Manejo de Excepciones en Delphi

Autor: Aquino, Germán.

Fecha: 24 de febrero de 2012

Índice

Introducción..... 3
Sintaxis..... 4
El bloque try/except..... 4
Sin manejadores específicos..... 4
Con manejadores de clases de excepción..... 5
Con manejadores de clases de excepción e identificadores para las excepciones..... 6
El bloque try/finally..... 8
Flujo de control de las excepciones..... 10
Creación explícita de excepciones..... 11
La sentencia *raise*..... 11
Los constructores de la clase *Exception*..... 12
Excepciones definidas por el usuario..... 14
Apéndice - La jerarquía de *Exception*..... 16
Bibliografía..... 17

Introducción

Una excepción es una condición de error u otro evento que interrumpe el flujo normal de ejecución en una aplicación. Las excepciones hacen que los programas sean más robustos ya que proporcionan un modo estándar de notificar y gestionar errores y situaciones inesperadas. Además las excepciones hacen que los programas sean más fáciles de escribir, leer y depurar porque permiten separar el código de gestión de errores de código normal.

Delphi lanza una excepción mediante la creación de un objeto *Exception*, cuando un error u otro evento interrumpe la ejecución normal de la aplicación. Como las excepciones son objetos, se pueden agrupar en jerarquías usando herencia, y se pueden introducir nuevas excepciones sin afectar el código existente.

Una excepción puede llevar información, como un mensaje de error, del punto en el que se lanzó al punto en el que es manejada. Cuando una aplicación usa la unidad *SysUtils* a través de la cláusula *uses*, todos los errores en tiempo de ejecución son convertidos automáticamente en excepciones. De esta forma errores que de otra forma hubieran causado la terminación de la aplicación – como memoria insuficiente, división por cero, y fallos generales de protección como acceso a punteros nulos – pueden ser tratados.

La excepción transfiere el control a un manejador, separando así la lógica del programa de la lógica para el tratamiento de errores. Si la aplicación no cuenta con un manejador de excepciones, se ejecutará el manejador por defecto de Delphi, el cual reporta el error con un mensaje e intenta continuar con la ejecución del programa.

El mecanismo de manejo de excepciones se basa en cuatro palabras clave:
- **try**: Delimita el comienzo de un bloque protegido de código.
- **except**: Delimita el final de un bloque protegido de código e introduce las sentencias de control de excepciones.
- **finally**: Se usa para especificar bloques de código que deben ejecutarse siempre, incluso cuando se producen excepciones. Este bloque se usa generalmente para realizar operaciones de limpieza que siempre se deberían ejecutar, como cerrar archivos o tablas de bases de datos, liberar objetos, liberar memoria y otros recursos adquiridos en el mismo bloque de programa.
- **raise**: Es la sentencia usada para generar una excepción. La mayoría de las excepciones que encontramos en Delphi las genera el sistema, pero también se pueden crear excepciones propias en el código, cuando se descubren datos no válidos o incoherentes en tiempo de ejecución. La palabra clave ***raise*** también puede usarse dentro de un manejador para volver a lanzar una excepción, es decir, para propagarla al siguiente manejador.

Sintaxis

El bloque try/except

Una aplicación responde a una excepción ejecutando algún código de terminación, manejando la excepción, o ambas cosas. La forma de manejar una excepción producida en el código consiste en que la excepción ocurra dentro de un bloque de sentencias protegido.
Delphi provee una construcción sencilla para proteger código con sentencias para el manejo de excepciones, el bloque ***try/except***. Cuando una sentencia dentro del bloque protegido lanza una excepción, el flujo de control pasa al código de manejo de excepciones. Al finalizar la ejecución del manejador, el bloque ***try/except*** termina y el control pasa a la sentencia siguiente que sigue al bloque. El control nunca vuelve a la sentencia que originó la excepción.

El bloque ***try/except*** se puede escribir de diferentes formas:

1 – La forma más simple es:

```
Try
   //Sentencias
Except
   //Sentencias
End;
```

Llamemos bloque ***try*** al bloque formado por las sentencias entre las palabras clave ***try*** y ***except***, y bloque ***except*** al bloque formado por las sentencias entre las palabras clave ***except*** y ***end***.
En este tipo de manejador, las sentencias del bloque ***except*** se ejecutarán sólo si se produce una excepción en el bloque ***try***. El manejador se va a ejecutar para cualquier excepción que surja, pero no se va a contar con ninguna información sobre la excepción lanzada, ni siquiera el tipo particular de excepción.
Un ejemplo de esta forma de usar el bloque ***try/except*** es el siguiente:

```
Procedure TForm1.Button1Click(Sender: TObject);
Var
 numero, cero : Integer;
Begin
 // Trata de dividir un entero por cero para lanzar una excepción
 Try
   cero   := 0;
   numero := 1 div cero;
   ShowMessage(' numero / cero = '+IntToStr(numero));
 Except
   ShowMessage('Se ha encontrado un error desconocido.');
 End;
End;
```

2 – Para saber el tipo de excepción lanzada y poder manejar excepciones de diferentes tipos, se puede escribir el bloque try de la siguiente forma:

```
Try
  // Sentencias
Except
  On tipo_de_excepcion Do
        // Sentencias
  On tipo_de_excepcion2 Do
        // Sentencias
  On tipo_de_excepcion3 Do
        // Sentencias
End;
```

Donde ***tipo_de_excepcion*** es la clase *Exception* o alguna de sus subclases, como *EDivByZero* (división entera por cero), *EZeroDivide* (división flotante por cero), *EAccessViolation* (acceso a un puntero nulo), o *EConvertError* (error de conversión de tipos).

En este tipo de bloque ***try/except***, se incluyen uno o más manejadores, cada uno de ellos de un tipo específico de excepción. Los manejadores comienzan con la palabra clave ***On***, y la sentencia o el bloque de sentencias asociados a cada uno de ellos comienza con la palabra clave ***Do***.

Se pueden incluir tantos manejadores como se quiera. El flujo de control pasa a través de los manejadores de forma secuencial, como en una sentencia ***case***, chequeando en cada uno si la clase de excepción lanzada coincide la clase de excepción declarada en el manejador, o es una subclase de la misma. Si esto se cumple para algún manejador, se ejecutan sus sentencias asociadas, y el bloque **try/except** termina su ejecución. Si ninguno de los manejadores corresponde a la excepción lanzada, la excepción se propaga los bloques que contengan a la sentencia **try/except,** y eventualmente se propaga a la rutina invocadora. Más adelante se detallará el flujo de control de las excepciones y su propagación.

Como todas las excepciones predefinidas son subclases de *Exception*, se puede incluir el manejador ***on*** *Exception* ***do*** *Sentencias,* que capturará cualquier excepción no manejada por los manejadores anteriores del bloque ***try/except***.
Sin embargo, como el flujo de control es secuencial y este manejador captura cualquier tipo de excepción, los manejadores que le sigan nunca se ejecutarán. Es por esta razón que este manejador, si se incluye, debe incluirse al final de la lista de manejadores del bloque.
También puede usarse opcionalmente la palabra clave **else**, como otra forma de capturar cualquier excepción no considerada por los manejadores anteriores del bloque ***try/except***:

```
Try
  // Sentencias
Except
  On tipo_de_excepcion Do
        // Sentencias
  On tipo_de_excepcion2 Do
        // Sentencias
  On tipo_de_excepcion3 Do
        // Sentencias
  Else
        // Sentencias
End;
```

Un ejemplo de esta forma de usar el bloque ***try/except*** es el siguiente:

```
Procedure TForm1.Button1Click(Sender: TObject);
var
  A, B: Integer;
begin
  {Se intenta convertir el contenido de los Edit 1 y 2 a enteros
  y almacenarlos en las variables A y B respectivamente, y se intenta
  dividir A por B y colocar el resultado en el Label 1.
  El código siguiente puede producir una excepción si B es cero,
  o si la conversión a entero del contenido de alguno de los Edit falla.}
  Try
    A := StrToInt(Edit1.text);
    B := StrToInt(Edit2.text);
    Label1.Caption := IntToStr(A div B);
   Except
    on EDivByZero do begin
    //Se maneja la excepción de división por cero.
       ShowMessage('No se puede dividir por cero.');
       Label1.Caption := IntToStr(0);
    end;
    on EConvertError do begin
    // Se maneja la excepción de error en la conversión.
        ShowMessage('Uno de los operandos no es un número válido.');
        Label1.Caption := 'Error';
    end
    else
     // Se maneja cualquier otra excepción.
        ShowMessage('Error desconocido.');
   end;
end;
```

3 – Finalmente, se puede utilizar un identificador para acceder a las propiedades del objeto *Exception* que es creado automáticamente por Delphi cuando se produce la excepción:

```
Try
   // Sentencias
Except
   On E : tipo_de_excepcion Do
            // Sentencias
   On E: tipo_de_excepcion2 Do
            // Sentencias
   On E: tipo_de_excepcion3 Do
            // Sentencias
End;
```

Donde E es un identificador cuyo alcance comprende el bloque de sentencias asociado al manejador donde E se declara.

Es por esta razón que diferentes manejadores pueden utilizar el mismo identificador E sin conflictos de nombres, aún cuando E pertenezca a diferentes tipos de excepciones en diferentes contextos.

En esta versión del bloque ***try/except***, además de poder identificar diferentes tipos de excepciones, se pueden acceder a diferentes propiedades de los objetos *Exception*, como por ejemplo la clase de excepción (*E.className*) y la descripción (*E.message*). Ambas propiedades son utilizadas por el *debugger* de Delphi para informar al programador sobre la excepción lanzada.

Nota: El *debugger* de Delphi captura cualquier excepción lanzada, incluso las que tengan un manejador correspondiente. El *debugger* informa la clase y la descripción de la excepción, y da la opción de interrumpir la ejecución del programa (*Break*), o continuar con la ejecución normal (*Continue*). Si la aplicación es ejecutada fuera del *debugger*, la excepción será manejada sólo una vez, ya sea por el manejador apropiado si lo hay, o por el manejador por defecto de Delphi si no hay otro disponible.

En Delphi 2010, si se quieren deshabilitar las notificaciones del *debugger*, se puede ir a *Tools > Options > Debugger Options > Language Exceptions* y destildar el checkbox *"Notify on language exceptions"*.

Las propiedades *message* y *className* son inicializadas en el constructor de la clase a la que pertenezca el objeto *Exception* apropiado, que es invocado implícitamente al producirse la excepción. Más adelante se va a detallar cómo se crean explícitamente las excepciones, ya sean propias del lenguaje o definidas por el usuario.

Un ejemplo de esta forma de usar el bloque ***try/except*** es el siguiente, reescribiendo el ejemplo de la sección anterior:

```
Procedure TForm1.Button1Click(Sender: TObject);
var
  A, B: Integer;
begin
  {Se intenta convertir el contenido de los Edit 1 y 2 a enteros
  y almacenarlos en las variables A y B respectivamente, y se intenta
  dividir A por B y colocar el resultado en el Label 1.
  El código siguiente puede producir una excepción si B es cero,
  o si la conversión a entero del contenido de alguno de los Edit falla.}
  Try
    A := StrToInt(Edit1.text);
    B := StrToInt(Edit2.text);
    Label1.Caption := IntToStr(A div B);
   Except
    on E : EDivByZero do begin
    //Se maneja la excepción de división por cero.
       ShowMessage('Se produjo un error de tipo: ' + E.ClassName);
       Label1.Caption := IntToStr(0);
    end;
    on E : EConvertError do begin
```

```
    // Se maneja la excepción de error en la conversión.
       ShowMessage('El mensaje de error es:' + E.Message);
       Label1.Caption := 'Error';
    end;
    on E : Exception do
    // Se maneja cualquier otra excepción.
    ShowMessage('Error desconocido:' + E.ClassName + ' ,con mensaje de error: ' + E.Message);
end;
```

Es importante notar que al utilizar ***On*** *E : Exception* ***Do*** en lugar de ***else*** se puede saber a qué clase pertenece la excepción, así como también se puede acceder al mensaje de error correspondiente. El efecto es el mismo, ambas alternativas capturan cualquier excepción no contemplada por los demás manejadores, pero en la opción que utiliza ***else*** no se tiene ninguna información sobre la excepción lanzada.

El bloque try/finally

Bajo circunstancias normales, el programador puede asegurar que una aplicación libera los recursos alocados, como por ejemplo discos, tablas de bases de datos, archivos, *handles* de Windows, etc., incluyendo el código apropiado para hacerlo. Como una excepción pasa el control a un manejador fuera del bloque protegido donde ocurre el error, es necesario asegurarse de que la aplicación libere los recursos en todos los casos, sin importar si el código lanza o no una excepción. Delphi provee para ello el bloque ***try/finally***, cuya sintaxis es:

```
Try
  //Sentencias
Finally
  //Sentencias
End;
```

En lugar de ejecutarse cuando ocurre una excepción, el bloque ***finally*** es ejecutado siempre después del bloque ***try,*** aun en los casos en los que el bloque ***try*** no termina su ejecución debido a una excepción. En el bloque ***finally*** se coloca el código para liberar recursos, o cualquier otra actividad que deba realizarse en todos los casos, aún en presencia de una excepción. El bloque ***finally*** no captura la excepción, la cual se propagará.
Un ejemplo del uso de un bloque ***try/finally*** es el siguiente:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 numero, cero : Integer;
begin
 // Se intenta dividir un entero por cero – para lanzar una excepción.
 numero := -1;
 Try
  cero   := 0;
  numero := 1 div cero;
  ShowMessage('numero / cero = ' + IntToStr(numero));
```

```
  Finally
    if numero = -1 then
    begin
      ShowMessage(' La variable "numero" no tiene un valor asignado – se usará el default');
      numero := 0;
    end;
  end;
end;
```

Si se ejecuta el código anterior, se observa que el mensaje del bloque ***finally*** se visualiza antes que el mensaje de error predefinido de la excepción *("Division by zero.")*, el cual corresponde al manejador por defecto de Delphi, dado que no se encontró un manejador apropiado para esta excepción. Cuando se produce una excepción en el bloque ***try***, el código del bloque ***finally*** es ejecutado antes de propagar la excepción.
Un bloque ***try*** puede estar seguido por un bloque ***except*** o por un bloque ***finally***, pero no por ambos.
Hay situaciones en las que se necesita una estructura como la siguiente:

```
Try
  // Sentencias
Except
  // Sentencias
Finally
  // Sentencias
End;
```

La forma de simular esta estructura en Delphi es usar bloques ***try*** anidados:

```
Try
  Try
    // Sentencias
  Except
    // Sentencias
  End;
Finally
  // Sentencias
End;
```

Los bloques **try/except** y **try/finally** se pueden anidar de forma arbitraria.

Podemos reescribir el ejemplo anterior usando bloques anidados, pero para preservar la semántica del ejemplo anterior de que el código del bloque **finally** se ejecute antes que el manejador de la excepción, anidamos el bloque **try/finally** dentro del bloque **try/except**:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  numero, cero: integer;
begin
  // Se intenta dividir un entero por cero – para lanzar una excepción.
  numero := -1;
  Try
    try
      cero := 0;
      numero := 1 div cero;
      ShowMessage('numero / cero = ' + IntToStr(numero));
    Finally
      if numero = -1 then
      begin
        ShowMessage( 'La variable "numero" no tiene un valor asignado – se usará el default.');
        numero := 0;
      end;
    end;
  except
    on E: EDivByZero do
      ShowMessage('No se puede dividir por cero.');
  end;
end;
```

Controlar la excepción es generalmente menos importante que utilizar los bloques ***finally***, dado que Delphi puede manejar a la mayoría de las excepciones y reanudar el funcionamiento normal del programa. Además, demasiados bloques para controlar excepciones en el código probablemente indican errores en el flujo del programa y una mala comprensión de la función de las excepciones en el lenguaje. Por esa razón, el uso de bloques ***try/finally*** es más común que el uso de bloques ***try/except***, y una prueba de ello es que el editor de texto de Delphi autocompleta los bloques ***try*** con bloques ***finally***.

Flujo de control de las excepciones

Si una excepción es lanzada durante la ejecución del bloque de sentencias del bloque ***try***, ya sea por una de las sentencias del bloque o por una llamada a procedimiento o función dentro del mismo, se intenta manejar la excepción de la siguiente forma:

1 – Si hay manejadores en el bloque ***except*** que coincidan con la excepción, el control pasa al primero de tales manejadores. Un manejador coincide con la excepción si la clase de la excepción lanzada es igual a la clase de excepción declarada en el manejador, o es subclase de ella.
Por esta razón, los manejadores de excepciones que sean subclases de otras excepciones deben colocarse antes que los manejadores de estas últimas, es decir que los manejadores de excepciones más específicas deben colocarse antes que los manejadores de excepciones más generales.

2 – Si no se encuentra un manejador, el control pasa a las sentencias de la cláusula ***else***, si está presente.
3 – Si el bloque ***except*** está formado únicamente por una secuencia de sentencias sin ningún manejador (como en la forma 1 del bloque ***try/except***), el control pasa a la primera sentencia de la secuencia.
4 – Si no se satisface ninguna de las condiciones anteriores, la búsqueda continúa en el bloque ***except*** del bloque ***try/except*** contenedor más cercano, es decir, el bloque ***try/except*** al que el flujo de control haya entrado más recientemente y todavía no haya salido. En estos casos se dice que la excepción se propaga: la unidad (procedimiento o función) que lanzó la excepción y no la manejó termina su ejecución, y la unidad que la haya invocado debe tratar la excepción como si la hubiera lanzado ella misma.
5- Si no se encuentra ningún manejador, cláusula ***else***, ni bloque de sentencias en el bloque ***try/except*** contenedor, la búsqueda continúa en el siguiente bloque ***try/except*** contenedor.
6- Si el bloque ***try/except*** más externo es alcanzado y la excepción no fue manejada, se ejecuta el manejador de excepciones de Delphi, que muestra un mensaje de error apropiado a la excepción y trata de continuar con la ejecución normal del programa.
Como el flujo de control en un programa Delphi es dirigido por eventos, generalmente una excepción no manejada termina la ejecución del método de un componente que responde a un evento, con lo que el programa, después de mostrar un mensaje de error, quedará en el estado de espera del siguiente evento.

Cuando la excepción es manejada, la pila de ejecución del programa vuelve hasta el procedimiento o función que contiene el bloque ***try/except*** donde se maneja la excepción, y el control es transferido al manejador, cláusula ***else***, o bloque de sentencias en cuestión.
Este proceso descarta de la pila de ejecución todas las llamadas a procedimiento o función que hayan ocurrido pasando el punto en el que la excepción es manejada, es decir, las funciones y procedimientos que propagaron la excepción son removidos de la pila de ejecución. El objeto *Exception* es destruido automáticamente a través de una llamada a su Destructor (método Destroy) y el control pasa a la sentencia que sigue al bloque ***try/except*** que manejó la excepción.

Creación explícita de excepciones

La sentencia *raise*

Las excepciones que surgen de errores en tiempo de ejecución son transformadas automáticamente por Delphi en objetos *Exception*, pero el programador también puede lanzar excepciones explícitamente mediante la sentencia ***raise***.
La sentencia ***raise*** puede usarse sin parámetros dentro de un bloque ***except***:

```
Try
    // Sentencias
Except
   // Sentencias
   Raise;
End;
```

Usada de ésta forma, la sentencia ***raise*** vuelve a lanzar la excepción que haya ocurrido en el bloque ***try***, para que además de ser manejada por el manejador actual, sea también manejada por un manejador de un nivel más alto (que contenga al manejador actual o que haya invocado al procedimiento o función que define al manejador actual).

La sentencia ***raise*** puede invocarse con un parámetro, que típicamente es un objeto *Exception*. En esta forma la sentencia ***raise*** puede usarse en cualquier parte. El parámetro pasado a ***raise*** puede ser de un objeto de cualquier tipo, pero se recomienda que sea una subclase de *Exception*.
El objeto *Exception* puede ser creado al mismo tiempo que se lanza, como por ejemplo en:

```
Raise Exception.Create('Error');
```

Para crear una excepción de una clase particular se debe invocar algún constructor de dicha clase. En el ejemplo se utilizó el constructor *Create* de *Exception* que recibe como parámetro un String. Es este String el mensaje que se puede recuperar con la propiedad *Message* del objeto *Exception*, como vimos antes.
La sentencia *raise* no retorna el control a la instrucción siguiente como lo hace una sentencia normal, sino que pasa el control al bloque ***except*** asociado al bloque ***try*** donde se haya ejecutado, si lo hay. La invocación de ***raise*** se comporta exactamente igual que una excepción lanzada automáticamente por Delphi, y sigue el mismo flujo de control que vimos anteriormente.

La sentencia ***raise*** puede recibir un segundo parámetro, un puntero, usado para sobreescribir la dirección de memoria del objeto *Exception* lanzado. En este caso la sentencia ***raise*** tiene la sintaxis:

```
Raise objeto_excepcion At puntero
```

Los constructores de la clase *Exception*

Los tipos *Exception* se declaran como cualquier otra clase. En realidad, es posible usar una instancia de cualquier clase como una excepción, pero como dijimos antes se recomienda utilizar una subclase de *Exception*, la cual está definida en la unidad *SysUtils*.
La unidad *SysUtils* declara varias rutinas estándares para manipular excepciones, como *ExceptObject*, *ExceptAddr*, y *ShowException*. *SysUtils* y otras unidades VCL (*Visual Component Library*) también incluyen docenas de clases de excepciones, de las cuales todas (excepto *OutlineError*) son subclases de *Exception*.
La clase *Exception* tiene una serie de constructores que proveen a los objetos *Exception* propiedades como *Message* y *HelpContext* que se pueden usar para tener una descripción del error y un ID de ayuda contextual.

En Delphi los constructores por convención se llaman *Create*. Sin embargo, una clase puede tener más de un constructor, cada uno con un identificador arbitrario.
La clase *Exception* tiene ocho constructores, que pueden ser usados con ***raise***, descriptos a continuación:

* El más comúnmente usado es *Create*, el cual tiene un solo argumento, *Msg*, un String que representa el mensaje que será mostrado al usuario cuando la excepción sea lanzada.

```
Raise Exception.Create('Mensaje de Error');
```

* *CreateFmt* es similar a *Create* pero permite dar formato especial al el mensaje de error. Tiene dos argumentos, un String que contiene la plantilla del mensaje, al estilo de la función *printf* del lenguaje C, y un arreglo de valores constantes que contiene los Strings que van a ser insertados en la plantilla. Los especificadores de formato para *CreateFmt* son los mismos que los disponibles para la función *Format*.

```
Raise Exception.CreateFmt('%s no es un número válido.', [Edit1.text]);
```

* *CreateRes* permite cargar un recurso de Strings para el mensaje de error. Desde Delphi 3 en adelante, ésta es una función sobrecargada, permitiendo referenciar una tabla de Strings o la dirección de un *resourceString*.

La tabla de Strings y los *resourceString* son mecanismos para facilitar la localización de una aplicación Delphi, es decir, la traducción de la aplicación a diferentes idiomas reemplazando los strings utilizados por la aplicación, ya sean *captions* de botones y formularios o mensajes mostrados al usuario. Ambos mecanismos consisten en reunir los strings en un solo recurso, para que pueda ser fácilmente modificado sin requerir la recompilación de toda la aplicación.

La versión de tabla de Strings del constructor *CreateRes* tiene un solo argumento, un entero sin signo que identifica el recurso en la tabla:

```
…
144, "Error"
…

Raise Exception.CreateRes(144);
```

La versión de *resourceString* tiene también un solo argumento, un puntero al *resourceString*:

```
…
resourcestring
        rsMsg = 'Error';
…
Raise Exception.CreateRes(@rsMsg);
```

El operador @ permite obtener la referencia (la dirección de memoria) de una variable, como el operador & del lenguaje C.

* *CreateResFmt* es como *CreateRes* y *CreateFmt* combinados. La tabla de Strings o el *resourceString* serán plantillas que contengan especificadores de formato.

La versión de tabla de Strings tiene dos argumentos, el número de recurso y un arreglo de constantes con los valores a insertar en la plantilla.

```
...
145, “s% no es un número válido.”
...
Raise Exception.CreateResFmt(145, [Edit1.text]);
```

La versión de *resourceString* también tiene dos argumentos, el puntero al *resourceString* y el arreglo de constantes a insertar en la plantilla:

```
resourcestring
...
rsMsg =  's% no es un número válido';
...
Raise Exception.CreateResFmt(@rsMsg, [Edit1.text]);
```

* *CreateHelp* es como *Create*, pero provee una forma de usar ayuda contextual al manejar excepciones. Recibe un segundo argumento, un ID de ayuda contextual (HelpContext ID) con el nombre *AHelpContext*. Este valor se puede usar como parámetro de *MessageDlg* dentro de un manejador, por ejemplo:

```
Try
  Raise Exception.CreateHelp(‘Error. Presione F1 para más información.’, 10);
Except
  On E : Exception do
        MessageDlg(E.Message, mtError, [mbOk, mbHelp], E.HelpContext);
End;
```

* *CreateFmtHelp* es similar a *CreateHelp* pero permite dar formato al mensaje, como hace *CreateFmt*.
* *CreateResHelp* es similar a *CreateHelp* pero toma el string del mensaje de un recurso, como hace *CreateRes*.
* *CreateResFmtHelp* es similar a *CreateResHelp* pero maneja *resourceStrings* con formato, como hace *CreateResFmt*.

Excepciones definidas por el usuario

Definir excepciones propias en Delphi es muy sencillo. Lo único que se debe hacer es definir una subclase de *Exception*, de la siguiente forma:

```
Type
        MiExcepcion = class(Exception);
```

Las excepciones se definen como cualquier otra clase, y como tales heredan los métodos, propiedades y constructores de sus superclases. Una excepción definida por el usuario puede derivar de alguna subclase de *Exception*, no necesariamente de la clase *Exception* misma. Las excepciones definidas por el usuario también pueden definir atributos propios, de forma que los manejadores correspondientes dispongan de la información que el usuario crea conveniente.

Un ejemplo sencillo de cómo definir, lanzar, y manejar una excepción propia es el siguiente:

```
…
// se define la nueva excepción como subclase de Exception.
type
  EEdadMinima = class (Exception);

const
   EDAD_MINIMA = 18;
   EDAD_JUBILACION = 65;
…
procedure TForm1.Button1Click(Sender: TObject);
var
   edad: Integer;
begin
  try
     edad := StrToInt(Edit1.Text);
     if sePuedeJubilar(edad) then
       ShowMessage('El empleado está en edad de jubilarse.')
     Else
        ShowMessage('El empleado aún no está en edad de jubilarse.');
  except
     on E : EConvertError do
       showMessage('El número ingresado no es válido.');
     // se define un manejador para la excepción.
     on E : EEdadMinima do
       showMessage(E.Message);
  end;
end;

function sePuedeJubilar(edad: integer): boolean;
begin
if edad < EDAD_MINIMA then
// se lanza explícitamente la excepción.
          raise EEdadMinima.CreateFmt('%d años no es edad laboral.', [edad]);
sePuedeJubilar := (edad >= EDAD_JUBILACION);
end;
```

Apéndice - La jerarquía de *Exception*

A continuación se muestran los tipos de excepciones más comunes que forman parte de la jerarquía que tiene a la clase *Exception* como raíz. Por convención, todas las clases que representan excepciones comienzan con la letra E.

```
Exception              Base class
EAbort                 Abort without dialog
EAbstractError         Abstract method error
AssertionFailed        Assert call failed
EBitsError             Boolean array error
ECommonCalendarError   Calendar calc error
  EDateTimeError       DateTime calc error
  EMonthCalError       Month calc error
  EConversionError     Raised by Convert
EConvertError          Object convert error
EDatabaseError         Database error
EExternal              Hardware/Windows error
  EAccessViolation     Access violation
  EControlC            User abort occured
  EExternalException   Other Internal error
EIntError              Integer calc error
  EDivByZero           Integer Divide by zero
  EIntOverflow         Integer overflow
  ERangeError          Out of value range
EMathError             Floating point error
  EInvalidArgument     Bad argument value
  EInvalidOp           Inappropriate operation
  EOverflow            Value too large
  EUnderflow           Value too small
  EZeroDivide          Floating Divide by zero
EStackOverflow         Severe Delphi problem
EHeapException         Dynamic memory problem
  EInvalidPointer      Bad memory pointer
  EOutOfMemory         Cannot allocate memory
EInOutError            IO error
EInvalidCast           Object casting error
EInvalidOperation      Bad component op
EMenuError             Menu item error
EOSError               Operating system error
EParserError           Parsing error
EPrinter               Printer error
EPropertyError         Class property error
EPropReadOnly          Invalid property access
EPropWriteOnly         Invalid property access
EThread                Thread error
EVariantError          Variant problem
```

Bibliografía

* La Biblia de Delphi 7, Marco Cantù.
* http://www.delphibasics.co.uk/Article.asp?Name=Exceptions
* http://www.delphibasics.co.uk/RTL.asp?Name=Try
* http://www.delphibasics.co.uk/RTL.asp?Name=Except
* http://www.delphibasics.co.uk/RTL.asp?Name=On
* http://www.delphibasics.co.uk/RTL.asp?Name=Finally
* http://www.delphibasics.co.uk/RTL.asp?Name=Raise
* http://delphi.about.com/od/objectpascalide/a/errorexception.htm
* http://www.ibobjects.com/docs/ti_ErrorHandling.pdf