Capítulo 10: Exceções no Delphi

Manipulação de Erro Durante a Execução

Manipulação de error é um modo de vida no desenvolvimento do software.

Porque as aplicações devem rodar sob uma variedade de situações e condições, o programador deve fornecer um meio de manipular gentilmente os erros inevitáveis que ocorrem.

Métodos Tradicionais

Em versões do Pascal anteriores ao Delphi, tratar os erros durante a execução era terrível! Para muitos dos erros baseados no sistema operacional, o programador tinha que recorrer a diretivas de compilação e variáveis de status para detectar os problemas. Em outras palavras, o programador deveria instruir o compilador a "desligar" os relatórios de error para certas seções do código, então verificar o valor de uma variável de status global para ver o que aconteceu. Aqui está um exemplo da abertura de arquivo usando os métodos tradicionais para tratamento de erros de execução.

```
{ $I- } { esta diretiva de compilação desliga a verificação I/O}
Assign(InFile, InputName);
Reset(InFile);
{ $I+ } { esta diretiva de compilação restaura a verificação I/O}
if IOResult <> 0 then
     { o erro é processado aqui};
```

Como você pode ver, este processo nos leva a um código confuso, que é difícil de ler e de manter. Há também muita verificação de error no código, esta é a razão para o código ser tão difícil de apurar. Existem mais linhas de código acima para verificar o error do que realizar a tarefa realmente! Como você verá, manipulação de exceção estruturada no Object Pascal soluciona estes problemas.

Manipulação de Erro Usando Exceções

Manipulação de Exceção Estruturada é um novo recurso de linguagem Object Pascal. Esta extensão da linguagem tem dois aspectos. Primeiro, ela fornece a garantia que qualquer recurso alocado ou mudado na aplicação possa ser devidamente restaurado, mesmo depois de graves erros de execução. Segundo, ela representa uma maneira estruturada de lidar com erros de execução de todos os tipos, mesmo quando estes são fatais à aplicação.

As seções a seguir mostram, primeiramente, a sintaxe de exceções no Object Pascal - variedades de proteção de recurso e manipulação de exceção. Seções mais a frente mostrarão como usar exceções para tornar sua aplicação mais legível e robusta. Mais técnicas avançadas de manipulação de exceção serão exibidas em um próximo capítulo que fala sobre manipulação de exceções avançadas.

Sintaxe de Exceções

Como foi dito acima, manipulação de exceção no Delphi tem dois tipos diferentes, mas de relacionados aspectos. Embora ambas as variedades de manipulação de exceção comecem com a mesma palavra reservada ”try”, elas servem para propósitos diferentes. Primeiramente, nós veremos a sintaxe de manipulação de exceção para proteção de recursos.

Try ... finally

A sintaxe do bloco **try...finally** está abaixo:

```
try
     < declaração >;
     < declaração >;
     ...
finally
     < declaração >;
     < declaração >;
     ...
end;
```

O bloco Try...finally começa com a palavra reservada try, seguida por uma ou mais linhas de código. Note que esta é outra construção Object Pascal que começa um bloco de comando mas não requer um begin. O corpo do bloco try...finally termina com a palavra reservada finally. Finally pode ser seguida de uma ou mais linhas de código, e finalmente o bloco try...finally termina com um end. Note também que esta é outra construção Object Pascal (como case) que possui um end sem um begin correspondente.

O bloco **try...finally** fornece uma maneira de garantir que o código na porção finally do bloco será executado não imposta o que aconteça. Em outras palavras, se nenhum erro ou circunstância anormal ocorrer enquanto o bloco **try...finally** estiver sendo executado, o código da parte finally do bloco será executado no curso normal dos eventos. Então se nada der errado, o código sob finally é executado como se não houvesse um bloco **try...finally.** Contudo, o verdadeiro poder desta construção torna-se aparente quando uma condição de erro ocorrer durante a execução do código da porção try do bloco. Nesta situação, qualquer código que esteja na parte finally do bloco será executado antes que o fluxo deixe este bloco de código. Então mesmo se um erro fatal de execução ocorrer na porção try do bloco, o código na porção finally do bloco será sempre executado.

Proteção de Recurso

O comportamento do bloco **try...finally** é particularmente útil em uma série de situações diferentes. O primeiro é a proteção de recurso. Primeiro, uma pequena idéia sobre o que são recursos no Windows. O Windows é responsável por todos os recursos alocados e usados por todas as aplicações Windows. Quando uma aplicação precisa mais de um recurso em particular (ex.: memória, manipuladores, gráficos, etc.), ela pede ao Windows que o aloque e o retorne para aplicação. É então responsabilidade da aplicação desalocar este recurso quando ele não for mais necessário. Você provavelmente já viu como alocar memória dinamicamente em uma

aplicação e eventualmente deslocar a mesma. No Windows, recursos trabalham da mesma maneira.

Quando aplicações windows alocam recursos e nunca desalocam os mesmos, diz-se que houve “vazamento” de recurso. Se a aplicação não liberar o recurso, este recurso será bloqueado até que a aplicação seja encerrada. Por exemplo, se um programa pede ao Windows um manipulador gráfico (para que ele possa desenhar algo na tela) e nunca liberar o manipulador, o número de manipuladores gráficos é reduzido. Esta aplicação é considerada uma aplicação Windows mal comportada.

O bloco **try...finally** é a solução perfeita para a proteção de recursos nas aplicações Delphi. Na verdade, a regra no Delphi é: se você alocar algo, você é responsável por desalocar. Isto conclui a alocação feita quando chamamos os construtores dos objetos. Se você disparar o construtor explicitamente, você é responsável por executar o destrutor também. O bloco **try...finally** fornece uma boa maneira de garantir que o destrutor será disparado quando você tiver terminado de utilizar o objeto.

Como um exemplo, a aplicação deve exibir uma caixa de diálogo Sobre quando o usuário selecionar **Ajuda/Sobre...**. Por ser uma perda de tempo permitir que a caixa de diálogo Sobre seja criada automaticamente através do projeto, é melhor alocar os recursos para o form (ex.: chamar o seu construtor) quando nós precisarmos exibir o form e chamar o seu destrutor quando terminarmos. O código para criar, exibir e destruir o form está abaixo:

```
procedure TfrmOrderEntryMain.mntmAboutClick(Sender: TObject);
begin
  frmAbout := TfrmAbout.create( Application );
  frmAbout.ShowModal;
  frmAbout.Free;
end;
```

Como dito acima, toda vez que você chamar o construtor, você também é responsável por chamar o destrutor, o que fica evidente no código acima. Contudo, se um erro ocorrer durante a execução, enquanto esse form estiver sendo exibido na tela, o seu destrutor não será chamado e todos os recursos alocados para este form (incluindo memória e recursos gráficos) serão perdidos.

É claro que a maneira de resolver este problema é através da manipulação exceção para proteção de recursos. Aqui está a versão melhorada do código acima:

```
procedure TformOrderEntryMain.mntmAboutClick(Sender: TObject);
begin
  frmAbout := TfrmAbout.create( Application );
  try
    frmAbout.ShowModal;
  finally
     frmAbout.Free;
  end; //finally
end;
```

Esta versão do código é mais segura e mais robusta. Agora, não importa o que aconteça enquanto este form estiver ativo, o Delphi assegurar-se-á que o destrutor do form seja disparado no momento apropriado.

"Operador de Encapsulação"

Blocos **try...finally** não são necessariamente apenas para situações de proteção de recursos como acima. Sempre que você alterar um recurso e quiser se assegurar que ele seja retornado ao estado anterior ao final do bloco de código, blocos **try...finally** são uma excelente fórmula de se fazer isto. Por exemplo, se você abrir um arquivo texto dentro de uma procedure para ler alguma informação, você irá querer ter certeza de que o arquivo seja fechado de forma apropriada quando você sair da rotina. Blocos **try...finally** são uma excelente fórmula de manipular esta situação.

O bloco try...finally pode ser chamado de operador de encapsulação. Observe que utilizando blocos try...finally, você pode garantir que todas as suas procedures e funções (e qualquer outro bloco lógico de programa) sempre será executado, não importa o código de limpeza necessário para retornar as coisas como eram. Como regra, sempre que voc6e alocar qualquer recurso ou alterar qualquer aspecto da aplicação que necessite ser retornado ao estado original, utilize um bloco try...finally.

```
Assign( TextFile, TextFileName );
reset( TextFile );
try
   readln( TextFile, buf );
   { more code that reads information from the text file }
finally
   CloseFile( TextFile );
end; {finally}
```

O bloco **try...finally** pode ser chamado de operador de encapsulação. Observe que utilizando blocos **try...finally**, você p[ode garantir que todas as suas procedures e funções (e qualquer outro bloco lógico de programa) será sempre executado, não importa o código de limpeza necessário para retomar as coisas como eram.

Como regra, sempre que você alocar qualquer recurso, ou alterar qualquer aspecto de aplicação que necessite ser retornado ao estado original, utilize um bloco **try...finally**.

try...except

O outro aspecto de manipulação de exceção no Delphi corresponde ao que a maioria dos desenvolvedores acha, quando pensa sobre a manipulação de exceções: manipulação de erros de tempo de execução.

A sintaxe para o bloco **try...except** é similar ao do bloco **try...finaly**:

```
try
   < statement >;
   < statement >;
   . . .
except
```

```
   on ExceptionType1 do
      <statement(s)>;
   on ExceptionType2 do
      <statement(s)>;
end; {except)
```

Aqui está um exemplo do código atual que incorpora manipulação de exceção para recuperação de erro:

```
try
   result := x/y;
except
   on EZeroDivide do
   begin
   MessageBeep( 0 );
   MessageDLG( 'Numbers too small for valid results',
               mtError, [mbOK], 0 );
   end; {on Eunderflow}
end;
```

Observe que, como os blocos **try...finally**, os blocos try...except possuem um end sem um begin correspondente. Funcionalmente, se o código na porção try do bloco não causar qualquer problema (em outras palavras, não causar um erro de tempo de execução), o código na parte except do bloco é pulado. O que acontece quando um erro ocorrer, veremos em breve. Primeiro, vamos ver um pouco sobre a classe exception no Delphi.

Exceções no Delphi são classes, assim como qualquer outra classe no Delphi. A convenção de nomeação para exceções inclui um “E” inicial para a designação da classe, ao invés da inicial “T” para todas as outras classes. No mais, as classes de exceção são sintaticamente idênticas a todas as outras classes.

Quando um erro de tempo de execução ocorre no Delphi, o ambiente de tempo de execução do Delphi instancia (ou sejam chama o construtor e cria um objeto para) uma classe de exceção que coincida com o tipo de execução de tempo de execução ocorrido. O termo para esse processo é “raise” ou indução. Quando ocorre um erro de tempo de execução, o Delphi gera uma exceção. O fluxo do código então deixa a linha de código que causou a exceção e vai imediatamente para a parte except do bloco **try...except**. Cada uma das entradas sob o bloco **try...except** é verificada para ver se coincide com o tipo de exceção gerada. Quando uma entrada coincidente for encontrada, a execução do programa prossegue com o código sob o tipo coincidente. Uma vez que o código de manipulação de erro seja executado, o objeto exceção gerado pelo Delphi é destruído automaticamente (seu destrutor é chamado) e a execução do programa prossegue após o final do bloco **try...except** inteiro. Uma exceção é manipulada quando seu tipo coincidente é encontrado e o código de correção é executado. Observe que o Delphi não tentará re-executar automaticamente a condição que fez com que ocorresse a exceção. Você deve escrever o código para fazer isto se for apropriado re-executar o código ofensor.

Considere o seguinte pedaço de código:

```
function GetRatio( x, y : integer ) Double;
begin
  try
    Result := x/y;
  except
    on EZeroDivide do
    begin
      Result := 0.0;
      ShowMessage( ‘Cannot divide by zero’);
    end; {EZeroDivide}
    on EUnderFlow do
    begin
      Result := 0.0;
      ShowMessage( ‘Difference is too great’ );
    end; {EUnderFlow}
    on EMathError do
    begin
      Result := 0.0;
      ShowMessage( ‘General math error’ );
    end; {EMathError}
  end; {except}
end; {procedure}
```

Esta função retorna a proporção entre os dois parâmetros passados. Se o parâmetro “y” for zero, o primeiro manipulador de exceção é ativado. Se a diferença entre os números for muito grande, um outro erro ocorrerá e será endereçado de forma apropriada. Observe que erros gerais de cálculo são verificados após estes dois erros específicos.

Como seria de esperar, DivByZero e UnderFlow são herdados de MathError. No trecho de código acima, o programador está verificando primeiro os dois erros específicos, **depois verificando por todos os erros de cálculo**. Lembre-se que, por causa da forma como hierarquias de classes trabalham no Delphi, qualquer classe pai de uma classe pode atuar como um stand-in para aquela classe. Isto lhe dá a flexibilidade para verificar tipos específicos de erros primeiro, e depois verificar erros mais genéricos. Além disso, observar a ordem em que as exceções aparecem na parte except da instrução tem grande impacto - se EMathError fosse verificado primeiro, as outras duas nunca seriam executadas.

Isto também significa que você pode interceptar qualquer tipo de exceção buscando pela classe de exceção. Como este é o ancestral final de todas as exceções, ele será substituído por qualquer classe de exceção.

```
Try
  { alguma operação que possa causar uma exceção }
except
  on Exception do
    < instrução >;
end; {except}
```

Entretanto, muito embora isto seja possível, você não deve utilizar este tipo de construção. Isto reduz a manipulação estruturada da Exceção do Delphi para relembrar o endereçamento de erros do tipo *On Error Goto* de outras linguagens.

Propagação de exceções

E o que acontece se ocorrer um erro que não seja um dos tipos que estamos endereçando especificamente? O Delphi irá deixar o bloco de código que contém o manipulador da exceção imediatamente (a função, procedure, o método) e irá ao bloco que chamou o bloco atual. Propagação é o excesso de "percorrer" de volta à pilha de chamadas para encontrar um bloco de exceção que enderece esta exceção. Este processo continuará até que um manipulador de exceção seja encontrado ou o objeto Application tome o controle. O objeto Application manipula todas as exceções não manipuladas exibindo uma caixa de mensagem na tela com uma mensagem de exceção. O capítulo sobre a manipulação avançada de exceções discute a utilização do objeto Application para manipular exceções. Desta forma, por exemplo, se ocorrer um erro OutOfMemory durante a execução da função GetRatio acima, o controle do programa passaria para a procedure ou método que tivesse chamado GetRatio. Isto continuaria até que o objeto Application tomasse o controle.

Para resumir este comportamento: quando ocorre uma exceção, ela existe até que seja manipulada. Ela se propaga automaticamente pela pilha de chamadas procurando por um manipulador. Se um manipulador é encontrado, o objeto exceção é destruído e a execução continua após o bloco do manipulador da exceção. Se não for manipulada, o objeto Application a manipulará com uma caixa de diálogo genérica.

Uma nota sobre como você verá exceções expressas no ambiente do Delphi. O comportamento default no Delphi é interceptar as exceções antes que sua aplicação as receba. O IDE do Delphi lhe dá uma caixa de diálogo de "prevenção" indicando que a exceção está para ocorrer em sua aplicação. Quando esta caixa é fechada, a aplicação é interrompida na linha de código que irá causar a exceção. Você pode então executar a aplicação e ver como ela manipula a exceção. Um controle de opção controla este comportamento.

Está na página Preferences da caixa de diálogo de environment options:

Se Break on Exception estiver selecionado, então o ambiente interceptará exceções antes que sua aplicação o faça. Se Break on Exception estiver ativo, você verá a caixa de diálogo acima bem como a caixa de diálogo a seguir. Note que esta opção somente se aplica a aplicações sendo executadas dentro do ambiente do Delphi. Se for executada fora do Delphi, você sempre verá somente a segunda caixa.

Raise

Object Pascal agora possui uma palavra reservada específica para exceções: raise. Raise possui dois diferentes usos: Primeiro, raise pode ser utilizado para re-gerar uma exceção que já tinha sido endereçada. Lembre-se de que uma vez que o bloco de manipulação de exceção tenha sido chamado para endereçar a exceção, o objeto exceção é destruído automaticamente. Em algumas situações você precisará manipular

a exceção, mas ainda permitir que ela se propague para indicar à rotina chamadora que ocorreu uma exceção.

Eis um exemplo da utilização de Raise para re-gerar uma exceção. Se ocorrer um divisão por zero, você quer que a exceção se propague até a procedure chamadora. Entretanto, você também irá querer que a função nunca retorne um valor inválido. Raise permitirá que você manipule a exceção, alem de permitir que ela se propague.

```
function GetAverage( Sum : integer; NumItems : word ) : Double;
begin
  try
    result := sum / NumItems;
  except
    on EDivByZero do
    begin
      result := 0.0;  // garantir o valor válido
      raise;          // indicar o erro ao chamador
    end; {EDivByZero}
  end; {except}
end;
```

Raise também pode ser utilizado para gerar manualmente uma exceção, simulando uma condição de erro. Se você quisesse gerar uma exceção EDivByZero por si próprio, você poderia fazê-lo com a seguinte sintaxe:

```
raise EDivByZero.create( 'Division by zero' );
```

Raise pode ser utilizado desta forma para chamar manualmente o construtor para uma exceção. O parâmetro para o construtor é a mensagem default que será exibida pelo objeto application. Entretanto, mesmo que você chame o construtor com raise, você não deve chamar o destrutor. O Delphi sempre chama destrutores para exceções, não importando como foram gerados. Raise também pode ser utilizado desta forma para construir exceções definidas pelo usuário, como veremos em uma seção mais adiante. Raise também tem um papel importante no capítulo a seguir, sobre manipulação avançada de exceção, particularmente para erros de engine de banco de dados.

Misturando proteção de recursos e exceções

Você pode, é claro, misturar livremente blocos **try...finally** e blocos **try...except**. Ambos os blocos de exceção se comportam como qualquer outro bloco Object Pascal. Entretanto, você não pode iniciar os blocos t**ry...finally** e **try...except** como o mesmo try. Eis um exemplo da mistura de blocos de proteção de recursos e de manipulação de exceção:

```
function GetAverageFromFile( FileName : string ) : Double;
var
  inputFile : Text;
  buf : string;
  sum, numItems ; integer;
begin
  sum ;= 0;
  AssignFile( inputFile, FileName );
```

```
  reset( inputFile );
  try
    while ( not eof( inputfile ) ) do
    begin
      readln( inputFile, buf );
      sum := StrToInt( buf );
      inc( numItems );
    end;
    try
      result := sum / numItems;
    except
      on EDivByZero do
      begin
        result := 0.0;
        raise;
      end; {EDivByZero}
    end; {except}
  finally
    CloseFile( inputFile );
  end; {finally}
end;
```

Neste exemplo, se qtdItens for zero, o valor do retorno da função é configurado para 0.0; mas a exceção é re-gerada para indicar ao chamador que o erro ocorreu. Observe que se ocorrer o erro, o Delphi executará o código no bloco finally antes de propagar para a procedure chamadora. Código em um bloco finally é **sempre** executado antes da saída do bloco. No caso acima, você pode descansar tranqüilo, porque seu arquivo sempre é fechado de forma apropriada em qualquer circunstância.

Exceções definidas pelo usuário

Como vimos, exceções são a forma preferida de manipular erros em aplicações Delphi. Entretanto, você só viu como utilizar as classes de exceção pré-definidas. Como exceções são simplesmente classes, você pode criar suas próprias exceções personalizadas através de uma sub-classe de uma das classes pré-definidas.

Criando Sub-classes da Classe Exceção

Para criar uma exceção personalizada, você só precisa criar uma sub-classe da classe exception. Por exemplo:

```
type
   EInvalidBalance = class( Exception );
```

Se você não quiser adicionar suas próprias propriedades, o acima será suficiente. Isto cria uma exceção que se comporta como exceções pré-definidas (com propagação automática e destruição).

Isto é essencialmente uma é essencialmente uma renomeação da classe exception, mas é uma boa idéia criar exceções definidas pelo usuário como esta, ao invés de utilizar o objeto exception. Quando você gera esta exceção, você pode interceptá-la

pelo nome. Se você tenta interceptar somente pelo tipo da exceção, você inadvertidamente irá interceptar todas as exceções - e não apenas a exceção na qual está interessado.

Gerando Exceções Personalizadas

Delphi, é claro, não irá gerar exceções personalizadas. O programador é responsável por gerar este tipo de exceções. A sintaxe é similar ao que foi apresentado acima, na geração de exceções pré-definidas. A única diferença é a classe personalizada ao invés da classe existente.

```
Raise EInvalidBalance.create( 'Invalid account balance' );
```

Novamente, o parâmetro é a mensagem que será exibida pelo objeto Application se esta exceção se propagar até ele. E assim como com exceções pré-definidas, você não deve chamar o destrutor da exceção - o Delphi sempre o chama automaticamente quando a exceção for manipulada.

A utilização de exceções personalizadas pode não ser aparente imediatamente. Entretanto, ela tem um papel importante no ambiente dirigido a eventos do Windows. O benefício real das exceções personalizadas tornar-se-á aparente no capítulo de Exceções Avançadas.

Capítulo 11: Validação de Dados

O que veremos neste capítulo:

- Validação de dados de campo.
- Manipulando validação server-side.

Validação Client-Side

Validação de dados client-side basicamente significa uma forma de controlar a validação da entrada de dados em sua aplicação antes de retornar valores ao servidor do banco de dados. Este tipo de validação é uma boa razão para criar objetos TField. A validação assegura que o dado em um campo siga uma determinada condição, qualquer que seja o formato ou valor que você tenha definido. Por exemplo, você pode querer se assegurar que o número digitado para um preço esteja dentro de uma determinada faixa de valores. Você pode utilizar o evento OnValidate do objeto TField para anexar estas regras de validação aos campos.

Vamos criar uma nova aplicação, utilizando o banco de dados OrderEntry. Ela terá um único form que exiba os dados da tabela Customer em um DBGrid. Coloque um componente DBNavigator no form para permitir a navegação pela tabela Customer. Agora podemos definir uma rotina de validação para o campo ACCOUNT_BALANCE. Selecione o objeto tblCustomerACCOUNT_BALANCE adicionando primeiro os campos no Fields Editor e depois selecionando-o.

No Object Inspector, dê um duplo clique no evento OnValidate para criar um novo manipulador de evento:

Como o evento de validação é como qualquer outro evento, você pode utilizar qualquer código. Object Pascal válido nele. Esta rotina de validação assegura que o usuário nunca entre em contato com um saldo menor que 0:

```
procedure TfrmCustomer.tblCustomerACCOUNT_BALANCEValidate(
  Sender: TField);
begin
  if tblCustomerAccount_Balance.value < 0.0 then
    raise EInvalidBalance.Create( 'Balances cannot be
     negative' );
end;
```

O código acima compara o valor do campo tblCustomerACCOUNT_BALANCE COM 0. Se o valor do campo for menor que 0, a rotina gera uma exceção `EInvalidBalance`.

Observe que esta rotina gera uma exceção `EInvalidBalance` se o usuário digitar um número negativo. Devemos definir esta exceção na seção `type` da unit.

```
EInvalidBalance = class(Exception );
```