

**Desenvolvimento de Software com ABL**

**Índice**

**Índice**

[1 10](#_Toc138832388)

[Apresentando o Progress ABL 10](#_Toc138832389)

[E no começo… FOR EACH Customer 11](#_Toc138832390)

[Características básicas do Progress ABL 17](#_Toc138832391)

[Salvando sua procedure de teste 18](#_Toc138832392)

[2 19](#_Toc138832393)

[Usando Construções ABL Básicas 19](#_Toc138832394)

[Control flow 42](#_Toc138832395)

[Expressions and operators 48](#_Toc138832396)

[Built-in functions 48](#_Toc138832397)

[Handle-based and class-based objects 49](#_Toc138832398)

[Code for portability 50](#_Toc138832399)

[3 51](#_Toc138832400)

[Procedures and User-defined Functions 51](#_Toc138832401)

[ABL procedures 51](#_Toc138832402)

[User-defined functions 55](#_Toc138832403)

[4 57](#_Toc138832404)

[Compile and Run 57](#_Toc138832405)

[R-code and the AVM 57](#_Toc138832406)

[COMPILE statement 58](#_Toc138832407)

[RUN statement 63](#_Toc138832408)

[Run using the command line 63](#_Toc138832409)

[5 65](#_Toc138832410)

[Work with the OpenEdge Database 65](#_Toc138832411)

[Tables and buffers 65](#_Toc138832412)

[Data access 66](#_Toc138832413)

[Queries 68](#_Toc138832414)

[Transactions 70](#_Toc138832415)

[Record locks 72](#_Toc138832416)

[Update records 73](#_Toc138832417)

[Online database changes using ABL 74](#_Toc138832418)

[6 75](#_Toc138832419)

[Temp-tables and Datasets 75](#_Toc138832420)

[Temp-tables 75](#_Toc138832421)

[ProDataSets 78](#_Toc138832422)

# 1

## Apresentando o Progress ABL

O Progress® ABL é uma linguagem de programação procedural de alto nível, desenvolvida para permitir que você construa quase todos os aspectos de um aplicativo de negócios corporativo, desde a interface do usuário até o acesso ao banco de dados e a lógica de negócios. Ao longo de décadas, o Progress ABL tornou-se uma ferramenta versátil e extraordinariamente ponderosa.

A linguagem de programação Progress é uma linguagem de quarta geração (4GL) porque possui declarações e palavras-chave poderosas que são especializadas para criar aplicativos de negócios.

Instruções de programação únicas na 4GL podem fazer o trabalho de dezenas ou possivelmente centenas de linhas de código em uma 3GL padrão, como Visual Basic, Java ou C++. Uma única instrução 4GL pode trazer dados do banco de dados do aplicativo para a interface do usuário ou retornar as alterações de um usuário de volta ao banco de dados. Outras instruções permitem que você programe com grande precisão, até mesmo no nível de extração de bits individuais de um fluxo de dados. Essa flexibilidade é o que dá ao Progress 4GL seu grande poder como linguagem de desenvolvimento. A maioria das ferramentas de desenvolvimento que você usará para desenvolver aplicativos OpenEdge™ são escritas em 4GL.

Em seus primeiros lançamentos, no início dos anos 1980, o Progress 4GL permitia aos desenvolvedores criar aplicativos de interface de personagem que rodavam em uma ampla variedade de plataformas de hardware, incluindo muitas variedades de UNIX, DOS e alguns outros sistemas operacionais não mais em uso. Os aplicativos Early Progress eram, desde o início, totalmente portáteis entre plataformas para que um desenvolvedor pudesse simplesmente mover programas aplicativos de um tipo de máquina ou um tipo de terminal de exibição para outro com a confiança de que eles funcionariam corretamente em qualquer lugar.

Com a crescente presença do Microsoft Windows como uma plataforma para interfaces gráficas, o 4GL evoluiu para suportar essas interfaces, com todos os seus vários controles visuais, bem como as construções de programação orientadas a eventos necessárias para um aplicativo orientado a menu e mouse. Hoje, o 4GL continua a crescer, com extensões mais recentes para fornecer uma definição cada vez mais dinâmica de componentes de aplicativos, bem como acesso a tecnologias abertas, como HTML e XML, e uma série de outras construções para oferecer suporte a um ambiente aberto de desenvolvimento e implantação de aplicativos.

E, ao mesmo tempo, os aplicativos baseados em Progress 4GL podem ser transferidos de uma versão para a próxima sem alterações. O progress fornece um grau de compatibilidade e migração ascendente de uma versão para a próxima incomparável por qualquer outra linguagem de programação de alto nível.

#### Sobre o banco de dados de amostra

Este livro usa um dos bancos de dados de amostra OpenEdge padrão para seus exemplos, o banco de dados Sports2000. Este é um exemplo simplificado de um banco de dados que pode ser usado em um aplicativo típico de entrada de pedidos, com clientes, pedidos, linhas de pedidos e outras informações para acompanhar os clientes e seus pedidos. Alguns dos mesmos procedimentos e janelas de exemplo que você criará são usados e estendidos ao longo do livro à medida que você é apresentado a novos conceitos de programação, mas você não criará um aplicativo de amostra completo. Isso ocorre principalmente porque, ao apresentar tantas construções de linguagem e técnicas de programação separadas, o livro oferece uma experiência única que não seria típica de um aplicativo padronizado. Depois de ler o livro, porém, você será capaz de usar o que aprendeu para criar muitos tipos de aplicativos abrangentes.

### E no começo… FOR EACH Customer

Existe um procedimento prototípico do Progress 4GL frequentemente usado como exemplo, que não poderia ser mais simples, mas que mostra muito sobre o poder da linguagem. Esta frase apareceu em canecas e camisetas por duas décadas. Aqui está:

FOR EACH Customer:

DISPLAY Customer.

END.

Uma procedure não poderia ser mais simples quanto esta, mas levaria horas para explicar em detalhes tudo o que esse pequeno código progress faz por você. Resumindo:

* **Customer** é o nome de uma tabela no banco de dados de amostra Sports2000 à qual você se conectará daqui a pouco. A instrução **FOR EACH** inicia um bloco de código que abre uma consulta nessa tabela de banco de dados e retorna cada registro da tabela, um por vez, em cada iteração do bloco.
* Cada registro da tabela **Customer** é exibido na tela. O código pega as informações de formatação e rótulo da definição do esquema do banco de dados e as usa para criar um formato padrão de exibição para todos os campos na tabela. **DISPLAY Customer** significa exibir todos os campos da tabela de clientes.
* À medida que cada registro é exibido, o visor move uma linha para baixo para exibir o próximo Cliente (**Customer**). O efeito é mais parecido com o que você veria em um relatório do que em uma navegação ou outro controle de grade.
* O bloco de código — tudo, desde a instrução FOR EACH até o END — itera uma vez para cada registro de Cliente (daí a sintaxe FOR EACH). Todo o código do meio (neste caso, apenas uma instrução DISPLAY) é executado para cada cliente retornado.
* Quando a exibição chega ao final da área de exibição disponível, ela pausa automaticamente, com uma mensagem solicitando que você pressione a barra de espaço para ver o próximo conjunto de linhas.
* Quando você pressiona a barra de espaço, a exibição é limpa e um novo conjunto de linhas aparece.
* Quando o Progress detecta o final do conjunto de registros (todos os registros do Cliente neste caso), ele encerra o procedimento com a mensagem “Procedimento concluído. Pressione a barra de espaço para continuar.”
* Se você quiser encerrar a consulta antes de chegar ao fim dos registros, você pode pressionar a tecla ESC a qualquer momento para encerrar o procedimento.

#### Iniciando sua sessão OpenEdge

Mas então, onde você digita esse pequeno pedaço de código? Primeiro, você precisa iniciar uma sessão do OpenEdge, criar e conectar-se a uma cópia do banco de dados de amostra que o exemplo usa e, em seguida, abrir o Procedure Editor.

**Para iniciar sua sessão OpenEdge:**

1. No menu Iniciar do Windows, procure pela pasta Progress e selecione a opção Desktop. A janela da área de trabalho do Ambiente de Desenvolvimento de Aplicativos é exibida:

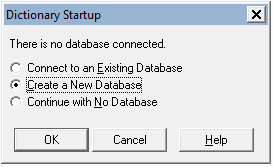


A partir daqui você pode acessar todas as ferramentas básicas de desenvolvimento do OpenEdge.

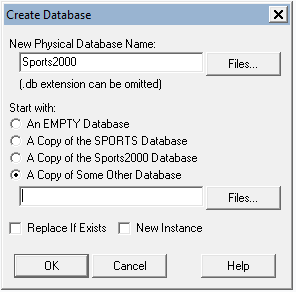
1. Escolha o primeiro ícone, Data Dictionary.

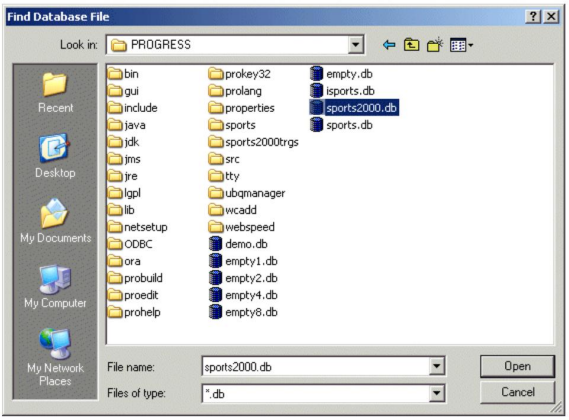
A caixa de diálogo Data Dictionary é aberta. O Dicionário de dados é onde você define e gerencia todas as tabelas e campos (colunas) do banco de dados do seu aplicativo. Para começar, você pode criar sua própria cópia do banco de dados Sports2000, que é o banco de dados de demonstração OpenEdge padrão. Você precisa copiar o banco de dados para que seus procedimentos de teste possam fazer alterações nele sem modificar a versão dele em seu diretório de instalação.

1. Na caixa de diálogo que se abriu, selecione a opção para criar um novo banco de dados e, em seguida, escolha OK:

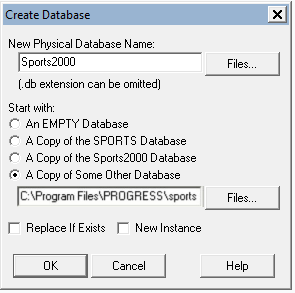


A caixa de diálogo Criar banco de dados aparece e solicita o nome da sua cópia do banco de dados.

1. Digite Sports2000. Por padrão, o OpenEdge cria um banco de dados denominado Sports2000 em seu diretório de trabalho. Se quiser que o banco de dados esteja em outro lugar, você pode escolher o botão **Arquivos** ao lado do campo de preenchimento **Novo Nome do Banco de Dados Físico** para procurar na estrutura de diretórios o local certo para seu banco de dados.
2. Depois de inserir o novo nome do banco de dados, selecione a opção Iniciar com uma cópia de algum outro banco de dados e escolha o botão Arquivos ao lado do campo de preenchimento: 
3. Localize o banco de dados Sports2000 no diretório de instalação do OpenEdge e clique duas vezes nele:



O nome do caminho para o banco de dados é preenchido quando você retorna à caixa de diálogo Create Database:



1. Escolha **OK** para aceitar este nome de caminho. A caixa de diálogo **Conectar banco de dados** é exibida.
2. Certifique-se de que o campo **Nome Físico** mostre Sports2000 e o **Tipo de Banco de Dados** seja Progress e escolha **OK**.

Como você criou esse banco de dados como parte da inicialização do Data Dictionary, que precisa ter um banco de dados conectado antes de permitir sua entrada, a janela principal do Data Dictionary agora é aberta. Você pode trabalhar com o Data Dictionary posteriormente, mas apenas para se familiarizar com as tabelas de banco de dados que usará neste capítulo, vá em frente e dê uma olhada nelas aqui.

**Para se familiarizar com as tabelas do banco de dados Sports2000:**

* 1. Selecione a tabela Cliente na lista Tabelas e escolha o botão **Fields ** na barra de ferramentas do dicionário. Todos os campos (ou colunas, para usar a terminologia SQL equivalente) na tabela são mostrados na lista Fields à direita.
  2. Percorra a lista para ver todos os campos da tabela. Você exibirá alguns desses campos posteriormente e usará outros para selecionar um subconjunto de Clientes para exibição.
  3. Percorra a lista de Tabelas e selecione a tabela Order. Você pode ver que, além de um campo OrderNum , que dá a cada pedido um número exclusivo, há também um campo CustNum, que você usará no Capítulo 2, “Usando construções ABL básicas”, para vincular ou unir as tabelas ORDER para o registro de Customer com o mesmo valor CustNum. Há muito mais para ver no Dicionário. Se você quiser passear pelos displays, vá em frente, mas não mude nada! Quaisquer alterações feitas podem afetar as etapas posteriores neste tutorial. Se você fizer alterações em tabelas ou campos, isso pode invalidar os registros que já estão no banco de dados de amostra ou impedir que alguns dos exemplos de procedures que faremos posteriores funcionem.
  4. Para sair do Dicionário de dados, selecione **Database** > **Exit** no menu **Dictionary**.

Você se mantem conectado ao banco de dados até você voltar ao dicionario de dados e desconectar ou até encerrar a sessão OpenEdge.

##### Escrevendo sua primeira procedure

Sua primeira procedure Progress será algo muito simples apenas para você começar a usar a linguagem. Mas antes de entrar nesta primeira procedure, você precisa fazer um ajuste no código - sua primeira alteração!

Há um bom número de campos na tabela Cliente, portanto, a formatação resultante de todos os campos em uma área de exibição limitada seria uma bagunça. Na verdade, um dos campos, o campo Comentários, é tão grande que o mecanismo de tempo de execução do Progress (também conhecido como interpretador) hesitará em exibi-lo sem alguma orientação sua sobre como formatá-lo e posicioná-lo.Portanto, é melhor selecionar apenas alguns campos da tabela para exibir, o que é mais gerenciável. Selecione o campo CustNum , que é o número exclusivo do cliente, o campo Name , que é o nome do cliente e o campo City, que faz parte do endereço do cliente.

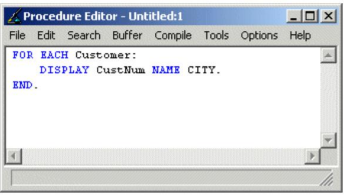
No Progress ABL, uma lista de campos é separada por espaços, então a nova procedure fica:

FOR EACH Customer:

DISPLAY CustNum Name City.

END.

Para inserir este procedimento no Procedure Editor retorne ao OpenEdge Desktop e escolha o segundo ícone. Na janela que aparece, digite seu bloco de código:



Você pode notar o seguinte sobre este código:

* As palavras **FOR** e **EACH** são automaticamente maiúsculas à medida que você as digita e coloridas de azul também. Isso ocorre porque o Procedure Editor usa um controle de edição inteligente que reconhece a sintaxe Progress ABL. Ele não apenas codifica por cores, mas também pode fazer preenchimento automático de sintaxe comum e pode aceitar abreviações de declarações de linguagem comum, para ajudar a acelerar sua programação. O editor também reconhece a instrução **FOR EACH** como sendo o início de um bloco de código e indenta automaticamente a instrução **DISPLAY** e, em seguida, remove a indentação da instrução **END** para corresponder ao início do bloco.
* Você pode digitar o nome da tabela Customer com C maiúsculo, ou digitá-lo em minúsculas, ou tudo em maiúsculas, ou como desejar. Progress não faz distinção entre maiúsculas e minúsculas quando se trata de palavras-chave da linguagem, tabelas de banco de dados e nomes de campo. Colocar as palavras-chave em letras maiúsculas é apenas uma convenção para ajudar na legibilidade do código, mas você pode digitar for each ou For Each também e funcionaria da mesma forma.
* Digitar CustNum em maiúsculas e minúsculas é apenas uma convenção para facilitar a leitura.
* Quando você digita o nome do campo **Name**, ele é maiúsculo e colorido de azul. Isso ocorre porque NAME também é uma palavra-chave no ABL. O Progress é relativamente indulgente desta forma: se o compilador Progress, que processa o código que você digita, puder distinguir entre uma palavra que é um nome de campo ou tabela e uma que deve ser uma palavra-chave, ele permitirá que você a use dessa maneira. No entanto, é uma boa ideia evitar usar palavras-chave como nomes de tabela ou campo na definição do banco de dados. Para obter uma lista de todas as palavras-chave do Progress ABL, consulte a documentação.

Para executar seu procedimento, pressione a tecla F2 em seu teclado. Este é o atalho de teclado para o comando RUN. Como alternativa, você pode selecionar o menu **Compile** > **Execute**.

O Editor mostra o primeiro conjunto de números de clientes, nomes e cidades:



Para ver as próximas páginas de números de Clientes, nomes e cidades, pressione a BARRA DE ESPAÇO algumas vezes. Pressione ESC para encerrar a exibição. Caso contrário, a exibição termina após a exibição da última página de **Customers**.

Este pequeno procedimento de três linhas certamente faz muito por você, mas não produz uma exibição de saída que você colocaria diretamente em um aplicativo. Existem muitos qualificadores para a declaração da palavra-chave **DISPLAY** que você pode usar para ajustar seu comportamento e aparência, alguns dos quais você verá mais adiante neste capítulo. Fundamentalmente, a instrução **DISPLAY** não se destina a produzir uma saída polida por si só para um aplicativo gráfico. Existem outras instruções na linguagem que você pode usar para definir as janelas, navegadores, botões da barra de ferramentas e outros recursos de uma interface de usuário completa. Você os conhecerá em capítulos posteriores.

Para fins de introdução à linguagem, deixe o formato de exibição como está, para que você possa ver como a estrutura da própria linguagem é montada. Você completará um pequeno procedimento de teste aqui que mostra muito sobre o Progress ABL, mas que não representa a maneira como você escreverá aplicativos realmente acabados. Para fazer isso, você trabalha junto com as ferramentas de desenvolvimento para permitir que as ferramentas façam grande parte do trabalho de montar a interface do usuário para você. Depois de apenas alguns capítulos, você estará construindo janelas de aplicativos completas e toda a lógica por trás delas.

### Características básicas do Progress ABL

Você acabou de escrever uma procedure inicial muito simples. O que você já pode aprender com ela?

#### O ABL é procedural

Progress é uma linguagem de programação procedural. Isso significa que você escreve conjuntos de instruções de código que podem ser salvas individualmente, chamados **Procedures.** As instruções de código geralmente são executadas ou processadas na ordem em que aparecem no procedimento. Em um procedimento simples como este, as instruções são executadas conforme aparecem. À medida que avança na construção de aplicativos orientados a eventos, onde o usuário tem várias maneiras de controlar o aplicativo, escolhendo botões ou fazendo seleções de menu, você aprenderá como definir o código de gatilho (**trigger**) que configura blocos de instruções a serem executadas posteriormente quando uma determinada ação ocorre.

#### O 4GL é estruturado em bloco

Uma procedure Progress ABL é composta de blocos. A procedure em si é o bloco principal da Procedure. Existem várias maneiras de definir outros blocos dentro do Main Block (bloco principal). A instrução **FOR EACH** e sua instrução **END** correspondente são um exemplo de um bloco aninhado, neste caso, um que itera por meio de um conjunto de registros do banco de dados e executa todo o código intermediário para cada registro no conjunto. Existem outras instruções de bloco que você pode usar para diferentes propósitos. Alguns deles também são de iteração e fazem com que o bloco seja executado várias vezes. Outros simplesmente definem um conjunto de instruções a serem executadas juntas. Você aprenderá sobre tudo isso em capítulos posteriores.

#### Uma procedure ABL consiste em INSTRUÇÕES (STATEMENTS)

Um procedimento Progress ABL é composto de uma sequência de instruções de linguagem. Cada instrução tem uma ou mais palavras-chave ABL, juntamente com outros tokens, como nomes de campos de banco de dados ou nomes de variáveis. Uma instrução ABL normalmente termina com um ponto. Por convenção, uma instrução que define o início de um bloco, como a instrução FOR EACH Customer, pode terminar com dois pontos. Não há significado para o fim da linha em Progress. Uma única instrução pode abranger várias linhas e, se desejar, pode haver várias instruções em uma única linha. Não há nenhuma sintaxe especial necessária para quebrar uma instrução entre as linhas, mas se você tiver um único elemento de uma instrução, como uma string entre aspas, que seja longa o suficiente para que você precise abranger linhas, termine a primeira linha com um caractere til (~).

Deve haver pelo menos um espaço (ou outro caractere de espaço em branco, como tabulação ou nova linha) entre cada token na instrução. O progress não é sensível a espaço em branco adicional. Você pode recuar linhas de código e colocar tabulações entre partes de uma instrução para facilitar a leitura, e nada disso afetará a execução da instrução.

O progress não diferencia maiúsculas de minúsculas. Isso significa que você pode escrever uma palavra-chave ABL ou uma referência a um nome de variável ou tabela de banco de dados ou nome de campo em qualquer combinação de letras maiúsculas e minúsculas. Como você já viu, o controle de edição inteligente do Progress é padronizado para uma convenção de palavras-chave ABL em letras maiúsculas. Para obter uma lista de todas as palavras-chave do Progress, você pode consultar o índice no submenu **Help Topics** do menu **Help** (por meio de qualquer uma das ferramentas de desenvolvimento do OpenEdge).

Ao definir variáveis em um procedimento e ao definir tabelas e campos do banco de dados, você deve dar a eles nomes que comecem com uma letra. Depois disso, o nome pode incluir letras, dígitos numéricos, pontos (.), hífens (-) e sublinhados (\_). Ao salvar seu procedimento ABL, você também dá um nome a ele, e isso segue as mesmas regras de nomenclatura. Observe que, embora o próprio ABL tenha muitas palavras-chave hifenizadas, você não deve usar hífens em nomes de campos e tabelas de banco de dados, bem como nomes de procedimentos e variáveis em seu aplicativo. Você deve seguir esta diretriz porque outras linguagens de programação que você pode querer combinar com procedimentos ABL geralmente não permitem um hífen como caractere em um nome de procedimento ou parâmetro. E a linguagem de acesso ao banco de dados SQL, que você também pode usar para gerenciar dados em seu banco de dados OpenEdge usando outras ferramentas, não permite hífens em nomes de tabelas e campos.

Uma boa convenção de nomenclatura para todos os tipos de nomes em procedimentos ABL é usar maiúsculas e minúsculas (como no campo CustNum ) para dividir um nome em várias palavras ou outras partes identificáveis, usando letras maiúsculas para identificar o início de cada palavra ou subparte dentro do nome para melhorar a legibilidade do seu código.

#### O ABL combina instruções procedurais, de banco de dados e de interface do usuário

Existem três tipos básicos de instruções em um programa ABL: instruções procedurais, instruções de acesso ao banco de dados e instruções de interface com o usuário. Às vezes, declarações individuais contêm elementos de todos os três. Seu primeiro procedimento simples contém todos os três tipos e ilustra o poder da linguagem.

A própria instrução **FOR EACH** pode ser considerada procedural, pois define uma ação dentro do programa, neste caso, repetindo um bloco de instruções até a instrução **END**. Mas a instrução **FOR EACH** também é uma instrução de acesso ao banco de dados, porque define um conjunto de resultados que o bloco **FOR** **EACH** irá iterar, neste caso, o conjunto de todos os registros do Cliente no banco de dados.

Essa convenção simples de combinar lógica processual com acesso ao banco de dados é um recurso fundamental e extremamente poderoso do Progress ABL. Em outra linguagem, você teria que definir uma consulta de banco de dados usando uma sintaxe que é basicamente distinta da linguagem de programação, em seguida, ter instruções adicionais para controlar o fluxo de dados em seu procedimento. No ABL, você pode combinar todos esses elementos de uma forma muito natural, flexível e eficiente, e se relaciona com a maneira como você pensa sobre como o programa usa os dados.

O bloco **FOR EACH** também é poderoso porque apresenta de forma transparente um registro por vez ao seu programa, de forma que o restante da lógica procedural possa agir sobre ele. Se você escreveu aplicativos que usam a linguagem **SQL** para recuperação de dados, pode comparar o bloco Progress **FOR EACH** com um programa em outra linguagem contendo instruções **SQL** incorporadas, em que a natureza da instrução **SQL SELECT** não é uma boa correspondência à maneira “registro-a-registro” em que seu programa normalmente deseja interagir com os dados.

A instrução **DISPLAY** mostra que as instruções da interface do usuário também estão estreitamente integradas ao restante do programa. O Progress ABL contém instruções de linguagem não apenas para exibir dados, mas também para criar, atualizar e excluir registros e para atribuir valores individuais. Todas essas declarações são totalmente integradas à linguagem. Em capítulos posteriores, você aprenderá como construir seus aplicativos de forma que os procedimentos que controlam a interface do usuário sejam claramente separados dos procedimentos que gerenciam os dados e definem sua lógica de negócios.

### Salvando sua procedure de teste

Antes de prosseguir para ampliar sua primeira procedure e aprender alguns dos outros fundamentos da linguagem ABL, você deve salvá-la no sistema operacional. Para fazer isso, dê um nome ao seu código de procedure. Isso segue as mesmas regras de outros nomes e, por convenção, tem uma extensão de nome de arquivo .p (para procedures).

Para salvar seu procedimento no Procedure Editor, selecione **Arquivo** > **Salvar** **como**. Nomeie o procedimento h-CustSample.p e salve o arquivo em seu diretório de trabalho ou em algum outro diretório que você criou. Agora você está pronto para estender sua procedure de várias maneiras.

# 2

## Usando Construções ABL Básicas

No Capítulo 1, “Apresentando o Progress ABL”, você aprendeu algumas das características básicas do Progress 4GL e criou um pequeno procedimento de teste. Neste capítulo, você estenderá seu procedimento de teste:

#### Refinando a seleção de dados com uma cláusula WHERE

Até agora você selecionou todos os registros de Clientes no banco de dados. Agora você refinará essa seleção para mostrar apenas os clientes do estado de New Hampshire. Há um campo State na tabela Customer que contém a abreviação de estado de duas letras para estados nos EUA.

O Progress ABL oferece suporte a uma cláusula **WHERE** em qualquer instrução que recupere dados do banco de dados, o que lhe será familiar se você tiver usado SQL ou outras linguagens de acesso a dados semelhantes. A palavra-chave **WHERE** pode ser seguida por qualquer expressão que identifique um subconjunto dos dados. Você aprenderá muito mais sobre isso em capítulos posteriores, mas, por enquanto, uma expressão simples é tudo de que você precisa.

Para refinar a seleção de dados em seu procedimento de teste:

1. Adicione a seguinte cláusula **WHERE** ao final de sua instrução **FOR EACH** :

FOR EACH Customer WHERE Customer.State = “NH”:

1. Pressione F2 para ver a lista reduzida de clientes. A lista agora deve usar apenas um pouco mais do que uma página:
2. Adicione uma expressão de classificação ao final de sua cláusula WHERE para classificar os clientes em ordem por sua cidade. O 4GL usa a palavra-chave BY para indicar uma sequência de classificação:

FOR EACH Customer WHERE Customer.State = “NH” BY Customer.City:

1. Pressione F2 para executar o procedimento novamente para ver os efeitos de sua alteração.

##### Operadores de comparação

O sinal de igual é apenas um dos vários operadores de comparação que você pode usar em expressões ABL. A Tabela 2–1 fornece uma lista completa.

|  |  |  |
| --- | --- | --- |
| **Palavra-Chave** | **Simbolo** | **Explicação** |
| EQ | = | Igual a |
| NE | <> | Diferente |
| GT | > | Maior que |
| LT | < | Menor que |
| GE | >= | Maior ou igual a |
| LE | <= | Menor ou igual a |
| BEGINS |  | Um valor de caractere que começa com esta substring |
| MATCHES |  | Um valor de caractere que corresponda a esta substring, que pode incluir caracteres curinga.  A expressão que você usa à direita de MATCHES pode conter os caracteres curinga:  • Um asterisco (\*) representa um ou mais caracteres ausentes.  • Um ponto (.) representa exatamente um caractere ausente. |
| CONTAINS |  | Um campo de texto do banco de dados que possui um tipo especial de índice chamado WORD-INDEX.  O WORD-INDEX indexa todas as palavras nas strings de texto de um campo, para todos os registros da tabela, permitindo que você localize palavras individuais ou palavras associadas nos registros do banco de dados, da mesma forma que você faz quando usa um mecanismo de pesquisa na Internet para localizar texto em documentos na web. |

##### Usando aspas

Você pode usar aspas simples (') ou aspas duplas (“) para definir uma constante de string. Você deve usá-las combinando os pares adequadamente, usando o mesmo tipo de aspas no início e no final da string. Use aspas duplas se sua string contiver uma aspa simples e vice-versa, como no exemplo a seguir:

DISPLAY “Sempre usar o mesmo tipo de aspas!”.

DISPLAY ‘Assim.’.

DISPLAY “Mas nunca assim!’.

#### Criando blocos aninhados para exibir dados relacionados

Para revisar, a instrução FOR EACH em seu procedimento cria um bloco de código aninhado dentro do bloco principal implícito do próprio procedimento. Agora você criará um segundo bloco FOR EACH aninhado dentro do primeiro FOR EACH, para exibir os registros da tabela Orders no banco de dados para cada Customer de New Hampshire.

Para criar um bloco aninhado, adicione outro bloco FOR EACH dentro do primeiro, então seu procedimento fica assim:

FOR EACH Customer NO-LOCK WHERE Customer.State = “NH” BY Customer.City:

DISPLAY Customer.CustNum Customer.Name Customer.City.

FOR EACH Order OF Customer NO-LOCK:

DISPLAY Order.OrderNum Order.OrderDate Order.ShipDate.

END.

END.

Este exemplo mostra o código identado para que o novo bloco seja visualmente aninhado no bloco externo, o que ajuda na legibilidade do código.

Primeiro, observe a nova instrução FOR EACH. A palavra-chave OF é um atalho para uma cláusula WHERE que une as duas tabelas. Quando você olhou para as duas tabelas e seus campos no Dicionário, viu que ambas as tabelas têm um campo **CustNum**. Esta é a chave primária da tabela **Customer**, o que significa que cada **Customer** recebe um número único para o campo **CustNum**, e este é o identificador primário para o **Customer**. Na tabela **Order**, o **OrderNum** é o identificador exclusivo do **Order** e sua chave primária. O campo **CustNum** na tabela de Pedidos (**Order**) aponta para o Cliente (**Customer**) ao qual o Pedido se destina. É uma chave estrangeira porque aponta para um registro em outra tabela. Para recuperar e exibir os Pedidos de um Cliente, você deve unir as duas tabelas através do campo **CustNum** que elas têm em comum. A cláusula WHERE completa para esta junção seria: WHERE Customer.CustNum = Order.CustNum. Esse tipo de sintaxe será familiar para você se já trabalhou com SQL.

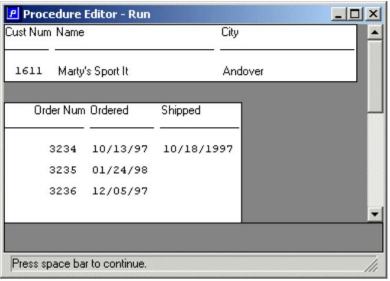
A cláusula WHERE está dizendo ao Progress para selecionar os registros em que o campo **CustNum** em uma tabela corresponde ao campo **CustNum** na outra. Para informar ao Progress qual campo é qual, ambos são qualificados pelo nome da tabela, seguido de um ponto.

No Progress ABL, você pode usar a sintaxe Order OF Customer como um atalho para essa junção se as duas tabelas tiverem um ou mais campos com nomes semelhantes em comum que constituem o relacionamento de junção e esses campos forem indexados em pelo menos uma das tabelas (normalmente devem ser indexadas em ambas). Você sempre pode usar a sintaxe completa da cláusula WHERE em vez da frase OF, se desejar; o efeito é o mesmo e, se houver alguma dúvida sobre como as tabelas estão relacionadas, isso torna o relacionamento que seu código está usando completamente claro. Na verdade, a frase OF é realmente um daqueles atalhos para iniciantes que torna fácil escrever um procedimento muito simples, mas que realmente não é uma boa prática em um aplicativo complexo, porque não fica claro apenas lendo a instrução quais campos estão sendo usados para relacionar as duas tabelas. Você geralmente deve ser explícito sobre seus relacionamentos de campo em suas aplicações.

Essas simples instruções FOR EACH aninhadas realizam algo que daria muito trabalho em outras linguagens. Para recuperar os clientes e seus pedidos separadamente, como se deseja, você teria que definir duas consultas separadas usando a sintaxe SQL incorporada, abri-las e controlá-las explicitamente em seu código. Isso daria muito trabalho. Por exemplo, a consulta SQL simples e direta para recuperar os mesmos dados seria:

SELECT Customer.CustNum, Name, City, OrderNum, OrderDate, ShipDate FROM Customer, Order WHERE Customer.State = “NH” AND Customer.CustNum = Order.CustNum;

Esse código recuperaria todos os clientes e pedidos relacionados em uma única tabela bidimensional, o que não é muito útil: todas as informações do cliente seriam repetidas para cada um dos pedidos do cliente e você mesmo teria que separá-las para exibir as informações bem como cabeçalho e detalhes, que provavelmente é o que você deseja. Por outro lado, quando você executa seu procedimento Progress ABL muito simples, obtém uma exibição padrão que representa os dados adequadamente como informações principais (cliente) e detalhadas (pedido), conforme mostrado na seguinte figura:



O Progress fornece automaticamente duas áreas de exibição separadas, uma para os campos do cliente mostrando um cliente por vez e outra para os pedidos do cliente. Essas áreas de exibição são chamadas de *frames*. Você aprenderá mais sobre os frames do progress em capítulos posteriores.

Ao contrário do primeiro exemplo, que exibe uma página inteira de Clientes, cada vez que você pressiona a BARRA DE ESPAÇO, o Progress exibe apenas o próximo Cliente e seus Pedidos. Por que o Progress fez isso? Os blocos FOR EACH aninhados informam ao Progress que há vários pedidos a serem exibidos para cada cliente, portanto, ele sabe que não faz sentido exibir mais de um cliente por vez. Então, ele cria um pequeno quadro com tamanho suficiente para exibir os campos de um cliente e, em seguida, cria outro quadro separado onde pode exibir vários pedidos. O último quadro é chamado de quadro inferior (*down frame)*, porque pode exibir várias linhas de dados à medida que se move para baixo na página. Você pode controlar o tamanho dos quadros, quantas linhas são exibidas e muitos outros atributos, anexando um WITH à sua instrução.

Para ver como a frase WITH pode afetar sua procedure de teste:

1. Adicione as palavras WITH CENTERED à instrução DISPLAY para o frame **Order**:

FOR EACH Customer WHERE State = “NH” BY City:

DISPLAY CustNum Name City.

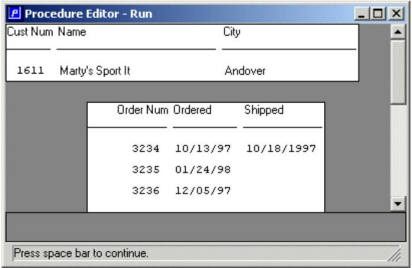
FOR EACH Order OF Customer:

DISPLAY OrderNum OrderDate ShipDate WITH CENTERED.

END.

END.

1. Execute a procedure novamente. O frame **Order** é centralizado na janela de exibição padrão:



Existem muitos atributos do frame que você pode especificar aqui (para obter uma descrição deles, leia a documentação ABL). Este livro não fala muito mais sobre eles, porque você não usará a maioria deles em suas aplicações GUI. Esses atributos são projetados para ajudá-lo a definir frames para uma aplicação no modo de caractere, onde a interface é basicamente apenas uma série de 24 ou 25 linhas de 80 caracteres cada. Para esse tipo de formato de exibição, uma sequência de quadros exibidos um abaixo do outro é uma maneira apropriada e conveniente de dispor a tela. Mas em uma aplicação GUI, em vez disso, você configura sua exibição usando uma ferramenta de design visual, como o OpenEdge AppBuilder, e gera o código ou os dados necessários para criar a interface do usuário em tempo de execução. Portanto, este capítulo mostra os fundamentos de como o ABL funciona e como ele foi projetado, mesmo que você faça a maior parte do seu trabalho de uma maneira diferente em seus novos aplicativos.

#### Alterando labels e formats

A label padrão para o campo **Order Number** é **Order Num** (você define os rótulos padrão no Data Dictionary ao configurar seu banco de dados) e talvez prefira que seja apenas **Order**. Além disso, o formato de exibição padrão do campo **ShipDate** é diferente do formato do campo **OrderDate**: um tem um ano de quatro dígitos e o outro um ano de dois dígitos. Você pode alterar rótulos (labels) e formatos (formats) padrão em sua instrução DISPLAY . Se você adicionar a palavra-chave LABEL ou o FORMAT após o nome do campo, seguido por uma string com o valor de sua preferência, então a exibição muda de acordo com o que você definiu.

Para fazer essas alterações em sua procedure, altere **OrderNum** LABEL para **Order** e **ShipDate** FORMAT para **99/99/99**.

No exemplo a seguir, cada campo tem sua própria linha no bloco de código, para facilitar a leitura do código e enfatizar que isso não altera o funcionamento da procedure. Tudo até o ponto final é uma instrução ABL:

Para visualizer o resultado destas mudanças, execute a procedure novamente:

FOR EACH Customer WHERE State = “NH” BY City:

DISPLAY CustNum Name City.

FOR EACH Order OF Customer:

DISPLAY OrderNum **LABEL** “**ORDER**”

OrderDate

ShipDate **FORMAT** “**99/99/99**” WITH CENTERED.

END.

END.



#### Usando variáveis e tipos de dados

Como a maioria das linguagens de programação, o Progress ABL permite definir variáveis para uso em uma procedure. Aqui está a sintaxe básica:

DEFINE VARIABLE varname AS datatype.

Nesta sintaxe, varname é o nome da variável, que deve seguir as mesmas regras dos outros nomes no ABL. (Consulte “Convenções de nomenclatura de variáveis” para obter detalhes.)

O Progress oferece suporte a uma variedade de tipos de dados. A Tabela a seguir lista os básicos. Existem alguns outros tipos de dados especiais disponíveis para programação mais avançada, mas estes são suficientes para você começar:

|  |  |  |  |
| --- | --- | --- | --- |
| **Nome do Tipo de dado** | **Formato de exibição padrão** | | **Valor inicial Padrão** |
| CHARACTER | X(8) | | “” (string vazia) |
| DATE |  | 99/99/99 | ? (unknown) |
| DECIMAL |  | ->>,>>9.99 | 0 |
| HANDLE |  | >>>>>>9 | ? (unknown) |
| INTEGER |  | ->,>>>,>>9 | 0 |
| LOGICAL |  | yes/no | não |

Aqui estão algumas considerações sobre os tipos de dados Progress:

* Todos os dados CHARACTER em variáveis Progress e em campos de banco de dados OpenEdge são armazenados como strings de comprimento variável. Não é necessário definir o comprimento de uma variável, apenas seu formato de exibição, que indica quantos caracteres (no máximo) são exibidos ao usuário. O comprimento de exibição padrão para campos CHARACTER é de 8 caracteres. O símbolo X representa qualquer caractere imprimível e o 8 entre parênteses efetivamente repete esse caractere, de modo que X(8) é o mesmo que XXXXXXXX. O valor padrão para uma variável CHARACTER é a string vazia.
* Você pode exibir datas de várias maneiras, incluindo anos de dois ou quatro dígitos. Para obter um ano de quatro dígitos, especifique 99/99/9999 como seu formato. Observe que alterar o formato de exibição de uma data não altera como ela é armazenada no banco de dados. As datas são armazenadas em um formato interno que pode ser convertido para qualquer um dos formatos de exibição disponíveis para você.
* O tipo de dados DECIMAL suporta um total de 50 dígitos, dos quais até 10 podem estar à direita do ponto decimal. Ao definir uma variável DECIMAL, você pode especificar o número de posições à direita do ponto decimal adicionando o qualificador DECIMALS <n> à instrução DEFINE VARIABLE.
* O Progress oferece suporte à conversão automática de DATAs e DECIMALs para os formatos geralmente usados na Europa e em outras partes do mundo fora dos EUA. Se você usar a opção de inicialização Europeia (–E), todos os caracteres de ponto ou decimal em formatos DECIMAL serão convertidos em vírgulas e as vírgulas serão convertidas em pontos. Além disso, a exibição padrão DATE se torna DD/MM/AA ou DD/MM/AAAA em vez de MM/DD/AA ou MM/DD/AAAA.
* O tipo de dados HANDLE é usado para armazenar um ponteiro para uma estrutura que representa uma procedure Progress em execução ou um objeto em uma procedure, como um campo ou botão. Em capítulos posteriors você terá mais informações sobre como usar variáveis HANDLE.
* O tamanho máximo de uma variável INTEGER é 2G (pouco mais de um bilhão de dígitos).
* Uma variável LOGICAL representa um valor YES/NO ou TRUE/FALSE. Você pode especificar qualquer par de literais que desejar para os valores TRUE e FALSE que são exibidos ao usuário para uma variável LOGICAL, com o valor TRUE ou YES primeiro. No entanto, não é aconselhável usar LOGICALs para representar valores de dados que não são realmente lógicos por natureza, mas que simplesmente têm dois valores válidos, como Masculino/Feminino, porque pode não estar claro qual deles o valor TRUE representa.
* Você aprenderá sobre o valor Progress **Unknown** (?) na seção “Usando o valor Progress Unknown”. O formato de exibição padrão para o tipo DATE que possuem o valor desconhecido é em branco; para outros tipos de dados, é um ponto de interrogação.

#### Definindo formatos

A lista de formatos padrão para diferentes tipos de dados nos mostrou alguns dos caracteres de formato suportados pelo Progress. A tabela a seguir fornece um resumo rápido dos símbolos de formato que você provavelmente usará:

|  |  |
| --- | --- |
| **Este caractere de formato...** | **Representa...** |
| **X** | Qualquer caractere único. |
| **N** | Um dígito ou uma letra. Um espaço em branco não é permitido. |
| **A** | Uma letra. Um espaço em branco não é permitido. |
| **!** | Uma letra que é convertida em maiúscula durante a entrada. Um espaço em branco não é permitido. |
| **9** | Um dígito. Um espaço em branco não é permitido. |
| **(n)** | Um número que indica quantas vezes repetir o caractere de formato anterior. |
| **>** | Um dígito inicial em um valor numérico, a ser suprimido se o número não tiver tantos dígitos. |
| **<** | Usado em conjunto com com > para implementar formatos floating-decimal. O character < (até 10) deve aparecer a direita da decimal e ser equilibrado por um número igual ou maior que os caracteres > à esquerda da decimal. Um dígito é exibido na posição formatada com < quando a correspondente > é um zero à esquerda. |
| **Z** | Um dígito inicial em um valor numérico, a ser substituído por um espaço em branco se o número não tiver tantos dígitos. |
| **\*** | Um dígito inicial em um valor numérico, a ser exibido como um asterisco se o número não tiver tantos dígitos. |
| **,** | Uma vírgula em um valor numérico maior que 1.000. Isso é substituído por um ponto no formato europeu. Ele é suprimido se for precedido por um Z, \* ou > e o número não tiver dígitos suficientes para exigir a vírgula. |
| **.** | Um ponto decimal em um valor numérico. Isso é substituído por uma vírgula no formato europeu. |
| **+** | Um sinal para um número positivo ou negativo. É exibido como + para um número positivo e a – para um número negativo. |
| **-** | Um sinal para um número negativo. É exibido como um – para um número negativo. Tabela 2–4: Desenvolvimento OpenEdge: Manual do Progress 4GL Para um número positivo, ele é suprimido se estiver à esquerda do ponto decimal no formato e substituído por um espaço em branco se estiver à direita do ponto decimal. |

Você pode inserir outros caracteres como desejar nos formatos e eles são exibidos como valores literais. Por exemplo, o valor **INTEGER** 1234 com o **FORMAT** $>,>>>ABC é exibido como $1.234ABC.

#### Outros qualificadores de variáveis

Você pode qualificar uma definição de variável de várias maneiras para modificar esses padrões. Para fazer isso, anexe uma das palavras-chave listadas na seguinte tabela ao final de sua definição de variável:

|  |  |
| --- | --- |
| **Keyword** | **Seguido por** |
| INITIAL | O valor inicial da variável. |
| DECIMALS | O número de casas decimais para uma variável **DECIMAL**. |
| FORMAT | O formato de exibição da variável, entre aspas. |
| LABEL | O rótulo a ser exibido com a variável. (O padrão é o próprio nome da variável.) |
| COLUMN-LABEL | O rótulo a ser exibido com a variável quando ela é exibida como parte de um coluna de valores. (O padrão é o **LABEL**, se houver, caso contrário, o nome da variável.) |
| EXTENT | Uma constante inteira. Esse qualificador permite definir uma variável que é uma matriz de valores baseada em um. Você pode fazer referência aos elementos individuais da matriz em seu código colocando o subscrito da matriz entre colchetes, como em myVar[2] = 5. |

Como alternativa para especificar tudo isso, você pode usar o seguinte formulário de sintaxe para definir uma variável que seja LIKE a outra variável ou campo de banco de dados que você definiu anteriormente:

DEFINE VARIABLE varname LIKE fieldname.

Nesse caso, ele herda o formato, rótulo, valor inicial e todos os outros atributos da outra variável ou campo. Esta é outra força do Progress 4GL. Suas procedures da aplicação podem herdar muitos atributos de campo do esquema do banco de dados, de modo que uma alteração no esquema seja propagada para todos os seus procedimentos automaticamente quando eles forem recompilados. Você também pode modificar esses padrões de esquema em procedimentos individuais.

Há mais uma keyword que você deve usar quase sempre no final de suas definições de variável, que é NO-UNDO. Esta keyword tem a ver com a forma como o Progress gerencia as transações que atualizam o banco de dados. Quando você define variáveis em seus programas ABL, o Progress as coloca em dois grupos. Aqueles que não possuem o qualificador NO-UNDO são tratados como se fossem um registro de banco de dados com essas variáveis como campos. Se qualquer um dos valores de variável for modificado durante uma transação de banco de dados e, em seguida, a transação for desfeita, as alterações nos valores de variáveis feitas durante a transação também serão desfeitas, para que sejam revertidas para seus valores antes do início da transação. Isso pode ser muito útil às vezes, mas, na prática, a maioria das variáveis não precisa desse tratamento especial e, como o Progress precisa fazer algum trabalho extra para gerenciá-las, é mais eficiente adquirir o hábito de acrescentar NO-UNDO ao final de suas definições de variável, a menos que a variável deva ser tratada como parte de uma transação.

##### Convenções de nomenclatura de variáveis

Não há requisitos de nomenclatura específicos para variáveis, mas há algumas diretrizes recomendadas que alinharão suas próprias variáveis com os padrões usados nas ferramentas de desenvolvimento OpenEdge e seu código de suporte.

Você deve começar uma variável com uma letra minúscula (ou às vezes duas) para indicar o tipo de dado da variável. Isso pode ajudar os leitores do seu código a entender rapidamente como uma variável está sendo usada. Quando você começar a fazer uma programação mais dinâmica posteriormente, é muito importante diferenciar entre uma variável que representa um valor diretamente e uma que é um manipulador para um objeto que tem um valor. Aqui estão alguns prefixos de tipo de dados recomendados:

* c para CHARACTER
* i para INTEGER
* f para DECIMAL (float)
* d para DATE
* h para HANDLE
* l para LOGICAL

O restante do nome deve estar em maiúsculas e minúsculas, onde letras maiúsculas são usadas para identificar diferentes palavras ou subpartes de um nome, como em **cCustName** ou **iOrderCount**.

##### Alocação das definições de variáveis

Onde você coloca variáveis em seu código também faz diferença. O Progress faz uma única passagem pelas instruções em uma procedure para construir o código intermediário que ele usa para executá-la. Como uma variável é um elemento de definição de um procedimento e não uma instrução executada em sequência, realmente não importa onde a definição da variável aparece. No entanto, devido à natureza de passagem única do analisador de sintaxe Progress, a definição deve aparecer antes que a variável seja usada no procedimento. Por convenção, geralmente é melhor definir todas as suas variáveis no início de um procedimento, para facilitar a leitura e garantir que todas sejam definidas antes de serem usadas.

Para a próxima alteração na procedure de teste, você colocará em prática vários dos conceitos que acabou de aprender. Você exibirá um valor especial para cada registro de **Order**. Esta tarefa envolve definir uma variável com um valor inicial, escrever uma construção IF-THEN com uma verificação para o valor **Unknown** e, em seguida, usar uma das muitas funções integradas do Progress ABL para extrair um valor a ser exibido. O valor que você exibirá é o mês da **ShipDate** (data de entrega) de um pedido abreviado com três caracteres, exemplo, JAN, FEB.

Para construir a lista de valores possíveis para o mês, você precisa definir uma variável CHARACTER para manter a lista. Adicione a seguinte definição de variável ao início de seu procedimento:

DEFINE VARIABLE cMonthList AS CHARACTER NO-UNDO

INIT "JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC".

#### Definindo uma condicional IF-THEN-ELSE

Você notará ao executar seu procedimento que algumas das datas de envio não estão definidas. Algumas dessas Ordens aparentemente estão em espera há muito tempo! Vamos construir uma instrução para ramificar no código verificando se o campo ShipDate está definido ou não. O ABL, como a maioria das linguagens, possui uma construção IF-THEN para definir condicionais. Também pode ter uma ramificação ELSE:

IF condition THEN { block | statement } [ ELSE { block | statement } ]

A condição é qualquer expressão avaliada como TRUE ou FALSE. Após a palavra-chave THEN, você pode colocar uma única instrução ABL ou um conjunto completo de instruções que devem ser executadas se a condição for verdadeira. A maneira de representar um bloco que simplesmente agrupa uma série de instruções que são executadas juntas é um bloco DO-END:

IF condition THEN

DO:

statement

.

.

.

END.

Se você incluir a keyword ELSE seguida por uma instrução ou um bloco de instruções, essa ramificação será executada se a condição IF-THEN for avaliada como falsa.

##### Usando o valor Progress Unknown

Embora **ShipDate** seja exibido em branco nesses casos, o valor armazenado no banco de dados não é um valor em branco, porque é uma data, não um valor de caractere. Se você adicionasse uma expressão ao seu código para comparar **ShipDate** com um valor em branco, obteria o primeiro erro do compilador Progress ABL:

FOR EACH Customer WHERE State = "NH" BY City:

DISPLAY CustNum NAME City.

FOR EACH Order OF Customer:

IF ShipDate NE "" THEN

DISPLAY OrderNum LABEL "Order"

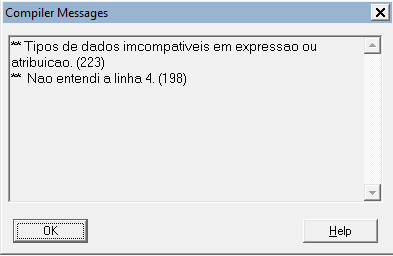
OrderDate

ShipDate FORMAT "99/99/99" WITH CENTERED.

END.

END.

Sempre que você pressiona a tecla **F2** para executar seu procedimento, o Progress executa uma validação de sintaxe. Se não puder processar corretamente o código que você escreveu, você receberá um erro como este.



Nesse caso, o Progress vê que você está tentando comparar um campo definido para ser uma data com uma constante de cadeia de caracteres e eles não correspondem. Para corrigir isso, você precisa alterar a natureza da sua comparação.

O valor armazenado no banco de dados para representar um valor que não está definido é chamado de valor **Unknown**. Em instruções ABL, você representa o valor Desconhecido com um ponto de interrogação (**?**). Observe que você não coloca aspas ao redor do ponto de interrogação; ele age como um símbolo especial por conta própria. Não é igual a nenhum valor definido. No ABL, você escreve uma instrução que parece estar comparando um valor a um ponto de interrogação, como IF ShipDate = ?, mas na verdade a instrução está perguntando se ShipDate tem o valor Unknown (Desconhecido), o que significa que não tem nenhum valor atrubuído. O valor ?(Unknown) é como o valor NULL em SQL e a expressão IF ShipDate = ? é como a expressão SQL IF ShipDate IS NULL.

##### Usando funções ABL integradas

Para concluir esta última alteração na procedure, você precisa definir a instrução que verifica se **ShipDate** é Unknown e, em seguida, seleciona o mês a partir da data, usando a variável **cMonthList** que você definiu logo acima e a converte em uma das abreviações de 3 letras da lista. Aqui está:

IF ShipDate NE ? THEN

DISPLAY ENTRY(MONTH(ShipDate), cMonthList) LABEL "Month".

Agora dê uma olhada mais de perto nos elementos da instrução DISPLAY. Primeiro, há uma nova palavra-chave, ENTRY. Este é o nome de uma função incorporada no ABL. Existem muitas dessas funções para fazer trabalhos úteis para você, para poupar o trabalho de escrever o código com toda a lógica. A função ENTRY recebe dois argumentos e, como você pode ver, esses argumentos estão entre parênteses. O primeiro é um valor INTEGER, que identifica uma posição em uma lista de strings separadas por vírgulas. Neste caso representa o mês do ano, de 1 a 12. O segundo argumento é a lista de valores string separadas por virgula que contém a posição que a função está recuperando. Neste caso, é a variável cMonthList que você acabou de definir.

Olhando mais de perto, você pode ver que o primeiro dos dois argumentos da função, MONTH(ShipDate), é outra função. Essa função usa um valor Progress DATE como argumento, extrai o número do mês da data e o retorna. O valor retornado é um INTEGER de 1 a 12 que a função ENTRY usa para selecionar a entrada correta da lista de meses em cMonthList. Portanto, se o mês for maio, a função MONTH retornará 5 e a função ENTRY seleciona a quinta entrada da lista de meses e a retorna para a instrução DISPLAY.

Aqui estão algumas observações gerais sobre funções internas:

* Uma função recebe um número variável de argumentos, dependendo do que a função requer. Algumas funções não aceitam nenhum argumento (por exemplo, a função TODAY, que retorna a data de hoje). Algumas funções têm um número variável de argumentos, de modo que um ou mais argumentos no final da lista de argumentos são opcionais. Por exemplo, a função ENTRY pode ter um terceiro argumento opcional, que é uma cadeia de caracteres que representa um delimitador a ser usado entre os valores da lista, caso você não queira usar uma vírgula (,). Como a vírgula é o delimitador padrão, ela é opcional. Você não pode deixar de fora os argumentos do meio da lista ou especificá-los em uma ordem diferente. Cada um dos argumentos deve ser do tipo de dados adequado, dependendo do que a função espera.
* Os argumentos para uma função podem ser valores constantes, nomes de variáveis, nomes de campo de banco de dados ou qualquer expressão que os envolva e que resulte em um valor do tipo de dados adequado.
* Cada função retorna um valor de um tipo de dado específico.
* Você pode aninhar funções em qualquer profundidade em seu código. O resultado de qualquer função é retornado para o próximo nível no código.
* Você pode colocar funções em qualquer lugar dentro de uma instrução onde um valor possa aparecer. Como uma função retorna um valor, ela pode aparecer apenas no lado direito de uma instrução de atribuição. Você não pode usar a função MONTH para atribuir o valor do mês a uma variável de data, por exemplo, ou a função ENTRY para atribuir o valor de uma entrada em uma lista. Existem keywords de instrução ABL em alguns casos para fazer esses tipos de atribuições.
* Se você estiver exibindo o resultado de uma função ou uma expressão envolvendo uma função, poderá especificar um LABEL ou FORMAT para ela, assim como para uma variável. O LABEL padrão para uma expressão é uma string de texto que representa a própria expressão. O FORMAT padrão é o padrão para o tipo de dados da função. Neste exemplo, você deve adicionar a LABEL **Month** à expressão.

Para exibir o mês junto com cada Pedido:

Adicione a nova instrução com as referências de função em seu procedimento, dentro do bloco de código que percorre os Pedidos:

You use the string + Concatenation operator to combine two or more strings:

string1 + string2 + string3

If you want spaces or punctuation to appear between string values, then you must explicitly include them. For example, to display a person's full name, you might write the following code:

VAR CHAR fullName.

VAR CHAR firstName = "Alexander". VAR CHAR middleInitial = "T".

VAR CHAR lastName = "Great".

fullName = firstName + " " + middleInitial + ". " + lastName. MESSAGE "My name is: " fullName.

Running the code produces the following output:

My name is: Alexander T. Great

###### SUBSTITUTE function

The SUBSTITUTE function returns a character string that is made up of a base string plus the substitution of parameter values in the string. This function allows you to use a single string in place of concatenated strings.

*new-string* = SUBSTITUTE( *base-string* [ ,

*arg* ] ... )

*new-string*

The string returned by the SUBSTITUTE function.

*base-string*

A character string optionally containing substitution parameters of the form &*n*, where *n* is an integer between 1 and 9, inclusive.

*arg*

A constant, field name, variable, or expression that results in a character string value. These values replace substitution parameters in *base-string*.

The following example code uses substitution rather than concatenation, but produces the same output as the previous example. In this example, firstName replaces &1; middleInitial replaces &2; and lastName replaces &3.

VAR CHAR newString.

VAR CHAR firstName = "Alexander". VAR CHAR middleInitial = "T".

VAR CHAR lastName = "Great".

newString = SUBSTITUTE("My name is: &1 &2. &3", firstName, middleInitial, lastName). MESSAGE newString.

###### REPLACE function

You can use the REPLACE function to replace characters in a variable with an expression. The syntax for the REPLACE function is:

*new-string* = REPLACE(*source-string, from-string, to-string*) *new-string*

The string returned by the REPLACE function.

*source-string*

Specifies the base string to make replacements in. The *source-string* parameter can be any expression that evaluates to a string. The REPLACE function does not change the value of *source-string* itself, unless *new-string* and *source-string* refer to the same variable.

*from-string*

Specifies the substring to replace. The *from-string* parameter can be any expression that evaluates to a string. Each occurrence of *from-string* within *source-string* is replaced.

*to-string*

Specifies the replacement substring. The *to-string* parameter can be any expression that evaluates to a string. Each occurrence of *from-string* in *source-string* is replaced by *to-string*.

Example: Replace a text string

The following example code creates a list of departments separated by commas. The REPLACE function changes the separator from a comma to a bar surrounded by spaces, for better readability.

VAR CHAR cDeptList.

FOR EACH Department:

cDeptList = cDeptList + "," + DeptName. end.

MESSAGE cDeptList VIEW-AS ALERT-BOX.

cDeptList = REPLACE(cDeptList, ",", " | "). MESSAGE cDeptList VIEW-AS ALERT-BOX.

The example code produces the following output:

,Consulting,Administration,Marketing,Sales,Training,Development,Finance,Shipping

| Consulting | Administration | Marketing | Sales | Training | Development | Finance | Shipping

###### Convert data to the ABL CHARACTER type

The STRING function allows you to convert ABL non-character data types to a character value. Many data types are supported, including:

* INTEGER or INT
* DECIMAL
* LOGICAL
* DATE
* DATETIME
* DATETIME-TZ

You can concatenate a string representing different data types, provided you first convert each non-character type to a string.

The STRING function syntax is shown:

|  |
| --- |
| STRING ( *value* [, "*format-string*" ] ) |

*value*

A variable, field, or an expression that evaluates to a numeric, date and time, or logical value.

*format-string*

The customized format for the string that is returned by this function.

Example: Use STRING() to create character data

You can create a message string from multiple values, some of which may not be CHARACTER types. In the following example code, the STRING function is used to convert a DATETIME value to a character string that is part of a concatenated message string.

|  |
| --- |
| VAR CHAR cTime.  VAR CHAR cDateStr.  VAR CHAR cTimeStr.  cDateStr = "Today's date is " + STRING(TODAY).  /\* convert the current time to a string \*/ cTime = STRING(TIME, "hh:mm:ss am").  cTimeStr = "and the time is " + cTime.  MESSAGE cDateStr SKIP cTimeStr VIEW-AS ALERT-BOX. |

The example code produces output similar to the following:

|  |
| --- |
| Today's date is 05/07/20 and the time is 11:20:10 am |

###### Work with LONGCHAR data

A LONGCHAR variable consists of CHARACTER data, but unlike the CHARACTER type, this type is not limited to 32K in size. A LONGCHAR variable can be larger, but it may be limited by available memory. You can use many of the ABL character handling capabilities with LONGCHAR strings. The LONGCHAR data type has metadata associated with it to designate its code page and other information. The default initial value of a LONGCHAR variable is the empty string ("").

ABL has another data type known as a character large object (CLOB). A CLOB field is a database table or temp-table field that contains a locater, which points to the associated CLOB data stored in the database. You cannot manipulate CLOB data directly. Instead, you must copy a CLOB to a LONGCHAR to manipulate the character contents of a CLOB field in ABL. You can do this with the COPY-LOB statement.

Like CLOB values, LONGCHAR values can be in any code page supported by the OpenEdge convmap.p file, and the code page information is retained by the LONGCHAR. ABL ensures that all characters are valid with respect to the code page of the LONGCHAR.

###### Display a LONGCHAR

Because LONGCHAR variables are large in size, you cannot display the values directly. However, there are some ways around this. For shorter LONGCHAR variables, you can assign the LONGCHAR to a CHARACTER variable and then display the CHARACTER variable. Another way is to use the STRING function to convert it to a CHARACTER value. The following example code demonstrates these ways:

|  |
| --- |
| VAR LONGCHAR str1 = "abcdefghijklmnopqrstuvwxyz". VAR CHAR str2.  MESSAGE str1. // Results in an error  /\* Assign the LONGCHAR to a CHARACTER variable \*/ str2 = str1.  MESSAGE str2.  /\* Use the STRING function \*/ MESSAGE STRING(str1). |

You can display a portion of a LONGCHAR variable using the SUBSTRING function, but you must still assign the LONGCHAR variable to a CHARACTER variable. Because the input to the SUBSTRING function is a LONGCHAR, its return value is also a LONGCHAR, and you cannot directly display a LONGCHAR. The following example code shows how to display a portion of a LONGCHAR variable.

|  |
| --- |
| VAR LONGCHAR str1 = "abcdefghijklmnopqrstuvwxyz". VAR CHAR str2.  MESSAGE SUBSTRING(str1,1,13,"CHARACTER"). // Results in an error  /\* Use the SUBSTRING function and assign it to a CHARACTER variable \*/ str2 = SUBSTRING(str1,1,13,"CHARACTER").  MESSAGE str2. |

You can also display a LONGCHAR by using the DISPLAY statement with the VIEW-AS EDITOR LARGE option. The following example code shows how to display a LONGCHAR using this option. Note that you must set the number of lines and the line width, in characters. The string in the following example code does not exceed 32K, but is used for illustration purposes only.

|  |
| --- |
| VAR LONGCHAR myLongchar = "Very long string...". DISPLAY myLongchar VIEW-AS EDITOR LARGE  INNER-LINES 60 INNER-CHARS 100 WITH FRAME f1 WIDTH 120. |

For more information, see the following topics in *ABL Data Types Addenda*:

* LONGCHAR data type
* Copying large objects
* What you can do with LONGCHAR values
* Restrictions
* LONGCHAR code pages

###### Work with lists

Comma-separated lists are used to store multiple elements in a single character variable. For example, you might want to store:

* Sales quotas for 12 months
* The department names in your company
* The names of the 12 months of the year

Define a list

A list is a string with substrings, each separated by a separator character. A comma is the default separator.

The following example code shows two examples of list variables. You do not need to initialize a list variable when it is defined, but it must have items in it in order for you to work with it.

|  |
| --- |
| VAR CHAR cMonths = "Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec". VAR CHAR cQuarterlyQuotas = "100500,400000,250000,175000". |

Access a value in a list

ABL provides functions to access and manipulate the items in a list. These functions allow you to:

* Locate the position of an item in the list (LOOKUP function).
* Extract a value from a particular position in the list (ENTRY function).
* Find out how many items are in the list (NUM-ENTRIES function).
* Set a particular item in a list to a value (ENTRY statement).

###### Look up an item in a list

Use the LOOKUP function to determine whether a specific item is in a list. The LOOKUP function returns 0 if the item is not in the list; otherwise, it returns an integer value of the item's position in the list.

The syntax for the LOOKUP function is:

|  |
| --- |
| *position* = LOOKUP(*expression*, *list*) |

*position*

Returns an INTEGER value of the position of *expression* in *list*. If not found, then *position*

is set to 0.

*expression*

A character expression to search for in *list*. *list*

A CHARACTER variable that contains a comma-separated list of items.

Example: Use LOOKUP with a list

In the following example code, a list called cMonths is defined, which contain the months of the year. expr

is the expression to search for in the list and is set to "Sep".

|  |
| --- |
| VAR CHAR cMonths = "Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec". VAR CHAR expr = "Sep".  VAR INT position.  position = LOOKUP(expr, cMonths).  MESSAGE "cMonths:" cMonths SKIP "expr:" expr SKIP  "position:" position VIEW-AS ALERT-BOX. |

The following output is produced from running the code. Note that the position is set to 9, because "Sep" occurs in the ninth position in the list.

|  |
| --- |
| cMonths: Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec expr: Sep  position: 9 |

###### Extract a value from a list

Use the ENTRY function to return an entry from a list based on an integer position.

|  |
| --- |
| *item* = ENTRY(*position*,*list*) |

The following example code extracts address data from a comma-separated list.

|  |
| --- |
| VAR CHAR cAddress.  cAddress = "Lift Tours,276 North Drive,,Burlington,MA,01730,USA".  MESSAGE "Name:" ENTRY(1,cAddress) SKIP  "Address line 1:" ENTRY(2,cAddress) SKIP "Address line 2:" ENTRY(3,cAddress) SKIP "City:" ENTRY(4,cAddress) SKIP  "State:" ENTRY(5,cAddress) SKIP "Zipcode:" ENTRY(6,cAddress) SKIP  "Country:" ENTRY(7,cAddress) VIEW-AS ALERT-BOX. |

The following output is produced from running the code.

|  |
| --- |
| Name: Lift Tours  Address line 1: 276 North Drive  Address line 2: City: Burlington State: MA Zipcode: 01730 Country: USA |

###### Determine how many entries are in a list

Use the NUM-ENTRIES function to return how many items are in a list. NUM-ENTRIES is useful when you want to process items in the list in a loop. The value of NUM-ENTRIES tells you the number of iterations you need to do.

The example code uses NUM-ENTRIES to determine the number of times to iterate in the REPEAT block.

|  |
| --- |
| VAR INT ix = 1.  VAR CHAR cMonths = "Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec".  REPEAT ix = 1 TO NUM-ENTRIES(cMonths):  DISPLAY ix " " ENTRY(ix, cMonths).  END. |

Running the code produces the following output:

|  |
| --- |
| ix   1. Jan 2. Feb 3. Mar 4. Apr   5 May   1. Jun 2. Jul 3. Aug 4. Sep 5. Oct 6. Nov 7. Dec |

###### Set a value in a list

You can substitute a value in a list with a different value by using the ENTRY statement. The syntax for the

ENTRY statement is:

|  |
| --- |
| ENTRY(*position*, *list*) = *expr*. |

*position*

An integer value that corresponds to the position of a character string in a list.

*list*

A list of character strings separated with a character delimiter.

*expr*

A character value that replaces the item in the list at the specified position.

The following example code substitutes character strings in French.

|  |
| --- |
| VAR CHAR cList = "North,South,East,West". MESSAGE "English:" cList VIEW-AS ALERT-BOX.  ENTRY(1, cList) = "Nord". ENTRY(2, cList) = "Sud". ENTRY(3, cList) = "Est".  ENTRY(4, cList) = "Ouest".  MESSAGE "French:" cList VIEW-AS ALERT-BOX. |

Running the code produces the following output:

|  |
| --- |
| English: North,South,East,West |

|  |
| --- |
| French: Nord,Sud,Est,Ouest |

##### Work with one-dimensional arrays

Arrays are used in situations where you want multiple values for a variable. In ABL, there are only

one-dimensional arrays. Unlike a list, which only applies to CHARACTER data, an array can be defined for any ABL data type. To define an array, you use the VAR statement or the DEFINE VARIABLE statement. You can optionally specify the size (extent) of the array, that is the number of elements the variable can hold. If you specify a size, then the variable is a determinate array. If no size is specified when defined, then the variable is an indeterminate array and its size can be set at runtime.

###### Define an array variable

The syntax for defining an array variable using the VAR statement is:

|  |
| --- |
| VAR *type-name[size] name* **[** = *[val1,...valn]* **]**. |

The syntax for defining a variable as an array using the DEFINE VARIABLE statement is:

|  |
| --- |
| DEFINE VARIABLE *name* AS *type-name* EXTENT [*size*] [NO-UNDO] [INITIAL *[val1,...valn]*]. |

*name*

The name of the variable.

*type-name*

An ABL data type.

*size*

An INTEGER value containing the maximum number of elements the array can hold. If *size* is specified, then the array is considered a determinate array and its size cannot change at run time. If *size* is not specified, then the array is considered an indeterminate array and its size can change at run time.

*val1*,...*valn*

An optional, comma-separated list of initial values for the variable. The values are enclosed in brackets. You are not required to initialize all elements of the array.

Example

The following statement defines an array to hold the monthly sales quotas for a sales representative. The array is of type INTEGER and contains 12 elements:

|  |
| --- |
| VAR INTEGER[12] monthlySalesQuotas. |

###### Access an element of an array

In ABL, array elements are indexed. The first element of the array has an index value of 1. Unlike lists, you cannot efficiently look up an element in an array by value. To find a specific value, you must iterate all elements using an index. To access an element of an array, you enclose the index number in brackets.

|  |
| --- |
| VAR INT[12] monthlySalesQuotas = [100500,125000,125000,175000,150000, 155000,145000,160000,130000,140000,145000,150000].  MESSAGE "Sales quota for April: $" monthlySalesQuotas[4] VIEW-AS ALERT-BOX. |

The example code produces the output:

|  |
| --- |
| Sales quota for April: $ 175000 |

For more information, see [ ] Array reference in the *ABL Reference*.

###### Determinate and indeterminate arrays

Whether the array is determinate (the size cannot change) or indeterminate (the size can change) depends on its definition.

To define a determinate array variable, specify an integer value for *size* using either the VAR or DEFINE VARIABLE statement.

To define an indeterminate array variable, do not specify a size in the variable definition statement. If initial values are provided in the definition, then the initial size of the array will be the number of values provided. If no initial values are provided, then the array remains uninitialized and its size must be set at runtime before you can work with it. The size of the indeterminate array can be set and reset at runtime using the EXTENT statement.

The example code shows variable definitions for determinate and indeterminate arrays:

|  |
| --- |
| VAR INT[3] a1. // determinate array VAR INT[3] a2 = [1,2,3]. // determinate array VAR INT[] a3. // indeterminate array VAR INT[] a4 = [4,5,6]. // indeterminate array |

An indeterminate array can be in one of two states: with a size (fixed) or without a size (unfixed). An indeterminate array does not have a size when first defined, unless initial values are provided. You can set the size of an indeterminate array with no size by:

* providing initial values. The size becomes the number of elements provided.
* setting the number of elements in the array using the EXTENT statement.
* assigning an array with a size to the indeterminate array. The size becomes the size of the source array.
* passing array parameters to a procedure, user-defined function, or class-based method, so that the indeterminate array value is the target for the passing of a source array with a size. The array size becomes the size of the source array.

###### EXTENT function

To determine how many elements are in an array, you can use the EXTENT function. A numeric value is returned if there are elements. If there are no elements, then the Unknown value (?) is returned.

|  |
| --- |
| /\* Indeterminate array \*/  VAR INT[] monthlySalesQuotas = [100500,125000,125000,175000,150000,155000,145000, 160000,130000,140000,145000,150000].  /\* Determinate array \*/ VAR INT[12] monthlySales.  /\* Indeterminate array \*/ VAR INT[] qtr1Sales.  MESSAGE "monthlySalesQuotas array has" EXTENT(monthlySalesQuotas) "elements." SKIP "monthlySales array has" EXTENT(monthlySales) "elements." SKIP  "qtr1Sales array has" EXTENT(qtr1Sales) "elements." VIEW-AS ALERT-BOX. |

Running the code produces the output:

|  |
| --- |
| monthlySalesQuotas array has 12 elements. monthlySales array has 12 elements. qtr1Sales array has ? elements. |

###### EXTENT statement

The EXTENT statement sets or resets the size of an indeterminate array after it is defined. The EXTENT

statement:

* Sets the number of elements for an indeterminate array variable or parameter
* Resizes an indeterminate array variable or parameter

After the size is set, you can use the indeterminate array as you would any determinate array. You cannot use the EXTENT statement on a determinate array; if you do, you get an error.

The example code uses the EXTENT statement to set the number of elements of an indeterminate array.

|  |
| --- |
| /\* Define an indeterminate array \*/ VAR INT[] qtr1Sales.  MESSAGE "qtr1Sales array has" EXTENT(qtr1Sales) "elements." VIEW-AS ALERT-BOX.  /\* Set the number of elements of the indeterminate array to 4 \*/  **EXTENT (qtr1Sales) = 4.**  MESSAGE "Set the size: qtr1Sales array has" EXTENT(qtr1Sales) "elements." VIEW-AS ALERT-BOX.  /\* Resize the indeterminate array to 12 elements \*/  **EXTENT (qtr1Sales) = 12.**  MESSAGE "Resize the array: qtr1Sales array has" EXTENT(qtr1Sales) "elements." VIEW-AS ALERT-BOX. |

Running the code produces the output:

|  |
| --- |
| qtr1Sales array has ? elements. |

|  |
| --- |
| Set the size: qtr1Sales array has 4 elements. |

|  |
| --- |
| Resize the array: qtr1Sales array has 12 elements. |

###### Assign a scalar value to all elements of an array

You can assign a scalar value to all the elements of a determinate array (or indeterminate array with a size) using the assignment operator.

|  |
| --- |
| VAR INT[2] a1. // determinate array VAR INT[] a2. // indeterminate array  a1 = 1. // Assign 1 to each element.  DISPLAY a1. // Display whole array (size known at compile time)  EXTENT(a2) = 2. // Set size to 2 a2 = 2. // Assign 2 to each element  DISPLAY a2[1] a2[2]. // Display elements individually  // (size not known at compile time) |

The example code produces the output:

|  |
| --- |
| a1[1] a1[2] a2[1] a2[2] 1 1 2 2 |

###### Assign an array to another array (deep copy)

You can assign one array to another (Deep copying arrays), provided it makes semantic sense to do so. When deep copying one array to another, the following rules apply:

* You can assign any array to an indeterminate array whose size is not set.
* You cannot assign an indeterminate array with no size to an array with a size (determinate or indeterminate)
* If either of the arrays are a determinate array, or an indeterminate array with a size, then the size must match. If the size does not match, then the AVM raises an error.
* You cannot assign a scalar value to an indeterminate array with no size.

The example code shows a determinate array assigned to another determinate array. They both have the same size.

|  |
| --- |
| VAR INT[3] sourceArray = [1,2,3]. VAR INT[3] targetArray.  targetArray = sourceArray. DISPLAY targetArray. |

Running the example code produces the output:

|  |
| --- |
| targetArray[1] targetArray[2] targetArray[3] 1 2 3 |

##### Work with handles

The HANDLE data type stores a pointer to a structure that represents a running built-in ABL procedure or an object in a procedure such as a socket, temp-table, or button. Handles allow you to access attributes and methods that ABL defines for each object, which you can use to interact with an object's state and behavior.

Handle-based objects are conceptually analogous to, but different from, class-based objects. Unlike class-based objects, the types and capabilities of all handle-based objects are built in to the AVM. You cannot define your own handle-based objects. Handle-based objects are also not garbage collected. You must manage the memory by cleaning up handle-based objects, using the DELETE OBJECT statement, when they are no longer needed.

ABL automatically creates instances of certain other handle-based object types in various ways, including system objects that you can access using an appropriate keyword-defined system handle. Examples of some system object handles include:

* SESSION system handle
* FILE-INFO system handle
* ERROR-STATUS system handle
* SECURITY-POLICY system handle

You define static instances, or create dynamic instances of many handle-based object types using the DEFINE and CREATE statements. You reference a static instance by the defined object name or its handle. You reference a dynamic instance only by its handle. The following example code show you how to define and access both a static and dynamic instance of an object.

|  |
| --- |
| /\* Define and access a static instance of an object \*/  /\* Define a temp-table \*/  DEFINE TEMP-TABLE ttCustomer NO-UNDO FIELD CustName AS CHARACTER  FIELD CustId AS CHARACTER.  /\* Define a handle variable \*/ VAR HANDLE tth.  /\* Set the handle to the statically defined temp-table \*/ tth = TEMP-TABLE ttCustomer:HANDLE.  /\* Access the HAS-RECORDS method using the handle \*/ DISPLAY "ttCustomer has records:" tth:HAS-RECORDS.  /\* Alternatively, access the HAS-RECORDS method using the object name\*/ DISPLAY "ttCustomer has records:" TEMP-TABLE ttCustomer:HAS-RECORDS. |

|  |
| --- |
| /\* Define and access a dynamic instance of an object \*/  /\* Define a handle variable \*/ VAR HANDLE tth.  /\* Create an empty temp-table \*/ CREATE TEMP-TABLE tth.  tth:ADD-NEW-FIELD("CustName","CHARACTER"). tth:ADD-NEW-FIELD("CustId","CHARACTER"). tth:TEMP-TABLE-PREPARE("cust").  /\* Access the HAS-RECORDS method using the handle \*/ DISPLAY "tth has records:" tth:HAS-RECORDS. |

Some commonly used object type handles in ABL are:

* Temp-table object handle
* Buffer object handle
* Buffer-field object handle
* Query object handle
* ProDataSet object handle
* Data-relation object handle
* Data-source object handle

For the complete list of system and object handles, see the Handle Reference. Attributes and methods for object type handles are described in the Handle Attributes and Methods Reference.

### Control flow

ABL supports many control flow statements. In the following topics, you learn about the IF THEN ELSE, CASE, DO, FOR, and REPEAT statements.

#### IF ... THEN ... ELSE statement

An IF...THEN...ELSE statement is used to execute one or more ABL statements, based upon a logical condition that evaluates to true.

Syntax

|  |
| --- |
| IF *expression* THEN **{** *block* **|** *statement* **} [** ELSE **{** *block* **|** *statement* **} ]** |

If the value of the condition following the IF keyword is true, the AVM executes the statement or statements following the THEN keyword. If you want to execute multiple statements, they must be contained in a DO block. A DO block begins with DO: and ends with END. You can add an ELSE keyword to your IF THEN statement to handle the false condition.

The following code example contains an IF THEN ELSE statement:

|  |
| --- |
| VAR CHAR custRating.  VAR INT balance.  balance = 3000.  IF (balance <= 1000) THEN custRating = "A". ELSE DO:  IF (balance <= 5000) THEN custRating = "B". ELSE custRating = "F".  END.  MESSAGE "Customer has a rating of" custRating. |

Running the code produces the following output:

|  |
| --- |
| Customer has a rating of B |

#### CASE statement

The CASE statement provides a multi-branch decision based on the value of a single expression. You can specify multiple conditions, each with its own code, and execute the code when a condition is true. Each condition is defined in a WHEN … THEN section. In addition, you can define an OTHERWISE section to handle any conditions that are not explicitly defined.

Syntax

|  |
| --- |
| CASE *expression* :  **{** WHEN *value* **[** OR WHEN *value* **] ...** THEN  **{** *block* **|** *statement* **}**  **} ... [** OTHERWISE  **{** *block* **|** *statement* **}**  **]**  END **[** CASE **]**. |

In the following example code, the CASE statement is used to assign the yearly quarter based on the month.

|  |
| --- |
| VAR CHAR qtr.  CASE MONTH(TODAY):  WHEN 1 OR WHEN 2 OR WHEN 3 THEN qtr = "Q1". WHEN 4 OR WHEN 5 OR WHEN 6 THEN qtr = "Q2". WHEN 7 OR WHEN 8 OR WHEN 9 THEN qtr = "Q3". OTHERWISE qtr = "Q4".  END CASE.  MESSAGE "Today’s date is" TODAY SKIP "The current quarter is" qtr. |

Running the code produces output similar to the following, depending on the current date:

|  |
| --- |
| Today’s date is 04/04/19 The current quarter is Q2 |

#### DO statement

The DO statement groups statements into a single block. An END statement ends the DO block.

The DO block can be a simple non-iterating block, or you can use it to iterate. There are two ways you can control iteration in a DO block:

* Iterate a specified number of times
* Iterate while a condition is true

Iterate a specified number of times

To iterate for a specified number of times, you provide a starting and ending integer value for the iteration. By default, the iteration value increments by 1 at the end of each iteration. Optionally, you can provide a different increment or decrement value using the BY clause. The iteration ends when the ending value is reached. This is the basic syntax for using DO to iterate a specified number of times:

|  |
| --- |
| DO *iteration-variable* = *starting-value* to *ending-value* [BY *increment-value*]:  <ABL statements> end. |

The following is an example

|  |
| --- |
| VAR INT ix.  DO ix = 1 TO 10:  MESSAGE "The value of ix is" ix.  END. |

Running the code produces the following output:

|  |
| --- |
| The value of ix is 1 The value of ix is 2 The value of ix is 3 The value of ix is 4 The value of ix is 5 The value of ix is 6 The value of ix is 7 The value of ix is 8 The value of ix is 9 The value of ix is 10 |

Iterate while a condition is true

You can also iterate while a condition is true by using the DO WHILE construct. The following is an example:

|  |
| --- |
| VAR INT ix.  ix = 10.  DO WHILE ix > 0:  MESSAGE "The value of ix is" ix. ix = ix - 1.  END. |

Running the code produces the following output:

|  |
| --- |
| The value of ix is 10 The value of ix is 9 The value of ix is 8 The value of ix is 7 The value of ix is 6 The value of ix is 5 The value of ix is 4 The value of ix is 3 The value of ix is 2 The value of ix is 1 |

#### FOR statement

The FOR statement is similar to DO, but is specifically for iterating through records. The FOR statement starts an iterating block that reads a record from one or more tables at the start of each block iteration. The END statement terminates the FOR block.

Syntax

|  |
| --- |
| FOR ... :  /\* ABL statements \*/ END. |

The following example uses a FOR EACH block to iterate through the records in the Customer table:

|  |
| --- |
| /\* Display customers with a balance under $1000 \*/ FOR EACH Customer WHERE Customer.Balance < 1000:  DISPLAY Customer.Name.  END. |

For more information, see FOR EACH CUSTOMER in *Develop ABL Applications*.

#### REPEAT statement

The REPEAT statement begins a block of ABL statements that are processed repeatedly until the block ends in one of several possible ways. You use an END statement to define the end of the block. REPEAT blocks can be explicitly terminated when runtime conditions, that you have specified in your code, occur. It is good programming practice to include a terminating condition to prevent infinite loops. REPEAT blocks are also useful for iterating over records.

Syntax

|  |
| --- |
| REPEAT ... :  /\* ABL statements \*/ END. |

In the following example the WHILE phrase causes the REPEAT block to terminate after 100 iterations.

|  |
| --- |
| VAR INT ix = 0.  REPEAT WHILE ix < 100:  ix = ix + 1. DISPLAY ix.  END. |

Running the code produces the following output (some output omitted for brevity):

|  |
| --- |
| ix 1  2  3  4  5  ... 99  100 |

In the following code a REPEAT block is used to iterate through database records.

|  |
| --- |
| REPEAT:  FIND NEXT Customer.  DISPLAY Customer.Name Customer.Balance.  END. |

In the following example the REPEAT block is explicitly terminated:

|  |
| --- |
| REPEAT:  FIND NEXT Customer.  IF Customer.CustNum > 1000 THEN LEAVE.  DISPLAY Customer.Name Customer.CustNum Customer.Balance.  END. |

### Expressions and operators

An ABL expression is a combination of one or more terms and operators that evaluate to a single value of a specific data type. An expression can include constants, variable names, database field names, or other expressions.

ABL has a set of operators for working with numeric, string, date, and logical data. The data type of the data must be compatible with the operator. One very useful operator is the concatenation operator (+), used to join two character strings or expressions:

|  |
| --- |
| FOR EACH Customer:  DISPLAY Customer.Name Customer.City + ", " + Customer.State FORMAT "X(20)".  END. |

More information on the ABL operators can be found in the ABL Syntax Reference.

### Built-in functions

ABL supports a lengthy list of built-in functions which provide the ABL developer with easy access to a specific set of capabilities. Each function accepts a well-defined list of parameters and optionally returns a value. There are many functions available for working with strings, numbers, dates and times, and other data. The ABL Syntax Reference contains reference entries for each of these functions.

The following example demonstrates calling a function called SUBSTRING to extract a portion of a string.

|  |
| --- |
| /\* Display the first three characters of the customer's name \*/ FOR EACH Customer:  DISPLAY SUBSTRING(Customer.Name, 1, 3).  END. |

**Handle-based and class-based objects**

Another useful function is the IF...THEN...ELSE function, not to be confused with the IF...THEN...ELSE statement. This function does not look like a typical function but can be very useful in the right situation. The function evaluates and returns one of two expressions, depending on the value of a specified condition. The following example demonstrates the IF...THEN...ELSE function.

|  |
| --- |
| VAR INT i. i = 2.  MESSAGE IF i EQ 1 THEN "low" ELSE "high" VIEW-AS ALERT-BOX. |

Running the code produces the following output:

|  |
| --- |
| high |

For more information, see Using built-In ABL functions in *Develop ABL Applications*.

### Handle-based and class-based objects

ABL supports two models for managing objects which provide access to a variety of capabilities:

* Handle-based objects
* Class-based objects

###### Handle-based objects

Handle-based objects represent built-in object types in ABL that you reference using weakly-typed object handles. These objects provide access to a variety of ABL capabilities, via their attributes and methods. You can define static or create dynamic instances of many handle-based object types. For example, the following program creates a socket object, attempts to connect to an identified resource and then deletes the object.

|  |
| --- |
| VAR HANDLE hSocket.  /\* Create socket and connect to server \*/ CREATE SOCKET hSocket.  hSocket:CONNECT('-H localhost -S 23456') NO-ERROR. IF hSocket:CONNECTED() = FALSE THEN  DO:  MESSAGE 'Unable to Connect' VIEW-AS ALERT-BOX. RETURN.  END.  DELETE OBJECT hSocket. |

In the example code you define a variable to be of type HANDLE. Then you create a socket object using the handle variable with the CREATE SOCKET statement. After creation, attributes and methods can then be called using the handle variable name, followed by a colon, and then the method or attribute name. In the example code, CONNECT and CONNECTED are methods that are called. After the object is no longer needed, DELETE OBJECT is called to clean up resources. Handle-based objects are not garbage collected. It is the responsibility of the application to delete them.

You can also chain handle references together to simplify coding. For more information, see "Chained handle references" in *ABL Reference*.

For more detail, see the ABL handle-based object reference guides:

* Handle Reference
* Handle Attributes and Methods Reference

###### Class-based objects

Class-based objects also represent built-in object types in ABL which you reference using strongly-typed object handles. These objects provide access to a variety of capabilities, via their properties and methods. You can define static or create dynamic instances of class-based object types.

The following program creates a FileInputStream object and reads the first 50 bytes from file sample.data. The AVM deletes this object (via garbage collection) when no more references to the object exist.

|  |
| --- |
| USING Progress.IO.\*.  VAR CLASS FileInputStream rfile. VAR MEMPTR mData.  /\* Create an instance of a FileInputStream object \*/ rFile = New FileInputStream ("sample.data").  rFile:Read(mData, 0, 50). rFile:Close( ). |

For more detail, see the ABL class-based object reference guides:

* Class, Interface, and Enumeration Reference
* Class Properties and Methods Reference

### Code for portability

Different operating systems have different restrictions and naming conventions. To ensure portability across platforms, take care to follow conventions when naming your fields, files, tables, and variables. The following are some recommended practices.

* Do not use a hyphen as the first character of a filename and do not use spaces in filenames.
* Use lowercase when specifying a procedure name in a RUN statement, and make sure your procedure files have lowercase names on UNIX.
* Use forward slashes (/) as separators in specifying file paths when using ABL statements such as RUN, INPUT FROM, or OUTPUT TO. The AVM automatically converts the UNIX pathname syntax to Windows pathname syntax, which helps maintain portability among operating systems.

See Naming conventions for files, tables, and variables for a complete list of conventions to follow to ensure portability across platforms.

# 3

## Procedures and User-defined Functions

In the following topics, you learn how to work with ABL procedures and user-defined functions. You learn the difference between external and internal procedures, how to run a procedure, how to return a value from a procedure, and use parameters with procedures. You also learn about persistent procedures and how best to organize your procedure files. User-defined functions are also introduced.

For details, see the following topics:

* [ABL procedures](#_bookmark27)

* [User-defined functions](#_bookmark28)

### ABL procedures

An ABL procedure file is a text file, with a .p extension, that contains source code for application procedures. The file itself is known as an external procedure, but it may contain within it other procedures, known as internal procedures. This topic contains information pertaining to ABL procedures.

* External and internal procedures
* Run a procedure
* Return a value from a procedure
* Use parameters with procedures
* Persistent procedures
* Organize procedure files

###### External and internal procedures

An ABL procedure (.p) file is known as an external procedure. It may accept parameters or return a string value. The following example code shows an external procedure, helloworld.p:

|  |
| --- |
| /\* helloworld.p \*/ MESSAGE "Hello World". |

An ABL internal procedure is another type of procedure that is defined by named entry points within an external procedure file. You define the start of an internal procedure using the PROCEDURE statement and indicate the end of the procedure using the END statement. You can have many internal procedures within the same external procedure file. The following example code has two internal procedures named proc1 and proc2 in the external procedure file, myprocedures.p.

|  |
| --- |
| /\* myprocedures.p \*/ PROCEDURE proc1:  /\* ABL code \*/  MESSAGE "In proc1".  END PROCEDURE.  PROCEDURE proc2:  /\* ABL code \*/ MESSAGE "In proc2".  END PROCEDURE.  RUN proc1.  RUN proc2. |

Scope of variables for internal procedures

You can define variables at a global level in the main block that can also be used by the internal procedures defined in the same file. Variables defined in the internal procedure, however, can only be used within the internal procedure.

###### Run a procedure

If you are running an external procedure file like helloworld.p, then you include the .p extension in the procedure name. You run internal procedures in almost exactly the same way that you run external procedures, except that there is no .p filename extension on the internal procedure name. The following example code shows running both an external and internal procedure:

|  |
| --- |
| /\* myprocedures.p \*/ PROCEDURE proc1:  /\* ABL code \*/  MESSAGE "In proc1".  END PROCEDURE.  PROCEDURE proc2:  /\* ABL code \*/ MESSAGE "In proc2".  END PROCEDURE.  RUN helloworld.p. // run external procedure RUN proc1. // run internal procedure |

For more information, see Running ABL procedures in *Develop ABL Applications*:

**ABL procedures**

###### Return a value from a procedure

You can use the RETURN statement to return execution to the caller, optionally specifying a string value to be returned. (Note that if you want to return a value other than a string, you need to use a FUNCTION, not a procedure.) In the caller you use the RETURN-VALUE function to retrieve the return value. Multiple RETURN statements within the same procedure are allowed. The following example code shows a return value being set in proc1, and the value retrieved in the caller:

|  |
| --- |
| /\* myprocedures.p \*/ PROCEDURE proc1:  /\* ABL code \*/  MESSAGE "In proc1".  **RETURN "1".**  END PROCEDURE.  PROCEDURE proc2:  /\* ABL code \*/ MESSAGE "In proc2".  END PROCEDURE.  RUN proc1.  MESSAGE "The return value from proc1 is" **RETURN-VALUE**. |

For more information, see RETURN statement and RETURN-VALUE in *Develop ABL Applications*.

###### Use parameters with procedures

To pass values in or out of a procedure you can define parameters using the DEFINE PARAMETER statement. The parameter definition names the parameter, specifies whether it receives input or provide output, or both, and specifies the data type. This is the simplified syntax for the DEFINE PARAMETER statement:

|  |
| --- |
| DEFINE **{** INPUT **|** OUTPUT **|** INPUT-OUTPUT **}** PARAMETER *parameter-name* AS *datatype* **[**  NO-UNDO **] [** INITIAL *initial-value* **]** |

INPUT | OUTPUT | INPUT-OUTPUT

* + INPUT - value is passed in to the procedure you are running.
  + OUTPUT - value is returned to the caller when the called procedure completes.
  + INPUT-OUTPUT - value is passed in to the procedure, which can modify the value. The AVM passes the value back to the caller when the procedure ends.

parameter-name

The name of the parameter.

datatype

The datatype of the parameter. You can use any one of the standard ABL data types, including CHARACTER, INTEGER, DECIMAL, LOGICAL, DATE. You can also specify TABLE, DATASET,

TABLE-HANDLE, or DATASET-HANDLE, but these use a slightly different syntax.

initial-value

The initial value of the parameter. This only applies to OUTPUT parameters.

You specify parameters in the RUN statement to pass a parameter from the calling procedure to the called procedure. When you specify the parameter, you also specify the kind of parameter you are using, (INPUT, OUTPUT, or INPUT-OUTPUT). INPUT is the default, but a best practice is to specify the kind of parameter so that your code is unambiguous to other developers.

You must ensure that when you call a procedure that takes parameters, the number and order of parameters match, as well as their ABL data types, and input/output types. In the calling procedure, when you use variables to pass values to and from a procedure, the names of the variables need not match the names used in the procedure.

For more details, see "Parameter definition syntax" and "Parameter passing syntax" in the *ABL Reference*

guide.

The following example code demonstrates passing parameters:

|  |
| --- |
| VAR CHAR v1 = "XXX". VAR CHAR v2 = "YYY". VAR CHAR v3 = "ZZZ".  /\* proc2 \*/ PROCEDURE proc2:  DEFINE INPUT PARAMETER p1 AS CHARACTER NO-UNDO. DEFINE OUTPUT PARAMETER p2 AS CHARACTER NO-UNDO.  DEFINE INPUT-OUTPUT PARAMETER p3 AS CHARACTER NO-UNDO.  p1 = "XXXXXX". p2 = "YYYYYY". p3 = "ZZZZZZ".  MESSAGE "In proc2: p1, p2, p3:" p1 p2 p3.  END PROCEDURE.  /\* main \*/  MESSAGE "Before running proc2: v1, v2, v3:" v1 v2 v3. RUN proc2 (INPUT v1, OUTPUT v2, INPUT-OUTPUT v3).  MESSAGE "After running proc2: v1, v2, v3:" v1 v2 v3. |

The following output is produced from the example code:

|  |
| --- |
| Before running proc2: v1, v2, v3: XXX YYY ZZZ In proc2: p1, p2, p3: XXXXXX YYYYYY ZZZZZZ  After running proc2: v1, v2, v3: XXX YYYYYY ZZZZZZ |

###### Persistent procedures

A persistent procedure is an instance of a procedure that stays resident in the AVM's memory until it is explicitly deleted. You can use persistent procedures as a “memory cache” of frequently used procedures and functions. The procedure maintains its state. For example, if an internal procedure you call sets a variable at the .p (main block) level, that value stays set and can then be used from a different internal procedure.

Since the procedure stays in memory, the AVM does not have to load the program into memory each time it is called. Thus, there is a performance benefit to your application.

To instantiate a persistent procedure in memory, you use the RUN statement with the keyword PERSISTENT. You also define a handle to the procedure so that you access it later.

To instantiate the persistent procedure, use the following syntax:

|  |
| --- |
| RUN *proc-name* PERSISTENT SET *proc-handle-name*. |

**User-defined functions**

In the following example code, we define the procedure handle variable, hEmpLibrary and instantiate the persistent procedure setting the procedure handle variable. You must ensure that the procedure you run is in your PROPATH at runtime.

|  |
| --- |
| VAR HANDLE hEmpLibrary.  RUN emplibrary.p PERSISTENT SET hEmpLibrary. |

Once the persistent procedure has been instantiated, you can run any of its internal procedures in the library. You use the procedure handle to reference its internal procedures.

|  |
| --- |
| RUN *proc-name* IN *proc-handle-name* [ (*parameter-list*) ]. |

In the following example code, we define the procedure handle variable, hEmpLibrary and instantiate the persistent procedure in the RUN statement. In the FOR EACH statement, we call the internal procedure calcvacation for every employee and display the information.

|  |
| --- |
| /\* eCheckEmpVacAvailable.p \*/ VAR DATE dtStart.  VAR DATE dtEnd.  VAR LOGICAL lOK.  **VAR HANDLE hEmpLibrary.**  **RUN emplibrary.p PERSISTENT SET hEmpLibrary.**  FOR EACH Employee: lOK = FALSE.  **RUN calcvacation IN hEmpLibrary(INPUT EMPLOYEE.EMPNUM, INPUT dtStart, INPUT dtEnd, OUTPUT lOK).**  DISPLAY Employee.FirstName Employee.LastName lOK label "Vac?".  END. |

###### Organize procedure files

There are multiple ways of organizing your procedure files. A good practice is to create procedure files of related functionality (libraries), which contain just internal procedures. You can then make these libraries available to other parts of your application. This promotes re-usability and makes it easier to maintain your application.

Another good practice is to separate user interface (UI) logic from business logic. The UI logic could be a mobile or web app, or an ABL GUI desktop client. Separating UI logic from business logic makes it easier to update the application because, in general, changes to one side of the application do not affect the other side. In a modern application, the UI logic does not directly access the database. The UI logic requests data from the business logic, which in turn, retrieves the data from the database. Exchanging data between the UI logic and the business logic is done using ABL temp-tables and datasets.

### User-defined functions

A user-defined function is a block of custom code in a procedure file that returns a value. It is similar to the built-in ABL functions. It is like an internal procedure, but with the requirement that it must return a value. A user-defined function can also define parameters that are used as input, output, or both.

Before you can use a user-defined function in your code, you must define it. The definition must precede the code where it is used. Here is the simplified syntax:

|  |
| --- |
| FUNCTION *function-name* RETURNS *type* **[** ( *parameter-list* ) **]** :  /\* ABL code \*/  RETURN *return-value*.  END FUNCTION. |

function-name

The name of the function.

type

The type of the return value.

parameter-list

Parameters are optional. A valid specification of parameters includes the parameter use (INPUT,

OUTPUT, INPUT-OUTPUT), its name, and its type.

return-value

The return value of the function. The value must match the type specified in *type*.

In the following example code we define two functions, OrderPrefix() and GeneratePO(). The function, OrderPrefix(), takes no parameters and returns the prefix for the purchase order (PO) as a CHARACTER type. The GeneratePO() function has two parameters and constructs a PO number using these parameters.

|  |
| --- |
| /\* eCustomFunctions.p \*/  FUNCTION OrderPrefix RETURNS CHARACTER(): RETURN "PO".  END FUNCTION.  FUNCTION GeneratePO RETURNS CHARACTER (INPUT pcString AS CHARACTER, INPUT-OUTPUT piNumber AS INTEGER):  VAR CHAR cResult.  cResult = pcString + STRING(100 \* piNumber). piNumber = piNumber + 1.  RETURN cResult.  END FUNCTION.  DEFINE VARIABLE iNum AS INTEGER NO-UNDO INITIAL 999.  /\* main procedure code \*/  MESSAGE "Purchase Order number is:" GeneratePO(OrderPrefix(),INPUT-OUTPUT iNum) SKIP "New iNum is: " iNum VIEW-AS ALERT-BOX. |

You can pass a function as a parameter to another function. In the example code, you see the OrderPrefix() function being passed as the first parameter to the GeneratePO() function. Running the code produces the following output:

|  |
| --- |
| Purchase Order number is: PO99900 New iNum is: 1000 |

# 4

## Compile and Run

In the following topics, you learn what r-code is and how to compile and run your code. For details, see the following topics:

* [R-code and the AVM](#_bookmark30)
* [COMPILE statement](#_bookmark31)
* [RUN statement](#_bookmark34)

* [Run using the command line](#_bookmark35)

### R-code and the AVM

R-code is the intermediate language to which ABL source is compiled. Once ABL has been converted to r-code, it can be run across different platforms using the ABL Virtual Machine (AVM). The AVM is the environment where ABL applications are executed. The AVM has its own directory path list, called the PROPATH, to define the directories to search and the search order when running ABL code. For more information, see "R-code Features and Functions" in*Manage ABL Applications*.

### COMPILE statement

The COMPILE statement compiles a procedure file or a class definition file into r-code. A compilation can be kept in memory during a session, or you can save it permanently for use in later sessions (as an r-code file, which has a .r extension). When you compile a class definition file, the AVM compiles the class definition file identified in the compile statement and all class files in its inherited class hierarchy, by default.

Basic usage

|  |
| --- |
| COMPILE helloworld.p SAVE. |

Running the code example produces a file named helloworld.r.

#### Set compiler options

You can change the behavior of the compiler, and enforce certain rules, by specifying compiler options. Using compiler options allows you to identify and fix issues in your code, making your code unambiguous and more robust. Rules you can specify include requiring full table and field names, fully qualified buffer references, full keywords, and return values for user-defined functions, non-VOID methods, and property getters. You can optionally specify a severity level (warning or error) for each type of option.

|  |  |
| --- | --- |
| **Option name** | **Description** |
| require-full-names | All table and field names must appear as they are in the schema. The compiler's ability to implicitly resolve abbreviated table names is disabled. |
| require-field-qualifiers | All buffer references (including database tables, temp-tables, and buffers) must be fully qualified. The compiler's ability to implicitly resolve the buffer to which a field reference refers is disabled. |
| require-full-keywords | All ABL language keywords must be fully spelled out. |
| require-return-values | In user-defined functions, non-VOID methods, and property getters, all logical code paths must have RETURN *value* statements. For more information, see [Require-return-values compiler option](#_bookmark33) on page 61. |

###### Setting the severity level for a compiler option

There are two possible severity levels which can be set by the user: warning or error. The warning level produces a message for each violation, but allows the compilation to complete successfully, provided that the only violations are related to the option. The error severity level produces a message for each violation and prevents the compilation from completing successfully. If you specify an option without a severity level, the default is warning. If you don't specify a compiler option, then no violation checking is performed for that option.

To set a severity level, append a colon (:) to the option name, followed by the severity level:

|  |
| --- |
| *option-name***:***severity-level* |

*option-name*

Valid values are require-full-names, require-field-qualifiers, require-full-keywords, or require-return-values.

*severity-level*

Valid values are warning or error.

For example, the following are possible ways to specify severity levels for the require-return-values

compiler option:

|  |
| --- |
| require-return-values |

|  |
| --- |
| require-return-values:warning |

|  |
| --- |
| require-return-values:error |

###### Ways to set compiler options

There are four different ways to set compiler options, which are described below. The first two ways use the

COMPILE statement. The third way uses the COMPILER system handle, and the fourth way uses the

-compileroptionsfile startup parameter.

1. The OPTIONS phrase on the COMPILE statement.

When compiling a file in ABL, use the OPTIONS phrase to specify the desired compiler options. Separate each option with a comma, and enclose the list in quotes. For example:

COMPILE basic.p OPTIONS "require-full-names:error,require-field-qualifiers:error".

For more information on the OPTIONS phrase, see the "COMPILE statement" in *ABL Reference*.

1. The OPTIONS-FILE phrase on the COMPILE statement.

When compiling a file in ABL, use the OPTIONS-FILE *filename* phrase on the COMPILE statement to specify the name of a file that contains the options. *filename* should be an absolute path or a path relative to PROPATH. For example:

COMPILE basic.p OPTIONS-FILE myCompilerOptions.

The options file is a plain text file. Anything between a hash tag (#) and the end of line in the file is treated as a comment. The compiler options must be separated by commas even if on different lines. For example:

# myCompilerOptions

require-full-keywords, # no severity specified; warning is the default require-full-names:error,

require-field-qualifiers:error, require-return-values:warning

For more information on the OPTIONS-FILE phrase, see the "COMPILE statement" in *ABL Reference*.

1. The OPTIONS attribute on the COMPILER system handle.

Before compiling a file in ABL, set the OPTIONS attribute on the COMPILER system handle. For example:

COMPILER:OPTIONS = "require-return-values:warning". COMPILE basic.p.

COMPILER:OPTIONS = "". // reset to no compiler options

1. The -compileroptionsfile startup parameter.

When starting up the ABL client, use the -compileroptionsfile startup parameter, to specify a file containing the options. This sets the OPTIONS attribute to the values in the file.

-compileroptionsfile *filename*

The file is the same format as described in the second method for the OPTIONS-FILE phrase of the COMPILE

statement. For more information on this startup parameter, see "Compiler Options File

(-compileroptionsfile)" in *Open Edge Start-up Command and Parameter Referemce*.

**Require-return-values compiler option**

The require-return-values compiler option helps developers identify and fix problems, resulting in more robust code. The option helps ensure all logical code paths have RETURN *value* statements in ABL user-defined functions, non-VOID methods, and property getters. Missing RETURN *value* statements may lead to incorrect behavior and code that is hard to debug. Enabling the option during code development can flag potential issues and prevent problems down the road. The option may also be useful during troubleshooting to identify fragile code that does not behave as expected.

###### Enabling the option and identifying violations

To enable the feature, use the require-return-values compiler option. For information on enabling compiler options, see Set compiler options.

During compilation, if the option is enabled and a violation is detected, a message is displayed:

|  |
| --- |
| All code paths in user-defined function, method, or property getter "*name*" must return a value due to the "require-return-values" compiler option. (19822) |

The message identifies the name of the user-defined function, method, or property getter with the violation. It does not identify which path or line is missing the RETURN *value* statement. You need to examine your code to determine how to make the appropriate fixes.

###### User-defined function, non-VOID method, or property getter block

If there are no control flow statements within the user-defined function, non-VOID method, or property getter, then the main block of the user-defined function, non-VOID method, or property getter requires a RETURN *value* statement. The following example shows a user-defined function that is not compliant with the require-return-values compiler option:

|  |
| --- |
| /\* basic.p \*/  /\* Not compliant with require-return-values \*/ FUNCTION retint RETURNS INTEGER ():  MESSAGE "function retint".  END FUNCTION.  MESSAGE "function return" STRING(retint()). |

Add a RETURN *value* statement to clear the warning/error, as shown in the following example:

|  |
| --- |
| /\* basic.p \*/  /\* Compliant with require-return-values \*/ FUNCTION retint RETURNS INTEGER ():  MESSAGE "function retint".  **RETURN 1.**  END FUNCTION.  MESSAGE "function return" STRING(retint()). |

As long as the user-defined function, non-VOID method, or property getter ends with a RETURN *value* statement, then the code is compliant. If it does not end with a RETURN *value* statement, then you must ensure all other code paths are compliant.

Ensuring all other code paths provide a RETURN *value* statement involves control flow analysis of ABL programs.

###### Ensuring other code paths are compliant

As stated previously, you simply need to end each user-defined function, non-VOID method, or property getter with a RETURN *value* statement in order to make your code compliant. However, if this is not desired, then you can add RETURN *value* statements to other constructs. The following sections describe different statements that alter the flow from within ABL and how the require-return-values compiler option affects them.

Before and after code examples are shown, along with information on how to make the code compliant. The assumption is that the containing user-defined function, method, or property getter does not end with a RETURN *value* statement, so other statements and blocks must be made compliant instead.

###### IF…THEN…ELSE statement

In an IF...THEN...ELSE statement, both the IF and ELSE portions require a RETURN *value* statement. Having one within either, but not both, never satisfies the require-return-values option. If you only have an IF clause, then you must add an ELSE clause and include a RETURN *value* statement for both.

|  |
| --- |
| /\* Not compliant with require-return-values \*/ VAR LOGICAL boolVar = FALSE.  IF boolVar THEN RETURN 1. |

|  |
| --- |
| /\* Compliant with require-return-values \*/ VAR LOGICAL boolVar = FALSE.  IF boolVar THEN RETURN 1.  **ELSE**  **RETURN 0.** |

###### CASE statement

In a CASE statement, there must be an OTHERWISE clause that includes a RETURN *value* statement, and all

WHEN..THEN clauses must also contain a RETURN *value* statement.

|  |
| --- |
| /\* Not compliant with require-return-values \*/ VAR INT casetest = 3.  CASE casetest:  WHEN 1 THEN  RETURN 2.  WHEN 2 THEN  RETURN 4.  END CASE. |

|  |
| --- |
| /\* Compliant with require-return-values \*/ VAR INT casetest = 3.  CASE casetest:  WHEN 1 THEN  RETURN 2.  WHEN 2 THEN  RETURN 4.  **OTHERWISE** |

|  |
| --- |
| **RETURN 0.**  END CASE. |

###### Nested control flow statements

Control flow statements can appear within each other. As long as each inner statement or block is compliant, then the outer statement or block is also compliant. To satisfy the require-return-values compiler option, it must be the case that there are no statements after the block.

###### CATCH and FINALLY blocks

Enforcement of the required RETURN *value* statement is not done in either a CATCH or FINALLY block. The best practice is to not use a RETURN *value* in a CATCH or FINALLY block.

### RUN statement

Once you are running an AVM session, you can run a procedure from within another ABL procedure, using the RUN statement.

If you have not already created a .r file for the .p you are running, the AVM compiles it for you automatically and saves the r-code in memory. However, once the session is over, the r-code is no longer available and would have to be compiled again the next time the .p is run.

**Basic usage**

|  |
| --- |
| RUN helloworld.p. |

### Run using the command line

OpenEdge provides the Proenv utility to run OpenEdge command line tools. To run a procedure from the command line, use the Startup Procedure (-p) parameter. This starts a new AVM and runs the file specified by the -p parameter.

Basic usage

|  |
| --- |
| prowin -p helloworld.p. |

See Use the Proenv utility for more detail.

# 5

## Work with the OpenEdge Database

In the following topics, you learn important concepts and ABL language constructs for working with the OpenEdge database. You first learn how to access tables and buffers. Then you learn how to construct queries to limit the data retrieved. You also learn how to manage transactions and how to handle record locking. Finally you learn how to update records.

For details, see the following topics:

* [Tables and buffers](#_bookmark37)
* [Data access](#_bookmark38)
* [Queries](#_bookmark39)
* [Transactions](#_bookmark40)
* [Record locks](#_bookmark41)

* [Update records](#_bookmark42)
* [Online database changes using ABL](#_bookmark43)

### Tables and buffers

Tables

A database table is a collection of logically related data treated as a unit. Tables contain rows and columns. All rows in a table comprise the same set of columns (fields). Tables often have indexes.

The OpenEdge database follows the relational model which organizes data into tables. Relationships between tables are defined by the data, that is, matching values in a field common to both tables. See Elements of a relational database for more detail.

In ABL, you use database access statements such as FIND statements, FOR EACH blocks, REPEAT FOR blocks, or DO FOR blocks to access the records in a table. If a database table name is specified in any of these statements, a record buffer for the table is automatically created and is populated with the current record’s data.

When you retrieve a record from the database the AVM keeps track of the current record position using an index cursor—a pointer to the record.

Buffers

Whenever you reference database tables using a database access statement, Record buffers for the table are created automatically for your convenience. The record buffer is a temporary storage area in memory where the AVM manages records as they pass between the database and the statements in your code. This default record buffer has the same name as the database table. This lets you think in terms of accessing database records directly because the name of the buffer is the name of the table the record comes from.

You can create additional buffers for a table when you need to have two or more different records from the same table available to your code at the same time. For example, when looking for duplicate records, you can read a record into the default buffer and then use another buffer to go through the rest of the records for the table. If one matches then you know you have a duplicate.

### Data access

ABL has four statements you can use to define a set of one or more database records: FIND, FOR, REPEAT, and OPEN QUERY. FIND fetches a single record when the statement is called. FOR , REPEAT, and OPEN QUERY identify a set of records. You can then loop through the records one at a time.

In this topic, we introduce FIND, FOR, and REPEAT. You learn about OPEN QUERY in the next topic on queries. In this topic, you also learn about ROWID, a unique internal identifier for records.

###### FIND statement

The FIND statement is a very powerful way to retrieve individual records from the database without having to set up a query or result set definition. FIND fetches a single record from a database table and moves that record into a record buffer. The basic (simplified) syntax for the FIND statement is:

|  |
| --- |
| FIND **[** FIRST **|** LAST **|** NEXT **|** PREV **]** *record*  **[** WHERE *expression* **]** |

record

Database table name.

FIRST | LAST | NEXT | PREV

Finds the first, last, next or previous record in the table that meets the specified characteristics.

WHERE *expression*

Restricts the query to only those rows in the database table that match the specified expression.

**Data access**

In the following example code, the first record from the Customer table, whose name starts with "A", is retrieved:

|  |
| --- |
| FIND FIRST Customer WHERE Customer.Name BEGINS "A". DISPLAY Customer.Name. |

For more information, see Data access without looping: the FIND statement in *Develop ABL Applications*.

###### FOR block

The FOR block iterates through a set of related records in the database table and moves each one in turn into the record buffer. The block iterates through all records that match the specified criteria and for as many iterations as requested. The FOR statement is very powerful and includes options for sorting, collating, and record-locking. When the block begins, the AVM evaluates the expression and retrieves the first record that satisfies it. This record is scoped to the entire block. Each time the block iterates, the AVM retrieves the next matching record and makes it available to the rest of the block. When the set of matching records is exhausted, the AVM automatically terminates the block. You don't have to add any checks or special syntax to exit the block at this point. The FOR block ends with an END statement. The basic (simplified) syntax of the FOR block is:

|  |
| --- |
| FOR **[** EACH **|** FIRST **|** LAST **]** *record* **[** WHERE *expression* **]**:  /\* ABL statements \*/ END. |

EACH | FIRST | LAST

* + EACH starts an iterating block, finding a single record on each iteration.
  + FIRST finds the first record.
  + LAST finds the last record.

record

Database table name.

WHERE *expression*

Restricts the query to only those rows in the database table that match the specified expression.

The following FOR EACH block displays the Customer.Name field for every record in the Customer table where the Customer.CustNum field is less than 100:

|  |
| --- |
| FOR EACH Customer WHERE Customer.CustNum < 100: DISPLAY Customer.Name.  END. |

###### REPEAT block

The REPEAT block is a set of statements that are processed repeatedly but it does not automatically read records as it iterates. This block lets you navigate through a set of records yourself, rather than simply proceeding to the next record automatically on each iteration. The REPEAT block ends with an END statement.

Often a REPEAT statement statement is used with a PRESELECT phrase to select the records that meet the criteria you specify. PRESELECT creates a result list of ROWID (unique identifier) values, so that records are then retrieved by ROWID.

Typically a FIND statement is used within a REPEAT (PRESELECT) block to read a record on each iteration. There are some best practices to follow when using a REPEAT block:

1. Use the NO-ERROR qualifier on the FIND statement. This suppresses the error message that you would ordinarily get when you are at the last record.
2. Use the AVAILABLE function to check for the presence of a record. Provide a matching ELSE statement to LEAVE the block when there is no record available.

The following example code uses a REPEAT block to loop through the customer records where the Country is “USA”. When the last matching record is read, the LEAVE statement breaks out of the REPEAT block.

|  |
| --- |
| FIND FIRST Customer NO-LOCK WHERE Customer.Country = "USA". DISPLAY Customer.CustNum Customer.Name Customer.Country.  REPEAT:  FIND NEXT Customer WHERE Customer.Country = "USA" NO-LOCK NO-ERROR. IF AVAILABLE Customer THEN  DISPLAY Customer.CustNum Customer.Name Customer.PostalCode.  ELSE LEAVE.  END. |

###### ROWID

Every database record has a unique internal identifier known as the ROWID. This identifier has the data type,

ROWID. You use the ROWID function to retrieve the ROWID of the database record currently in the record buffer.

The following example demonstrates using the ROWID function to retrieve the identifier of the record currently in the buffer. The identifier is later used to re-fetch the record with locking so it can be updated.

|  |
| --- |
| VAR ROWID custrid.  FIND FIRST Customer NO-LOCK.  custrid = ROWID(Customer). // Get the rowid and save it, so it can be refetched.  IF Customer.balance > 0 THEN DO:  FIND Customer WHERE ROWID(Customer) = custrid EXCLUSIVE-LOCK.  Customer.Comments = "Balance remaining". // Update the Comments field in the Customer record  DISPLAY Customer.Name Customer.Balance Customer.Comments FORMAT "X(20)". RELEASE Customer.  END. |

### Queries

A query defines a set of data to retrieve from the database. It provides similar functionality as the data access blocks DO, FOR, and REPEAT. The main difference is that the result set defined by a query is not scoped to the block where it is defined. Using the handle to a query, you can access the query and its result set from anywhere in your application. This gives you the ability to modularize your application in ways that can't be done with block-oriented result sets.

Queries give your data access language these important characteristics:

* Scope independence — You can refer to the records in the query anywhere in your application.
* Record retrieval independence — You can move through the result set under complete control of either program logic or user events.

**Queries**

* Repositioning flexibility — You can position to any record in the result set at any time.

To get a query to retrieve data, you need to open it. When you are done with a query you should close it to free the system resources used by the query.

###### Static queries

Static queries are used when the definition of the query is known during development. You define a static query using the DEFINE QUERY statement to create the details for the query. The query must be opened with an OPEN QUERY statement before it can be used.

The following code defines and opens a query. It then cycles through all of the customer records and counts them as it goes.

|  |
| --- |
| VAR INT iCount.  DEFINE QUERY qCust for Customer.  OPEN QUERY qCust FOR EACH Customer. GET FIRST qCust.  DO WHILE AVAILABLE Customer:  iCount = iCount + 1. GET NEXT qCust.  END.  DISPLAY iCount.  CLOSE QUERY qCust. |

###### Dynamic queries

Dynamic queries are used when the definition of the query is not known until runtime. For example you might want the user to input a query string. To construct a dynamic query you define a handle variable for the query and you use the CREATE QUERY statement to create an empty query at runtime. You then set the buffers using the SET-BUFFERS method and then prepare the query using the QUERY-PREPARE( ) method, where you pass the dynamic query string as a parameter. Finally, the query must be opened using the QUERY-OPEN(

) method before it can be used.

|  |
| --- |
| VAR HANDLE hQuery.  DEFINE INPUT PARAMETER bufHandle AS HANDLE. DEFINE INPUT PARAMETER qryString AS CHARACTER.  CREATE QUERY hQuery.  hQuery:SET-BUFFERS(bufHandle). hQuery:QUERY-PREPARE(qryString). hQuery:QUERY-OPEN(). |

For more information, see Using Queries in *Develop ABL Applications* and Query object handle in the *ABL Reference*.

###### GET statement

The GET statement returns one record, and optionally related records, from an opened query.

Syntax

|  |
| --- |
| GET { FIRST | NEXT | PREV | LAST | CURRENT } query [ SHARE-LOCK | EXCLUSIVE-LOCK | NO-LOCK ]. |

FIRST | NEXT | PREV | LAST | CURRENT

* + FIRST returns the first record from the query.
  + NEXT returns the first or next record from the query.
  + PREV returns the preceding or last record from the query.
  + LAST returns the last record from the query.
  + CURRENT refreshes the current record or records from the query.

query

The name of the query.

SHARE-LOCK | EXCLUSIVE-LOCK | NO-LOCK

The specified lock is applied to the record. Overrides the default locking of the OPEN QUERY statement.

### Transactions

Transactions ensure that the data in your database maintains integrity. In this topic, you learn about implicit and explicit transactions, and how the UNDO statement undoes a transaction.

###### Implicit transactions

During a transaction, information on all database activity occurring during that transaction is written to a before-image (or BI) file that is associated with the database. The BI file is located on the server with the other database files. The information written to the before-image file is coordinated with the timing of the data written to the actual database files. That way, if an error occurs during the transaction, the AVM automatically uses the before-image file to restore the database to the condition it was in before the transaction started.

The statements which start an implicit transaction are:

* FOR blocks that directly update the database
* REPEAT blocks that directly update the database
* Procedure blocks that directly update the database
* DO blocks with the ON ERROR phrase that contain statements that update the database

The following example code starts a transaction for each iteration, resulting in the updates for each customer record as a separate transaction.

|  |
| --- |
| FOR EACH Customer:  /\* Customer update block \*/ END. |

If your application has multiple nested blocks, each of which would be a transaction block if it stood on its own, then the outermost block is the transaction and all nested transaction blocks within it become subtransactions. All database activity occurring during a subtransaction is written to a local-before-image (or LBI) file.

**Transactions**

A subtransaction block can be:

* A procedure block that is run from a transaction block in another procedure
* Each iteration of a FOR EACH block nested within a transaction block
* Each iteration of a REPEAT block nested within a transaction block
* Each iteration of a DO TRANSACTION or DO ON ERROR inside a transaction block

If an application error occurs during a subtransaction, all the work done since the beginning of the subtransaction is undone. You can nest subtransactions within other subtransactions. You can use the UNDO statement to programmatically undo a transaction or subtransaction.

The following example code starts a transaction at the Customer level and Order changes are done in a subtransaction:

|  |
| --- |
| FOR EACH Customer: // Starts a transaction  /\* Customer update block \*/  FOR EACH Order WHERE Order.CustNum = Customer.CustNum: // Starts subtransaction  /\* Order update block \*/ END.  END. |

###### Explicit transactions

You can also start a transaction by adding the TRANSACTION keyword to a DO, FOR, or REPEAT block. If your code starts a transaction in one procedure and then calls another procedure, whether internal or external, the entire subprocedure is contained within the transaction that was started before it was called. If a subprocedure starts a transaction, then it must end within that subprocedure as well, because the beginning and end of the transaction are always the beginning and end of a particular block of code.

The following example code puts a transaction block around the whole update of both the Customer and any modified Orders:

|  |
| --- |
| DO TRANSACTION:  DO:  /\* Customer update block \*/ END.  FOR EACH Order WHERE Order.CustNum = Customer.CustNum:  /\* Order update block \*/ END.  END. // TRANSACTION block |

###### UNDO statement

A transaction is automatically undone when an unhandled error occurs that kicks you out of the transaction block. Your application logic can also undo a transaction when you detect a violation within your business logic. The UNDO statement lets you control when to cancel the effects of a transaction on your own. It also lets you define just how much of your procedure logic to undo.

You can use the UNDO keyword as its own statement. In this case, the AVM undoes the innermost containing block with the error property, which can be:

* A FOR block
* A REPEAT block
* A procedure block
* A DO block with the TRANSACTION keyword or ON ERROR phrase

The basic syntax for the UNDO statement is:

|  |
| --- |
| UNDO **[** LEAVE **|** NEXT **|** RETRY **|** THROW *error-or-stop-object-expression*  **|** RETURN **[** ERROR **|** NO-APPLY **] [** return-value **] ] ]** |

In the following example code, a transaction is started at the Customer level and Order changes are done in a subtransaction:

|  |
| --- |
| FOR EACH Customer: // Starts a transaction  /\* Customer update block \*/  FOR EACH Order WHERE Order.CustNum = Customer.CustNum: // Starts a subtransaction  /\* Order update block \*/  /\* If validation fails, exit \*/  UNDO, LEAVE. // This undoes the subtransaction END.  END. |

You can also specify UNDO as an option on a DO, FOR, or REPEAT block. In the following example code, upon encountering an error, the current block is undone and execution resumes in the code following the block.

|  |
| --- |
| DO TRANSACTION ON ERROR UNDO, LEAVE: DO:  /\* Customer update block \*/ END.  FOR EACH Order WHERE Order.CustNum = Customer.CustNum:  /\* Order update block \*/ END.  END. // TRANSACTION block |

You can also specify UNDO, LEAVE *label*, if you want to leave a specified transaction block identified by

*label*, upon encountering the condition. See the LEAVE statement for more detail on using a label with LEAVE. For more information, see Managing transactions and Understanding the UNDO concept.

### Record locks

When you read records from a database table, the AVM applies a level of locking to the record that you can control, so that you can prevent conflicts where multiple users of the same data are trying to read or modify the same records at the same time. This locking does not apply to temp-tables since they are strictly local to a single ABL session and never shared between sessions. There are three locking levels:

* A NO-LOCK is a read-only option that might read incomplete transactional data.
* A SHARE-LOCK is held until the end of the transaction or the record release, whichever is later.
* An EXCLUSIVE-LOCK is held until the end of the transaction. It is then converted to a SHARE-LOCK if the record scope is larger than the transaction and the record is still active in any buffer. It is best to explicitly release the record after the update is complete.

When you read records using a FIND statement, a FOR EACH block, or the GET statement on a query, by default the record is read with a SHARE-LOCK. Another user can also read the same record using another SHARE-LOCK.

**Update records**

You can read records using a different lock level. If you intend to change a record, you can use the EXCLUSIVE-LOCK keyword. This marks the record as being reserved for your exclusive use. If any other user has a SHARE-LOCK on the record, an attempt to read it with an EXCLUSIVE-LOCK fails. Thus, a SHARE-LOCK assures you that while others can read the same record you have read, they cannot change it. You can also read a record using NO-LOCK.

When a transaction is undone, locks acquired within the transaction are released or they are changed to

SHARE-LOCK if it locked the records prior to the transaction.

Optimistic locking

In a traditional host-based or client/server application, you can enforce what is referred to as a pessimistic locking strategy. This means that your application always obtains an EXCLUSIVE-LOCK when it first reads any record that might be updated, to make sure that no other user tries to update the same record.

In a distributed application, this technique simply does not work. If you read and pass records to a client session, your server-side session cannot easily hold locks on the records while the client is using them. When the server-side procedure ends and returns the temp-table of records to the client, the server-side record buffers are out of scope and the locks released. In addition, you would not want to maintain record locks for this extended duration, as it would lead to likely record contention.

The best way to make sure you get the locking you want is to be explicit about it. Follow these two guidelines for using locks:

* Always start a transaction before reading records, even with NO-LOCK, if you are going to update it inside the transaction.
* Release records explicitly when you are done updating them with the RELEASE statement

The following example shows first retrieving a record from the Customer table using NO-LOCK. The record is later retrieved using EXCLUSIVE-LOCK.

|  |
| --- |
| VAR ROWID custrid.  FIND FIRST Customer **NO-LOCK** WHERE Customer.Balance > 1000.  /\* Get the rowid and save it, so it can be refetched. \*/ custrid = ROWID(Customer).  IF balance > 0 THEN DO TRANSACTION:  FIND Customer WHERE ROWID(Customer) = custrid **EXCLUSIVE-LOCK**.  /\* Update the Comments field in the Customer record \*/ Customer.Comments = "Balance remaining".  DISPLAY Customer.Name Customer.Balance Customer.Comments FORMAT "X(20)". RELEASE Customer.  END. |

For more information, see Handling Data and Locking Records in *Develop ABL Applications*.

### Update records

To add records to a database, use the CREATE statement which places a newly created record in the database. All fields in the record are set to the initial values specified in the schema. The CREATE statement causes any CREATE trigger associated with the table to execute. This trigger may set fields in the record to new values.

To delete records from a database, use the DELETE statement. The DELETE statement causes any DELETE

trigger associated with the table to execute.

Use the ASSIGN statement or the Assignment (=) statement to update a database record. The ASSIGN statement does not actually write records to the database until the end of a transaction, after the record goes out of scope, or after an explicit RELEASE.

The following code creates a new record in the Customer table:

|  |
| --- |
| CREATE Customer. // Create the record  Customer.Name = "myNewCustomer". // Assign values to fields Customer.Address = "123 Main St.".  RELEASE Customer. // Release the record  FIND FIRST Customer WHERE Customer.Name = "myNewCustomer". DISPLAY Customer.Name Customer.CustNum Customer.Address. |

For more information, see Adding and deleting records in *OpenEdge Programming Interfaces*.

### Online database changes using ABL

You can make certain changes to your database schema, to ABL triggers, and to some database fields, while the database remains online, thereby ensuring the continuous operation of your ABL application. Learn more about the types of online database changes you can make in the following topics:

* Change schemas while the database is running
* Change non-structural fields of \_File and \_Field online
* ABL online schema changes to triggers
* Change ABL triggers online
* Change Int to Int64 online
* Rename fields using ABL
* Drop a field online

# 6

## Temp-tables and Datasets

In the following topics, you learn about temp-tables and datasets (also known as ProDataSets). Temp-tables and datasets are important constructs for working with data in an OpenEdge database. Temp-tables are relational structures and allow you to define a set of data. Datasets are collections of one or more temp-tables and allow you to specify the relationships between tables. You can also easily convert temp-tables and datasets to XML and JSON formats for use in other applications.

For details, see the following topics:

* [Temp-tables](#_bookmark45)

* [ProDataSets](#_bookmark49)

### Temp-tables

A temporary table (temp-table) is a very important ABL construct that allows you to define a set of data. Temp-tables are relational-based structures, visible to the OpenEdge session that creates them, and only last the duration of the session. Temp-tables may represent data from one or more tables in your database or they may contain different data.

Use cases

There are two main use cases for temp-tables. You can use a temp-table to represent data from one or more tables in your database. In this use case you perform similar operations on the temp-table, as you would on a database table, however the database itself is not affected. This allows operations to take place in the application logic, without any involvement from the database server. You initially fill the temp-table with data from one or more tables in the database, You can then manipulate the data in the temp-table without tying up the database. The database can be updated with changes from the temp-table at a later time, although care must be given to data integrity since the database might have changed during this time.

The second use case is when you want to work with a set of local data and even pass this set of data to another procedure or session. You can think of a temp-table in this case as a columnar structure where each row uses the same local schema definition.

Static temp-tables

Static temp-tables are used when the schema of the table is known during development. You define a static temp-table using the DEFINE TEMP-TABLE statement to create the schema for the temp-table. You create a record in the temp-table using the CREATE statement. The following example code defines a temp-table called ttCustomer with two fields (columns), creates a record, assigns values to the fields, and displays the record.

|  |
| --- |
| **DEFINE TEMP-TABLE** ttCustomer NO-UNDO // Define the temp-table FIELD CustName AS CHARACTER  FIELD CustId AS CHARACTER.  **CREATE** ttCustomer. // Create a record  CustName = "John Smith". // Assign values to the fields CustId = "98765".  DISPLAY ttCustomer. // Display the record |

You can pass a static temp-table to another procedure using the TABLE parameter. In this case the table data is copied from one procedure to the other. Static temp-tables require a complete, static definition of the table on each side of the transfer, because the schema is not passed as part of the parameter. For more information see Using a temp-table as a parameter.

Dynamic temp-tables

Dynamic temp-tables are used when the schema of the table is not known until runtime. You use the CREATE TEMP-TABLE statement to create an empty temp-table at runtime. You then define the schema using ADD methods such as ADD-NEW-FIELD( ) and ADD-NEW-INDEX( ). Once the definition for the dynamic temp-table is complete, you call TEMP-TABLE-PREPARE( ) to signal that the table definition is complete. The temp-table is now ready to hold data.

The following example code demonstrates creating a dynamic temp-table.

|  |
| --- |
| VAR HANDLE tth.  VAR HANDLE bufferh.  /\* Create an empty, undefined temp-table \*/  **CREATE TEMP-TABLE** tth.  /\* Add fields to the temp-table \*/  tth:**ADD-NEW-FIELD**("custName","character"). tth:ADD-NEW-FIELD("custNum","integer").  /\* Signal that the temp-table definition is complete and assign it the name "ttCust"  \*/  tth:**TEMP-TABLE-PREPARE**("ttCust").  /\* Get the buffer-handle for the temp-table \*/ bufferh = tth:DEFAULT-BUFFER-HANDLE.  /\* Create a record and store it in the buffer \*/ bufferh:BUFFER-CREATE().  /\* Assign values \*/  bufferh:BUFFER-FIELD("custName"):BUFFER-VALUE = "John Smith". bufferh:BUFFER-FIELD("custNum"):BUFFER-VALUE = "12345".  ... |

To pass a dynamic temp-table you pass the handle itself. You can also pass a dynamic temp-table using the

TABLE-HANDLE parameter. For more information on parameter passing, see Parameter passing syntax.

**Temp-tables**

For more information on temp-tables, see Defining and Using Temp-tables and Temp-table object handle.

#### Include files for temp-table definitions

An ABL include file is a file that contains ABL code that is included in another procedure or class when the procedure or class is compiled. The extension of an ABL include file is .i. You use include files to place common code in a separate file where the common code is typically shared by other procedures or classes in your application. A best practice is to place the definitions of a temp-table that is shared by user interface logic and business logic into an include file.

The following example code shows a temp-table definition placed in a separate file called ttOrder.i.

|  |
| --- |
| /\* ttOrder.i \*/  DEFINE TEMP-TABLE ttOrder NO-UNDO FIELD OrderNum AS INTEGER  FIELD OrderDate AS DATE FIELD ShipDate AS DATE FIELD PromiseDate AS DATE FIELD OrderTotal AS DECIMAL  INDEX OrderNum IS UNIQUE PRIMARY OrderNum. |

To use the include file in your code, surround the pathname of the include file with curly braces ({}) and place it in the code where you would like it to go. A temp-table definition would typically go near the beginning. You can specify a path relative to the PROPATH environment variable.

|  |
| --- |
| /\* myProcedure.p \*/  BLOCK-LEVEL ON ERROR UNDO, THROW.  **{ttOrder.i}**  /\* code to populate the ttOrder temp-table\*/ |

For more information, see { } Include file reference and Using include files to duplicate code.

#### Empty a temp-table

Static temp-tables remain for the life of the procedure where it is defined. It's a good practice to pay attention to scope of the temp-table and to empty it when it's no longer needed, or when you want a fresh set of records. Failing to empty it results in a temp-table that may contain records from the previous call, which might be unexpected and should be avoided. To empty a temp-table you use the EMPTY TEMP-TABLE statement. The basic syntax is:

|  |
| --- |
| EMPTY TEMP-TABLE *temp-table-name*. |

#### Copy records from a database to a temp-table

You can populate a temp-table by copying records into it from the database. You can copy the entire record or select fields. To copy a record from the database table to the temp-table you must:

1. Create a temp-table record.
2. Copy the record from the database (source) into the temp-table record (target).

Syntax for creating a temp-table record

The basic syntax for creating a temp-table record is the same as for creating a database table record:

|  |
| --- |
| CREATE *ttName*. |

Where *ttName* is the name of the temp-table. This creates the temp-table record. See the CREATE statement for more detail.

Syntax for buffer-copy

Use the following syntax to copy a database record buffer into a temp-table record buffer:

|  |
| --- |
| BUFFER-COPY *source* TO *target*. |

See the BUFFER-COPY statement for more detail.

The following example code shows how to populate the ttCustomer temp-table from the database. Note that the BUFFER-COPY only copies the matching fields; fields in the database or temp-table that do not match the other are ignored.

|  |
| --- |
| DEFINE TEMP-TABLE ttCustomer NO-UNDO // Define the temp-table FIELD CustNum AS INTEGER  FIELD Name AS CHARACTER FIELD City as CHARACTER FIELD State as CHARACTER  FIELD Country as CHARACTER.  FOR EACH Customer WHERE Customer.Country = "USA": // Iterate through Customer table CREATE ttCustomer. // Create the temp-table record  BUFFER-COPY Customer TO ttCustomer. // Copy database record into temp-table record DISPLAY ttCustomer.  END.  /\* return the temp-table to the UI client \*/ |

### ProDataSets

A Progress DataSet, or ProDataSet, is also commonly referred to as a dataset. The ProDataSet is a very powerful construct that widens and extends the functionality of temp-tables. A ProDataSet object is basically a collection of one or more member temp-tables. It also optionally contains a collection of data relationships among the member tables.

The benefits of using ProDataSets include:

* Ability to group multiple temp-tables into a single dataset object.
* Ability to define relationships between those tables.
* Ability to associate a data source with each dataset or member temp-table.
* Ability to track changes while manipulating the dataset data.
* Ability to save the changed data back to the data source.
* Ability to pass the dataset as a single parameter from one procedure to another, within a single ABL session or between sessions.
* Simplified management of complex transactions.

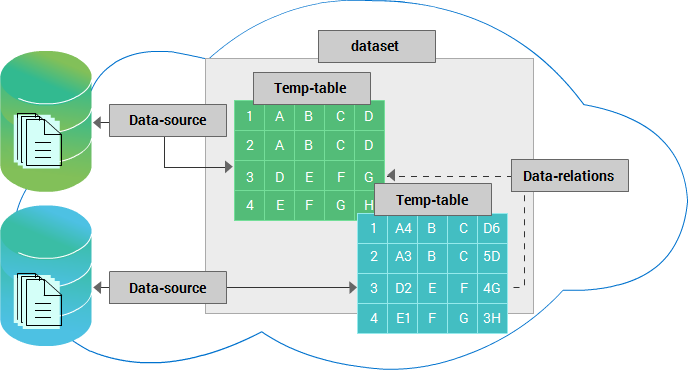
For more information, see Introduction to the OpenEdge DataSet in *Use ProDataSets*.

#### ProDataSet concepts

A dataset is a single, in-memory business object composed of ABL elements including temp-tables, data relationships, sources of data, and events.

A dataset often specifies the relationships between the component tables. In addition, it may be attached to data sources that can be used to populate the dataset with data. Changes to data in the dataset may then be stored back in the data sources. A dataset provides a mapping between a set of database tables or other data sources and their in-memory, possibly remote, representation.

Figure 2: ProDataSet



For more information, see ProDataSet goals and Architecture in *Use ProDataSets*

#### Define and use a dataset

The main steps for defining and using a dataset are:

1. Define a temp-table for each table you want to use in the dataset.
2. Define the dataset by specifying the temp-tables and their relationships (if any).
3. Identify, define, and attach the data-sources to populate the dataset.
4. Populate the dataset with data.
5. Update the dataset data and sync it back to the data-source.
6. Cleanup after using a dataset.

The first two steps are discussed next in [Define a dataset](#_bookmark52) on page 80. The rest of the steps are discussed in the Datasets Guided Journey.

##### Define a dataset

This main steps for defining a dataset are:

1. Define a temp-table for each table you want to use in the dataset.
2. Define the dataset by specifying the temp-tables and their relationships (if any).

###### Define the temp-tables

To define a temp-table you use the DEFINE TEMP-TABLE statement. When defining a temp-table manually, you specify each field and index in the temp-table definition. You can define each field as a specific data-type, as you would in a DEFINE VARIABLE statement. To define a temp-table with specific fields and indexes, use this syntax:

|  |
| --- |
| DEFINE TEMP-TABLE *table-name* **[** NO-UNDO **]** BEFORE-TABLE *before-table-name*  **{** FIELD *field-name* AS *data-type* **}**  **{** INDEX *index-name* **[** IS **[** UNIQUE **] [** PRIMARY **] ] {** *index-field* **} }**. |

The before table is used when a dataset is updated so that the original and updated data can be stored in the dataset before it is committed to the database.

Here is an example of the DEFINE TEMP-TABLE statement:

|  |
| --- |
| /\* ttOrder.i \*/  DEFINE TEMP-TABLE ttOrder NO-UNDO BEFORE-TABLE bttOrder FIELD OrderNum AS INTEGER  FIELD OrderDate AS DATE FIELD ShipDate AS DATE FIELD PromiseDate AS DATE FIELD OrderTotal AS DECIMAL  INDEX OrderNum IS UNIQUE PRIMARY OrderNum. |

###### Define the dataset

After you define the temp-tables, you use the DEFINE DATASET statement to define the dataset that is composed of those temp-tables. Similar to defining temp-tables in their own include files, you define a dataset in its own include file. The DEFINE DATASET statement is a complex statement with multiple components.

You use it to:

* Name the dataset and specify the temp-tables that comprise the dataset.
* Define any data-relations between those temp-tables.

To define a dataset, you use the DEFINE DATASET statement. Here is the simplified syntax:

|  |
| --- |
| DEFINE DATASET *dataset-name* FOR *temp-table-name* **[**,*temp-table-name* **]**... **[** DATA-RELATION **[***data-relation-name***]** FOR *parent-temp-table-name*,  *child-temp-table-name*  RELATION-FIELDS (*parent-field1*, *child-field1* **[**, *parent-fieldn*, *child-fieldn*  **]**...) **]** |

dataset-name

Specifies the name of the dataset.

temp-table-name

Specifies the name of the temp-table(s) in the dataset.

data-relation-name

Specifies a data-relation object. See ProDataSet relations for more detail.

parent-temp-table-name, child-temp-table-name

Identifies the parent and child temp-tables for the data relation.

parent-field*n*, child-field*n*

Define the relationship between fields in the temp-tables. For the most efficient joins, the

RELATION-FIELDS fields should be indexed.

The following example defines a dataset called dsOrderOrderLine. It is comprised of two temp-tables called ttOrder and ttOrderLine. It states that the relationship between the two tables is based upon the ordernum fields. In this example the fields in the two tables have the same name, but they don't need to be. You can also specify relation-fields that have different names.

|  |
| --- |
| /\* dsOrderOrderLine.i \*/  {include/ttOrder.i}  {include/ttOrderLine.i}  DEFINE DATASET dsOrderOrderLine FOR ttOrder, ttOrderline DATA-RELATION drOrderOrderLine FOR ttOrder, ttOrderline RELATION-FIELDS (ordernum,ordernum). |

You can define more than one data-relation for a dataset and include more than two temp-tables. The following is an example:

|  |
| --- |
| /\* dsOrderOrderLineItem.i \*/  {include/ttOrder.i}  {include/ttOrderLine.i}  {include/ttItem.i}  DEFINE DATASET dsOrderOrderLineItem FOR ttOrder, ttOrderLine, ttItem DATA-RELATION drOrderOrderLine FOR ttOrder, ttOrderLine  RELATION-FIELDS (Ordernum, Ordernum)  DATA-RELATION drOrderLineItem FOR ttOrderline, ttItem RELATION-FIELDS (Itemnum, Itemnum). |

You can learn more about datasets in the Datasets Guided Journey or Use ProDatasets guide.