Formatted Text Processing  
In Docx DOM

By: Jarosław Kuchta

When editing text in OpenXml Wordprocessing data object model we have a problem with searching the structure of the document. The XML structure of the main part of the document is described by the OpenXml Document Format standards – ECMA-376 and ISO/IEC 29500. There are software libraries for .NET such as Open XML SDK for Office which define strongly typed classes that represent structure elements such as Paragraph and Table and help to access their properties and members. However, the functionality of these libraries is limited to reading and retrieving XML elements and attributes and does not include text or graphics manipulation.

This document describes how to manipulate text stored in the OpenXml structure with respect to text formatting. We will see methods for searching and replacing text while preserving or changing formatting. We will see also ways to process special characters such as tabs and hyphens. Finally, we will see how to deal with graphic elements in text. All these possibilities have been implemented by the author in his own library Qhta.OpenXml.Tools.

# Formatted text find and replace

OpenXml SCK libraries do not handle text searching and replacing. I defined the extension methods of GetText() and SetText() for such OpenXml elements like Paragraph, Run, and Text in the Qhta.OpenXml.Tools library. These simple methods are to get and set plain text. We can use them to find and replace (or process in other way) plain, not-formatted text. However, when we want to keep or change the text formatting, we meet a problem of the text-in-Xml structure.

## Problem of structured text processing

In OpenXml Wordprocessing Data Object Model, the text is stored in the hierarchical XML structure. To consider this here, we will abstract of the high document structure and focus on the lowest one.

The highest level of the interested element containment are Paragraph classes, which contain Run elements (and other, non-textual elements). also, Paragraph class can contain also a ParagraphProperties element, which represents the formatting of the text at the paragraph level – vertical spacing before and after the paragraph, interline spacing, left and right indenting or horizontal justification of the text. These formatting can be stored in the ParagraphProperties directly or it can be defined by the paragraph style. To distinguish between the ParagraphProperties and other elements of the Paragraph (which are Runs and other elements, like Hyperlinks, BookmarkStart, and BookmarkEnd element), we call the other elements – members of the paragraph.

The Run class is the middle-level class. It can contain a RunProperties element and other elements (run members). One of the member elements is a Text class, which directly contains the text values. The RunProperties class stores the run-level formatting if the text (like bold or italic attributes, font names or sizes).

Example: Let’s consider the following text:

This text is **bold,** and this is *italicized*.

To be more precise, we mark the bold text with <b></b> tags and the italicized text with <i></i> tags:

This text is <b>**bold,**</b> and this is <i>*italicized*</i>.

This text is stored in OpenXml in the following structure:

<w:p w14:paraId="763D35DE" w14:textId="4BD8BFE4" w:rsidR="00B65346" w:rsidRDefault="006F2D84" w:rsidP="00307B9B">

<w:pPr>

<w:pStyle w:val="Normal"/>

</w:pPr>

<w:r>

<w:t xml:space="preserve">This text is </w:t>

</w:r>

<w:r w:rsidRPr="006F2D84">

<w:rPr>

<w:b/>

<w:bCs/>

</w:rPr>

<w:t>bold,</w:t>

</w:r>

<w:r w:rsidRPr="006F2D84">

<w:t xml:space="preserve"> and this is </w:t>

</w:r>

<w:r w:rsidRPr="006F2D84">

<w:rPr>

<w:i/>

<w:iCs/>

</w:rPr>

<w:t>italicized</w:t>

</w:r>

<w:r w:rsidRPr="006F2D84">

<w:t>.</w:t>

</w:r>

</w:p>

The meanings of the XML elements are the following:

<w:p> - paragraph

<w:pPr> - paragraph properties

<w:pStyle> - paragraph style

<w:r> - run

<w:t> - text

<w:rPr> - run properties

<w:b> - bold

<w:bCs> - bold for complex script

<w:i> - italics

<w:iCs> - italics for complex script

The meanings of the XML attributes are the following:

w14:paraId – paragraph identifier

w14:textId – text identifier

w:rsidR – revision identifier for paragraph

w:rsidRDefault – default revision identifier for runs

w:rsidP – revision identifier for paragraph properties

w:rsidRPr – revision identifier for run properties

As we can see, there is plenty of excessive information: identifiers of paragraphs, runs, and revisions, properties of paragraphs and runs. This information would be lost if we simply get the text, edit it, and set back.

Let’s try to change the text “bold" to “boldfaced" and “italicized" to “italic". To do it, first we get plain text from the structure. It is:

This text is bold, and this is italicized.

Next, we will change the extracted text to:

This text is boldfaced, and this is italic.

Finally, we will set the text back to the paragraph.

To do so, we call GetText() and SetText() extension methods of the Paragraph class and Replace() method of the String class:

Paragraph.SetText(Paragraph.GetText().Replace(“bold", “boldfaced").Replace(“italicized", “italic"))

In the naive solution, in the first step of the Paragraph.SetText() method, we remove all the paragraph member elements (except the paragraph properties).

<w:p w14:paraId="763D35DE" w14:textId="4BD8BFE4" w:rsidR="00B65346" w:rsidRDefault="006F2D84" w:rsidP="00307B9B">

<w:pPr>

<w:pStyle w:val="Normal"/>

</w:pPr>

</w:p>

In the next step we create a new Run with the changed text.

<w:p w14:paraId="763D35DE" w14:textId="4BD8BFE4" w:rsidR="00B65346" w:rsidRDefault="006F2D84" w:rsidP="00307B9B">

<w:pPr>

<w:pStyle w:val="Normal"/>

</w:pPr>

<w:r>

<w:t xml:space="preserve">This text is boldfaced, and this is italic.</w:t>

</w:r>

</w:p>

As we can see, we have lost all the run-level formatting, and the changed text simply looks as:

This text is boldfaced, and this is italic.

To preserve run-level formatting, we must use another solution.

## Preserve run-level formatting

Let’s take all run elements and store them with their text in a list. We define the type of the items as a RunText record:

public record RunText

{

public readonly DXW.Run Run;

public string Text;

public RunText(DXW.Run run, string text)

{

Run = run;

Text = text;

}

}

We will name the type of list a FormattedText class as each string is associated with a Run element that can hold formatting in the RunProperties. Creating a FormattedText on a Paragraph is done by reading all Run members.

public class FormattedText: List<RunText>

{

private readonly TextOptions GetTextOptions = TextOptions.PlainText;

public FormattedText(DXW.Paragraph paragraph)

{

foreach (var member in paragraph.Elements<DXW.Run>())

{

var text = member.GetText(GetTextOptions);

this.Add(new RunText(member, text));

}

}

Before testing the FormattedText, we enter my:runId attributes to each run. These attributes (marked in green) are not defined in OpenXml and are shown below only for presentation purposes.

<w:p w14:paraId="763D35DE" w14:textId="4BD8BFE4" w:rsidR="00B65346" w:rsidRDefault="006F2D84" w:rsidP="00307B9B">

<w:pPr>

<w:pStyle w:val="Normal"/>

</w:pPr>

<w:r w:rsidRPr="006F2D84" my:runId="00000001">

<w:t xml:space="preserve">This text is </w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000002">

<w:rPr>

<w:b/>

<w:bCs/>

</w:rPr>

<w:t>bold,</w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000003">

<w:t xml:space="preserve"> and this is </w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000004">

<w:rPr>

<w:i/>

<w:iCs/>

</w:rPr>

<w:t>italicized</w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000005">

<w:t>.</w:t>

</w:r>

</w:p>

In the example, we have five runs with five strings, so we get the following list:

|  |  |
| --- | --- |
| Run | Text |
| <w:r my:runId="00000001"> | “This text is “ |
| <w:r my:runId="00000002"> | “bold," |
| <w:r my:runId="00000003"> | “ and this is “ |
| <w:r my:runId="00000004"> | “italicized" |
| <w:r my:runId="00000005"> | “." |

Let’s define a GetText() method of the FormattedText class, which simply concatenates the strings to produce a plain text for the paragraph:

This text is bold, and this is italicized.

We can use GetText() method to search for a plain text, but to change the text of the paragraph, we need a SetText() method operating on an indexed item of FormattedText structure.

public void SetText(int index, string text)

{

this[index].Text = text;

this[index].Run.SetText(text);

}

To search and replace the text and preserve the formatting, we create the FormattedText instance on the Paragraph and invoke Replace() method of the FormattedText class:

var formattedText = new FormattedText(Paragraph);

formattedText.Replace(“bold", “boldfaced");

formattedText.Replace(“italicized", “italic");

At first, we implement the Replace() method of the FormattedText as follows:

public bool Replace(string searchText, string replacementText)

{

var s = GetText();

var k = s.IndexOf(searchText);

if (k >= 0)

return ReplaceAt(k, searchText.Length, replacementText);

return false;

}

The ReplaceAt() method first iterates over list items summarizing string lengths until we achieve a given position.

private void ReplaceAt(position, length, replacementText)

{

var sumLength = 0;

var selectedItem = -1;

for (int i=0; i < Count; i++)

{

var itemText = this[i].Text;

if (sumLength + itemText.Length > position)

{

selectedItem = i;

break;

}

sumLength += itemText.Length;

}

Then, in the selected string, it deletes a substring at index of (position – sumLength) with a given length and inserts the replacementText at this index. The modified text is set to the Run element.

if (selectedItem ≥ 0)

{

var itemPosition = position - sumLength;

SetText(selectedItem, this[selectedItem].Text.Remove(itemPosition, length).Insert(itemPosition, replacementText));

return true;

}

return false;

}

In our example we have changed the FormattedText in the following way:

|  |  |
| --- | --- |
| Run | Text |
| <w:r my:runId="00000001"> | “This text is “ |
| <w:r my:runId="00000002"> | “boldfaced," |
| <w:r my:runId="00000003"> | “ and this is “ |
| <w:r my:runId="00000004"> | “italic" |
| <w:r my:runId="00000005"> | “." |

Because the SetText() method of the FormattedText invokes the SetText() method of the appropriate Run, we have changed the Paragraph as following:

<w:p w14:paraId="763D35DE" w14:textId="4BD8BFE4" w:rsidR="00B65346" w:rsidRDefault="006F2D84" w:rsidP="00307B9B">

<w:pPr>

<w:pStyle w:val="Normal"/>

</w:pPr>

<w:r w:rsidRPr="006F2D84" my:runId="00000001">

<w:t xml:space="preserve">This text is </w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000002">

<w:rPr>

<w:b/>

<w:bCs/>

</w:rPr>

<w:t>boldfaced,</w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000003">

<w:t xml:space="preserve"> and this is </w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000004">

<w:rPr>

<w:i/>

<w:iCs/>

</w:rPr>

<w:t>italic</w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000005">

<w:t>.</w:t>

</w:r>

</w:p>

and the formatting is preserved. We’ve got the resulted text as:

This text is <b>**boldfaced**,</b> and this is <i>*italic*</i>.

## Multiple runs problem

The above solution works well when we change the text which fits completely to a single Run. But what will happen when we change the text that is stored in more than one subsequent Run elements, as below?

textProcessor.Replace(", and this is", ", and this text is");

Notice that the comma character is contained in the “bold” Run, and the rest of the search text is contained in the next “no-bold” Run.

We need to recode the change algorithm. Now we will delete only as much text in the selected FormattedText item as it fits in this item. The rest of the text will be deleted from the next item (or items). We will also introduce some auxiliary variables to debug the process of change.

We organize an internal while loop which starts at the selected FormattedText item and continues for the next items to delete the rest of text which does not fit to this item. The first part of the internal loop (in green) evaluates the length of the deletion. The last part of the loop (in red) is executed only when there is more text to delete, and it prepares the next iteration of the loop.

if (selectedItem >= 0)

{

while (selectedItem < this.Count && (length > 0 || replacementText.Length > 0))

{

var itemText = this[selectedItem].Text;

var itemOldLength = itemText.Length;

var itemPosition = position - sumLength;

var itemRestLength = itemText.Length - itemPosition;

var delLength = length;

if (itemRestLength < length)

{

delLength = itemRestLength;

length -= delLength;

}

else

length = 0;

if (delLength>0)

itemText = itemText.Remove(itemPosition, delLength);

if (replacementText.Length > 0)

itemText = itemText.Insert(itemPosition, replacementText);

this[selectedItem].Text = itemText;

this[selectedItem].Run.SetText(itemText);

if (length == 0)

break;

replacementText = String.Empty;

position += delLength;

sumLength += itemOldLength;

selectedItem++;

}

return true;

}

In our example, the found text was properly deleted from selected and subsequent Run elements and the replacement text was inserted to the selected Run:

<w:p w14:paraId="763D35DE" w14:textId="4BD8BFE4" w:rsidR="00B65346" w:rsidRDefault="006F2D84" w:rsidP="00307B9B">

<w:pPr>

<w:pStyle w:val="Normal"/>

</w:pPr>

<w:r w:rsidRPr="006F2D84" my:runId="00000001">

<w:t xml:space="preserve">This text is </w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000002">

<w:rPr>

<w:b/>

<w:bCs/>

</w:rPr>

<w:t xml:space="default">bold and this text is</w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000003">

<w:t xml:space="preserve"> </w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000004">

<w:rPr>

<w:i/>

<w:iCs/>

</w:rPr>

<w:t>italicized</w:t>

</w:r >

<w:r w:rsidRPr="006F2D84" my:runId="00000005">

<w:t>.</w:t>

</w:r>

</w:p>

The formatted result looks like this:

This text is <b>**bold, and this text is**</b> <i>*italicized*</i>.

The inserted text has got the formatting of the first found Run.

## Problem of replacement format

Now, how could we change the text and set it to some specific format?

First, we need to pass a specific format to the Replace() method. We cannot use a RunProperies element to represent the format for two reasons. First is that RunProperies element is not the only one description of the text formatting. The current formatting is applied using a *style hierarchy* in the following order:

1. DefaultRunProperties of the DocDefaults element,
2. ConditionalRunProperties of a table style,
3. RunProperties of a numbering style,
4. RunProperties of a paragraph style,
5. RunProperties of a character style,
6. RunProperties of the Run element

The second reason is that we will use this formatting structure not only at run-level, but also at paragraph level.

So, we should declare a TextFormat record with text formatting attributes like Bold and Italic. The IsSame() method compares the text format with another text format. If some format attributes are not set, they are not compared.

public record TextFormat

{

public bool? Bold { get; set; }

public bool? Italic { get; set; }

public bool IsSame(TextFormat other)

{

if (Bold.HasValue && other.Bold.HasValue && Bold != other.Bold)

return false;

if (Italic.HasValue && other.Italic.HasValue && Italic != other.Italic)

return false;

return true;

}

}

We should also declare GetFormat() extension method of the Run element. It should have an implementation like:

public static TextFormat GetFormat(this DXW.Run run)

{

return new TextFormat()

{

Bold = run.IsBold(),

Italic = run.IsItalic()

};

}

We will need also a SetFormat() extension method with an implementation like this:

public static void SetFormat(this DXW.Run run, TextFormat format)

{

if (format.Bold.HasValue)

run.SetBold(format.Bold);

if (format.Italic.HasValue)

run.SetItalic(format.Italic);

}

The methods: IsBold(), IsItalic(), SetBold(), SetItalic() are all the extension methods of the Run class. They support the fact that the RunProperties element has separate Bold and Italic attributes for text in European languages, and separate ones for bidirectional script.

We declare a replacementFormat parameter of the TextFormat type in the Replace and ReplaceAt methods.

Replace(string searchText, string replacementText, TextFormat? replacementFormat = null);

ReplaceAt(int position, int length, string replacementText, TextFormat? replacementFormat = null);

For now, we must change the part of the code which inserts the replacementText.

If the replacementFormat is not null and is not same as the format of the selected item Run, the replacementText is not inserted to the selected Run, but the new Run is created and inserted.

First, we will save the text after deletion.

if (delLength > 0)

{

itemText = itemText.Remove(itemPosition, delLength);

SetText(selectedItem, itemText);

}

There are three cases of new Run insertion:

* Before the selected Run – this is the case when item position of insertion is 0 (green code),
* After the selected Run –when item position is at the end of the item text (blue code),
* Inside the selected Run – then the selected Run must be split into two parts, and the new Run must be inserted between them (red code).

The violet code handles the case when there is no need to create a new Run.

var nextItem = selectedItem + 1;

if (replacementText.Length > 0)

{

if (replacementFormat != null && !replacementFormat.IsSame(this[selectedItem].Run.GetFormat()))

{

if (itemPosition == 0)

{

InsertBefore(selectedItem, replacementText, replacementFormat);

}

else if (itemPosition == itemText.Length)

{

InsertAfter(selectedItem, replacementText, replacementFormat);

nextItem++;

}

else

{

InsertWithSplit(selectedItem, itemPosition, replacementText, replacementFormat);

}

}

else

{

itemText = itemText.Insert(itemPosition, replacementText);

this.SetText(selectedItem, itemText);

}

replacementText = String.Empty;

}

if (length == 0)

break;

position += delLength;

sumLength += itemOldLength;

selectedItem = nextItem;

}

The InsertBefore() method is implemented as:

public void InsertBefore(int index, string text, TextFormat format)

{

var newRun = new DXW.Run();

newRun.AppendText(text);

newRun.SetFormat(format);

this.Insert(index, new RunText(newRun, text));

this[index].Run.InsertBeforeSelf(newRun);

}

Analogously, the InsertAfter() method is implemented as:

public void InsertAfter(int index, string text, TextFormat format)

{

var newRun = new DXW.Run();

newRun.AppendText(text);

newRun.SetFormat(format);

this.Insert(index + 1, new RunText(newRun, text));

this[index].Run.InsertAfterSelf(newRun);

}

Implementation of the InsertWithSplit() method is more complex. We use the extension method SplitAt() of the Run element. This method divides the Run element in two parts at the given character position and returns the second part to the caller.

public void InsertWithSplit(int index, int itemPosition, string text, TextFormat format)

{

var tailRun = this[index].Run.SplitAt(itemPosition, GetTextOptions);

var newRun = new DXW.Run();

newRun.AppendText(text);

newRun.SetFormat(format);

this.Insert(index + 1, new RunText(newRun, text));

this[index].Run.InsertAfterSelf(newRun);

if (tailRun != null)

{

this[index].Text = this[index].Text.Substring(0, itemPosition);

this.Insert(index + 2, new RunText(tailRun, tailRun.GetText(GetTextOptions)));

newRun.InsertAfterSelf(tailRun);

}

}

Now we can invoke:

Replace(", and this is", ", and this text is", new TextFormat{Bold = false});

In the result, we can see a newly added Run (marked in red):

<w:p w14:paraId="763D35DE" w14:textId="4BD8BFE4" w:rsidR="00B65346" w:rsidRDefault="006F2D84" w:rsidP="00307B9B">

<w:pPr>

<w:pStyle w:val="Normal"/>

</w:pPr>

<w:r w:rsidRPr="006F2D84" my:runId="00000001">

<w:t xml:space="preserve">This text is </w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000002">

<w:rPr>

<w:b/>

<w:bCs/>

</w:rPr>

<w:t xml:space="default">bold</w:t>

</w:r>

<w:r>

<w:rPr>

<w:b w:val="false"/>

<w:bCs w:val="false"/>

</w:rPr>

<w:t xml:space="preserve"> and this text is</w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000003">

<w:t xml:space="preserve"> </w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000004">

<w:rPr>

<w:i/>

<w:iCs/>

</w:rPr>

<w:t>italicized</w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000005">

<w:t>.</w:t>

</w:r>

</w:p>

And the resulted text is as follows:

This text is <b>**bold**</b>, and this text is <i>*italicized*</i>.

## Finding text with a format

Now, we must change the example to show search text with a search format. We change the first “is” word to boldface and the next “is” to italic.

This text <b>**is bold,**</b> and <i>*this is italicized*</i>.

Let’s try to change the second “is”, which is italicized, and omit the first “is” (a bold one).

To search for a text with formatting, we need to add a searchFormat parameter to the Replace() method.

Replace(string searchText, TextFormat? searchFormat,   
string replacementText, TextFormat? replacementFormat)

Following this, we define a Find() method to get the position of text with format in the FormattedText starting at the given startPosition. This method returns the character position of the text, if found, or -1, if the text was not found. Comparing text (and format) must not be limited to the single Run, but it must be collected from the subsequent Runs (if the format is the same). It is achieved by collecting text from the subsequent Runs with the same format as searchFormat in a searchInText variable.

public int Find(int startPosition, string searchText, TextFormat? searchFormat)

{

var searchTextLength = searchText.Length;

var sumLength = 0;

for (int i = 0; i < this.Count; i++)

{

var itemText = this[i].Text;

if (sumLength + itemText.Length > startPosition)

{

if (searchFormat == null || searchFormat.IsSame(this[i].Run.GetFormat()))

{

var searchInText = itemText;

int j = i + 1;

while (searchInText.Length > searchTextLength && j < this.Count &&

(searchFormat == null || searchFormat.IsSame(this[j].Run.GetFormat())))

{

searchInText += this[j].Text;

j++;

}

itemText = this[i].Text;

if (searchInText.Length >= searchTextLength)

{

var k = searchInText.IndexOf(searchText);

if (k >= 0)

{

return sumLength + k;

}

}

}

}

sumLength += itemText.Length;

}

return -1;

}

The Find() method is used in the Replace() method as follows:

public bool Replace(string searchText, TextFormat? searchFormat,   
string replacementText, TextFormat? replacementFormat)

{

int k;

if (searchFormat != null)

k = Find(0, searchText, searchFormat);

else

{

var s = GetText();

k = s.IndexOf(searchText);

}

if (k >= 0)

{

ReplaceAt(k, searchText.Length, replacementText, replacementFormat);

return true;

}

return false;

}

We invoke the Replace() method requesting replace the “ is ” text with the format of italic by the text “ is ” with no italic. We need spaces around “is” to omit “is” in the word “this”.

Replace(" is ", new TextFormat { Italic = true }, " is ", new TextFormat { Italic = false });

The result is as below. We can see two Runs with the same runId. The second one is created by the SplitAt() of Run extension method. The newly created Run is marked in red.

<w:p w:rsidR="00B65346" w:rsidP="00307B9B" w:rsidRDefault="006F2D84" w14:paraId="763D35DE" w14:textId="4BD8BFE4">

<w:pPr>

<w:pStyle w:val="Normal"/>

</w:pPr>

<w:r w:rsidRPr="006F2D84" my:runId="00000001">

<w:rPr/>

<w:t xml:space="preserve">This text </w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000002">

<w:rPr>

<w:b/>

<w:bCs/>

</w:rPr>

<w:t>is bold,</w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000003">

<w:rPr/>

<w:t xml:space="preserve"> and </w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000004">

<w:rPr>

<w:i/>

<w:iCs/>

</w:rPr>

<w:t xml:space="default">this</w:t>

</w:r>

<w:r>

<w:rPr>

<w:b w:val="false"/>

<w:bCs w:val="false"/>

</w:rPr>

<w:t xml:space="preserve"> is </w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000004">

<w:rPr>

<w:i/>

<w:iCs/>

</w:rPr>

<w:t>italicized</w:t>

</w:r>

<w:r w:rsidRPr="006F2D84" my:runId="00000005">

<w:rPr/>

<w:t>.</w:t>

</w:r>

</w:p>

The formatted result is:

This text <b>**is** **bold,**</b> and <i>*this*</i>is <i>*italicized*</i>.

## Finding the whole words

We used a special trick to search the word “is” by surrounding the search text with spaces. We can get rid of this trick if we introduce FindAndReplaceOptions to the Replace() method. Define the options as:

public record FindAndReplaceOptions

{

public bool FindWholeWordsOnly;

}

The Find() method must use this option in code. The first part of the code (green) searches text when searchFormat is null. The searchInText variable represents the whole text of the Run. If the option FindWholeWordsOnly is set, then a NUL character (code ‘\0’) is added to both sides of searchInText. It is necessary in case we need to check no-word characters before and after the found text (there MUST NO be no NUL character at the beginning or end of the search text). When the searchText is found in the searchInText variable, the method returns the position of the found text minus one.

The rest of the Find() method code is executed when the searchFormat is not null. The searchInText string is collected from the subsequent Runs with the same format as given in the searchFormat parameter. If the option FindWholeWordsOnly is set, then the last character of the previous Run is added to the start of the searchInText (or NUL character if there is no such Run). Similarly, the first character of the next Run (or NUL character) is added to the end of the searchInText. When the searchText is found is the searchInText, the characters before and after the found text are checked.

public int Find(int startPosition, string searchText, TextFormat? searchFormat, FindAndReplaceOptions? options = null)

{

var findWholeWordsOnly = options?.FindWholeWordsOnly ?? false;

if (searchFormat == null)

{

var searchInText = GetText();

if (findWholeWordsOnly)

{

searchInText = '\0' + searchInText + '\0';

var k = searchInText.IndexOf(searchText, startPosition);

while (k > 0)

{

if (char.IsLetterOrDigit(searchInText[k - 1]) || k + searchText.Length < searchInText.Length && char.IsLetterOrDigit(searchInText[k + searchText.Length]))

k = searchInText.IndexOf(searchText, k + 1);

else

return k - 1;

}

return -1;

}

else

return searchInText.IndexOf(searchText, startPosition);

}

var searchTextLength = searchText.Length;

var sumLength = 0;

for (int i = 0; i < this.Count; i++)

{

var itemText = this[i].Text;

if (sumLength + itemText.Length > startPosition)

{

if (searchFormat.IsSame(this[i].Run.GetFormat()))

{

var searchInText = itemText;

if (findWholeWordsOnly)

{

if (i > 0)

searchInText = this[i - 1].Text.LastOrDefault() + searchInText;

else

searchInText = '\0' + searchInText;

}

int j = i + 1;

while (searchInText.Length > searchTextLength && j < this.Count && searchFormat.IsSame(this[j].Run.GetFormat()))

{

searchInText += this[j].Text;

j++;

}

if (findWholeWordsOnly)

{

if (i < this.Count - 1)

searchInText = searchInText + this[i + 1].Text.LastOrDefault();

else

searchInText = searchInText + '\0';

}

if (searchInText.Length >= searchTextLength)

{

var k = searchInText.IndexOf(searchText);

while (k >= 0)

{

if (findWholeWordsOnly && k > 0)

{

if (!char.IsLetterOrDigit(searchInText[k - 1])   
&& !char.IsLetterOrDigit(searchInText[k + searchTextLength]))

return sumLength + k - 1;

k = searchInText.IndexOf(searchText, k + 1);

}

else

return sumLength + k;

}

}

}

}

sumLength += itemText.Length;

}

return -1;

}

Replace() method can be simplified. There is no need to handle the case of null searchFormat, as the Find() method handles this case. However, we must pass options and startPosition parameters to the Find() method.

public bool Replace(int startPosition, string searchText, TextFormat? searchFormat, string replacementText, TextFormat? replacementFormat, FindAndReplaceOptions? options = null)

{

var k = Find(startPosition, searchText, searchFormat, options);

if (k >= 0)

{

return ReplaceAt(k, searchText.Length, replacementText, replacementFormat);

}

return false;

}

Now we invoke Replace() method with FindWholeWordsOnly option:

Replace("is", new TextFormat { Italic = true }, "is", new TextFormat { Italic = false },  
new FindAndReplaceOptions { FindWholeWordsOnly = true});

We get:

This text <b>**is bold,**</b> and <i>*this* </i>is<i> *italicized*</i>.

## Case-insensitive search and replace

Now, let’s focus on the case-insensitive search. We declare MatchCaseInsentive option:

public record FindAndReplaceOptions

{

public bool FindWholeWordsOnly;

public bool MathCaseInsensitive;

}

We can introduce a stringComparison variable and use the IndexOf() method with the stringComparison parameter. New implementation of the Find() method is as follows:

public int Find(int startPosition, string searchText, TextFormat? searchFormat, FindAndReplaceOptions? options = null)

{

var findWholeWordsOnly = options?.FindWholeWordsOnly ?? false;

var matchCaseInsensitive = options?.MatchCaseInsensitive ?? false;

var stringComparison =

matchCaseInsensitive ? StringComparison.CurrentCultureIgnoreCase : StringComparison.CurrentCulture;

if (searchFormat == null)

{

var searchInText = GetText();

if (findWholeWordsOnly)

{

searchInText = '\0' + searchInText + '\0';

var k = searchInText.IndexOf(searchText, startPosition, stringComparison);

while (k > 0 && k + searchText.Length < searchInText.Length)

{

if (!char.IsLetterOrDigit(searchInText[k - 1])   
&& !char.IsLetterOrDigit(searchInText[k + searchText.Length]))

return k - 1;

k = searchInText.IndexOf(searchText, k + 1, stringComparison);

}

return -1;

}

else

return searchInText.IndexOf(searchText, startPosition, stringComparison);

}

var searchTextLength = searchText.Length;

var sumLength = 0;

for (int i = 0; i < this.Count; i++)

{

var itemText = this[i].Text;

if (sumLength + itemText.Length > startPosition)

{

if (searchFormat.IsSame(this[i].Run.GetFormat()))

{

var searchInText = itemText;

if (findWholeWordsOnly)

{

if (i > 0)

searchInText = this[i - 1].Text.LastOrDefault() + searchInText;

else

searchInText = '\0' + searchInText;

}

int j = i + 1;

while (searchInText.Length > searchTextLength && j < this.Count && searchFormat.IsSame(this[j].Run.GetFormat()))

{

searchInText += this[j].Text;

j++;

}

if (findWholeWordsOnly)

{

if (i < this.Count - 1)

searchInText = searchInText + this[i + 1].Text.LastOrDefault();

else

searchInText = searchInText + '\0';

}

if (searchInText.Length >= searchTextLength)

{

var k = searchInText.IndexOf(searchText, stringComparison);

while (k >= 0)

{

if (findWholeWordsOnly && k > 0)

{

if (!char.IsLetterOrDigit(searchInText[k - 1])   
&& !char.IsLetterOrDigit(searchInText[k + searchTextLength]))

return sumLength + k - 1;

k = searchInText.IndexOf(searchText, k + 1, stringComparison);

}

else

return sumLength + k;

}

}

}

}

sumLength += itemText.Length;

}

return -1;

}

Let’s test our example by changing “this” text to “that” text with MatchCaseInsensitive option. We need to invoke the Replace() method twice to change both occurrences of “this”.

Replace("this", "that" , new FindAndReplaceOptions{ MatchCaseInsensitive = true });

Replace("this", "that" , new FindAndReplaceOptions{ MatchCaseInsensitive = true });

We get:

that text <b>**is bold,**</b> and <i>*that is italicized*</i>.

It is not exactly what we have expected, because the first “that” is lowercase and should start with uppercase letter. To repair this, we need to change the case of replacementText in the Replace() method. The code is as follows:

public bool Replace(string searchText, TextFormat? searchFormat,   
string replacementText, TextFormat? replacementFormat, FindAndReplaceOptions? options = null)

{

var k = Find(0, searchText, searchFormat, options);

if (k >= 0)

{

if (options?.MatchCaseInsensitive == true)

{

var foundText = GetText().Substring(k, searchText.Length);

if (foundText != searchText)

{

if (foundText.IsUppercase())

replacementText = replacementText.ToUpper();

else if (foundText.IsLowercase())

replacementText = replacementText.ToLower();

else if (foundText.IsTitlecase())

replacementText = replacementText.TitleCase();

}

}

return ReplaceAt(k, searchText.Length, replacementText, replacementFormat);

}

return false;

}

Now we get the correct result:

That text <b>**is bold,**</b> and <i>*that is italicized*</i>.

## Finding and replacing format despite of text content

Sometimes, there is a need to find text using a given text format and replace the text format without changing text.

First, we will divide the Find() method into two: FindTextWithoutFormat() and FindTextWithFormat().

private int FindText(int startPosition, string searchText, FindAndReplaceOptions? options = null)

{

var findWholeWordsOnly = options?.FindWholeWordsOnly ?? false;

var matchCaseInsensitive = options?.MatchCaseInsensitive ?? false;

var stringComparison =

matchCaseInsensitive ? StringComparison.CurrentCultureIgnoreCase : StringComparison.CurrentCulture;

var searchInText = GetText();

if (findWholeWordsOnly)

{

searchInText = '\0' + searchInText + '\0';

var k = searchInText.IndexOf(searchText, startPosition, stringComparison);

while (k > 0 && k + searchText.Length < searchInText.Length)

{

if (!char.IsLetterOrDigit(searchInText[k - 1])   
&& !char.IsLetterOrDigit(searchInText[k + searchText.Length]))

return k - 1;

k = searchInText.IndexOf(searchText, k + 1, stringComparison);

}

return -1;

}

return searchInText.IndexOf(searchText, startPosition, stringComparison);

}

We do not change the implementation of them.

public int FindTextWithFormat(int startPosition, string searchText, TextFormat searchFormat, FindAndReplaceOptions? options = null)

{

var findWholeWordsOnly = options?.FindWholeWordsOnly ?? false;

var matchCaseInsensitive = options?.MatchCaseInsensitive ?? false;

var stringComparison =

matchCaseInsensitive ? StringComparison.CurrentCultureIgnoreCase : StringComparison.CurrentCulture;

var searchTextLength = searchText.Length;

var sumLength = 0;

for (int i = 0; i < this.Count; i++)

{

var itemText = this[i].Text;

if (sumLength + itemText.Length > startPosition)

{

if (searchFormat.IsSame(this[i].Run.GetFormat()))

{

var searchInText = itemText;

if (findWholeWordsOnly)

{

if (i > 0)

searchInText = this[i - 1].Text.LastOrDefault() + searchInText;

else

searchInText = '\0' + searchInText;

}

int j = i + 1;

while (searchInText.Length > searchTextLength && j < this.Count && searchFormat.IsSame(this[j].Run.GetFormat()))

{

searchInText += this[j].Text;

j++;

}

if (findWholeWordsOnly)

{

if (i < this.Count - 1)

searchInText = searchInText + this[i + 1].Text.LastOrDefault();

else

searchInText = searchInText + '\0';

}

if (searchInText.Length >= searchTextLength)

{

var k = searchInText.IndexOf(searchText, stringComparison);

while (k >= 0)

{

if (findWholeWordsOnly && k > 0)

{

if (!char.IsLetterOrDigit(searchInText[k - 1])   
&& !char.IsLetterOrDigit(searchInText[k + searchTextLength]))

return sumLength + k - 1;

k = searchInText.IndexOf(searchText, k + 1, stringComparison);

}

else

return sumLength + k;

}

}

}

}

sumLength += itemText.Length;

}

return -1;

}

We also declare a method of FindFormat(). This method does not have a searchText parameter, but it has foundLength output parameter, which is set to the length of the text when searchFormat is found. We can use a findWholeWordsOnly option to check if there is no letter or digit before or after the formatted text.

public int FindFormat(int startPosition, TextFormat searchFormat, FindAndReplaceOptions? options, out int foundLength)

{

var findWholeWordsOnly = options?.FindWholeWordsOnly ?? false;

var sumLength = 0;

for (int i = 0; i < this.Count; i++)

{

var itemText = this[i].Text;

if (sumLength + itemText.Length > startPosition)

{

if (searchFormat.IsSame(this[i].Run.GetFormat()))

{

var searchInText = itemText;

if (findWholeWordsOnly)

{

if (i > 0)

searchInText = this[i - 1].Text.LastOrDefault() + searchInText;

else

searchInText = '\0' + searchInText;

}

int j = i + 1;

while (j < this.Count && searchFormat.IsSame(this[j].Run.GetFormat()))

{

searchInText += this[j].Text;

j++;

}

if (findWholeWordsOnly)

{

if (i < this.Count - 1)

searchInText = searchInText + this[i + 1].Text.LastOrDefault();

else

searchInText = searchInText + '\0';

}

if (findWholeWordsOnly)

{

if (!char.IsLetterOrDigit(searchInText[0]) && !char.IsLetterOrDigit(searchInText[searchInText.Length - 1]))

{

foundLength = searchInText.Length - 2;

return sumLength;

}

}

else

{

{

foundLength = searchInText.Length;

return sumLength;

}

}

}

}

sumLength += itemText.Length;

}

foundLength = 0;

return -1;

}

To end with the Find() method we change the type of searchText parameter to null-allowing string and invoke the above private methods. We also add foundLength output parameter to pass output from FindFormat() method. When FindText() or FindTextWithFormat() are invoked, the foundLength returns just the length of the searchText.

public int Find(int startPosition, string? searchText, TextFormat? searchFormat, FindAndReplaceOptions? options, out int foundLength)

{

foundLength = searchText?.Length ?? 0;

if (searchText != null && searchFormat == null)

return FindText(startPosition, searchText, options);

if (searchText != null && searchFormat != null)

return FindTextWithFormat(startPosition, searchText, searchFormat, options);

if (searchText == null && searchFormat != null)

return FindFormat(startPosition, searchFormat, options, out foundLength);

throw new ArgumentException("Both search text and search format are null.");

}

Now, there is a time to change implementation of the Replace() method. We change types of searchText and replacementText to nullable string. At the beginning of this method (green code), we invoke the Find() method to evaluate the foundPosition and foundLength. If the searchText or the searchFormat is not found, the Replace() method finishes.

Then we handle the case when we specify the searchText, and we want to change it’s format without changing the found text (orange code). So, if the searchText is not null and the replacementText is null we assume that the replacementText is the same as the searchText.

Next, we consider the MatchCaseInsensitive option (blue code). This option can be used when the replacementText is specified. The case of replacementText is adjusted to match that of foundText.

The last part of the method invokes one of ReplaceTextAt() or ReplaceFormatAt() methods.

public bool Replace(int startPosition, string? searchText, TextFormat? searchFormat, string? replacementText, TextFormat? replacementFormat, FindAndReplaceOptions? options = null)

{

var foundPosition = Find(startPosition, searchText, searchFormat, options, out var foundLength);

if (foundPosition<0)

return false;

if (searchText != null && replacementText == null)

replacementText = searchText;

if (options?.MatchCaseInsensitive == true && replacementText != null)

{

var foundText = GetText().Substring(foundPosition, searchText.Length);

if (foundText != searchText)

{

if (foundText.IsUppercase())

replacementText = replacementText.ToUpper();

else if (foundText.IsLowercase())

replacementText = replacementText.ToLower();

else if (foundText.IsTitlecase())

replacementText = replacementText.TitleCase();

}

}

if (replacementText != null)

return ReplaceTextAt(foundPosition, foundLength, replacementText, replacementFormat, options);

if (replacementFormat != null)

return ReplaceFormatAt(foundPosition, foundLength, replacementFormat, options);

throw new ArgumentException("Both replacement text and replacement format are null.");

}

The implementation of ReplaceTextAt() method is the same as before in ReplaceAt(). The implementation of ReplaceFormatAt() is simpler. We just take the Run items covered by position and length parameters and change their formatting.

private bool ReplaceFormatAt(int position, int length, TextFormat replacementFormat, FindAndReplaceOptions? options = null)

{

var sumLength = 0;

var selectedItem = -1;

for (int i = 0; i < this.Count; i++)

{

var itemText = this[i].Text;

if (sumLength + itemText.Length > position)

{

selectedItem = i;

break;

}

sumLength += itemText.Length;

}

if (selectedItem >= 0)

{

while (selectedItem < this.Count && (length > 0))

{

var itemText = this[selectedItem].Text;

var itemOldLength = itemText.Length;

length -= itemText.Length;

var nextItem = selectedItem + 1;

if (!replacementFormat.IsSame(this[selectedItem].Run.GetFormat()))

{

SetFormat(selectedItem, replacementFormat);

}

if (length <= 0)

break;

sumLength += itemOldLength;

selectedItem = nextItem;

}

return true;

}

return false;

}

We test this solution using the following requests:

Replace(null, new TextFormat { Italic = true }, null, new TextFormat { Bold = true });

Replace(null, new TextFormat { Bold = true, Italic = false }, null, new TextFormat { Bold = false, Italic = true });

First, we request to change the italic formatting to the bold one. Hoverer, as we did not specify Bold = false in the replacementFormat, the resulted formatting is bold added to italic.

This text <b>**is bold,**</b> and <b><i>***this is italicized***</i></b>.

The second request is to change bold but non-italic to italic but no-bold formatting. Only the first “is bold,” text is reformatted.

This text <i>*is bold,*</i> and <b><i>***this is italicized***</i></b>.

Next, we try to test FindWholeWordsOnly option when finding format. We prepare for this test by changing the first “is” in the word “This” to italic.

Replace("is", null, null, new TextFormat { Italic = true }, new FindAndReplaceOptions { FindWholeWordsOnly = false });

We get just this:

Th<i>*is*</i> text <i>*is bold,*</i> and <b><i>***this is italicized***</i></b>.

Then we request to change the first italic formatting searching with FindWholeWordsOnly option to bold-italic.

Replace(null, new TextFormat { Italic = true }, null, new TextFormat { Bold = true, Italic = true }, new FindAndReplaceOptions { FindWholeWordsOnly = true });

Because FindWholeWordsOnly option is set, then the first italic formatting of “is” in the word “This” does not match, and the second italic formatting is changed.

Th<i>*is*</i> text <b><i>***is bold,***</i></b> and <b><i>***this is italicized***</i></b>.

The formatted text in the examples is not self-explanatory. It shows the formatting changes to the original text.

# Processing special text elements

Run elements can hold not only Text, but also some other element types to store other information, both textual and non-textual. These types are the following (in the alphabetic order):

* AnnotationReferenceMark,
* Break,
* CommentReference,
* ContentPart,
* ContinuationSeparatorMark,
* CarriageReturn,
* DayLong,
* DayShort,
* DeletedFieldCode,
* DeletedText,
* Drawing,
* EndnoteReferenceMark,
* EndnoteReference,
* FieldChar,
* FootnoteReferenceMark,
* FootnoteReference,
* FieldCode,
* LastRenderedPageBreak,
* MonthLong,
* MonthShort,
* NoBreakHyphen,
* EmbeddedObject,
* PageNumber,
* PositionalTab,
* Ruby,
* SeparatorMark,
* SoftHyphen,
* SymbolChar,
* TabChar,
* YearLong,
* YearShort.

We should consider the role of these elements in the Run and how it influences the find and replace functions of FormattedText.

## Special characters

Generally, text in wordprocessing documents are stored in 16-bit Unicode charset. It means that such characters like tab, carriage return, no-break space, soft hyphen, hyphen, non-breaking hyphen, en-dash, em-dash, bullet can be contained among other text with the codes ‘\u0009’, ‘\u000D’, ‘\u00A0’, ‘\u00AD’, ‘\u2010’, ‘\u2011’, ‘\u2013’, ‘\u2014’, ‘\u2022’ accordingly. However, a few of these characters are stored separately, in the following element classes:

* TabChar – represents character code ‘\u0009’,
* CarriageReturn – represents character code ‘\u000D’,
* SoftHyphen – represents character code ‘\u00AD’,
* NoBreakHyphen – represents character code ‘\u2011’.

There are also other Run member elements which represent some non-Unicode special symbols:

* FootnoteReferenceMark – specifies the presence of a footnote reference mark in a footnote text. This mark is replaced by an automatically numbered text which follows the numbering format of footnotes. If it is contained in a run which is not part of a footnote, then it can be ignored.
* EndnoteReferenceMark– specifies the presence of an endnote reference mark in a footnote text. This mark is replaced by an automatically numbered text which follows the numbering format of endnotes. If it is contained in a run which is not part of a endnote, then it can be ignored.
* SeparatorMark – implies rendering of a separator mark within the current run. A separator mark is a horizontal line that separates the contents of the main document story from the contents of footnotes or endnotes that begins on that page.
* ContinuationSeparatorMark – implies rendering of a continuation separator mark within the current run. A continuation separator mark is a horizontal line that separates the contents of the main document story from the contents of footnotes or endnotes which began on a previous page.
* AnnotationReferenceMark – specifies the presence of an annotation reference mark at the current location in the comment. It is usually replaced by the initials and a unique integer associated with its position in the document.
* PageNumber – implies rendering a current page number using decimal numbers.
* DayLong – implies rendering of a current date using the date format “DDDD”.
* DayShort – implies rendering of a current date using the date format “DD”.
* MonthLong – implies rendering of a current date using the date format “MMMM”.
* MonthShort – implies rendering of a current date using the date format “MM”.
* YearLong – implies rendering of a current date using the date format “YYYY”.
* YearShort – implies rendering of a current date using the date format “YY”.
* LastRenderedPageBreak – specifies that this position delimited the end of a page when this document was last saved by an application which paginates its content.

The GetText() extension method of the Run element must detect the presence of the above elements and convert them to their character code representations according to the Table 1. Also, the SetText() extension method must recognize the character codes in the text and create the corresponding special-character elements. As all these element classes derive from OpenXml EmptyType class, this is sufficient to ensure a mutually unique mapping from special characters to the members of the Run element.

Table . Special characters code mapping of Run member elements

|  |  |
| --- | --- |
| Run Member Element Class | Special Character Code |
| TabChar | 0009 (\t) |
| Break (Type=TextWrapping) | 000A (\n) |
| Break (Type=Column) | 000B (\v) |
| Break (Type=Page) | 000C (\f) |
| CarriageReturn | 000D (\r) |
| SoftHyphen | 00AD |
| NoBreakHyphen | 2011 |
| AnnotationReferenceMark | E00A |
| LastRenderedPageBreak | E00B |
| ContinuationSeparatorMark | E00C |
| SeparatorMark | E00D |
| EndnoteReferenceMark | E00E |
| FootnoteReferenceMark | E00F |
| PageNumber | E010 |
| DayLong | E011 |
| DayShort | E012 |
| MonthLong | E013 |
| MonthShort | E014 |
| YearLong | E015 |
| YearShort | E016 |
| FieldChar (FieldCharType=Begin) | E021 |
| FieldChar (FieldCharType=Separate) | E022 |
| FieldChar (FieldCharType=End) | E023 |
| SymbolChar | E025 |
| Ruby | E026 |
| EmbeddedObject | E027 |
| Drawing | E028 |
| PositionalTab | E029 |
| CommentReference | E02A |
| FieldCode | E02B |
| DeletedFieldCode | E02C |
| DeletedText | E02D |
| EndnoteReference | E02E |
| FootnoteReference | E02F |
| ContentPart | E030 |

Some special character codes are in the Control Characters C0 area (below ‘\u0020’). For others, we use the Private Use Area of Unicode charset (from ‘\uE000’ to ‘\uF8FF’).

For these Run member classes which can have some properties (like Break or PositionalTab) and for such members which are derived from OpenXmlCompositeElement (like Drawing or EmbeddedObject), we also use special character codes to specify the presence of these elements, but we must provide some other way to support access to the instances of these elements. We will see it in the following sections:

## Breaks and Last Rendered Page Break

There is a special member of the Run to represent a break in the text. This element moves the position of text rendering vertically (and horizontally) to the beginning of the:

* next page,
* next column of text,
* next line.

The position of the text is determined by the Type property of the Break element. In the GetText() method of the Run, the Break element is converted to a special character (according to the Type of the Break):

* Page break – is represented by character code ‘\u000C’,
* Column break – is represented by character code ‘\u000B’,
* Line break – (named as TextWrapping) is represented by character code ‘\u000A’.

In the SetText() method of the Run, these characters codes are recognized and converted to the Break element with the appropriate Type.

Example: Let’s consider the document in which there is a page break between two lines. Usually, the Break element is placed in the separate Paragraph and in the separate Run like below:

<w:p w14:paraId="20A860B5" w14:textId="15C3B12D" w:rsidR="00AC7F04" w:rsidRDefault="00AF29AF" w:rsidP="00F043F7">

<w:r>

<w:t xml:space="preserve">This text is on one page.</w:t>

</w:r>

</w:p>

<w:p w14:paraId="0F70CDD0" w14:textId="77777777" w:rsidR="00AC7F04" w:rsidRDefault="00AC7F04">

<w:r>

<w:br w:type="page"/>

</w:r>

</w:p>

<w:p w14:paraId="0B3545CE" w14:textId="4FF83E36" w:rsidR="007D6C6C" w:rsidRPr="00C0491F" w:rsidRDefault="00AC7F04" w:rsidP="00F043F7">

<w:r>

<w:lastRenderedPageBreak/>

<w:t>And this is on another page.</w:t>

</w:r>

</w:p>

But it can be left at the end of the first line:

<w:p w14:paraId="20A860B5" w14:textId="15C3B12D" w:rsidR="00AC7F04" w:rsidRDefault="00AF29AF" w:rsidP="00F043F7">

<w:r>

<w:t xml:space="preserve">This text is on one page.</w:t>

<w:br w:type="page"/>

</w:r>

</w:p>

<w:p w14:paraId="0B3545CE" w14:textId="4FF83E36" w:rsidR="007D6C6C" w:rsidRPr="00C0491F" w:rsidRDefault="00AC7F04" w:rsidP="00F043F7">

<w:r>

<w:lastRenderedPageBreak/>

<w:t>And this is on another page.</w:t>

</w:r>

</w:p>

It is also OpenXml complaint to place the Break element in the middle of a paragraph:

<w:p w14:paraId="20A860B5" w14:textId="15C3B12D" w:rsidR="00AC7F04" w:rsidRDefault="00AF29AF" w:rsidP="00F043F7">

<w:r>

<w:t xml:space="preserve">This text is on one page.</w:t>

<w:br w:type="page"/>

<w:lastRenderedPageBreak/>

<w:t>And this is on another page.</w:t>

</w:r>

</w:p>

The LastRenderedPageBreak element give us additional information that Word has processed the Break element and moved the text to the next page. The text in the last example would be encoded as:

This text is on one page.\f\uE00BAnd this is on another page.

## Tabs and positional tabs

The TabChar element in the Run moves the horizontal position text rendering to the next entry in tabs definitions in ParagraphProperties element.

Example Let’s define two custom tab stops at 1.5" and 3.5" in the ParagraphProperties element. It will be stored as:

<w:pPr>

<w:tabs>

<w:tab w:val="left" w:pos="2160" />

<w:tab w:val="left" w:pos="5040" />

</w:tabs>

</w:pPr>

If the current rendering position of the text is between 1.5" and 3.5" and the TabChar element occurs, the next character after TabChar will be rendered at 3.5" position.

If there is no custom tab stops defined in the ParagraphProperties, then the next rendering position is a multiple of DefaultTabStop setting of the document, e.g.:

<w:settings ...>

<w:defaultTabStop w:val="708"/>

The PositionalTab element moves the rendering to an explicitly specified position. This element can have three attributes:

* Alignment – declares Left, Center or Right alignment,
* RelativeTo – declares object to which to align; it can be one of: Margin or Indent,
* Leader – declares a leader character to render before text: Dot, MiddleDot, Hyphen, Underscore (or None).

The PositionalTab element with its properties can’t be converted to a single Unicode character. We need to use some special coding. We could use XML coding, but it would be complicated to evaluate character positions properly. The solution would be to code such Run members as single Unicode characters and refer to original OpenXml member elements to get the properties.

We would code PositionalTab elements using the private-code area of Unicode charset (codes from ‘\uE000’ to ‘\uF8FF’). We chose the character code of ‘\uE009’ to represent the PositionalTab.

## Encoding Run members with properties

Let’s consider the SymbolChar element. It has two properties:

* Char – specifies the hexadecimal code of the character,
* Font – specifies the font name to render the character.

Example: Consider such text:

Cut it here ✁

It can be expressed in OpenXml as:

<w:p w14:paraId="45B0D433" w14:textId="512D2B5C" w:rsidR="007D6C6C" w:rsidRPr="00C0491F" w:rsidRDefault="00872D7C" w:rsidP="00F043F7">

<w:r>

<w:t xml:space="preserve">Cut it here </w:t>

<w:sym w:font="Wingdings" w:char="F023"/>

</w:r>

</w:p>