Data Validation Levels

by Bob Swart

Everyone is sure to face this problem at one time or another: data validation on user input. Especially for data-entry screens with multiple fields of (possibly multiple) records and tables. Data validation can come in at several levels: we could apply it at the input level (edit masks), field level, record/table level and even dialog or application level. This article will show what each level of data validation consists of, how we can apply it and hence when to use (or not use) it.

Input Level

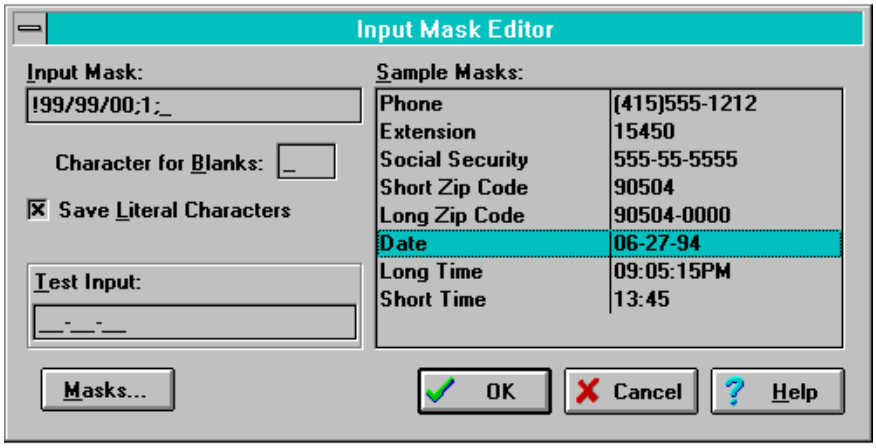
The lowest level of data validation is directly at the user input level and can be done using edit masks or discrete selection lists (a combo-box or listbox where only one item can be selected).

We can implement input level data validation for date type fields, for example, by allowing only numbers and separators. A `TDBEdit` component has an `EditMask` property. If you click on the ellipsis (...) in the Object Inspector, the Input Mask Editor fires up and you can define the mask for this particular editbox for user input level validation. For a date type field (birthday for example), this would be as shown in Figure 1.

Note that in this case we can simply select one of the predefined sample masks. After clicking OK, the resulting `EditMask` field of a `TDBEdit` class has the value `!99/99/00;1;_` which does not require any additional coding at all.

Field Level

Even when we've applied input level data validation, we cannot be sure that the entire field value is valid. For example, the valid date that was input using the `EditMask` in the previous example might not be a valid birthday (it might be a *future* date). So, we must add semantic checks to the individual fields as well: field level data validation.



➤ *Figure 1*

```
procedure TForm1.Table1BIRTH_DATEValidate(Sender: TField);
begin
  if (Sender AS TDateField).Value > Now then
    raise Exception.Create('Birthday is a future date...')
end;
```

➤ *Listing 1*

```
procedure TForm1.Table1BeforePost(DataSet: TDataset);
begin
  if (DataSet.FieldByName('NUMBER_OF_CHILDREN').AsInteger > 0) and
    ((Now - DataSet.FieldByName('BIRTH_DATE').AsDateTime) < (12 * 365)) then
    raise Exception.Create(
      'Birthday inconsistent with number of children...')
end;
```

➤ *Listing 2*

We can easily implement field level data validation using the `OnValidate` event of any `TField` class by selecting the field in the Object Inspector and clicking on the `OnValidate` event in the events page. The code in Listing 1 checks for a valid birthday, for example.

Now, when we enter a value of, say, 07/11/97, an exception is raised and we get a message dialog. The exception will also make sure that the update will not happen to the field (and hence not to the record or to the table). We'd have to cancel the update or modify the value.

Record/Table Level

Sometimes, when we have individual fields checked for their validity, we're still not sure if the record itself is valid. For example, when the birthday field leads to an age of 2 and the number-of-children field also has a valid value 2, you still know something is wrong at record level...

We can implement record level data validation by using the `BeforePost` even of a `TTable` class as shown in Listing 2 (note that these conditions span multiple fields and require more coding than field level validations).

Now, when we enter a value of, say, 07/11/94 and a value of 2 in a number-of-children field then an exception is raised and we get a message dialog.

Rules like these are often also referred to as *Business Rules*,

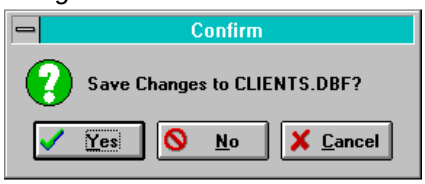
especially when combining more tables together (which is much like referential integrity).

Dialog/Form Level

Sometimes, we may want to close a dialog before we've saved the current (edited) record in the table. This may be on purpose, but it may be a mistake and we actually mean the record to be saved. Or we mean the record to be checked and saved at a multi-table level (for example to check referential integrity). Fortunately, every `TForm` has an `OnCloseQuery` event that we use to check if a table is currently in the edit or insert state. See Listing 3.

Now, if we want to close the dialog or form while the table is still in insert or edit mode we get the dialog shown in Figure 2, where we can decide whether to post the updates to the table. Of course, if we click `Yes` and a post is done the `BeforePost` event of the table can be fired which can result in an exception if the record/table data validation rules indicate a violation.

➤ *Figure 2*



Summary

Input level validation can be added to the `EditMask` property of `TDBEdit` controls, field level validation can be added to the `OnValidate` event of the individual `TField` classes, record/table level validation can be added to the `OnBeforePost` event of the `TTable` classes and finally dialog/form level validation can be added to the `OnCloseQuery` event of the entire dialog or form class. Each level of data entry validation can build upon the previous one, and depending on the situation you may need one or more of them to support the full range of validation.

Bob Swart (aka Dr.Bob, check out http://www.pi.net/~drbob/) is a professional software developer using Delphi and C++ for Bolesian, and a free-lance technical author for The Delphi Magazine. In his spare time, he likes to watch video tapes of Star Trek Voyager and Deep Space Nine with his 2.5 year old son Erik Mark Pascal.

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
var Result: Word;
begin
  if Table1.State in [dsEdit,dsInsert] then begin
    Result := MessageDlg('Save Changes to '+Table1.TableName+'?',
      mtConfirmation, [mbYes, mbNo, mbCancel], 0);
    CanClose := True;
    case Result of
      MrYes    : Table1.Post;
      MrNo     : Table1.Cancel;
      MrCancel : CanClose := False
    end
  end
end;
```

➤ *Listing 3*