cada instancia de la clase. El constructor tiene dos parámetros de entrada, **team_id** y **members** los cuales tienen que pasarse en la sentencia **create object** cuando se crea una instancia de la clase **c_biker**.

La referencia en la variable referenciada privada **bike** de cada instancia **c_biker** apunta a la correspondiente instancia de la clase **c_bicycle**.

El método público dependiente de instancia **select_action** puede ser llamado por cualquier usuario de la clase que tenga una variable referenciada con una referencia que apunte a una instancia de la clase. El método llama a la pantalla de selección 200 y analiza la entrada del usuario. Después de esto llama al método privado de la misma clase **biker_action**. La llamada al método usa la forma abreviada para pasar el parámetro real **activity** al parámetro formal **action**.

El método privado dependiente de instancia **biker_action** sólo puede ser llamado desde los métodos de la clase **c_biker**. El método llama a otros métodos en la instancia de la clase **c_bicycle** a la cual la referencia en la variable referenciada **bike** está apuntando, dependiendo del valor en el parámetro de entrada **action**.

El método público dependiente de instancia **status_line** puede ser llamado por cualquier usuario de la clase que tenga una variable referenciada con una referencia a una instancia de la clase. El método rellena la estructura de salida **line** con los actuales valores de los atributos de la correspondiente instancia.

Los métodos de la clase **c_bicycle** van en otro bloque:

```
CLASS C_BICYCLE IMPLEMENTATION.
*********************************************************************
METHOD DRIVE.
  SPEED = SPEED + GEAR * 10.
  VELOCITY = SPEED.
ENDMETHOD.
*********************************************************************
METHOD STOP.
  SPEED = 0.
  VELOCITY = SPEED.
ENDMETHOD.
*********************************************************************
METHOD CHANGE_GEAR.
  GEAR = ME->GEAR.
  GEAR = GEAR + CHANGE.
  IF GEAR GT MAX_GEAR.
    GEAR = MAX_GEAR.
    RAISE GEAR_MAX.
  ELSEIF GEAR LT MIN_GEAR.
    GEAR = MIN_GEAR.
    RAISE GEAR_MIN.
  ENDIF.
  ME->GEAR = GEAR.
ENDMETHOD.
*********************************************************************
ENDCLASS.
```

El método público dependiente de instancia **drive** puede ser llamado por cualquier usuario que tenga una variable referenciada con una referencia a una instancia de la clase. El método cambia el valor del atributo privado **speed** y lo devuelve en el parámetro de salida **velocity**.

El método público dependiente de instancia **stop** puede ser llamado por cualquier usuario que tenga una variable referenciada con una referencia a una instancia de la clase. El método cambia el valor del atributo privado **speed** y lo devuelve en el parámetro de salida **velocity**.

El método público dependiente de instancia **change_gear** puede ser llamado por cualquier usuario que tenga una variable referenciada con una referencia a una instancia de la clase. El método cambia el valor

del atributo privado **gear**. Debido a que el parámetro formal con el mismo nombre 'oculta' el atributo en el método, el atributo tiene que ser direccionado con la referencia a sí mismo **me->gear**.

Utilización en programas

Ahora veremos cómo se pueden usar en un programa las clases que hamos declarado e implementado. Las declaraciones de las pantallas de selección, de las clases locales y la implementación de los métodos deben ser también parte del programa.

```
REPORT OO_METHODS_DEMO NO STANDARD PAGE HEADING.
*---------------------------------------------------------------------*
* Declarations and Implementations
*---------------------------------------------------------------------*
...
*---------------------------------------------------------------------*
* Global Program Data
*---------------------------------------------------------------------*
TYPES TEAM TYPE REF TO C_TEAM.
DATA: TEAM_BLUE TYPE TEAM,
      TEAM_GREEN TYPE TEAM,
      TEAM_RED TYPE TEAM.
DATA COLOR(5).
*---------------------------------------------------------------------*
* Program events
*---------------------------------------------------------------------*
START-OF-SELECTION.
  CREATE OBJECT: TEAM_BLUE,
                 TEAM_GREEN,
                 TEAM_RED.
  CALL METHOD: TEAM_BLUE->CREATE_TEAM,
               TEAM_GREEN->CREATE_TEAM,
               TEAM_RED->CREATE_TEAM.
  SET PF-STATUS 'TEAMLIST'.
  WRITE ' Select a team! ' COLOR = 2.
*---------------------------------------------------------------------
AT USER-COMMAND.
  CASE SY-UCOMM.
    WHEN 'TEAM_BLUE'.
      COLOR = 'BLUE '.
      FORMAT COLOR = 1 INTENSIFIED ON INVERSE ON.
      CALL METHOD TEAM_BLUE->SELECTION.
    WHEN 'TEAM_GREEN'.
      COLOR = 'GREEN'.
      FORMAT COLOR = 5 INTENSIFIED ON INVERSE ON.
      CALL METHOD TEAM_GREEN->SELECTION.
    WHEN 'TEAM_RED'.
      COLOR = 'RED '.
      FORMAT COLOR = 6 INTENSIFIED ON INVERSE ON.
      CALL METHOD TEAM_RED->SELECTION.
    WHEN 'EXECUTION'.
      CASE COLOR.
        WHEN 'BLUE '.
          FORMAT COLOR = 1 INTENSIFIED ON INVERSE ON.
          CALL METHOD TEAM_BLUE->SELECTION.
          CALL METHOD TEAM_BLUE->EXECUTION.
        WHEN 'GREEN'.
          FORMAT COLOR = 5 INTENSIFIED ON INVERSE ON.
          CALL METHOD TEAM_GREEN->SELECTION.
          CALL METHOD TEAM_GREEN->EXECUTION.
        WHEN 'RED '.
          FORMAT COLOR = 6 INTENSIFIED ON INVERSE ON.
          CALL METHOD TEAM_RED->SELECTION.
```

```
          CALL METHOD TEAM_RED->EXECUTION.
      ENDCASE.
  ENDCASE.
*****************************************************************
```

El programa tiene tres variables referenciadas a la clase **c_team**. Se crean tres objetos de la clase a los cuales apuntan las tres variables referenciadas. En cada objeto se llama al método **create_team**. El método **class_constructor** de la clase **c_team** es ejecutado antes de que el primero de los objetos sea creado. El estatus **teamlist** para la lista básica permite al usuario elegir una de las cuatro funciones. Cuando el usuario escoge una de las funciones, el evento **AT USER-COMMAND** es disparado y los métodos públicos son llamados en una de las tres instancias de **c_team**, dependiendo de la elección del usuario. El usuario puede cambiar el estado de un objeto seleccionando la correspondiente linea en la lista.

HERENCIA

La herencia permite crear una nueva clase a partir de una nueva existente heredando la nueva clase sus propiedades. Esto se realiza añadiendo la adición **INHERITING FROM** a la sentencia de definición de la clase:

```
CLASS <subclass> DEFINITION INHERITING FROM <superclass>.
```

La nueva clase **<subclass>** hereda todos los componentes de la clase ya existente **<superclase>**. La nueva clase se conoce como la subclase de la clase de la que procede. La clase original se conoce como la superclase de la nueva clase. Si no se añade ninguna declaración a la subclase, esta contiene los mismos componentes que la superclase. De cualquier manera, sólo los componentes públicos y privados de la superclase son visibles en la subclase. Aunque los componentes privados de la superclase existen en la subclase, no son visibles.

Se pueden declarar componentes privados en una subclase que tengan los mismos nombres que componentes privados de la superclase. Cada clase trabaja con sus propios componentes privados. Los métodos que una subclase hereda de una superclase usan los atributos privados de la superclase y no ningún componente privado de la subclase con el mismo nombre.

Si la superclase no tiene una sección privada, la subclase es una réplica exacta de la superclase. De todos modos podemos añadir nuevos componentes a la subclase. Esto permite convertir a la subclase en una versión especializada de la superclase. Si una subclase es ella misma una superclase de otras clases, se está introduciendo un nuevo nivel de especialización.

Una clase puede tener más de una subclase de las cuales es superclase, pero sólo puede tener una superclase de la cual es subclase. Esto se conoce como herencia simple, en contraposición con la herencia múltiple donde una clase hereda de varias superclases. Cuando una subclase hereda de una superclase que a su vez hereda de otra superclase de la cual es subclase, se forma una estructura de árbol en la cual el grado de especialización aumenta con cada nivel jerárquico que se añada. A la inversa, las clases se hacen más generales hasta que se alcanza el nodo raiz del árbol de herencia. El nodo raiz de todos los árboles de herencia en ABAP Objects es la clase predefinida vacía **OBJECT**. Esta es la más general de todas las clases posibles ya que no contiene ni atributos ni métodos. Cuando se define una nueva clase no se tiene que especificar explícitamente esta clase como superclase, esta relación está definida implícitamente. Dentro de un árbol de herencia, dos nodos adyacentes son la superclase y la subclase directamente uno de otro. Las declaraciones de componentes en una subclase están distribuidas a través de todos los niveles superiores en el árbol de herencia.

Redefinición de métodos

Todas las subclases contienen los componentes de todas las clases existentes entre ellas mismas y el nodo raiz del árbol de herencia. La visibilidad de un componente no puede ser cambiada nunca. En cambio se puede usar la adición **REDEFINITION** en la sentencia **METHODS** para redefinir un método público o protegido dependiente de instancia en una subclase y hacer que realice una función más especializada. Cuando se redefine un método no se puede cambiar su interface, el método mantiene el mismo nombre y la misma interface de parámetros, pero tiene una nueva implementación. La declaración y la implementación de un método en una superclase no se ve afectada cuando se redefine un método en una subclase. La implementación de la redefinición en la subclase 'oculta' la implementación original en la superclase.

Cualquier referencia que apunte a un objeto de la subclase usa el método redefinido, incluso si la referencia fue definida con referencia a la superclase. Esto se aplica particularmente a la referencia a sí mismo **me->**. Si por ejemplo un método **M1** de una superclase contiene una llamada **CALL METHOD**

[ME->]M2 y M2 está redefinido en una subclase, la llamada a **M1** desde una instancia de la superclase hará que el método original **M2** sea llamado, mientras que la llamada a **M1** desde una instancia de la subclase hará que el método redefinido **M2** sea llamado (llaman a 'distintos' métodos auquen tengan el mismo nombre). Dentro de un método redefinido se puede usar la referencia **SUPER->** para acceder al método 'oculto'. Esto permite usar la funcionalidad existente en el método de la superclase sin tener que codificarla de nuevo en la subclase.

Clases y métodos abstractos y finales

Las adiciones **ABSTRACT** y **FINAL** en las sentencias **METHODS** y **CLASS** permiten definir métodos o clases abstractos y finales.
Un método abstracto se define en una clase abstracta y no puede ser implementado en esa clase, tiene que ser implementado en una subclase de la clase. Las clases abstractas no pueden ser instanciadas.
Un método final no puede ser redefinido en una subclase. Las clases finales no pueden tener subclases, son las que finalizan el árbol de herencia.

Referencias a subclases y polimorfismo

Las variables referenciadas con referencia a una superclase o a una interface también pueden contener referencias a algunas de sus subclases. Debido a que las subclases contienen todos los componentes de todas sus superclases y dado que las interfaces de los métodos no pueden ser cambiadas, una variable referenciada definida con referencia a una superclase o a una interface implementada por una superclase, puede contener referencias a instancias de cualquiera de las subclases.
En particular, se pueden definir variables con referencia a la clase genérica **OBJECT** para así poder almacenar cualquier referencia a objetos en ellas.
Cuando se crea un objeto con la sentencia **CREATE OBJECT** y una variable referenciada con referencia a una subclase, se puede usar la adición **TYPE** para crear una instancia de la subclase a la cual apuntará la referencia en la variable referenciada
Un usuario estático puede usar una variable referenciada para acceder a los componentes visibles de la superclase a la cual la variable referenciada apunta.
Si se redefine un método dependiente de instancia en una o más subclases, se puede usar una única variable referenciada para llamar a las diferentes implementaciones de los métodos, dependiendo de la posición en el árbol de herencia del objeto referenciado. Este concepto de que diferentes clases pueden tener la misma interface y por lo tanto se puede acceder a ellas usando variables referenciadas con un único tipo se llama polimorfismo.

Nombres de los componentes

Las subclases contienen todos los componentes de todas sus superclases dentro del árbol de herencia. De esos componentes, sólo los componentes públicos y protegidos son visibles. Todos los componentes públicos y protegidos dentro de un árbol de herencia tienen que tener nombres únicos. Por otro lado, los nombres de los componentes privados deben ser únicos sólo dentro de su clase.
Cuando se redefinen los métodos, la nueva implementación 'oculta' al método de la superclase con el mismo nombre, pero la nueva definición sustituye a la anterior de manera que el nombre del método sigue siendo único. Se puede usar la pseudoreferencia **SUPER->** para acceder a la definición de un método en una superclase que ha sido oscurecida por la redefinición en la subclase.

Herencia y atributos estáticos

Al igual que todos los componentes, los atributos estáticos sólo existen una vez en cada árbol de herencia. Una subclase puede acceder a los atributos estáticos públicos y protegidos de todas sus superclases. A la inversa, una superclase comparte sus atributos públicos y protegidos con todas sus subclases. En términos de herencia, los atributos estáticos no están asignados a una única clase, si no que forman parte de todo el árbol de herencia. Se pueden cambiar desde fuera de la clase usando el componente elegido con algún nombre de alguna de las clases del árbol de herencia, o desde dentro de cualquier clase en la que los atributos sean compartidos, o sea, dentro de cualquier clase del árbol de herencia. Son visibles en todas las clases del árbol de herencia.
Cuando se direcciona un atributo estático que pertenece a una parte del árbol de herencia, siempre se direcciona a la clase en la que el atributo es declarado, independientemente de la clase especificada en la selección de la clase. Esto es importante cuando se llama al constructor estático de clases dentro de la herencia. Los constructores estáticos son ejecutados la primera vez que se accede a una clase. Si se accede

a un atributo estático declarado en una superclase usando el nombre de la subclase, únicamente el constructor estático de la superclase es ejecutado.

Herencia y constructores

Hay una serie de reglas especiales que se aplican a los constructores dentro de la herencia.

Constructores dependientes de instancia

Cada clase tiene un constructor dependiente de instancia llamado **CONSTRUCTOR**. Esta es una excepción a la regla de que los nombres de los componentes dentro de un árbol de herencia deben ser únicos. Los constructores dependientes de instancia de varias clases en un árbol de herencia son completamente independientes uno de otro. No se puede redefinir el constructor dependiente de instancia de una superclase en una subclase, ni llamar específicamente al método con **CALL METHOD CONSTRUCTOR**, de manera que no pueden tener lugar conflictos con los nombres.

El constructor dependiente de instancia de una clase es llamado por el sistema cuando se instancia la clase usando la sentencia **CREATE OBJECT**. Debido a que una subclase contiene todos los atributos visibles de sus superclases, los cuales pueden ser fijados por constructores dependientes de instancia, el constructor dependiente de instancia de una subclase tiene que asegurarse de que todos los constructores dependientes de instancia de todas las superclases también son llamados. Para que esto ocurra, el constructor dependiente de instancia de cada clase tiene que contener la sentencia **CALL METHOD SUPER->CONSTRUCTOR**. La única excepción a esta regla son las subclases directas del nodo raíz **OBJECT**. En las superclases sin un constructor dependiente de instancia definido explícitamente, es llamado el constructor dependiente de instancia implícito.

Cuando se llama a un constructor dependiente de instancia se deben proporcionar valores para todos los parámetros de la interface que no sean opcionales. Hay varias maneras de hacer esto:

- Usando la sentencia **CREATE OBJECT** – Si la clase que se está instanciando tiene un constructor dependiente de instancia con una interface entonces se tienen que pasar los valores usando **EXPORTING**. Si la clase que se está instanciando no tiene explícitamente un constructor dependiente de instancia, se tiene que mirar en el árbol de herencia en el siguiente nivel por encima con un constructor dependiente de instancia explícitamente declarado. Si éste tiene una interface entonces se tienen que pasar los valores con **EXPORTING**. De otra manera no habría que pasarle valores.
- Usando la sentencia **CALL METHOD SUPER->CONSTRUCTOR** – Si la superclase directa tiene un constructor dependiente de instancia con una interface entonces se tienen que pasar los valores usando **EXPORTING**. Si la superclase directa tiene un constructor dependiente de instancia sin interface entonces no se tienen que pasar parámetros. Si la superclase directa no tiene explícitamente un constructor dependiente de instancia se tiene que mirar en el árbol de herencia en el siguiente nivel por encima con un constructor dependiente de instancia explícitamente declarado. Si éste tiene una interface entonces se tienen que pasar los valores con **EXPORTING**. De otra manera no habría que pasarle valores.

En ambos casos se tiene que mirar en el siguiente constructor dependiente de instancia explícitamente disponible y si tiene una interface pasarle los valores. Lo mismo se aplica al control de excepciones para los constructores dependientes de instancia. Cuando se trabaja con herencia se necesita un conocimiento preciso de todo el árbol de herencia. Cuando se instancia una clase en la parte de abajo del árbol de herencia se tienen que pasar los parámetros del constructor de clase más cercano al nodo raiz.

El constructor dependiente de instancia de una subclase está dividido en dos partes por la sentencia **CALL METHOD SUPER->CONSTRUCTOR**. En las sentencias anteriores a la llamada el constructor se comporta como un método estático, esto es, no puede acceder a los atributos dependientes de instancia de su clase. No se puede acceder a los atributos dependientes de instancia hasta después de la llamada. Por esto, las sentencias anteriores a la llamada se usan para determinar los parámetros reales para la interface del constructor dependiente de instancia de la superclase. Sólo se pueden usar atributos estáticos o datos locales para hacer esto.

Cuando se instancia una subclase, los constructores dependientes de instancia son llamados jerárquicamente, el primer nivel anidado en el cual se direccionan atributos dependientes de instancia es el nivel más alto de la superclase. Cuando se llega a los constructores de las clases de los niveles más bajos, se pueden direccionar sucesivamente sus atributos dependientes de instancia.

En un método constructor, los métodos de las subclases de la clase no son visibles. Si un constructor dependiente de instancia llama a un método dependiente de instancia de la misma clase usando la referencia implícita a sí mismo **ME->**, el método es llamado como implementado en la clase del constructor dependiente de instancia y no en ninguna forma redefinida que pueda haber en la subclase que se quiere instanciar. Esto es una excepción a la regla de que cuando se llaman métodos dependientes de

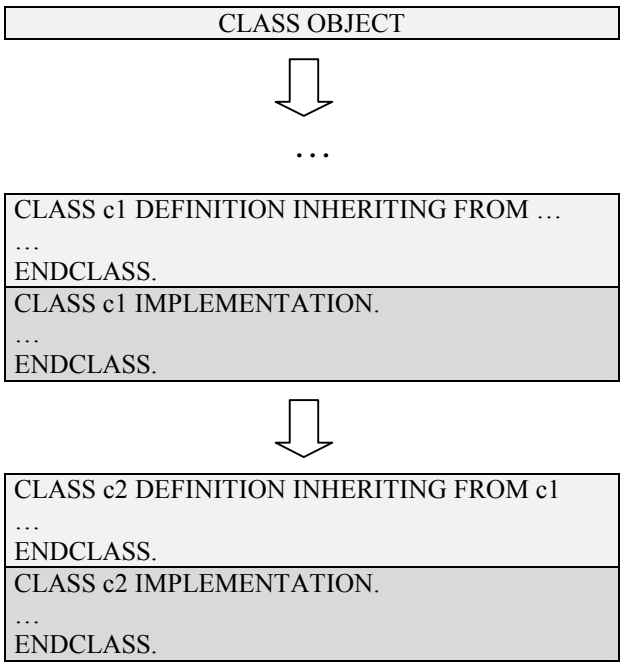
instancia el sistema siempre llama al método implementado en la clase a cuya instancia la referencia está apuntando.

Constructores estáticos

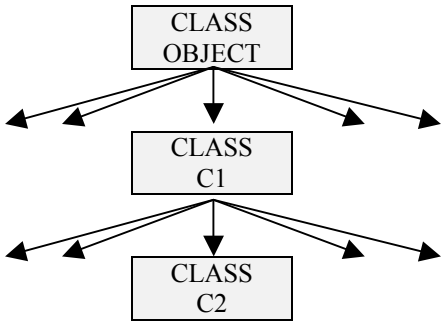
Cada clase tiene un constructor estático llamado **`CLASS_CONSTRUCTOR`**.

Lo referente a la nomenclatura de los componentes en el árbol de herencia se aplica igualmente a los constructores estáticos como a los constructores dependientes de instancia. La primera vez que se accede a una subclase en un programa, su constructor estático es ejecutado. De cualquier manera, antes de que pueda ser ejecutado, el constructor estático de todas sus superclases debe haber sido ejecutado antes. Un constructor estático sólo puede ser llamado una vez por programa. Por lo tanto cuando se accede por primera vez a una subclase el sistema busca la superclase del siguiente nivel por encima cuyo constructor estático aún no haya sido ejecutado. Se ejecuta el constructor estático de esa clase, seguido de todos los de las clases entre esa clase y la clase a la que accedemos.

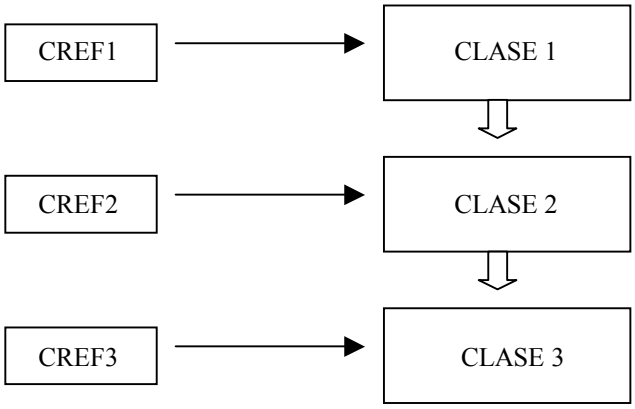
HERENCIA: VISIÓN DE CONJUNTO



La subclase c2 deriva de la superclase c1. En lo más alto del árbol de herencia está la clase **`OBJECT`**.



La herencia simple consiste en que cada clase sólo deriva directamente de una superclase, pero puede tener varias subclases directas. La clase vacía OBJECT es el nodo raíz de cada árbol de herencia en ABAP Objects.



Este gráfico muestra cómo variables referenciadas con referencia a una superclase pueden apuntar a objetos de sus subclases. Tenemos una instancia de la clase 3. Las variables referenciadas a clases cref1, cref2 y cref3 tienen el tipo de las clases 1, 2 y 3 respectivamente. Las tres variables referenciadas pueden apuntar a la clase 3, pero la variable cref1 sólo puede acceder a los componentes públicos de la clase 1, cref2 puede acceder a los componentes públicos de las clases 1 y 2, y cref3 puede acceder a los componentes públicos de todas las clases.
Si se redefine un método de una superclase en una subclase, se pueden usar variables referenciadas definidas con referencia a la superclase para direccionar objetos con diferente implementación de métodos. Cuando se direcciona la superclase, el método tiene su implementación original, pero cuando se direcciona la subclase, el método tiene la nueva implementación. El usar una misma variable referenciada para llamar a métodos con el mismo nombre pero que se comportan distinto se conoce como polimorfismo.

EJEMPLO DE HERENCIA

El siguiente ejemplo muestra el concepto de herencia en ABAP Objects. Una nueva clase **`counter_ten`** hereda de la clase ya existente **`counter`**.

```
REPORT demo_inheritance.
*********************************************************************
  CLASS counter DEFINITION.
    PUBLIC SECTION.
      METHODS: set IMPORTING value(set_value) TYPE i,
               increment,
               get EXPORTING value(get_value) TYPE i.
    PROTECTED SECTION .
      DATA count TYPE i.
  ENDCLASS.
  CLASS counter IMPLEMENTATION.
    METHOD set.
      count = set_value.
    ENDMETHOD.
    METHOD increment.
      ADD 1 TO count.
    ENDMETHOD.
    METHOD get.
      get_value = count.
    ENDMETHOD.
  ENDCLASS.
*********************************************************************
  CLASS counter_ten DEFINITION INHERITING FROM counter.
    PUBLIC SECTION.
      METHODS increment REDEFINITION .
      DATA count_ten.
  ENDCLASS.
```

```
  CLASS counter_ten IMPLEMENTATION.
    METHOD increment.
      DATA modulo TYPE I.
      CALL METHOD super->increment .
      WRITE / count.
      modulo = count mod 10.
      IF modulo = 0.
        count_ten = count_ten + 1.
        write count_ten.
      ENDIF.
    ENDMETHOD.
  ENDCLASS.
*****************************************************************
  DATA: count TYPE REF TO counter,
        number TYPE i VALUE 5.

START-OF-SELECTION.
  CREATE OBJECT count TYPE counter_ten .
  CALL METHOD count->set EXPORTING set_value = number.
  DO 20 TIMES.
    CALL METHOD count->increment.
  ENDDO.
```

La clase **counter_ten** se deriva de la clase **counter**. Se redefine el método **increment**. Para hacer esto se tiene que cambiar la visibilidad del atributo **count** de privado a protegido. El método redefinido llama al método 'oscurecido' de la superclase usando la pseudoreferencia **SUPER->**. El método redefinido es una especialización del método heredado. El ejemplo instancia la subclase. La variable referenciada que apunta a la subclase tiene el tipo de la superclase. Cuando el método **increment** es llamado usando la referencia de la superclase, el sistema ejecuta el método redefinido de la subclase.

INTERFACES

Las clases, sus instancias (los objetos) y el acceso a los objetos usando variables referenciadas son la base de la programación orientada a objetos en ABAP.
Además, hay veces en las que es necesario para clases similares proporcionar funcionalidades similares pero que están codificadas diferentes en cada clase, las cuales dan un punto de contacto común con el usuario. Por ejemplo, podríamos tener dos clases similares, cuenta corriente y cuenta de ahorro, las cuales tienen un método para calcular las comisiones del año. Las interfaces y nombres de los métodos son los mismos pero la implementación es diferente. El usuario de las clases y las instancias tiene que ser capaz de ejecutar el método para todas las cuentas sin preocuparse del tipo de cada cuenta individual.
ABAP Objects hace posible esto mediante el uso de las interfaces. Las interfaces son estructuras independientes que se pueden implementar en una clase para extender el ámbito de esa clase. El ámbito específico de una clase viene definido por sus componentes y sus secciones de visibilidad. Por ejemplo, los componentes públicos de una clase definen su ámbito público, ya que todos sus atributos y los parámetros de los métodos pueden ser utilizados por todos los usuarios. Los componentes protegidos de una clase definen su ámbito en lo que se refiere a sus subclases.
Las interfaces extienden el ámbito de una clase añadiendo sus propios componentes a la sección pública. Esto permite a los usuarios acceder a diferentes clases por medio de un punto de contacto común. Las interfaces junto con la herencia proporcionan uno de los pilares básicos del polimorfismo, ya que permiten que un sólo método con una interface se comporte distinto en diferentes clases.

Definición de interfaces

Al igual que las clases, las interfaces se pueden definir o bien globalmente en el R/3 Repository o localmente en un programa ABAP. La definición de una interface local es el código existente entre las sentencias:

```
INTERFACE <intf>.
…
ENDINTERFACE.
```

La definición contiene la declaración de todos los componentes (atributos, métodos, eventos) de la interface. Se pueden definir los mismos componentes en una interface que en una clase. Los componentes

de las interfaces no tienen que ser asiganados a ninguna sección de visibilidad ya que automáticamente pertenecen a la sección pública de la clase en la que la interface es implementada.
Las interfaces no tienen una parte de implementación ya que sus métodos son implementados en la clase que implementa la interface.

Implementación de interfaces

A diferencia de las clases, las interfaces no tienen instancias, en vez de eso, las interfaces son implementadas por las clases. Para implementar una interface en una clase se usa la sentencia **INTERFACES <intf>.** en la parte declarativa de la clase. Esta sentencia sólo puede aparecer en la sección pública de la clase.
Cuando se implementa una interface en una clase, los componentes de la interface se añaden al resto de componentes de la sección pública. Un componente **<icomp>** de una interface **<intf>** puede ser direccionado como si fuese un miembro de la clase bajo el nombre **<intf~icomp>**. La clase tiene que implementar los métodos de todas las interfaces implementadas en ella. La parte de implementación de la clase debe contener la implementación de cada método de la interface **<imeth>**:

```
METHOD <intf~imeth>.
…
ENDMETHOD.
```

Las interfaces pueden ser implementadas por diferentes clases. Cada una de las clases es ampliada con el mismo conjunto de componentes, aunque los métodos de la interface pueden ser implementados de manera distinta en cada clase.
Las interfaces permiten usar diferentes clases de una manera uniforme aprovechando las referencias a las interfaces (polimorfismo). Por ejemplo, las interfaces implementadas en diferentes clases amplían el ámbito público de cada clase en el mismo conjunto de componentes. Si la clase no tiene ningún componente público específico de ella misma entonces la interface describe completamente el ámbito público de la clase.

Referencias a interfaces

Las variables referenciadas permitan acceder a los objetos. En lugar de crear variables referenciadas con referencia a una clase, se pueden crear con referencia a una interface. Este tipo de variables referenciadas puede contener referencias a los objetos de las clases que implementen esa interface.
Para definir una referencia a una interface se usa la adición **TYPE REF TO <intf>** en las sentencias **DATA o TYPES**. **<intf>** tiene que ser una interface que haya sido declarada en el programa antes de que esta declaración tenga lugar. Una variable referenciada con el tipo referenciado a una interface se llama variable referenciada a una interface, o referencia a interface simplemente.
Una referencia a interface **<iref>** permite al usuario usar **<iref>-><icomp>** para acceder a todos los componentes visibles de la interface **<icomp>** del objeto al cual la referencia está apuntando. Esto permite al usuario acceder a todos los componentes del objeto que fueron añadidos a su definición al implementar la interface.

Direccionar objetos usando referencias a interfaces

Para crear un objeto de la clase **<class>** primero se tiene que haber declarado una variable referenciada **<cref>** con referencia a la clase. Si la clase **<class>** implementa una interface **<intf>**, se puede usar la siguiente asignación entre la variable referenciada a la clase **<cref>** y una referencia a interface **<iref>** para hacer que la referencia a interface **<iref>** apunte al mismo objeto que la referencia a clase en **<cref>**:

```
<iref> = <cref>.
```

Si la interface **<intf>** contiene un atributo dependiente de instancia <attr> y un método dependiente de instancia <meth> se puede direccionar los componentes de la interface como sigue:
Usando la variable referenciada a una clase **<cref>**:

- Para acceder a un atributo **<attr>**: **<cref>-><intf~attr>**
- Para acceder al método **<meth>**: **CALL METHOD <cref>-><intf~meth>**

Usando la variable referenciada a una interface **<iref>**:

- Para acceder a un atributo **<attr>**: **<iref>-><attr>**
- Para acceder al método **<meth>**: **CALL METHOD <iref>-><meth>**

Siempre que los componentes estáticos de las interfaces estén implicados sólo se puede usar el nombre de la interface para acceder a las constantes:
Para acceder a una constante **<const>**: **< intf>=><const>**

Para todos los demás componentes estáticos de una interface, sólo se pueden usar referencias a objetos o la clase que implementa la interface:
Para acceder a un atributo estático <attr>: **`<class>=><intf~attr>`**
Para llamar a un método estático <meth>: **`CALL METHOD <class>=><intf~meth>`**

Asignación usando referencias a interfaces

Al igual que las referencias a clases, se pueden asignar referencias a interfaces a diferentes variables referenciadas. También se pueden realizar las asignaciones entre variables referenciadas a clases y a interfaces. Cuando se usa la sentencia **`MOVE`** o el operador de asignación (=) para asignar variables referenciadas, el sistema debe ser capaz de reconocer en la comprobación de la sintaxis si la asignación es posible.
Supongamos que tenemos una referencia a clase <cref> y tres referencias a interfaces <iref>, <iref1> y <iref2>. Las siguientes asignaciones con referencias a interfaces pueden ser comprobadas estáticamente:

- <iref1> = <iref2>. Ambas referencias a interfaces deben referirse a la misma interface o la interface de <iref1> tiene que tener a la interface <cref2> como componente.
- <iref> = <cref>. La clase de la referencia a clase <cref> tiene que implementar la interface de la referencia a interface <iref>
- <cref> = <iref>. La clase de <cref> tiene que ser la clase predefinida vacía **`OBJECT`**.

En todos lo demás casos se tiene que trabajar con la sentencia **`MOVE … ?TO`** o con el operador de asignación **`?=`** en lugar de los usados antes. Cuando se usan estos dos operadores no se realiza el chequeo estático. En su lugar el sistema comprueba en tiempo de ejecución si la referencia al objeto apunta a un objeto al cual puede apuntar. Si la asignación es posible el sistema la realiza y si no lo es genera el error en tiempo de ejecución **`MOVE_CAST_ERROR`** .
Siempre que se asigne una referencia a una interface a una referencia a una clase y esta referencia no refiere a la clase **`OBJECT`** se tiene que usar este tipo de asignación:

```
<cref> ?= <iref>.
```

Para que esta asignación ocurra correctamente el objeto al cual apunta **`<iref>`** tiene que ser un objeto de la misma clase que el tipo de la variable referenciada a clase **`<cref>`**.

HERENCIA: VISIÓN DE CONJUNTO.

```
INTERFACE I1.
  DATA:        A1…
  METHODS:  M1…
  EVENTS :    E1…
ENDINTERFACE.
```

```
CLASS c1 DEFINITION.
  PUBLIC SECTION.
    INTERFACES   I1.
    DATA:            A1  …
  PROTECTED SECTION.
  …
  PRIVATE SECTION.
  …
ENDCLASS.
```

```
CLASS c1 IMPLEMENTATION.
  METHOD I1~M1.
    …
  ENDMETHOD.
  …
ENDCLASS.
```

El diagrama muestra la definición de la interface local I1 y la declaración y la implementación de una clase local c1 que implementa la interface I1 en su sección pública. El método de la interface I1~M1 se implementa en la clase. No se pueden implementar interface en ninguna otra sección de visibilidad. Los componentes de la interface amplían el ámbito público de la clase.

EJEMPLO DE INTERFACES

El siguiente ejemplo muestra cómo se puede usar una interface para implementar dos contadores diferentes pero que se pueden llamar de la misma manera.

```
*********************************************************************
INTERFACE I_COUNTER.
  METHODS: SET_COUNTER IMPORTING VALUE(SET_VALUE) TYPE I,
           INCREMENT_COUNTER,
           GET_COUNTER EXPORTING VALUE(GET_VALUE) TYPE I.
ENDINTERFACE.
*********************************************************************
CLASS C_COUNTER1 DEFINITION.
  PUBLIC SECTION.
    INTERFACES I_COUNTER.
  PRIVATE SECTION.
    DATA COUNT TYPE I.
ENDCLASS.
*********************************************************************
CLASS C_COUNTER1 IMPLEMENTATION.
  METHOD I_COUNTER~SET_COUNTER.
    COUNT = SET_VALUE.
  ENDMETHOD.
  METHOD I_COUNTER~INCREMENT_COUNTER.
    ADD 1 TO COUNT.
  ENDMETHOD.
  METHOD I_COUNTER~GET_COUNTER.
    GET_VALUE = COUNT.
  ENDMETHOD.
ENDCLASS.
*********************************************************************
CLASS C_COUNTER2 DEFINITION.
  PUBLIC SECTION.
    INTERFACES I_COUNTER.
  PRIVATE SECTION.
    DATA COUNT TYPE I.
ENDCLASS.
*********************************************************************
CLASS C_COUNTER2 IMPLEMENTATION.
  METHOD I_COUNTER~SET_COUNTER.
    COUNT = ( SET_VALUE / 10) * 10.
  ENDMETHOD.
  METHOD I_COUNTER~INCREMENT_COUNTER.
    IF COUNT GE 100.
      MESSAGE I042(00).
      COUNT = 0.
    ELSE.
      ADD 10 TO COUNT.
    ENDIF.
  ENDMETHOD.
  METHOD I_COUNTER~GET_COUNTER.
    GET_VALUE = COUNT.
  ENDMETHOD.
ENDCLASS.
*********************************************************************
```

La interface **i_counter** tiene tres métodos, **set_counter**, **increment_counter** y **get_counter**. Las clases **c_counter1** y **c_counter2** implementan la interface en sus secciones públicas. Ambas clases tienen que implementar los tres métodos de la interface en su parte de implementación. **c_counter1** es una clase para contadores que pueden tener un valor inicial y son incrementados de uno en uno. **c_counter2** es una clase para contadores que sólo pueden ser

incrementados de diez en diez. Ambas clases tienen el mismo aspecto visto desde el exterior. Su interface con el exterior está completamente definida por la interface ya que no hay componentes adicionales en ninguno de los métodos.

```
DATA cref1 TYPE REF TO c_counter1.
DATA cref2 TYPE REF TO c_counter2.
DATA iref  TYPE REF TO i_counter.
```

Se declaran dos variables referenciadas **cref1** y **cref2**,a las clases **c_counter1** y **c_counter2** respectivamente. También se declara una variable referenciada **iref** a la interface **i_counter**. Los valores de todas las referencias son **initial**.

```
DATA cref1 TYPE REF TO c_counter1.
DATA cref2 TYPE REF TO c_counter2.
DATA iref  TYPE REF TO i_counter.
CREATE OBJECT: cref1, cref2.
```

La sentencia **CREATE OBJECT** crea un objeto de cada clase a los cuales apuntan las referencias en **cref1** y **cref2**.

```
DATA cref1 TYPE REF TO c_counter1.
DATA cref2 TYPE REF TO c_counter2.
DATA iref  TYPE REF TO i_counter.
CREATE OBJECT: cref1, cref2.
iref = cref1.
```

Cuando la referencia de cref1 se asigna a iref, la referencia en iref apunta también al objeto con nombre interno **c_counter<1>**.

DISPARAR Y MANEJAR EVENTOS

En ABAP Objects hay ciertos métodos que se conocen como disparadores (*triggers*) y otros que se conocen como manejadores (*handlers*). Los triggers son los métodos que disparan un evento, mientras que los handlers son los métodos que se ejecutan cuando ocurre un evento.

Eventos disparadores

Para disparar un evento una clase tiene que:

- declarar el evento en la parte declarativa
- disparar el evento en uno de sus métodos.

Declaracion de eventos

Los eventos se declaran en la parte declarativa de una clase o en una interface. Para declarar eventos dependientes de instancia se usa la sentencia:

EVENTS <evt> EXPORTING... VALUE(<e_i>) TYPE type [OPTIONAL]..

Para declarar eventos estáticos se usa la sentencia:

CLASS-EVENTS <evt>...

Ambas sentencias tienen la misma sintaxis.

Cuando se declara un evento se puede usar la adición **EXPORTING** para especificar parámetros que se pasan al manejador del evento. Los parámetros se pasan siempre por valor. Los eventos dependientes de instancia siempre contienen el parámetro implícito **SENDER**, el cual tiene el tipo de una referencia al tipo o a la interface en la cual el evento es declarado.

Disparar eventos

Un evento dependiente de instancia en una clase puede ser disparado por cualquier método en la clase. Los eventos estáticos son disparados por métodos estáticos. Para disparar un evento en un método se usa la siguiente sentencia:

RAISE EVENT <evt> EXPORTING... <e_i> = <f $_i$>...

Por cada parámetro formal **<e_i>** que no esté definido como opcional se tiene que pasar el correspondiente parámetro real **<f $_i$>** en la adición **EXPORTING**. La referencia a sí mismo **ME** is pasada automáticamente al parámetro implícito **SENDER**.

Eventos manejadores

Los eventos se usan para ejecutar una serie de métodos. Estos métodos tienen que:

- estar definidos como eventos manejadores (handler) de ese evento

– estar registrados en tiempo de ejecución para el evento.

Declaración de métodos manejadores de eventos

Una clase puede contener métodos manejadores de eventos para eventos tanto de su propia clase como de otras clases. Para declarar un método manejador de eventos dependiente de instancia se usa la siguiente sentencia:

METHODS <meth> FOR EVENT <evt> OF <cif> IMPORTING.. <e_i>..

Para métodos estáticos se usa la misma sentencia con **CLASS-METHODS** en vez de **METHODS**. **<evt>** es un evento declarado en la clase o en la interface **<cif>**.

La interface de un método manejador de eventos sólo puede contener parámetros formales definidos en la declaración del evento. Los atributos de los parámetros también son adoptados por el evento. El método manejador de eventos no tiene por que usar todos los parámetros pasados en la sentencia **RAISE EVENT**. Si se quiere usar también el parámetro implícito **SENDER**, se tiene que listar en la interface. Este parámetro permite al manejador dependiente de instancia acceder al disparador para por ejemplo permitir devolver resultados. Si se declara un método manejador de eventos en una clase quiere decir que las instancias de la clase o la misma clase va a ser en principio capaz de manejar un evento disparado en un método.

Registro de métodos manejadores de eventos

Para permitir a un método manejador de eventos reaccionar a un evento, se tiene que determianr en tiempo de ejecución el disparador al cual va a reaccionar. Esto se hace con la siguiente sentencia:

SET HANDLER... <h_i>... [FOR]...

Esta sentencia relaciona los métodos manejadores de eventos con sus correspondientes métodos. Hay cuatro tipos diferentes de eventos:

– Eventos dependientes de instancia declarados en una clase.
– Eventos dependientes de instancia declarados en una interface.
– Eventos estáticos declarados en una clase.
– Eventos estáticos declarados en una interface.

La sintaxis y el efecto de la sentencia **SET HANDLER** depende de cual de los cuatro casos de arriba tenga lugar.

Para un evento dependiente de instancia se tiene que usar la adición **FOR** para especificar la instancia para la cual se quiere registrar el manejador. Se puede especificar una sóla instancia como disparador usando una variable referenciada **<ref>**:

SET HANDLER... <h_i>...FOR <ref>.

o se puede registrar el manejador para todas las instancias que puedan disparar el evento:

SET HANDLER... <h_i>...FOR ALL INSTANCES.

En este caso el registro se aplica incluso a las instancias que aún no han sido creadas cuando se registra el manejador.

No se puede usar la adición **FOR** para los eventos estáticos:

SET HANDLER... <h_i>...

El registro se aplica automáticamente a la clase entera o a todas las clases que implementan la interface que contiene el evento estático. En el caso de las interfaces, el registro también se aplica a las clases que aún no han sido cargadas cuando el manejador se registra.

Coordinación en el manejo de eventos

Después de la sentencia **RAISE EVENT**, todos los métodos manejadores registrados son ejecutados antes de que la siguiente sentencia sea procesada (manejo de eventos sincrónico). Si un método manejador de eventos desencadena eventos, los correspondientes métodos manejadores de eventos son ejecutados antes de que el método manejador original continue. Para evitar la posibilidad de un bucle infinito, actualmente los eventos sólo se pueden anidar 64 niveles.

Los métodos manejadores de eventos son ejecutados en el orden en el que son registrados. Debido a que los manejadores de eventos son registrados dinámicamente, no se puede saber el orden en el que serán procesados. Por esto se deben programar todos los manejadores de eventos como si se fuesen a ejecutar todos simultáneamente.

EVENTOS: VISIÓN DE CONJUNTO

Tenemos dos clases c1 y c2:

La clase c1 contiene un evento e1, el cual es desencadenado por el método m1. La clase c2 contiene un método m2 el cual responde al evento e1 de la clase c1.

CLASE DISPARADORA DEL EVENTO

```
CLASS C1 DEFINITION.
  PUBLIC SECTION.
    EVENTS E1 EXPORTING VALUE(P1) TYPE I.
    METHODS M1.
  PRIVATE SECTION.
    DATA A1 TYPE I.
ENDCLASS.
CLASS C1 IMPLEMENTATION.
  METHOD M1.
    A1 = …
    RAISE EVENT E1 EXPORTING P1 = A1.
  ENDMETHOD.
ENDCLASS.
```

CLASE MANEJADORA DEL EVENTO

```
CLASS C2 DEFINITION.
  PUBLIC SECTION.
    METHODS M2 FOR EVENT E1 OF C1
                          IMPORTING P1.
  PRIVATE SECTION.
    DATA A2 TYPE I.
ENDCLASS.
CLASS C2 IMPLEMENTATION.
  METHOD M2.
    A2 = P1.
    …
  ENDMETHOD.
ENDCLASS.
```

El registro del manejador se haría como sigue:

```
DATA r1 TYPE REF TO c_1.
DATA h1 TYPE REF TO c_2.
DATA h2 TYPE REF TO c_2.
CREATE OBJECT: r1, h1, h2.
SET HANDLER h1->m2 h2->m2 FOR r1.
CALL METHOD r1->m1.
```

El programa crea una instancia de la clase **c1** y dos de la clase **c2**. Los valores de las variables referenciadas **r1**, **h1** y **h2** apuntan a estas instancias.
La sentencia **SET HANDLER** crea una tabla de manejadores invisible al usuario para cada evento para el cual un método manejador ha sido registrado. Esta tabla contiene los nombres de los métodos manejadores de eventos y de las referencias de las instancias registradas. Las entradas de esta tabla son administradas dinámicamente por la sentencia **SET HANDLER**. Una referencia a una instancia en esta tabla es como una referencia en una variable referenciada. En otras palabras, cuentan como utilizaciones de la instancia, lo cual afecta directamente al tiempo de vida de la instancia. Por ejemplo, las instancias C2<1> y C2<2> no son borradas totalmente aunque las variables h1 y h2 sean inicializadas debido a que continuan registradas en la tabla de manejadores.
Para los eventos estáticos, el sistema crea una tabla de manejadores independiente de instancia para la clase dada. Cuando un evento es desencadenado, el sistema busca en la tabla correspondiente y ejecuta el método en las instancias adecuadas (o en la clase correspondiente si se trata de un método manejador estático.

EVENTOS: EJEMPLO

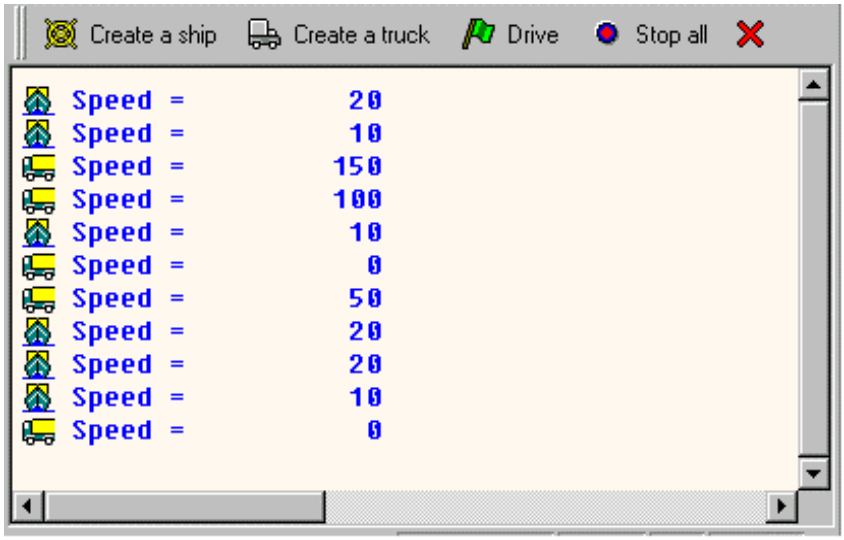
El siguiente ejemplo muestra cómo se trabaja con eventos en ABAP Objects:

```
REPORT demo_class_counter_event.
*********************************************************************
CLASS counter DEFINITION.
  PUBLIC SECTION.
    METHODS increment_counter.
      EVENTS critical_value EXPORTING value(excess) TYPE i.
  PRIVATE SECTION.
    DATA: count TYPE i,
          threshold TYPE i VALUE 10.
ENDCLASS.
*********************************************************************
CLASS counter IMPLEMENTATION.
  METHOD increment_counter.
    DATA diff TYPE i.
    ADD 1 TO count.
    IF count > threshold.
      diff = count - threshold.
      RAISE EVENT critical_value EXPORTING excess = diff.
    ENDIF.
  ENDMETHOD.
ENDCLASS.
*********************************************************************
CLASS handler DEFINITION.
  PUBLIC SECTION.
    METHODS handle_excess
            FOR EVENT critical_value OF counter
            IMPORTING excess.
ENDCLASS.
*********************************************************************
CLASS handler IMPLEMENTATION.
  METHOD handle_excess.
    WRITE: / 'Excess is', excess.
  ENDMETHOD.
ENDCLASS.
*********************************************************************

DATA: r1 TYPE REF TO counter,
h1 TYPE REF TO handler.

START-OF-SELECTION.
  CREATE OBJECT: r1, h1.
  SET HANDLER h1->handle_excess FOR ALL INSTANCES.
  DO 20 TIMES.
    CALL METHOD r1->increment_counter.
  ENDDO.
```

La clase **counter** implementa un contador. Se desencadena el evento **critical_value** cuando el valor umbral (**threshold**) es excedido, y se visualiza la diferencia. La clase **handler** puede manejar las excepciones en la clase **counter**. El manejador es registrado en tiempo de ejecución para todas las variables referenciadas que apunten al objeto.
El siguiente ejemplo muestra cómo declarar, llamar y manejar eventos en ABAP Objects. Este objeto trabaja con la lista interactiva desplegada debajo. Cada interacción del usuario desencadena un evento en ABAP Objects. La lista y sus datos son creados en la clase **c_list**. Hay una clase **status** para procesar las acciones del usuario. Se desencadena el evento **button_clicked** en el evento at **user-command**. El evento es manejado en la clase **c_list**. Hay un objeto de la clase **c_ship** o **c_truck** para cada linea de la lista. Ambas clases implementan la interface **i_vehicle**. Cuando la velocidad de cualquiera de los objetos cambia, el evento **speed_change** es desencadenado. La clase **c_list** reacciona a esto actualiza la lista.



Restricciones

Las sentencias ABAP usadas para procesar listas aún no están completamente disponibles en ABAP Objects, aunque sí algunas de ellas. Lo mismo ocurre con el resto de sentencias ABAP, no todas están disponibles en ABAP Objects.

Declaración

Se declaran los siguientes eventos en el ejemplo:

- El evento dependiente de instancia **`speed_change`** en la interface **`i_vehicle`**.
- El evento estático **`button_clicked`** en la clase **`status`**.

La clase **`c_list`** tiene los métodos manejadores de eventos de ambos eventos.
La clase **`status`** no tiene ningún atributo y por lo tanto sólo trabaja con métodos y eventos estáticos.

```
*********************************************************************
* Interface and Class declarations
*********************************************************************
INTERFACE I_VEHICLE.
  DATA MAX_SPEED TYPE I.
  EVENTS SPEED_CHANGE EXPORTING VALUE(NEW_SPEED) TYPE I.
  METHODS: DRIVE,
           STOP.
ENDINTERFACE.
*********************************************************************
CLASS C_SHIP DEFINITION.
  PUBLIC SECTION.
    METHODS CONSTRUCTOR.
    INTERFACES I_VEHICLE.
  PRIVATE SECTION.
    ALIASES MAX FOR I_VEHICLE~MAX_SPEED.
    DATA SHIP_SPEED TYPE I.
ENDCLASS.
*********************************************************************
CLASS C_TRUCK DEFINITION.
  PUBLIC SECTION.
    METHODS CONSTRUCTOR.
    INTERFACES I_VEHICLE.
  PRIVATE SECTION.
    ALIASES MAX FOR I_VEHICLE~MAX_SPEED.
    DATA TRUCK_SPEED TYPE I.
ENDCLASS.
*********************************************************************
```

```
CLASS STATUS DEFINITION.
  PUBLIC SECTION.
    CLASS-EVENTS BUTTON_CLICKED EXPORTING VALUE(FCODE) LIKE SY-UCOMM.
    CLASS-METHODS: CLASS_CONSTRUCTOR,
                   USER_ACTION.
ENDCLASS.
*****************************************************************
CLASS C_LIST DEFINITION.
  PUBLIC SECTION.
    METHODS: FCODE_HANDLER FOR EVENT BUTTON_CLICKED OF STATUS
                                          IMPORTING FCODE,
             LIST_CHANGE FOR EVENT SPEED_CHANGE OF I_VEHICLE
                                          IMPORTING NEW_SPEED,
             LIST_OUTPUT.
  PRIVATE SECTION.
    DATA: ID TYPE I,
          REF_SHIP TYPE REF TO C_SHIP,
          REF_TRUCK TYPE REF TO C_TRUCK,
    BEGIN OF LINE,
      ID TYPE I,
      FLAG,
      IREF TYPE REF TO I_VEHICLE,
      SPEED TYPE I,
    END OF LINE,
    LIST LIKE SORTED TABLE OF LINE WITH UNIQUE KEY ID.
ENDCLASS.
*****************************************************************
```

Implementación

El método estático **user_action** de la clase **status** desencadena el evento estático **button_clicked**. Los métodos dependientes de instancia **i_vehicle~drive** y **i_vehicle~stop** desencadenan el evento dependiente de instancia **i_vehicle~speed_change** en las clases **c_ship** y **c_truck**.

```
*****************************************************************
* Implementations
*****************************************************************
  CLASS C_SHIP IMPLEMENTATION.
    METHOD CONSTRUCTOR.
      MAX = 30.
    ENDMETHOD.
    METHOD I_VEHICLE~DRIVE.
      CHECK SHIP_SPEED < MAX.
      SHIP_SPEED = SHIP_SPEED + 10.
      RAISE EVENT I_VEHICLE~SPEED_CHANGE
         EXPORTING NEW_SPEED = SHIP_SPEED.
    ENDMETHOD.
    METHOD I_VEHICLE~STOP.
      CHECK SHIP_SPEED > 0.
      SHIP_SPEED = 0.
      RAISE EVENT I_VEHICLE~SPEED_CHANGE
         EXPORTING NEW_SPEED = SHIP_SPEED.
    ENDMETHOD.
  ENDCLASS.
*****************************************************************
  CLASS C_TRUCK IMPLEMENTATION.
    METHOD CONSTRUCTOR.
      MAX = 150.
    ENDMETHOD.
```

```
    METHOD I_VEHICLE~DRIVE.
      CHECK TRUCK_SPEED < MAX.
      TRUCK_SPEED = TRUCK_SPEED + 50.
      RAISE EVENT I_VEHICLE~SPEED_CHANGE
        EXPORTING NEW_SPEED = TRUCK_SPEED.
    ENDMETHOD.
    METHOD I_VEHICLE~STOP.
      CHECK TRUCK_SPEED > 0.
      TRUCK_SPEED = 0.
      RAISE EVENT I_VEHICLE~SPEED_CHANGE
        EXPORTING NEW_SPEED = TRUCK_SPEED.
    ENDMETHOD.
  ENDCLASS.
*****************************************************************
  CLASS STATUS IMPLEMENTATION.
    METHOD CLASS_CONSTRUCTOR.
      SET PF-STATUS 'VEHICLE'.
      WRITE 'Click a button!'.
    ENDMETHOD.
    METHOD USER_ACTION.
      RAISE EVENT BUTTON_CLICKED EXPORTING FCODE = SY-UCOMM.
    ENDMETHOD.
  ENDCLASS.
*****************************************************************
  CLASS C_LIST IMPLEMENTATION.
    METHOD FCODE_HANDLER .
      CLEAR LINE.
      CASE FCODE.
        WHEN 'CREA_SHIP'.
          ID = ID + 1.
          CREATE OBJECT REF_SHIP.
          LINE-ID = ID.
          LINE-FLAG = 'C'.
          LINE-IREF = REF_SHIP.
          APPEND LINE TO LIST.
        WHEN 'CREA_TRUCK'.
          ID = ID + 1.
          CREATE OBJECT REF_TRUCK.
          LINE-ID = ID.
          LINE-FLAG = 'T'.
          LINE-IREF = REF_TRUCK.
          APPEND LINE TO LIST.
        WHEN 'DRIVE'.
          CHECK SY-LILLI > 0.
          READ TABLE LIST INDEX SY-LILLI INTO LINE.
          CALL METHOD LINE-IREF->DRIVE.
        WHEN 'STOP'.
          LOOP AT LIST INTO LINE.
            CALL METHOD LINE-IREF->STOP.
          ENDLOOP.
        WHEN 'CANCEL'.
          LEAVE PROGRAM.
      ENDCASE.
      CALL METHOD LIST_OUTPUT.
    ENDMETHOD.
    METHOD LIST_CHANGE .
      LINE-SPEED = NEW_SPEED.
      MODIFY TABLE LIST FROM LINE.
    ENDMETHOD.
    METHOD LIST_OUTPUT.
      SY-LSIND = 0.
```

```
      SET TITLEBAR 'TIT'.
      LOOP AT LIST INTO LINE.
        IF LINE-FLAG = 'C'.
          WRITE / ICON_WS_SHIP AS ICON.
        ELSEIF LINE-FLAG = 'T'.
          WRITE / ICON_WS_TRUCK AS ICON.
        ENDIF.
        WRITE: 'Speed = ', LINE-SPEED.
      ENDLOOP.
    ENDMETHOD.
ENDCLASS.
*****************************************************************
```

Utilización de las clases en un programa

El programa crea un objeto de la clase **c_list** y registra el método manejador de eventos **fcode_handler** del objeto para el evento estático **button_clicked**. También registra el método manejador de eventos **list_change** para el evento **speed_change** de todas las instancias que implementan la interface **i_vehicle**.

```
REPORT OO_EVENTS_DEMO NO STANDARD PAGE HEADING.
*****************************************************************
* Global data of program
*****************************************************************
  DATA LIST TYPE REF TO C_LIST.
*****************************************************************
* Program events
*****************************************************************
START-OF-SELECTION.
  CREATE OBJECT LIST.
  SET HANDLER: LIST->FCODE_HANDLER,
  LIST->LIST_CHANGE FOR ALL INSTANCES.
*****************************************************************
AT USER-COMMAND.
CALL METHOD STATUS=>USER_ACTION.
*****************************************************************
```

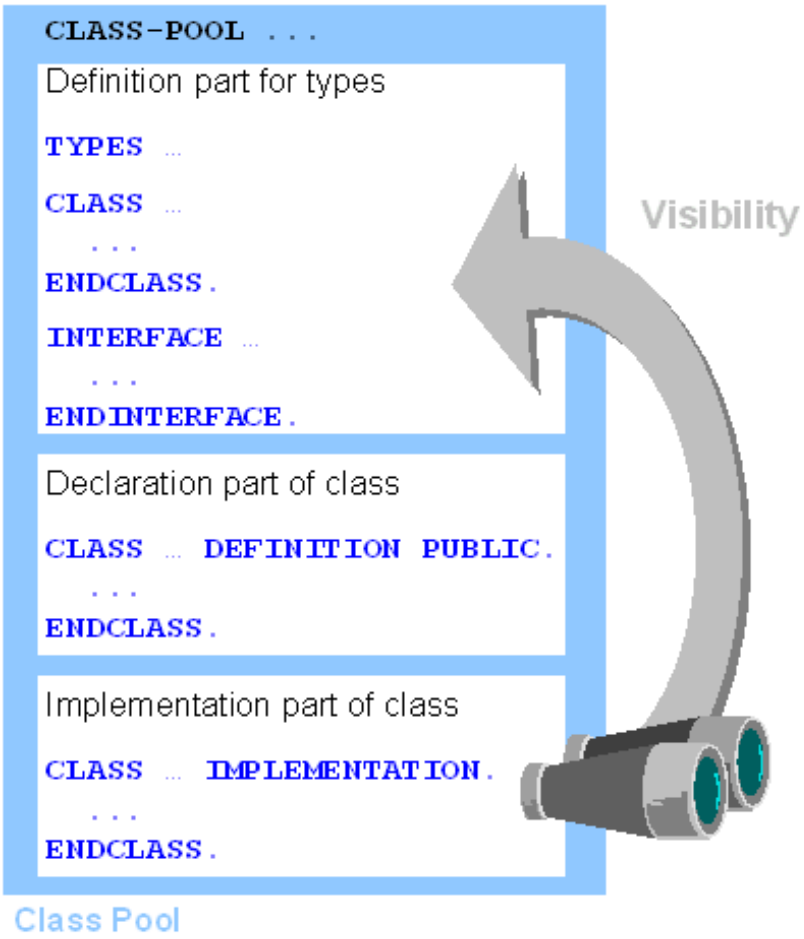
POOLS DE CLASES

Clases globales y interfaces

Las clases y las interfaces son tipos de objetos. Se pueden definir o bien globalmente en R/3 Repository o bien localmente en un programa ABAP. Si se definen clases e interfaces globalmente, entonces son almacenadas en un tipo especial de programas ABAP llamados class pools (tipo K) o interface pools (tipo J) los cuales sirven como contenedores de los respectivos tipos de objetos. Cada pool de clases o interfaces contiene la definición de una sóla clase o interface. El programa es automáticamente generado por el constructor de clases cuando se crea la clase o la interface.
Un pool de clases es comparable a un modul pool o a un grupo de funciones. Contiene sentencias ABAP, tanto declarativas como ejecutables, pero no puede ser ejecutado por si mismo. El sistema sólo puede ejecutar las sentencias en el pool de clases a petición, esto es, cuando la sentencia **CREATE OBJECT** crea una instancia de la clase.
Los pools de interfaces no contienen ninguna sentencia ejecutable. Son usados como contenedores de las definiciones de la interface. Cuando se implementa una interface en una clase, la definición de la interface queda implícitamente incluida en la definición de la clase.

Estructura de un pool de clases

Los pools de clases se estructuran de la siguiente manera:



Los pools de clases contienen una parte de definiciones para las declaraciones de los tipos, otra parte para la declaración de la clase y otra para la implementación de la clase.

Diferencias con otros programas ABAP

Los pools de clases son diferentes de otros programas ABAP por las siguientes razones:

- Los programas ABAP como los ejecutables, los modul pool o los módulos de funciones normalmente tienen una parte declarativa en la que se definen los datos globales del programa. Estos datos son visibles en todos los bloques de proceso del programa. Los pool de clases en cambio tienen una parte de definición en la cual se definen datos y tipos de objetos, pero no datos de objetos o field-symbols. Los tipos que se definen en un pool de clases son sólo visibles en la parte de implementación de la clase global.
- Los únicos bloques de proceso que se pueden usar son la parte declarativa y la parte de implementación de la clase global. La parte de implementación sólo puede implementar los métodos declarados en la clase global. No se pueden usar ninguno de los demás bloques de proceso de ABAP (módulos de diálogo, bloques de eventos, subrutinas, módulos de funciones).
- Los bloques de procesamiento de los pools de clases no están controlados por el entorno de ejecución ABAP. No tiene lugar ningún evento, ni se puede llamar a ningún módulo de datos ni a ningún procedimiento. Los pools de clases sirven exclusivamente para programar clases. Sólo se puede acceder a los datos y a las funciones de una clase usando su interface.
- Debido a que los eventos y los módulos de diálogo no están permitidos en las clases no se pueden procesar pantallas en las clases. Actualmente no se pueden programar listas ni pantallas de selección en las clases ya que no pueden reaccionar a los eventos apropiados.

Clases locales y pools de clases

Las clases y las interfaces que se definen en la parte de definiciones de un pool de clases no son visibles externamente. Dentro del pool de clases tienen la misma función que las clases y las interfaces locales en cualquier programa ABAP. Las clases locales sólo serán instanciadas en los métodos de la clase global.

Debido a que las subrutinas no están permitidas en los pool de clases, las clases locales son la única manera posible de modularización en las clases globales. Las clases locales respecto de las clases globales tienen más o menos la misma función que las subrutinas en los módulos de funciones, con la excepción de que no son visibles externamente.

CONSTRUCTOR DE CLASES

El constructor de clases es una herramienta que permite crear, definir y probar las clases e interfaces globales.

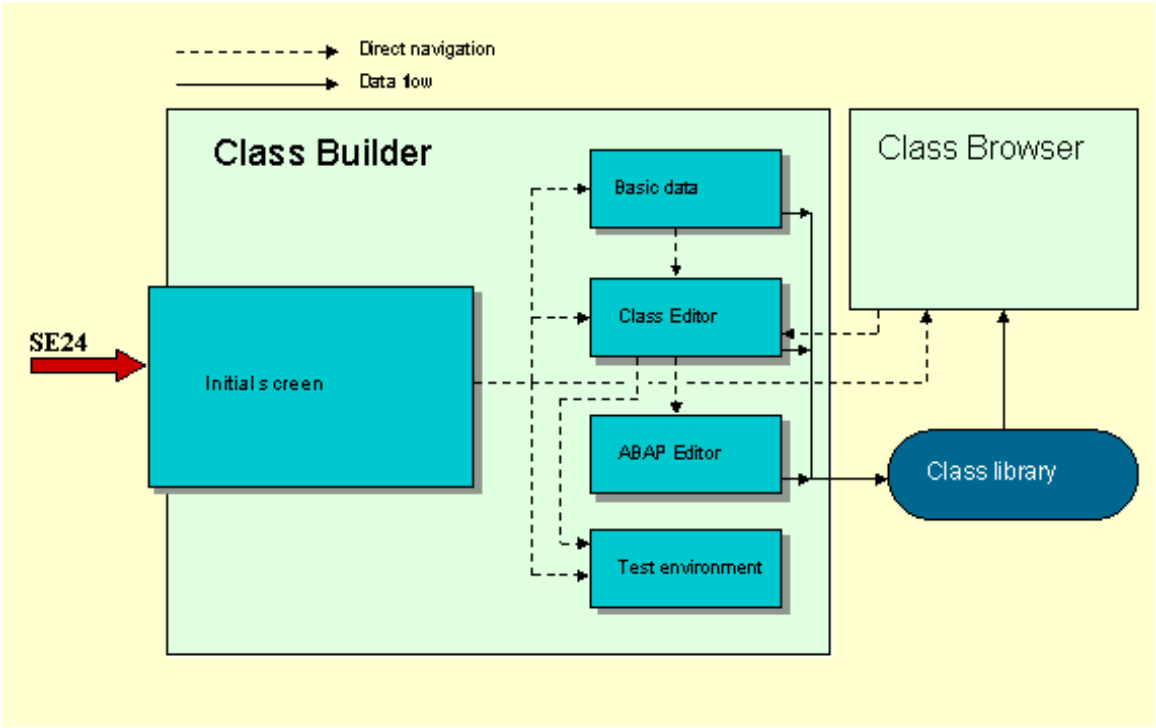
INTRODUCCIÓN AL CONSTRUCTOR DE CLASES

Propósito

El constructor de clases permite crear y mantener clases e interfaces globales. Ambos tipos de objetos, al igual que los tipos de datos globales son definidos en el R/3 Repository. Forman una librería de clases central y son visibles en todo el sistema. Se pueden visualizar las clases e interfaces existentes en la librería de clases usando el class browser.
Se pueden definir tanto clases locales como clases globales. Las clases locales se definen en programas, grupos de funciones o en un pool de clases de clases auxiliares dentro de una clase global. Son sólo visibles en el módulo en el que son definidas.

Integración

El constructor de clases es una herramienta completamente integrada en el ABAP Workbench que permite crear, visualizar y mantener tipos de objetos globales de la librería de clases.
El siguiente diagrama ilustra la arquitectura del constructor de clases y las relaciones entre sus componentes.



Para acceder a la pantalla inicial del constructor de clases se utiliza la taransacción **SE24** o bien se accede desde la pantalla inicial del ABAP Workbench en Desarrollo → Constructor de clases. Desde la pantalla inicial del constructor de clases se puede o bien visualizar los contenidos de la librería de clases o bien crear una clase usando el editor. Una vez que se haya definido el tipo de objeto se pueden implementar sus métodos. Desde la pantalla inicial del constructor de clases se puede acceder también la entorno de test del constructor de clases.

Tipos de objetos ya existentes

Para visualizar los tipos de objetos existentes en la librería de clases se usa el class browser. Se puede usar también el sistema de información del repository. Desde la pantalla inicial del ABAP Workbench se escoge: Resumen → Sistema de información. Desde dentro se escoge ABAP Objects y se puede abrir todo o parte de la librería de clases.
El class browser está integrado en el constructor de clases, se puede iniciar desde el constructor o bien desde la transacción **CLABAP**.
Existe un rango preconfigurado de vistas que se pueden usar para visualizar tipos de objetos. Se puede seleccionar por una serie de filtros:

- Todas las clases. Se visualizan todas las clases e interfaces de la librería de clases.
- Objetos de negocio (business objects). Se visualizan los tipos de objetos de negocio de la librería de clases
- Otras selecciones. Se pueden usar otros tres filtros:
  - Tipos de objetos
  - Relaciones entre objetos
  - Otros

No se pueden crear nuevos tipos de objetos desde el class browser.

Crear tipos de objetos

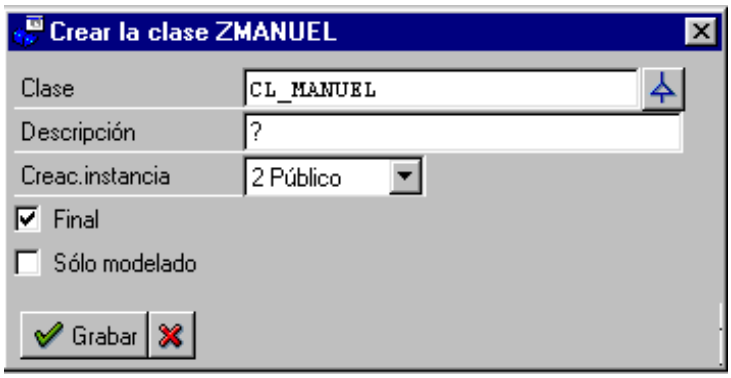
Cuando se crean clases o interfaces sólo se especifican sus datos básicos. Esta definición crea una entrada en la tabla del diccionario para el tipo de objeto dado. Una vez que se han definido los datos básicos se puede continuar trabajando en los componentes en más detalle. Una vez creados los datos básicos, el sistema abre automáticamente el editor de clases.

CREAR NUEVAS CLASES

Para crear una nueva clase desde la pantalla inicial del ABAP Workbench:

1) En tipo de objeto se escribe el nombre de la nueva clase.
2) Se selecciona el tipo de objeto clase.

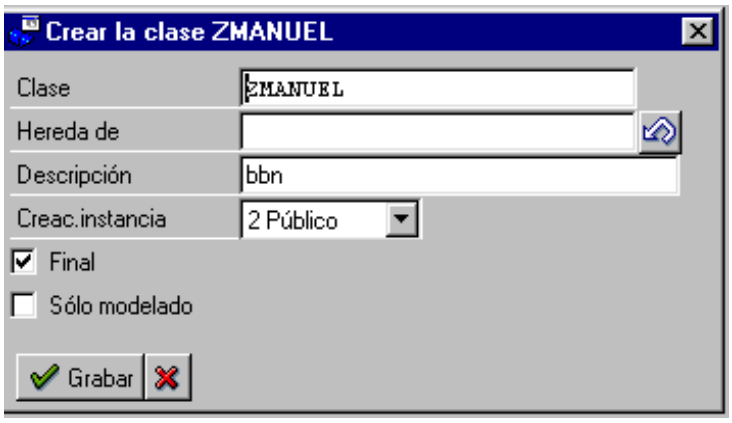
3) Se elige crear. Aparece el siguiente cuadro de diálogo:



4) Se definen los datos básicos, que son la siguiente información:
   - Clase – nombre de la nueva clase.
   - Descripción – descripción de la nueva clase.
   - Crear instancia – el valor por defecto es público. Esto quiere decir que cualquier usuario puede instanciar la clase usando la sentencia **`CREATE OBJECT`**. Si se selecciona protegido, sólo la

propia clase y sus subclases pueden instanciar la clase. Si se elige privado sólo la propia clase puede instanciarse (desde uno de sus métodos). Si se selecciona abstracto se crea una clase que no puede ser instanciada. Las clases abstractas sirven como plantillas para las subclases y sólo se puede acceder a ellas usando sus atributos estáticos o los de sus subclases.

- Final – si se selecciona final, se define una clase final. Una clase final es la última del árbol de herencia y no puede tener subclases. Si se crea una clase abstracta final sólo se puede acceder a sus componentes estáticos.
- Modelado – si se selecciona el sistema no definirá la subclase en el pool de clases
- Icono de herencia – Si se selecciona esta función (botón al lado de la clase) aparece un cuadro de diálogo. Desde aquí se puede introducir el nombre de una superclase. La superclase puede ser cualquier clase de la librería de clases que no esté definida como final.



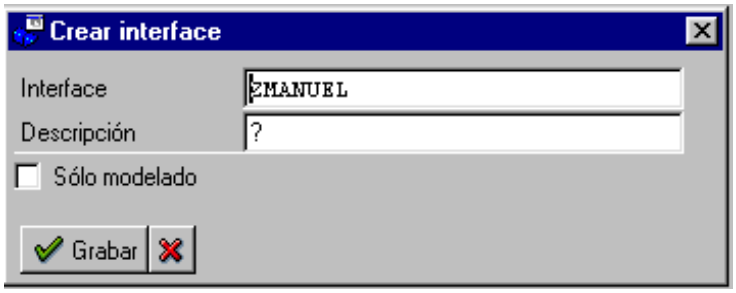
5) Se elige grabar y aparece el cuadro de diálogo en el que se indica la clase de desarrollo.
6) Se graba y aparece el editor de clases. Desde aquí se pueden definir los componentes de la clase e incluir interfaces.

Ya se ha definido una nueva clase. Cuando se introducen los datos básicos el sistema crea un pool de clases para la nueva clase (siempre que no se haya definido como sólo modelado). Un pool de clases contiene la definición de una sóla clase global. Son similares a los grupos de funciones en el sentido de que se pueden definir clases locales auxiliares y tipos locales.

CREAR NUEVAS INTERFACES

Para crear una nueva interface desde la pantalla inicial del ABAP Workbench

1) Se escribe el nombre de la interface.
2) Se selecciona el tipo de objeto interface.
3) Se escoge crear. Aparece el cuadro de diálogo siguiente:



4) Se introduce la siguiente información:
   - Descripción – breve descripción de la interface.
   - Sólo modelado – Si se selecciona esta opción no se genera un pool de interfaces para la interface y no se puede acceder a ella en tiempo de ejecución. En el futuro esta opción permitirá diseñar interfaces basándose en un modelo gráfico.
5) Se elige grabar y aparece el cuadro de diálogo en el que se indica la clase de desarrollo.

6) Se graba y aparece el editor de clases. Desde aquí se pueden definir los componentes de la interface. Una vez definida la interface se puede incluir en la definición de las clases.

Se crea una nueva interface con sus datos básicos en el diccionario. El sistema genera un pool de interfaces para la interface siempre que no se haya creado con la opción sólo modelado.

DEFINICIÓN DE COMPONENTES

Una vez creada una clase o una interface como se acaba de describir accedemos a la pantalla desde la cual podemos definir los componentes de nuestra clase o interface. Desde esta pantalla podemos:

- Definir las clases o las interfaces asignándoles sus componentes.
- Implementar los métodos de las clases.
- Añadir interfaces a las clases e implementar sus métodos en las clases.
- Cambiar las definiciones ya existentes y la implementación de las clases.
- Definir tipos de datos locales dentro de las clases.

Características

Asignamos componentes definiendo:

- Atributos.
- Métodos.
- Eventos.
- Tipos locales en las clases.
- Interfaces.

En la pestaña métodos podemos:

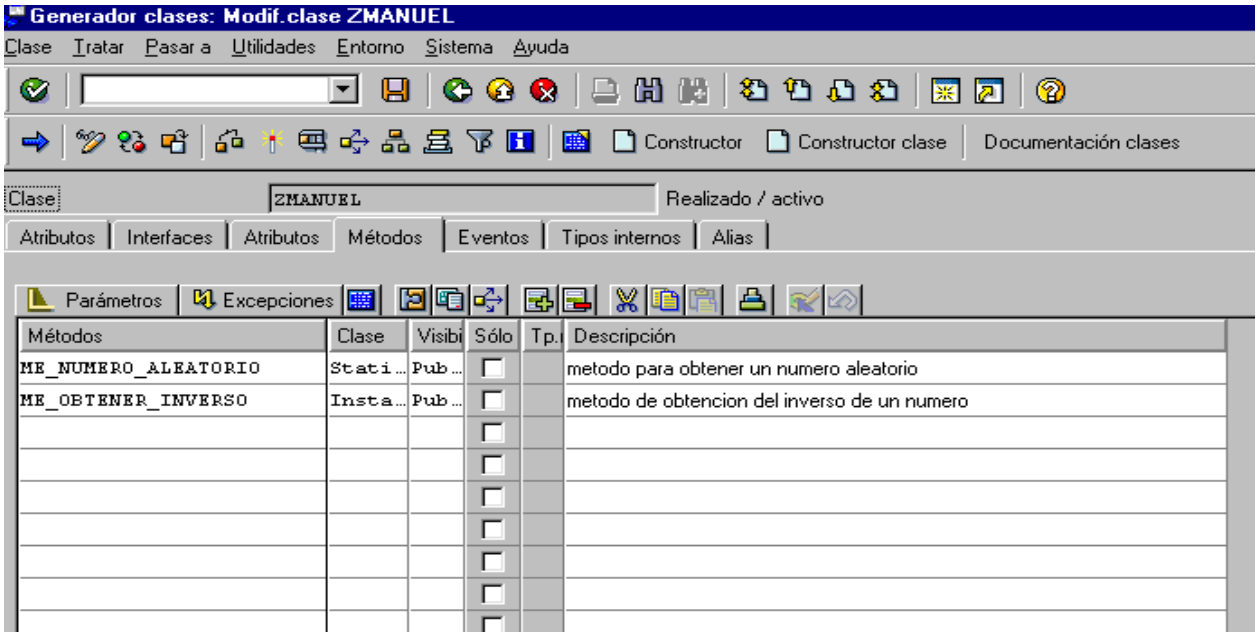
- Definir los parámetros de los métodos.
- Definir las excepciones de los métodos.
- Implementar los métodos.

En la pestaña interfaces podemos:

- Asignar interfaces a las clases.
- Implementar los métodos de las interfaces en las clases.

EDITOR DE CLASES

El editor de clases es una parte del constructor de clases en la que se definen los atributos, métodos, eventos, y tipos de datos que conforman los componentes de uan clase. Las clases o interfaces definidas en el editor de clases son guardadas en la librería de clases. Se puede acceder al editor ABAP al escribir la implementación de un método.



Características

Las funciones básicas son las siguientes:
- Creación de atributos.
- Creación de métodos.
- Creación de eventos.
- Implementación de métodos.
- Creación de interfaces en clases.
- Creación de tipos internos en una clase.

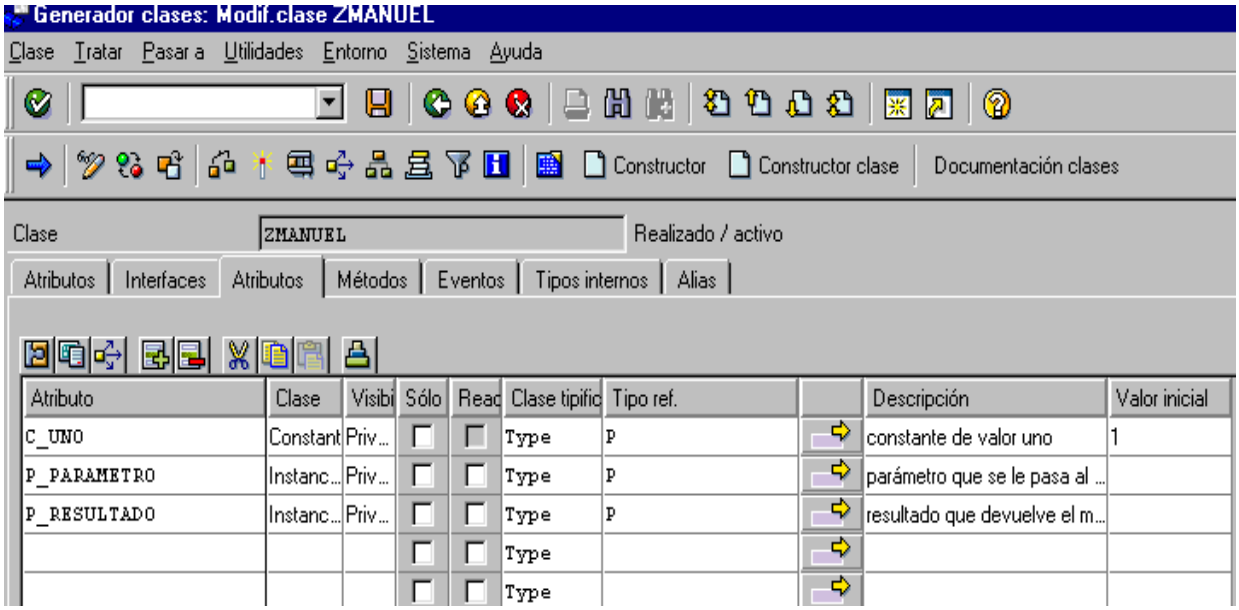
Otras funciones que se realizan desde aquí son:
- La función Clases Locales (definición de tipos locales, ctrl + F5) permite crear clases locales auxiliares en el pool de clases de la clase global.
- La función Alias permite definir alias para los componentes.
- La función Documentación permite documentar las clases o las interfaces y sus componentes.
- La función Pasar a permite acceder al código (parte pública, protegida o privada)

CREACIÓN DE ATRIBUTOS

Los atributos contienen datos. Son los que definen el estado de un objeto. Si se quieren crear atributos con tipos internos. Estos deberán estar creados antes de crear los atributos.

Procedimiento

Se va al editor de clases en modo modificar y se escoge la pestaña de atributos.



Para crear un atributo hay que rellenar los siguientes campos:
- Atributo – nombre que identifica al atributo.
- Clase – tipo de atributo, ya sea constante, dependiente de instancia o estático (compartido por todas las instancias de la clase)
- Visibilidad – Define la visibilidad de los atributos para los usuarios de la clase. Si un atributo es público, se asigna a la sección pública de la clase y puede acceder a él cualquier usuario de la clase. Los atributos públicos forman el punto de contacto externo de la clase. Los atributos protegidos son visibles unicamente desde las subclases de la clase.. Los atributos privados son sólo visibles desde la propia clase, y no son visibles ni siquiera para las propias subclases.
- Sólo modelado – Si se elige esta opción el sistema no guarda el componente en el pool de clases y el componente no puede ser llamado en tiempo de ejecución.
- Sólo lectura – mediante esta opción podemos impedir que el usuario cambie el valor del atributo.
- Clase tipif. – Para especificar la referencia al tipo ABAP. Puede ser **`TYPE`**, **`LIKE`** o **`TYPE REF TO`** para las referencias a clases.

- Tipo ref. – puede ser un tipo ABAP elemental o un tipo de objeto (clase o interface).
- Descripción – Descripción breve del componente.
- Valor inicial – Si el atributo es una constante es necesario especificar este campo.

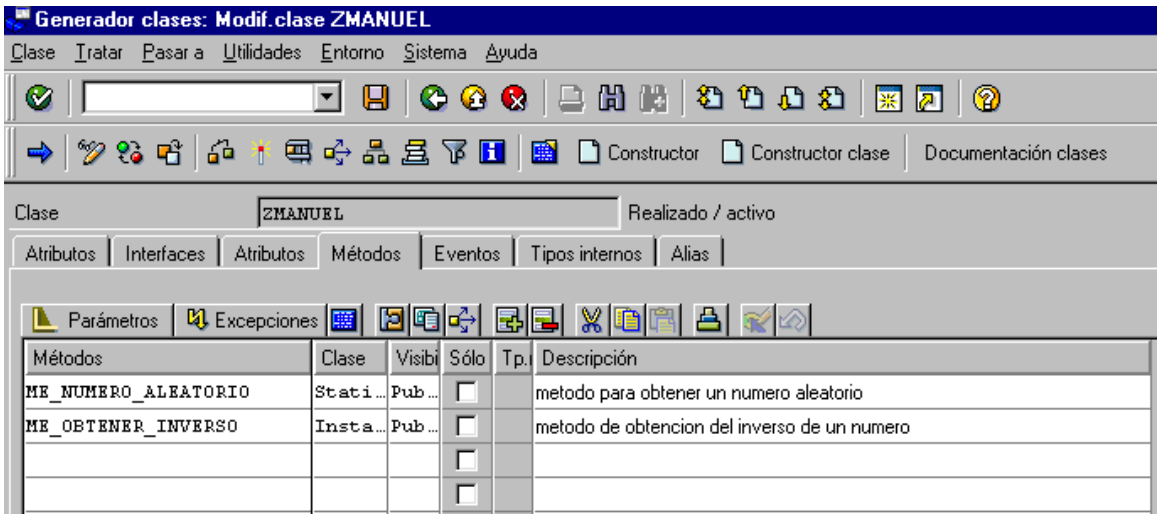
El sistema genera el correspondiente código ABAP en la definición de la clase o de la interface en el pool de clases para todos los atributos excepto para aquellos marcados como Sólo modelado.

CREACIÓN DE MÉTODOS

Los métodos describen el comportamiento de los objetos. Se implementan usando funciones definidas dentro de las clases. Los métodos son operaciones que cambian los atributos de una clase o una interface. Hay dos tipos de métodos, los métodos dependientes de instancia, que son aquellos que se refieren a cada instancia particular y los métodos estáticos, que son aquellos compartidos por todas las instancias de una clase. Los métodos estáticos sólo pueden usar atributos estáticos. Para crear los métodos es conveniente haber creado antes los atributos de la clase para así poder implementarlos directamente.

Procedimento

Se va al editor de clases en modo modificar y se escoge la pestaña de métodos.



Para crear un método hay que rellenar los siguientes campos:

- Métodos – nombre que identifica al método.
- Clase – tipo de método, dependiente de instancia o estático.
- Visibilidad – tipo de método, visible, protegido o privado. Un método privado sólo puede llamarse desde dentro de la clase, o sea desde otro método de la clase, mientras que un método protegido sólo puede llamarse desde otro método de la misma clase o bien de una subclase.
- Sólo modelado – Si se selecciona esta opción el sistema no guarda el método en el pool de clases.
- Descripción – Descripción breve del método.

Si se crea un método constructor, el sistema asigna el nombre **`CONSTRUCTOR`** o **`CLASS_CONSTRUCTOR`** automáticamente. El constructor de clases también define otros atributos en este caso.

Antes de implementar el método es conveniente crear sus parámetros y sus excepciones.

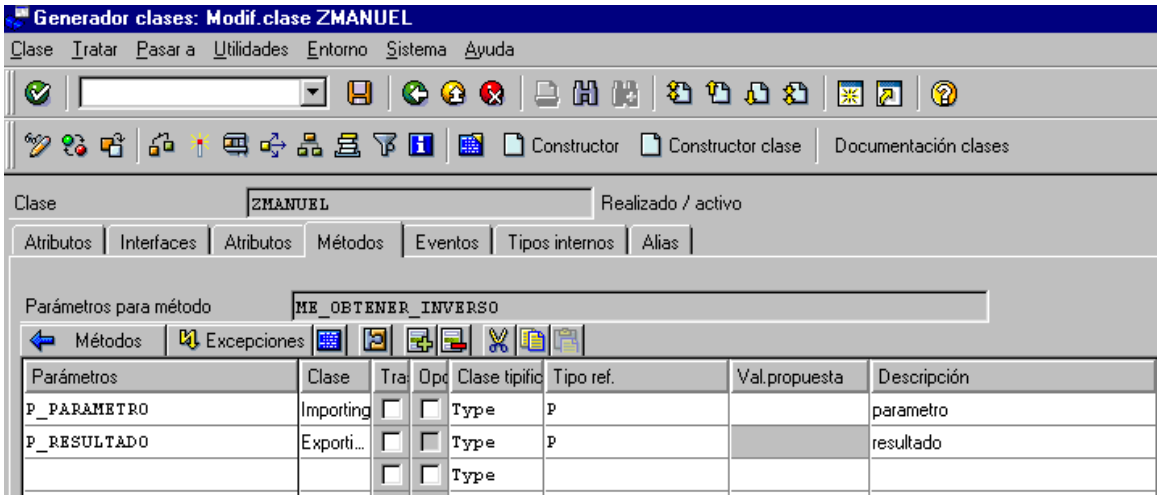
CREACIÓN DE PARÁMETROS Y EXCEPCIONES

Los métodos se definen de manera similar a los módulos de funciones. Primero, se crea la interface de parámetros y las excepciones, después se implementa el método. Los métodos pueden tener parámetros de entrada (importing o changing) y parámetros de salida (exporting, changing y returning).

Para crear los parámetros y las excepciones se tienen que haber creado antes los métodos, los atributos y los eventos de la clase o la interface. Cuando se redefinen métodos heredados, no se puede cambiar la interface de parámetros o añadir nuevos parámetros.

Procedimiento

Se posiciona en el nombre del método o del evento y se escoge parámetros.



Se necesita completar la siguiente información:

- Parámetro – Nombre que identifica al parámetro.
- Clase – Puede tener el tipo importing, exporting, changing o returning. Si se usan parámetros returning no se pueden usar ni parámetros exporting ni parámetros changing. Si se usan parámetros changing no se pueden usar parámetros returning. Los métodos constructores sólo pueden tener parámetros importing.
- Traspasar valores – al contrario que los módulos de funciones, el modo por defecto de pasar valores a un método es por referencia. Se puede forzar al sistema a pasar un parámetro por valor seleccionando esta opción. Esto es sólo posible para parámetros importing, exporting y changing. Los parámetros returning sólo pueden ser pasados por valor. El constructor de clases chequea esta regla para que se cumpla siempre.
- Opcional – Si se selecciona esta opción el parámetro no tiene que ser especificado necesariamente cuando se llama al método.
- Clase tipif. - Para especificar la referencia al tipo ABAP. Puede ser **`TYPE`**, **`LIKE`** o **`TYPE REF TO`** para las referencias a clases.
- Tipo ref. - puede ser un tipo ABAP elemental o un tipo de objeto (clase o interface). Se puede especificar el tipo de un parámetro de un método privado o protegido usando un tipo de datos interno definido en la clase.
- Valor propuesta – valor por defecto del parámetro.
- Descripción – descripción breve del parámetro.

Para crear excepciones se realiza un proceso similar, se posiciona sobre el nombre del método y se selecciona excepciones. Se necesitan rellenar los siguientes campos:

- Excepción – nombre único que identifica a la excepción.
- Descripción – descripción breve de la excepción.

IMPLEMENTACIÓN DE MÉTODOS

Para implementar un método hay que crear previamente el método y los atributos de la clase. Si se quieren implementar métodos de las interfaces, estas tienen que haber sido incluidas en la definición de la clase. Se tienen que haber creado igualmente todos los parámetros y excepciones necesarias.

Procedimiento

En el editor de clases se escoge la pestaña métodos. Se posiciona sobre el método elegido y se hace doble click o se elige código fuente con lo que se accede al editor ABAP. Ahí se escribe el código del método.

CREACIÓN DE EVENTOS

Los objetos pueden indicar que su estado ha cambiado disparando un evento. Los eventos se pueden definir tanto en clases como en interfaces. Los eventos se desencadenan desde un método mediante la sentencia **`RAISE EVENT`**. Cada clase o interface que va a manejar el correspondiente evento debe implementar el método manejador de eventos apropiado y registrarlo usando la sentencia **`SET HANDLER`**. Cuando un evento tiene lugar, el sistema llama al método manejador registrado para ese

evento. Al igual que las definiciones de los métodos, los eventos tienen una interface de parámetros. La única diferencia es que los eventos sólo pueden tener parámetros EXPORTING.

Procedimiento

En el editor de clases se escoge la pestaña Eventos. Se tiene que introducir la siguiente información:
- Evento – Nombre que identifica al evento.
- Clase – Especifica el tipo de evento, estático o dependiente de instancia.
- Visibilidad – Especifica el tipo de evento, público, protegido o privado.
- Sólo modelado – Si se selecciona esta opoción, el sistema no crea el evento en el pool de clases. No se podría entonces acceder al componente en tiempo de ejecución.
- Descripción – descripción breve del evento.

Los eventos se listan en la parte declarativa de la clase o la interface después de la sentencia **`EVENTS`**.

CREACIÓN DE TIPOS INTERNOS EN LAS CLASES

No se deben crear tipos de datos públicos dentro de las clases globales.

Procedimiento

Desde el editor de clases se elige la pestaña Tipos internos. Para crear un tipo interno dentro de una clase se tiene que rellenar la siguiente información:
- Tipo – Nombre que identifica al tipo.
- Visibilidad - tipo de método, visible, protegido o privado.
- Sólo modelado - Si se selecciona esta opoción, el sistema no crea el tipo en el pool de clases. No se podría entonces acceder al componente en tiempo de ejecución.
- Clase tipif. - Para especificar la referencia al tipo ABAP. Puede ser **`TYPE`**, **`LIKE`** o **`TYPE REF TO`** para las referencias a clases.
- Tipo ref. - puede ser un tipo ABAP elemental o un tipo de objeto (clase o interface).
- Descripción – descripción breve del tipo.

Con esto no se crean tipos de datos internos en la clase. Se pueden usar estos tipos en la clase y son protegidos también en las subclases (no pueden ser públicos). Se pueden definir atributos protegidos y privados y parámetros de interface usando la adición **`TYPE`**.

DEFINICIÓN DE RELACIONES ENTRE TIPOS DE OBJETOS

Se pueden definir las siguientes relaciones entre dos tipos de objetos:
- Herencia entre dos clases.
- Extensión de la funcionalidad de una clase mediante la implementación de interfaces. Esta se una relación entre clases e interfaces.
- Interfaces compuestas. Esta es una relación entre dos interfaces.

La herencia es una relación entre clases. Permite derivar una nueva clase a partir de la definición de una clase ya existente. La nueva clase se conoce como subclase mientras que la clase ya existente es la superclase. La herencia se utiliza para crear subclases más especializadas que la superclase. Se pueden añadir nuevos componentes a la subclase, incluso se pueden redefinir los métodos que hereda la subclase. El resultado de la herencia es una jerarquía de clases. El constructor de clases permite crear una jerarquía de clases de una manera muy simple. Se puede crear una subclase de todas las clases que no estén definidas como finales. También se puede definir una superclase directa de una clase si esta clase no fue a su vez derivada de otra.
Las interfaces permiten extender la definición de una clase. Cuando una clase implementa una interface, todos los componentes de la interface aparecen como componentes de la clase. Se puede aceder a estos componentes usando una referencia a la clase o una referencia a la interface. Las interfaces permiten trabajar con diferentes clases de manera uniforme. La implementación real de los componentes de la interface tiene lugar en las clases. Por esto, las interfaces proporcionan un método de separar la definición de la implementación de los componentes.
Las interfaces pueden contener atributos, métodos y eventos pero también otras interfaces. Las clases que implementan una interface compuesta tienen también que implementar todos sus componentes.

IMPLEMENTACIÓN DE INTERFACES EN LAS CLASES

Las interfaces son extensiones a las definiciones de las clases y proporcionan un punto de contacto uniforme entre distintas clases. Al contrario que las clases, las interfaces no pueden ser inicializadas. En

su lugar, las clases implementan las interfaces implementando todos sus métodos. Se puede acceder a los métodos de la interface bien mediante referencias a clases o referencias a interfaces. Cada clase puede implementar interfaces de manera distinta a las demás. Debido a esto, las interfaces son la base del polimorfismo en ABAP Objects.

Procedimiento

Es necesario haber creado antes la interface y la clase en el constructor de clases. En la pestaña Interfaces del editor de clases se añade la interface a la clase especificando:

- Interface – Nombre que identifica a la interface. Al pulsar enter, el sistema chequea que la interface exista en la librería de clases.
- Sólo modelado – - Si se selecciona esta opción, el sistema no crea la interface en el pool de clases. No se podría entonces acceder al componente en tiempo de ejecución.

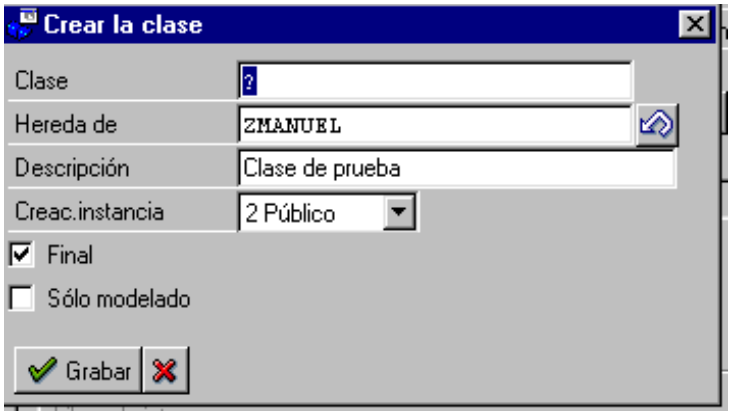
Las interfaces están en la parte declarativa de la clase en el pool de clases bajo la sentencia **`INTERFACES`**. Ahora se tienen que implementar los métodos de la interface de la manera que ya hemos visto. Los componentes de la interface, atribuos, métodos y eventos aparecen en la claseen la forma <nombre de la interface>~<nombre del componente>. Esto asegura que no existan conflictos con los nombres de los componentes de la clase.

CREACIÓN DE SUBCLASES

La herencia permite derivar clases de clases ya existentes. La nueva clase contiene un mayor rango de funciones específicas que su superclase. Esto se puede hacer añadiendo nuevos componentes a la subclase y redefiniendo métodos heredados de la superclase.

Procedimiento

Para crear una subclase directa de una clase desde el editor de clases se escoge la pestaña Atributos (datos básicos) y dentro de ella subclase (siempre que la clase no sea final). Aparece el siguiente cuadro de diálogo.



La clase se crea ahora como una clase normal.
La subclase hereda todos los componentes de la superclase excepto los métodos constructores. Las interfaces implementadas en la superclase también lo son en la subclase.

AMPLIACIONES EN SUBCLASES

Los cambios en las subclases son aditivos, esto es, no se puede borrar un componente de una clase si ha sido heredado de una superclase. Una clase se puede ampliar de varias maneras:

- Añadiendo nuevos componentes.
- Redefiniendo métodos heredados.

Sólo se pueden redefinir métodos dependientes de instancia. Los atributos, los métodos estáticos y otros componentes heredados no se pueden redefinir. Además, los métodos que se quieren redefinir no pueden haber sido definidos como finales en la superclase. Los métodos constructores no pueden ser redefinidos ya que son declarados finales implícitamente. Una redefinición de un método afecta únicamente a la implementación del método, los nombres y los tipos de los parámetros del método no pueden cambiarse.

La interface de parámetros del método redefinido debe permanecer igual que en el método original de la superclase.

Procedimiento

Se pueden definir nuevos componentes en las tres zonas de visibilidad (público, protegido o privado) de una subclase teniendo en cuenta que los nombres han de ser distintos entre los componentes heredados y creados en la subclase ya que si no existirán conflictos entre ellos.
Para redefinir un método heredado en una subclase:

- Se visualizan todos los métodos de la subclase. En principio no aparecen los métodos heredados, para que aparezcan tenemos que ir a Utilidades → Opciones y seleccionar Visualizar también componentes heredados.
- Si se siguen los pasos indicados en la ayuda de la versión 4.6B lo que se hace es modificar el método de la superclase. Para redefinir el método hay que realizar lo siguiente.
- Se selecciona el método elegido y se pulsa el botón de redefinir:

(segundo por la derecha de la subscreen). Se entra directamente a la redifinición del método. Si se quiere anular la redifinición y volver al método original el botón situado a la derecha lo permite.

Los métodos refefinidos aparecen en diferente color en el editor de clases. Si se redefine un método en una subclase, el método correspondiente en la superclase permanece sin cambios.
Para acceder a los componentes de la superclase (por ejemplo al método sin redefinir se puede usar la pseudoreferencia **`SUPER`**.

INTERFACES ANIDADAS

ABAP Objects soporta anidamiento de interfaces. Una interface compuesta tiene una o más interfaces como componentes. Cada una de ellas puede tener a su vez más interfaces como componentes, permitiendo un mayor anidamiento. Las interfaces que no contienen interfaces se conocen como interfaces simples.
Todos los componentes de las interfaces que se añaden a una clase están en el mismo nivel de anidamiento visto desde la clase. Cuando una clase usa una interface debe implementar todos los métodos de todos los componentes de la interface independientemente de su nivel de anidamiento. Se accede a sus componentes usando sus nombres originales, esto es, <nombre de la interface>~<componente>

ACTIVACIÓN DE CLASES E INTERFACES

Todos los componentes de una clase global que se quiera instanciar en un programa tienen que ser activados explícitamente. Todas las clases globales tienen una entrada en la tabla TADIR. El correspondiente objeto de transporte tiene el nombre R3TR CLAS <nombre de la clase> y contiene un rango de componentes, cada uno de los cuales es una unidad separada de cara al transporte. Hay que atender a una serie de normas:

- Los datos básicos y los componentes públicos de una clase no pueden ser activados por separado.
- Sólo los datos básicos y las secciones pública, protegida y privada de una clase afecta al estatus en el editor de clases. Si se activa el objeto entero y luego se cambia la implementación de un método, el estatus permanece activo.

El objeto para el transporte de una interface tiene el nombre R3TR INTF <nombre de la interface>. Contiene un solo objeto con el nombre INTF. Cuando se activa una clase que implementa una interface hay que asegurarse de que la interface haya sido previamente activada o si no la sección pública contendrá un error de sintaxis.
El estatus de una clase o una interface aparece siempre en el editor de clases y está determinado por:

- Relevancia en tiempo de ejecución – (Implementado / Sólo modelado)
- Estado en la base de datos – (Revisado / Grabado)
- Activación – (Inactivo / activo)

Pueden aparecer ocho posibles estatus.

PRUEBAS

Se puede usar el entorno de pruebas (test environment) para probar componentes en el constructor de clases y objetos de negocio en el Business Object Builder. El sistema genera dinamicamente un programa de test que simula la ejecución de varias sentencias ABAP. Sólo se pueden probar los componentes públicos de los objetos.
Antes de probar una clase hay que chequear la sintaxis y si hay algún error solucionarlo ya que no se puede usa el entorno de test con errores de sintaxis. El entorno de test se encuentra desde el editor de clases en Clase → Verif (F8). El sistema abre el entorno de test y se vualiza la clase en forma de árbol. En el entorno de test se pueden crear instancias, probar atributos, probar métodos, eventos e interfaces.