ITEXT

pdfDebug

catches PDF output bugs easily and early

www.itextpdf.com

As an add-on for iText 7, pdfDebug enables developers to catch and correct bugs in PDF files during creation. It reflects the internal structure of PDF documents for advanced debugging on programs that use iText to create or manipulate PDFs – while allowing developers to work within the environment of their trusted IDE.

Why pdfDebug

PDF files consist of data structures and instructions that describe how to generate a specific output. As such, these instructions are akin to a piece of software code, potentially suffering from the same scourge: bugs! As with all bugs, it is important to catch them early and with the highest possible efficiency. Thus it is necessary to provide iText users with similar tools as available in the software development world: debugging tools.

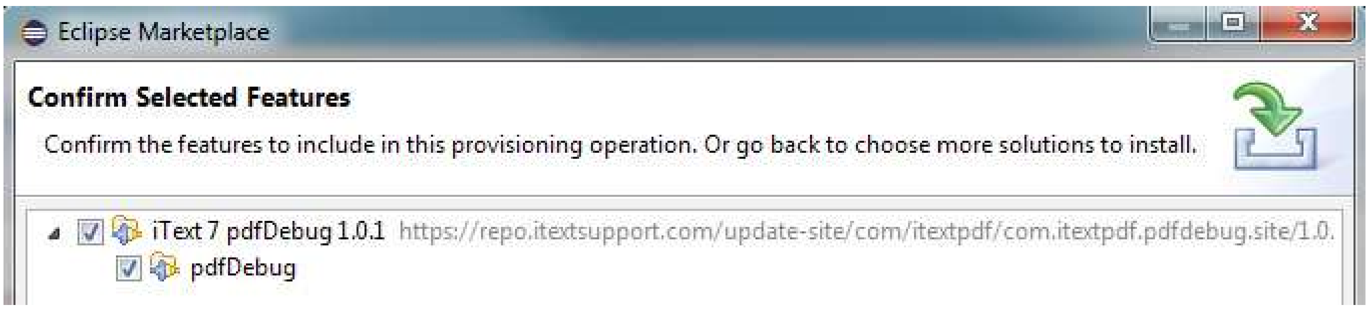
What is pdfDebug

pdfDebug is the debugging tool for users of iText while creating or processing PDF files. pdfDebug helps you visualize the structure of PDF documents in real time, while remaining in a trusted integrated development environment (IDE). It allows the developer to browse the document in a logical manner, using the known mechanism of breakpoints, and highlighting changes made throughout the PDF file. Today, it is the only PDF programming debugging tool in the world, for an early and easy detection of PDF bugs.

How to start

pdfDebug is an add-on to iText 7, for use in combination with an IDE. It requires an initial installation in this IDE. The first IDE supported by pdfDebug is the Eclipse environment – version 'Mars' and higher (for Java). Additional IDE's are planned such as Visual Studio, IDEA and more, based on the demand by developers. We call upon the community of iText users to collaborate with us in this endeavor, in order to extend this unique tool to as many IDE's as required by these users. Input regarding user priorities and development efforts are welcome. Contact us: https://itextpdf.com/contact.

To install pdfDebug, download the tool at the Eclipse marketplace (for the Eclipse version) and proceed with the standard Eclipse install procedure. To install the add-on, the install button can be dragged into the Eclipse workspace. A dialog will appear as shown below:



Confirm Selected Features

Confirm the features to include in this provisioning operation. Or go back to choose more solutions to install.

iText 7 pdfDebug 1.0.1 https://repo.itextsupport.com/update-site/com/itextpdf/com.itextpdf.pdfdebug.site/1.0.

pdfDebug

Simply select pdfDebug and follow the prompts to install it.

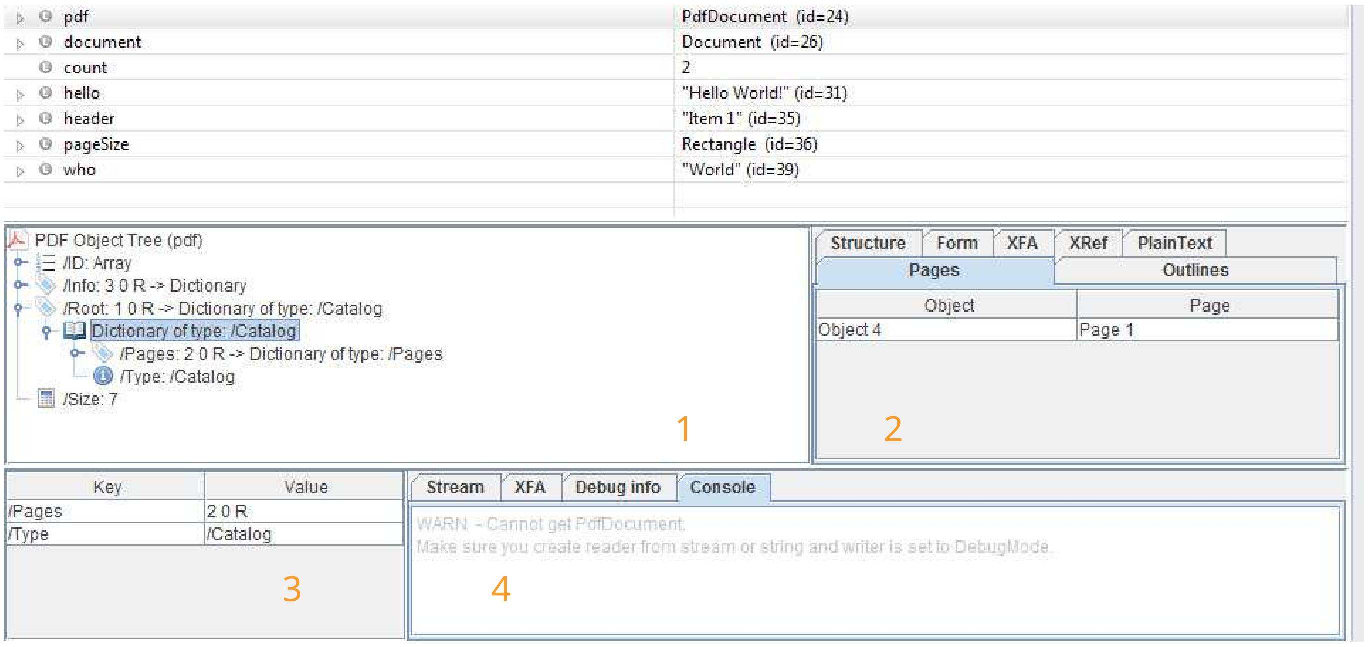
Attention:	pdfDebug requires iText 7.0.1 and higher. It is not compatible with version 7.0.0.

Once the environment has been set up, you need to activate the debug mode in iText. To enable debug mode, you'll need to add one line of code to your codebase:

```
PdfWriter writer = new PdfWriter(
    destination, new WriterProperties().useDebugMode()
);
```

This setting enables pdfDebug to access the internal iText data structures in order to visualize the PDF document structure in a user friendly manner. After adding a breakpoint to your code, you can access pdfDebug views and controls to navigate and inspect the current state of the PDF document. This breakpoint should be placed after instantiating the PdfWriter and PdfDocument classes.

In order to proceed with the debug procedure, select the PdfDocument object to view the pdfDebug screen:



```
pdf          PdfDocument  (id=24)
document     Document  (id=26)
count        2
hello        "Hello World!" (id=31)
header       "Item 1" (id=35)
pageSize     Rectangle  (id=36)
who          "World" (id=39)
```

```
PDF Object Tree (pdf)
  /ID: Array
  /Info: 3 0 R -> Dictionary
  /Root: 1 0 R -> Dictionary of type: /Catalog
    Dictionary of type: /Catalog
      /Pages: 2 0 R -> Dictionary of type: /Pages
      /Type: /Catalog
  /Size: 7
```

Object	Page
Object 4	Page 1

Key	Value
/Pages	2 0 R
/Type	/Catalog

```
WARN - Cannot get PdfDocument.
Make sure you create reader from stream or string and writer is set to DebugMode.
```

The pdfDebug screen in the Eclipse IDE (showing correct installation)

Explanation of the panels:

1. Tree view of the PDF document. Use this to navigate throughout the document structure.
2. Collection of panels that contain useful shortcuts to interesting objects in the PDF file.
3. Detailed view of the selected item in the tree view.
4. Informational panels of pdfDebug. Contains debugging information and console streams.

If the plugin has been installed correctly, the main panel should display the tree structure of the document. If not, one should check the console panel (lower right) for error messages (as e.g. 'WARN - Cannot get PdfDocument' – requires check if you made the code changes mentioned earlier).

How to use pdfDebug

For the use of pdfDebug, start your IDE, followed by iText and pdfDebug. In order to make the best of use of pdfDebug, some insight in the view on your PDF file is required.

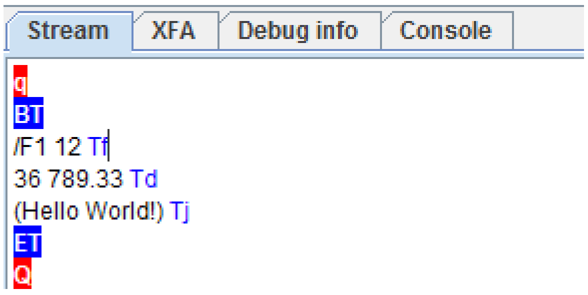
Abstractly a PDF file is a graph: a set of nodes that can point to one another. A node can be one of a few data types, including

- Dictionaries, a series of key-value pairs
- Streams, binary data
- Arrays, heterogeneous
- Reference to one of these objects

One of the more important constructs in a PDF file is a page. Pages are independent of each other and are selfcontained. For most people the following entries in the Page dictionary will be the most interesting ones:

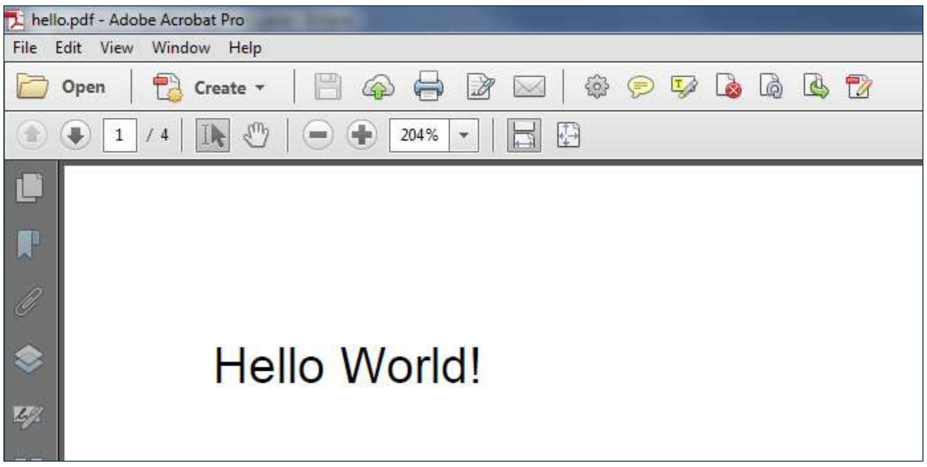
1. /Contents
2. /Resources
3. /MediaBox

/Contents is where the syntax of the page is kept. The syntax can be stored in one or multiple streams. If you have difficulties finding your content directly, look through multiple streams in the /Contents entry. PDF syntax looks like this:



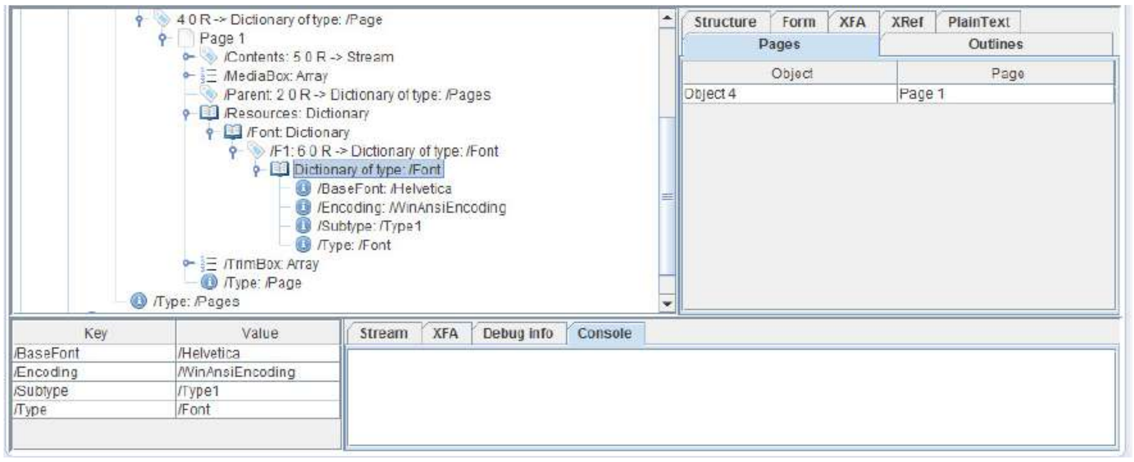
```
q
BT
/F1 12 Tf
36 789.33 Td
(Hello World!) Tj
ET
Q
```

The **BT** indicates that a text block has been initiated, while the **ET** signals the end of the text block. In PDF syntax you have operators and operands. The operands precede the operator. In this small section we have three operators, Tf, Td, and Tj. **Tf** sets the font to “/F1” and the size to 12. **Td** moves the text position to 36,789.33. And finally **Tj** draws “Hello World” to the screen:



The output of the sample code as seen in a PDF viewer (Adobe Acrobat)

Looking in the **/Resources** entry, we see the resources the page needs to render the output. In the case of our simple Hello World sample, we can see that there is one entry: /F1.



```
4 0 R -> Dictionary of type: /Page
  Page 1
    /Contents: 5 0 R -> Stream
    /MediaBox: Array
    /Parent: 2 0 R -> Dictionary of type: /Pages
    /Resources: Dictionary
      /Font: Dictionary
        /F1: 6 0 R -> Dictionary of type: /Font
          Dictionary of type: /Font
            /BaseFont: /Helvetica
            /Encoding: /WinAnsiEncoding
            /Subtype: /Type1
            /Type: /Font
    /TrimBox: Array
    /Type: /Page
/Type: /Pages
```

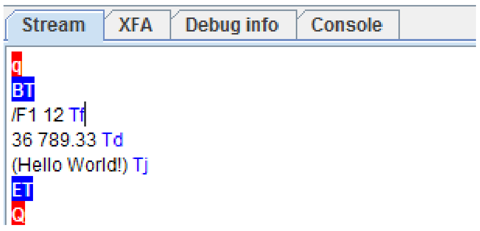
Object	Page
Object 4	Page 1

Key	Value
/BaseFont	/Helvetica
/Encoding	/WinAnsiEncoding
/Subtype	/Type1
/Type	/Font

This is the font referenced earlier in the content stream. You can see that in this case we added the font Helvetica.

How to catch a bug

On the second page in the code sample, a second paragraph to the second page was added. When inspecting the PDF in a PDF viewer, however, you won't see the text. Inspecting the content stream of the second page in pdfDebug, we see:

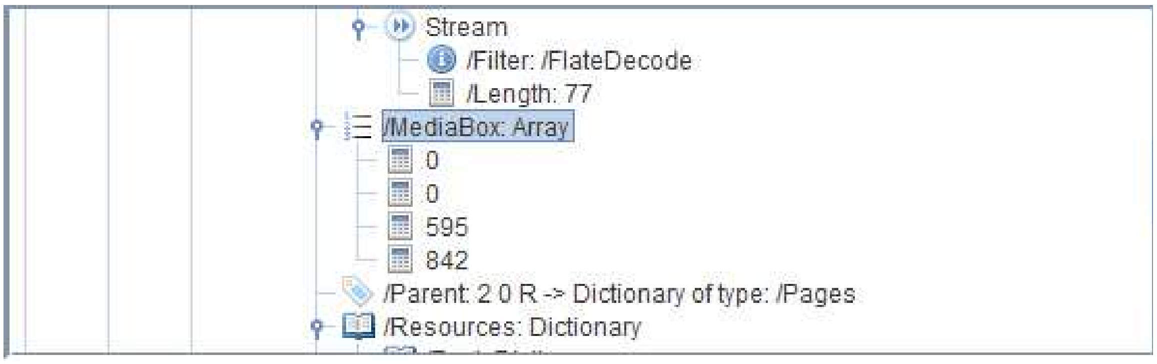


```
q
BT
/F1 12 Tf
36 789.33 Td
(Hello World!) Tj
ET
Q
```

On initial inspection, everything appears to be present: BT-ET, the font has been set, the text position has been set and the text has been added to the page. So what's the problem, the bug?

The answer is in another entry in the page dictionary: /MediaBox.

The PDF coordinate space is much bigger than the screen or print dimension. What you're seeing and using is basically a view port onto this coordinate space. This view port is defined in the media box:



```
Stream
    /Filter: /FlateDecode
    /Length: 77
/MediaBox: Array
    0
    0
    595
    842
/Parent: 2 0 R -> Dictionary of type: /Pages
/Resources: Dictionary
```

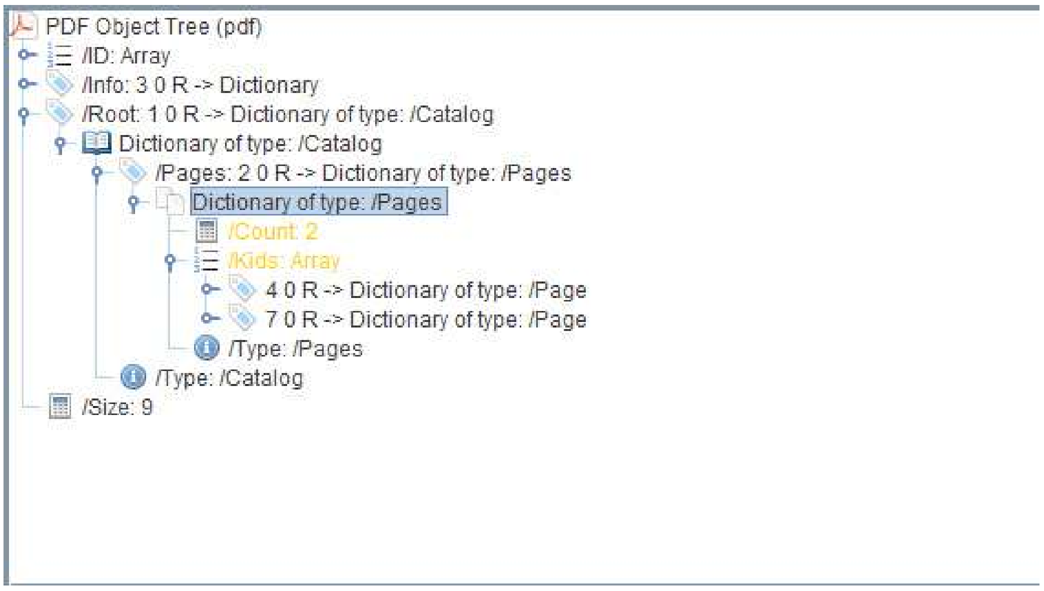
These 4 values define two points in the coordinate space, specifying 2 opposite corners of a rectangle. In this case, the lower left corner (0,0) and the upper right corner (595,842) are used. This means that the page has a width of 595 points and a height of 842 points.

Looking back at the syntax in the code sample, we see that the text position has been set to 900,702.98 – a mistake made by a user when creating the file in iText. With pdfDebug, this can be found fast and easy, and if the values in the code are changed to values within the bounds of the MediaBox, the text will appear.

Breakpoints

pdfDebug makes use of breakpoints in your iText code, an analogy with breakpoints in pieces of code in other programming languages, helping developers to go through a file in a logical way.

Going from breakpoint to breakpoint, it might be hard to track all the changes that are made to the PDF document. Or it might be confusing to find where exactly a change has happened. That's why we added highlighting of objects that have been changed:



```
PDF Object Tree (pdf)
  /ID: Array
  /Info: 3 0 R -> Dictionary
  /Root: 1 0 R -> Dictionary of type: /Catalog
    Dictionary of type: /Catalog
      /Pages: 2 0 R -> Dictionary of type: /Pages
        Dictionary of type: /Pages
          /Count: 2
          /Kids: Array
            4 0 R -> Dictionary of type: /Page
            7 0 R -> Dictionary of type: /Page
          /Type: /Pages
      /Type: /Catalog
  /Size: 9
```

What you see in the shortcuts panel

In the IDE, pdfDebug populates a slew of useful panels with information facilitating the inspection of PDF. The shortcuts panel (upper right panel) offers alternative ways to navigate through the PDF structures. All the information presented in these tabs is also available in the tree structure view, but to access specific structures, these shortcuts are more convenient.

- **Pages**
  An overview of all pages in this document. Clicking on a page will instantly select the page dictionary in the tree view.
- **Outlines**
  An overview of all the bookmarks in the PDF.
- **Structure**
  PDFs can contain information on the content of the PDF. If structure has been added, a PDF will know that a certain text block is a paragraph.

- **Form**
  An overview of all the form fields present in the loaded document.
- **XFA**
  An overview of the XML Forms Architecture.
- **XRef**
  An index of every object in the PDF file. Clicking on a reference will automatically select the corresponding object in the tree view.
- **PlainText**
  Shows a plain text representation of the PDF file.

Conclusion

In the overall document workflow and lifecycle, it is to the advantage of companies to develop PDF files quickly and efficiently. That applies to speedy availability of the documents and the shortest possible development time, as well as the most cost efficient way of developing the related PDF files. By using pdfDebug in combination with iText, developers can avoid output mistakes while developing and adapting those files, rather than amending them at a later stage. pdfDebug is a unique tool performing similar duties as debugging tools in traditional software development environments, including the use of trusted IDEs.

About iText

iText is the world's foremost platform to create PDF files and integrate them in corporate applications. Originally developed by Bruno Lowagie, iText is available as an open source product with community support, as well as a commercial product, worldwide supported by the iText Group NV (with offices in Europe, North America and Asia). In 2016, the popular iText 5 version was succeeded by the fully re-written and re-architectured iText 7 platform, with advanced add-on options. iText represents a unique capability of facilitating the use of dynamic and complex PDF documents in corporate document workflows, throughout the whole lifecycle of the documents.

For more information visit our website https://itextpdf.com/itext7/pdfDebug