

# Treinamento Progress

2018

## Programa para capacitação de desenvolvedores Progress Zallpy Group – 2018

Este programa tem como objetivo servir como material de apoio durante o curso de capacitação de desenvolvedores Progress, ministrado pela Zallpy Group.

### Sumário

Introdução ao Progress .....	5
Ambientes Progress .....	5
Conceito do banco de dados.....	6
Componentes do Progress .....	6
Criando o Banco de Dados .....	7
<b>Dicionário de dados</b> .....	7
<b>Data Administration</b> .....	7
<b>Definindo uma tabela</b> .....	8
<b>Tipos de dados permitidos e formatos de exibição padrão (default)</b> .....	9
<b>Definindo índices</b> .....	10
<b>Tipos de índices</b> .....	10
<b>Ler registros (<i>find</i>)</b> .....	12
<b>Ler registros com restrição <i>where</i> (<i>find</i>)</b> .....	12
<b>Ler registros por <i>for each</i></b> .....	13
<b>Leitura de registros com <i>no-lock</i></b> .....	15
<b>Atualização de registros com <i>exclusive-lock</i></b> .....	16
<b>Comparação entre <i>find</i> e <i>for each</i></b> .....	16
<b>Classificando registros por <i>by</i></b> .....	17
<b>Lendo registros com <i>begins</i></b> .....	18
<b>Lendo registros com <i>matches</i></b> .....	18
<b>Pesquisa por um índice informado</b> .....	19



Inserindo dados via <i>Procedure Editor</i> .....	20
Alterando dados via <i>procedure Editor</i> .....	21
Utilizando a função Repeat .....	21
Modificando registros com <i>For Each</i> .....	22
Excluindo registros ( <i>delete</i> ).....	23
Padrão de indentação adotado na Zallpy .....	24
Relacionamento entre tabelas .....	24
Pesquisa entre tabelas mãe e filha com o comando <i>of</i> .....	26
Utilizando a quebra <i>break by</i> .....	27
Utilizando a função <i>First-of / last-of</i> .....	27
<b>First-of</b> .....	28
<b>Last-of</b> .....	28
Operações com String .....	29
<b>Lookup</b> .....	29
<b>Entry</b> .....	30
<b>Can-do</b> .....	30
<b>Index</b> .....	30
<b>Num-entries</b> .....	30
If Can-find (first ou last) .....	30
Definindo e trabalhando com <i>temp-tables</i> .....	31
Buffer-copy.....	32
New shared e shared variables .....	34
Definindo procedimentos ( <i>procedures</i> ) .....	34
Definindo funções ( <i>function</i> ).....	35
Definição de <i>buffer</i> .....	36
<b>Utilizando buffer nas pesquisas</b> .....	36

Operações com arquivos.....	38
Utilizando o comando <i>input from</i> .....	38
Input from com Delimiter.....	38
Input from com comando unformatted .....	40
Utilizando comando <i>export</i> .....	40
Enviando dados para um arquivo com formato para leitura em <i>Excel</i> (padrão) .....	41
Sequences .....	41
Overview de Progress Gráfico .....	44

## Introdução ao Progress

### Ambientes Progress

Ambiente de desenvolvimento - para desenvolver as aplicações Progress.

Ambiente de execução de aplicações - ambiente para execução das aplicações desenvolvidas.

### Sumário Básico

Item	Descrição
.	(ponto final)
end	Fechamento de comando
Null	Fechamento de bloco
Then	Palavra Reservada
Do	Palavra Reservada
+	Palavra Reservada
" ou '	Utilizado na concatenação de Strings
: (dois pontos)	Aspas simples/aspas duplas são utilizadas para delimitar String
<	Início de um bloco "do"
<=	Representação para menor
>	Representação para menor ou igual
>=	Representação para maior
<>	Representação para maior ou igual
?	Representação para diferente
/*	Representa "Nulo"
*/	Abertura de comentário
By	Fechamento de comentário
*	Palavra reservada que implica em ordenação
*	Em Strings, é um caractere coringa

## Conceito do banco de dados

A definição de banco de dados pode ser descrita como um conjunto de registros dispostos em uma estrutura regular que possibilita sua reorganização e produção de informação. Um banco de dados agrupa registros utilizáveis para um mesmo fim, e são, portanto, um conjunto de arquivos relacionados entre si.

Um banco de dados é normalmente mantido e acessado por meio de software conhecido como SGDB (sistema gerenciados de banco de dados).

1. Banco de dados: Estrutura onde ficam armazenados os dados. Efetuando uma analogia, pode ser comparado a uma biblioteca.
2. Tabela: Estrutura onde são armazenadas as informações referente à um contexto, exemplo: tabela “Pessoa” guarda informações cadastrais das pessoas.
3. Registro: É um conjunto de dados, formado pelas colunas que compõe a tabela. Cada registro é único, e possui um valor de localização física (rowid).
4. Campo: São os campos (colunas) da tabela. Cada um possui um nome e um tipo que especifica qual tipo de informação irá guardar. Os tipos de dados disponíveis serão tratados posteriormente.
5. Índice – São estruturas complexas que permitem a criação de sumários em diferentes ordenações da tabela e/ou campos, possibilitando que o SGDB consiga, ao ler um índice, saber e conseguir acessar diretamente o registro através da sua localização física, não necessitando “procurar” o registro.

## Componentes do Progress

Os principais componentes do Progress são descritos a seguir:

1. Dicionário de dados - Define a estrutura das tabelas, registros, campos, índices e sequências utilizadas no banco de dados.
2. Linguagem - É a linguagem de programação propriamente dita, e que será usada para escrever os programas e aplicativos.
3. Compilador - Verifica a sintaxe de programa (programas .p, .w, triggers e etc), lê o esquema do banco de dados e transforma os programas em linguagem de máquina (os arquivos .r).
4. Editor - Permite criar, e editar os programas escritos na linguagem Progress 4GL.
5. Gerenciador do banco de dados - Manipula as interações com o banco de dados. Deltas, dumps e loads de tabelas, descritivos das tabelas e campos e etc.
6. Construtor de interface com o usuário - Utilizado para criar as aplicações gráficas usando o progress (interface utilizada no sistema GUS o progresss gráfico). Conhecida como UIB.
7. Includes – São componentes utilizados para importar outros códigos-fonte progress (no formato .i) para dentro do código atual. O uso de includes possui muitas vantagens, como manutenção centralizada, redução de código, otimização da

programação, como pelo uso de funções e variáveis, que podem ser declaradas somente na include.

8. Propath - O propath é uma lista de diretórios que define a ordem de prioridade em que aplicação procura os programas.
9. Programas – São arquivos em texto nos formatos .p, .w ou .i. O arquivo compilado, somente para interpretação da máquina é um .r.

## Criando o Banco de Dados

### Dicionário de dados

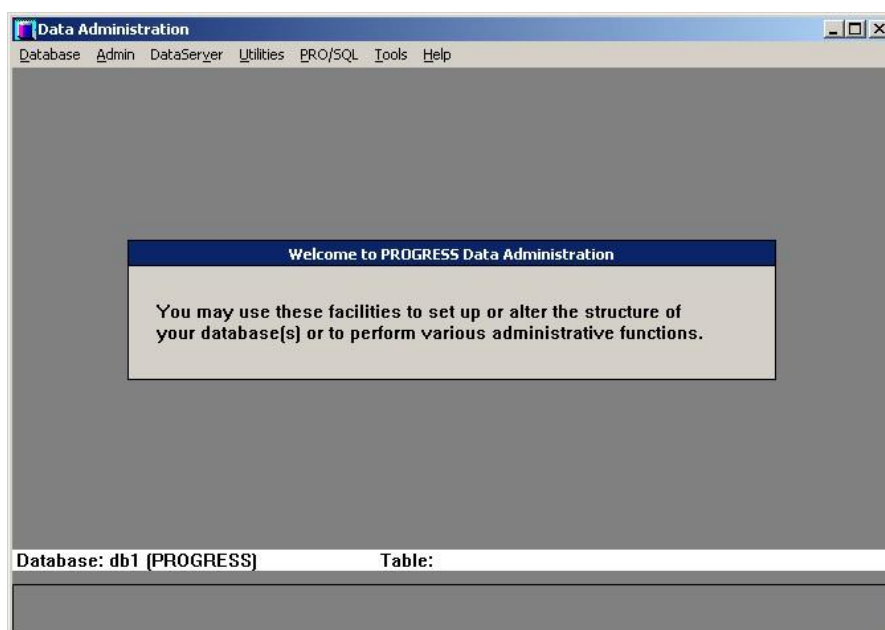
Abaixo o dicionário de dados acessado do ambiente Windows.



O dicionário de dados é a maneira mais fácil de se alterar, criar ou excluir tabelas, campos, índices e etc. As funções do dicionário de dados serão explicadas a medida da necessidade.

### Data Administration

O data administration é basicamente utilizado para que sejam feito “dump” e “load” das tabelas, deltas e descrições das tabelas para arquivos. Existem inúmeras outras funções que não serão abordadas neste curso. Para acessá-lo, entrar no menu “tools” no Dicionário de Dados ou no App Builder, e depois em “data administration”.



## Definindo uma tabela

1. Definir o nome da tabela: Escolher sempre nomes que lembrem o conteúdo da mesma. Escolha sempre palavras no singular como por exemplo: “empregado” e não “empregados”, “edifício” e não “edifícios”, “usuário” e não “usuários”.
  - a. Nomes podem ter até 32 caracteres de tamanho, devem iniciar com uma letra e não pode conter espaços no nome. É permitido o uso de hifens.
2. Definir os campos que existirão na tabela:
  - a. Defina nomes de campos fáceis de lembrar e de serem usados nos programas.
  - b. Nomes de campos podem conter até 32 caracteres e precisam começar com uma letra. É permitido também o uso de hífen, assim como os caracteres especiais: \$, &, - (hífen) e \_ (underline).
  - c. Para definição de campos em uma tabela pode-se ter como exemplo a tabela abaixo:

Tabela: Venda

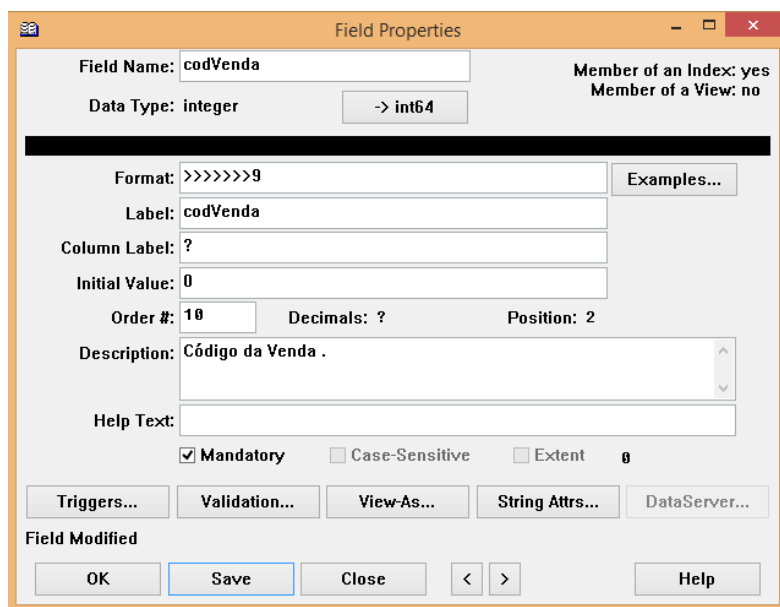
Conteúdo do campo	Nome do campo
Código da Venda	codVenda
Data da Venda	DatVenda
Lista de produtos	Prod-Ist
CPF do comprador	CPF
Valor total da venda	Valor
Quantidade de produtos na venda	qtdTotal

Usando como exemplo o campo codVenda, cada campo criado deve-se levar em consideração as seguintes variáveis:



- O nome do campo: codVenda
- O tipo de dados que será gravado no banco de dados: inteiro (integer)
- O tamanho do campo: 8 dígitos (no banco de dados este valor refere-se a máscara, trataremos este assunto mais tarde)
- O label para exibição: codVenda
- Se o campo é mandatorio (mandatory): se este campo não aceita nulo e deve ser informado na criação do registro.

Abaixo a figura da criação de um campo no dicionário de dados:



### Tipos de dados permitidos e formatos de exibição padrão (default)

Os exemplos abaixo correspondem aos valores que o Progress atribui automaticamente como valor padrão (default) para cada campo novo.

Tipo de dado	Formato de exibição default	Valor inicial default
character	x(8)	(branco)
inteiro	->, >>>, >>9	0
decimal	->, >>>9.99	0
date	99/99/99	?
logical	yes/no	no
recid	>>>>>>9	?

#### Observações:

- Conforme informado anteriormente, o formato de exibição do Progress não afeta o modo de armazenamento do banco de dados.
- A modificação no formato de um campo no dicionário de dados não afeta os dados que estão na tabela.
- Ao criar um campo como mandatório na criação do registro, não será possível criar este campo com valor desconhecido, isto é, nulo (?).
- O valor inicial informado na criação do campo na tabela será o valor inicial ao se efetuar a criação do registro na tabela.
- A ordem de exibição default é incrementada de 10. Você pode adicionar números de campos novos no fim ou inserir campos novos entre outros dois.
- Uma vez criado o campo no dicionário de dados não é possível alterar seu tipo de dado (criado como tipo lógico e depois querer alterar para inteiro). Para se alterar o tipo de dado de um campo deve-se excluir o campo e depois criá-lo novamente.

#### Definindo índices

Os índices são criados para que seja possível recuperar rapidamente um registro, ordenação automática dos registros, forçar a unicidade da chave de um registro e processar rapidamente o relacionamento entre tabelas.

Há um custo na criação de registros, pois a cada inserção de novo registro, o índice também é atualizado. Contudo, o uso de índices traz muitos benefícios, e em muitos casos (por exemplo, de tabelas grandes), é indispensável.

#### Tipos de índices

- Índice Primário – Corresponde à Chave primária (Primary Key - PK) da tabela. Toda a tabela deve possuir, obrigatoriamente, uma coluna (ou uma composição de colunas) cujo valor é único para todas as instâncias. É o índice que indica como a tabela é ordenada em seu armazenamento.
- Índices Secundários – Correspondem aos demais índices que são criados para uma tabela, os quais podem ter outros campos (que não os da PK) e outra ordenação. Quando criados, são armazenados em um arquivo e tem o propósito de indexar os dados no agrupamento/ordenação desejados, mapeando a localização de cada registro e facilitando o acesso nas buscas.

#### Chave estrangeira (Foreign Key - FK)

Uma chave estrangeira em um banco de dados é composta por um ou mais campos que apontam para a chave primária de outra tabela ou da mesma tabela e tem a finalidade de garantir a integridade dos dados referenciais, pois apenas serão permitidos valores que supostamente vão aparecer na base de dados.

Exemplo:

Tabela “pessoa”.

Campo “UF” só pode ser populado com um valor que seja válido (esteja presente) em uma tabela de cadastro de estados, exemplo, tabela “Estados”, que contém os dados dos estados brasileiros.

A nomenclatura adotada para o nome dos índices varia de empresa para empresa, sendo comum o uso do nome da tabela mais algarismos, ou apenas um nome genérico para o índice, como “idx-001”.

Primeiro índice da tabela Venda: idx001

Segundo índice da tabela Venda: idx002

Vamos usar como exemplo a tabela de “venda” para criação de 2 índices (abaixo as razões para criação dos dois índices):

Idx001 (campo utilizado no índice: codVenda)

- Recuperação rápida de um registro
- Ordenação automática dos registros
- Forçar a unicidade da chave de um registro
- Processar rapidamente o relacionamento entre as tabelas

Idx002 (campo utilizado no índice: datVenda – ordenação **Decrescente**)

- Recuperação rápida de um registro
- Ordenação automática dos registros

Definições dos índices da tabela Venda

Abaixo, as definições dos índices para a tabela Venda:

Nome do índice	Primário	Único	Ativo	Campo(s)	Características	
					Ascendente	Abreviado
Idx001	sim	sim	sim	codVenda	sim	não
Idx002	não	não	sim	datVenda	não	não

Observações:

- O índice primário é utilizado por default no processamento dos registros
- Normalmente deve-se ter pelo menos um índice único para cada tabela do banco de dados, não permitindo assim duplicação de dados iguais
- Um índice pode ter até 10 campos na sua composição

- Ascendente define a ordem de classificação do campo do índice
- Abreviado, apenas para campos tipo caractere, permite que o Progress ache um registro apenas usando os primeiros caracteres do campo
- Se você já sabe que sempre irá acessar uma tabela usando uma combinação de campos defina um índice com esta combinação para agilizar as suas pesquisas

### Ler registros (*find*)

O find é um comando utilizado para que seja possível recuperar (posicionar) um registro da tabela. O find deve sempre retornar apenas um registro, e é por isso que é sempre recomendado o uso de “first” ou “last” após o comando, pois assim, caso exista na tabela solicitada, dois registros que satisfazem as condições de busca, o Progress recupere/encontre apenas um deles (o first ou o last), não retornando erro.

Exemplo de utilização de find first (procura o primeiro registro seguindo o índice primário):

```
FIND first venda NO-LOCK NO-ERROR.  
IF AVAILABLE venda THEN DISP venda.
```

O find first encontra o primeiro empregado na tabela empregado e mostra na tela o nome do mesmo.

Exemplo de utilização de find last (procura o último registro seguindo o índice primário):

```
FIND last venda NO-LOCK NO-ERROR.  
IF AVAILABLE venda THEN DISP venda.
```

O find last encontra o último empregado da tabela empregado e mostra na tela o nome do mesmo.

Existe também as opções next e prev (próximo e antecessor) que busca registros conforme a ordenação dos mesmos na tabela.

### Ler registros com restrição *where* (*find*)

A restrição where serve para dar especificidade ao que se deseja encontrar.

Exemplos de find utilizando a condição where:

```
FIND first produto where produto.origem = "Brasil" NO-LOCK NO-ERROR.
IF AVAILABLE produto THEN DISP produto.nome.
```

```
FIND first produto where produto.origem <> "Brasil" and
                        Produto.vigente = yes      and
                        (Produto.tipo      = 1      or
                        Produto.tipo      = 2)
```

NO-LOCK NO-ERROR.  
IF AVAILABLE produto THEN DISP produto.nome.

Ler registros por *for each*

O bloco `for each` (em português, “para cada”) executa a busca com as condições solicitadas, e é capaz de percorrer todos os registros da tabela, antes de finalizar o comando. Esta leitura é feita em default pela ordem do índice primário da tabela.

Na execução do “for each”, deve-se usar um dos dois comandos a seguir:

- No-lock – Para consulta apenas
- Exclusive-lock – Para alteração da tabela

Existe também o “Share-lock”, porém, devido à sua implementação não totalmente tolerante à falhas, não recomendamos o uso.

Exemplo de um comando for each:

```
/* início do programa */
for each autor no-lock.
    disp autor.registro.
end.
/* fim do programa */
```

O script acima percorre toda tabela autor mostrando, para cada registro de autor, o seu número de registro profissional.

### Ler registros por *for each* com restrição *where*

O bloco `for each` utilizado em conjunto da condição `where` é a maneira mais comum de ser utilizado. Ele percorre a tabela e utiliza a condição especificada no `where` para trazer os dados.

Exemplos de utilização do for each com condição where:

```

/* início do programa */
for each produto where produto.origem = "Brasil" NO-LOCK.
    DISP produto.nome.
end.
/* fim do programa */

```

O programa acima mostra o nome de todos os produtos cadastrados na tabela produto, cuja origem é brasileira.

## Next

Uma outra maneira de se fazer um descarte condicional, sem utilizar a cláusula where, é utilizando a função next. Esta função é indicada para o descarte de registros que não satisfazem alguma condição que não pode (ou não deva) ser colocada em uma cláusula where. Vamos utilizar como exemplo a tabela Pessoa. Na tabela pessoa, possuímos apenas um índice, o índice idx001, cuja estrutura está listada abaixo:

Nome do índice	Primário	Único	Ativo	Campo(s)	Características	
					Ascendente	Abreviado
Idx001	sim	sim	sim	CPF	sim	não

O índice idx001 indica que a tabela está ordenada por CPF (character), em formato ascendente. Se desejarmos efetuar uma busca que traga, em ordem alfabética, apenas as pessoas que possuem o nome de “João”, seria ineficiente, pois não temos um índice para “nome”, esta busca seria menos eficiente se a condição fosse colocada na cláusula where, deste jeito:

```

/* início do programa */
for each pessoa where pessoa.nome = "João" NO-LOCK.
    DISP pessoa.nome.
end.
/* fim do programa */

```

Como não existe um índice por pessoa, e uma busca baseada em **condições de string** é mais pesada, pois há a necessidade de o SGBD acessar todos os registros existentes, além de ordenar a tabela por nome, é sugerido o uso do descarte com o next, ficando assim:

```

/* início do programa */
for each pessoa NO-LOCK.
    If pessoa.nome = "João" = no then next.
    DISP pessoa.nome.
end.
/* fim do programa */

```

Com o código acima, garantimos que, já que será necessário percorrer toda a tabela para verificar as pessoas com o nome “João”, a ausência de uma condição where permite que o banco de dados efetue uma leitura na tabela utilizando o índice primário (muito rápido), efetuando assim, para dos os registros, a condição “if”, descartando os registros que não interessam.

### Leitura de registros com *no-lock*

O tipo mais comum de leitura de tabelas no banco de dados é com a condição no-lock. Esta condição faz com que o registro lido não seja aberto com exclusividade para o usuário e não trave os outros usuários que estão utilizando o banco de dados e especificamente a mesma tabela. Esta funcionalidade é comumente utilizada quando os registros não serão alterados, e o acesso é apenas para leitura da tabela.

Exemplo desta funcionalidade abaixo:

```
/* início de programa */
for each pessoa no-lock.
    message pessoa.nome
        pessoa.sexo label "Sexo da Pessoa" view-as alert-box.
end.
/* fim do programa */
```

O programa mostra acima a leitura com no-lock da tabela pessoa utilizando uma funcionalidade de mensagem (“message”). A funcionalidade “message” é usada para mostrar alerta ao usuário pela informação de “view-as alert-box”, lhe solicitar confirmação por “view-as alert-box question buttons yes-no-cancel” e também tem seu título atualizado por “tittle 'Mensagem importante'”.

Outros exemplos:

```
message "Erro" view-as alert-box.
```

Abaixo a saída do Progress para o usuário.

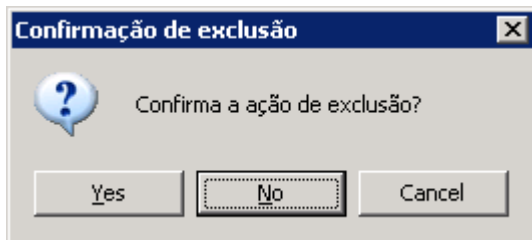


```
message 'Confirma a ação de exclusão?' view-as alert-box
```

```

question buttons yes-no-cancel
title 'Confirmação de exclusão'
update v-resposta as logical.
case v-resposta:
  when true then do: /* yes */
    /* aqui estaria excluindo um registro */
  end.
  when false then do: /* no */
    /* neste caso não teria a exclusão */
  end.
  otherwise /* cancel */
    stop.
end case.

```



### Atualização de registros com *exclusive-lock*

Quando se deseja atualizar registros de uma tabela deve-se utilizar este tipo de pesquisa. Este tipo de posicionamento para atualização deve ser utilizado com cautela pois bloqueia acesso de outros usuários a tabela. Evite manter registros em *exclusive-lock* por muito tempo.

Observações: A opção *no-lock* é útil para momentos em que você não queira atualizar o registro. Outros usuários podem ler, modificar ou excluir um registro que você esteja lendo por *no-lock*.

### Comparação entre *find* e *for each*

Principais diferenças entre um *find* e um *for each*:

- Find procura e disponibiliza apenas um registro.
- For each pode trazer mais de um registro por bloco.
- No find, sempre utilizar o "if available", que verifica se o registro pesquisado está disponível, não ocasionando erro.
- A utilização do "if available" pode ser utilizada a qualquer momento em que se deseje atualizar um registro, e não necessariamente logo após o find. Por exemplo, se você quiser escolher entre atualizar um campo de uma tabela ou criar um novo registro, você deve utilizar o "if available". Se o registro estiver disponível, e for este que você



deseja alterar, então deve-se fazer a alteração, se não estiver disponível, pode ser criado o registro.

- O for each não necessita de “if available” porque este bloco somente executa a instrução contida nele caso seja encontrado pelo menos um registro.

### Classificando registros por *by*

A classificação de registros utilizando o *by* pode ser feita conforme os exemplos abaixo:

Classificação por nome da pessoa:

```
/* início do programa */
for each pessoa no-lock by pessoa.nome.
    disp pessoa.nome.
end.
/* fim do programa */
```

Classificação pelo nome da pessoa em ordem alfabética descending.

```
/* início do programa */
for each pessoa no-lock by pessoa.nome descending.
    disp pessoa.nome.
end./* fim do programa */
```

Classificação utilizando vários campos.

```
/* início do programa */
for each produto no-lock by produto.tipo
    by produto.nome.
    disp produto.nome produto.tipo.
end./* fim do programa */
```

No exemplo acima, a saída será esta:

Nome Produto	tipo
A Cabana	1
A Revolucao dos Bichos	1
Harry Potter e a Pedra Filosofal	1
Livro de Receitas	1
Manual Político	1
Trem Bala	1
Capricho	2
Casa & Construção	2
Noticias Automotivas	2
DVD Jota Quest	3
DVD Skank	3
Correio do Povo	4
Zero Hora	4
ZH Classificados	4

A ordenação será feita primeiramente pelo código do produto, e posteriormente pelo nome do produto em ordem alfabética.

O Progress tenta usar todos os componentes de um índice, caso ele não consiga o banco cria um índice temporário para sua pesquisa.

### Lendo registros com *begins*

Podemos usar o comando *begins* para especificar um critério de seleção.

Abaixo segue o exemplo ilustrando esta utilização:

```
/* início do programa */  
for each pessoa where pessoa.nome begins "F" no-lock.  
    disp pessoa.nome.  
end. /* fim do programa */
```

O programa acima lista todas as pessoas da tabela *pessoa* em que o nome inicie com a letra F. Para esta busca não há diferença em utilização de aspas simples ou duplas para especificar a busca: 'F' ou "F" são igualmente tratadas.

Se foi definido um índice onde o campo sendo testado seja o primeiro componente, o *begins* é mais eficiente porque o banco de dados não precisará ler todos registros da tabela.

### Lendo registros com *matches*

O *matches* é um outro comando para especificar o critério de seleção:

Segue exemplo abaixo:

```
/* início do programa */  
for each pessoa where pessoa.nome matches("Fa*") no-lock.  
    disp pessoa.nome.  
end. /* fim do programa */
```

Este exemplo busca todas pessoas da tabela *pessoa* onde o nome se inicia com "Fa\*" em alguma parte da sua composição. Exemplos de nomes elegíveis: Fabrício, Fabíola.

O uso do *matches* pode ser também aplicado das seguintes maneiras:

'\*to' = todos registros que terminam por "to". Ex: Roberto, Augusto.

'\*silva' = todos nomes que possuam a estrutura "silva" no nome. Ex: Silvana, Marcio Silva.

## Considerações entre Begins e Matches

BEGINS	MATCHES
Pesquisa por um campo que comece com um caractere específico.	Pesquisa por um campo que bata exatamente com o padrão de caracteres especificado.
Usa o índice para um campo indexado, dessa forma, ela não precisa ler todos os registros da tabela.	Lê todos os registros da tabela para encontrar aqueles cujo campo bata com o padrão especificado.
Você não pode usar caracteres coringa	Você pode usar caracteres coringa para representar um único caractere (.) ou vários caracteres (*).

## Pesquisa por um índice informado

Quando se efetua um for each na tabela, por exemplo a empregado, o progress classifica os registros pelo índice primário.

Exemplo:

```
/* início do programa */
```

```
for each venda no-lock.
```

```
    MESSAGE venda.codVenda venda.datVenda VIEW-AS ALERT-BOX INFO BUTTONS OK.
```

```
end. /* fim do programa */
```

```
/* saída gerada pelo progress */
```

Cód. Data

-----

1 25/02/17

2 03/02/17

3 26/01/17

4 13/02/17

5 05/01/17

Pode ser informado um índice que se deseja:

Exemplo:

```
/* início do programa */
```

```
for each venda use-index idx002 no-lock.
```

```
    MESSAGE venda.codVenda venda.datVenda
```

```
    VIEW-AS ALERT-BOX INFO BUTTONS OK.
```

```
end. /* fim do programa */
```

Saída:

Cód. Data

```
-----
138 02/03/17
244 02/03/17
292 02/03/17
348 02/03/17
491 02/03/17
1169 01/03/17
```

O índice em questão (idx002) “ordena” a tabela venda pela data da venda, em ordem decrescente, ao invés da ordem ascendente do índice idx001, que ordena por código. O índice idx002 é um sumário ordenado seguindo estas colunas, contendo a localização física de cada registro, de modo que o acesso à cada registro seja direto. A verdadeira ordenação da tabela não é alterada, e segue respeitando o índice primário.

Também é possível efetuar as buscas em ordens diferentes do índice primário sem que seja criado um índice para a ordenação desejada, e esta ordenação é feita através da função “by”. No exemplo abaixo, vamos realizar a mesma busca acima, onde o índice idx002 nos mostra os resultados ordenados por data (decrescente), apenas utilizando a função by.

Exemplo:

```
/* início do programa */
for each venda no-lock by venda.datVenda descending.
    MESSAGE venda.codVenda venda.datVenda VIEW-AS ALERT-BOX INFO BUTTONS OK.
end./* fim do programa */
```

Neste caso o programa efetuará a ordenação do maior data para a menor. A aplicação da classificação "by" pode ser feita para qualquer campo constante na tabela.

### Inserindo dados via *Procedure Editor*

Abaixo o script que cria um registro na tabela empregado:

```
(abrir o procedure editor e digitar)
create produto.
update produto with 1 col.
(apertar F2 para se executar o comando)
```

Esta é a maneira mais simples de se criar um registro na tabela produto, contudo se a tabela tiver muitos campos estes ficarão dispostos de uma forma não muito agradável na tela.

Abaixo como fazer da forma que os campos a serem preenchidos ficam dispostos em uma coluna melhorando a visualização e também não confundindo o usuário.

(abrir o procedure editor e digitar)  
create produto.  
update produto with 1 col.  
(apertar F2 para se executar o comando)

Observação: "Col" é a abreviação aceita pelo Progress para column.

Após digitar as informações nos campos desejados deve-se pressionar F2 para confirmar a entrada de dados.

### Alterando dados via *procedure Editor*

Para se alterar um dado via procedure editor utilizaremos como exemplo o script abaixo:

```
/* início do programa */  
find first produto exclusive-lock.  
if available produto then update produto with 1 col.  
/* fim do programa */
```

O script acima procura o primeiro registro na tabela produto, reserva exclusivamente e caso exista o registro abre o mesmo para atualização de dados descrito em uma coluna. Após atualizar as informações no registro da tabela empregado o pressiona-se F2 para confirmar a atualização.

### Utilizando a função Repeat

Esta opção é utilizada para se efetuar uma repetição por um determinado número de vezes ou até o usuário via código achar que alterou todos os registros que procurava, caso contrário o repeat será executado infinitamente. Neste caso o programa entra em loop infinito.

Exemplo de um determinado número de vezes:

Observação: comentários no ambiente Progress devem ser feitos iniciando por barra + asterisco e terminando por asterisco + barra. Comentários são extremamente úteis para lembrarmos o que faz um determinado código ou programa, e também para que outros desenvolvedores consigam dar manutenção em um programa mais rapidamente.

```
/*definida variável vi-time como uma variável */
def var vi-time as int.

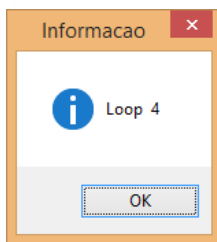
/* atribui para a variável vi-time em formato inteiro a hora, minuto e segundo do banco de dados */
vi-time = int(time).
repeat:
    /* confere se o horário atual é maior que o horário de início do programa masi 10 segundos,
    se sim sai do laço do repeat */
    if time >= vi-time + 10 then leave.
end.
```

Este script acima faz o programa aguardar 10 segundos e depois sai do laço do repeat, se não houvesse a condição do IF o repeat ficaria sendo executado infinitamente.

O exemplo abaixo utiliza o repeat junto à uma variável inteira (vint-cont), fazendo com que a variável receba o valor 1 e para cada repetição, mostre uma mensagem na tela. Ao chegar no valor 4, o comando "leave" faz com que o laço acabe.

```
DEFINE VARIABLE vint-cont AS INTEGER.

REPEAT vint-cont = 1 TO 5:
    MESSAGE "Loop " vint-cont VIEW-AS ALERT-BOX INFO BUTTONS OK.
    IF vint-cont = 4 THEN LEAVE.
END.
```



Observação: para se abortar um programa que esteja sendo executado utilize a combinação das teclas ALT + BREAK do teclado.

### Modificando registros com *For Each*

O exemplo abaixo ilustra esta funcionalidade:

```
/* Início do programa */
for each produto exclusive-lock.
    assign produto.preco = produto.preco + (produto.preco * 0.2).
end.
/* Fim do programa */
```

O script acima este faz a seguinte ação: para cada produto, acesso exclusivo, atribua ao preço um aumento de 20% (preço atual + 0,2 preço), fim. Este Script irá aumentar o preço de TODOS os produtos da tabela. Este tipo de comando deve ser utilizado com muito cuidado.

Observações: O bloco for each requer um "end" ao final. O código entre o início do for each e o end será executado a cada registro da tabela no caso tabela empregado.

### Excluindo registros (*delete*)

A utilização do comando delete deve ser feita com muita cautela e sempre está vinculada ao uso de um posicionamento do registro da tabela.

Usando delete com o comando find first:

```
/* Início do programa */  
find first produto where produto.cod = 1 exclusive-lock.  
if available produto then delete produto.  
/* Fim do programa */
```

O script acima efetua a seguinte ação: procura o primeiro produto onde o código do produto é o número 1 com ação exclusiva e ignorando erros de posicionamento, se o registro estiver disponível então exclua o mesmo.

Observação: o comando find first resultaria em erro se o comando no-error não estivesse escrito ao final da linha e o registro empregado não estivesse disponível, por isto, usa-se o “if available”.

O registro pode não estar disponível por dois motivos:

1. O registro estar sendo alterado em modo exclusive-lock.
2. Não existir o produto com o código 1.

Usando o comando delete com o bloco for each:

O comando delete também pode ser executado em conjunto com o bloco for each para executar uma exclusão em massa.

O script abaixo limpa a tabela produto, note que o for each sem restrição neste caso exclui toda tabela.

```
/* Início do Programa */  
for each produto exclusive-lock.  
    delete produto.  
end.  
/* Fim do Programa */
```

Este tipo de script deve ser executado com muito cuidado, pois neste formato, ou seja, sem cláusulas “where”, ele deletaria a tabela inteira.

## Padrão de indentação adotado na Zallpy

Dois programas podem ser escritos de maneiras diferentes mas executar a mesma coisa. Este cuidado de indentação faz com o que a manutenção seja simplificada tanto para quem fez lembrar o que o trecho ou programa faz quanto para um outro desenvolvedor.

Exemplo com boa indentação:

```
/* início do programa */  
for each empregado no-lock.  
    disp empregado.emp-cod  
        empregado.emp-nome with 1 col.  
end.  
/* fim do programa */
```

Exemplo com indentação ruim:

```
/* início do programa */  
for  
    each empregado no-lock. disp  
empregado.emp-cod empregado.emp-nome  
with 1  
col. end.  
/* fim do programa */
```

Abaixo algumas regras que precisam ser seguidas:

- Todo comando Progress termina com um ponto (.) a menos que seja um comando de início de bloco, que neste caso termina com dois pontos (:).
- Todo bloco necessita de um comando end. no final.
- Comentários necessitam ser iniciados por /\* e terminados por \*/, pode ser incluído em qualquer local do programa.
- Programas em Progress não são *case sensitive*, isto é, não diferenciam letras maiúsculas de minúsculas.
- Pode haver qualquer número de espaços ou linhas em branco entre as palavras de um comando. O compilador considera o ponto ou o dois pontos como a marca de fim de comando.
- Para diferenciar blocos de comando deve-se dar quatro espaços ou um TAB para reinício de linha.

## Relacionamento entre tabelas

Vamos gerar um relacionamento entre as tabela pessoa, usuário e autor. Como disponível no dicionário de dados, as tabela Usuario e Autor possuem CPF, porém, não seria correto que elas possuísssem cadastros que não existem na tabela Pessoa. Portanto, a medida que foi adotada na criação dos registros destas tabelas foi de que um autor ou usuário só



poderia ser inserido no sistema se ele possuísse um cadastro válido na tabela pessoa, cujo índice primário não permite CPFs repetidos. Ou seja, existe um relacionamento entre as tabelas que visa não permitir dados inconsistentes dentro do sistema. Como o campo CPF é comum nas três tabelas, e ele é o “campo chave” (compõe sozinho o índice primário) sabemos que acessar os dados da pessoa através deste campo, através das tabelas Usuario e Autor, é a maneira correta de acesso.

No código abaixo, iremos utilizar os comandos já vistos “for each”, “find”, “if available”, “where” e “no-lock”, para buscar o login (Usuario.login), Pseudônimo (Autor.nick) e Telefone (Pessoa.tel), através do campo CPF.

```
/*Inicio do programa*/
for each Autor no-lock.
    find first pessoa where pessoa.cpf = autor.cpf no-lock no-error.
    if available pessoa then do:
        find first usuario where usuario.cpf = pessoa.cpf no-lock no-error.
        if available usuario then do:
            disp autor.nick usuario.login pessoa.tel.
        end.
    end.
end.
/* Fim do programa */
```

Note que iniciamos a busca pela tabela autor, pois é a tabela que nos limita ao menor número de iterações até o resultado final. Explicando: Sabemos que necessitamos de dados de autor, pessoa e usuário, porém, sabemos que **nem toda pessoa é autor**, portanto, é mais eficiente iniciar a busca por autor, pois todo autor **será uma pessoa**, e então, não será necessário o acesso à tabela inteira de pessoa, e sim, somente às pessoas cujo CPF também esteja listado na tabela Autor, ou seja, são autores. O mesmo vale para a tabela usuário, **todo Usuario é uma pessoa**, porém, **nem todo Usuario é um autor**, portanto, iniciar a busca por Usuario não seria eficiente.

Porém, se a busca fosse solicitada da seguinte forma: “Buscar o login (Usuario.login), Pseudônimo (Autor.nick) **para o caso de autores** e Telefone (Pessoa.tel)”, a busca terá de percorrer todas as pessoas, visto que a condição de ser um autor está sendo colocada como opcional. A busca correta seria assim:

```
/* início do programa */
for each pessoa no-lock.
    find first usuario where usuario.cpf = pessoa.cpf no-lock no-error.
    if available usuario then do:
        find first autor where autor.cpf = pessoa.cpf no-lock no-error.
        disp (if available autor then autor.nick else "") label "Pseudonimo" usuario.login pessoa.tel.
```

```

    if available autor then release autor.
  end.
end.
/* fim do programa */

```

Note que nesta busca não existe uma obrigatoriedade da existência de um autor e a condição de "if available" só é utilizada no momento da exibição do campo autor.nick. Se não existir um autor disponível para esta pessoa.cpf, então mostra vazio (else ""), mantendo o nome da coluna como "Pseudônimo". O comando release é utilizado para que, quando não for encontrado um autor para um cpf, não exista o risco de ser mostrado o valor do autor previamente posicionado. O comando release "desposiciona" a tabela autor. A saída desta busca é esta:

Pseudonimo	login	Telefone
James	JOR2665	51999945009
	ELI5506	53999961375
	Jam1919	519875824852
	FER3119	52999193395
	CAM1040	52999997965
	AND7847	53999795547
	CAR3411	52998568780
	STE3704	54999844689
	AND1384	54999632924
	GUI2002	55999825149

Note que, nesta demonstração de saída (a saída completa possui mais de 7 mil pessoas) apenas uma pessoa também é "Autor".

## Pesquisa entre tabelas mãe e filha com o comando *of*

O comando "of" é equivalente a utilização do trecho "where autor.cpf = pessoa.cpf".

Exemplo da utilização do of:

```

/* início do programa */
for each pessoa no-lock.
    find first autor of pessoa no-lock no-error.
    if available autor then disp autor.nick.
end.
/* fim do programa */

```

A utilização do "of" deve seguir as seguintes regras:

- O campo cpf usado no exemplo deve ser unicamente indexado em ao menos uma das tabelas.

- O campo em comum também precisa ter o mesmo nome (escrito da mesma forma) e o mesmo tipo de dado nas duas tabelas. Caso haja diferença não será possível utilizar o comando `of` e então deverá se utilizar o comando `where`.
- A tabela mãe precisa ter um índice único no campo.

Não é recomendado, na Zallpy Group, o uso do “`of`”, pois nesse caso, e como as tabelas e índices são geralmente grandes e extensos, utilizar a chave de busca explícita (`where...`) é mais indicado, pois também facilita a manutenção.

### Utilizando a quebra *break by*

A opção `break` define uma categoria de quebra.

```
/* início do programa */
FOR EACH venda NO-LOCK BREAK BY venda.qtdTotal DESCENDING.
    disp Venda.datVenda
        trim(Venda.prod-lst) label "Lista de Produtos"
        Venda.qtdTotal.
end.
/* fim do programa */
```

Neste caso acima o programa irá mostrar todas vendas, agrupadas pela quantidade de produtos comprados. Como mostrado abaixo:

Data	Lista de Produtos	Total de itens
12/01/17	10,13,1,	8
15/02/17	11,10,5,	8
11/01/17	13,7,10,	7
08/02/17	13,12,10	7
04/01/17	6,5,12,3	7
06/01/17	6,1,13,4	7
12/02/17	11,2,6,1	7

Obs:

- O comando `trim()` serve para retirar os espaços em branco dos campos em texto, no caso `venda.prod-lst`. O `trim()` retira espaços em branco à direita e à esquerda do texto, e possui duas variações: `left-trim` (retira somente da esquerda) e `right-trim` (retira somente da direita).

### Utilizando a função *First-of / last-of*

## First-of

A função first-of deve ser utilizada quando se utiliza no bloco o break by, pois serve para saber se é o primeiro registro da quebra informada.

O exemplo abaixo agrupa as vendas por quantidade de produtos. O comando first-of neste caso servirá para saber se é a primeira venda (seguindo a ordenação da busca) obedecendo à quebra informada:

```
/* início do programa */
FOR EACH venda NO-LOCK BREAK BY venda.qtdTotal DESCENDING.
    if first-of (venda.qtdTotal) then disp  Venda.datVenda
                                         trim(Venda.prod-1st) label "Lista de Produtos"
                                         Venda.qtdTotal.
end./* fim do programa */
```

Resultado:

Data	Lista de Produtos	Total de itens
12/01/17	10,13,1,	8
11/01/17	13,7,10,	7
21/02/17	7,4,5,1	6
02/03/17	1,2,9	5
18/02/17	9,1,6,4	4
20/02/17	5,3,2	3
03/01/17	11,5	2
02/01/17	12	1

## Last-of

O last-of é o inverso do first-of, e disponibiliza o registro quando ele for o último da quebra. Este comando é muito utilizado para que sejam feitos totalizadores, por exemplo, a quantidade de vendas para cada quantidade de itens vendidos (busca acima). Observa-se que há a definição de uma variável vi-total que será alimentada com 1 a cada venda do agrupamento antes de cair no last-of (ultimo da quebra). A variável é inicializada com valor zero para não necessitarmos fazer esta inicialização já no decorrer do código. Ao entrar no trecho do last-of de vendas, imprimimos os valores e reinicializamos a variável para reiniciar a contagem.

```
/* início do programa */
define variable vi-total as integer initial 0 format ">>>9" label "Total de Vendas".
FOR EACH venda NO-LOCK BREAK BY venda.qtdTotal DESCENDING.
    if last-of (venda.qtdTotal) then do:
        disp  Venda.qtdTotal
```

```

vi-total.
assign vi-total = 0.
end.
else assign vi-total = vi-total + 1.
end./* fim do programa */

```

Parte do Resultado:

Total de itens	Total de Vendas
8	1
7	15
6	128

## Operações com String

### Lookup

A função lookup como validadora retorna um inteiro dando a posição em que se encontra a expressão procurada ou então zero para caso não encontre.

O programa abaixo ilustra esta utilização:

```

/* início do programa */
def var vc-estados as char init 'RS,SC,PR,RJ,SP,BA'.
def var vc-procura as char.
Update vc-procura.
If lookup(vc-procura, vc-estados) = 0 then
    Message "Estado não encontrado" view-as alert-box.
Else message "Estado existente na listagem do sistema" view-as alert-box.
/* fim do programa */

```

Neste programa inicializamos a lista de estados na variável vc-estados, inicializamos uma variável que será atualizada pelo update manualmente pelo usuário e efetuamos a validação pela função lookup. Ao retornar zero significa que a sigla imputada pelo usuário não foi encontrada na listagem do programa, caso contrário retorna a sua posição na listagem.

O separador padrão no lookup é a vírgula. Caso o separador utilizado no texto seja diferente de vírgula, isto deve ser informado no comando. Abaixo, o mesmo comando, utilizando o separador ";" (ponto e vírgula).

```

If lookup(vc-procura, vc-estados, ";") = 0 then

```

## Entry

A função entry possibilita o acesso à uma entrada de uma String determinada por uma posição. No exemplo abaixo, a mensagem mostrada será “B”, pois é o texto dentro da String “texto”, e com o separador padrão (vírgula), que equivale a 2ª entrada.

```
/* Inicio do Programa */  
DEFINE VARIABLE texto AS CHAR INITIAL "A,B,C,D,E".  
MESSAGE ENTRY(2, texto) VIEW-AS ALERT-BOX INFO BUTTONS OK.  
/* O valor mostrado sera "B" */  
/* Final do Programa */
```

## Can-do

A função can-do permite que seja verificado, dentro de sentença, a existência de um caractere ou palavra. O valor retornado é falso ou verdadeiro. No exemplo abaixo, o retorno é true.

can-do(“a,b,c”, “b”) retorna true/false (true)

## Index

A função index permite que seja retornada a posição de uma sentença dentro de um texto de origem. A principal diferença com o Lookup é de que não considera separadores, portanto, no exemplo abaixo, onde a sentença é composta por letras e vírgulas, o retorno seria 3, que dentro da String, é a primeira posição da letra “b”. Se existissem outros “b” no texto, o retorno é da posição da primeira sentença encontrada.

Index( “a,b,c”, “b”) retorna um inteiro (3)

## Num-entries

A função num-entries retorna o número de entradas em uma string, baseando-se em um separador. No exemplo abaixo, é retornado 4. Como a vírgula é o separador padrão, não é necessário informar um separador para a sentença abaixo, mas se, ao invés de vírgulas, existisse um “\$”, ele deveria ser informado como separador.

num-entries(“a,b,c,d”, “,”) retorna um inteiro (4)

## If Can-find (first ou last)

Outro tipo de validação pode ser feito pelo comando if can-find. O benefício desta validação ou confirmação é que o progress consegue verificar a existência do registro sem posicioná-lo, sendo muito mais rápido que um find first. Esta validação pode ser executada conforme o exemplo abaixo:

```
/* início de programa */  
if can-find(first autor no-lock) then disp "Existe um Autor".  
/* fim do programa */
```

Outra utilização do can-find pode ser feito na validação negativa conforme exemplo abaixo:

```
/* início de programa */  
if not can-find(first autor no-lock) then do:  
    create autor.  
    Update autor.  
End.  
Else disp "Existe um Autor".  
/* fim do programa */
```

### Definindo e trabalhando com *temp-tables*

As temp-tables são tabelas temporárias, residentes somente em disco que podem ser usadas para armazenar registros de maneira semelhante a uma tabela física do banco de dados. Estas valem apenas para o programa que as criou ou então para um outro programa caso estas temp-tables tenham sido criadas como shared.

As temp-tables são muito utilizadas, especialmente em relatórios, porque servem para armazenar dados de várias tabelas e criamos dinamicamente registros para cálculos que servirão somente para a emissão de um determinado relatório por exemplo.

As temp-tables podem ser criadas campo a campo como uma tabela comum do banco de dados ou então com o comando like e o nome de uma tabela física abreviarmos esta criação. Pode-se usar em conjunto de um comando de definição de uma temp-table com like de uma tabela física a inserção de campos que a tabela física não possuía.

Segue exemplo abaixo desta definição.

```
/* início do programa */  
define temp-table tt-resumo  
    field cpf like pessoa.cpf  
    field valor as decimal  
index idx001 is primary unique cpf .  
  
for each venda no-lock break by venda.cpf.  
    if first-of(venda.cpf) then do:  
        create tt-resumo.  
        assign tt-resumo.cpf = venda.cpf  
        tt-resumo.valor = venda.valor.  
    end.  
else do:
```

```

find first tt-resumo where tt-resumo.cpf = venda.cpf exclusive-lock no-error.
if available tt-resumo then assign tt-resumo.valor = tt-resumo.valor + venda.valor.
end.

end.

for each tt-resumo no-lock by tt-resumo.valor descending.
disp tt-resumo.cpf tt-resumo.valor.
end.
/* fim do programa */

```

A temp-table tt-resumo acima é utilizada para armazenar, para cada cpf, a soma dos seus gastos com compras. Posteriormente, é realizado um for each na temp-table, e exibido o resultado, mas com a tabela já criada e alimentada, é possível utilizá-la quantas vezes for necessário, pois os registros permanecerão existindo até que a execução de todo o programa acabe, a menos que sejam deletados através de um comando delete.

Obs: O comando "like" copia o formato de dado do banco de dados e máscara para a variável ou campo de temp-table que está sendo definido. Sua definição se torna muito importante porque ao se alterar o banco de dados não há necessidade de se alterar o programa que utiliza tal funcionalidade. O contrário aconteceria se um campo fosse definido como dec (decimal) e no banco houvesse a conversão para int (inteiro), haveria a necessidade de se alterar todos os programas e verificar o funcionamento correto da funcionalidade.

### Buffer-copy

O comando "buffer-copy" serve para que seja possível a cópia de valores entre tabelas, sendo muito utilizado para a cópia de valores do banco de dados para temp-tables, de modo a otimizar a performance. Imaginando a situação:

- Existe uma tabela muito grande no banco de dados, a qual, para a execução de um procedimento, precisa ser lida (percorrida) várias vezes.
- A massa de dados que vai ser útil para o procedimento em questão é pequena, muito menor do que o tamanho e/ou quantidade de registros existentes no banco de dados.

Neste caso, é interessante a definição de uma temp-table igual à tabela original, e que será alimentada somente com os dados que interessam para a execução de um procedimento.

Vamos imaginar que a tabela Venda possua 5 milhões de registros, mas necessitamos apenas das vendas ocorridas entre 01/01/2017 e 05/01/2017, e não possuímos um índice por data. Neste cenário, ler a tabela do banco de dados várias vezes pode afetar a performance, e uma solução como a apresentada abaixo é indicada:



```
DEFINE TEMP-TABLE tt-venda LIKE Venda.

FOR EACH venda WHERE venda.datVenda >= 01/01/2017 AND
                        venda.datVenda <= 01/05/2017 NO-LOCK.
    CREATE tt-venda.
    BUFFER-COPY venda TO tt-venda.
END.
```

No código acima, todos os registros que compõem a massa de dados “útil” ao procedimento estarão dentro da tt-venda, e a partir deste ponto, iremos efetuar consultar (find..for each) apenas em “tt-vendas”.

Observações:

- O comando buffer-copy apenas agiliza um processo que poderia ser efetuado com “create” e “assign” dos campos de uma tabela, um por um.
- O comando pode ser usado para copiar dados entre tabelas de bancos de dados distintos, bastando o Progress estar conectado em ambas as bases de dados. Neste caso, cada tabela deve referenciar o banco de dados em questão:

```
FOR EACH banco1.venda WHERE banco1.venda.datVenda >= 01/01/2017 AND
                        banco1.venda.datVenda <= 01/05/2017 NO-LOCK.
    CREATE banco2.venda.
    BUFFER-COPY banco1.venda TO banco2.venda.
END.
```

Onde: Banco1 e Banco2 são dois bancos de dados distintos, mas que possuem uma tabela igual.

- Deve-se ter muito cuidado ao copiar dados entre tabelas, pois se ocorrer uma violação de integridade, como uma chave primária duplicada ou índice repetido, o processo irá falhar.
- comando “Using”. Neste caso, só os campos citados nesta cláusula serão copiados. Também pode-se excluir campos da cópia utilizando o comando “Except”: Pode-se utilizar o buffer-copy para somente alguns campos da tabela, através do

```
/* Somente os campos CodVenda e CPF serão copiados */
BUFFER-COPY venda USING Venda.codVenda Venda.CPF TO tt-venda.

/* Somente os campos datVenda e CPF serão ignorados */
BUFFER-COPY venda EXCEPT Venda.datVenda Venda.CPF TO tt-venda.
```

- **Não** existe “receita de bolo” para melhorias de performance, pois existem muitas variáveis envolvidas, como tamanho da massa dos dados, índices, relacionamentos entre as tabelas, tipos de consulta, performance do banco de dados, entre outras, portanto, não é sempre que o buffer-copy é a melhor opção, também não é garantido que criar uma temp-table com dados de uma tabela sempre tonará a execução mais rápida. Cada caso é um caso e deve ser avaliado individualmente.

## New shared e shared variables

Para podermos compartilhar dados entre programas devemos seguir duas etapas:

1. A definição na origem de uma variável new shared
2. A definição no destino de uma variável shared

Tipos shared podem ser aplicados em variáveis, frames, buffers, stream e temp-tables e work-tables. São mais utilizadas em variáveis e temp-tables. O conceito de temp-tables será tratado a frente.

Definição no primeiro programa:

```
/* início do programa origem */  
define new shared var vi-tipo as int.  
define new shared temp-table tt-eqpto  
    field ele-cod as int  
    field dta-entrega as date.  
/* fim do programa origem */
```

```
/* início do programa destino */  
define shared var vi-tipo as int.  
define shared temp-table tt-eqpto  
    field ele-cod as int  
    field dta-entrega as date.  
/* fim do programa destino */
```

- Estes comandos permitem que vários programas de uma aplicação compartilhem os mesmos tipos de dados.
- Deve-se usar o comando define ou somente def new shared no primeiro programa em que se referencia o elemento que se quer compartilhar e nos demais utilizar somente define shared ou somente def shared.
- Os tipos de dados da origem e do destino devem ser iguais.

## Definindo procedimentos (*procedures*)

Procedures internas são sub-programas em Progress colocados dentro de outro programa. Pode ser usado para não repetirmos trechos de códigos e necessita ser chamada por um comando “run” seguido do nome da procedure. Esta procedure pode receber ou não parâmetros e devolver ou não parâmetros.

Segue um exemplo de programa com procedure abaixo:

```

/* início do programa */
define variable vcha-editora like editora.nome.
for each autor no-lock.
    run p-nome-editora(input autor.edi-cod, output vcha-editora).
    disp vcha-editora.
end.

procedure p-nome-editora.
    define input parameter vint-edi-cod as integer.
    define output parameter vcha-editora-nome as character.

    find first editora where editora.edi-cod = vint-edi-cod no-lock no-error.
    if available editora then assign vcha-editora-nome = editora.nome.
end procedure.
/* fim do programa */

```

Obs:

- O compilador Progress não valida a existência de uma procedure ao compilar um programa. Portanto, mesmo que uma procedure inexistente seja chamada, não haverá erro de sintaxe, porém, o Progress irá procurar a procedure em todo o propath do sistema, antes de acusar o erro de execução. Isto é indesejável pois pode tornar a execução muito lenta, e pior ainda, executar uma procedure com mesmo nome, em outro programa, com outra funcionalidade. Por isto, uma boa prática ao chamar procedures é utilizar o comando “in this-procedure” antes de finalizar a sentença. Este comando formaliza que o progress somente deve procurar a procedure chamada dentro do programa chamador, e não em todo o propath.

### Definindo funções (*function*)

As funções na linguagem progress são semelhantes as procedures internas. As diferenças são: não são chamadas por um comando run e por terem na sua definição um parâmetro de retorno que será atribuído no decorrer da função (sempre retornam um valor, mesmo que "" (vazio)) .

Segue abaixo um exemplo de função:

```

/* início de programa */
function f-get-nick returns character(input cpf as char).
    find first autor where autor.cpf = cpf no-lock no-error.
    if available autor then return autor.nick.
    else return "".
end function.

```

```

for each autor no-lock.
  disp autor.cpf
  f-get-nick(autor.cpf) label "Pseudonimo".
end./* fim do programa */

```

O programa acima inicia definindo uma função e informa o tipo de dado que irá retornar, além do tipo de dado de entrada. Ele efetua a pesquisa a partir do dado de entrada e caso encontre o registro, retorna a informação tratada convertendo (efetuando parse, se necessário) o pseudônimo do autor. Note que a function deve ser declarada antes da sua chamada, algo que não ocorre com as procedures. Se você desejar implementar a função no final do código, também é possível, porém, sua definição deverá ser declarada no início do programa, seguida da palavra “forward”. Exemplo:

```
function f-get-nick returns character(input cpf as char) forward.
```

```
.... (programa inteiro)
```

```
function f-get-nick returns character(input cpf as char).
```

```
...
```

```
End function.
```

### Definição de *buffer*

Um buffer é uma representação da mesma tabela referenciada de um nome diferente. Ao se atualizar um buffer está se atualizando a tabela propriamente dita. É utilizado comumente para se acessar duas vezes a mesma tabela em um bloco ou quando não se deseja perder o posicionamento anterior.

Para se definir um buffer em Progress utilizamos a sintaxe abaixo:

```
define buffer bf-pessoa for pessoa.
```

### Utilizando buffer nas pesquisas

A pesquisa nesta tabela segue a mesma estrutura da pesquisa na tabela normal.

```

/* início do programa */
define buffer bf-pessoa for pessoa.
find first pessoa no-lock no-error.
If available pessoa then do:
  find first bf-pessoa where bf-pessoa.nome <> pessoa.nome no-lock no-error.

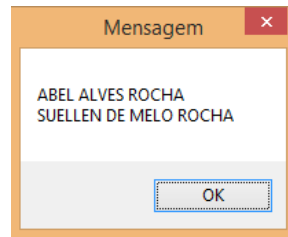
```

```

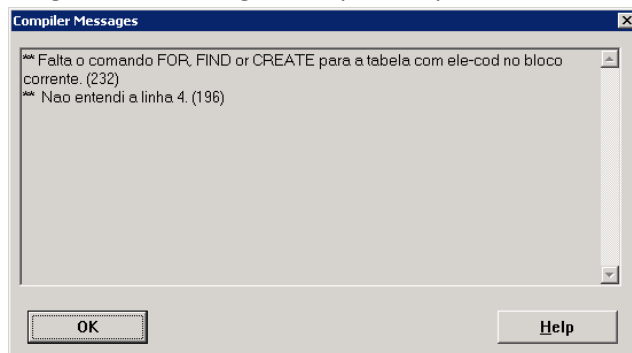
if available bf-pessoa then message pessoa.nome skip bf-pessoa.nome view-as alert-box.
end.
/* fim do programa */

```

Neste caso, estamos acessando a tabela pessoa ao mesmo tempo, posicionando registros diferentes (nome <> nome), e mostrando ambos. Resultado:



Deve-se ter sempre a atenção de se referenciar o buffer no caso acima, caso contrário o Progress não consegue compilar e apresenta o erro mostrado abaixo:



Abaixo, a utilização de um buffer de empregado para buscar o empregado superior à ele, durante a execução do laço principal, podendo assim, manter dois registros posicionados.

```

/* início do programa */
def buffer bf-empregado for empregado.

For each empregado no-lock.
    Find first bf-empregado where bf-empregado.emp-cod = empregado.emp-sup no-lock no-error.
    If available bf-empregado then do:
        disp "Empregado: " empregado.emp-nome
        "Chefe      : " bf-empregado.emp-nome.
    End.
end.
/* fim do programa */

```

## Operações com arquivos

### Utilizando o comando *input from*

Este comando é comumente efetuado para leitura de um arquivo txt e a alimentação de tabelas físicas ou então de temp-tables.

Abaixo um exemplo simples da utilização dos input de arquivos txt.

```
/* exemplo do conteúdo do arquivo txt */
```

```
1 Caneta  
2 Pendrive  
3 Livro  
4 Revista
```

```
/* início de programa */
```

```
def temp-table tt-produto  
    field cod as int  
    field nome as char.
```

```
input from c:\temp\arq.txt.
```

```
repeat:  
    create tt-produto.  
    import cod nome.
```

```
end.
```

```
input close.
```

```
/* fim do programa */
```

O arquivo txt está definido com o código do produto, e separado por um espaço, o nome do produto. Definimos uma temp-table para inserirmos os dados que serão lidos.

- O input from define o caminho do arquivo que estamos lendo os dados.
- O comando repeat é utilizado para, enquanto houver linhas a serem lidas executar o código no seu bloco.
- Criamos o registro na temp-table e depois importamos as informações na ordem que estão dispostas no arquivo.

Finalizamos o programa com fechando a entrada do arquivo.

### Input from com Delimiter

Outra funcionalidade bastante utilizada no comando input é o delimiter. O delimiter é utilizado para definirmos que caractere separa cada informação no arquivo origem.

Abaixo o exemplo ilustrativo:

```
/* exemplo do conteúdo do arquivo txt */
```



```
1;Caneta
2;Pendrive
3;Livro
4;Revista
/* início de programa */
def temp-table tt-produto
    field cod as int
    field nome as char.

input from c:\temp\arq.txt.
repeat:
    create tt-produto.
    import delimiter ';' cod nome.
end.
input close.
/* fim do programa */
```

As informações estão separadas pelo caractere ';'. O caractere “;” é muito utilizado para arquivos que são importados ou exportados do programa Excel.

Finalizamos o programa com fechando a entrada do arquivo.

Se o arquivo contém mais campos do que o programa está definido para leitura, o progress simplesmente ignora os demais. Caso houvesse no arquivo de entrada o exemplo abaixo o progress executaria o programa normalmente ignorando as informações a partir do segundo ';':

```
/* exemplo do conteúdo do arquivo txt */
1;Caneta;bic
2;Pendrive
3;Livro;importado
4;Revista

/* início de programa */
def temp-table tt-produto
    field cod as int
    field nome as char.

input from c:\temp\arq.txt.
repeat:
    create tt-produto.
    import delimiter ';' cod nome.
end.
input close.
```

```
for each tt-produto no-lock.
    disp tt-produto.cod tt-produto.nome.
end.
/* fim do programa */
```

As informações que o for each trará serão as mesmas nos três programas apresentados anteriormente.

### Input from com comando unformatted

Outro tipo de importação de informação de arquivo é se utilizar no import o comando UNFORMATTED. Usando este comando o progress não separa campos, lê toda linha e a insere em uma variável ou campo. Esta funcionalidade é utilizada quando se quer tratar os campos por definição posicional.

O programa fica conforme abaixo:

```
/* exemplo do conteúdo do arquivo txt */
1;Caneta;bic
2;Pendrive
3;Livro;importado
4;Revista

/* início de programa */
def var vc-linha as char.

input from c:\temp\arq.txt.
repeat:
    import unformatted vc-linha.
    disp vc-linha.
end.
input close.
/* fim do programa */
```

A saída do programa será exatamente conforme cada linha disposta no arquivo.

### Utilizando comando *export*

O comando export é comumente utilizado para exportação de dados de uma tabela para um arquivo.

O exemplo abaixo ilustra esta funcionalidade:



```
/* início de programa */  
output to c:\temp\pessoas.txt.  
for each pessoa no-lock.  
    export pessoa.  
end.  
output close.  
/* fim do programa */
```

Pode-se exportar a tabela completa por linha conforme o exemplo acima ou então selecionar alguns campos para envio para o arquivo como o exemplo abaixo.

```
/* início de programa */  
output to c:\temp\pessoas.txt.  
for each pessoa no-lock.  
    export pessoa.emp-cod pessoa.emp-desc.  
end.  
output close.  
/* fim do programa */
```

### Enviando dados para um arquivo com formato para leitura em *Excel* (padrão)

O envio de dados muito utilizado para ser aberto e separado em células no excel é facilitado com a utilização do export em conjunto com o delimiter ';'.

O exemplo abaixo ilustra esta utilização.

```
/* início de programa */  
output to c:\temp\pessoas.txt.  
for each pessoa no-lock.  
    export delimiter ';' pessoa.  
end.  
output close.  
/* fim do programa */
```

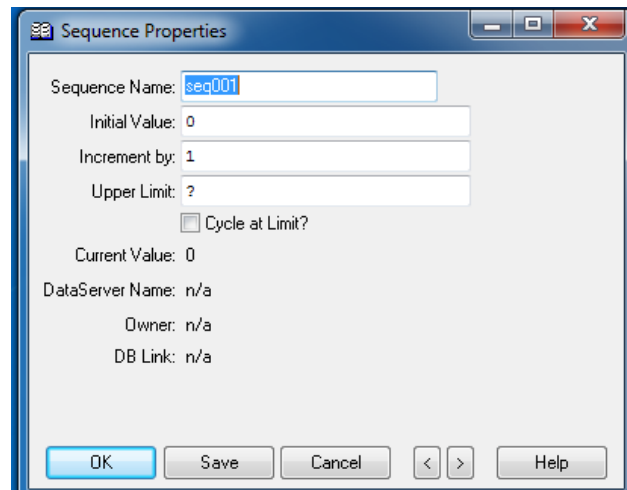
Após a execução basta abrir o excel e procurar o arquivo txt. Na abertura, selecionar o tipo de dados de importação pela escolha de delimitador. Clique em avançar e depois escolher o delimitador ';'. Assim as informações separadas por ';' serão dispostas separadamente em células diferentes.

### Sequences

As sequences são contadores globais do banco de dados, sem vínculos com nenhuma tabela. Elas são muito utilizadas para o controle/contagem de processos, sendo vinculadas

apenas à sua execução, e não às tabelas e/ou valores contidos no resultado de cada procedimento.

Na imagem abaixo, temos as propriedades da sequência chamada “seq001”. É possível ver que ela possui um valor inicial = 0 e a cada chamada é acrescida em 1. O comando para visualizar o seu valor atual é o “current-value”, e o comando para o próximo valor é “next-value”:

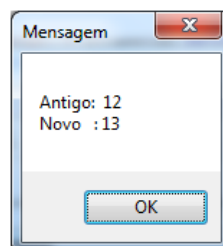


```
DEFINE VARIABLE vi-antigo AS INTEGER NO-UNDO.
DEFINE VARIABLE vi-novo AS INTEGER NO-UNDO.

ASSIGN vi-antigo = CURRENT-VALUE(seq001).
ASSIGN vi-novo = NEXT-VALUE(seq001).

MESSAGE "Antigo: " vi-antigo SKIP "Novo : " vi-novo
        VIEW-AS ALERT-BOX.
```

E a saída :



Na próxima vez que o comando for executado, os valores serão 13 e 14, respectivamente, pois ao se utilizar o comando “next-value”, a sequence já recebe o novo valor.

## Triggers

Triggers são eventos de programação executados sempre quando uma ação ocorrer. As ações podem ser CRUD (Create, retrieve, update e delete): criação, recuperação, atualização e deleção.

Triggers podem ser de dois tipos:

1. Trigger de banco de dados

## 2. Trigger de interface de usuário

Triggers de interface de usuários são usadas no desenvolvimento de aplicações gráficas e triggers de banco serão tratadas neste momento. As triggers são programas .p que podem estar salvas no banco de dados ou então somente referenciadas nele, sendo assim, salvas em uma pasta.

A trigger abaixo é executada quando da atualização do código da cidade do cliente. Lendo o código, este executa a seguinte instrução:

- Procura o código da cidade na tabela de cidades igual ao código informado na atualização da tabela cliente.
- Caso não encontre a referência informada na tabela ele retorna com a mensagem do comando "Message", veja que é informado na mensagem o nome da trigger. Isto é muito útil pois só saberemos de antemão que se trata de uma trigger e não de um trecho do programa por esta mensagem. Caso não houvesse referência neste ponto do programa do nome da trigger iríamos perder bastante tempo pesquisando em qual trecho do programa está a mensagem, se é de algum outro processo ou então algum programa chamado.
- Em caso de não encontrar a cidade a trigger ainda desfaz toda alteração e retorna como status de erro encontrado.
- Caso contrário, em positivo de encontrar o código da cidade do cliente na tabela de cidades a trigger ainda preenche na tabela de cliente o código de país e a UFE (unidade federativa) da cidade.

```
/* início do programa */
```

```
TRIGGER PROCEDURE FOR ASSIGN OF Cliente.cid-cod.
```

```
find first Cidade where
```

```
    Cidade.cid-cod = Cliente.cid-cod no-lock no-error.
```

```
if not available Cidade then do:
```

```
    message "Verifique a Cidade indicada, pois a mesma não esta cadastrada! (TAclt-01)"
```

```
    view-as alert-box information button OK.
```

```
    undo,return error.
```

```
end.
```

```
assign Cliente.pai-cod = Cidade.pai-cod
```

```
    Cliente.ufe-cod = Cidade.ufe-cod.
```

```
/* fim do programa */
```

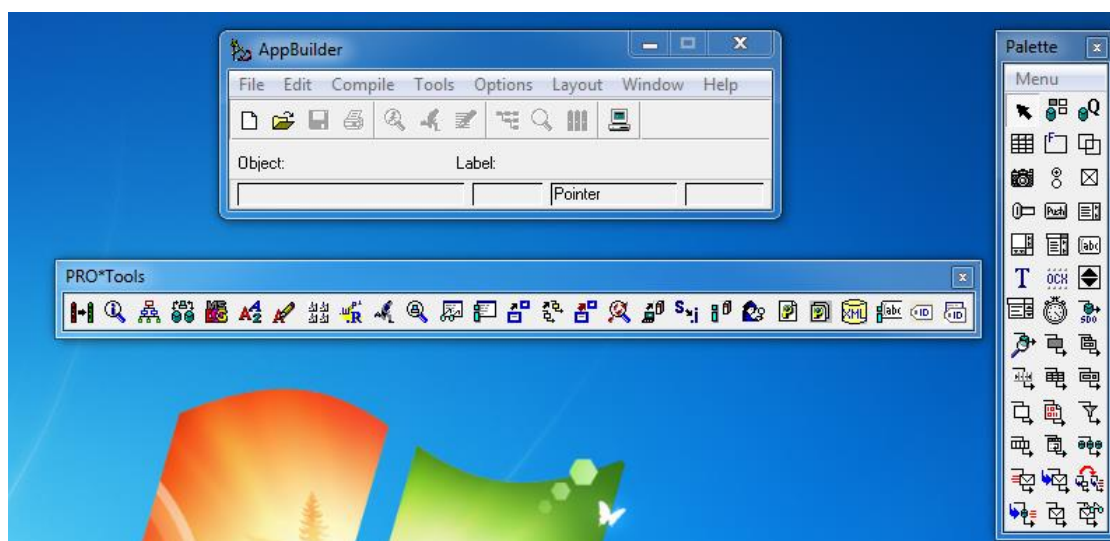
## Overview de Progress Gráfico

### Parte Conceitual

O Progress 4GL possibilita a criação de sistemas totalmente voltados à desktop Windows, oferecendo uma ferramenta de desenvolvimento gráfico que permite a criação, manipulação e execução das janelas e fluxos desenvolvidos, podendo estar ou não conectado à um banco de dados.

Neste *overview* serão apresentados a ferramenta de desenvolvimento gráfico (AppBuilder), os principais componentes disponíveis para uso e as principais características próprias do desenvolvimento desktop, aplicados no desenvolvimento de duas janelas, que, quando conectadas ao banco de dados Sports, permitem a visualização, inserção, e atualização de dados cadastrais dos *Customers*.

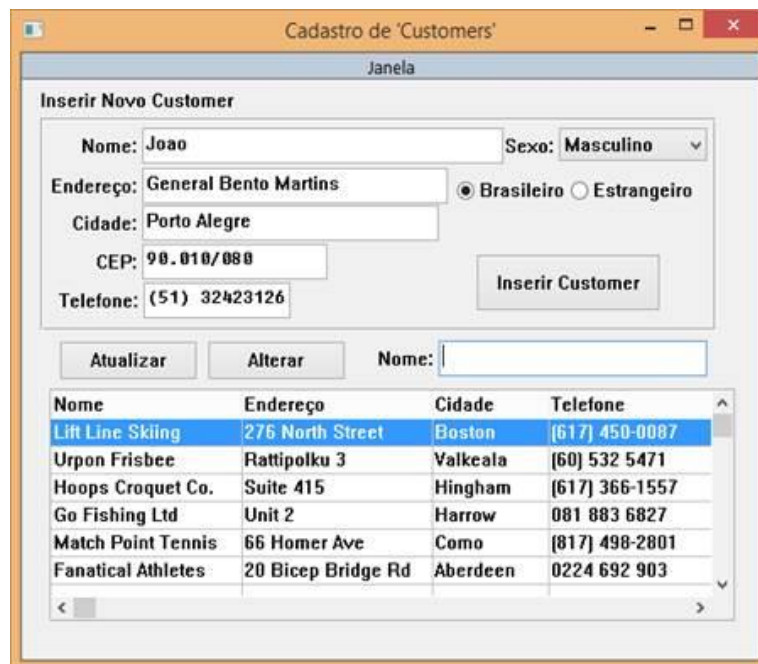
Abaixo, uma figura que ilustra o AppBuilder, junto com outros dois componentes; *PRO\*Tools* e *Palette*. O *PRO\*Tools* disponibiliza ao desenvolvedor o acesso à algumas funcionalidades do banco de dados, como como gestão de propath, gerenciamento de bancos de dados conectados, personalização de cores, entre outras. A *Palette* disponibiliza os componentes que podem ser inseridos em um programa, como por exemplo: browser, radio-button, combo-box, label, fill-in (campo de texto), frame, entre outros.



### Observações:

- Através do APPBuilder é possível acessar todos os outros componentes do Progress, como Editor, Dicionário de Dados, Data Administration e compilador.
- Alguns dos componentes da *Palette* são “de arrastar”, ou seja, é necessário que já exista uma janela (e em alguns casos, um frame), para que o objeto possa ser inserido.

As janelas que serão ilustradas neste *overview* e desenvolvidas neste curso são:



**Cadastro de 'Customers'**

Janela

**Inserir Novo Customer**

Nome:  Sexo:

Endereço:  ☒ Brasileiro ☐ Estrangeiro

Cidade:

CEP:

Telefone:

Nome:

Nome	Endereço	Cidade	Telefone
Lift Line Skiing	276 North Street	Boston	(617) 450-0087
Urpon Frisbee	Rattipolku 3	Valkeala	(60) 532 5471
Hoops Croquet Co.	Suite 415	Hingham	(617) 366-1557
Go Fishing Ltd	Unit 2	Harrow	081 883 6827
Match Point Tennis	66 Homer Ave	Como	(817) 498-2801
Fanatical Athletes	20 Bicep Bridge Rd	Aberdeen	0224 692 903

Figura 1 - Tela de Cadastro de 'Customers'

A figura 1 ilustra a tela de cadastro que será desenvolvida neste curso. Nesta tela, podemos notar os componentes:

- Frame
- Fill-in - campos Nome, Endereço, Cidade, CEP, Telefone, e Nome - pesquisa
- Browser – Mostra os campos escolhidos (Nome, Endereço...etc) dos Customers já cadastrados no banco de dados.
- Botão – Botão Atualizar, Alterar, Inserir Customer
- Combo-box – Permite a escolha do sexo do Customer
- Radio-Button – Permite a escolha entre Brasileiro ou estrangeiro
- Máscaras de formatação – Aplicadas diretamente nos campos fill-in, permite que o valor seja preenchido/mostrado em um determinado formato. É o caso dos fill-in CEP e Telefone, os quais possuem formato character, e máscaras de texto para correta inserção dos valores CEP e Telefone.

A Figura 2 mostra a Tela onde é possível atualizar os dados cadastrais do Customer cujo mouse está selecionando na janela 1.

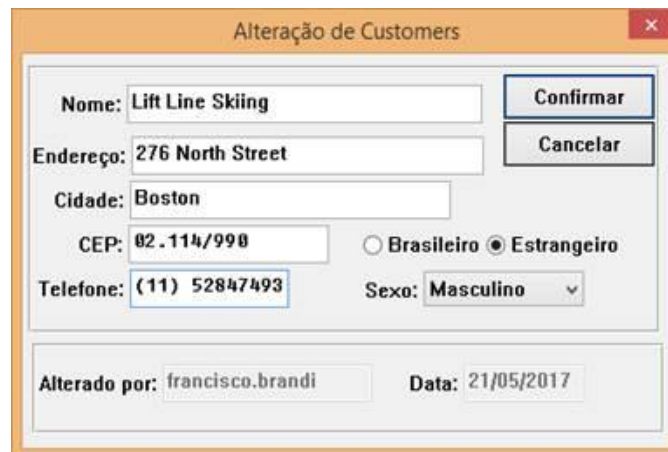


Figura 2 - Tela de Alteração Cadastral do 'Customer'

A figura 2 possui os mesmos componentes da figura 1, tendo porém, dois fill-in ("Alterado por" e "data") que estão inativos, isto é, são apenas informativos ao usuário, e não podem receber nenhum valor via interação.

#### Observações

- Algumas funcionalidades interessantes estão disponíveis para cada um dos componentes, como por exemplo, o browser, que, se habilitado, pode permitir multi-seleção de registros, pode ser redimensionável pelo usuário, ordenado por alguma coluna, possuir uma *query* própria ou ser totalmente vinculado à alguma tabela do banco de dados, além de possuir muitas possibilidades de disparo de eventos durante sua manipulação. Um exemplo disto, é que pode-se atribuir um evento (e se desejado, execute algum processo) de interações com o mouse (clique simples, duplo), interações com o teclado (exemplo da busca por nome, disponível na figura 1, onde a cada letra digitada, o browser irá filtrando os resultados), entre outros. Para acessar a janela de configuração e características, basta que seja efetuado um duplo clique em cima do componente.
- Os componentes do Progress permitem a manipulação da sua visibilidade, ou seja, mesmo – que eles existam na tela, eles podem ser ocultados e mostrados novamente, através de código.
- É possível criar validações dos dados inseridos em cada componente, através de uma trigger chamada "leave", presente na maioria dos componentes, ou pode-se implantar as validações centralizadas em um lugar, como por exemplo, ao clicar em "Inserir Customer".
- A passagem de parâmetros para a tela de alteração cadastral pode conter tanto os dados do Customer em questão, passados um a um, em variáveis distintas, como pode ser feita passando o *recid* do registro selecionado (apenas um parâmetro), obrigando a segunda tela a efetuar um find first (ou for each) na tabela Customer para posicionar o mesmo registro que está selecionado no browser da janela 1.

- Triggers são intensamente utilizadas nativamente no Progress gráfico, ou seja, fazem parte do comportamento padrão de uma janela e/ou componente.
- Procedures são intensamente utilizadas no Progress gráfico.