




# Programação orientada a objeto em Abap/4

Carlos Eduardo Candido de Oliveira  
Consultor Abap - Aspen BH  
[carlos.candido@procwork.com.br](mailto:carlos.candido@procwork.com.br)

<b><u>PRINCIPAL OBJETIVO DA ORIENTAÇÃO A OBJETO.....</u></b>	<b>4</b>
<b><u>TAD, CLASSES E OBJETOS.....</u></b>	<b>4</b>
<b><u>A LINGUAGEM ABAP É ORIENTADA A OBJETO? .....</u></b>	<b>4</b>
<b><u>VISIBILIDADE.....</u></b>	<b>4</b>
<b><u>HELP.SAP.COM .....</u></b>	<b>5</b>
<b><u>O QUE É ORIENTAÇÃO A OBJETO? ++</u></b>	<b>5</b>
<i><u>Usos de Orientação a Objeto.....</u></i>	<i>6</i>
<i><u>Leitura Posterior.....</u></i>	<i>7</i>
<i><u>O ambiente de tempo de execução.....</u></i>	<i>9</i>
<i><u>A extensão da linguagem orientada a objeto .....</u></i>	<i>9</i>
<b><u>DE GRUPOS DE FUNÇÕES A OBJETOS .....</u></b>	<b>9</b>
<i><u>Classes.....</u></i>	<i>13</i>
<i><u>Objetos como Instancias de uma Classe .....</u></i>	<i>15</i>
<i><u>Declarando Métodos.....</u></i>	<i>25</i>
<i><u>Implementando Métodos .....</u></i>	<i>26</i>
<i><u>Chamando Métodos.....</u></i>	<i>26</i>
<i><u>Métodos de Manipulação de Eventos.....</u></i>	<i>27</i>
<i><u>Constructors.....</u></i>	<i>28</i>
<b><u>MÉTODOS EM OBJETOS ABAP - EXEMPLO ++</u></b>	<b>28</b>
<i><u>Visão Geral.....</u></i>	<i>28</i>
<b><u>HERANÇA ++</u></b>	<b>39</b>
<i><u>Redefinindo Métodos.....</u></i>	<i>40</i>
<i><u>Métodos e Classes Abstratos e Finais.....</u></i>	<i>40</i>
<i><u>Referências a Subclasses e Polimorfismo .....</u></i>	<i>41</i>
<i><u>Espaço de Nomes para Componentes .....</u></i>	<i>41</i>
<i><u>Herança e Atributos Estáticos.....</u></i>	<i>41</i>
<i><u>Herança e Constructors.....</u></i>	<i>42</i>
<b><u>GRÁFICO GERAL DE HERANÇA ++</u></b>	<b>44</b>
<i><u>Herança: Visão Geral .....</u></i>	<i>44</i>
<i><u>Herança Simples .....</u></i>	<i>45</i>
<i><u>Herança e Variáveis Referenciais.....</u></i>	<i>46</i>
<b><u>INTERFACES ++</u></b>	<b>47</b>
<i><u>Definindo Interfaces.....</u></i>	<i>48</i>
<i><u>Implementando Interfaces .....</u></i>	<i>48</i>
<i><u>Referência de Interface .....</u></i>	<i>49</i>
<i><u>Endereçando Objetos Usando Referências de Interfaces .....</u></i>	<i>49</i>
<i><u>Atribuição Usando Referências de Interfaces .....</u></i>	<i>50</i>
<b><u>GRÁFICOS GERAIS ++</u></b>	<b>51</b>

<u>Interfaces</u> .....	51
<u>INTERFACES - EXEMPLO INTRODUTÓRIO</u> 	52
<u>ATIVANDO E MANIPULANDO EVENTOS</u> 	56
<u>Ativando Eventos</u> .....	56
<u>Declarando Eventos</u> .....	56
<u>Disparando Eventos</u> .....	57
<u>Manipulando Eventos</u> .....	57
<u>Declarando Métodos de Manipulação de Eventos</u> .....	57
<u>Registrando Métodos de Manipulação de Eventos</u> .....	57
<u>Timing de Manipulação de Eventos</u> .....	58
<u>GRÁFICO GERAL - EVENTOS</u> 	58
<u>EVENTOS: EXEMPLO INTRODUTÓRIO</u> 	61
<u>EVENTOS EM OBJETOS ABAP - EXEMPLO</u> 	62
<u>Visão Geral</u> .....	62
<u>POOL DE CLASSES</u> 	68
<u>Classes Globais e Interfaces</u> .....	68
<u>Estrutura de um Pool de classes</u> .....	68
<u>Diferenças De Outros Programas ABAP</u> .....	69
<u>Classes Locais em Pool de Classes</u> .....	70

## Principal objetivo da orientação a objeto.

Reaproveitamento de código.

## TAD, Classes e objetos

**TAD**(Tipo Abstrato de Dados) são estruturas que visam representar objetos do mundo real em uma forma computacional. São compostos de atributos e métodos.

Os atributos são os dados referentes ao TAD e devem ser manipulados apenas pelo métodos que são operações que alteram o ambiente e os atributos. Os métodos disponibilizam aos desenvolvedores as funcionalidades do TAD.

As **Classes** são TAD propriamente definidos e implementados por um desenvolvedor como um tipo não nativo da linguagem. Em uma analogia simples, as classes são molduras, formas de onde irão nascer várias representações dela.

Cada representação (instância) de uma classe recebe o nome de **objeto**. Cada objeto é alocado em diferentes posições da memória.

## A linguagem ABAP é orientada a objeto?

Sim. Uma linguagem para ser considerada como orientada a objeto deve possibilitar os seguintes recursos:

- Classes e objetos
- Herança
- Polimorfismo

O ABAP disponibiliza tais recursos. A linguagem, porém, não é totalmente orientada a objetos por combinar elementos estruturados e não estruturados.

Os conceitos de OOP apresentados na linguagem ABAP são os mesmos em sua maioria apresentados pela linguagem JAVA como classes finais, abstratas, coletor de lixo e herança simples.

## Visibilidade

Atributos e métodos podem ser públicos, privados ou protegidos.

### Público

Podem ser acessados e modificados de dentro e de fora dos domínios da classe. O acesso direto aos atributos de fora da classe deve ser evitado.

PUBLIC SECTION.

DATA: Counter type i.

### Privado

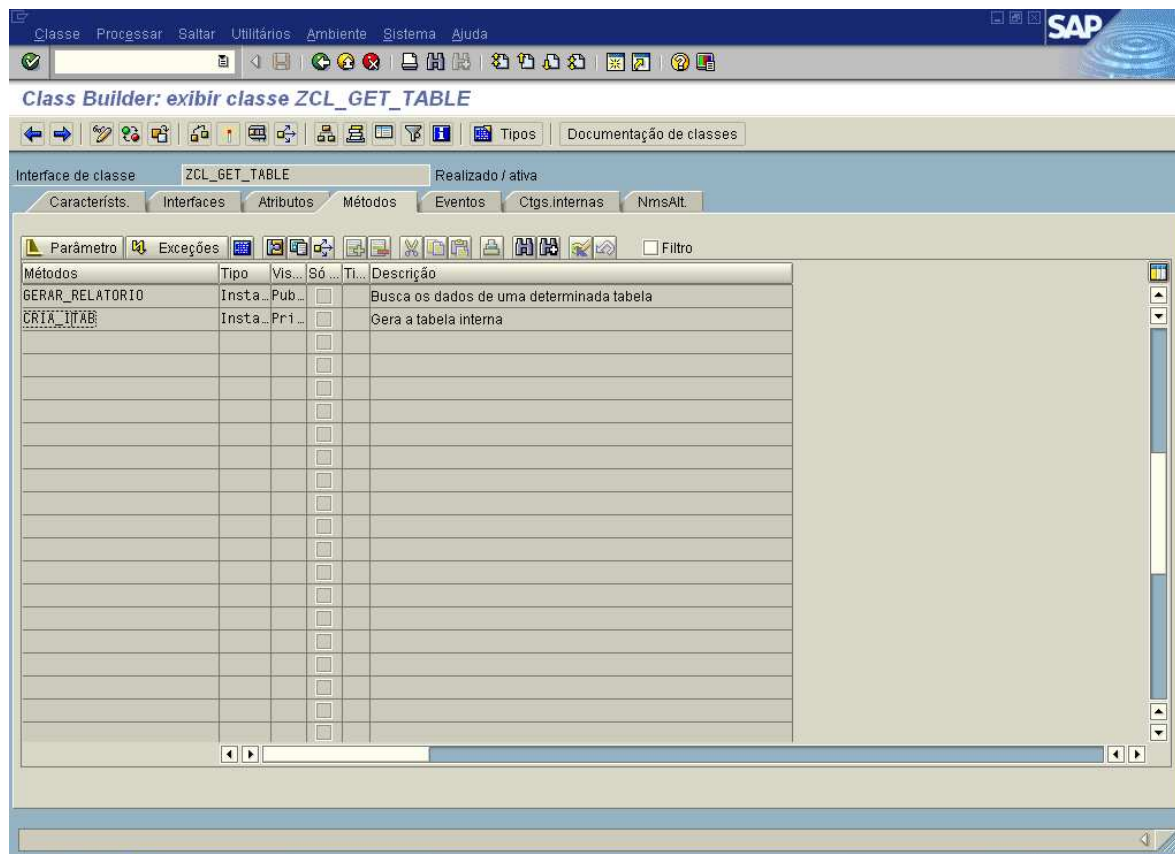
Atributos e métodos definidos como privados não podem ser acessados fora dos domínios da classe.

PRIVATE SECTION.

DATA: name(25) TYPE c,

planetype LIKE saplane-planetyp,

No R3 pode-se definir classes através da transação SE24 ou no próprio corpo do programa Abap.



Nas pastas públicas, recomendo a leitura da apostila OOP em ABAP e Class Builder. A seguir, temos o material traduzido sobre Abap Objects extraído do help.sap.com.

**Help.sap.com**

## O que é Orientação a Objeto?

Orientação a objeto, ou para ser mais preciso, programação orientada a objeto, é um método de resolução de problemas no qual a solução do software reflete objetos do mundo real.

Uma introdução abrangente à orientação a objetos como um todo iria muito além dos limites da introdução aos objetos ABAP. Esta documentação introduz uma seleção de termos que são usados universalmente na orientação a objetos e também ocorre em objetos ABAP. Em sessões

subseqüentes, continua-se a discutir em mais detalhes como esses termos são usados em objetos ABAP. O fim desta seção contém uma lista de **leitura posterior**, com uma seleção de títulos sobre orientação a objeto.

### Objetos

Um objeto é seção de código que contém dados e fornece serviços. Os dados formam os atributos do objeto. Os serviços são conhecidos como métodos (também conhecido como operações ou funções). Tipicamente, métodos operam em dados privados (os atributos, ou estado do objeto), que é apenas visível para os métodos do objeto. Logo os atributos de um objeto não podem ser modificados diretamente pelo usuário, mas apenas pelos métodos do objeto. Isso garante a consistência interna do objeto.

### Classes

Classes descrevem objetos. De um ponto de vista técnico, objetos são instancias em tempo de execução de uma classe. Em teoria, você pode criar qualquer número de objetos baseados em uma única classe. Cada instancia (objeto) de uma classe tem uma identidade única e seu próprio conjunto de valores para seus atributos.

### Referências a Objetos

Em um programa, você identifica e endereça objetos usando referências únicas a objetos. Referências a objetos permitem que acesse os atributos e métodos de um objeto.

Em programação orientada a objeto, objetos geralmente têm as seguintes propriedades:

### Encapsulamento

Objetos restringem a visibilidade de seus recursos (atributos e métodos) aos outros usuários. Todo objeto tem uma **interface**, que determina como os outros objetos podem interagir com ele. A implementação do objeto é encapsulada, isso é, invisível for a do próprio objeto.

### Polimorfismo

Métodos idênticos (mesmo nome) se comportam diferentemente em diferentes classes. Orientação orientada a objeto contém construções chamadas interfaces. Elas permitem que enderece métodos com mesmo nome em diferentes objetos. Apesar de a forma de endereçamento é sempre a mesma, a implementação do método é específica a uma particular classe.

### Herança

Você pode usar uma classe existente para derivar uma classe nova. Classes derivadas herdam os dados e métodos da superclasse. No entanto, eles podem substituir métodos existentes, e também adicionar novos.

### Usos de Orientação a Objeto

Abaixo estão algumas vantagens da programação orientada a objeto:

- Sistemas de software complexos se tornam mais fáceis de serem compreendidos, já que a estrutura orientada a objeto fornece uma representação muito mais próxima da realidade do que as outras técnicas de programação.

- Em um sistema orientado a objeto bem desenvolvido, é possível implementar mudanças em nível de classe, sem ter que realizar alterações em outros pontos do sistema. Isto reduz a quantidade total de manutenção requerida.
- Através do polimorfismo e herança, a programação orientada a objeto permite a reutilização de componentes individuais.
- Em um sistema orientado a objeto, a quantidade de trabalho de manutenção e revisão envolvido é reduzida, já que muitos problemas podem ser detectados e corrigidos em fase de projeto.

Para atingir estes objetivos requer:

- Linguagens de programação orientada a objetos
- Técnicas de programação orientadas a objeto não necessariamente dependem em linguagens de programação orientada a objeto. No entanto, a eficiência da programação orientada a objeto depende diretamente de como as técnicas de programação orientada a objetos são implementadas no sistema kernel.

- Ferramentas de orientação a objeto

Ferramentas de orientação a objeto permite que se crie programas orientados a objetos em linguagem orientada a objetos. Eles permitem que se modele e guarde objetos e relações entre eles.

- Modelagem orientada a objeto

A modelagem orientada a objeto de um sistema de software é o mais importante, mais demorado, e o requerimento mais difícil para alcançar acima dos objetivos. Design orientado a objeto envolve mais do que apenas programação orientada a objeto, e fornece vantagens lógicas que são independentes da verdadeira implementação.

Esta seção do guia do usuário ABAP fornece uma visão geral da extensão orientada a objeto da linguagem ABAP. Nós temos usado apenas exemplos simples para demonstrar como se utilizam as novas ferramentas. No entanto, estas pretendem ser um modelo para design orientado a objeto. Mais informação detalhada sobre cada dos comandos dos objetos ABAP é contida na documentação de palavras chaves no editor ABAP. Para uma introdução compreensiva ao desenvolvimento de software orientado a objeto, você deve ler um ou mais títulos listados abaixo.

### ***Leitura Posterior***

Há muitos livros sobre orientação a objeto, linguagem de programação orientado a objeto, análise orientada a objeto e design, gerenciamento de projeto para projetos OO, padrões e frameworks, e muitos outros. Esta é uma pequena seleção de bons livros cobrindo os tópicos principais:

- **Scott Ambler, The Object Primer, SIGS Books & Multimedia (1996), ISBN: 1884842178**

Uma introdução muito boa à orientação a objetos para programadores. Ele fornece explicações compreensivas de todas as essências dos conceitos OO, e contém um modo de aprendizagem muito rápido e eficiente. É fácil e prático de se ler, mas totalmente baseado em teoria.

- **Grady Booch, Object Solutions: Managing the Object-Oriented Project, Addison-Wesley Pub Co (1995), ISBN: 0805305947**

Um bom livro sobre todos os aspectos não técnicos de OO que são igualmente importantes para uma efetiva programação orientada a objeto. Fácil de ler e cheio de dicas práticas.

- **Martin Fowler, UML Distilled: Applying the Standard Object Modeling Language, Addison-Wesley Pub Co (1997), ISBN: 0201325632**

Um excelente livro sobre UML (Modelagem de Linguagem Unificada – a nova linguagem padronizada OO e notações para modelagem). Assume conhecimento anterior e experiência em orientação a objeto.

- **Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Pub Co (1998), ISBN: 0201634988**

Fornece um padrão, mostrando como problemas de design recorrentes podem ser resolvidos utilizando objetos. Este é o primeiro grande livro padrão, contendo muitos exemplos de bom design OO.

- **James Rumbaugh, OMT Insights: Perspectives on Modeling from the Journal of Object-Oriented Programming, Prentice Hall (1996), ISBN: 0138469652**

Uma coleção de artigos endereçando muitas perguntas e problemas de análise e design OO, implementação, gerenciamento de dependência, e muitos outros. Altamente recomendado.

### Notas

Se você é novo a orientação a objetos, você deveria ler Scott Ambler's 'The Object Primer' e então adquirir alguma experiência prática você próprio. Você deve definitivamente utilizar as técnicas CRC descritas por Ambler e Fowler para análise e design orientados a objeto. Após isso, você deve aprender UML, já que este é a análise e design OO universal. Finalmente, você deve ler pelo menos um dos livros sobre padrões.

No início de um grande projeto OO, as questões imediatamente emergem sobre como a seqüência que as coisas devem ser feitas, quais fases devem terminar e a qual hora, como dividir e organizar o trabalho de desenvolvimento, como minimizar riscos, como montar uma boa equipe, e assim continua. Muitas das melhores práticas em gerenciamento de projetos tiveram que ser redefinidas para o mundo orientado a objeto, e as oportunidades que isto produz são significantes. Para mais informações sobre como utilizar, veja o livro de Grady's Brooch 'Object Solutions' , ou o capítulo intitulado 'An outline development process' do livro de Martin Fowler.

Há, claro, muitos outros livros bons sobre orientação a objeto. Os acima listados não clamam estarem totalmente completos, ou necessariamente os melhores livros disponíveis.



### *O que são Objetos ABAP?*

Objetos ABAP são um novo conceito do R/3 Release 4.0. O termo tem dois conceitos. No primeiro, os objetos compõem todo o ambiente de tempo de execução ABAP. No segundo, ele representa a extensão orientada a objeto da linguagem ABAP.

#### ***O ambiente de tempo de execução***

O novo nome ABAP Objects para todo o ambiente de tempo de execução é uma indicação do modo que o SAP R3 tem, por algum tempo, movendo em direção a orientação a objeto, e de seu compromisso em perseguir esta linha. O ABAP Workbench permite que você crie objetos R/3 Repository, como programas, objetos de autorização, objetos de travamento, objetos de customizing, e muitos outros. Usando módulos de funções, você pode encapsular funções em programas separados com uma interface definida. O Business Object Repository (BOR) permite que você crie SAP Business Objects para uso interno e externo (DCOM/CORBA). Até agora, as técnicas de orientação a objeto têm sido usadas exclusivamente em projeto de sistemas, e ainda não foram suportadas pela linguagem ABAP.

#### ***A extensão da linguagem orientada a objeto***

Objetos ABAP são um conjunto completo de comandos que foram introduzidos dentro da linguagem ABAP. Esta extensão orientada a objeto constrói sobre a linguagem existente, sendo totalmente compatível com ela. Você pode utilizar objetos ABAP em já programas existentes, e pode também usar o ABAP “convencional” em novos objetos de programas ABAP.

Objetos ABAP suporta programação orientada a objeto. Orientação a objeto (OO), também conhecida como paradigma orientado a objeto, é um modelo de programação que une dados e funções em objetos. O resto da linguagem ABAP é primeiramente intencionado para programação estruturada, onde os dados são guardados em tabelas de banco de dados e programas orientados por funções acessam e trabalham com eles.

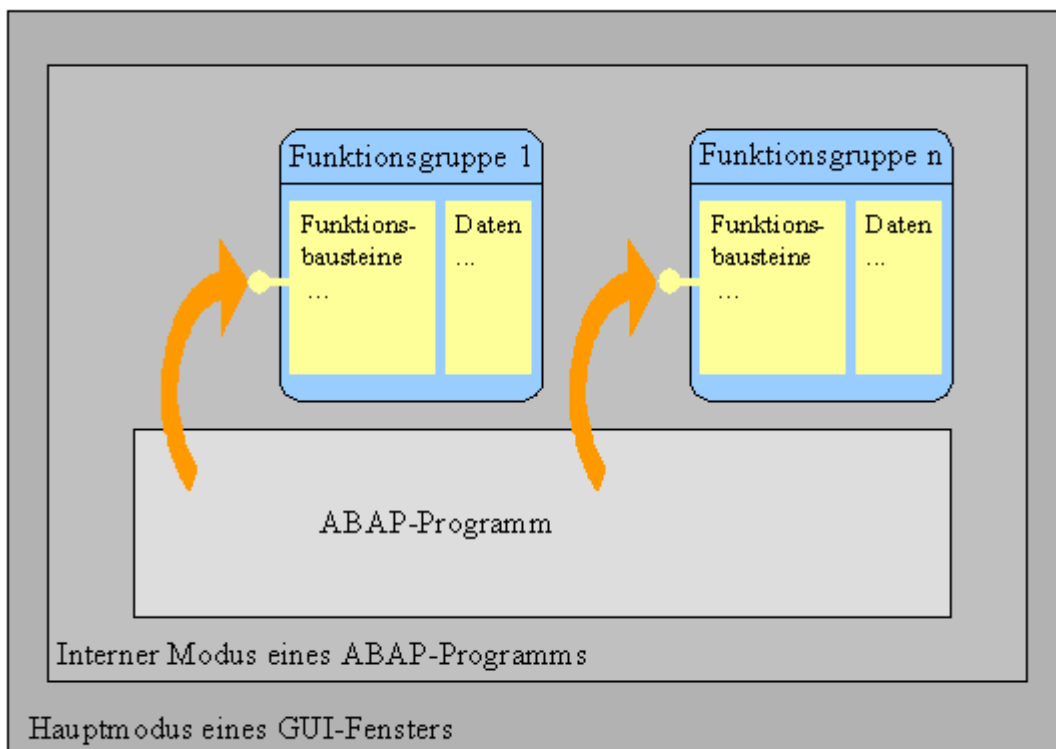
A implementação do modelo orientado a objeto do ABAP é baseado em modelos de Java e C++. É compatível com objetos externos como DCOM e CORBA. A implementação de elementos orientados a objeto no kernel da linguagem ABAP tem aumentado consideravelmente o tempo de resposta quando se trabalha com objetos ABAP. SAP Business Objects e objetos GUI – vão também lucrar de serem incorporados nos objetos ABAP.

### *De Grupos de Funções a Objetos*

No centro de qualquer modelo orientado a objeto estão os objetos, que contém atributos (dados) e métodos (funções). Objetos deveriam permitir programadores a mapear um problema real e soluções propostas do software em uma base um-por-um. Objetos típicos no ambiente de negócios são, por exemplo, ‘consumidor’, ‘ordem’, ou ‘pedido’. Do release 3.1 em diante, o Business Object Repository (BOR) contém exemplos de tais objetos. O objeto modelo dos objetos ABAP, a extensão orientada a objeto do ABAP, são compatíveis com o objeto modelo do BOR.

Antes do R/3 Release 4.0, o que existia mais próximo de objetos no ABAP eram módulos de funções e grupos de funções. Suponha que tenhamos um grupo de funções para processar ordens. Os atributos de uma ordem correspondem aos dados globais do grupo de função, enquanto os módulos de funções individuais representam ações que manipulam aqueles dados (métodos). Isto significa que a verdadeira ordem dos dados é encapsulada no grupo de funções. Deste modo, os módulos de funções podem garantir que os dados são consistentes.

Quando você executa um programa ABAP, o sistema inicia uma nova sessão interna. A sessão interna tem uma área de memória que contém o programa ABAP e os dados associados. Quando você chama um módulo de função, uma **instancia** de um grupo de funções mais seus dados, é carregado na memória da sessão interna. Um programa ABAP pode carregar várias instancias através da chamada de módulos de funções de **diferentes** grupos de funções.

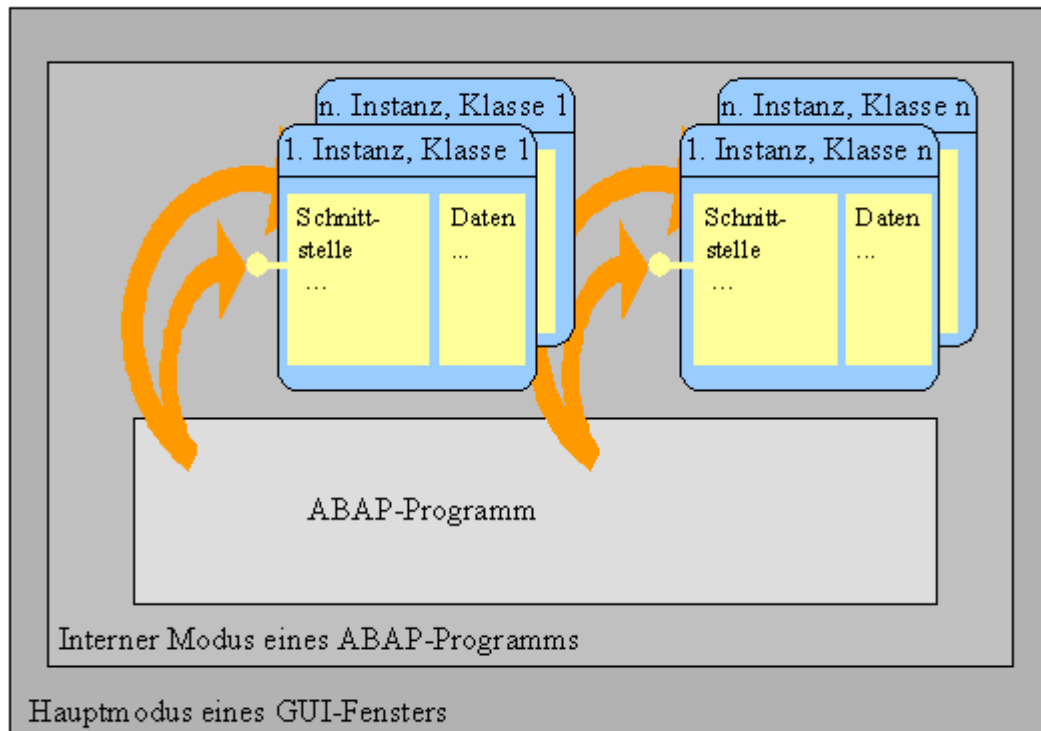


A instancia de um grupo de funções na área de memória de uma sessão interna quase representa um objeto no senso de orientação a objeto. Quando você chama um modulo de função, o programa chamador usa a instancia de um grupo de funções, baseada na descrição usada no Function Builder. O programa não pode acessar os dados no grupo de funções diretamente, mas apenas através do modulo de função. Os módulos de funções e seus parâmetros são a **interface** entre o grupo de funções e o usuário.

A diferença principal entre a real orientação a objeto e grupo de funções é que apesar de um programa poder trabalhar com instancias de diversos grupos de funções simultaneamente, ele não pode trabalhar com diversas instancias do mesmo grupo de funções. Suponha que um programa necessite vários contadores independentes, ou processar várias ordens ao mesmo tempo. Neste caso, você teria que adaptar o grupo de funções a incluir administração de instancia, por exemplo, usando números para diferenciá-las.

Na prática, isto é muito estranho. Conseqüentemente, os dados são usualmente guardados no programa chamador, e os módulos de funções são chamados para trabalhar com ele (programação estruturada). Um problema é, por exemplo, que todos os usuários do módulo de função devem utilizar os mesmos dados assim como o grupo de funções. Modificando a estrutura interna dos dados de um grupo de funções afeta vários usuários, e é freqüentemente difícil de prever implicações. O único modo de evitar é depender muito de interfaces e de uma técnica que garante que as estruturas internas das instancias vão permanecer ocultas, permitindo que as modifique mais tarde sem causar problemas.

Este requisito é atingido por orientação a objeto. Objetos ABAP permitem que você defina dados e funções em classes ao invés de grupo de funções. Usando classes, um programa ABAP pode trabalhar com qualquer número de instancias (objetos) baseados na mesma classe.



Ao invés de carregar uma única instancia de um grupo de funções dentro de uma memória implícita quando um modulo é chamado, o programa ABAP pode agora gerar as instancias de classes explicitamente usando o novo comando ABAP **CREATE OBJECT**. As instancias individuais representam objetos únicos. Você endereça estes usando referências a objetos. As referências aos objetos permitem que o programa ABAP acesse as interfaces das instancias.

## Exemplo

O exemplo a seguir demonstra o aspecto orientado a objeto de grupo de funções no simples caso de um contador.



Suponha que tenhamos um grupo de função chamado COUNTER:

```
FUNCTION-POOL COUNTER.
```

```
DATA COUNT TYPE I.
```

```
FUNCTION SET_COUNTER.
```

```
* Local Interface IMPORTING VALUE(SET_VALUE)
```

```
  COUNT = SET_VALUE.
```

```
ENDFUNCTION.
```

```
FUNCTION INCREMENT_COUNTER.
  ADD 1 TO COUNT.
ENDFUNCTION.
```

```
FUNCTION GET_COUNTER.
* Local Interface: EXPORTING VALUE(GET_VALUE)
  GET_VALUE = COUNT.
ENDFUNCTION.
```

O grupo de funções tem um campo inteiro global COUNT, e três módulos de funções, SET\_COUNTER, INCREMENT\_COUNTER, e GET\_COUNTER, que trabalham com o campo. Dois dos módulos de funções têm parâmetros de saída e de entrada. Estes formam a interface de dados do grupo de funções.

Qualquer programa ABAP pode então trabalhar com este grupo de funções. Exemplo:

```
DATA NUMBER TYPE I VALUE 5.

CALL FUNCTION 'SET_COUNTER' EXPORTING SET_VALUE = NUMBER.

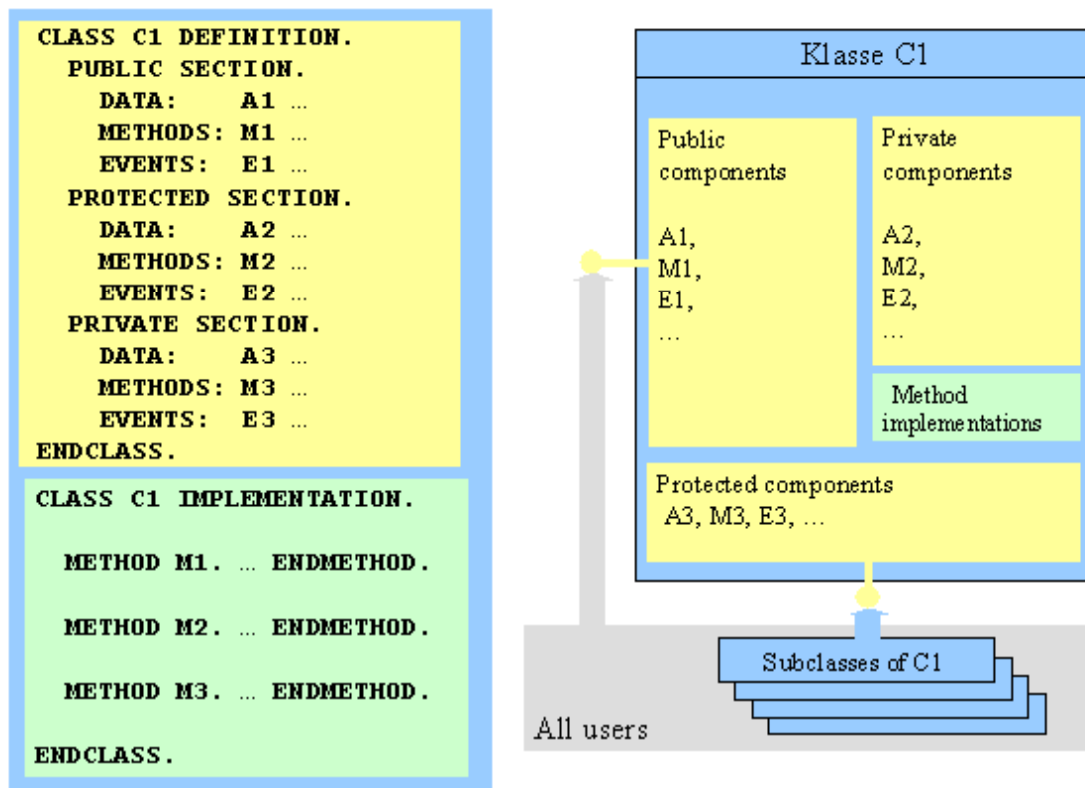
DO 3 TIMES.
  CALL FUNCTION 'INCREMENT_COUNTER'.
ENDDO.

CALL FUNCTION 'GET_COUNTER' IMPORTING GET_VALUE = NUMBER.
```

Após esta seção do programa tiver sido processada, a variável do programa NUMBER terá valor 8. O programa por ele próprio não pode acessar o campo COUNT no grupo de funções. Operações neste campo estão completamente encapsuladas no módulo de funções. O programa pode apenas comunicar com os grupos de funções chamando os seus módulos de função.

**Gráfico Geral** 

## Classes



A figura a esquerda demonstra as partes de declaração e implementação de uma classe local C1. A figura a direita ilustra a estrutura da classe com os componentes em suas respectivas áreas de visibilidade, e a implementação de métodos.

Os componentes públicos da classe formam a **interface entre a classe e seus usuários**. Os componentes protegidos são uma interface para que as subclasses de C1 acessem atributos da superclasse. Os componentes privados não estão visíveis externamente, e estão completamente encapsulados dentro da classe. Os métodos ,na parte de implementação, tem acesso irrestrito a todos os componentes da classe.

## Classes – Exemplo Introdutório

O simples exemplo a seguir utiliza Objetos ABAP para programar um contador. Para comparação, veja também o exemplo em De Grupo de Funções a Objetos:



```

CLASS C_COUNTER DEFINITION.

    PUBLIC SECTION.
        METHODS: SET_COUNTER IMPORTING VALUE(SET_VALUE) TYPE I,
                  INCREMENT_COUNTER,
                  GET_COUNTER EXPORTING VALUE(GET_VALUE) TYPE I.

    PRIVATE SECTION.
        DATA COUNT TYPE I.

ENDCLASS.

CLASS C_COUNTER IMPLEMENTATION.

    METHOD SET_COUNTER.
        COUNT = SET_VALUE.
    ENDMETHOD.

    METHOD INCREMENT_COUNTER.
        ADD 1 TO COUNT.
    ENDMETHOD.

    METHOD GET_COUNTER.
        GET_VALUE = COUNT.
    ENDMETHOD.

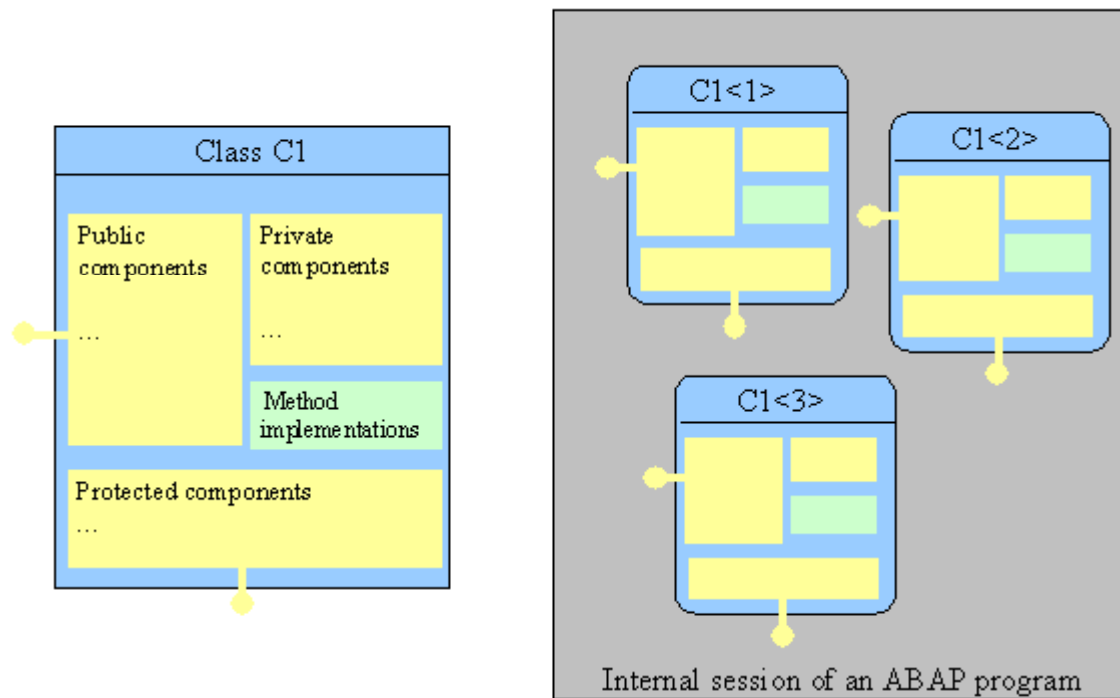
ENDCLASS.

```

A classe C\_COUNTER contém três métodos públicos - SET\_COUNTER, INCREMENT\_COUNTER, e GET\_COUNTER. Cada um destes trabalham com o campo inteiro privado COUNT. Dois dos métodos tem parâmetros de entrada e saída. Estes formam a interface de dados da classe. O campo COUNT não é visível externamente.

## Gráfico Geral

## Objetos como Instancias de uma Classe



Class → Instances of the class

A ilustração acima mostra uma classe C1 a esquerda, com suas instancias representadas na sessão interna de um programa ABAP a direita. Para distinguí-los de classes, instancias são desenhadas com cantos redondos. Os nomes das instancias acima usam a mesma notação como é usada para variáveis de referência no Debugger.

## Objetos – Exemplo Introdutório



O exemplo a seguir demonstra como criar e utilizar uma instancia da classe C\_COUNTER que nós criamos no capítulo anterior:

### Exemplo:

```
DATA CREF1 TYPE REF TO C_COUNTER.
```

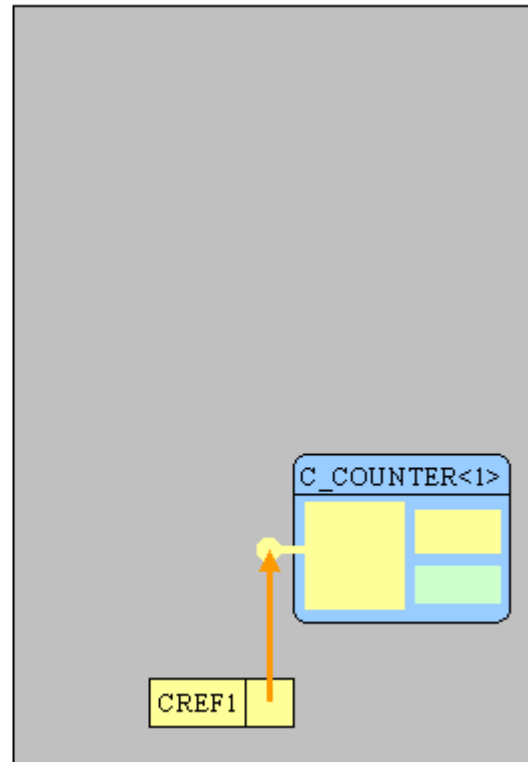
CREF1 →

O comando DATA cria uma variável CREF1 de referência a classe com tipo C\_COUNTER. Esta variável pode conter referências a todas as instâncias da classe C\_COUNTER. A classe C\_COUNTER deve ser conhecida ao programa quando o comando DATA ocorre. Você pode tanto declará-la localmente antes do comando data no mesmo programa, ou globalmente usando o Class Builder(SE24). O conteúdo de CREF1 são iniciais. A referência não aponta para uma instância (Em uma analogia, seria um ponteiro *dangling* que não possui memória alocada) .



```
DATA CREF1 TYPE REF TO C_COUNTER.
```

```
CREATE OBJECT CREF1.
```



O comando `CREATE OBJECT` cria um objeto (instancia) da classe `C_COUNTER`. A referência na variável de referência `CREF1` indica este objeto.

A instancia da classe `C_COUNTER` é chamada `C_COUNTER<1>`, e ela guarda o conteúdo da variável de objeto `REF_COUNTER_1`, mostradas no debugger após o comando `CREATE OBJECT` tiver sido executado. Este nome é apenas usado para administração de programas internos – não ocorre dentro do próprio programa ABAP.

```
DATA CREF1 TYPE REF TO C_COUNTER.

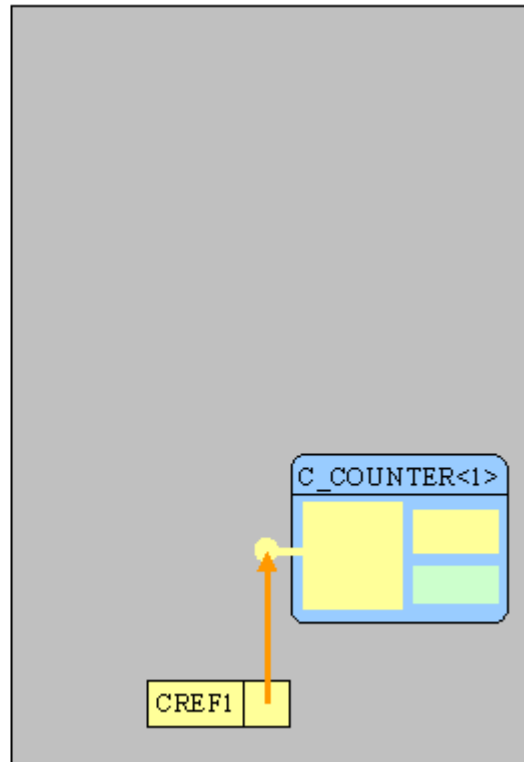
DATA NUMBER TYPE I VALUE 5.

CREATE OBJECT CREF1.

CALL METHOD CREF1->SET_COUNTER
  EXPORTING SET_VALUE = NUMBER.

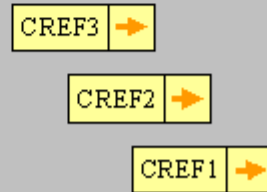
DO 3 TIMES.
  CALL METHOD CREF1->INCREMENT_COUNTER.
ENDDO.

CALL METHOD CREF1->GET_COUNTER
  IMPORTING GET_VALUE = NUMBER.
```



O programa ABAP pode acessar os componentes públicos do objeto usando a variável de referência CREF1, isso é neste caso, pode chamar os métodos públicos da classe C\_COUNTER. Após o programa à esquerda tiver sido executado, as variáveis do programa NUMBER e o atributo do objeto privado COUNT ambos terão valor 8.

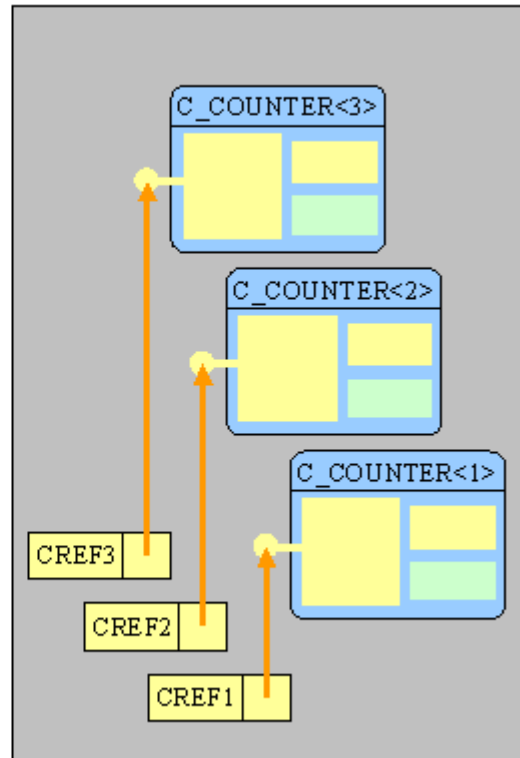
```
DATA: CREF1 TYPE REF TO C_COUNTER,  
      CREF2 TYPE REF TO C_COUNTER,  
      CREF3 LIKE CREF1.
```



O programa acima declara três diferentes variáveis de referência a classes para a classe C\_COUNTER. Todas as referências são iniciais.

```
DATA: CREF1 TYPE REF TO C_COUNTER,  
      CREF2 TYPE REF TO C_COUNTER,  
      CREF3 LIKE CREF1.
```

```
CREATE OBJECT: CREF1, CREF2, CREF3.
```



O sistema cria três objetos da classe de três variáveis de referências de classes. As referências nestas três variáveis de referência apontam para um destes objetos.

No gerenciamento interno do programa, as instancias individuais são chamadas `C_COUNTER<1>`, `C_COUNTER<2>`, e `C_COUNTER<3>`. Elas são nomeadas na ordem em que são criadas.

```

DATA: CREF1 TYPE REF TO C_COUNTER,
      CREF2 TYPE REF TO C_COUNTER,
      CREF3 LIKE CREF1.

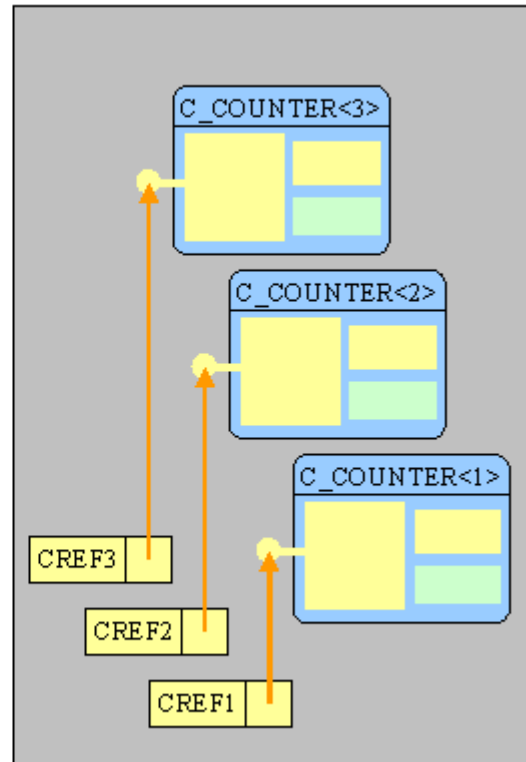
DATA: NUMBER1 TYPE I VALUE 5,
      NUMBER2 TYPE I VALUE 0,
      NUMBER3 TYPE I VALUE 2.

CREATE OBJECT: CREF1, CREF2, CREF3.

CALL METHOD: CREF1->SET_COUNTER
  EXPORTING SET_VALUE = NUMBER1,
  CREF2->SET_COUNTER
  EXPORTING SET_VALUE = NUMBER2,
  ...
...
CALL METHOD CREF2->INCREMENT_COUNTER.
...
CALL METHOD CREF1->INCREMENT_COUNTER.
...

CALL METHOD: CREF1->GET_COUNTER
  IMPORTING GET_VALUE = NUMBER1,
  CREF2->GET_COUNTER
  IMPORTING GET_VALUE = NUMBER2,
  ...

```

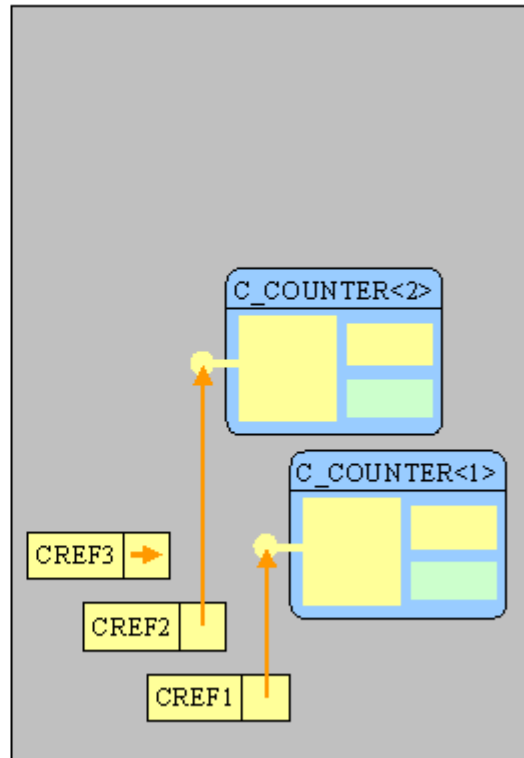


O programa ABAP pode usar as variáveis de referência para acessar objetos individuais, isto é neste caso, para chamar os métodos públicos da classe C\_COUNTER.

Cada objeto tem seu próprio estado, já que o atributo da instancia privada COUNT tem um valor separado para cada objeto. O programa a esquerda administra vários contadores independentes.

```
DATA: CREF1 TYPE REF TO C_COUNTER,  
      CREF2 TYPE REF TO C_COUNTER,  
      CREF3 LIKE CREF1.
```

```
CREATE OBJECT: CREF1, CREF2.
```

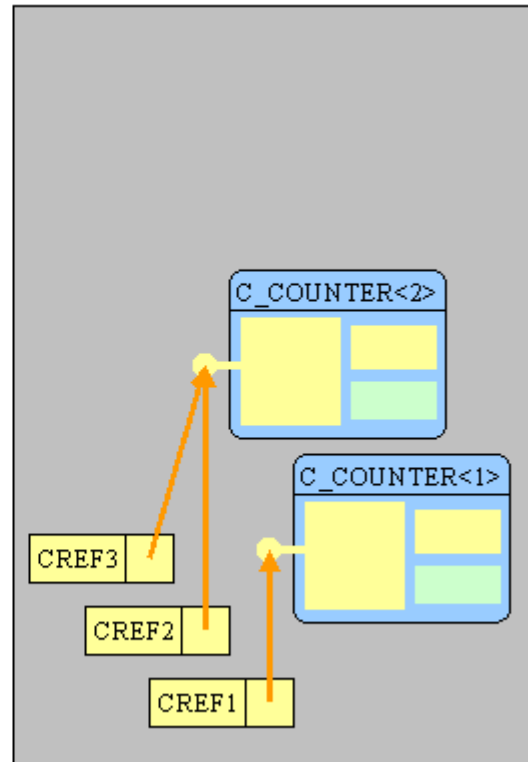


Aqui, três variáveis de referência a classes são declaradas para a classe C\_COUNTER, e dois objetos são criados para a classe. As referências nas variáveis CREF1 e CREF2 apontam para um destes objetos. A referência CREF3 é inicial.

```
DATA: CREF1 TYPE REF TO C_COUNTER,  
      CREF2 TYPE REF TO C_COUNTER,  
      CREF3 LIKE CREF1.
```

```
CREATE OBJECT: CREF1, CREF2.
```

```
MOVE CREF2 TO CREF3.
```



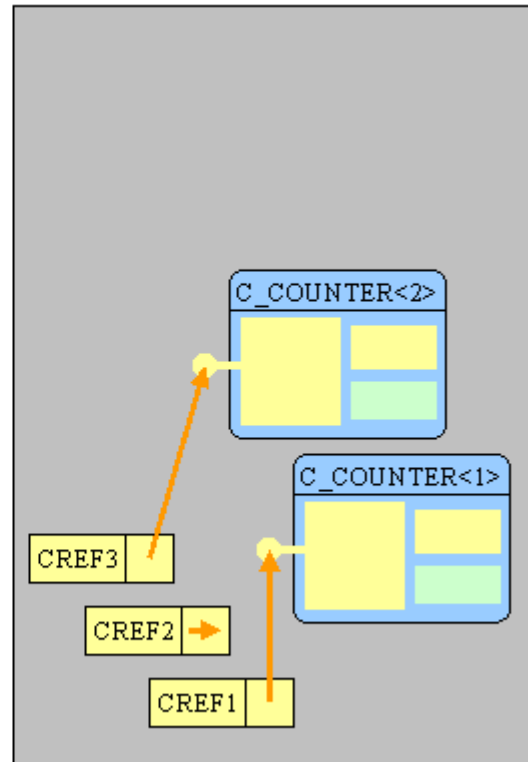
Após o comando MOVE, CREF3 contém as mesmas referências de CREF2, e ambas indicam o objeto C\_COUNTER<2>. Um usuário pode então usar ambas para indicar o objeto C\_COUNTER<2>.

```
DATA: CREF1 TYPE REF TO C_COUNTER,  
      CREF2 TYPE REF TO C_COUNTER,  
      CREF3 LIKE CREF1.
```

```
CREATE OBJECT: CREF1, CREF2.
```

```
MOVE CREF2 TO CREF3.
```

```
CLEAR CREF2.
```



O comando **CLEAR** desfaz a referência em **CREF2** para seu valor inicial. A variável de referência **CREF2** então contém o mesmo valor que continha após sua declaração, ela não mais indica um objeto.



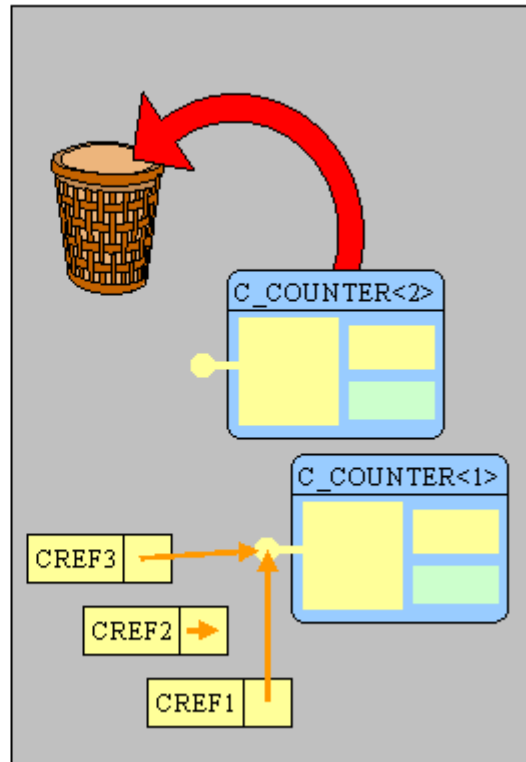
```
DATA: CREF1 TYPE REF TO C_COUNTER,  
      CREF2 TYPE REF TO C_COUNTER,  
      CREF3 LIKE CREF1.
```

```
CREATE OBJECT: CREF1, CREF2.
```

```
MOVE CREF2 TO CREF3.
```

```
CLEAR CREF2.
```

```
CREF3 = CREF1.
```



O efeito do comando de igualdade é copiar a referência de CREF1 para CREF2. Como resultado, a referência em CREF3 também indica o objeto C\_COUNTER<1>. Mais nenhuma referência indica o objeto C\_COUNTER<2>, e este é automaticamente excluído pelo coletor de lixo. O nome interno C\_COUNTER<2> está agora livre novamente.

## Declarando e Chamando Métodos

Esta seção explica como trabalhar com métodos em objetos ABAP. Para detalhes precisos de comandos relevantes ABAP, refira-se a palavra chave na documentação do editor. O exemplo mostra como declarar, implementar e chamar métodos.

### Declarando Métodos

Você pode declarar métodos na parte de declaração de uma classe ou em uma interface. Para declarar métodos de instancia (pertencentes ao objeto), use o seguinte comando:

```
METHODS <meth> IMPORTING.. [VALUE(<i>[<e>]) TYPE type [OPTIONAL]..  
EXPORTING.. [VALUE(<e>[<e>]) TYPE type [OPTIONAL]..  
CHANGING.. [VALUE(<c>[<e>]) TYPE type [OPTIONAL]..  
RETURNING VALUE(<r>)  
EXCEPTIONS.. <e>..
```

E as adições apropriadas.

Para declarar métodos estáticos(pertencem à classe), use o seguinte comando:

CLASS-METHODS <meth>...

Ambos têm a mesma sintaxe.

Quando você declara um método, você também define sua interface de parâmetros usando as adições IMPORTING, EXPORTING, CHANGING, e RETURNING. As adições definem parâmetros de entrada/saída, e o código de retorno. Elas também definem os atributos dos parâmetros da interface, principalmente quando um parâmetro é passado por referência ou valor (VALUE), seu tipo(TYPE), e se é opcional (OPTIONAL, DEFAULT). Ao contrário de módulos de funções, o modo default de passar um parâmetro em um método é por referência. Para passar um parâmetro por valor, você deve explicitamente especificar a adição VALUE. O valor de retorno (RETURNING) deve sempre ser passado explicitamente como um valor. Isto é ótimo para métodos que retornam um único valor de saída. Se você usá-lo, você não pode usar EXPORTING ou CHANGING.

Assim como em módulos de funções, você pode usar parâmetros de exceções (EXCEPTIONS) para permitir que o usuário reaja a situações de erro quando o método é executado.

## **Implementando Métodos**

Você deve implementar todos os métodos dentro de uma classe na parte de implementação.

METHOD <meth>.

...

ENDMETHOD.

Quando você implementa o método, não precisa especificar quaisquer parâmetros de interface, já que são definidos dentro da declaração do método. Os parâmetros de interface de um método comportam-se como variáveis locais dentro do método de implementação. Pode-se definir variáveis locais adicionais dentro de um método usando o comando DATA.

Assim como módulos de funções, você pode usar o comando RAISE <exception> e MESSAGE RAISING para lidar com situações de erro.

Quando você implementa um método estático, lembre-se que isto pode apenas trabalhar com atributos estáticos da sua classe. Métodos de instancias podem trabalhar com ambos atributos estáticos ou de instancia.

## **Chamando Métodos**

Para chamar um método, use o seguinte comando:

```
CALL METHOD <meth> EXPORTING... <i> =.<f i>...
      IMPORTING... <e> =.<g i>...
      CHANGING ... <c> =.<f i>...
      RECEIVING      r = h
      EXCEPTIONS... <e> = rc i...
```

O modo como se acessa o método <meth> depende de onde você o está chamando. Dentro da parte de implementação da classe, você pode chamar os métodos da mesma classe diretamente usando o nome <meth> deles.

CALL METHOD <meth>...

Fora da classe, a visibilidade do método depende da visibilidade atribuída a ele(public). Métodos de instancia visíveis podem ser chamados de fora da classe usando:

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

Onde <ref> é uma variável de referência cujo valor indica uma instancia da classe. Métodos de classe visíveis podem ser chamados de fora da classe usando:

CALL METHOD <class>=><meth>...

onde <class> é o nome da classe relevante.

Quando você chama um método, você deve passar todos os parâmetros de entrada obrigatórios usando EXPORTING ou CHANGING no comando CALL METHOD. Você pode (se quiser) importar os parâmetros de saídas para dentro de seu programa usando IMPORTING ou RECEIVING. Igualmente, você pode (se quiser) manipular quaisquer exceções disparadas pelos métodos usando a adição EXCEPTIONS. No entanto, isto é altamente recomendado.

Você passa e recebe valores em métodos da mesma maneira dos módulos de funções, incluindo a sintaxe após a adição correspondente:

... <Parâmetro Formal > = <Parâmetro Atual>

Os parâmetros de interface (Parâmetros Formais) estão sempre do lado esquerdo do sinal de igualdade. O novo parâmetro está sempre a direita. O sinal de igual não é um operador de atribuição neste contexto, ele meramente serve para copiar variáveis do programa para os parâmetros de interface de método.

Se a interface de um método consiste apenas de um simples parâmetro IMPORTING, você pode usar a seguinte forma resumida de chamada de método:

CALL METHOD <method>( f ).

O parâmetro <f> é passado para os parâmetros de entrada do método.

Se a interface de um método consiste apenas de parâmetros IMPORTING, você pode usar a seguinte forma resumida:

CALL METHOD <method>( ....<i<sub>i</sub>> =.<f<sub>i</sub>>... ).

Cada actual parameter <f<sub>i</sub>> é passado correspondenteis ao formal parameter <i<sub>i</sub>> .

## **Métodos de Manipulação de Eventos**

Métodos de manipulação de eventos são métodos especiais que não podem ser chamados usando o comando CALL METHOD. Eles são ativados usando eventos. Você define um método como um manipulador de eventos usando a adição

... FOR EVENT <evt> OF <clif>...

No comando METHODS ou CLASS-METHODS.

As seguintes regras especiais se aplicam para a interface de um método manipulador de eventos:

- A interface pode apenas consistir de parâmetros IMPORTING.
- Cada parâmetro IMPORTING deve ser um parâmetro EXPORTING no evento <evt>.
- Os atributos dos parâmetros são definidos na declaração do evento <evt> (comando EVENTS) e são adotados pelo método manipulador de eventos.

## Constructors

Constructors são métodos especiais que não podem ser chamados usando CALL METHOD. Ao invés, eles são chamados automaticamente pelo sistema para selecionar o estado inicial de um novo objeto ou classe. Há dois tipos de constructors – instance constructors e static constructors. Constructors são métodos com um nome pré-definido. Para usá-los, você deve explicitamente declará-los dentro da classe.

O instance constructor de uma classe é o método predefinido de um objeto CONSTRUCTOR. Você o declara da seguinte maneira na seção publica da classe.

## METHODS CONSTRUCTOR

```
IMPORTING.. [VALUE(<i>i</i>)] TYPE type [OPTIONAL]..
```

EXCEPTIONS..  $\langle e_i \rangle$ .

E implementa-lo na seção de implementação como qualquer outro método. O sistema chama o instance constructor uma vez para cada instancia da classe, diretamente **após** o objeto ter sido criado no comando CREATE OBJECT. Você pode passar parâmetros de entrada do instance constructor e lidar com exceções usando as adições EXPORTING e EXCEPTIONS.

O static constructor de uma classe é um método estático com o nome `CLASS_CONSTRUCTOR` pré-definido. Você o declara na seção pública como segue:

## CLASS-METHODS CLASS CONSTRUCTOR.

E implementa-lo na seção de implementação como qualquer outro método. O static constructor não possui parâmetros. O sistema chama o static constructor uma vez para cada classe, **antes** da classe ser acessada pela primeira vez. O static constructor **pode** então acessar componentes de sua própria classe.

## Métodos em Objetos ABAP - Exemplo

O exemplo a seguir demonstra como declarar, implementar, e usar métodos em objetos ABAP.

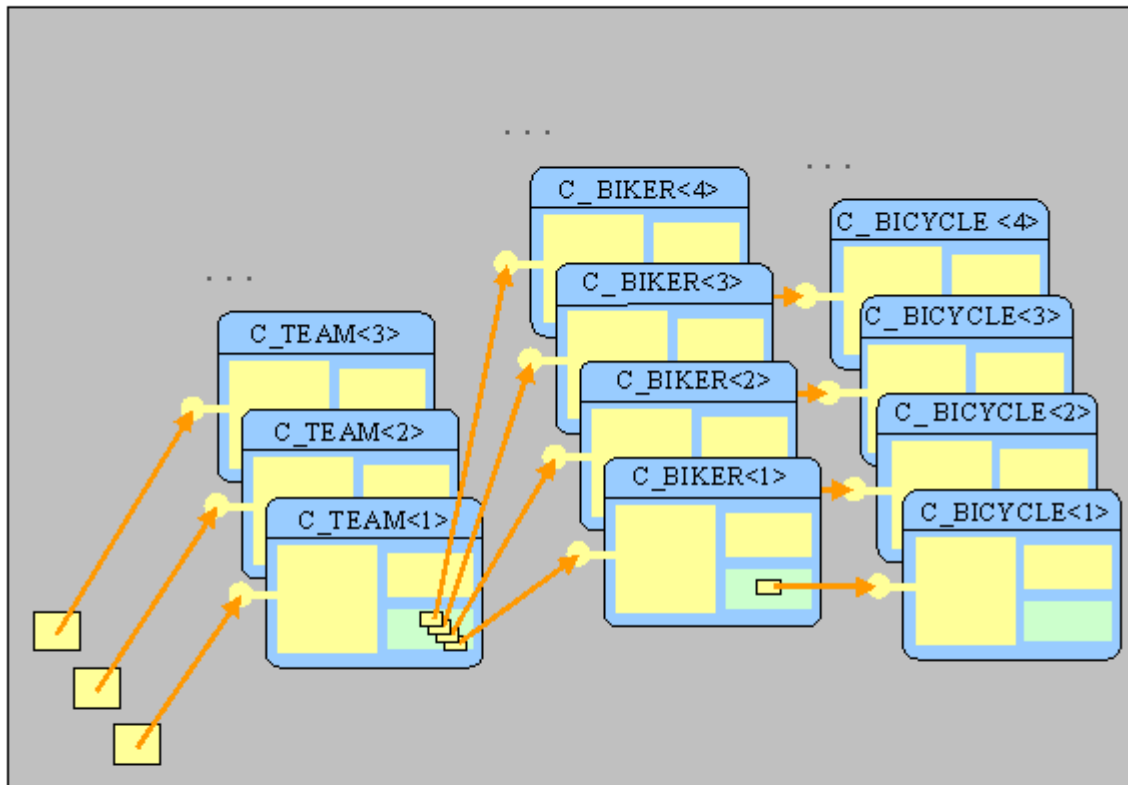
## Visão Geral

Este exemplo usa três classes chamadas C\_TEAM, C\_BIKER, e C\_BICYCLE. Um usuário (um programa) pode criar objetos da classe C\_TEAM. Em uma *selection screen*, a classe C\_TEAM pergunta pelo número de membros em cada time.

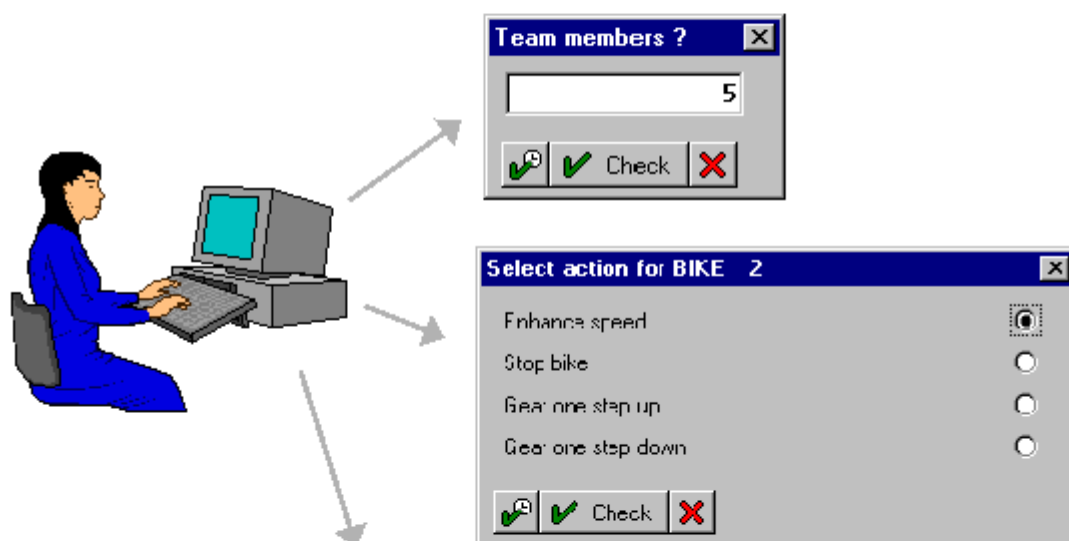
Cada objeto na classe C\_TEAM pode criar um máximo de instancias igual ao total da classe C\_BIKER de membros no time. Cada instancia da classe C\_BIKER cria uma instancia da classe C\_BICYCLE.

Cada instancia da classe C\_TEAM pode comunicar com o usuário do programa através de uma lista interativa. O usuário do programa pode escolher membros do time individuais para ações. As instancias da classe C\_BIKER permite que o usuário escolha a ação em uma *selection screen* posterior.

## Referenzvariablen und Instanzen:



## Benutzerinteraktion:



	Execute	Blue Team	Green Team	Red Team
<input type="checkbox"/> Biker	1	Status: Gear =	3	Speed = 0
<input checked="" type="checkbox"/> Biker	2	Status: Gear =	1	Speed = 10
<input checked="" type="checkbox"/> Biker	3	Status: Gear =	2	Speed = 0
<input type="checkbox"/> Biker	4	Status: Gear =	1	Speed = 20
<input type="checkbox"/> Biker	5	Status: Gear =	1	Speed = 0

## Conflitos



Os comandos usados para processamento de listas ainda não estão completamente disponíveis com objetos ABAP. No entanto, para produzir um simples teste, você pode usar o seguinte:

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

Nota: O comportamento da formatação e funções interativas de listagem em seu estado atual não são garantidos. Mudanças incompatíveis podem ocorrer em um release futuro.

## Declarações

Este exemplo foi implementado utilizando classes locais, já que *selection screens* pertencem ao programa ABAP, e não pode ser definido ou chamado em classes globais. Abaixo há definições de duas *selection screens* e três classes:

```
*****
* 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.

*****

```

Note que nenhuma das três classes tem atributos públicos. Os estados das classes podem ser modificados apenas pelos próprios métodos. O classe C\_TEAM Contém um CLASS\_CONSTRUCTOR. C\_TEAM e C\_BIKER ambos contém instance constructors.

## Implementações

As seções de implementação contêm as implementações de todos os métodos declarados nas partes correspondentes de declaração. As interfaces dos métodos já foram definidas na declaração. Nas implementações, os parâmetros de interface se comportam como dado local.

### Métodos da Classe C\_TEAM

Os métodos seguintes são implementados nesta seção:

```

CLASS C_TEAM IMPLEMENTATION.

...

ENDCLASS.

```



## CLASS\_CONSTRUCTOR

```
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.
```

O static constructor é executado antes da classe C\_TEAM ser usada pela primeira vez no programa. Ele chama a selection screen 100 e seleciona o atributo TEAM\_MEMBERS para o valor entrado pelo usuário. Este atributo tem o mesmo valor para todas as instancias da classe C\_TEAM.

## CONSTRUCTOR

```
METHOD CONSTRUCTOR.
  COUNTER = COUNTER + 1.
  ID = COUNTER.
ENDMETHOD.
```

O instance constructor é executada diretamente após cada instancia da classe C\_TEAM ser criada. Ela é usada para contar o número de instancias de C\_TEAM no atributo estático COUNTER, e assimila o número correspondente para o atributo ID de cada instancia da classe.

## CREATE\_TEAM

```
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.
```

O método público de instancia CREATE\_TEAM pode ser chamado por qualquer usuário da classe contendo uma variável com uma referência para a instancia da classe. É usada para criar instancias da classe C\_BIKER, usando a variável referencial **privada** BIKER na classe C\_TEAM. Você deve passar ambos parâmetros de entrada para o instance constructor da classe C\_BIKER no comando CREATE OBJECT. As referências para as novas instancias são inseridas dentro de uma tabela interna privada BIKER\_TAB. Após o método tiver sido executado, cada linha da tabela interna contém uma referência para uma instancia da classe C\_BIKER. Estas referências são apenas visíveis dentro da classe C\_TEAM. Usuários externos não podem endereçar os objetos da classe C\_BIKER.

CREATE\_TEAM também chama o método STATUS\_LINE para cada objeto novo, e usa a work area STATUS\_LINE para acrescentar o parâmetro de saída LINE à tabela interna privada STATUS\_LIST.

## SELECTION

```

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.

```

O método de instancia pública SELECTION pode ser chamado por qualquer usuário da classe contendo uma variável referencial com referência a uma instancia de uma classe. Ela seleciona todas as linhas na lista atual em quais a caixa de verificação na primeira coluna é selecionada. Para estas linhas, o sistema copia as variáveis referenciais correspondentes da tabela BIKER\_TAB dentro de uma tabela interna BIKER\_SELECTION. SELECTION então chama o método privado WRITE\_LIST, que mostra a lista.

## EXECUTION

```

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.

```

O método de instancia pública EXECUTION pode ser chamado por qualquer usuário da classe. O método chama dois métodos SELECT\_ACTION e STATUS\_LINE para cada instancia da classe C\_BIKER para qual há uma referência na tabela BIKER\_SELECTION. A linha da tabela STATUS\_LIST com a mesma chave do componente ID na work area STATUS\_LINE é sobrescrita e mostrada pelo método privado WRITE\_LIST.

## WRITE\_LIST

```

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.

```

```
ENDLOOP.
ENDMETHOD.
```

O método privado de instancia WRITE\_LIST pode apenas ser chamado de métodos da classe C\_TEAM. É usada para mostrar a tabela interna privada STATUS\_LIST na lista básica (SY-LSIND = 0) do programa.

## Métodos da Classe C\_BIKER

Os métodos seguintes são implementados na seção:

```
CLASS C_BIKER IMPLEMENTATION.
```

```
...
```

```
ENDCLASS.
```

## CONSTRUCTOR

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

O instance constructor é executado diretamente após cada instance da classe C\_BIKER ser criada. É usada para contra o número de instance de C\_BIKER no atributo estático COUNTER, e define os números correspondents para o atributo de instance ID de cada instance da classe. O constructor tem dois parâmetros de entrada – TEAM\_ID e MEMBERS – os quais você deve passar no comando CREATE OBJECT quando você cria uma instance em C\_BIKER.

O instance constructor também cria uma instance da classe C\_BICYCLE para cada nova instance da classe C\_BIKER. A referência na variável referencial privada BIKE de cada instance de C\_BIKER indica para uma instance correspondente da classe C\_BICYCLE. Nenhum usuário externo pode endereçar estas instances da classe C\_BICYCLE.

## SELECT\_ACTION

```
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.
```

O método da instancia público SELECT\_ACTION pode ser chamado por qualquer usuário da classe contendo uma variável referencial com uma referência para uma instancia da classe. O método chama a selection screen 200 e analisa a entrada do usuário. Depois disto, chama o método privado BIKER\_ACTION da mesma classe. A chamada do método usa a forma reduzida para passar o parâmetro atual ACTIVITY para parâmetro formal ACTION.

## BIKER\_ACTION

```
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.
```

O método de instancia privado BIKER\_ACTION pode ser apenas chamado dos métodos da classe C\_BICYCLE para qual a referência na variável referencial BIKE está indicando, dependendo do valor na entrada do parâmetro ACTION.

## STATUS\_LINE

```
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.
```

O método da instancia pública STATUS\_LINE pode ser chamado por qualquer usuário da classe. Isto atribui ao parâmetro estruturado de saída LINE com os valores de atributos atuais da instancia correspondente.

## Métodos da Classe C\_BICYCLE

Os métodos seguintes são implementados na seção:

```
CLASS C_BICYCLE IMPLEMENTATION.
```

```
...
```

```
ENDCLASS.
```

### DRIVE

```
METHOD DRIVE.  
    SPEED = SPEED + GEAR * 10.  
    VELOCITY = SPEED.  
ENDMETHOD.
```

O método da instancia pública DRIVE pode ser chamado por qualquer usuário da classe. O método muda os valores do atributo privado SPEED e passa-o para o usuário usando o parâmetro de saída VELOCITY.

### STOP

```
METHOD STOP.  
    SPEED = 0.  
    VELOCITY = SPEED.  
ENDMETHOD.
```

O método da instancia pública STOP pode ser chamado por qualquer usuário da classe. O método muda os valores do atributo privado SPEED e passa-o para o usuário o parâmetro de saída VELOCITY.

### CHANGE\_GEAR

```
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.
```

O método da instancia pública CHANGE\_GEAR pode ser chamado por qualquer usuário da classe. O método muda o valor do atributo privado GEAR. Já que o parâmetro formal com o mesmo nome obscurece o atributo dentro do método, o atributo tem de ser endereçado usando a referência própria ME->GEAR (ME referencia a instância que chamou o método).

Usando as Classes dentro de um Programa

O programa a seguir mostra como as classes acima podem ser usadas em um programa. As declarações da selection screen e classes locais, e as implementações dos métodos também devem ser parte do 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.
*****

```

O programa contém três variáveis referenciais a classes que se referem a classe C\_TEAM. Ele cria 3 instancias para as variáveis de referência e cada objeto, chama-se o método CREATE\_TEAM. O método CLASS\_CONSTRUCTOR da classe C\_TEAM é executado antes da criação do primeiro objeto. O status TEAMLIST para a lista básica permite que o usuário escolha uma de quatro funções:

F5	TEAM_BLUE	Blue Team	
F6	TEAM_GREEN	Green Team	
F7	TEAM_RED	Red Team	
F8	EXECUTION	Execute	

Quando o usuário escolhe uma função, o evento AT USER-COMMAND é ativado e os métodos públicos são chamados em uma das três instancias de C\_TEAM, dependendo da escolha do usuário. O usuário pode mudar o estado objeto selecionando a linha correspondente na lista de status.

## Herança

A herança permite que você derive uma nova classe de uma já existente. Você faz isso usando a adição INHERITING FROM do modo a seguir:

```
CLASS <subclasse> DEFINITION INHERITING FROM <superclasse>.
```

A nova classe <subclasse> herda todos os componentes da classe existente <superclasse>. A nova classe é chamada derivada da qual ela herdou. E a original é chamada superclasse da nova classe.

Se você adicionar novas declarações à subclasse ela conterá os mesmos componentes da superclasse, mais as adições. No entanto, apenas os componentes **públicos** e **protegidos** da superclasse estão visíveis na subclasse. Apesar dos componentes privados da superclasse existirem na subclasse, eles não estão visíveis. Você pode declarar componentes privados em uma subclasse que tem os mesmos nomes dos componentes privados da superclasse. Cada classe trabalha com seus próprios componentes privados.

Se a superclasse não tem uma seção de visibilidade privada, a subclasse é uma réplica exata da superclasse. No entanto, você pode adicionar novos componentes à subclasse. Isto lhe permite que torne uma subclasse em uma versão especializada da superclasse. Se uma subclasse é também uma superclasse, você introduz um novo nível de especialização.

Uma classe pode ter mais do que uma subclasse direta, mas apenas uma superclasse direta. Isto é chamada **herança simples**. Quando subclasses herdarem de superclasses e a superclasse é também uma subclasse de outra classe, todas as classes envolvidas formam uma árvore de herança, cujos degraus de especialização aumentam com cada novo nível hierárquico que você adicionar. Ao contrário, as classes se tornam mais generalizadas até se chegar a raiz da árvore de herança. A raiz de todas as árvores de herança em Objetos ABAP é a predefinida classe vazia OBJECT. Esta é a classe mais generalizada possível, já que não contém métodos nem atributos. Quando você define uma classe, você não precisa especificar explicitamente a superclasse – a relação é sempre definida implicitamente. Dentro de uma árvore de herança, dois nós adjacentes são a superclasse direta ou subclasse direta uma da outra. Outros nós relacionados são referidos como superclasse ou subclasse. A declaração de componentes de uma subclasse são distribuídas através todos os níveis da árvore de herança.

### **Redefinindo Métodos**

Todas as subclasses contêm componentes de todas as classes entre elas e o nó-raiz em uma árvore de herança. A visibilidade de um componente não pode ser mudada. No entanto, você pode usar a adição REDEFINITION no comando METHODS para redefinir método de instancia público ou protegido em uma subclasse e fazê-la mais especializada. Quando se redefine um método, você não pode mudar sua interface. O método continua com o mesmo nome e interface, mas possui uma nova implementação.

A declaração e implementação do método na superclasse não são afetadas quando você redefine o método dentro de uma subclasse. A implementação da redefinição na subclasse obscurece a implementação original na superclasse.

Qualquer referência que indique para um objeto da subclasse usa o método redefinido, até mesmo se a referência foi definida com referência a superclasse. Isto particularmente se aplica a auto-referência ME->. Se, por exemplo, um método de superclasse M1 contém uma chamada CALL METHOD [ME->]M2, e M2 é redefinido em uma subclasse, chamando M1 de uma instancia da subclasse que vai fazer com que o método M2 original seja chamado, e chamando M1 de uma instancia da subclasse vai fazer com que o método M2 redefinido seja chamado.

Dentro de um método redefinido, você pode usar a pseudo-referência SUPER-> para acessar o método obscuro. Isto lhe permite usar a função existente do método dentro da superclasse sem ter de recodificá-lo na subclasse.

### **Métodos e Classes Abstratos e Finais**

As adições **ABSTRACT** e **FINAL** nos comandos METHODS e CLASS permitem que você defina métodos e classes abstratos e finais.



Um método abstrato definido dentro de uma classe abstrata e não pode ser implementado naquela classe. Ao invés, é implementada em uma subclasse da classe. Classes abstratas não podem ser instanciadas.

Um método final não pode ser redefinido em uma subclasse. Classes finais não podem ter subclasses. Eles concluem uma árvore de herança.

### **Referências a Subclasses e Polimorfismo**

Variáveis referenciais definidas como referência para uma superclasse ou uma interface definida com referência para ela também contém referências para qualquer uma de suas subclasses. Já que as subclasses contêm todos os componentes de todas as suas superclasses, e dado que a interface dos métodos não pode ser mudada, uma variável referencial definida com referência a uma superclasse ou uma interface implementada por uma superclasse pode conter referências para instâncias de qualquer de suas subclasses. Em particular, você pode definir o variável alvo com referência a classe genérica OBJECT.

Quando você cria um objeto usando o comando CREATE OBJECT e uma variável referencial preenchida com referência a uma subclasse, você pode usar a adição TYPE para criar uma instância de subclasse, para qual a referência na variável referencial irá indicar.

Um usuário estático pode usar uma variável referencial para endereçar os componentes visíveis a ela na superclasse para qual a variável referencial indica. No entanto, não pode acessar qualquer especialização na subclasse. Se você usar um método de chamada dinâmico, você pode endereçar todos os componentes da classe.

Se você redefinir um método de instancia em um ou mais subclasses, você pode usar uma única variável referencial para chamar diferentes implementações do método, dependendo da posição na árvore de herança na qual o objeto referido ocorre. Este conceito que diferentes classes podem ter a mesma interface e, portanto serem endereçados usando variáveis referenciais com um simples tipo é chamado polimorfismo.

### **Espaço de Nomes para Componentes**

As subclasses contêm todos os componentes de todas as superclasses dentro da árvore de herança acima. Destes componentes, apenas os públicos e protegidos são visíveis. Todos os componentes públicos e protegidos dentro de uma árvore de herança pertencem ao mesmo espaço de nomes, e conseqüentemente devem possuir nomes únicos. O nome dos componentes privados, por outro lado, devem apenas ser únicos dentro de sua classe.

Quando você redefine métodos, a nova implementação dos métodos obscurece o método da superclasse com o mesmo nome. No entanto, a nova definição substitui o método anterior de implementação, então o nome ainda é único. Você pode usar pseudo-referência SUPER-> para acessar uma definição de métodos dentro de uma superclasse que tenha sido obscura por uma redefinição de uma subclasse.

### **Herança e Atributos Estáticos**

Como todos componentes, atributos estáticos apenas existem uma vez na árvore de herança. Uma subclasse pode acessar os atributos estáticos e protegidos de todas as suas superclasses. Ao contrário, uma superclasse compartilha os atributos estáticos protegidos e públicos com todas as suas subclasses. Em termos de herança, atributos estáticos não são atribuídos para uma única classe, mas a uma parte da árvore de herança. Você pode mudá-las de fora da classe usando o seletor de componentes de classes com qualquer nome de classe, ou dentro de qualquer classe em que ela sejam compartilhadas. Elas são visíveis em todas as classes na árvore de herança.

Quando se acessa um atributo estático, sempre se endereça a classe cujo atributo é declarado. Isto é particularmente importante quando você chama o static constructor de classes em herança. Static constructors são executados na primeira vez em que a classe é acessada. Se você acessar

um atributo estático declarado em uma superclasse usando o nome da classe de uma subclasse, apenas o static constructor é executado.

### ***Herança e Constructors***

Há regras especiais gerenciando constructors em herança.

### **Instance Constructors**

Toda classe tem um instance constructor chamado CONSTRUCTOR. Esta é uma exceção a regra de que nomes de componentes dentro de uma árvore de herança devem ser únicos. No entanto, os instance constructors de várias classes dentro de uma árvore de herança são totalmente independentes um do outros. Você não pode redefinir o instance constructor de uma superclasse dentro de uma subclasse, nem chamar um especificamente usando o comando CALL METHOD CONSTRUCTOR. Conseqüentemente, conflitos de nomes não podem ocorrer.

O instance constructor de uma classe é chamado pelo sistema quando se cria uma instancia de classe usando CREATE OBJECT. Já que uma subclasse contém todos os atributos visíveis de suas superclasses, que são também preparadas pelos instance constructors de todas as suas superclasses, tais métodos também devem ser chamados. Para fazer isso, o instance constructor de cada subclasse deve conter um comando CALL METHOD SUPER->CONSTRUCTOR. As únicas exceções a essa regra são subclasses diretas do nó-raiz OBJECT.

Em superclasses sem um instance constructor explicitamente definido, o instance constructor implícito é chamado. Isto automaticamente garante que o instance constructor da superclasse imediata é chamada.

Quando você chama um instance constructor, você deve passar valores para todos os parâmetros de interface obrigatórios. Há varias maneiras de fazer isto:

- Usando CREATE OBJECT

Se a classe que você está criando uma instancia tem um instance constructor com uma interface, deve-se passar valores para ela usando EXPORTING.

Se a classe que você está criando uma instancia tem um instance constructor sem uma interface, não se deve passar parâmetros.

Se a classe que você está criando uma instancia não possui um instance constructor explícito, você deve olhar na árvore de herança pela próxima superclasse que possua um instance constructor explícito. Caso haja uma interface, você deve passar valores usando EXPORTING, senão não há necessidade de passar qualquer valor.

- Usando CALL METHOD SUPER->CONSTRUCTOR

Se a superclasse direta tem um instance constructor com uma interface, você deve passar valores para ela usando EXPORTING.

Se a superclasse direta tem um instance constructor sem uma interface, não se passam parâmetros.

Se a superclasse direta não tem um instance constructor explícito, você deve olhar na árvore de herança pela próxima superclasse com um instance constructor explícito. Se possuir uma interface, você deve passar valores usando EXPORTING. Senão, não se deve passar quaisquer valores.

Em ambos CREATE OBJECT e CALL METHOD SUPER->CONSTRUCTOR, você deve olhar ao próximo instance constructor explícito disponível e, se tiver uma interface, passar valores para ele. O mesmo se aplica para manipulação de exceções para instance constructors. Quando se trabalha com herança, é necessário um conhecimento preciso de toda a árvore de herança. Quando você cria uma instância para uma classe no nível inferior da árvore de herança, você pode precisar passar parâmetros para o constructor de uma classe que esteja mais próxima do nó-raiz.

O instance constructor de uma subclasse é dividido em duas partes pelo comando CALL METHOD SUPER->CONSTRUCTOR. Nas linhas antes da chamada, o construtor se comporta como um método estático, isto é, não pode acessar os atributos de instância da sua classe. Você não pode acessar atributos de instância até depois da chamada. Use os comandos antes da chamada para determinar os parâmetros atuais para a interface do instance constructor da superclasse. Você pode apenas usar atributos estáticos ou dados locais para fazer isto.

Quando você cria uma instância de uma subclasse, o instance constructor é chamado hierarquicamente. O primeiro nível em qual você pode acessar atributos de instância é o nível mais alto superclasse. Quando você retorna ao constructor da classe com nível mais baixo, você pode também sucessivamente acessar seus atributos de instância.

Em um método constructor, os métodos das subclasses da classe não são visíveis. Se um instance constructor chama um método de instância da mesma classe usando a auto-referência implícita ME->, o método é chamado como se fosse implementado na classe do instance constructor, e não em qualquer forma redefinida que pode ocorrer na subclasse que você que criar uma instance. Esta é uma exceção à regra que determina que quando você chamar métodos de instances, o sistema sempre chama o método como se tivesse sido implementado na classe cuja instance a referência está indicando.

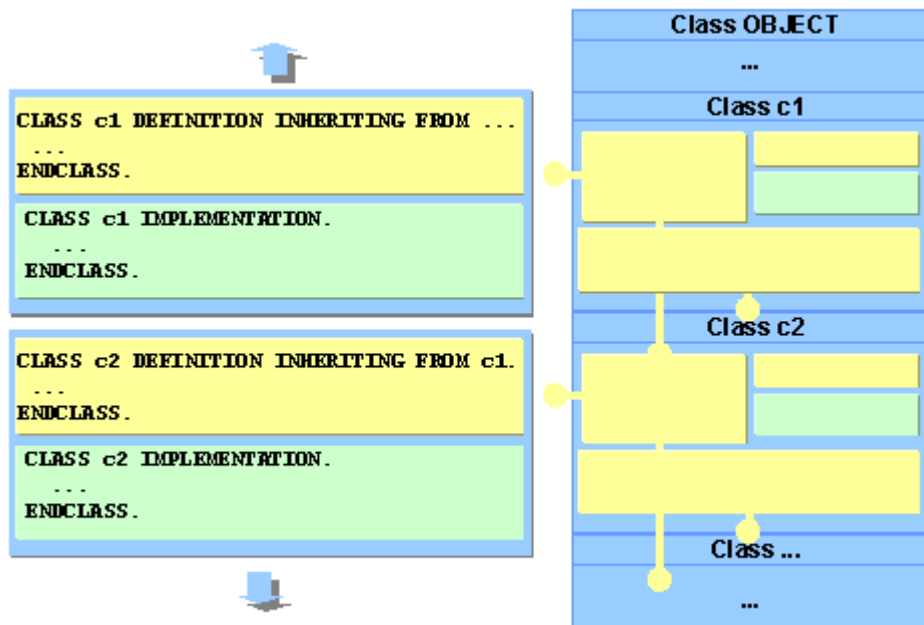
## Static Constructors

Toda classe tem um static constructor chamado CLASS\_CONSTRUCTOR. Assim como o espaço para nomes dentro de uma árvore hierárquica, as mesmas regras de um instance constructor se aplicam a um static constructor.

A primeira vez que você acessa uma subclasse em um programa, o static constructor é executado. No entanto, antes que possa ser executado, os static constructors de todas as superclasses devem já ter sido executados. Um static constructor pode apenas ser chamado uma vez por programa. Portanto, quando você acessa uma subclasse, o sistema procura pela superclasse mais próxima cujo static constructor ainda não foi executado, seguido de todas as classes entre aquela e a subclasse que você acessou.

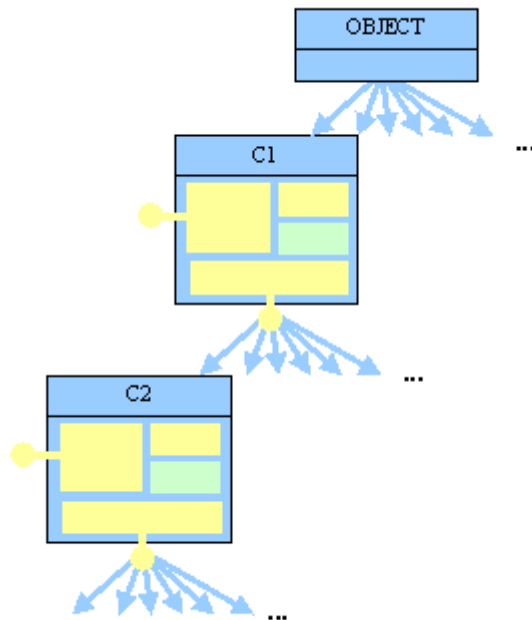
## Gráfico Geral de Herança

### Herança: Visão Geral



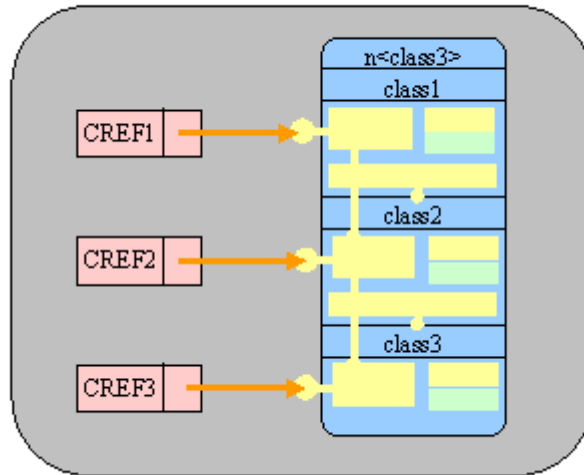
A parte esquerda do gráfico demonstra como você pode derivar uma subclasse c2 de uma superclasse c1 usando a adição INHERITING FROM no comando CLASS. A parte direita do gráfico demonstra como é a distribuição da subclasse dentro da árvore de herança, que alcança a classe vazia default OBJECT. Uma subclasse contém todos os componentes declarados acima dela na árvore de herança, e pode acessar aqueles que são declarados públicos ou protegidos.

## Herança Simples



Este gráfico ilustra herança simples. Uma classe pode apenas ter uma superclasse direta, mas pode ter mais do que uma subclasse direta. A classe vazia OBJECT é o nó-raiz de toda árvore de herança em objetos ABAP.

## Herança e Variáveis Referenciais



Este gráfico demonstra como variáveis referenciais definidas com referência para uma superclasse pode indicar objetos de subclasses. O objeto a direita é uma instancia da classe class3. As variáveis referenciais de classe CREF1, CREF2, e CREF3 são digitadas com referência a class1, class2, e class3. Todas as três variáveis referenciam ao objeto. No entanto, CREF1 pode apenas acessar os componentes públicos da classe class1, CREF2 pode acessar os componentes públicos da class1 e class2. CREF3 pode acessar os componentes públicos de todas as classes.

Se você redefinir um método de uma superclasse em uma subclasse, você pode usar uma variável referencial definida com referência a superclasse para acessar objetos com diferentes implementações de métodos. Quando se acessa a superclasse, o método tem a implementação original, mas quando você acessa a subclasse, o método tem a nova implementação. O uso de uma simples variável de referência para chamar métodos com mesmo nome que se comportam diferentemente é chamado polimorfismo. O tipo estático é sempre uma superclasse. O tipo dinâmico(tipo da instância) pode ser o da superclasse ou subclasse.

```

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.

```

A classe COUNTER\_TEN é derivada de COUNTER. Ela redefine o método INCREMENT. Para fazer isto, deve-se modificar a visibilidade do atributo COUNT de privado para protegido. O método redefinido chama o método obscuro da superclasse usando a pseudo-referência SUPER->. O método redefinido é uma especialização do método herdado.

O exemplo cria uma instancia da subclasse. A variável referencial indicando para ela tem o tipo da superclasse (tipo estático). Quando o método INCREMENT é chamado usando a referência superclasse, o sistema executa o método redefinido da subclasse.

## Interfaces

Classes, suas instancias (objetos), e acesso a objetos usando variáveis referenciais formam o básico de objetos ABAP. Estes meios já lhe permitem modelar aplicações típicas de negócios, como consumidores, ordens, ordens de itens, entre outros, usando objetos, e para implementar soluções usando objetos ABAP.

No entanto, é freqüentemente necessário para classes similares fornecer funções similares que são codificadas diferentemente em cada classe, mas que deveria fornecer um ponto uniforme de contato com o usuário. Por exemplo, você pode ter duas classes similares, conta poupança e contas corrente, ambas as quais tem um método de calcular juros ao ano. As interfaces e nomes dos métodos são os mesmos, mas a verdadeira implementação é diferente. O usuário das classes e suas instancias devem também estar aptos a executar o método de final de ano para todas as contas, sem ter de se preocupar individualmente com cada tipo de conta.

Objetos ABAP permitem isto através do uso de interfaces. Interfaces são estruturas independentes que você pode implementar em uma classe para ampliar o corpo da classe. O corpo específico de uma classe é definido por seus componentes e seções de visibilidade. Por exemplo, os componentes públicos de uma classe definem o seu corpo público, já que todos os seus atributos podem ser endereçados por todos os usuários. Os componentes protegidos de uma classe definem seu corpo em conjunto com suas subclasses. (No entanto, herança não é suportada no Release 4.5B).

Interfaces ampliam o corpo de uma classe adicionando seus próprios componentes a seção pública. Isto permite que usuários enderecem diferentes classes via um ponto universal de contato. Interfaces, junto com herança, fornece um dos pilares do polimorfismo, já que eles permitem um método simples dentro de uma interface se comporte diferentemente em diferentes classes.

### ***Definindo Interfaces***

Como classes, você pode definir interfaces tanto globalmente no R/3 Repository ou localmente em um programa ABAP. Para informações sobre como definir interfaces locais leia a seção Class Builder na documentação de ferramentas do ABAP Workbench. A definição de uma interface local <intf> é incluída nos comandos:

```
INTERFACE <intf>.
```

```
...
```

```
ENDINTERFACE.
```

A definição contém a declaração para todos os componentes (atributos, métodos, eventos) da interface. Você pode definir os mesmos componentes em uma interface assim como em uma classe. Os componentes de interfaces não têm de ser atribuídos individualmente para uma seção de visibilidade, já que eles automaticamente pertencem a seção pública da classe em qual a interface é implementada. Interfaces não possuem uma parte de implementação, já que seus métodos são implementados dentro da classe que implementa a interface.

### ***Implementando Interfaces***

Ao contrário de classes, interfaces não possuem instancias. Ao invés, interfaces são implementadas por classes. Para implementar uma interface dentro de uma classe, use o comando:

```
INTERFACES <intf>.
```

Na parte de declarações da classe. Este comando pode apenas aparecer na seção pública da classe.

Quando você implementa uma interface dentro de uma classe, os componentes da interface são adicionados aos outros componentes na seção pública. Um componente <icomp> de uma interface <intf> pode ser acessada como se fosse uma membro da classe sob o nome <intf~icomp>.



A classe deve implementar os métodos de todas as interfaces implementadas nela. A parte de implementação da classe deve conter um método de implementação para cada método de interface <imeth>:

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

```
...
```

```
ENDMETHOD.
```

Interfaces podem ser implementadas por diferentes classes. Cada uma destas classes é ampliada pelo mesmo conjunto de componentes. No entanto, os métodos da interface podem ser implementados diferentemente dentro de cada classe.

Interfaces permitem que você use diferentes classes em um modo uniforme usando referências de interface (polimorfismo). Por exemplo, interfaces que são implementadas em diferentes classes e ampliam o corpo público de cada classe em um mesmo conjunto de componentes. Se uma classe não tem nenhum componente público específico dela, as interfaces definem todo o corpo público da classe.

### **Referência de Interface**

Variáveis referenciais lhe permitem acessar objetos. Ao invés de criar variáveis referenciais com referência a uma classe, você pode também definí-los com referência a uma interface. Este tipo de variável referencial pode conter referências a objetos de classes que implementam a interface correspondente.

Para definir uma referência de interface, use a adição TYPE REF TO <intf> no comando TYPES ou DATA. <intf> deve ser uma interface que foi declarada ao programa antes da verdadeira declaração da referência ocorrer. Uma variável referencial com a referência de interface type é chamada uma variável referencial de interface ou referência de interface.

Uma referência de interface <iref> permite que um usuário utilize a forma <iref>-><icomp> para endereçar todos os componentes visíveis da interface <icomp> do objeto cujo objeto de referência está indicando. Isso permite que o usuário acesse todos os componentes do objeto que foram adicionados a sua definição pela implementação da interface.

### **Endereçando Objetos Usando Referências de Interfaces**

Para criar um objeto da classe <class>, você deve primeiro ter declarado uma variável referencial <cref> com referência à classe. Se a classe <class> implementa uma interface <intf>, você pode usar a seguinte atribuição entre a variável referencial da classe <cref> e uma referência de interface <iref> para fazer com que a referência de interface em <iref> indique o mesmo objeto da referência de classe <cref>:

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

Se a interface <intf> contém um atributo de instancia <attr> e um método de instancia <meth>, você pode acessar os componentes da interface como a seguir:

Usando a **variável referencial de classe** <cref>:

- Para acessar um atributo <attr>: <cref>-><intf~attr>
- Para chamar um método <meth>: **CALL METHOD** <cref>-><intf~meth>

Usando a **variável referencial de interface** <iref>:

- Para acessar um atributo <attr>: `< iref>-><attr>`
- Para chamar um método <meth>: `CALL METHOD <iref>-><meth>`

Em relação a componentes estáticos de interfaces, você pode apenas utilizar o nome da interface para acessar constantes:

Acessando uma constante <const>: `< intf>=><const>`

Para todos os outros componentes estáticos de uma interface, você pode apenas usar referências a objetos ou a classe <class> que implementa a interface.

Endereçando um atributo estático <attr>: `< class>=><intf~attr>`

Chamando um método estático <meth>: `CALL METHOD <class>=><intf~meth>`

## ***Atribuição Usando Referências de Interfaces***

Assim como referências de classes, você pode atribuir referências de interfaces para diferentes variáveis referenciais. Você pode também fazer atribuições entre variáveis referenciais de classes e variáveis referenciais de interfaces. Quando você usa o comando MOVE ou o operador de atribuição (=) para atribuir variáveis referenciais, o sistema deve estar apto a reconhecer na verificação de sintaxe se a atribuição é possível.

Suponha que tenhamos uma referência de classe <cref> e referências de interfaces <iref>, <iref1>, e <iref2>. As seguintes atribuições com referências de interfaces podem ser checadas estaticamente:

- `<iref1> = <iref2>`

Ambas referências de interfaces devem indicar à mesma interface, ou a interface de <iref1> deve conter a interface <iref2> como um componente.

- `<iref> = <cref>`

A classe da referência de classe <cref> deve implementar a interface da referência de interface <iref>.

- `<cref> = <iref>`

A classe de <cref> deve ser predefinida como a classe vazia OBJECT.

Em todos os outros casos, você teria que trabalhar com o comando MOVE ...?-TO ou o **casting operator** (=?). O casting operator substitui o operador de atribuição (=). No comando MOVE...?-TO, ou quando você utiliza o casting operator, não há verificação de tipo estático. Ao invés, o sistema verifica em **tempo de execução** se a referência do objeto na variável de origem indica um objeto cuja referência de objeto no alvo a variável também pode indicar. Se a atribuição é possível, o sistema a faz, senão, o erro em tempo de execução MOVE\_CAST\_ERROR ocorre.

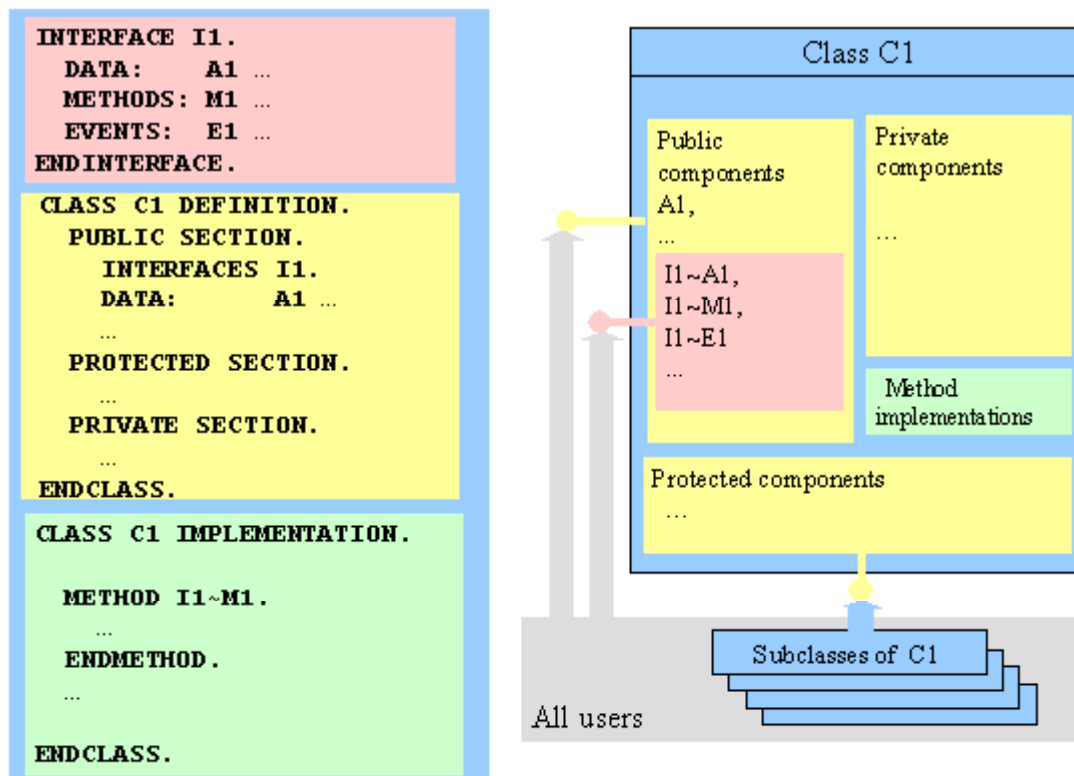
Você deve sempre usar casting para atribuir uma referência de interface a uma referência de classe se <cref> não indica a classe predefinida vazia OBJECT:

`<cref> ?= <iref>`

para o type cast ter sucesso, o objeto o qual <iref> indica deve ser um objeto da mesma classe como o tipo da variável de classe <cref>.

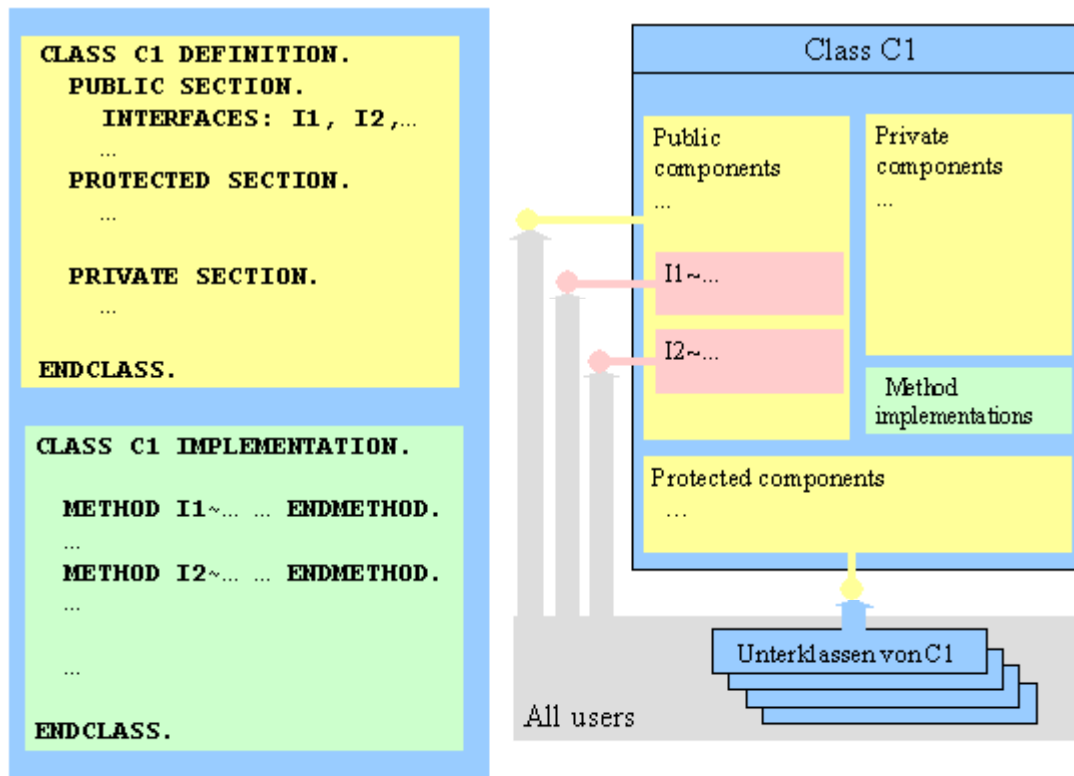
## Gráficos Gerais

### Interfaces



A figura à esquerda do diagrama demonstra a definição de uma interface local I1 e as partes da declaração e implementação da classe local C1 que implementa a interface I1 em sua **seção pública**. O método de interface I1~M1 é implementado na classe. Você não pode implementar interfaces em outras seções de visibilidade.

A figura à direita ilustra a estrutura da classe com os componentes em suas respectivas áreas de visibilidade, e a implementação dos métodos. Os componentes da interface ampliam o corpo público da classe. Todos os usuários podem acessar os componentes públicos específicos da classe e os componentes da interface.



Uma classe pode implementar um número indefinido de interfaces em sua seção pública. A figura à esquerda do diagrama demonstra a parte de declaração e implementação de uma classe local C1, que implementa diversas interfaces em sua seção pública. A classe deve implementar todos os métodos de interface em sua parte de implementação.

A figura à direita ilustra a estrutura da classe com os componentes em suas respectivas áreas de visibilidade, e a implementação dos métodos. Os componentes da interface ampliam a seção pública da classe em qual eles foram declarados.

## Interfaces - Exemplo Introductório

O exemplo simples a seguir demonstra como se pode usar uma interface para implementar dois contadores que são diferentes, mas podem ser acessados da mesma maneira. Veja também o exemplo da seção de Classes.



```
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.

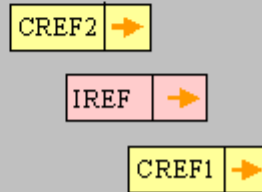
CLASS C_COUNTER2 IMPLEMENTATION.
  METHOD I_COUNTER~SET_COUNTER.
    COUNT = ( SET_VALUE / 10 ) * 10.
  ENDMETHOD.
  METHOD I_COUNTER~INCREMENT_COUNTER.
    IF COUNT GE 100.
      MESSAGE I042(00).
      COUNT = 0.
    ELSE.
      ADD 10 TO COUNT.
    ENDIF.
  ENDMETHOD.
  METHOD I_COUNTER~GET_COUNTER.
    GET_VALUE = COUNT.
  ENDMETHOD.
ENDCLASS.

```

A interface I\_COUNTER contém três métodos: SET\_COUNTER, INCREMENT\_COUNTER, e GET\_COUNTER. As classes C\_COUNTER1 e C\_COUNTER2 implementam a seção pública da interface. Ambas as classes devem implementar os três métodos de interface em sua parte de implementação. C\_COUNTER1 é uma classe para contadores que pode ter qualquer valor inicial e então são acrescidos de um. C\_COUNTER2 é uma classe para contadores que podem apenas ser acrescidos de 10 em 10. Ambas as classes tem uma interface de saída idêntica.

A seção a seguir explica como um usuário pode usar uma referência de interface para acessar os objetos em ambas as classes.

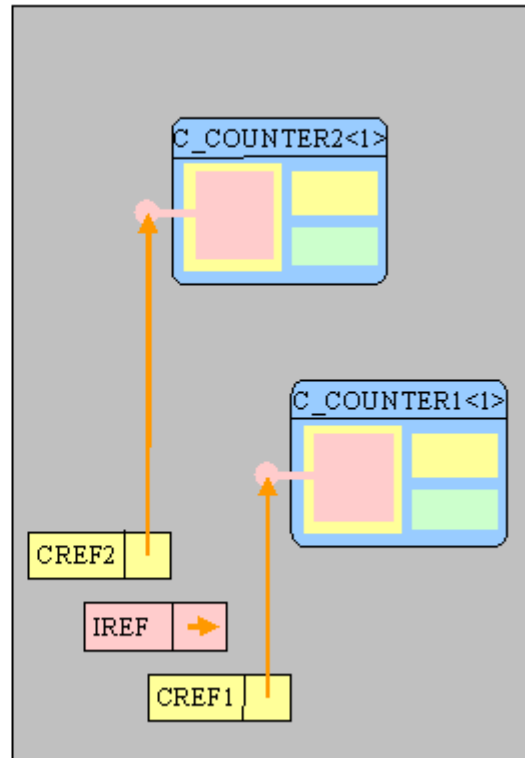
```
DATA: CREF1 TYPE REF TO C_COUNTER1,  
      CREF2 TYPE REF TO C_COUNTER2,  
      IREF  TYPE REF TO I_COUNTER.
```



Primeiro, duas variáveis referenciais de classe CREF1 e CREF2 são declaradas para as classes C\_COUNTER1 e C\_COUNTER2. Uma referência de interface IREF é também declarada para a interface I\_COUNTER. Todas as variáveis referenciais são iniciais.

```
DATA: CREF1 TYPE REF TO C_COUNTER1,  
      CREF2 TYPE REF TO C_COUNTER2,  
      IREF  TYPE REF TO I_COUNTER.
```

```
CREATE OBJECT: CREF1, CREF2.
```

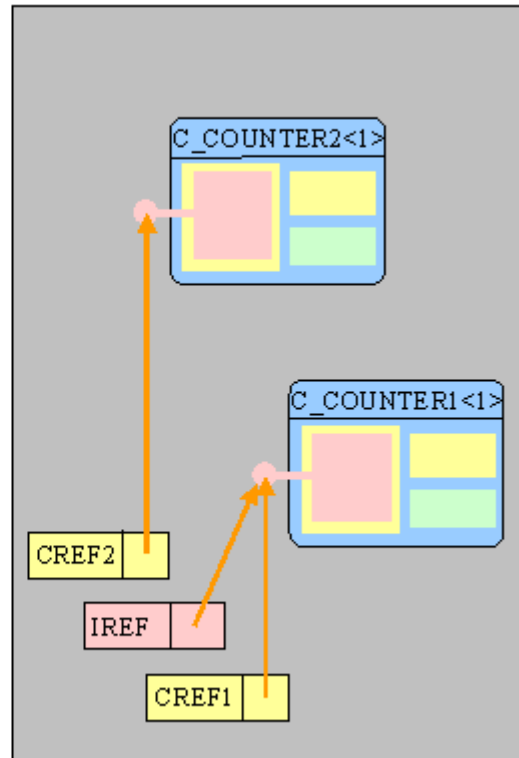


O comando `CREATE OBJECT` cria um objeto para cada classe que as referências `CREF1` e `CREF2` indicam.

```
DATA: CREF1 TYPE REF TO C_COUNTER1,  
      CREF2 TYPE REF TO C_COUNTER2,  
      IREF TYPE REF TO I_COUNTER.
```

```
CREATE OBJECT: CREF1, CREF2.
```

```
IREF = CREF1.
```



Quando a referência de CREF1 é atribuída a IREF, a referência em IREF também indica ao objeto com o nome interno C\_COUNTER<1>.

## Ativando e Manipulando Eventos

Em objetos ABAP, ativar e manipular um evento significa que certos métodos agem como **gatilhos** que disparam eventos, ao quais outros métodos – os **manipuladores** – reagem. Isto significa que os métodos de manipulação são executados quando o evento ocorre.

Esta seção contém explicação de como trabalhar com eventos em objetos ABAP. Para detalhes precisos dos comandos ABAP relevantes, procure a palavra-chave correspondente no ABAP Editor.

### Ativando Eventos

Para disparar um evento, uma classe deve

- Declarar o evento em sua parte de declarações
- Ativar o evento em um de seus métodos

### Declarando Eventos

Você declara eventos na parte de declaração de uma classe ou interface. Para declarar eventos de instancia, use o seguinte comando:



EVENTS <evt> EXPORTING... VALUE(<e<sub>i</sub>>) TYPE type [OPTIONAL]..

Para declarar eventos estáticos, use o comando a seguir:

CLASS-EVENTS <evt>...

Ambos os comando têm a mesma sintaxe.

Quando se declara um evento, você pode usar a adição EXPORTING para especificar parâmetros que são passados ao manipulador de eventos. Os parâmetros são sempre passados por valor. Eventos de instancia sempre contém o parâmetro implícito SENDER, que tem o tipo de uma referência para o tipo ou interface em qual o evento é declarado.

### **Disparando Eventos**

Um evento de instancia em uma classe pode ser disparado por qualquer método na classe. Eventos estáticos podem ser ativados por qualquer método estático. Para ativar um evento em um método, use o comando a seguir:

RAISE EVENT <evt> EXPORTING... <e<sub>i</sub>> = <f<sub>i</sub>>...

Para cada parâmetro formal <e<sub>i</sub>> que é definido como obrigatório deve-se passar um parâmetro atual correspondente <f<sub>i</sub>> na adição EXPORTING. A auto-referência ME é automaticamente passada para o parâmetro implícito SENDER.

### **Manipulando Eventos**

Eventos são manipulados usando métodos especiais. Para manipular um evento, um método deve:

- Ser definido como manipulador para aquele evento
- Ser registrado em tempo de execução para o evento.

### **Declarando Métodos de Manipulação de Eventos**

Qualquer classe pode conter métodos de manipulação de eventos para eventos de outras classes. Você pode, claro, também definir métodos de manipulação de eventos na mesma classe que o próprio evento. Para declarar um método de manipulação de evento de instancia, use o comando a seguir:

METHODS <meth> FOR EVENT <evt> OF <cif> IMPORTING.. <e<sub>i</sub>>..

Para um método estático, use CLASS-METHODS ao invés de METHODS: <evt> é um evento declarado na classe ou interface <cif>.

A interface de um método de manipulação pode apenas conter parâmetros formais definidos na declaração do evento <evt>. Os atributos do parâmetro também são adotados pelos eventos. O método de manipulação do evento não tem que usar todos os parâmetros passados no comando RAISE EVENT. Se você quer que o parâmetro implícito SENDER seja usado também, você deve listá-lo na interface. Este parâmetro permite que um manipulador de evento de instancia acesse o gatilho, por exemplo, para permitir que retorne resultados.

Declarar um método de manipulação de evento em uma classe significa que instancias da classe ou a própria classe são, em princípio, capazes de manipular um evento <evt> disparado por um método.

### **Registrando Métodos de Manipulação de Eventos**

Para permitir que um método de manipulação de eventos reaja a um evento, você deve determinar em tempo de execução o gatilho ao qual ele deve reagir. Você pode usar o seguinte comando:

SET HANDLER... <h<sub>i</sub>>... [FOR]...

Ele conecta uma lista de métodos de manipulação com métodos de disparo correspondentes. Há quatro métodos diferentes de eventos.

Podem ser:

- Um evento de instancia declarado em uma classe
- Um evento de instancia declarado em uma interface
- Um evento estático declarado em uma classe
- Um evento estático declarado em uma interface

A sintaxe e efeito do SET HANDLER depende de qual dos quatro casos se aplica.

Para um evento de instancia, você deve usar a adição FOR para especificar a instancia para a qual você quer registrar o handler. Você pode tanto especificar uma única instancia como o gatilho, usando uma variável referencial <ref>:

SET HANDLER... <h<sub>i</sub>>...FOR <ref>.

Ou você pode registrar o handler para todas as instancia que podem disparar o evento:

SET HANDLER... <h<sub>i</sub>>...FOR ALL INSTANCES.

O registro então se aplica até mesmo a instancia de disparo que ainda não foram criadas quando você registra o handler.

Você não pode usar a adição FOR para eventos estáticos:

SET HANDLER... <h<sub>i</sub>>...

O registro se aplica automaticamente a classe inteira, ou a todas as classes que implementam a interface contendo o evento estático. Em caso de interfaces, o registro também se aplica a classes que não carregadas até depois do handler ter sido registrado.

## **Timing de Manipulação de Eventos**

Após o comando RAISE EVENT, todos os métodos de manipulação registrados são executados antes do próximo comando ser processado (manipulação de evento sincronizado). Se o próprio método de manipulação disparar o evento, seus métodos de manipulação são executados antes do método de manipulação original continuar. Para evitar a possibilidade de recursão sem fim, eventos podem apenas ser subdivididos em uma profundidade de 64.

Métodos de manipulação são executados na ordem em qual são registrados. Já que manipuladores de eventos são registrados dinamicamente, não se deve assumir que todos serão processados em uma ordem particular. Ao invés, deve-se programar como se todos os manipuladores de eventos fossem executados simultaneamente.

## **Gráfico Geral - Eventos**

Suponha que tenhamos duas classes, C1 e C2:

## Event trigger

```

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.

```

## Event handler

```

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.

```

A classe C1 contém um evento E1, que é disparado pelo método M1. Classe C2 contém um método M2, que pode manipular evento E1 da classe C1.

O diagrama a seguir ilustra o registro de handler:

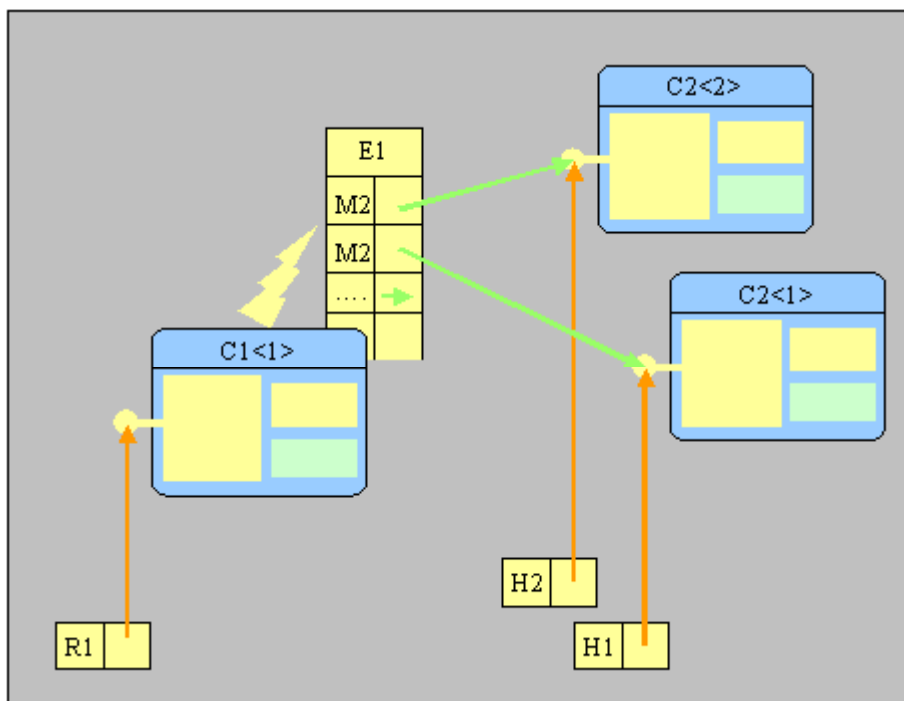
## Registering handlers

```
DATA: R1 TYPE REF TO C1,
      H1 TYPE REF TO C2,
      H2 TYPE REF TO C2.

CREATE OBJECT: R1,
              H1,
              H2.

SET HANDLER H1->M2
              H2->M2 FOR R1.

CALL METHOD R1->M1.
```



O programa cria uma instancia da classe C1 e duas instancias da classe C2. Os valores das variáveis referenciais R1, H1 e H2 indicam estas instancias.

O comando SET HANDLER cria uma tabela de handler, invisível ao usuário, para cada evento que um método de manipulação foi registrado.

A tabela de handlers contém nomes dos métodos de manipulação e **referências** às instancias registradas. As entradas na tabela são administradas dinamicamente pelo comando SET HANDLER. Uma referência para uma instancia em uma tabela de handler é como uma referência

em uma variável referencial. Em outras palavras, conta como um uso da instancia, e portanto diretamente afeta seu tempo de vida. No diagrama acima, isto significa que as instancias C2<1> e C2<2> não são excluídas pela coleta de lixo, mesmo se H1 e H2 são *initial*, enquanto seu registro não é excluído da tabela de handler.

Para eventos estáticos, o sistema cria uma tabela de handler independente de instancia para a classe relevante.

Quando um evento é disparado, o sistema procura na tabela correspondente de eventos e executa os métodos nas instancias apropriadas (ou na classe correspondente para um método de manipulação estático).

## Eventos: Exemplo Introdutório

O exemplo simples a seguir mostra o princípio de eventos dentro de objetos ABAP. É baseado na Introdução Simples a Classes. Um evento `critical_value` é declarado e disparado na classe `counter`.



```
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.
```

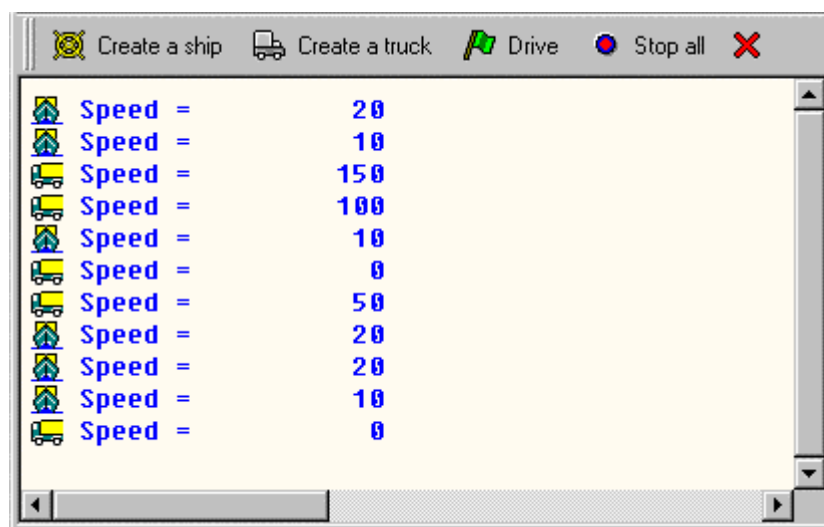
A classe COUNTER implementa um contador. Ele ativa o evento CRITICAL\_VALUE quando um valor limite é excedido, e mostra a diferença. HANDLER pode manipular a exceção em COUNTER. Em tempo de execução, o handler é registrado para todas as variáveis referenciais que apontam para o objeto.

## Eventos em Objetos ABAP - Exemplo

O exemplo a seguir mostra como declarar, chamar, e manipular eventos em objetos ABAP.

### Visão Geral

Este objeto trabalha com a lista interativa mostrada abaixo. Cada interação do usuário dispara um evento em objetos ABAP. A lista juntamente com seus dados é criada na classe C\_LIST. Há uma classe STATUS para processar ações do usuário. Ela dispara um evento chamado BUTTON\_CLICKED no evento AT USER-COMMAND. O evento é manipulado na classe C\_LIST. Ela contém um objeto da classe C\_SHIP ou C\_TRUCK para cada linha da lista. Ambas estas classes implementam a interface I\_VEHICLE. Quando a velocidade de um destes objetos muda, o evento SPEED\_CHANGE é ativado. A classe C\_LIST reage e atualiza a lista.



## Contras



Os comandos ABAP usados para processamento de lista ainda não estão completamente disponíveis em objetos ABAP. No entanto, para produzir uma única saída de teste, pode-se usar os seguintes comandos:

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

Nota: O comportamento das funções de formatação e interação de lista em seu estado atual não é garantido. Mudanças incompatíveis podem ocorrer em um release futuro.

## Declarações

Este exemplo é implementado usando interfaces e classe **locais**. Abaixo estão as declarações das interfaces e classes:

```
*****
* 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.
```

```
*****
```

Os seguintes eventos são declarados no exemplo:

- O evento de instancia SPEED\_CHANGE na interface I\_VEHICLE
- O evento estático BUTTON\_CLICKED na classe STATUS

A classe C\_LIST contém métodos de manipulação de eventos para ambos os eventos.

Note que a classe STATUS não tem qualquer atributo, e, portanto apenas trabalha com métodos estáticos e eventos.

## Implementações

Abaixo estão as implementações dos métodos das classes acima:

```
*****
```

```
* 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.

```

```

*****

```

O método estático USER\_ACTION da classe STATUS dispara o evento estático BUTTON\_CLICKED. Os métodos de instancia I\_VEHICLE~DRIVE e I\_VEHICLE~STOP disparam o evento de instancia I\_VEHICLE~SPEED\_CHANGE nas classes C\_SHIP e C\_TRUCK.

## Usando As Classes em um Programa

O programa a seguir usa as classes acima:

```

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.

\*\*\*\*\*

O programa cria um objeto da classe C\_LIST e registra o método de manipulação de evento FCODE\_HANDLER do objeto para o evento de classe BUTTON\_CLICKED, e o método de manipulação de evento LIST\_CHANGE para o evento SPEED\_CHANGE de todas as instancias que implementam a interface I\_VEHICLE.

## Pool de Classes

Esta seção discute a estrutura e características especiais de pool de classes.

### Classes Globais e Interfaces

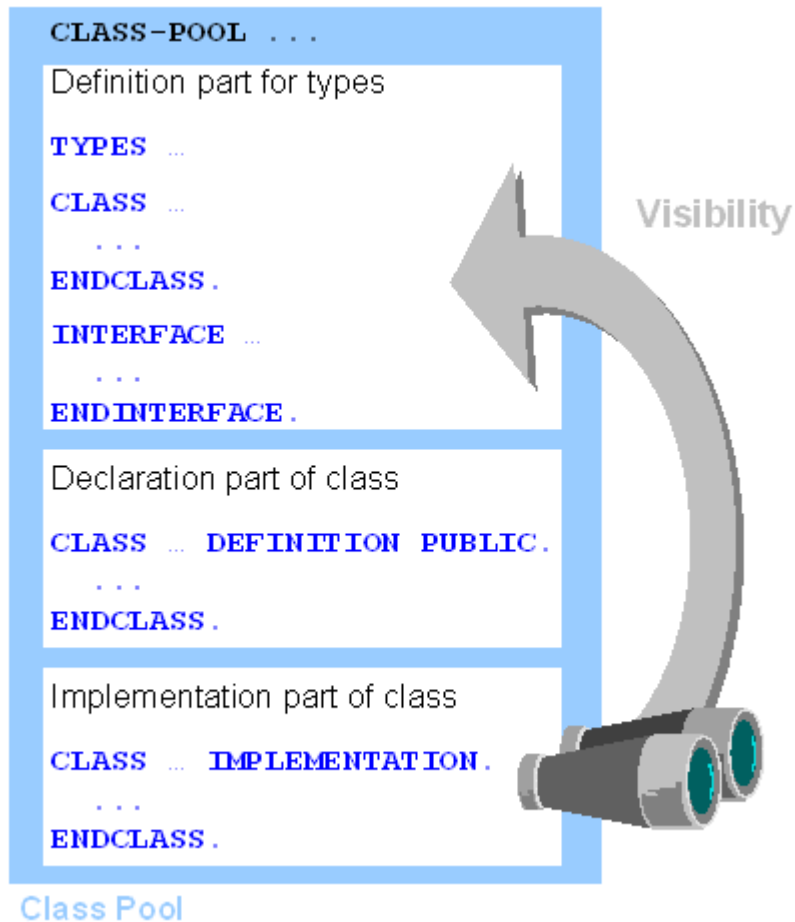
Classes e interfaces são ambos tipos de objeto. Você pode defini-los tanto globalmente no R/3 Repository ou localmente em um programa ABAP. Se você definir classes e interfaces globalmente, elas são guardadas em programas especiais ABAP chamados pool de classes (tipo K) ou pool de interfaces (tipo J), que serve como container para os respectivos tipos de objeto. Cada classe ou pool de interfaces contém a definição de uma única classe ou interface. Os programas são automaticamente gerados pelo Class Builder quando você cria uma classe ou interface.

Um pool de classes é comparável a um pool de módulos ou grupo de funções. Ele contém ambos os comandos declarativos e executáveis ABAP, mas não pode ser iniciado por si próprio. Ao invés, o sistema pode somente executar os comandos dentro do pool de classes processado, isto é, quando o comando CREATE OBJECT ocorrer para criar instancias da classe.

Pool de interfaces não contém qualquer comando executável. Ao invés, eles são usados como container para definições da interface. Quando você implementa uma interface dentro de uma classe, a definição da interface está implicitamente inclusa na definição da classe.

### Estrutura de um Pool de classes

Pool de classes são estruturadas assim:



Pool de classes contém uma parte de definição para declarações de tipos, e as partes de declaração e implementação da classe.

## Diferenças De Outros Programas ABAP

Pool de classes são diferentes de outros programas ABAP pelos seguintes motivos:

- Programas ABAP como programas executáveis, pool de módulos, ou pool de funções geralmente tem uma parte de declaração em qual os dados globais para o programa é definido. Estes dados são visíveis em todo os blocos de processamento do programa. Pool de classes, por outro lado, tem uma parte de definição, na qual você pode definir tipos de dados e objetos, mas **não** objetos de dados ou campos de símbolos. Os tipos que você define em um pool de classes são **somente** visíveis na parte de implementação da classe global.
- Os únicos blocos de processamento que você pode usar são as partes de declaração e implementação da classe global. A parte de implementação pode apenas implementar os métodos declarados na classe global. Você **não pode** usar qualquer um dos outros blocos

de processamento (módulos de diálogo, blocos de eventos, sub-rotinas, módulos de funções).

- Os blocos de processamento do pool de classes não são controlados pelo ambiente de tempo de execução ABAP. Se nenhum evento ocorrer, você não pode chamar qualquer módulo de diálogos ou procedimentos. Pool de classes serve exclusivamente para programação de classes. Você pode apenas acessar os dados e funções de uma classe usando sua **interface**.
- Módulos de diálogo não são permitidos em classes, não se pode processar telas em classes. Não se pode programar listas e telas de seleção em classes, já que estas não reagem aos eventos apropriados. Há a intenção de se fazer telas disponíveis para classes. Ao invés de módulos de diálogo, será possível chamar métodos da classe de um fluxo lógico de telas.

### ***Classes Locais em Pool de Classes***

As classes e interfaces que você define na parte de definição de um pool de classes não estão visíveis externamente. Dentro de pool de classes, eles têm uma função similar a classes locais e interfaces em outros programas ABAP. Classes locais podem apenas ser criados como instâncias para elas em métodos da classe global. Já que sub-rotinas não são permitidas em um pool de classes, classes locais são a única unidade possível de modularização em classes globais. Classes locais tem uma função em relação a classes globais similar a função de sub-rotinas em relação a um grupo de funções, mas com a significativa exceção que não são visíveis externamente.