Aplicações Robustas

O tratamento de exceção é um mecanismo capaz de dar robustez a uma aplicação, permitindo que os erros sejam manipulados de uma maneira consistente e fazendo com que a aplicação possa se recuperar de erros, se possível, ou finalizar a execução quando necessário, sem perda de dados ou recursos.

Para que uma aplicação seja segura, seu código necessita reconhecer uma exceção quando esta ocorrer e responder adequadamente a essa exceção. Se não houver tratamento para uma exceção, será exibida uma mensagem padrão descrevendo o erro e todos os processamentos pendentes não serão executados. Uma exceção deve ser respondida sempre que houver perigo de perda de dados ou de recursos do sistema.

Exceções

Exceções são classes definidas pelo Delphi para o tratamento de erros. Quando uma exceção é criada, todos os procedimentos pendentes são cancelados e, geralmente é mostrada uma mensagem de erro para o usuário. As mensagens padrão nem sempre são claras, por isso é indicado criar seus próprios blocos protegidos.

Blocos Protegidos

Um bloco protegido é um grupo de comandos com uma seção de tratamento de exceções.

```
try
  A := StrToFloat(EdtA.Text);
  B := StrToFloat(EdtB.Text);
  ShowMessage(Format('%f / %f = %f', [A, B, A + B]));
except
  ShowMessage('Números inválidos.');
end;
```

Algumas vezes você pode precisar especificar quais exceções quer tratar, como mostrado abaixo.

```
try
  Soma := StrToFloat(EdtSoma.Text);
  NumAlunos := StrToInt(EdtNum.Text);
  ShowMessage(Format('Média igual a %f.', [Soma / NumAlunos]));
except
  on EConvertError do
    ShowMessage('Valor inválido para soma ou número de alunos.');
  on EZeroDivide do
    ShowMessage('O número de alunos tem que ser maior que zero.');
  else
    ShowMessage('Erro na operação, verifique os valores digitados.');
end;
```

Principais Exceções

O Delphi define muitas exceções, para cada erro existe uma exceção correspondente.

Classe	**Descrição**
Exception	Exceção genérica, usada apenas como ancestral de todas as outras exceções
EAbort	Exceção silenciosa, pode ser gerada pelo procedimento Abort e não mostra nenhuma mensagem
EAccessViolation	Acesso inválido à memória, geralmente ocorre com objetos não inicializados
EConvertError	Erro de conversão de tipos
EDivByZero	Divisão de inteiro por zero
EInOutError	Erro de Entrada ou Saída reportado pelo sistema operacional
EIntOverFlow	Resultado de um cálculo inteiro excedeu o limite
EInvalidCast	TypeCast inválido com o operador as
EInvalidOp	Operação inválida com número de ponto flutuante
EOutOfMemory	Memória insuficiente
EOverflow	Resultado de um cálculo com número real excedeu o limite
ERangeError	Valor excede o limite do tipo inteiro ao qual foi atribuída
EUnderflow	Resultado de um cálculo com número real é menor que a faixa válida
EVariantError	Erro em operação com variant
EZeroDivide	Divisão de real por zero
EDatabaseError	Erro genérico de banco de dados, geralmente não é usado diretamente
EDBEngineError	Erro da BDE, descende de EDatabaseError e traz dados que podem identificar o erro

Blocos de Finalização

Blocos de finalização são executados sempre, haja ou não uma exceção. Geralmente os blocos de finalização são usados para liberar recursos.

```
FrmSobre := TFrmSobre.Create(Application);
try
  FrmSobre.Img.LoadFromFile('Delphi.bmp');
  FrmSobre.ShowModal;
finally
  FrmSobre.Release;
end;
```

Você pode usar blocos de proteção e finalização aninhados

```
FrmOptions := TFrmOptions.Create(Application);
try
  FrmOptions.ShowModal;
  try
    Tbl.Edit;
    TblValor.AsString := EdtValor.Text;
  except
    on EDBEngineError do
      ShowMessage('Alteração não permitida.');
    on EConvertError do
      ShowMessage('Valor inválido.');
  end;
finally
  FrmOptions.Release;
end;
```

Geração de Exceções

Você pode provocar uma exceção usando a cláusula *raise*.

```
raise EDatabaseError.Create('Erro ao alterar registro.');
```

Também é possível criar seus próprios tipos de exceções.

```
type
  EInvalidUser = class (Exception);

raise EInvalidUser.Create('Você não tem acesso a essa operação.');
```

Se você quiser que uma exceção continue ativa, mesmo depois de tratada, use a cláusula *raise* dentro do bloco de tratamento da exceção. Geralmente isso é feito com exceções aninhadas.

```
try
  Tbl.Edit;
  TblContador.Value := TblContador.Value + 1;
  Tbl.Post;
except
  ShowMessage('Erro ao alterar contador.');
  raise;
end;
```

Erros de Bancos de Dados

A exceção EDBEngineError permite a identificação de erros de bancos de dados gerados pela BDE.

```
try
  TblCli.Post;
except
  on E: EDBEngineError do
    if E.Errors[0].ErrorCode = DBIERR_KEYVIOL then
      ShowMessage('Cliente já cadastrado.');
end;
```

Note que a variável E, que vai identificar o erro, só precisa ser declarada no bloco de tratamento da exceção. No help você pode consultar outras propriedades de EDBEngineError que podem ser importantes.

Você também pode usar os eventos de erro do componente Table, sem precisar de blocos de tratamento.

```
procedure TFrmCadCli.TblCliPostError(DataSet: TDataSet; E: EDatabaseError;
  var Action: TDataAction);
begin
  if(E is EDBEngineError) then
    with EDBEngineError(E) do
      case Errors[0].ErrorCode of
        DBIERR_KEYVIOL: ShowMessage('Cliente já cadastrado.');
        DBIERR_REQDERR: ShowMessage('Campo obrigatório não preenchido.');
      end
  else
    ShowMessage('Erro no banco de dados:' + #13#13 + E.Message);
  Action := daAbort;
end;
```

Alguns códigos de erro da BDE estão listados abaixo. Todas as constantes e funções relacionadas à API da BDE no Delphi 3 estão na Unit BDE, que deve ser adicionada à cláusula uses. No BDE API Help você pode encontrar referência sobre as funções nativas da BDE, como também alguns exemplos em Delphi.

Constante	**Descrição**
DBIERR_KEYVIOL	Violação de chave primária
DBIERR_MAXVALERR	Valor máximo excedido
DBIERR_FORIEGNKEYERR	Erro de chave externa, como em integridade referencial
DBIERR_LOCKED	Registro travado
DBIERR_FILELOCKED	Arquivo travado
DBIERR_NETMULTIPLE	Mais de um diretório usado como NetFileDir

DBIERR_MINVALERR	Campo com valor mais baixo que valor mínimo
DBIERR_REQDERR	Campo obrigatório faltando
DBIERR_LOOKUPTABLEERR	Erro em tabela Lookup

Se você quiser mais informações a respeito do erro pode usar o procedimento DBIGetErrorContext, como na função mostrada abaixo que retorna determinadas informações sobre o erro.

```
function GetErrorInfo(Context: SmallInt): string;
begin
  SetLength(Result, DBIMAXMSGLEN + 1);
  try
    DbiGetErrorContext(Context, PChar(Result));
    SetLength(Result, StrLen(PChar(Result)));
  except
    Result := '';
  end;
end;
```

No evento OnEditError, usado no exemplo abaixo, se ocorrer um erro ao tentar alterar um registro, podemos identificar o usuário da rede que está alterando esse registro usando a função criada anteriormente.

```
if Pos('locked', E.Message) > 0 then
  ShowMessage('Usuário ''' + GetErrorInfo(ecUSERNAME) + ''' está alterando o registro.');
```

Note que foi usada uma outra técnica de identificação do erro, usando a própria mensagem de erro e não o código, como mostrado anteriormente. Você pode usar a função criada acima mandando como parâmetro os valores mostrados abaixo, que podem ser encontrados no help da BDE.

Constante	**Descrição**
ecTABLENAME	Nome da Tabela
ecFIELDNAME	Nome do campo
ecUSERNAME	Nome do usuário, muito usado para identificar qual usuário travou o registro
ecFILENAME	Nome do arquivo
ecINDEXNAME	Nome do índice
ecDIRNAME	Pasta
ecKEYNAME	Chave primária
ecALIAS	Alias
ecDRIVENAME	Drive
ecNATIVECODE	Código de erro nativo
ecNATIVEMSG	Mensagem de erro nativa
ecLINENUMBER	Número da linha, usado em instruções SQL

Para desenvolver um sistema genérico de tratamento de erros, considere a opção de criar esse tratamento em um DataModule genérico para ser usado como ancestral por todos os DataModules do sistema, utilizando a herança visual.

Se o único problema for traduzir as mensagens, localize os arquivos CONSTS.INT e DBCONSTS.INT e crie uma nova Unit de definição de strings com uma estrutura semelhante a mostrada abaixo e juntando todas as definições das constantes das duas Units devidamente traduzidas. Depois, basta usar essa Unit em seus projetos que as novas mensagens irão sobrepor as anteriores.

```
unit NewConsts;

interface

resourcestring
  SAssignError = 'Não é possível atribuir %s a %s';
  SFCreateError = 'Não é possível criar arquivo %s';
  SFOpenError = 'Não é possível abrir arquivo %s';
  SInvalidFieldSize = 'Tamanho de campo inválido';
  SInvalidFieldRegistration = 'Registro de campo inválido';
  SUnknownFieldType = 'Campo ''%s'' tem um tipo desconhecido';

implementation

end.
```

Uma outra opção seria criar um método para o evento OnException do objeto Application, esse método seria chamado sempre que houvesse uma exceção em qualquer parte do sistema.