Working with Text

Overview

Text in documents can be accessed by finding the glyphs nodes on a page. Each glyphs node will contain a run of text.

> **Mako Text Analysis**
>
> Some documents can be poorly constructed when it comes to text. You may find a document where each character is individually placed. In this case, when Mako imports a document, it uses intelligent heuristics to try and join these individual characters up into runs of text to simplify subsequent processing.

Finding Text

The following code finds all glyphs nodes font anywhere within the *content* node.

C#

```
CEDLVectIDOMNode glyphsNodes = content.findChildrenOfType(eDOMNodeType.eDOMGlyphsNode);
```

Updating Text

Updating text requires a little more knowledge about how documents are constructed. Specifically, a document may contain a font subset, which one that only contains glyphs for the characters used in the document.

For example, if a document contains glyphs with the characters 'a', 'b' and 'c', the embedded font may only contain visual information for the characters 'a', 'b' and 'c'. Therefore, if you update the glyphs text from 'abc' to 'abcd', you may find that the character 'd' is not rendered correctly when the document is output and viewed.

To avoid this issue, the full font should be loaded. This can be done by retrieving a specific font from disk, or by finding the font using Mako APIs.

Once the full font has been loaded or found, the text can be updated by using:

C#

```
glyphs.setFont(fullFont);
glyphs.setUnicodeString(updatedText);
glyphs.setIndices(string.Empty);
```

> **Font Indices**
>
> If the replacement font is different, it's always a good idea to reset the font's indices in the glyphs node.

Loading a Font from Disk

The code below finds the path to the Arial font in the Windows font folder.

C#

```
var arialFontPath = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.Fonts), "Arial.ttf");
using var stream = IInputStream.createFromFile(factory, arialFontPath);

using var font = IDOMFontOpenType.create(factory, stream);
```

Finding a Font using Mako

The code below finds the name of the font used in an existing glyphs node. This font is then attempted to be found using the Mako API `IJawsMako::findFont(...)`. If it can't find it, the code returns Arial as a default.

C#

```
using var originalFont = glyphs.getFont();

if (originalFont.getFontType() != IDOMFont.eFontType.eFontTypeOpenType)
    throw new InvalidOperationException("Font is not compatible with Mako.");

var originalTrueTypeFont = IDOMFontOpenType.fromRCObject(originalFont.toRCObject());
var fontName = originalTrueTypeFont.getFullName(factory, (int) glyphs.getFontIndex());

try
{
    return jawsMako.findFont(fontName, out _);
}
catch (Exception e)
{
    Console.WriteLine($"Failed to find font on local system: {e.Message}");

    // Fallback to Arial for testing.
    return jawsMako.findFont("Arial", out _);
}
```

Converting Text to Paths

In some cases, it may be desired to convert all text into paths. Thankfully Mako makes this very simple.

The code below calls `getEquivalentPath(...)` to get a new `IDOMPathNode`. This `IDOMPathNode` contains the vector content that represents the original text. It's been setup correctly so the next step is simple to replace the existing `IDOMGlyphs` node.

C#

```
using var path = glyphs.getEquivalentPath();
glyphs.getParentNode().replaceChild(glyphs, path);
```

Merging Duplicate Fonts

During serialization, the Mako SDK will assess the fonts used across the pages in the document and see if it's possible to merge duplicate fonts, and/or subset them futher. This is the default behaviour, when a full serialization is made.

This default behaviour can be skipped by either using the options in *IPDFOutput* as appropriate, or using incremental serialization instead of full serialization. When incremental serialization is used, the changes are appended to the original document, meaning that fonts in the original document will not be merged or edited. Incremental serialization is generally faster, but it results in larger output.

Accessing Font Tables

In some situations, it may be useful to access the tables of a font. This can be useful for finding font metrics, such as the font's ascend or descent values.

This can be done by getting the font's table accessor and writing the appropriate table out to a buffer. This buffer can then be used to find the required values at the appropriate offset.

C++

```
typedef union {

    uint16   uiValue;
    int16    iValue;
    char     bytes[sizeof(uint16)];

} Int16Union;

Int16Union convertToLittleEndian(const unsigned char* buffer)
{
    Int16Union    int16Value;

    int16Value.bytes[0] = buffer[1];
    int16Value.bytes[1] = buffer[0];

    return int16Value;
}

std::vector<uint8> getFontTable(const IDOMFontOpenTypePtr& openTypeFont, const std::string& id)
{
    auto tableAccessor = openTypeFont->getFontOpenTypeTableAccessor(0);

    const auto tableId = IDOMFontOpenType::Signature(id.c_str()).getSignatureID();

    if (!tableAccessor->hasTable(tableId))
        throw std::runtime_error("Font does not contain OS/2 data.");

    const auto tableSize = tableAccessor->getTableSize(tableId, false);

    std::vector<uint8> tableBuffer(tableSize);
    tableAccessor->tableCopy(&tableBuffer[0], tableSize, tableId, false);

    return tableBuffer;
}

void LogFontAscendDescend(const IDocumentPtr& document, const IJawsMakoPtr& jawsMako)
{
    const auto pageContent = document->getPage(0)->getContent();

    CEDLVector<IDOMNodePtr> glyphsNodes;
    pageContent->findChildrenOfType(eDOMGlyphsNode, glyphsNodes);

    const auto font = edlobj2IDOMGlyphs(glyphsNodes[0])->getFont();

    auto openTypeFont = edlobj2IDOMFontOpenType(font);

    // Grab the tables
    auto os2Buffer = getFontTable(openTypeFont, "OS/2");
    auto headBuffer = getFontTable(openTypeFont, "head");

    // These are in font design units
    const auto typographicAscender = convertToLittleEndian(&os2Buffer[68]);
    const auto typographicDescender = convertToLittleEndian(&os2Buffer[70]);

    // You can convert using the font header 'units per em'.
    const auto unitsPerEm = convertToLittleEndian(&headBuffer[18]);

    std::cout << "Font table details for: " << openTypeFont->getFullName(jawsMako, 0) << std::endl;

    std::cout << "Typographic ascender: " << typographicAscender.iValue << std::endl;
    std::cout << "Typographic descender: " << typographicDescender.iValue << std::endl;
    std::cout << "Units per EM: " << unitsPerEm.uiValue << std::endl;
}
```