TMS FNC ResponsiveList
DEVELOPERS GUIDE

September 2016

Web: http://www.tmssoftware.com
Email: info@tmssoftware.com

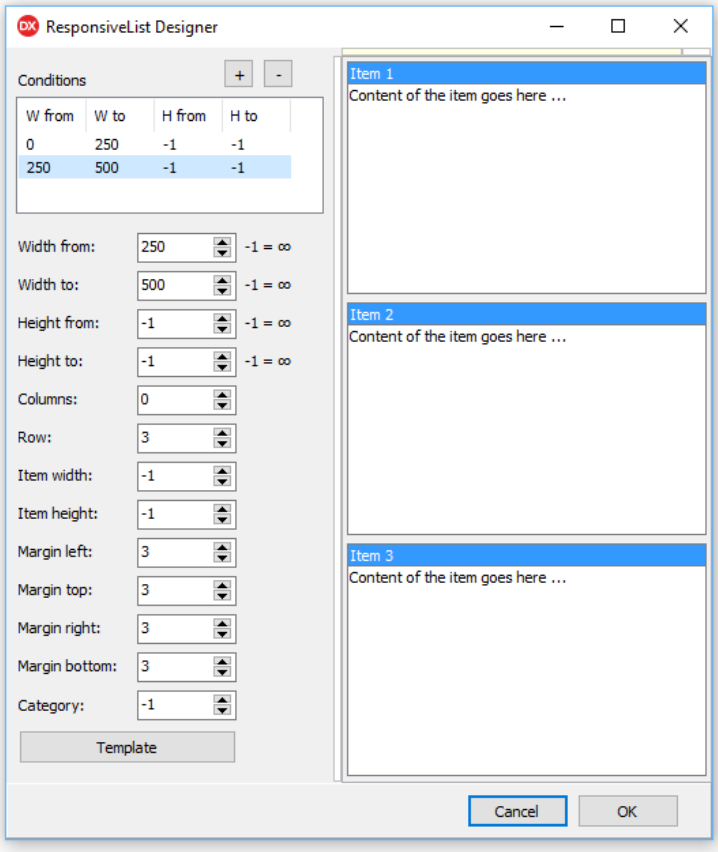
TTMSFNCResponsiveList

Introduction

The component TTMSFNCResponsiveList brings responsive design (https://en.wikipedia.org/wiki/Responsive_web_design) methodology to FMX applications. While responsive design's origin is in accommodating a web page layout dynamically to the size of the browser, similar techniques can also prove useful in native Windows application design. As such application is typically offered in a resizable window, it can greatly improve the user experience when the layout adapts to the size the user chooses for the application.

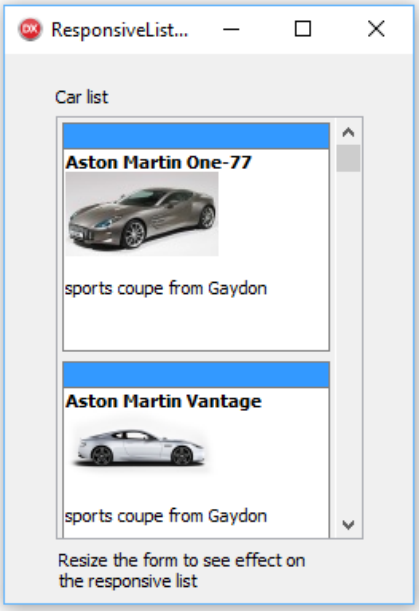
Architecture

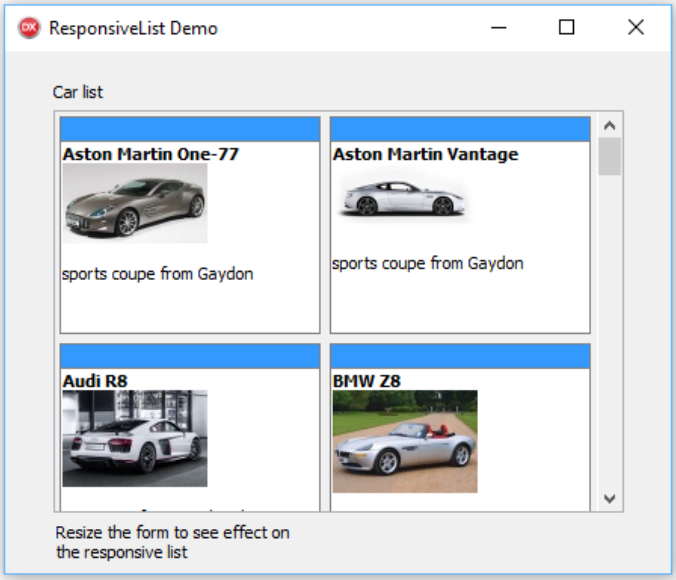
TTMSFNCResponsiveList is designed around a configurable matrix of cells depending on the client area of the control. This configurability is controlled by a collection of conditions. For each condition, the range of client width and client height can be set for which a given number of columns or rows is used or a cell width or height is used.
This conditions collection can be programmatically configured but can of course also be edited at design-time with this design-time editor:

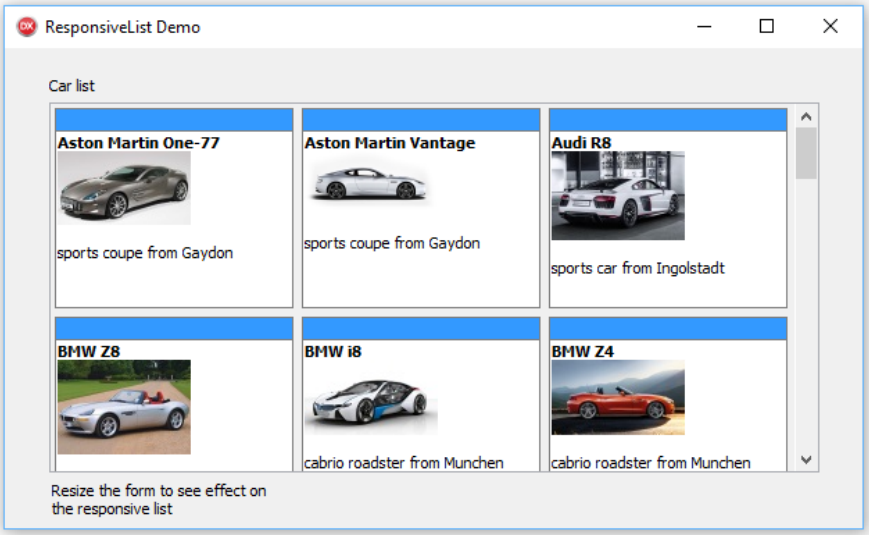


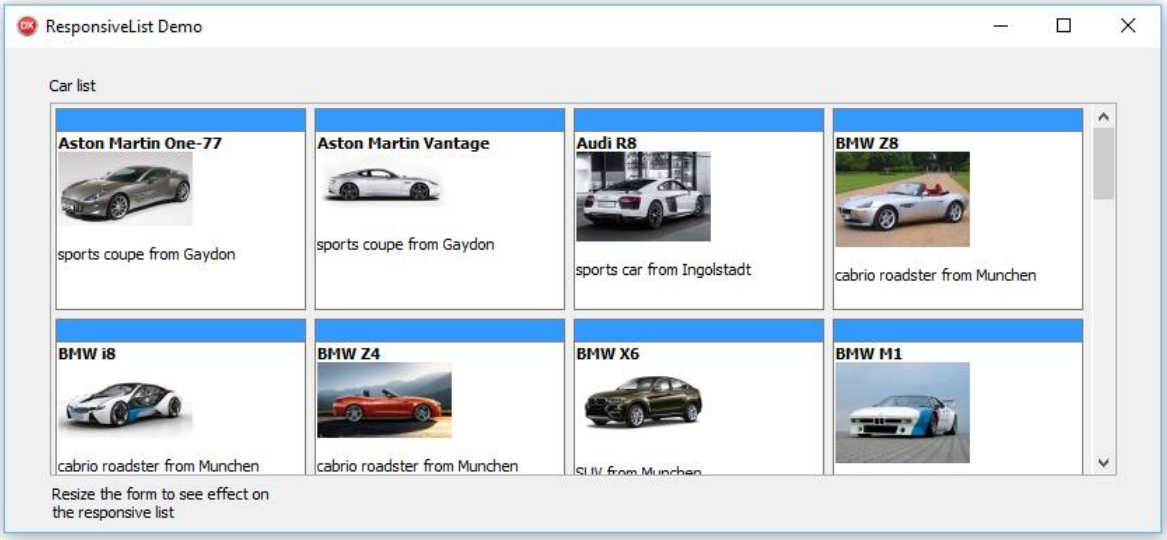
In the top left listview, the conditions are listed. In this example, 4 conditions for 4 different width ranges are configured, i.e. from 0..250pixels, from 250..500pixels, from 500..750pixels and from 750 to any higher width. In this sample, in the smallest width range, the number of columns is set to 1, the next width it is set to 2, then 3 and finally 4 columns when the width is larger than 750pixels. The item height is then in all circumstances configured to 150pixels. As the item width is set to -1, this means the width of items will size proportionally with the width of the control. This width could have been set to a fixed width in pixels as well.

To understand the basics of the architecture better, this leads to following behaviour:









Items

The TTMSFNCResponsiveList has a collection of items that are rendered in the list. The item can be fully custom drawn, but it already supports rendering of HTML formatted text and with this also the rendering of images and hyperlinks. In addition to this, an item can have a header and a footer. The properties of the base item class are:

BorderColor: TTMSFNCGraphicsColor
Color of the border of the item

BorderStyle: TBorderStyle
Sets the border style to bsNone or bsSingle

Color: TTMSFNCGraphicsColor
Sets the background color of the item

Content: string
Sets the text content (can be HTML formatted content) of the item

FooterColor: TTMSFNCGraphicsColor
Sets the color of the footer. When the color is clNone, no footer is drawn

FooterTextColor: TTMSFNCGraphicsColor
Sets the font color of the footer

FooterText: string
Sets the text of the footer

HeaderColor: TTMSFNCGraphicsColor
Sets the color of the header. When the color is clNone, no header is drawn

HeaderTextColor: TTMSFNCGraphicsColor
Sets the font color of the header

HeaderText: string
Sets the text of the header

Height: integer
Defines the height of the item when the height type is different from isAuto

HeightType: TItemSizeType
Can be:
-isAuto: height of the item is automatically determined by the conditions
-isFixed: height of the item is fixed in pixels
-isPerc: height of the item is fixed in percentage
-isFill: height of the item is equal to the control height

SelectedBorderColor: TTMSFNCGraphicsColor
Sets the border color of the item when it is in selected state

SelectedColor: TTMSFNCGraphicsColor
Sets the background color of the item when it is in selected state

SelectedTextColor: TTMSFNCGraphicsColor
Sets the text color of the item when it is in selected state

Shadow: Boolean
When true, the item is drawn with a shadow

Tag: NativeInt
Generic item tag property

TextColor: TTMSFNCGraphicsColor
Color of the item text in normal state

Visible: Boolean
Controls the item visibility state

Width: Integer
Defines the width of the item when the width type is different from isAuto

WidthType: TItemSizeType
Can be:
-isAuto: width of the item is automatically determined by the conditions
-isFixed: width of the item is fixed in pixels
-isPerc: width of the item is fixed in percentage
-isFill: width of the item is equal to the control height

Responsive templated items

Not only can the size of the item be responsively determined but also the formatting of the content. This is done via responsive templates. Part of the conditions is a template. A template is a HTML formatted string with value placeholders. The control will then automatically resolve the value placeholders with the actual values held by an item.

This way, in a small version of an item, it could show less text, i.e. in the condition for the width of the control being <= 250pixels, the condition template could have been set to:

<b>(%TITLE)<b>
(%PRICE)

while for a control width > 250pixels, the template could have been set to:

<b>(%TITLE)<b>
(%PRICE)
(%DESCRIPTION)

When the item contains values for TITLE, PRICE and DESCRIPTION, the control will automatically resolve the correct template according to the selected condition depending on the width of the control.

The values of the item are a NAME/VALUE pair collection where the VALUE is a variant type. For this particular case, the item's NAME/VALUE pairs could have been set via

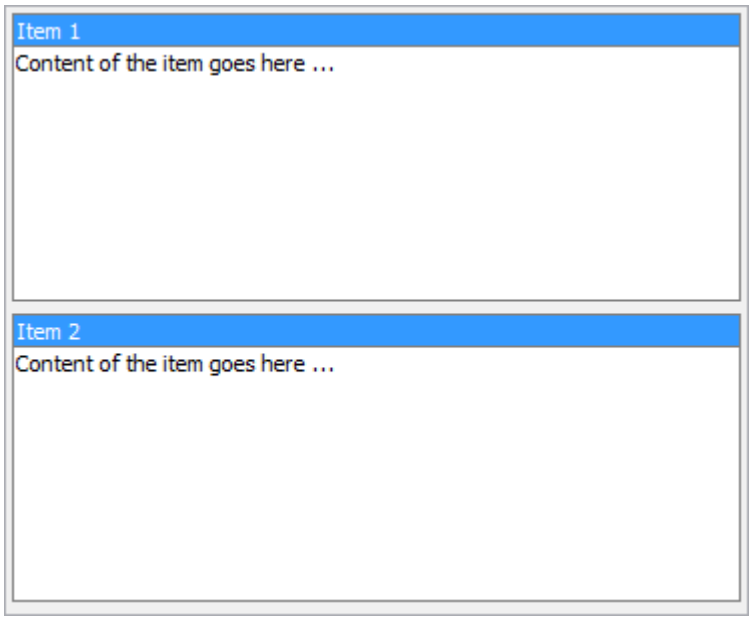
```
var
  it: TResponsiveListItem;

it := TMSFNCResponsiveList.Items.Add;

it.Values['TITLE'] := 'Lion King';
it.Values['PRICE'] := 123.456;
it.Values['DESCRIPTION'] := 'The Lion King is a 1994 American animated epic musical film produced by Walt Disney Feature Animation and released by Walt Disney Pictures.';

it := TMSFNCResponsiveList.Items.Add;
```

it.Values['TITLE'] := 'Nemo';
it.Values['PRICE'] := 210.987;
it.Values['DESCRIPTION'] := 'Finding Nemo is a 2003 American computer-animated comedy-drama TMSFNCenture film produced by Pixar Animation Studios';

Responsive lists in TTMSFNCResponsiveList

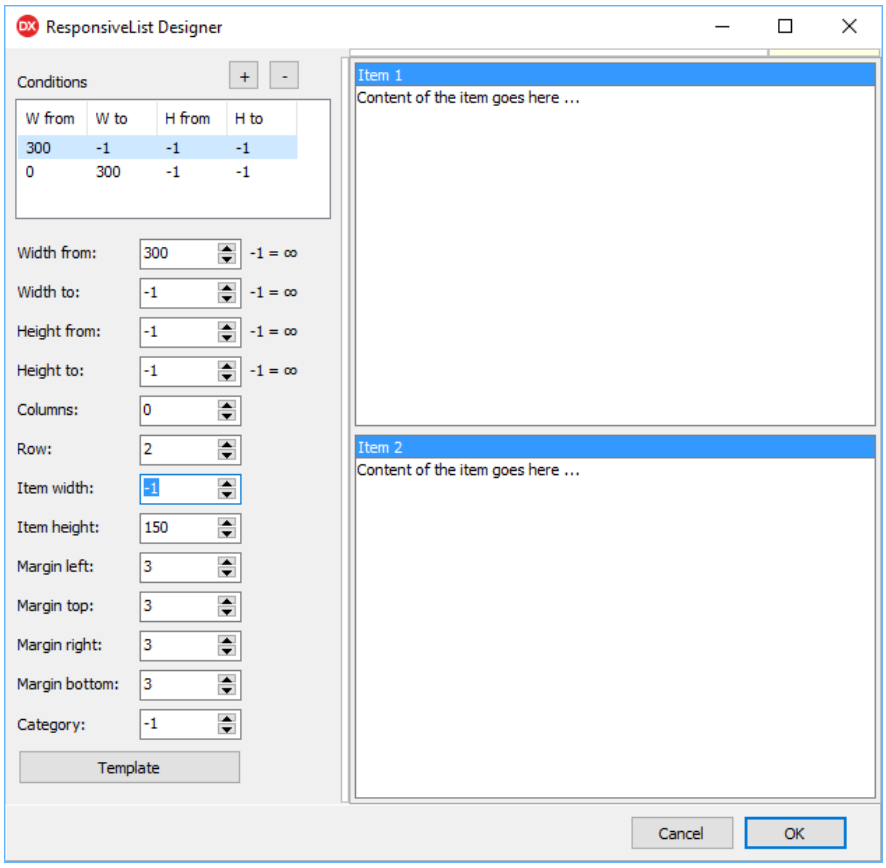
The number of possibilities of using TTMSFNCResponsiveList becomes sheer unlimited when hosting a TTMSFNCResponsiveList within a TTMSFNCResponsiveList.

To illustrate this concept, let's start with a TTMSFNCResponsiveList with 2 items:

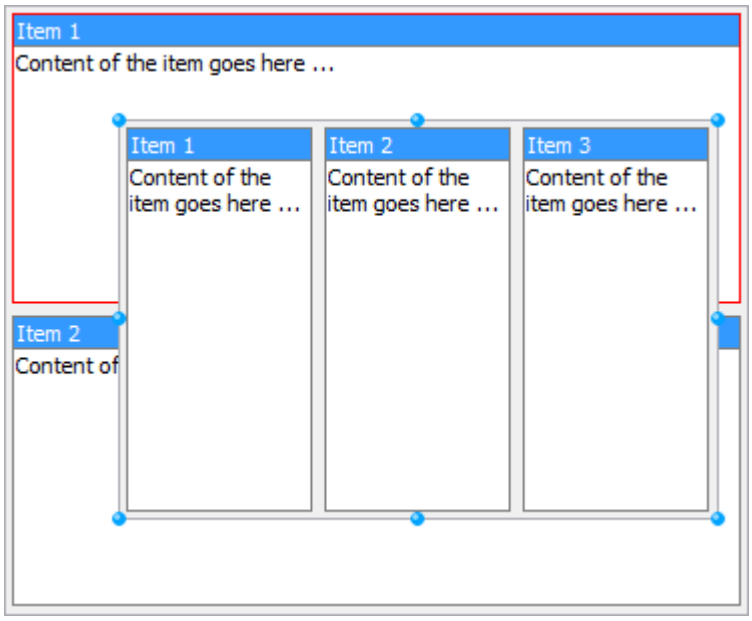


And configure 2 conditions:

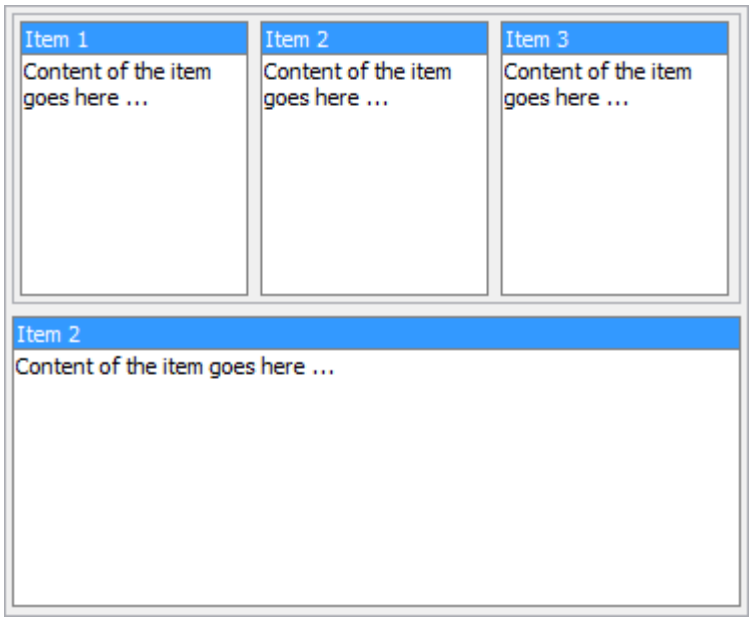
The first condition will render the list as a list of items in 2 rows when the width of the control is 300pixels or wider and a list of items in 2 columns when the width is smaller than 300.



Next drop on this TTMSFNCResponsiveList a second TTMSFNCResponsiveList:

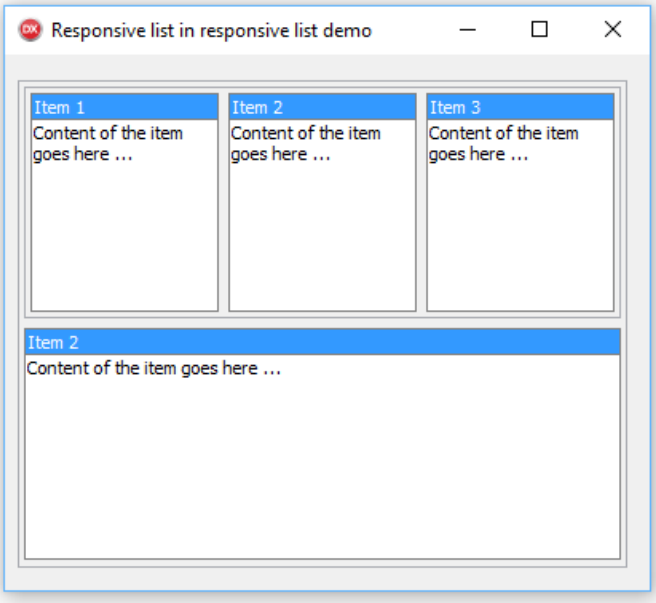


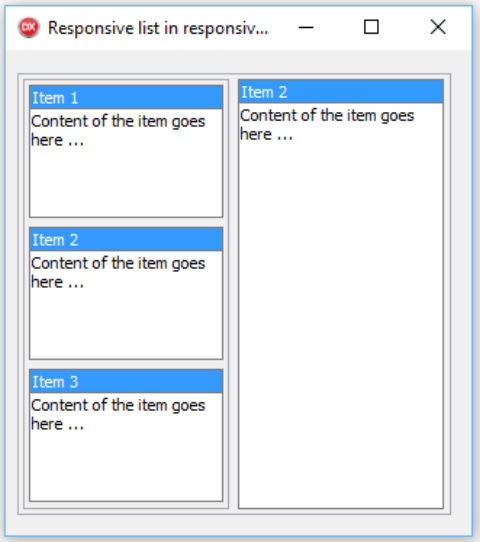
At design time, the item in the parent TTMSFNCResponiveList where the 2nd TTMSFNCResponsiveList is dropped on will display with a red border. At design time the 2nd TTMSFNCResponsiveList can be dragged over the 2nd item. Once the parent TTMSFNCResponsiveList item to connect the 2nd TTMSFNCResponsiveList is chosen, you can set the child TTMSFNCResponsiveList.Align to alClient and it will align to the chosen parent list item. When the application is run, this results in:



Now, on this child TTMSFNCResponsiveList, we can again add conditions for responsive behaviour. In this case, we add the condition that for a width of 300pixels or higher, it will render its items in columns and for a width of less than 300pixels, it will render the items in rows.

With these conditions in place, resizing the parent TTMSFNCResponsiveList will result in the child TTMSFNCResponsiveList to adapt to its parent item and as such, also responsively adapt to render its items in rows:





Creating custom TTMSFNCResponsiveList controls

It is straightforward to creating custom TTMSFNCResponsiveList controls that use custom items.

To do so, create a custom TResponsiveListItem class that descends from TResponsiveListItem:

```
type
  TResponsiveListItemEx = class(TResponsiveListItem)
  private
    FPicture: TPicture;
    FCustomProp: string;
    procedure SetPicture(const Value: TPicture);
  protected
    procedure DrawItem(ACanvas: TCanvas; ATemplate: string; ARect: TRect); override;
    procedure PictureChanged(Sender: TObject);
  public
    constructor Create(Collection: TCollection); override;
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); override
  published
    property Picture: TPicture read FPicture write SetPicture;
    property CustomProp: string read FCustomProp write FCustomProp;
  end;
```

and then with overriding the GetItemClass protected method in a descending control of TTMSFNCResponsiveList, start using this custom item class:

```
  TTMSFNCResponsiveListEx = class(TTMSFNCResponsiveList)
  private
  protected
    function GetItemClass: TCollectionItemClass; override;
  published
  end;
```

with:

```
function TTMSFNCResponsiveListEx.GetItemClass: TCollectionItemClass;
begin
  Result := TResponsiveListItemEx;
end;
```

In this example, we have added a TPicture property to the custom TResponsiveListItemEx class as well as a CustomProp string property. By then overriding the item's DrawItem() protected method, the item becomes responsible to draw itself within the responsive list:

```
procedure TResponsiveListItemEx.DrawItem(ACanvas: TCanvas; ATemplate: string;
  ARect: TRect);
begin
  inherited DrawItem(ACanvas, ATemplate, ARect);

  if Assigned(FPicture.Graphic) and not FPicture.Graphic.Empty then
    ACanvas.Draw(ARect.Left + 10, ARect.Top + 10, FPicture.Graphic);

  ACanvas.TextOut(ARect.Left, ARect.Top, CustomProp);
end;
```

As the default item class DrawItem() method is still called here, this means the custom item will draw a picture and text on top of the existing item.