



**pdfToolbox**

# **New in callas pdfToolbox 9**



**callas**

# Table of Contents

Large format .....	4
Add ink layer .....	5
Adding grommets .....	10
Tiling.....	13
Add borders .....	17
Add bleed .....	21
Variables and JavaScript: JavaScript.....	24
Taking variables to the next level .....	25
Variables and JavaScript: Variables in general .....	30
Variables using JavaScript: Overview .....	39
Variables using JavaScript: pdfToolbox objects and methods .....	53
Extracting information from an XML Report file via XPath (9.1).....	58
Using an external JSON jobticket file (9.1) .....	60
Defining variables using app.requires with closed choice of allowed values (9.1) .....	62
Using "trigger" values to adjust processing in a Process Plan (9.1) .....	64
Debugging JavaScript Variables (9.1) .....	67
Shapes .....	71
Shapes: An overview .....	72
Defining shapes .....	74
Applying shapes.....	91
"Shapes" features extended in pdfToolbox 9.1 .....	103
Efficiently creating varnish or white background (requires at least v9.1) .....	110
Spectral color and CxF .....	118
Embed CxF data (import) .....	119
Extract and remove CxF information.....	123
Analyze CxF information .....	129
Introduction: CxF and spectral data .....	133
New and extended properties .....	134
New and enhanced Properties in 9.0.....	135

New and enhanced Properties in 9.1.....	136
How to use the "Number of hits in the check" property (9.0) .....	137
New and extended Fixups.....	139
New and enhanced Fixups in pdfToolbox 9.1 .....	140
Wireframe and selective viewing.....	142
Examining page content.....	143
Advanced barcode and matrix code features .....	146
Advanced 2D code use cases: Deutsche Post DP Matrix, Data Matrix Industry, rainbow colored QR Code (requires pdfToolbox 9.1).....	147
Debugging of Profiles and Process plans (9.1) .....	153
How to create a detailed log when executing Process Plans (or Profiles, Checks or Fixups) ..	154
Processing Steps: Overview (9.1) .....	161
Design and more.....	162
Using metadata for standardisation.....	164
Viewing the layers in a document.....	166
Working with processing steps metadata for a layer.....	168
Checking processing steps information .....	171
Fixing processing steps data .....	174

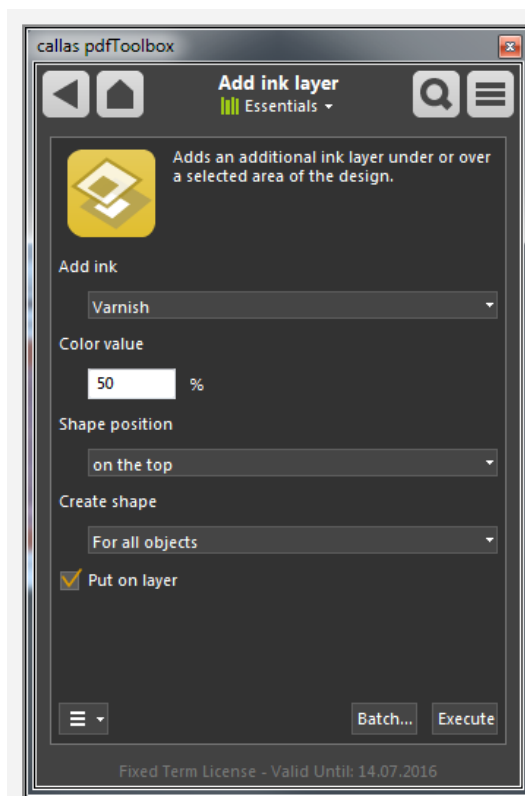
# Large format



# Add ink layer

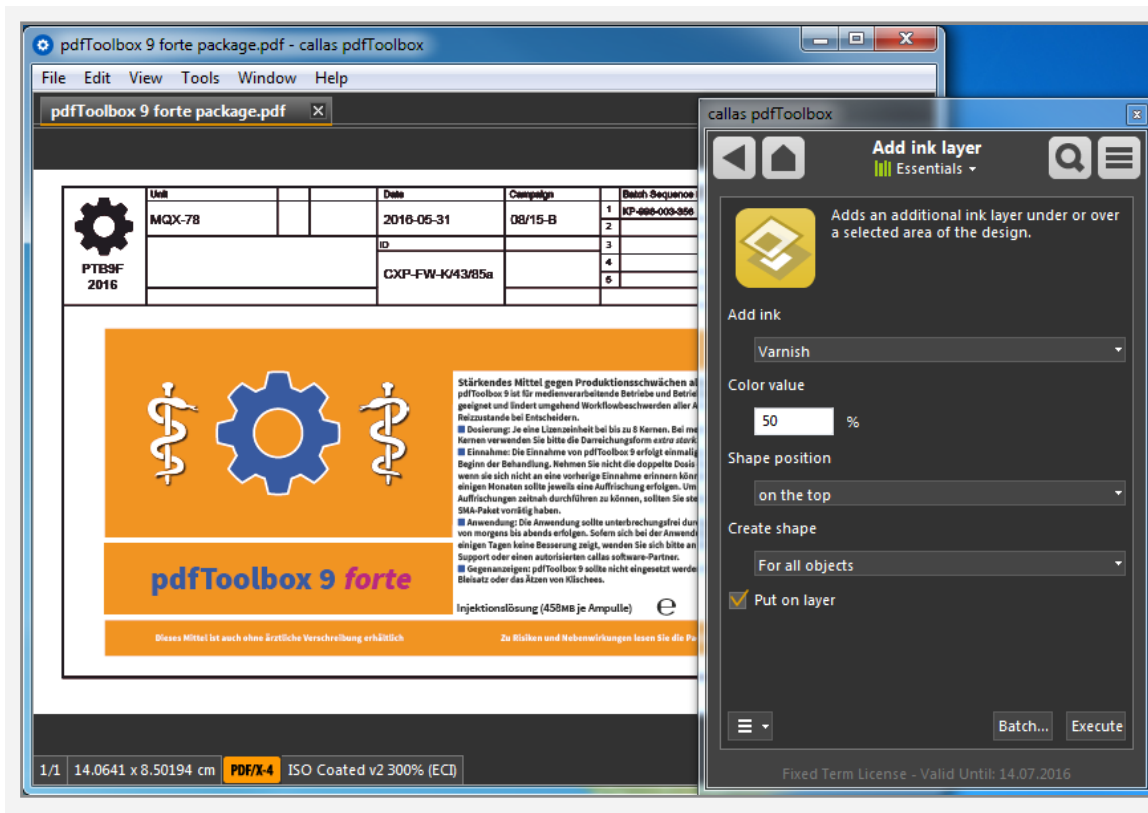
This functionality adds an additional, colored object to the PDF. For determining where existing objects are painting, the page will be internally rendered. A new shape will be created based on this result as a vector object.

## Available settings



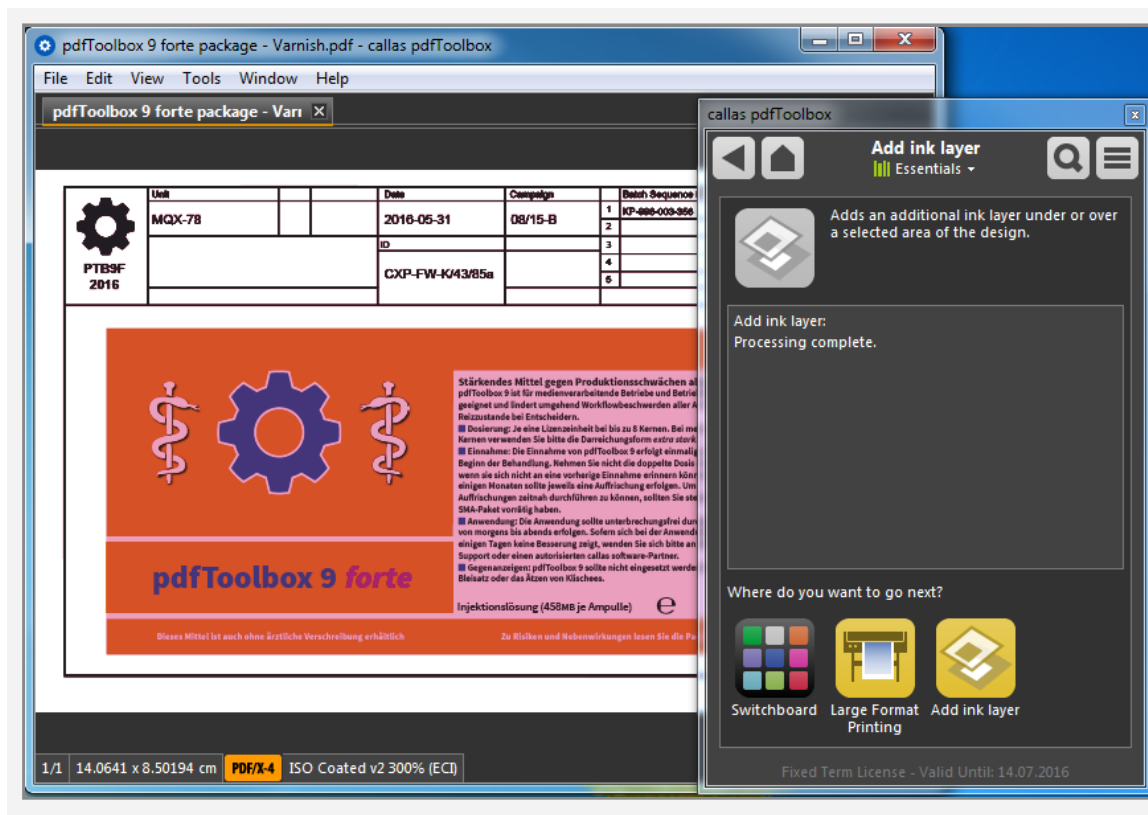
The name of the created spot color can be chosen and the color tint value be defined. Also the the position of the shape can be set. Optionally, the new object can be created on a layer.

## Create "Varnish" object for all painting content



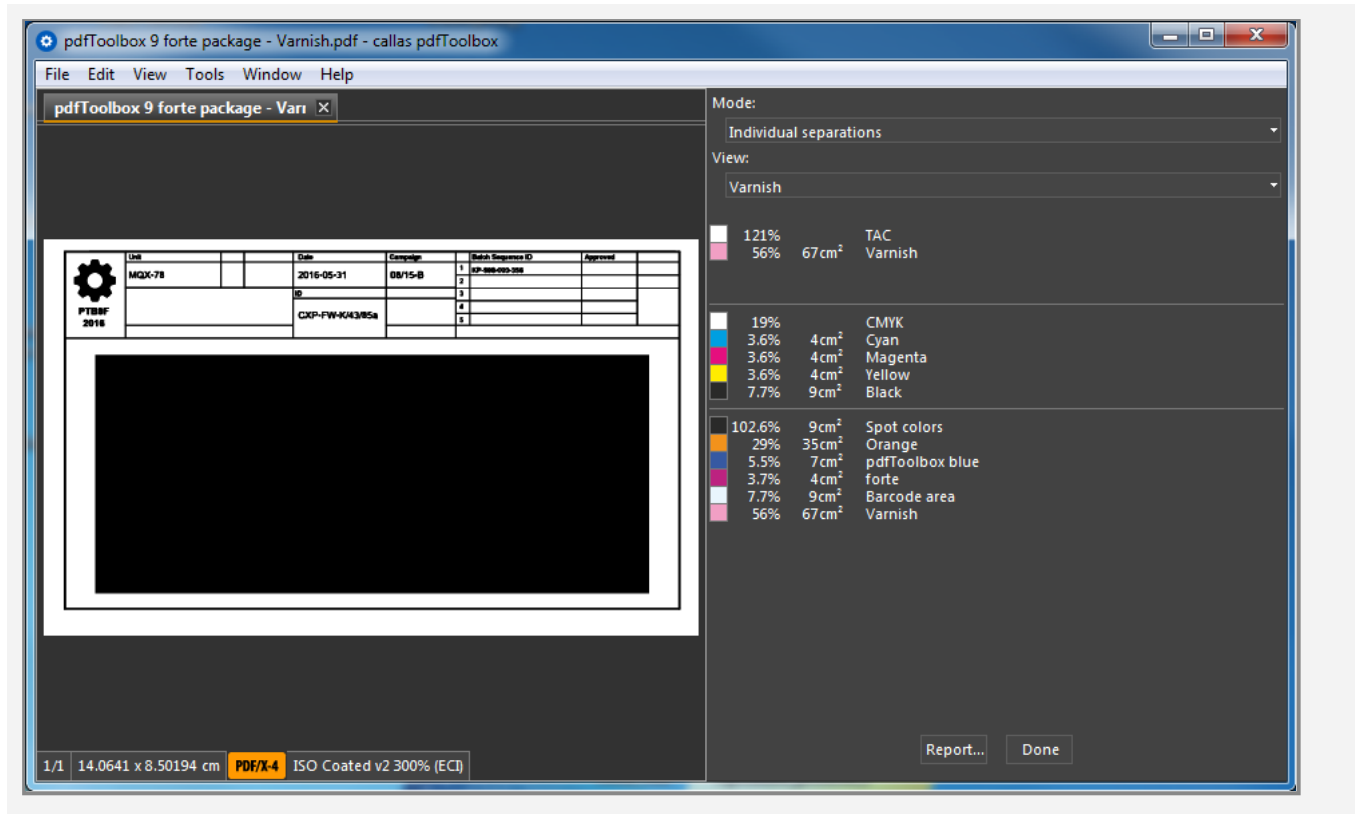
For example, to create a new object, which covers all painting content, just choose "For all objects" and define a color value.

## The result



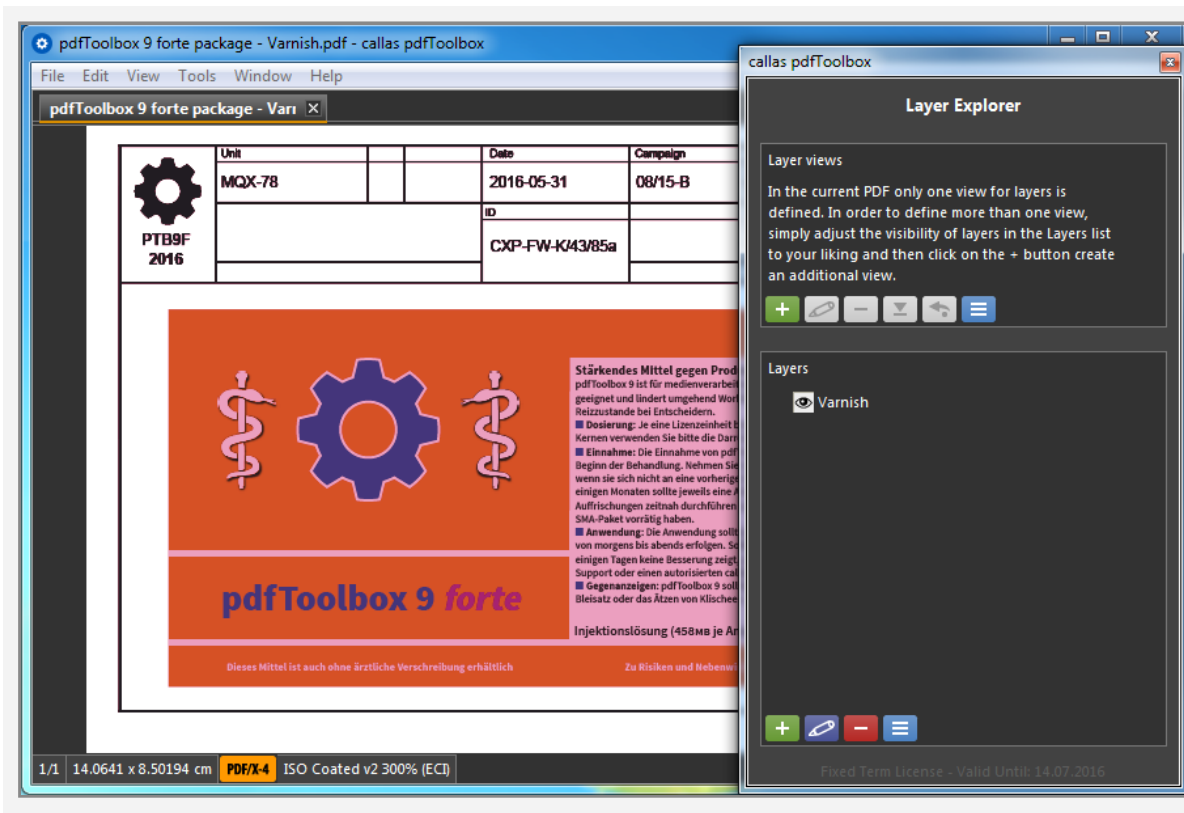
After processing a new vector object, using the spot color "Varnish" will be added. Covering all content, also objects using white.

## Inspecting the result - individual separation for spot color



When inspecting the result using "Visualize individual separations", the new created object using "Varnish" can easily be reviewed.

## Creation on a layer

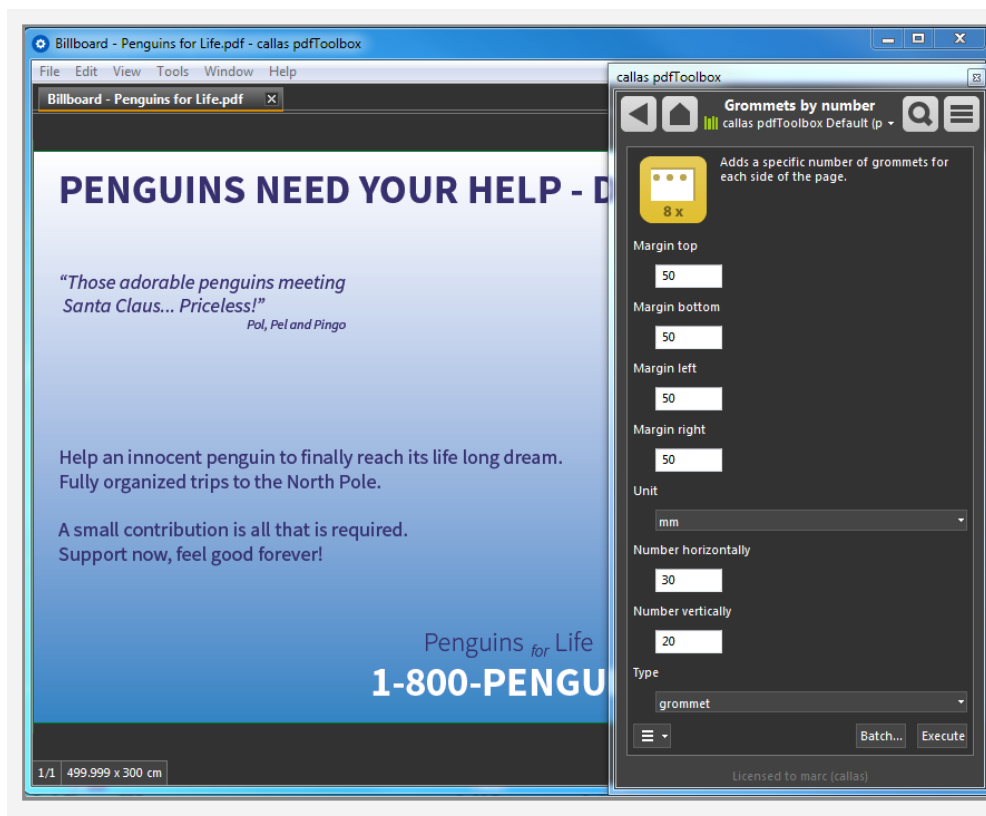


If required, the newly added object will be placed automatically on a layer, which will have the same like the spot color.

# Adding grommets

During the production of banner or other large format products, sometimes grommets must be added. To add marks, where these grommets shall be placed after the product is printed, this Switchboard action can be used.

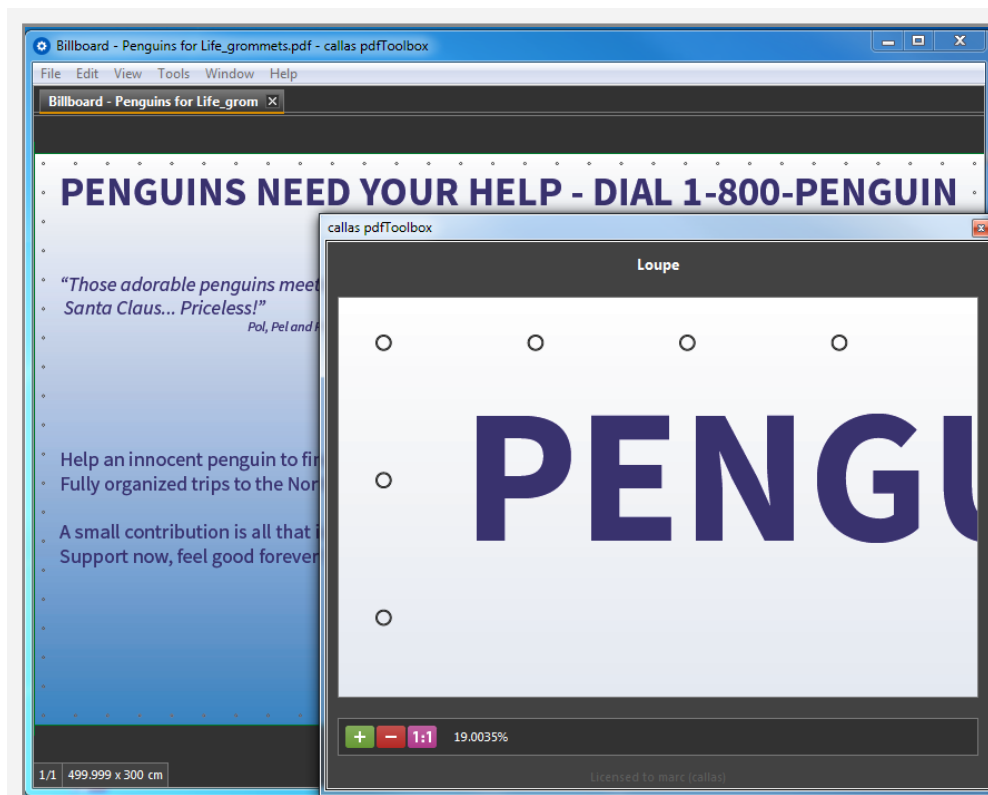
## Define the settings



To define the positioning of the grommets, the margin for all 4 edges can be defined.

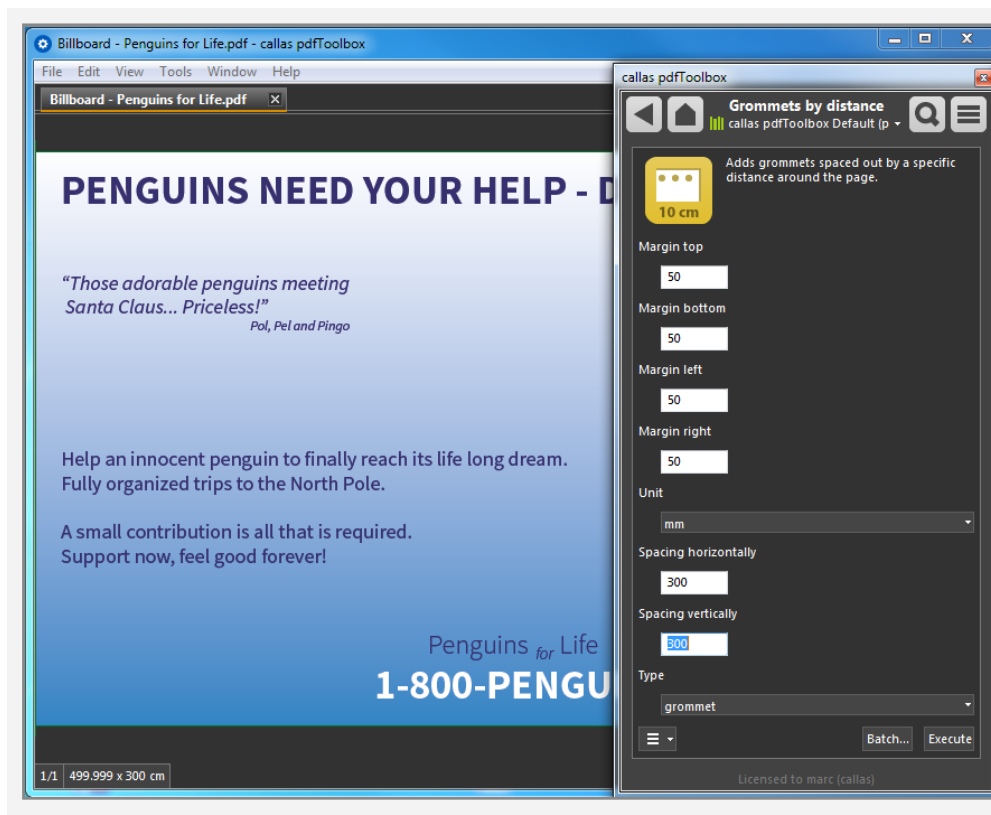
Of course the number of grommets for the horizontal and vertical edges must be defined. The internal calculation will determine the distance between the grommets.

## Inspecting the result



Marks for the grommets will be positioned accordingly to the defined settings.

## Grommets by distance



An additional way to add grommets to the document is by defining the distance between the grommets vertically and horizontally.

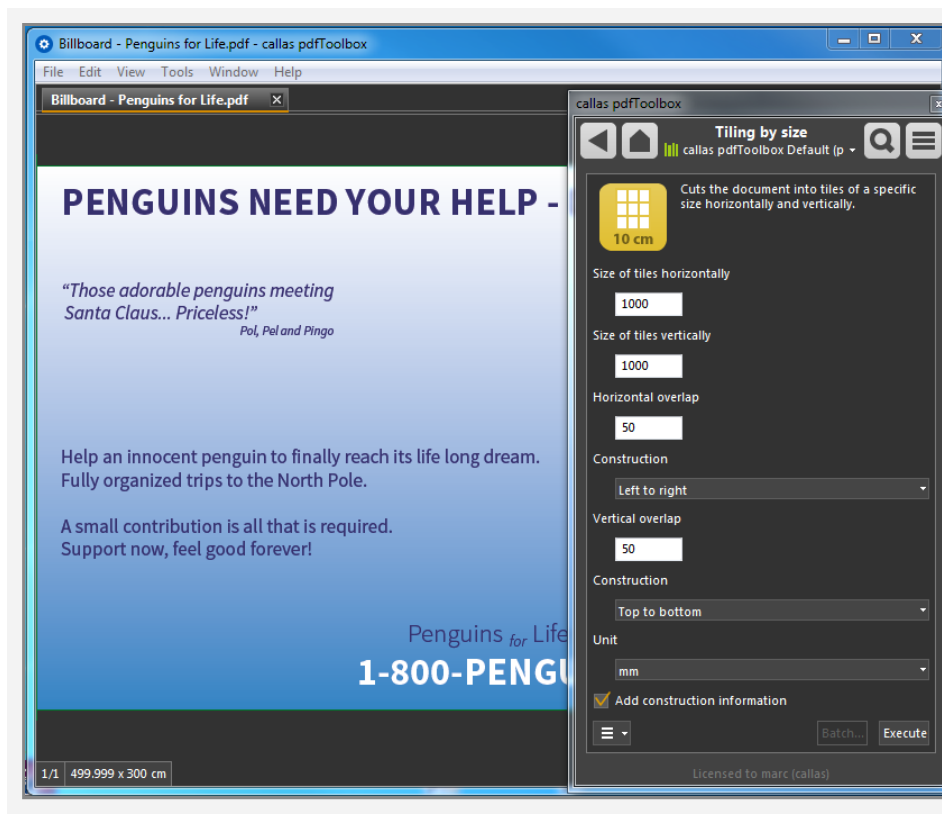


# Tiling

Tiling will cut the document in a number of parts to prepare the document for various large format printing methods.

The document can either be cut by a defined size for the resulting tiles or by the number of tiles horizontally and vertically.

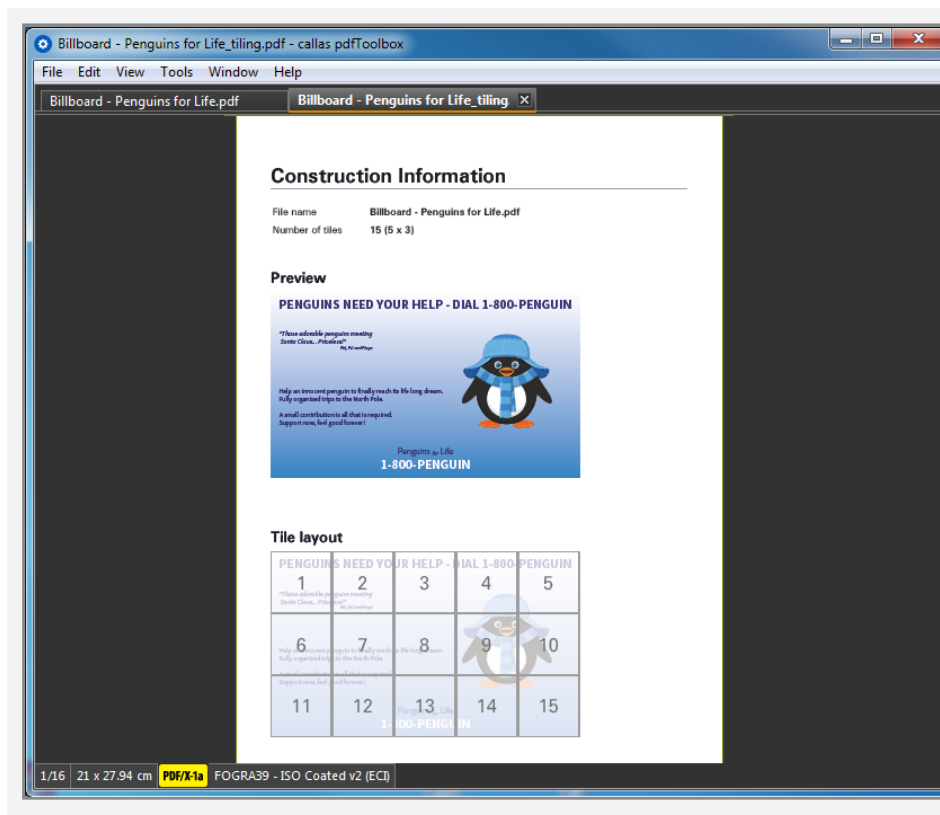
## Define the settings



Using the Switchboard action, it is possible to define the size of the resulting tiles and the overlap. Additionally the construction direction vertically and horizontally can be defined.

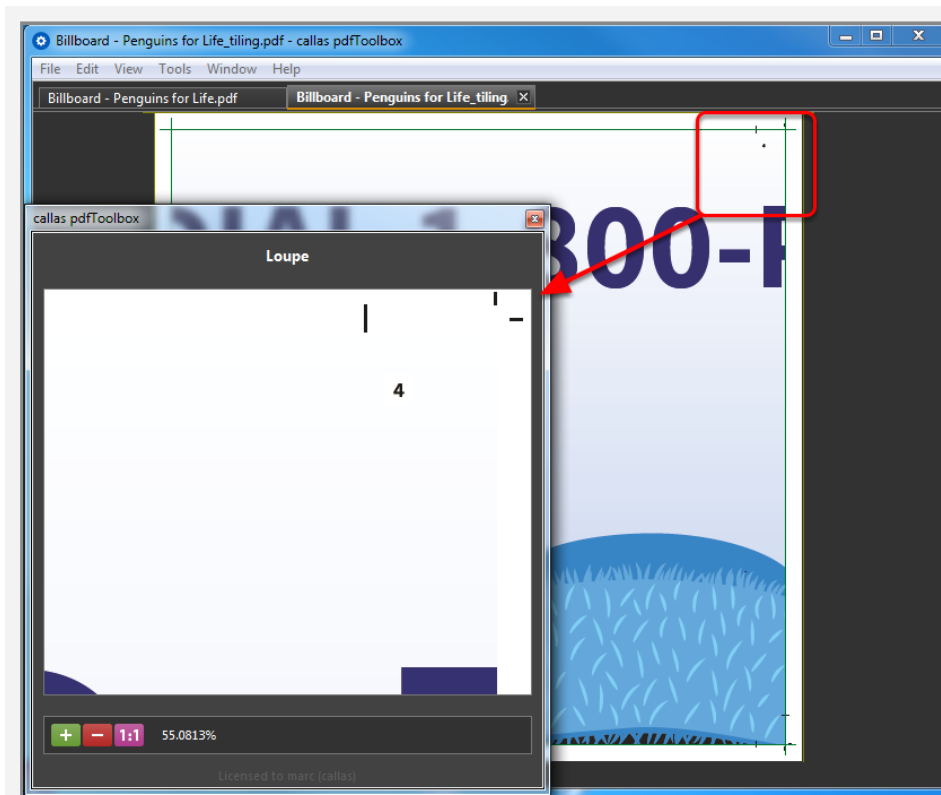
An additional page with construction information can be added as well.

## Construction information



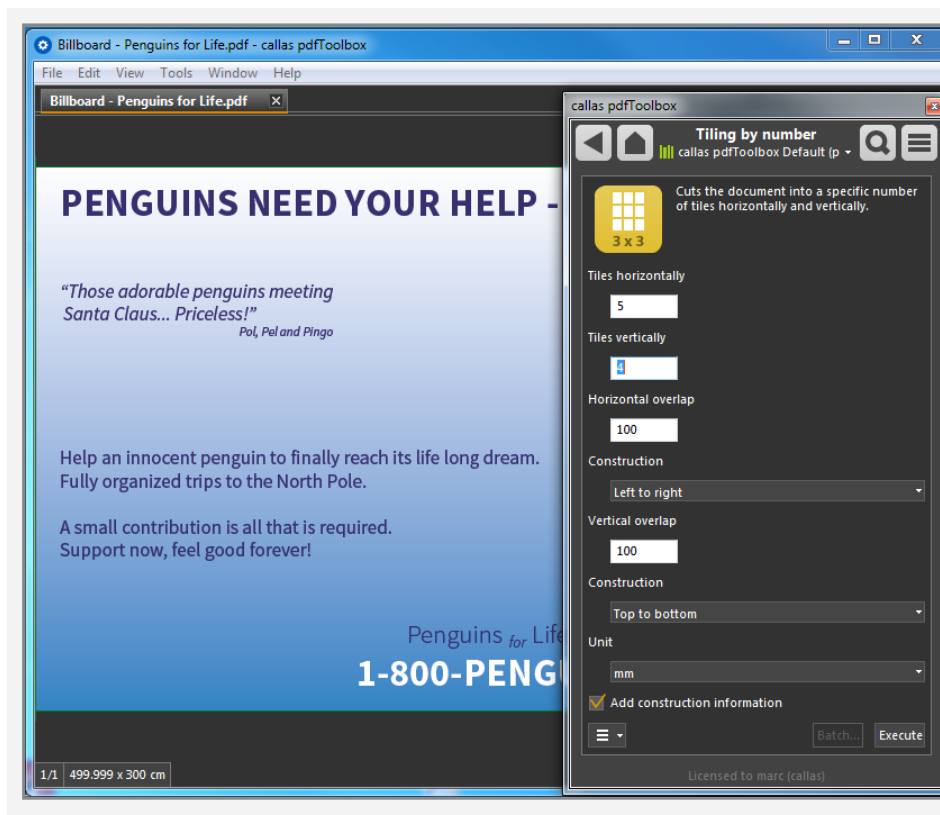
The construction information contains some basic information about the file and the number of tiles, as well as a sketch for the tile layout.

## Information for overlap



If an overlap has been defined, a small mark to indicate there the next, overlapping tile has to be positioned will be added. Also the number of the actual tile will be printed into the overlapping area (which will become covered by the next tile).

## Tiling by number

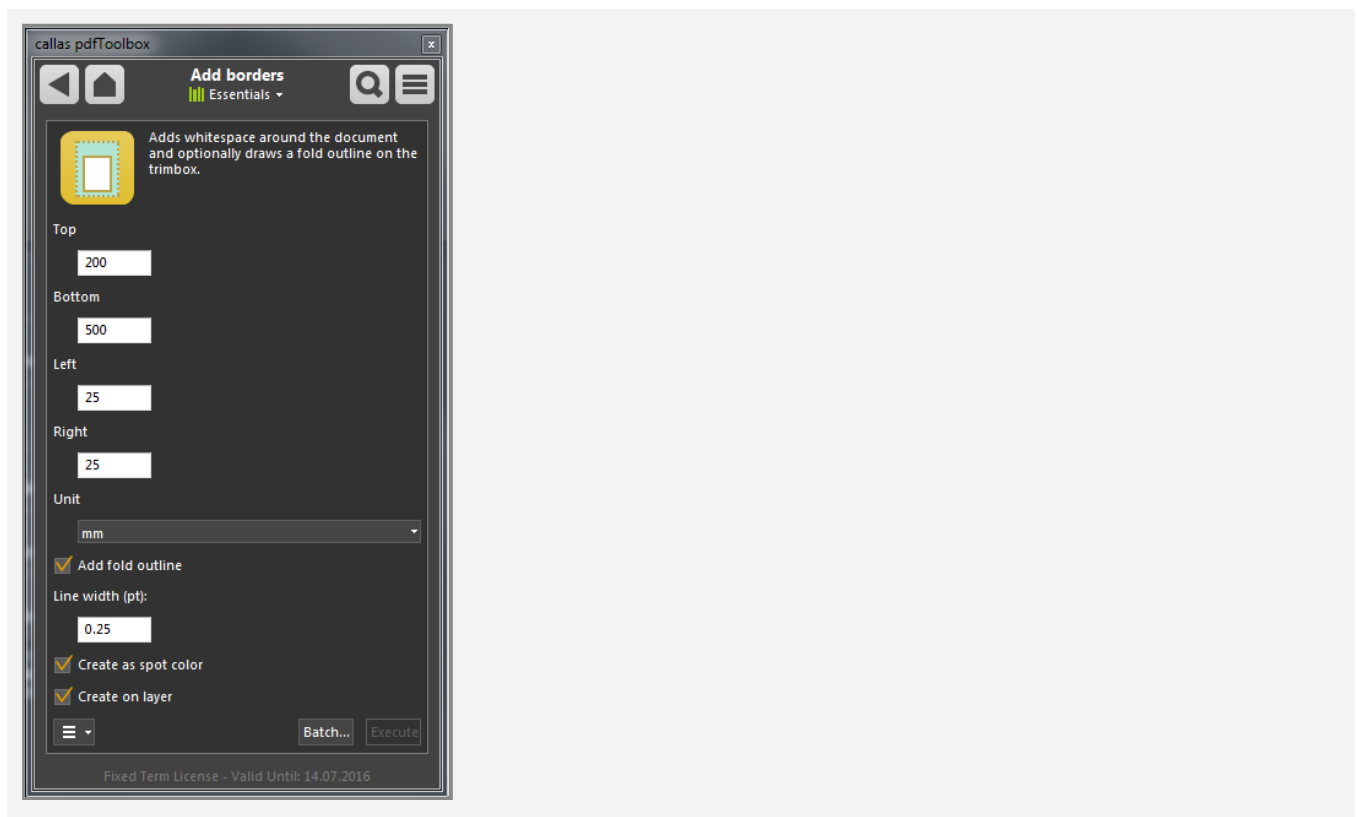


An additional way to tile the document is by defining the number of tiles vertically and horizontally. The overlap can be defined in the same way as described above.

# Add borders

Adds whitespace around the document by enlarging the existing visible area (defined by the CropBox). This can be useful for producing large format products like Roll Up Banners

## Available settings

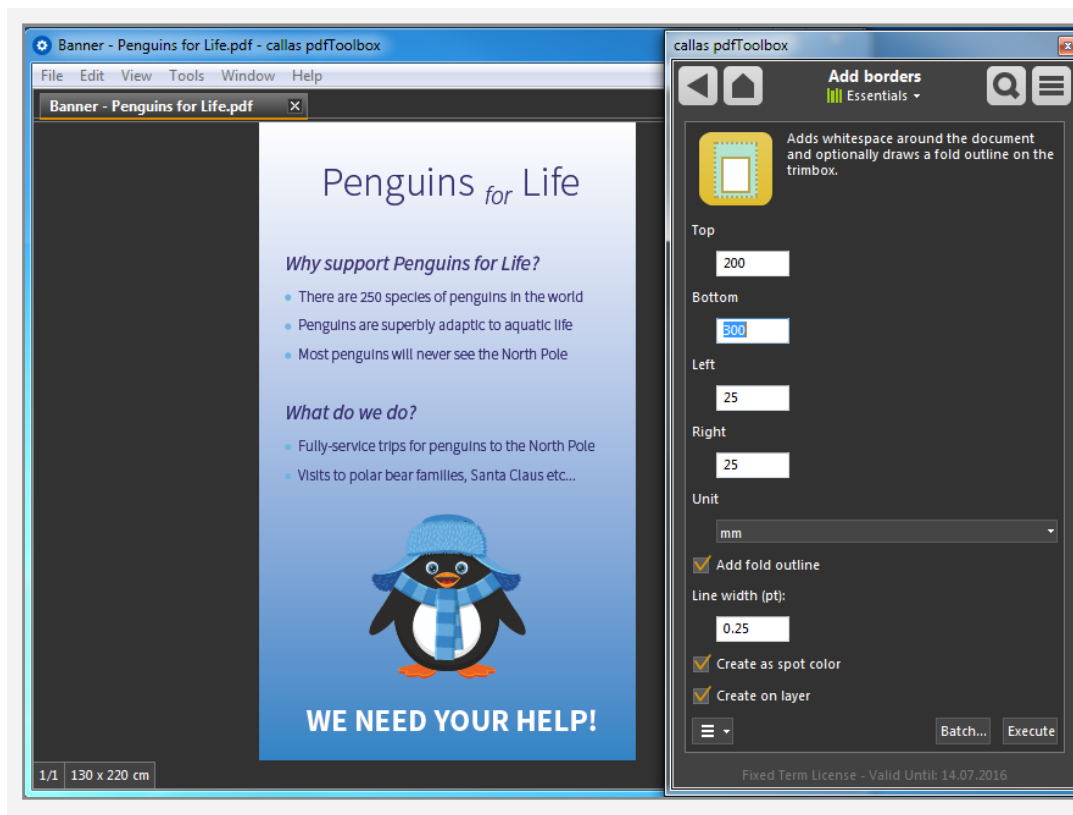


The margin to be added can be defined for all 4 edges of the document. The used unit for these values can be chosen as well.

By activating "Add fold outline", the former size of the page will be marked by an outline, which allows cutting or positioning during production afterwards.

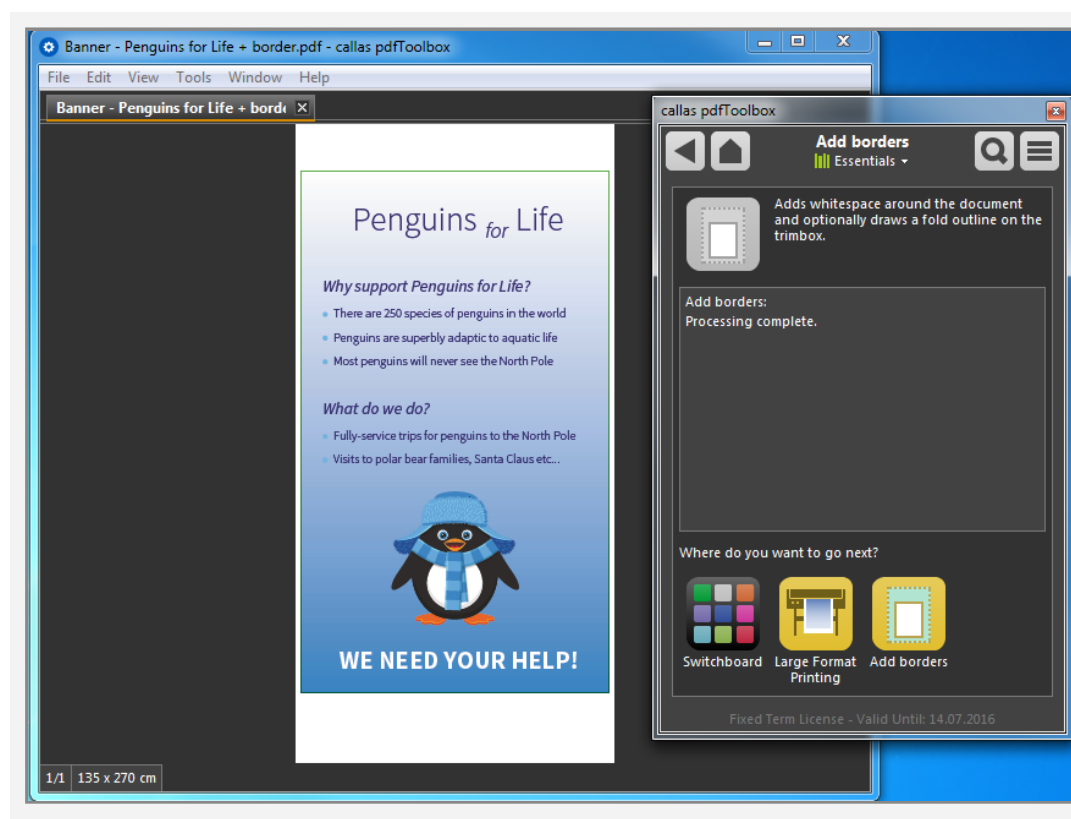
The line width, creation as a spot color or separating this outline on a layer can be optionally activated as well.

## Extend for Roll Up Banner



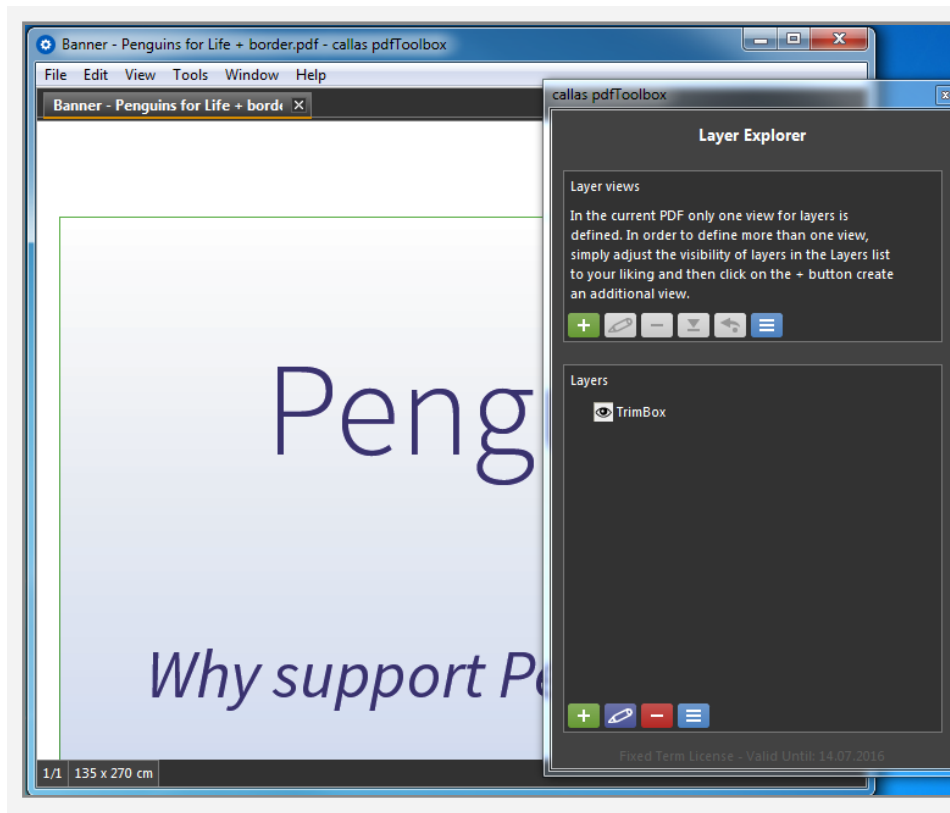
For adding space to prepare the PDF (which has the right dimension) for a Roll Up Banner, just enter the required values. Press "Execute" afterwards.

## Extended result PDF



The file gets enlarged by the defined values.

## Fold outline on a layer



As the "Add fold outline" was activated, the former page size became outlined. Causes by "Create on layer" this outline is placed on a layer, so it can be switch on or off easily.



# Add bleed

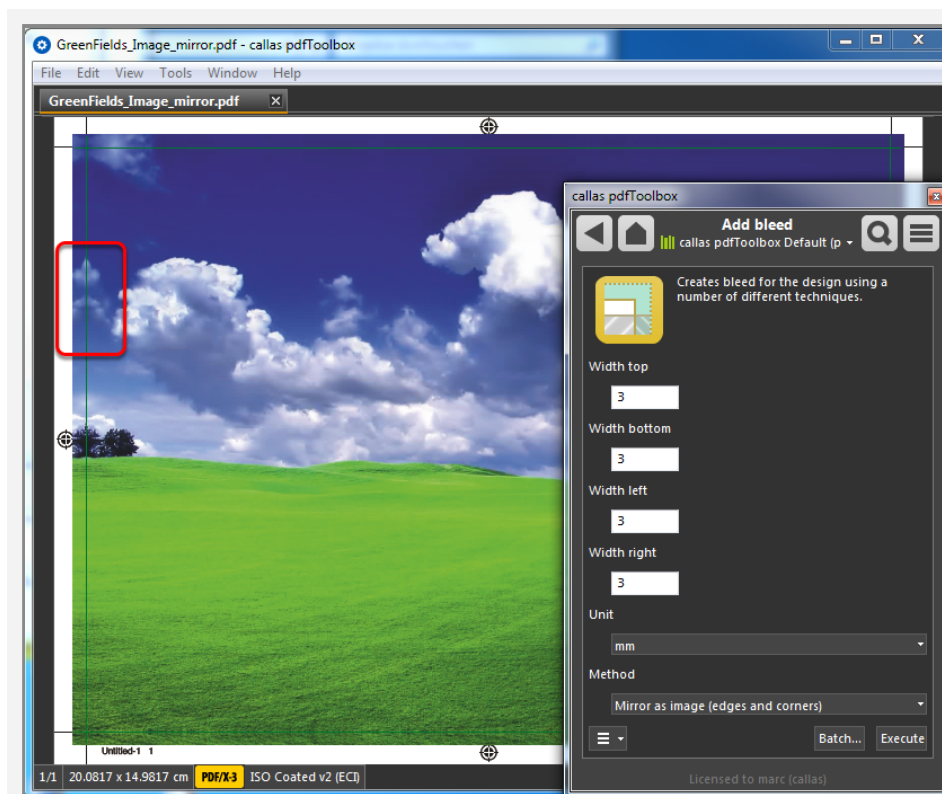
If a PDF has been created without bleed, or if the bleed is not sufficient, generating bleed out of page content is needed to avoid register problems.

To solve this task, pdfToolbox offers a variety of methods to add bleed from page content:

- Mirror content as image
- Repeat last pixel as image
- Mirror page objects

It is possible to define the width of the bleed per edge. The different methods can even become combined when using this feature as a Fixup.

## Mirror content as image

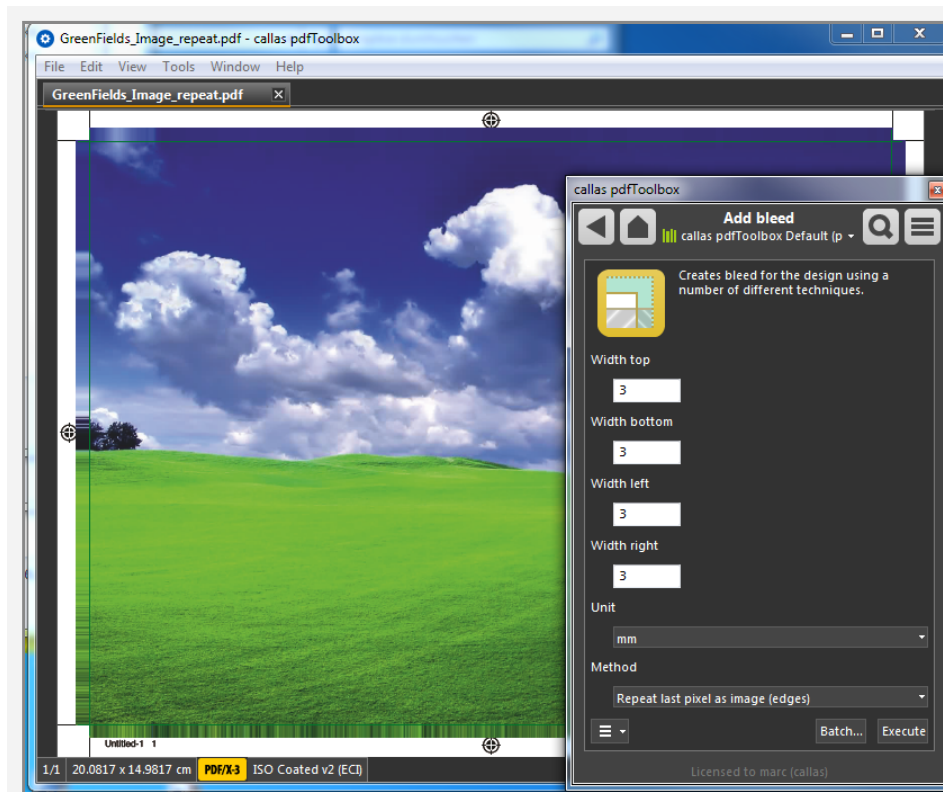


Using the "Mirror as image" option will create an image for each edge (and if selected: corner) at the TrimBox from the

content. This image will be placed mirrored outside of the TrimBox.

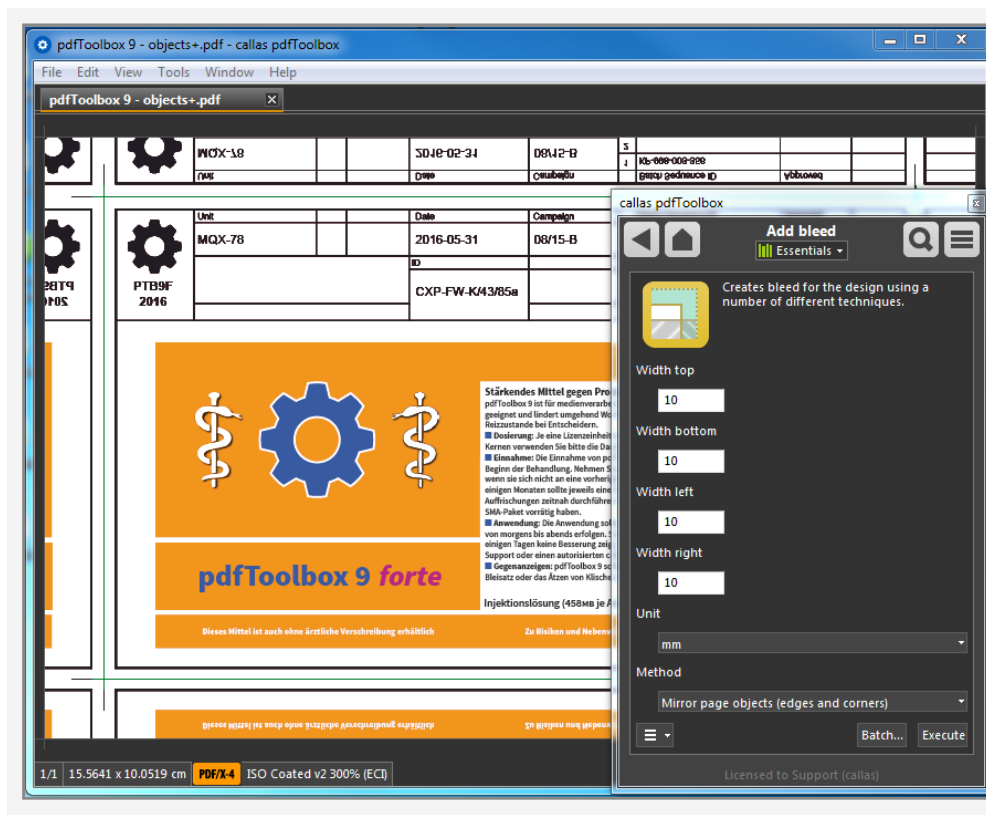
The red rectangle shows a good example of the effect of this method.

## Repeat last pixel as an image



The "Repeat last pixel as an image" method is rendering a thin strip inside the TrimBox and creates a wider image out of this last visible strip. This will result in an image, which "smears" out the last used color inside of the TrimBox outside as the bleed.

## Mirror page objects



Yet another approach is to use all of the page objects, duplicate them (four times), flip them (two of them horizontally, the other two vertically), and then append them one by one to each of the four sides of the page. This will result in bleed with the same rendering accuracy and rendering quality as the page content itself. Different from what one may expect, the impact on file size is minimal (the page objects are not actually duplicated, but instead four references are created). The only real downside is that both a PDF viewer as well as a PDF output system will have to process – to a certain degree – five times as much page content data. This can lead to longer processing times during output, especially on systems with relatively small amounts of working memory.

# Variables and JavaScript: JavaScript

# Taking variables to the next level

Variables have originally been introduced in 2009 in pdfToolbox 4, and have proven to be a very flexible and powerful instrument to develop efficient workflows. Based on extensive feedback from our customers and OEM partners, we have extended the way variables work in pdfToolbox in several ways:

- More aspects of profiles, checks and fixups can be handled through variables; for example, it is now possible to turn checks on or off, change their severity, or to use variables for check boxes and pop-up menus
- Variables are now self contained data objects; this is especially useful where the same variable is used in more than one place; in the past, a pdfToolbox user had to ensure that the same variable used in more than one place was using the exact same configuration string.
- Variables can now also be used as a step in a Process Plan, such that the execution of the following steps can depend on the evaluation of the variable in this step.
- Beyond being a kind of an advanced placeholder with predefined default values, variables can now also be defined in the form of a JavaScript; this implies the possibility to derive the value for a variable from other variables, or from the metadata or filename of the current PDF, or from result data from a previous preflight check.
- Where JavaScript is used, internal variables can be defined and used, without ever confronting a pdfToolbox Desktop user with it.
- In the context of a pdfToolbox Profile, it is possible to include a JavaScript that could for example do preparatory calculations, or determine the value of other variables depending on a document's metadata, filename, or other information.

## The concept of "variables"

Variables as used in pdfToolbox 9 are small information objects that come in two flavors:

- simple value variables
- script based variables

Each data object for a variable has four properties:

- a key (for use when configuring values in Process Plans, Profiles, Checks or Fixups, and for working with variables in JavaScript)
- a label (for use in the user interface, for example in the "Ask at runtime" dialog)
- a value (either, in the case of a simple variable, a default value to be used unless a different value is provided at runtime, or a JavaScript that once evaluated will return the applicable value)
- an internal unique ID (not displayed in the user interface, but can be retrieved using JavaScript)

A variable can be used in almost any context where something can be configured in any of the following:

- Process Plans
- Steps in Process Plans
- Profiles
- Checks
- Fixups
- Checks used as filters in Fixups

The places where variables can be used are for example:

- name and description fields
- fields of type check box, popup, or input fields for text or numbers
- severity for Checks and Fixups inside a Profile
- ON/OFF switch for Checks and Fixups inside a Profile

Variables make it possible to determine some information that is useful when executing a Process Plan, Profile, Check or Fixup at the time of execution, instead of having to predefine such information beforehand. A simple example would be a Check that analyses the minimally required resolution of images. Sometimes 300 ppi are needed (for high quality

printing, in other cases 72 ppi or 96 ppi could be sufficient (when sharing a PDF via email). While it is possible to configure three separate checks for 300 ppi, 96 ppi and 72 ppi, it is much more elegant to only define a single Check, where a place holder is used which is then filled when executing the Check. Not only is just one Check needed instead of three, it is also absolutely easy to use the same Check for altogether different required minimal resolutions, like 144 ppi or 450 ppi or any other value.

Thus, the major benefit of variables is the option to postpone the decision, which values to use for processing PDFs, to the moment when processing is started. This includes the possibility to choose different values each time. Furthermore, the introduction of JavaScript makes it possible to derive further information based on information provided at runtime or based on information through metadata, including the option to use relative complex calculations.

## Variables in the desktop, server, and command line SDK versions of pdfToolbox 9

In principle not much has changed here in comparison to pdfToolbox versions before version 9. In the desktop version of pdfToolbox 9 (whether Acrobat plug-in or standalone), when running a Process Plan, Profile, Check or Fixup that contains one or more variables for which input is needed, the "Ask at runtime" dialog will open and will request that user enters values as desired (or leaves the pre-populated default values as they are). For the server and command line versions, the values have to be provided as command line parameters or by means of a configuration file.

What has changed - mostly due to the extended capabilities - are the following:

- Values entered by the user have to be suitable for the type of field for which they are to be used; for example, it is now impossible to provide arbitrary text when the expected value is a number.
- Additional information is provided to the user in the "Ask at runtime" dialog in case there is a problem with the value(s) entered.
- Using a special option in the "Ask at runtime" dialog, it is possible to analyze the way variables are collected or cal-

culated; this not meant to be used by the typical user, but rather by the person in charge of developing advanced uses of variables – which could become quite complex in Profiles that use variables in many places in Checks and Fixups.

## How powerful is the JavaScript engine in pdfToolbox 9?

The JavaScript engine in pdfToolbox 9 is based on Google's V8 JavaScript engine (see <https://developers.google.com/v8/> for more information). Those who work with JavaScript in browsers will know, that only the sky is the limit there. One could carry very extreme tasks using JavaScript inside a browser, including reaching out to all kinds of services and data sources over the internet.

The way JavaScript functionality is provided inside pdfToolbox 9 takes a slightly different approach:

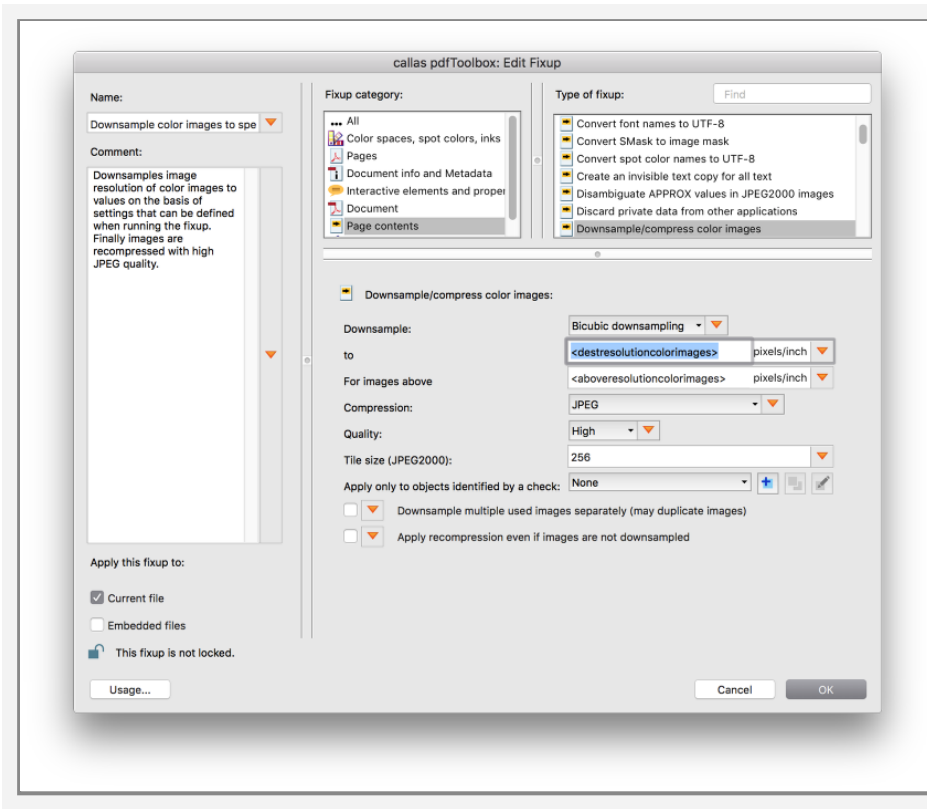
- pdfToolbox 9 (through the underlying V8 engine) supports the complete set of JavaScript features as defined in ECMAScript is specified in ECMA-262, 5th edition (see <http://www.ecma-international.org/publications/standards/Ecma-262-arch.htm>)
- several pdfToolbox specific internal data objects, in order to allow for access to document metadata, filename, and so on; and to store data in a place shared across Process Plans, Profiles, Checks and Fixups through one execution context.
- The pdfToolbox 9 JavaScript engine comes with a powerful runtime evaluation architecture, that ensures that variables relying on each other do actually work consistently without the user having to meticulously take care of such dependencies.
- pdfToolbox 9 does not offer any access to outside data (except where provided through pdfToolbox specific internal data objects), whether to the local file system, or to web services or data accessible "over the web"
- pdfToolbox also does not offer the possibility to reference JavaScript files, as is often used to provide JavaScript libraries; where library-like functionality is needed, suitable JavaScript code must be included in the JavaScript



snippet associated with a variable, or with the Profile JavaScript.

# Variables and JavaScript: Variables in general

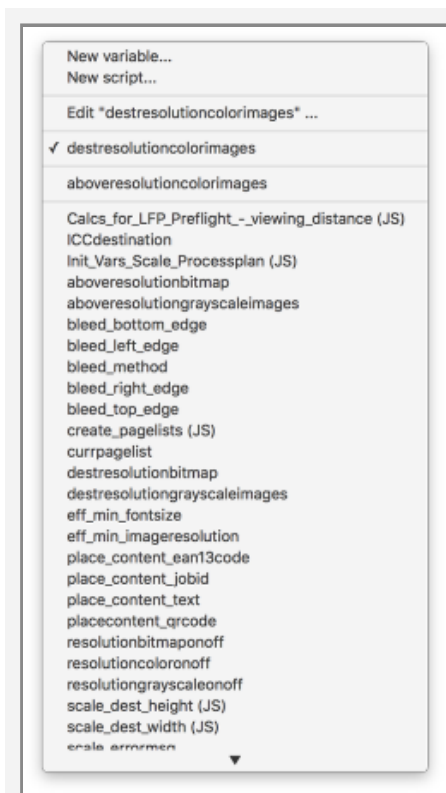
## Variables can be assigned to everything



Variables can be defined in several places throughout the Profile/Checks/Fixups editor. Variables may be assigned to virtually every control including the severity for a check:

- Text input fields
- Checkboxes
- Pop Up fields
- Severities
- On/Off switch in order to enable/disable Checks or Fixups in a Profile

## Assign a variable



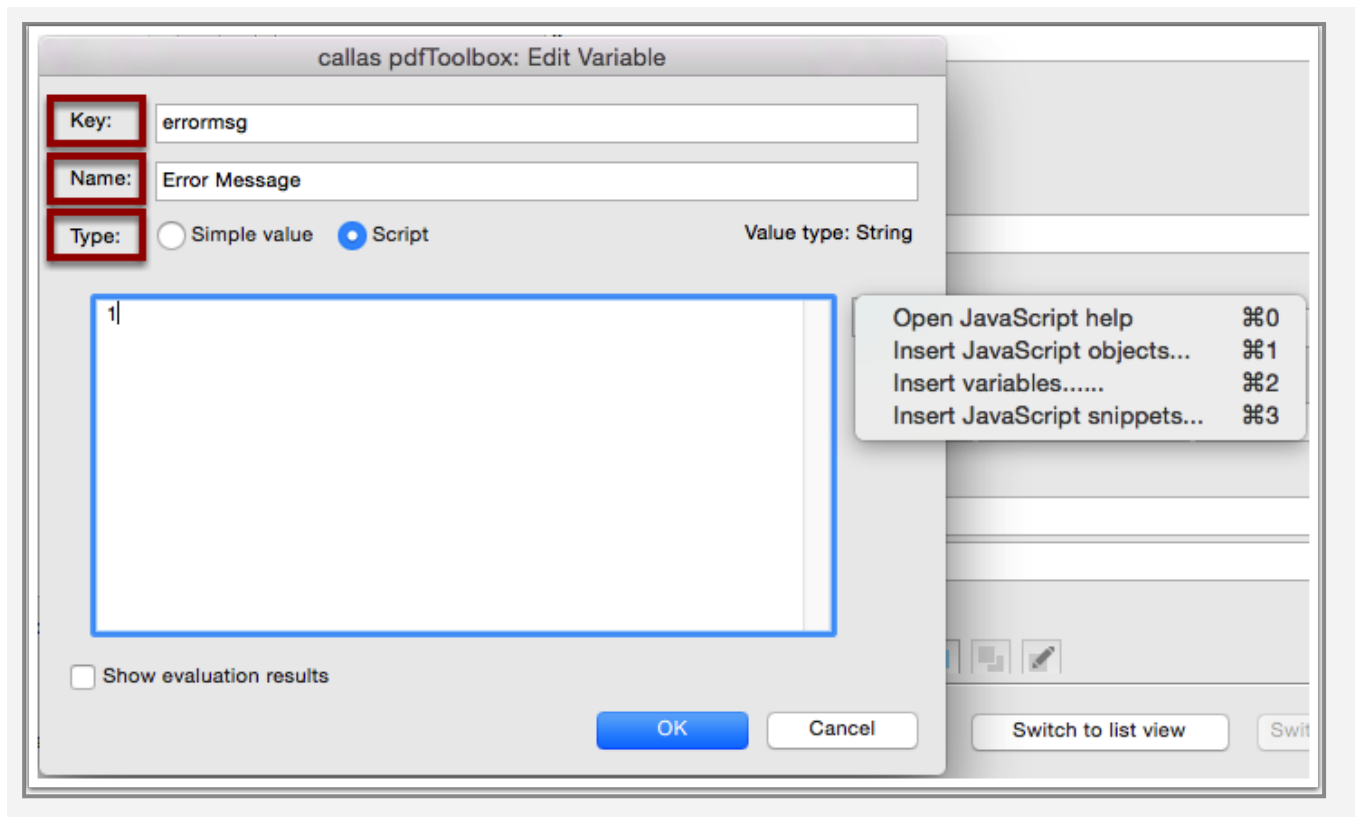
When you click on a variable icon in the pdfToolbox Profiles/Checks/Fixups editor, you will see a list of all variables that are present in the system. Variables that are used in the current context (e.g. the current Profile) appear at the top. You may pick any of the existing variables, create a new one or edit one that is already assigned.

Deleting a variable is currently only possible in the Library Manager and only if the variable is not used. Whether or not a variable is used can also be seen there.

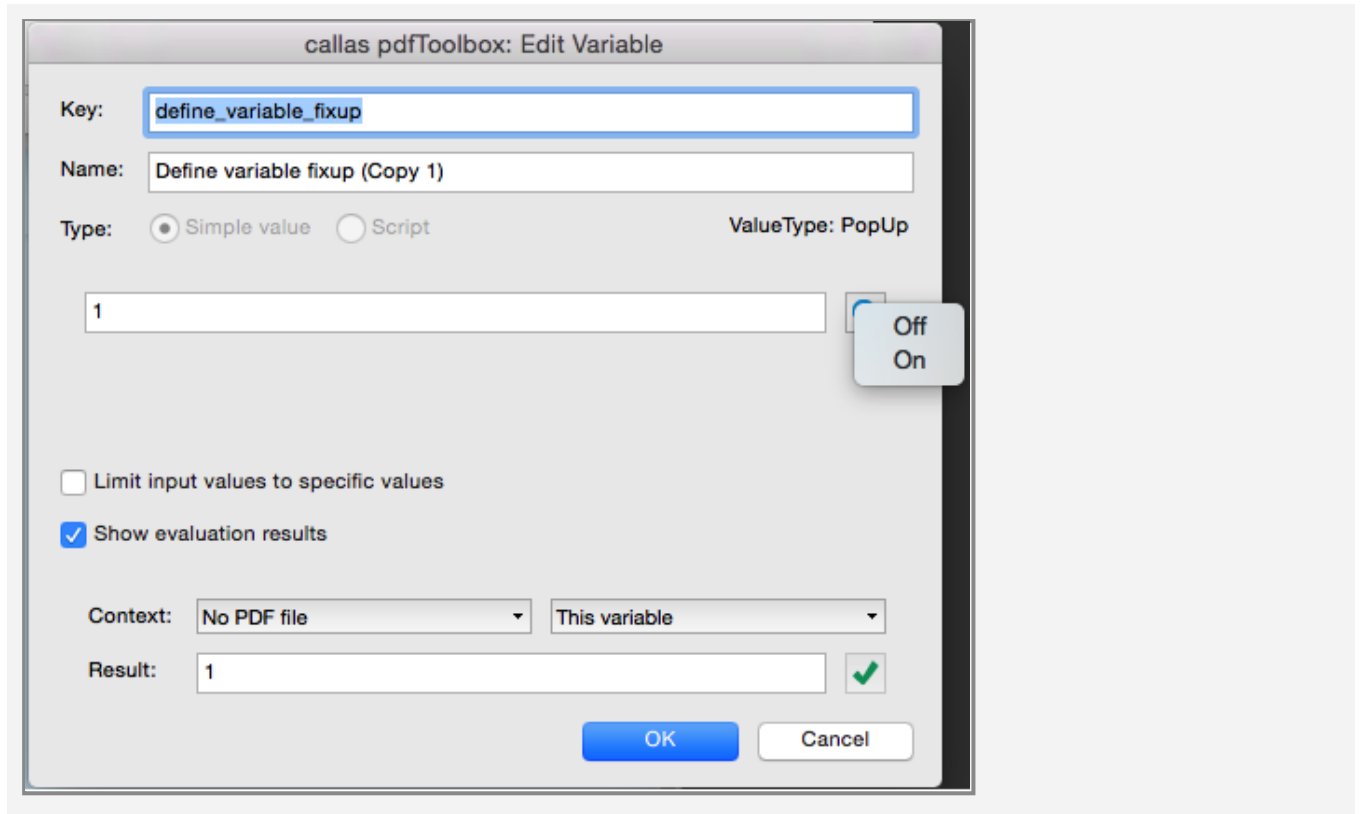
## Variable Editor: Creating a new variable

You have to define a key for the variable, an optional Name that will be used in a pdfToolbox Desktop dialogue and the default value. If you use a Simple value Variable these are the most important ones, but you may in addition define Constraints (this is explained in a later step in this chapter). For a Script Variable you may use the help that is available via the

info button, the JavaScript help has the same content as the respective chapter in this manual.



**For Checkboxes and Pop ups you can use the info button to pick one of the possible default values**



## Constraints

callas pdfToolbox: Edit Variable

Key: destresolutioncolorimages

Label: Destination resolution of color images

Type: ☒ Simple value ☐ Script

Value type: Integer

360

☒ Limit input values to specific values

☐ Allow Browse button if feasible

☐ List ☒ Range

Values: 300  
400

☒ Show evaluation results

Context: PDFX4 blendspace DeviceCMYK with DefaultCMYK.pdf

Downsample color images to specified value (high JPEG)

Result: 360

OK Cancel

Constraints can be defined by using the "Limit input values to specific values" checkbox in the variable editor. In this context the expected value type of the variable is important which is indicated above of input field for the default value.

You may in addition specify:

- whether a Browse button shows up when executed in pdfToolbox Desktop that allows a user to pick a file from the system (e.g. to load an ICC profile)
- whether the entries in Values are used as list, in which case a Pop up would show up in the dialogue of pdfToolbox Desktop or as a range. In the latter case invalid values that are out of range will be indicated by a red cross.

## Constraints - Range specifics

The screenshot shows the 'callas pdfToolbox: Edit Variable' dialog box. The 'Key' field contains 'resolution\_\_ppi' and the 'Name' field contains 'Resolution (ppi)'. The 'Type' is set to 'Simple value' and the 'Value type' is 'Float'. The 'Limit input values to specific values' checkbox is checked. The 'Allow Browse button if feasible' checkbox is unchecked. The 'Range' radio button is selected. The 'Values' list contains '100', '300', and '500'. The 'Show evaluation results' checkbox is unchecked. The 'OK' and 'Cancel' buttons are at the bottom right. Red arrows point from labels to specific fields: 'Constraints' points to the 'Limit input values to specific values' checkbox, 'Value range' points to the 'Range' radio button, and 'Value list' points to the 'Values' list.

callas pdfToolbox: Edit Variable

Key: resolution\_\_ppi

Name: Resolution (ppi):

Type: ☒ Simple value ☐ Script Value type: Float

☒ Limit input values to specific values

☐ Allow Browse button if feasible

☐ List ☒ Range

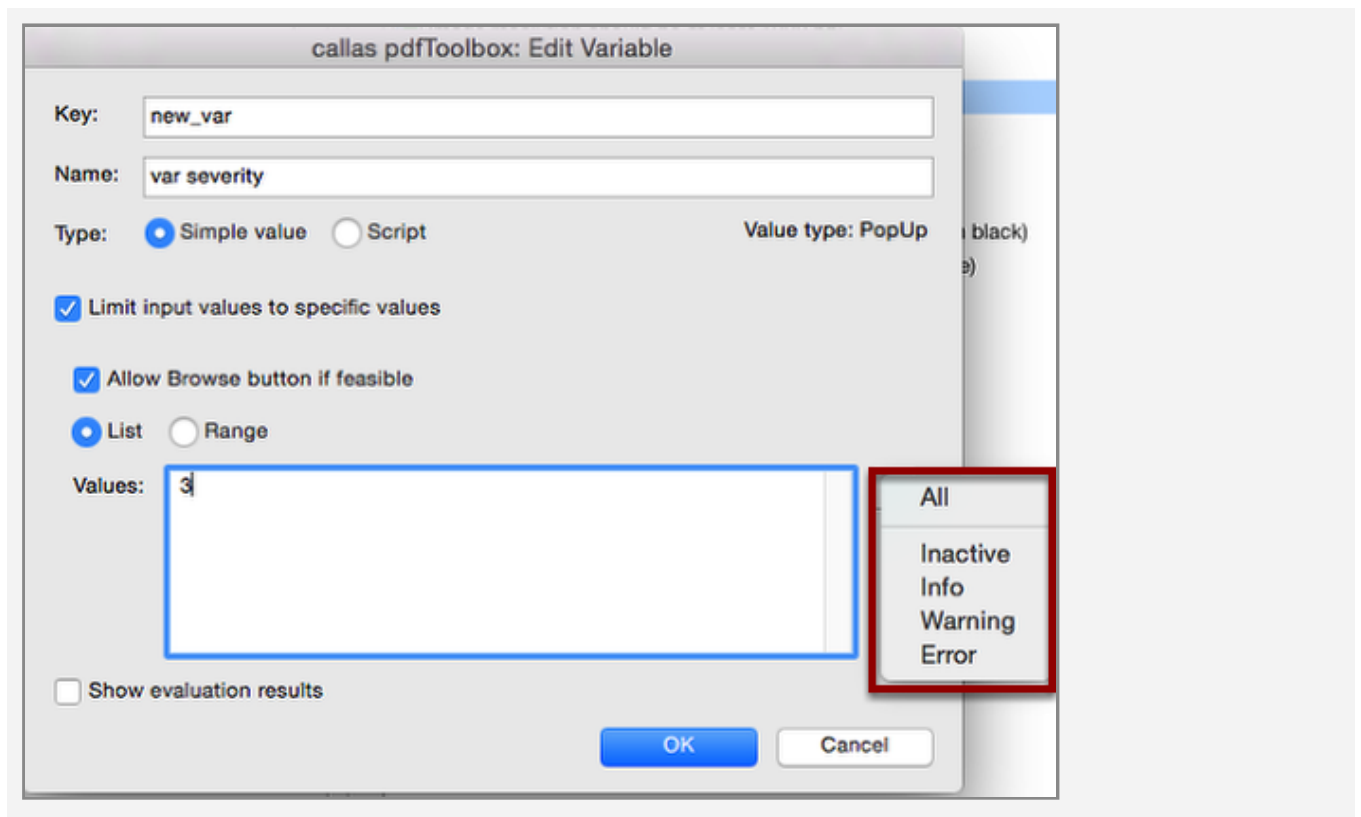
Values: 100  
300  
500

☐ Show evaluation results

OK Cancel

When you are using the Range option, two values will define a range and a single value defines a single allowed value. In this example 100-300 and 500 are permitted values (which usually does not make sense).

## Constraints for Pop up fields



You may again use the info button in order to pick possible values for the Pop up. In this example this is done for a severity, but it works in the same way for other Pop up fields.

## Profile Script Scope

It is possible to set a value for a variable in other script variables by means of Javascript. And this can and will usually be done on Profile level. In turn it only makes sense to use Script values at this place. Please go to the pdfToolbox Javascript in variables documentation for further information.



This screenshot shows the configuration window for a profile named "Scale to page size from first 7 characters in file name v9.0". The window has a "General" tab and includes the following fields:

- Locked:** A lock icon and the text "Unlocked".
- Name:** "Scale to page size from first 7 characters in file name v9.0".
- Purpose:** "If a PDF file name starts with three numbers followed by any character followed by another three numbers, the numbers are used as the target size of the PDFs pages which are then scaled to that size. This profile is not intended for production use but should rather show how such calculations can take place."
- Author:** "callas software".
- Email:** "support@callassoftware.com".
- Group:** "Custom profiles".
- Script:** "{Init\_Vars\_Scale\_Processplan}".

## Variables in Processplans

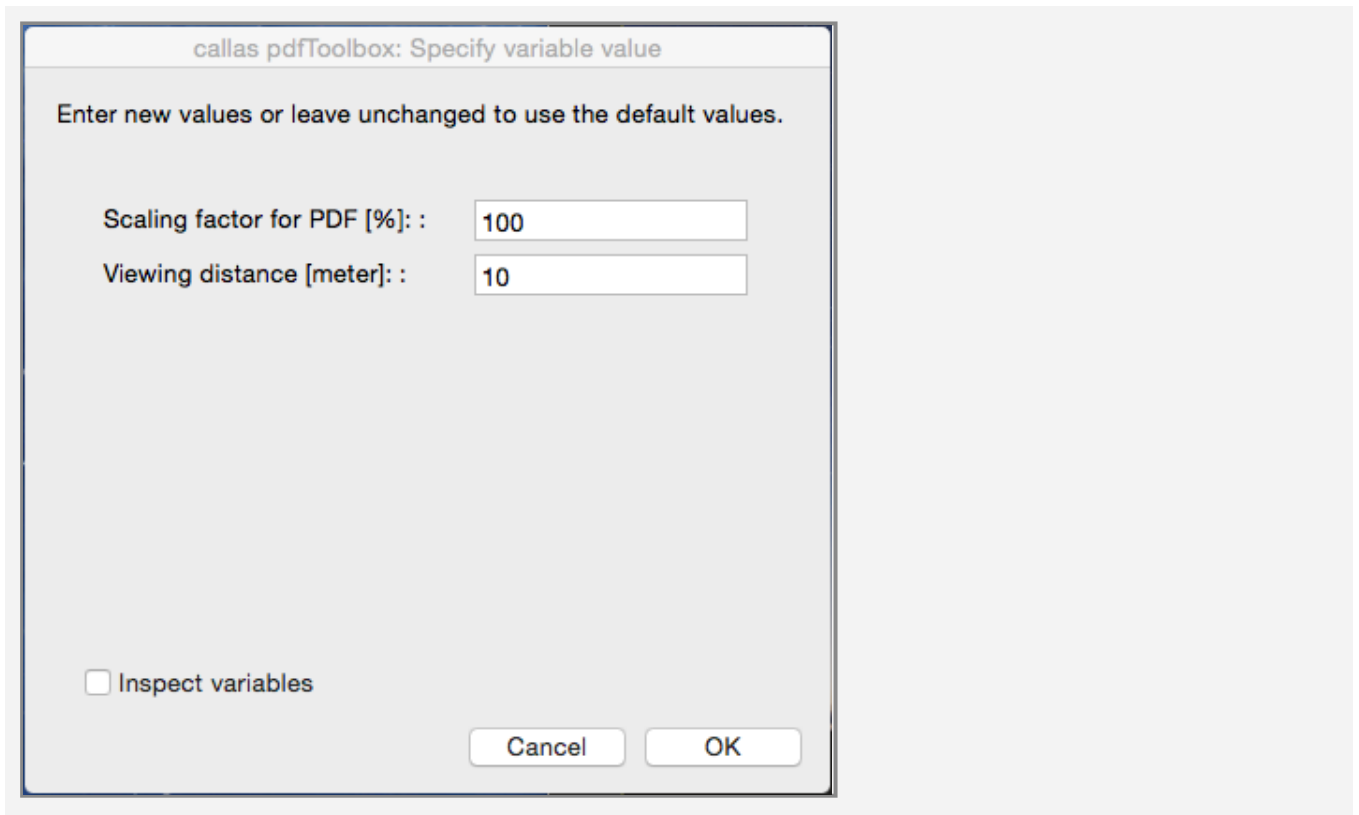
This screenshot shows the "callas pdfToolbox: Edit Process Plan" dialog. It displays a sequence of steps in a process plan:

- Profile:** Name: "Scale to page size from first 7 characters in file name v9.0". It includes options for "On Error", "On Warning", "On Info", and "On Success", all set to "Continue with next step".
- Fixup:** Name: "Place error messages at the center of each page v9.0". It includes options for "On Success" and "On Failure", both set to "Continue with next step".
- Variable:** Name: "<barcode>". It includes options for "On Success" and "On Failure", both set to "Continue with next step".

The "Variable" step is highlighted with a red box. The dialog also includes a "Cancel" button and an "OK" button at the bottom right.

It is possible to define a variable as a step in a Process Plan. This will work similar to a Variable on Profile level in does on-ly makes sense and work for Script variables.

## The "Ask at runtime" in pdfToolbox Desktop



In pdfToolbox Desktop a dialogue shows up, when a Profile/Check/Fixup is executed that has editable variables, i.e. variables that are not calculated by means of Scripts. If you want to see the dialogue for a Profile/Check/Fixup that only has calculated variables you will have to add one additional variable that is not calculated.

Script variables are hidden as long as no evaluation errors had occurred. If an evaluation error has occurred the OK button is disabled. Details can be displayed by clicking the "Inspect variable" checkbox. It allows you to analyze the structure of all variables in the given context.

# Variables using JavaScript: Overview

## Where can JavaScript variables be used

Script variables can be used wherever Simple variables can be used:

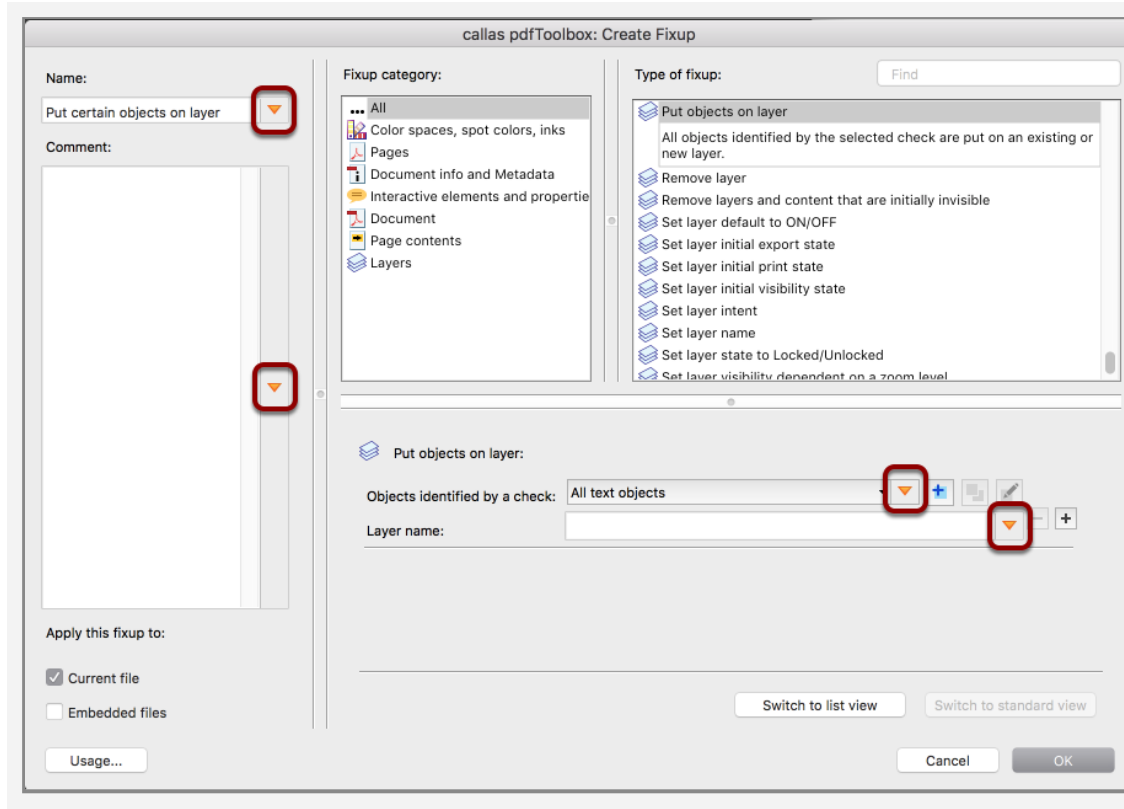
- In Checks or Fixups for text input fields, pop ups, check-boxes
- Severities of checks
- On/Off variables for Checks and Fixups

It in addition is possible to use Script variables (but not Simple variables):

- in a Profile as Profile script
- as a Process Plan step

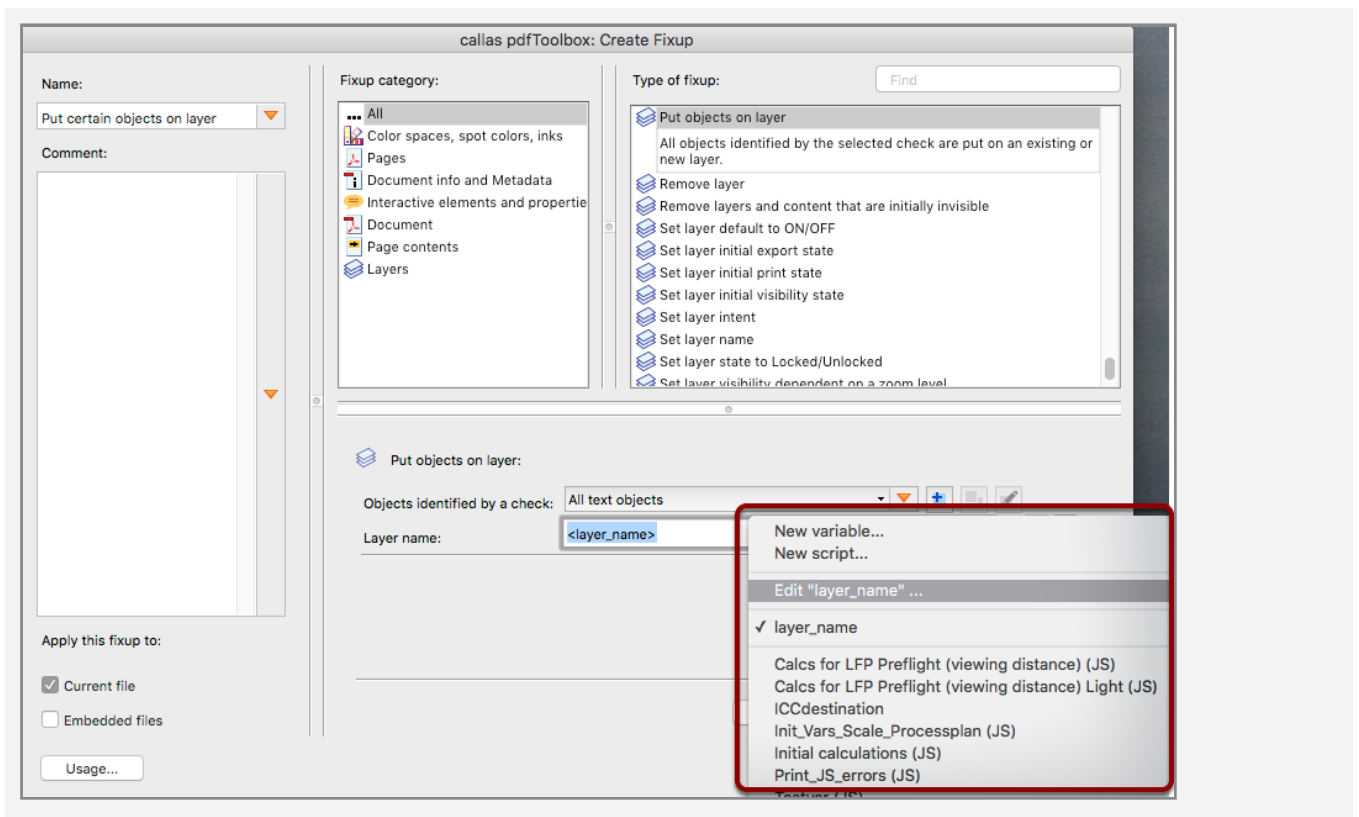
In all places where Simple or Script variables can be used, the variable editor allows you to switch between both by means of a radio button. After a variable has been saved as Simple variable it is possible at any time to convert it into a Script variable here. However, it is not possible to convert a Script variable into a Simple variable. The reason is that this could potentially lead to problems when the same variable would be used in a place where only a Script variable is allowed.

## Assigning a variable to a pdfToolbox Desktop control



Wherever you see the variable icon in pdfToolbox Desktop you can click on it in order to assign a variable from a list of all variable keys that are defined in the current Library to the respective control.

## Creating or modifying a JavaScript variable



You can create a new variable (either as Simple or as Script variable) and assign it; or you open the variable editor in order to modify a variable that has already been assigned.

The list of variables in the pop up shows those variables that are already used in the current context (Profile, Check, Fixup) first and then all variables in the current Library. Script variable keys are followed by "(JS)" to indicate that these are JavaScript variables. After assigning a variable to a pdfToolbox Desktop control the variable key is displayed in the respective field (for text input fields or pop ups) or next to it (for checkboxes). Simple variables are displayed as <Simple variable>, Script variables as {Script variable}.

In order to un-assign a variable from a control you simply have to remove it from a text input field, to pick any other value in a pop up or to check/uncheck a checkbox.

## Creating or modifying a JavaScript variable: Important differences to pdfToolbox versions earlier than version 9

In pdfToolbox versions earlier than version 9 it was possible to copy a variable out of a pdfToolbox Desktop control and insert it into another control in order to assign it to both controls. This is not possible in pdfToolbox 9. You have to select the variable key from the variable pop up in the second or any further control.

From pdfToolbox 9 on it is no more possible to make two variable occurrences using the same value simply by using the same variable name ("key"). Variables are only then the same if any additional occurrence is selected from the variable pop up in pdfToolbox Desktop. Otherwise two variables using the same key would be present which would at least be confusing when evaluated.

But: It would be difficult to resolve such conflicts when a Profile is imported as kfpX file, if the imported Profile uses the same variable key as a variable that is already present in the current Library. Therefore in such cases internally a variable merge process takes place that merges all variables that are defined in the very same way (key, default value and label) into a single variable.

## Defining a variable in a script

If you want to define a variable in a script that is not used in any pdfToolbox Desktop control you may do so by writing at the top of your script:

```
app.requires("myvar")
```

myvar will then be created and show up in the Ask at Runtime dialogue or in --listvariables on command line. If you also want to set a default value and a display name (label) you can write:

```
app.requires("myvar",100,"Input a value for myvar")
```

## Setting the value for a Script variable in it's own script

A Script variable is populated with the value that is the result of the last statement in the script. So, if a Script variable would end with a statement like "pdfToolbox" it would have this string as its value, independent from what code has been executed beforehand. A return statement as in a JavaScript function is neither required nor would it have any effect.

## Setting the value for another Script variable with app.vars

It is possible to set a value for another variable in JavaScript code by means of an `app.vars.<variable key>` statement. The `app.vars` object is a `pdfToolbox` object that is available throughout the context (Process Plan, Profile, single Check, single Fixup) in which processing takes place. It allows you to store and retrieve variables within this context:

```
app.vars.myvar = "pdfToolbox";
```

or:

```
localvar = app.vars.myvar;
```

are valid statements. The first statement would create the variable "myvar" if not already present in `app.vars`. You may e.g. use `app.vars` to set a value for a variable on Profile level, which is then used in a Fixup in the Profile.

In order to set a value for a Simple variable you can use `app.vars.<variable key>`. A list of all variables that are present in the current Library can be displayed in the Script editor by using [`<command>-2`].

Setting a value for a variable via JavaScript code should only take place on Profile level or as a "Variable" step of a Process Plan. The reason is, that it is not defined in which order scripts on "lower levels" (Checks, Fixups, Severities, On/Off) are executed during runtime and therefore the result of e.g. one Fixup modifying a variable in another Fixup is undefined.

You also have access to variables in `app.vars` in the "Place content on page" fixup. This fixup allows you to place content defined in HTML templates that may internally use JavaScript. If you want to read, write or create a variable in `app.vars` from that JavaScript you have access to all of them in the array

[cals\\_doc\\_info.document.variables](#)

## Using variables that are defined elsewhere

In order to use the value of a variable in JavaScript it can be accessed using the `app.vars` object with the key of the variable as already described above: `app.vars.<variable key>`. If the other variable is defined in a Script variable it has to be defined using `app.vars` there as well. If the other variable is a Simple variable it is always present in the `app.vars` object.

When retrieving variable values from `app.vars` it is important to know that all variables are stored there as strings. With simple value types you will most probably not even notice this, because a string is automatically converted if necessary and possible, e.g. into a number. However, if you are working with more complex variable types like with arrays or objects there will obviously be differences and you might have to work around this limitation.

In complex profiles - actually when a Script variable is used in another Script variable - `app.requires("<variable key>")` has to be defined at the top of the referencing Script, in addition to the actual reference with `app.vars`. This is required in order to make sure that the referenced variable is evaluated before the referencing variable is calculated. So, it is good practice to at the top of each Script, list all variables which are not defined in the Script itself in `app.requires` entries.

## Profile level scripts versus Check/Fixup level scripts

When you "design" a profile with JavaScript based calculations you have to decide whether you want to put the "intelligence" (the calculations) into Fixups and Checks that actually apply things to the PDF or into a Profile level script and set



values for the variables that are then used in Fixups and Checks from there.

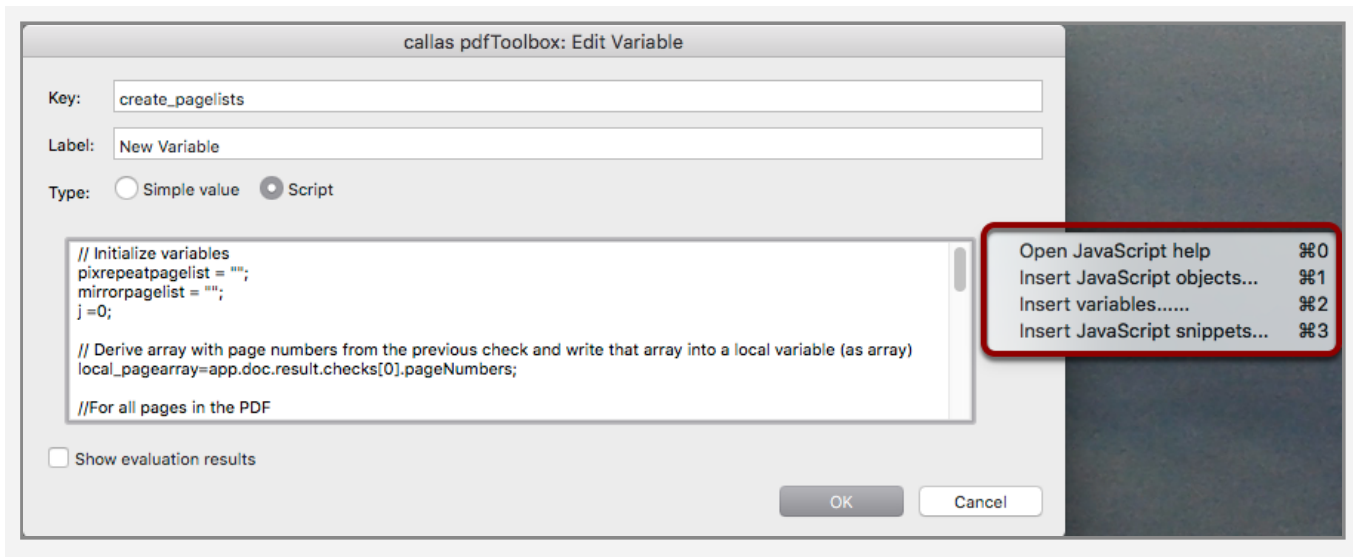
Example: Downsample images in pdfToolbox usually requires to set up three fixups: for color images, grayscale images and for bitmap images. Each of the fixups has two input fields that you may want to make variable: The destination resolution and the minimum resolution for an image to be downsampled. Assume that you want to downsample color and grayscale images to the same resolution. Images should be downsampled if the original image resolution is 1.5 times as high as the destination resolution. Destination resolution for bitmap images should be 3 times as high as for color images, with the same relative minimum resolution (effectively 4.5 times color images' minimum resolution). You may now either make the destination resolution for color images a Simple variable, e.g. "dest\_col\_res" and make any of the other 5 variables a Script variable that uses dest\_col\_res and calculates the actual value. Or you set up a Profile level script, do all the calculations there and put the results into a bunch of Simple variables that you assign to each of the 6 variable input fields. (You will have to use app.vars in order to use variables throughout the Profile and in the second case you would use app.requires to define a variable for the destination image resolution in the Profile script.)

Each of the two approaches has advantages:

- If you put the intelligence into Fixups and Checks it is easier to make it possible to use them as Single Fixups or Checks, independent from the Profile.
- If you put the intelligence into the Profile it is usually easier to see what a profile is actually doing and - even more important - to maintain it in the future.

As a result and a rule of thumb it can be said, that it usually makes sense to put as much intelligence into the Profile level script. The more complex a Profile is, the more important it is to follow this approach.

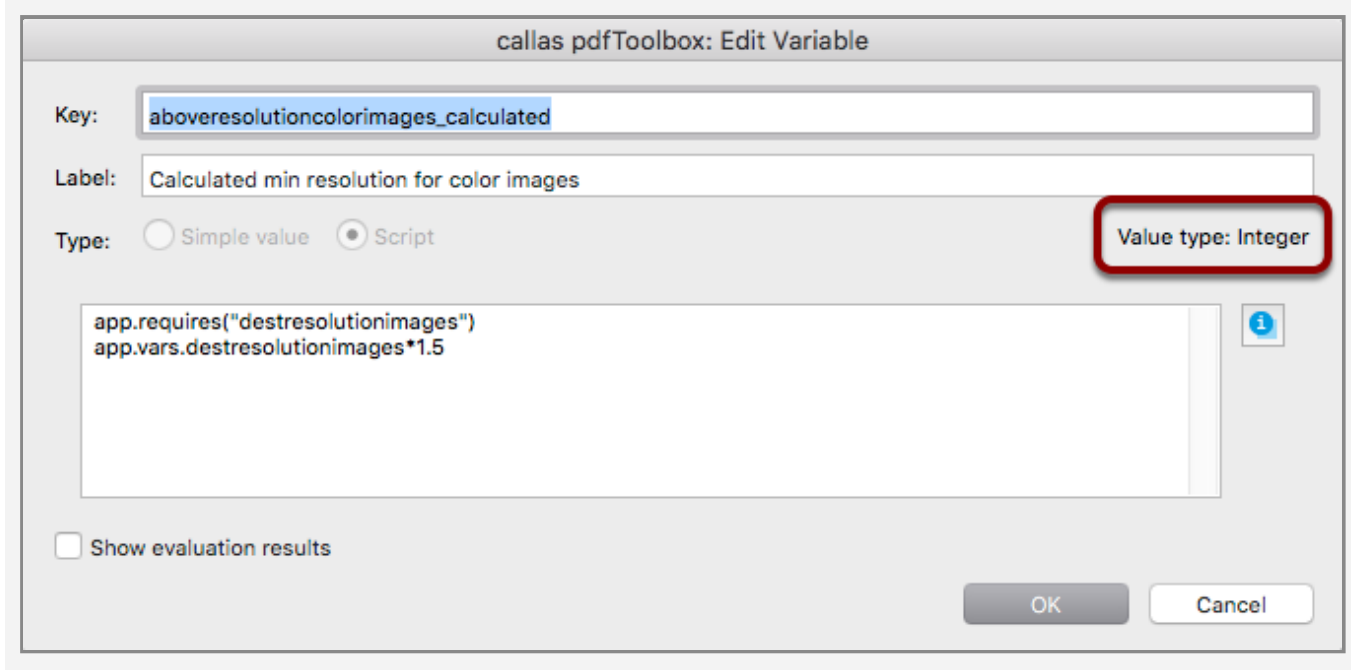
## The Script editor: User interface elements: Help



When the variable editor is switched into Script "mode", you can find help with the info button on the upper right side of the Script input field. You will find more information if you click into the Script input field first. This gives you access to

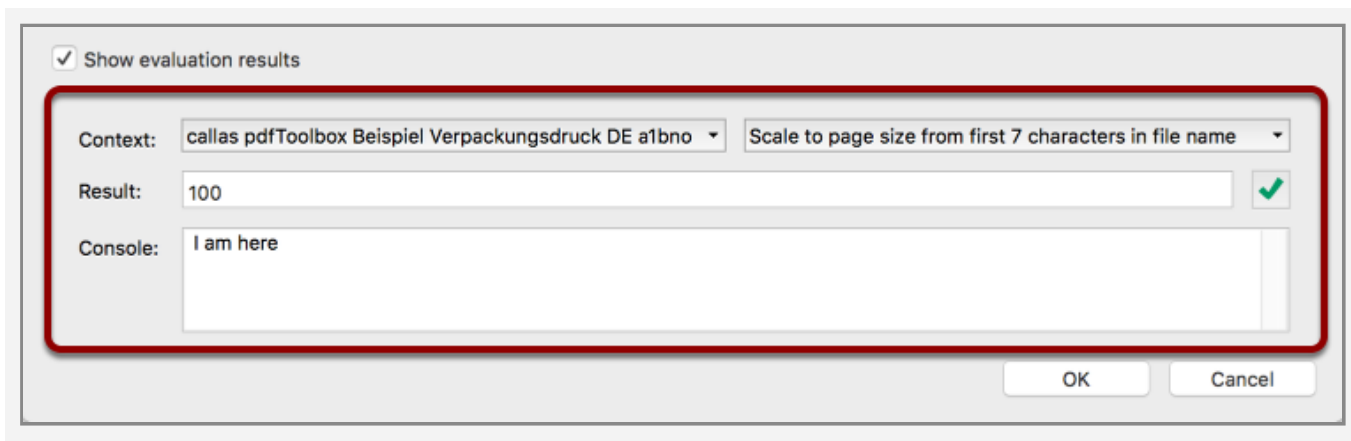
- a general help text (this text) [`<command>-0`],
- a list of all pdfToolbox specific JavaScript objects and methods [`<command>-1`],
- a list of all variables that are present in the current Library [`<command>-2`]
- useful code snippets [`<command>-3`].

## The Script editor: User interface elements: Value type



Above of the info button you see the value type of the pdfToolbox control to which the variable is currently assigned. This information is useful to know what type of result is expected from your script.

## The Script editor: User interface elements: Show evaluation results



Below the text input field you can switch "Show evaluation results" on, which will help you to find out what the result of your JavaScript code is.

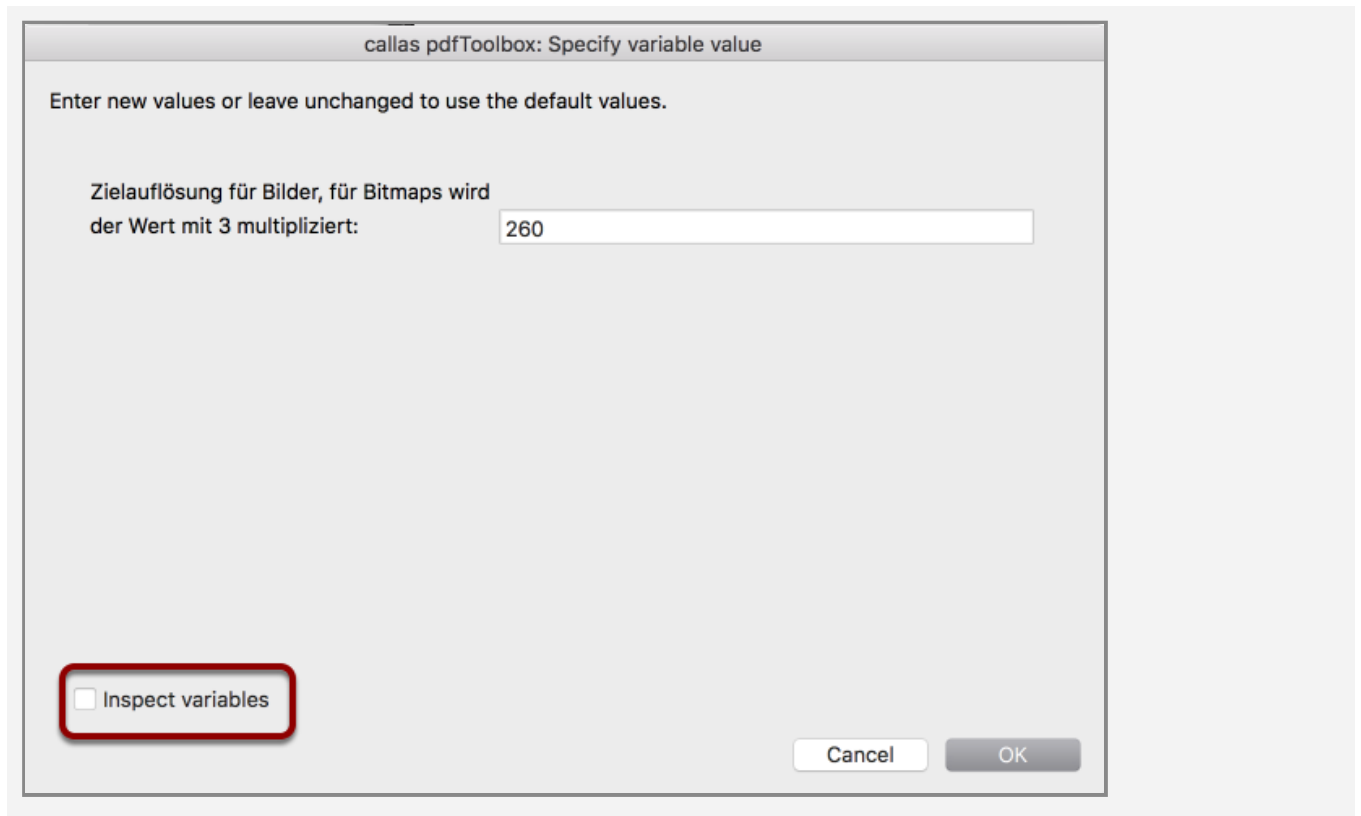
It is important to remember that the result of JavaScript code used in a variable is the result of the last statement. The only exception is if invalid code is used, e.g. if a closing parenthesis is missing, in which case you will see a "Syntax error" with an explanation.

In order to modify evaluation you can simulate how a JavaScript works different when used in a different "Context": You may either load a PDF (it will not open in pdfToolbox) to simulate how your script works on that PDF, e.g. when you are using the PDF path inside of your script. Or you switch between evaluation only for the script (in which case values that are set via other variables in your context are not evaluated) or evaluation within the context. All this information is helpful for debugging your scripts.

A button at the right hand side indicates whether the result of the script works in the current pdfToolbox control. If the result is "pdfToolbox" and you are using the variable for a text input field you will see a green checkmark. However, if the variable is used for an integer number field you will see a red error cross. When you click on it you will read: ""pdfToolbox" cannot be converted to integer".

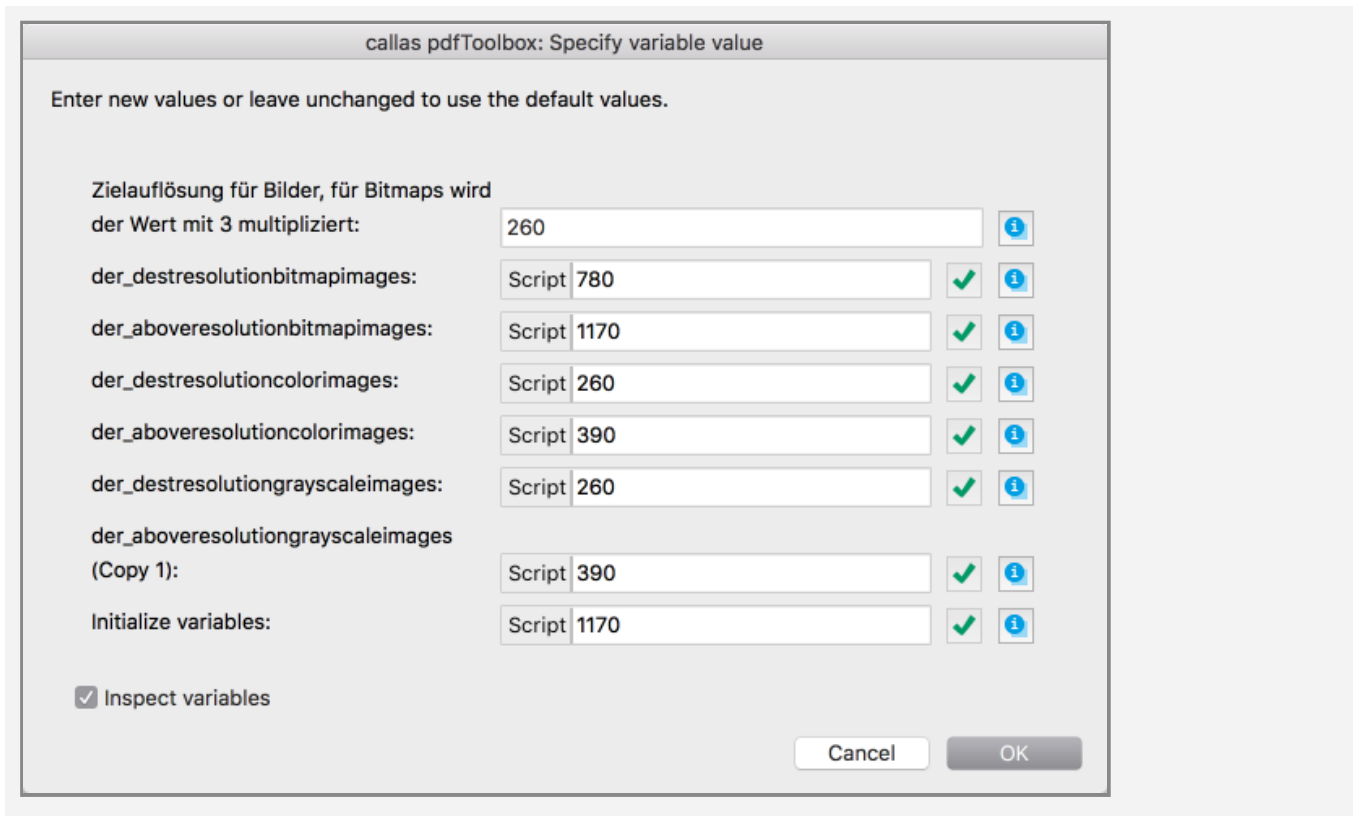
The console window displays information that the JavaScript sends to it. It can be used with `console.log`.

## Inspecting the variable structure in the Ask at Runtime dialogue in pdfToolbox Desktop: Activating the "debug view"



When you run a Process Plan, Profile, Check or Fixup in pdfToolbox Desktop that uses variables you will see the "Ask at Runtime" that allows for updating variable values. A checkbox at the bottom of the dialogue allows the user to "Inspect variables".

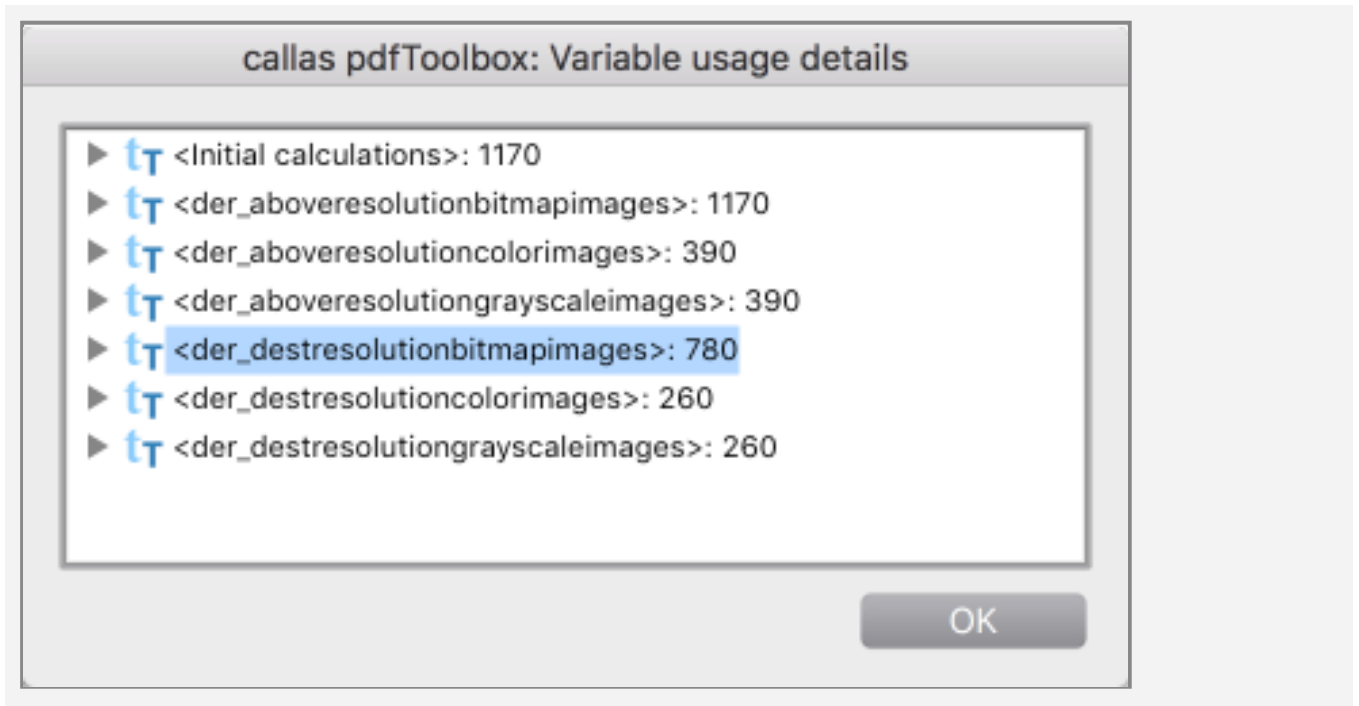
## Inspecting the variable structure in the Ask at Runtime dialogue in pdfToolbox Desktop: The "debug view"



If activated all calculated variables are displayed, not only those ones that allow for user input. A "Script" indicator shows values that cannot be modified in this dialogue because they are already calculated in the scripts. A button behind the Script variable fields indicates whether there is a type conflict, e.g. if the variable is used for a number field but the value is a string that cannot be converted to a number. In that case you will see a red cross. You may click on any of those red cross buttons in order to see details for the problem.

The Ask at Runtime dialogue will only appear if a Profile/Check/Fixup has at least one variable that does not already have a value. If you want to enable the debug view in a Profile/Check/Fixup in which all variables are set by means of scripts, you will have to add at least one additional variable, e.g. a Simple variable for that purpose.

## Inspecting the variable structure in the Ask at Runtime dialogue in pdfToolbox Desktop: The "debug view" info button



Next to each variable you will see an info button. Clicking on this button allows for accessing information that is useful for debugging purposes. A list of all variables that are defined in the current context is displayed. (The content of the info window is actually the same for each of the info buttons, the only difference is that the control for the respective variable is opened by default.) Each of these variables is followed by the result that has been calculated for the respective variable. If you open the triangle for a variable you see an entry "Variables" that shows details about how that variable has been defined. Below are all contexts listed in which the respective variable is used.

## pdfToolbox specific JavaScript objects and methods

pdfToolbox provides a number of objects and methods that can be accessed in JavaScript variables. This includes infor-

mation about the PDF, like its name or file path, the metadata in the PDF and even results from a previous Check or Profile. You can display a full list of all objects and methods by using [`<command>-1`] when in the Script editor, select any of the entries and insert them into your script.

A complete list of these objects and methods can also be found in the chapter "Variables using JavaScript: pdfToolbox objects and methods".



# Variables using JavaScript: pdfToolbox objects and methods

This article provides an overview of all JavaScript objects and methods that are specific to pdfToolbox Variables. It is the same information that can be displayed in the Script editor of pdfToolbox.

**app** Returns the global application object.

**app.requires(key)** Defines a variable key with default value 0 that is required by the current script.

Example:

```
app.requires("myvar")
```

**app.requires(key,value)** Defines a variable key and its default value that is required by the current script.

Example:

```
app.requires("myvar",10)
```

**app.requires(key,value,label)** Defines a variable key and its default value and a display name (label) that is required by the current script.

Example:

```
app.requires("myvar",10,"Input value for myvar")
```

**app.name** Returns the application name.

**app.version** Returns the application version string.

**app.vars** Returns the var objects containing all variables defined in the current context.

**app.vars.varkey** Returns the value of the variable "varkey" if that exists in app.vars.

Example:

```
app.vars.varname
```

**app.doc** Returns the doc object for the current PDF document or 'undefined' if no PDF is open.

**app.doc.info** Returns the docinfo object containing all document info entries of the current PDF document.

**app.doc.path** Returns the full platform dependent file path of the current pdf document.

**app.doc.documentFileName** Returns the file name of the current PDF document.

**app.doc.numPages** Returns the number of pages of the current PDF document.

**app.doc.getPageBox()** Returns an array containing the left, top, right and bottom coordinates of the TrimBox of the first page in pt.

**app.doc.getPageBox(pageBox)** Returns an array containing the left, top, right and bottom coordinates of the specified page box of the first page in pt. 'pageBox' must be one of "Art", "Bleed", "Crop", "Trim" and Media.

Example:

```
app.doc.getPageBox(Trim)
```

**app.doc.getPageBox(pageBox,pageNumber)** Returns an array containing the left, top, right and bottom coordinates of the specified page box of the specified page in pt. 'pageBox' must be one of Art, Bleed, Crop, Trim and Media.

Example:

```
app.doc.getPageBox(Trim,0)
```

**app.doc.getPageBox(pageBox,pageNumber,precision)** Returns an array containing the left, top, right and bottom coordinates of the specified page box of the specified page with the given precision in pt. 'pageBox' must be one of Art, Bleed, Crop, Trim and Media.

Example:

```
app.doc.getPageBox(Trim,0,2)
```

**app.doc.getPageRotation()** Returns the page rotation of the first page.

**app.doc.getPageRotation(pageNumber)** Returns the page rotation of the specified page.

Example:

```
app.doc.getPageRotation(0)
```

**app.doc.pages** Returns an array with page objects for the current PDF document.

**app.doc.pages[i].inks** Returns an array of inks used by on the page.

Example:

```
app.doc.pages[0].inks
```

**app.doc.pages[i].inks[j].name** Returns the name of the ink.

Example:

```
app.doc.pages[0].inks[0].name
```

**app.doc.pages[i].getPageBox()** Returns an array containing the left, top, right and bottom coordinates of the TrimBox of the specified page box in pt.

Example:

```
app.doc.pages[0].getPageBox()
```

**app.doc.pages[i].getPageBox(pageBox)** Returns an array containing the left, top, right and bottom coordinates of the specified page box in pt. 'pageBox' must be one of Art, Bleed, Crop, Trim and Media.

Example:

```
app.doc.pages[0].getPageBox(Trim)
```

**app.doc.pages[i].getPageBox(pageBox,precision)** Returns an array containing the left, top, right and bottom coordinates of the specified page box with the given precision in pt. 'pageBox' must be one of Art, Bleed, Crop, Trim and Media.

Example:

```
app.doc.pages[0].getPageBox(Trim,2)
```

**app.doc.pages[i].getPageRotation()** Returns the page rotation of the page.

**app.doc.xmp** Returns a XMP object for the document XMP metadata of the current PDF document.

**app.doc.xmp.getProperty(ns,property)** Returns the value of the specified property in the specified namespace or 'undefined' if the property does not exist. 'ns' must be the full namespace uri. For namespaces defined in the XMP spec the predefined namespace

prefix can be used as well.

Examples:

```
app.doc.xmp.getProperty(http://purl.org/dc/elements/1.1/,format)
```

```
app.doc.xmp.getProperty(dc,format)
```

**app.doc.metadata** Returns the document XMP metadata as plain XML

**app.doc.result** Returns a preflight result object or 'undefined' if no preflight result is available. A preflight result is only available inside process plans if a profile or check was executed in a previous step.

**app.doc.result.hits** Returns information about a previous preflight result.

**app.doc.result.hits.numErrors** Returns the number of errors of a previous preflight result.

**app.doc.result.hits.numWarnings** Returns the number of warnings of a previous preflight result.

**app.doc.result.hits.numInfos** Returns the number of info hits of a previous preflight result.

**app.doc.result.checks** Returns an array of Check objects for the previous preflight result.

**app.doc.result.checks.length** Returns the length of the array of Check objects for the previous preflight result.

**app.doc.result.checks[i].id** Returns the check ID of the specified check for the previous preflight result.

Example:

```
app.doc.result.checks[0].id
```

**app.doc.result.checks[i].name** Returns the display name of the specified check of the previous preflight result.

Example:

```
app.doc.result.checks[0].name
```

**app.doc.result.checks[i].severity** Returns the severity of the specified check for the previous preflight result:  
1: Info, 2: Warning, 3: Error.

**Example:**

*app.doc.result.checks[0].severity*

*app.doc.result.checks[i].numHits* Returns the number of hits of the specified check for the previous preflight result.

**Example:**

*app.doc.result.checks[0].numHits*

*app.doc.result.checks[i].pageNumbers* Returns an array of page numbers (starting with 0) for pages that had hits with the specified check for the previous preflight result.

**Example:**

*app.doc.result.checks[0].pageNumbers*

# Extracting information from an XML Report file via XPath (9.1)

The pdfToolbox specific "app.doc.result.reports" object returns an array of reports that have been generated in a previous Process Plan step. It can be combined with file.read which would read an XML report into a string and to then convert that string back into an XML object with xml=new XML().

Then xml.registerNamespace allows for associating the XML Report namespace, which is ["http://www.callassoftware.com/namespace/pi4"](http://www.callassoftware.com/namespace/pi4) for pdfToolbox 9 XML Reports with a abbreviation.

Finally xml.path can be used to read information from the XML object via an XPath expression.

The example below extracts the information about what plates are used by a PDF file from the XML report and writes that information into a variable "text". In the Process Plan example which is attached to this article the value of this variable is then used in a later step to write that information onto all PDF pages.

```
//Get first report, assign it to "file", read it's content into a string and
//convert that string into an XML object
app.vars.report = app.doc.result.reports[0];
var file = new File( app.doc.result.reports[0] );
var string = file.read();
var xml = new XML(string);

//Register the XML namespace with p
xml.registerNamespace("p","http://www.callassoftware.com/namespace/pi4");

//Get the list of platenames
app.vars.plates = xml.xpath( "//p:report/p:document/p:doc_info/p:platenames/
p:platenamename/text()" );

//Write the list of platenames into a variable that is available throughout the
execution context
app.vars.text = app.vars.plates;
```



Evaluate\_XML\_Report\_-\_Place\_Plate\_names\_extracted\_from\_XML\_report.kfpx

# Using an external JSON jobticket file

## (9.1)

This example shows how processing information can be taken from a jobticket file, which has been saved next to the currently processed PDF. The jobticket file uses JSON and does in this example only have one key value pair.

The download contains a Profile with a Process Plan, a sample PDF and a JSON jobticket.



Reading\_a\_jobticket\_from\_a\_sidecar\_file\_(JSON).zip

The first step in the Process Plan is a Variable that tries to read the jobticket and stores one of the values in it into the variable "text". It does the same for the text size which is different for regular content and for an error message that is saved into "text" instead if reading the jobticket fails. This variable is made available via app.vars to the next step which takes it and prints its contents onto the PDF page.

This is the variable from the first step.

```
debug = true;
function buildSidecarFileName( extension )
{
    var path = app.doc.path.split(app.env.pathDelimiter);
    if (debug) console.log( "buildSidecarFileName 1 Path: " + path);
    var name = path[path.length-1].split(".");
    if (debug) console.log( "buildSidecarFileName 2 Name: " + name);
    name.pop();
    if (debug) console.log( "buildSidecarFileName 3 Name: " + name);
    name.push(extension);
    if (debug) console.log( "buildSidecarFileName 4 Name: " + name);
    path.pop();
    if (debug) console.log( "buildSidecarFileName 5 Path: " + path);
    path.push(name.join("."));
    if (debug) console.log( "buildSidecarFileName 6 Path: " + path);
    path = path.join(app.env.pathDelimiter);
    if (debug) console.log( "buildSidecarFileName 7 Path: " + path);
    return new File(path);
}
try
```



```
{
    app.vars.sidecar = JSON.parse( buildSidecarFileName("json").read());
    if (debug) console.log( "main 1 The jobticket file: " + JSON.
stringify(app.vars.sidecar));
    app.vars.text = app.vars.sidecar.msg;
    app.vars.fontsize = 25;
    app.vars.ok = true;
}
catch( x )
{
    app.vars.text = "ERROR: Could not read message from sidecar file: " + x;
    app.vars.fontsize = 20;
    app.vars.ok = false;
}
app.vars.ok;
```

The first call sets the debug variable to true. This allows for reading the current state of processing in the Console window of the JavaScript editor (if "Show evaluation results" is on). This is the output for "testimonial Mercedes.PDF".

```
buildSidecarFileName 1 Path: ,Users,d.seggern,Doku,Reading a jobticket from a
sidecar file (JSON),testimonial Mercedes.pdf
buildSidecarFileName 2 Name: testimonial Mercedes, pdf
buildSidecarFileName 3 Name: testimonial Mercedes
buildSidecarFileName 4 Name: testimonial Mercedes, json
buildSidecarFileName 5 Path: ,Users,d.seggern,Doku,Reading a jobticket from a
sidecar file (JSON),
buildSidecarFileName 6 Path: ,Users,d.seggern,Doku,Reading a jobticket from a
sidecar file (JSON),testimonial Mercedes.json
buildSidecarFileName 7 Path: /Users/d.seggern/Doku/Reading a jobticket from a
sidecar file (JSON)/testimonial Mercedes.json
main 1 The jobticket file: {"msg":"This string came from a sidecar file!"}
```

The Console output shows how the path for the jobticket file is built from the path of the PDF file, in the lines starting with "buildSidecarFileName" and the contents of the jobticket file, which is rather short in this example.

# Defining variables using app.requires with closed choice of allowed values (9.1)

The app.requires object can be used for two reasons:

To explicitly define dependencies in a Script variable from another variable. That should always be done when variables are used e.g. in Checks or Fixups where it is important that a certain order is used when evaluation the variables.

The other more important use case is to define a variable within a JavaScript variable. This will be described in this article.

The example is variant of the "Viewing Distance related checks" Profile that you will find in pdfToolbox in the "Shapes, Variables, JavaScript, Place content" library.



Viewing\_Distance\_related\_checks.kfpx

The predefined Profile has a Profile level JavaScript variable that starts with

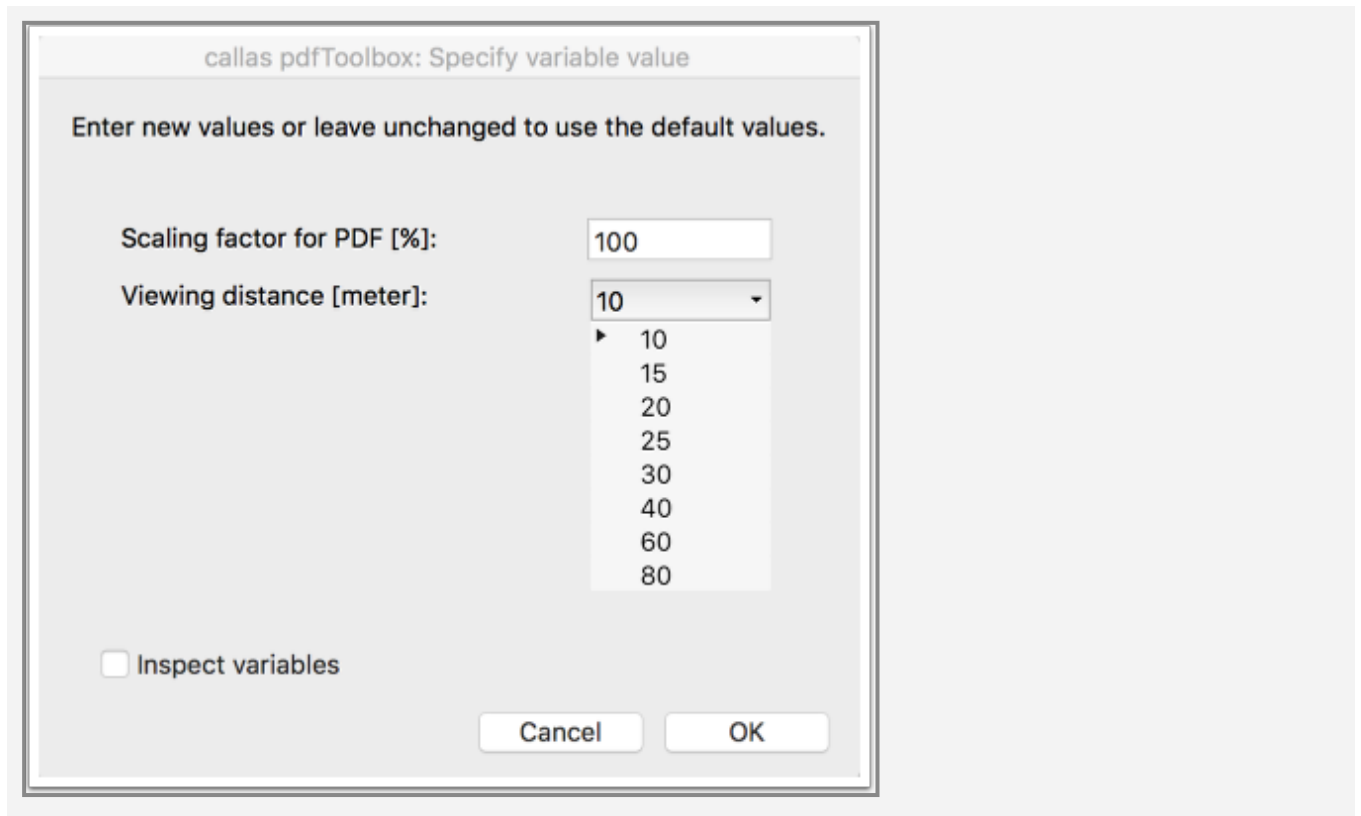
```
app.requires("input_viewingdistance",10,"Viewing distance [meter]");  
app.requires("input_scalingfactor",100,"Scaling factor for PDF [%]");
```

The two app.requires calls define two variables, "input\_viewingdistance" and "input\_scalingfactor", with default values (second parameter) and a label text that is used when the Profile is executed in pdfToolbox Desktop (third parameter).

With pdfToolbox 9.1 it is possible to define a list of possible values.

```
app.requires("input_viewingdistance",10,"Viewing distance  
[meter]", [10,12,14,18,20]);
```

If you want to only allow certain viewing distance values for a variable that is defined using app.requires you can use a fourth parameter that takes an array of values.



(For variables that are defined via the variable pop up you can do the same by using the "Limit input values to specific values" option.)

## Using "trigger" values to adjust processing in a Process Plan (9.1)

pdfToolbox 9.1 lets you access trigger values via a JavaScript object.

Trigger values are values that pdfToolbox reports back for objects that are identified by a check. The type of value corresponds to the check property that is used. E.g. for a check that finds all image objects with an image resolution below 300 ppi the trigger value reports the actual image resolution for each such image. For a check that finds objects that are close to TrimBox the trigger value has the actual distance between any such object and the TrimBox. The examples below use these two check properties and their trigger values to adjust processing:

- List images with lowest resolution per page (uses trigger values)  
Uses the image resolution trigger values and prints on each page of a PDF file the lowest resolution that is used by an image on that page.
- Use trigger values to calculate the width for page mirroring for bleed creation  
Uses the smallest distance between a text object and the TrimBox to define the width for mirroring page content. That makes sure that text objects do not show up in the bleed. It does this in a loop until there is no text closer than 3 mm to the mirrored page content.

The download contains kfx Process Plans for the two examples as well as a simple demo file that can be used to try them out.

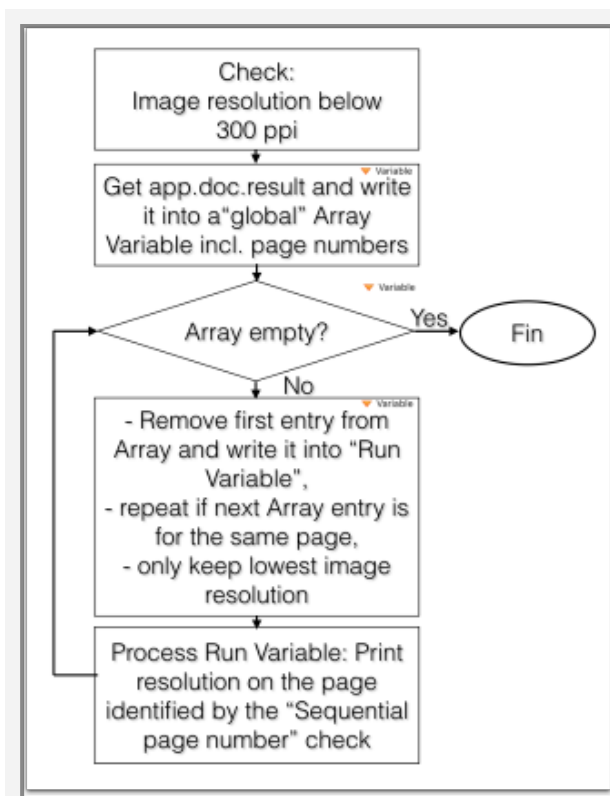


Trigger\_value\_example.zip

### List images with lowest resolution per page (uses trigger values)

This Process Plan runs in a loop. The flow chart below gives an overview about how processing takes place in the 5 steps

of the Process Plan and how that makes sure that each page is processed separately.



## Use trigger values to calculate the width for page mirroring for bleed creation

The 6 steps of this Process Plans are:

1. Set ArtBox to TrimBox  
This is done to "remember" the TrimBox since that has to be modified during processing.
2. Check: Text is closer than 3 mm inside to TrimBox  
Finds all text objects that are so close to TrimBox that they would be mirrored by a bleed generation fixup.
3. Mirror page into bleed with a width that text is not mirrored  
The width of bleed is defined by using a Script Variable "width\_of\_bleed". That first copies app.doc.result.checks[0].hits into a local Variable. It then generates an array variable "loc\_triggerarray" into which all trigger values are written. Finally Math.min.apply is used to determine what is the smallest distance.

4. Then the TrimBox is set to the new CropBox because that makes it easier to find out whether we already have generated enough bleed.

After that processing goes back to the Check in step 2. If by then there is no text that is close to the (new) TrimBox (that at the same time is the CropBox) processing goes forth to step 5, otherwise the procedure above is repeated.

5. Processing get to here only if there is no text closer than 3 mm to TrimBox. It sets the TrimBox back to the ArtBox which was the original TrimBox of the file after step 1.
6. Finally the ArtBox that was only used temporarily has to be removed.

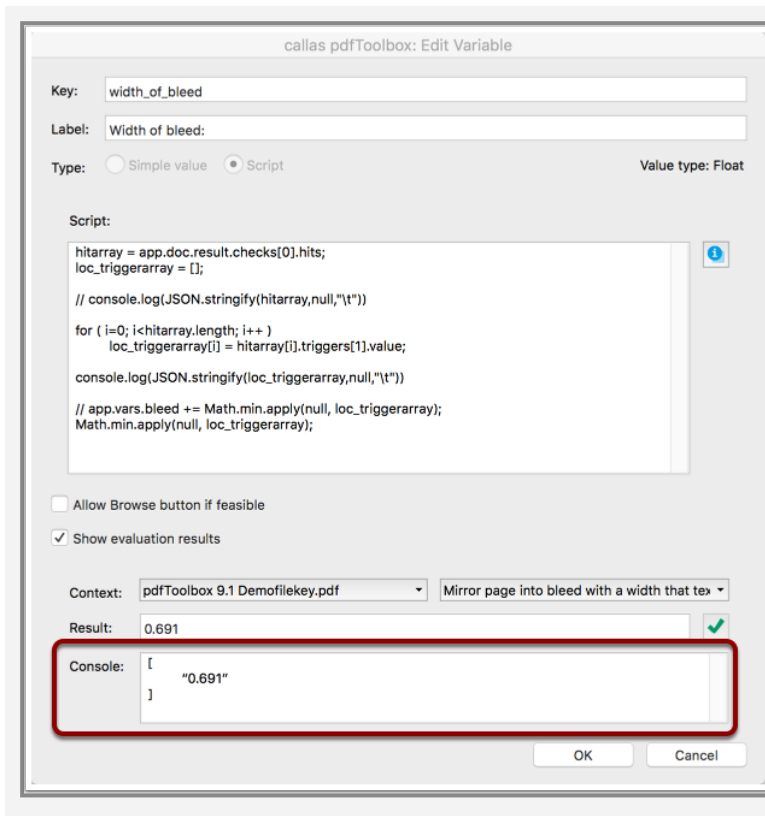
# Debugging JavaScript Variables (9.1)

pdfToolbox provides features that are designed specifically to make debugging of Script Variables easier:

- Console
- Serialisation of pdfToolbox JavaScript objects
- Variable values are listed when "Log Profile Execution" is active

## Console

The Console is available below the Script Editor if "Show evaluation result" is enabled.



It works in the same way as in other JavaScript editors.

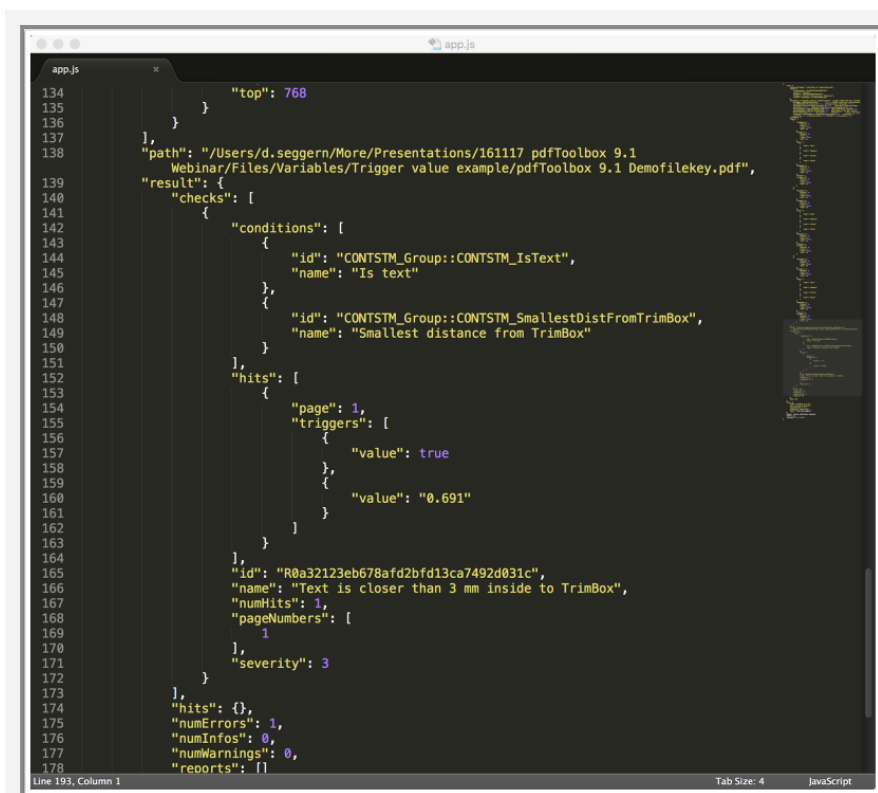
Since JavaScript "snippets" may be used at various places in a Profile it is sometimes required to temporarily copy code from somewhere else into the first lines of the Script Variable you are currently working on to make sure that the Variable has the same information as when it will be used in the Profile context.

In the example above the check that would fill the `app.doc.result.checks[0].hits` array has to be manually performed before the code in this Variable can be debugged. Only then the variable `"loc_triggerarray"` has a meaningful value. It is printed to the Console via `console.log` using "pretty printing" with the optional parameters `null, "\t"`.

## Serialisation of pdfToolbox JavaScript objects

In order to access information in any of the pdfToolbox JavaScript objects you may need to visualise the structure of these sometimes complex objects. The most complex object is the `app` object since that is the parent object for all other more specific objects.

This and all other pdfToolbox JavaScript objects can be serialised by the `JSON.stringify` function, if used with the optional parameters `null, "\t"` they are formatted nicely.



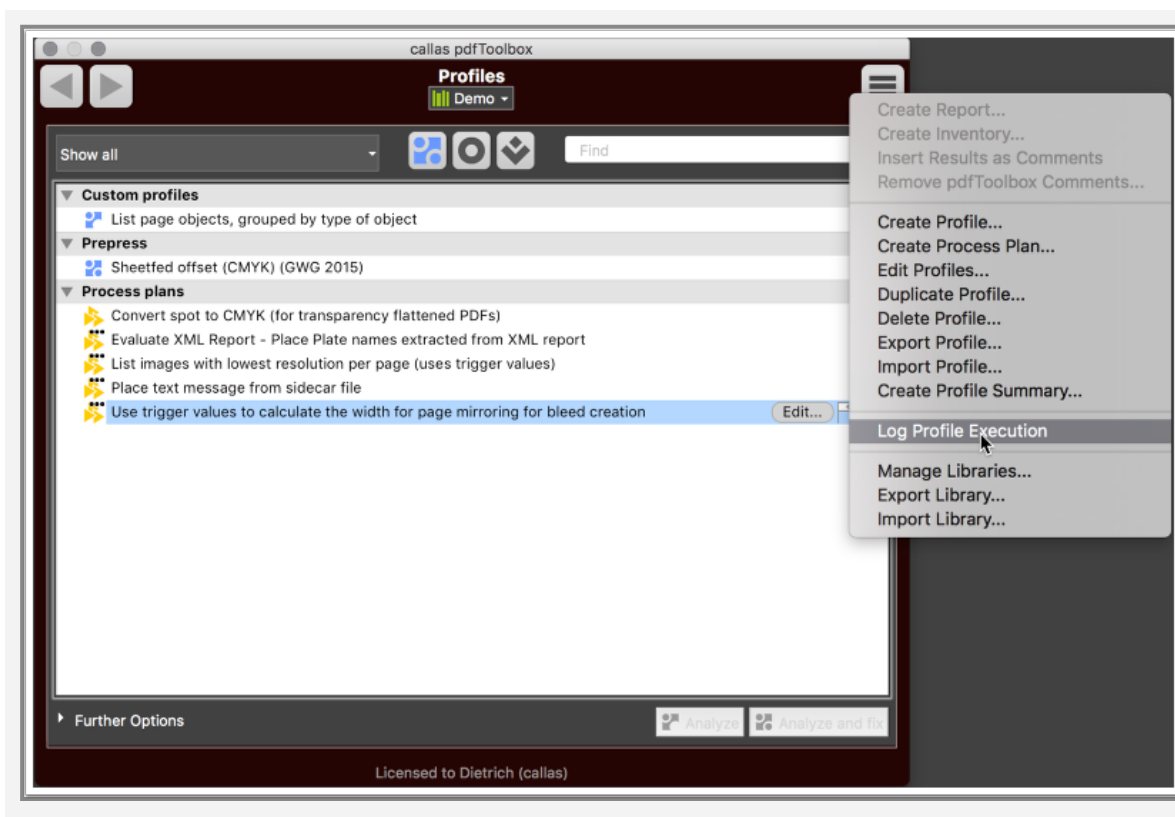
This screenshot shows parts of the `app` object after that has been output via `console.log(JSON.stringify(app,null,'\t'))` and copied from the Console into the Sublime editor. The



part above shows the app.doc.result.checks part with hits and their trigger values.

## Variable values listed in a logfile created via Log Profile Execution

If you are using Script Variables in a Process Plan you may need to know how Variable values are changed throughout processing. You enable logging via the options menu in the pdfToolbox main window.



If switched on a logfolder will be created that amongst other things includes a log file "process.log". The other contents of this folder are explained in further detail in a different article of this documentation: <http://help.callassoftware.com/m/pdftoolbox9-en/l/656888>.

All Variables that are stored in app.vars so that they are available throughout a Process Plan are listed for each of the steps of the Process Plan with their current values.

```
2016-11-16 12:49:15 Continuing with step 3
2016-11-16 12:49:15 [9-3] ExitIfAllHitsProcessed
2016-11-16 12:49:15 Var BuildArrayWithImageResolutions {"doc":{"documentFileName":"Wine VI.pdf","info":{"CreationDate":"D:20130711152259+08'00'","Creator":"XMPie-uCreate
2016-11-16 12:49:15 Var ExitIfAllHitsProcessed 1
2016-11-16 12:49:15 Var MakeTemparray [{"page":2,"triggers":[{"value":8}, {"value":237.1903533935547}]}], {"page":2,"triggers":[{"value":8}, {"value":118.47789001464844}]}]}
2016-11-16 12:49:15 Var hitarray [{"page":2,"triggers":[{"value":8}, {"value":237.1903533935547}]}], {"page":2,"triggers":[{"value":8}, {"value":118.47789001464844}]}]}
2016-11-16 12:49:15 Var temparray [{"page":1,"triggers":[{"value":8}, {"value":119.35264587402344}]}]}
2016-11-16 12:49:15 Result Success

2016-11-16 12:49:15 Continuing with step 4
2016-11-16 12:49:15 [10-4] MakeTemparray
2016-11-16 12:49:15 Var BuildArrayWithImageResolutions {"doc":{"documentFileName":"Wine VI.pdf","info":{"CreationDate":"D:20130711152259+08'00'","Creator":"XMPie-uCreate
2016-11-16 12:49:15 Var ExitIfAllHitsProcessed 1
2016-11-16 12:49:15 Var MakeTemparray []
2016-11-16 12:49:15 Var hitarray []
2016-11-16 12:49:15 Var temparray [{"page":2,"triggers":[{"value":8}, {"value":118.47789001464844}]}]}
2016-11-16 12:49:15 Result Success
```

This screenshot shows that the variable "hitarray" had two entries in step 3 and was empty in step 4. (The empty lines before, between and after these two steps have been added to make the example more readable.)

# Shapes

# Shapes: An overview

Shapes are a new way to use existing content or page information and either derive new content or clip the existing content on a page.

This can come in handy in number of scenarios, e.g. when:

- adding white background to be printed behind the page content, but not in areas where nothing is printed
- adding partial varnish on certain objects
- creating a dieline based on page content or page geometry
- clipping page content where for example irregularly shapes have to be imposed without wasting space on the imposed sheet as would be the case if the imposition would be based on the bounding rectangle of the label
- create versions of complex production files that clip or overlay distracting technical content and allow an unobstructed view of the main page content, e.g. the label as it will actually appear out of the printing process

In order to enable these and many more uses, the Shapes feature are configured in two steps

- the actual shape(s) have to be defined; at this stage "shapes" are considered to just be abstract definitions of some area(s) on a page
- next, an action is defined that is executed using the defined shape(s); for example, a shape can be filled, outlined or used for clipping

Both steps come with a number of parameters that determine exactly how shapes are created, or how actions are to be executed. The necessary details are discussed in the next two articles.

## Designing shapes

In many cases, the use of shapes will be obvious. For example, when creating a die line based on the trim box, optionally with rounded corners, a user would simply define shape based on the page's trim box, set rounded corner radius to 3mm, and would then define the action to be applied to the shape as a spot color outline with a spot color named "Dieline".

In other cases more complex requirements may have to be accommodated. For example, when a partial has to be created over any part of the page where something is actually printed, except one area where the barcode goes, as this area shall not become glossy (it is difficult to read barcodes with a glossy surface). While this can be achieved relatively easily using the Shapes feature, it can quickly become confusing, especially at the beginning, as Shapes are defined in a very abstract fashion.

In all cases where shapes are to be defined and used in non-trivial ways, it is highly recommended to make a simple drawing, using old fashioned pencil and paper, reflecting the expected page content page geometry and so forth, and then draw the shape information to be derived, and how it is to be derived based on existing information. When doing this, please also note, that in some cases two or more separate steps may be necessary. In such a case a Process Plan may be used, that runs a sequence of Shape steps.

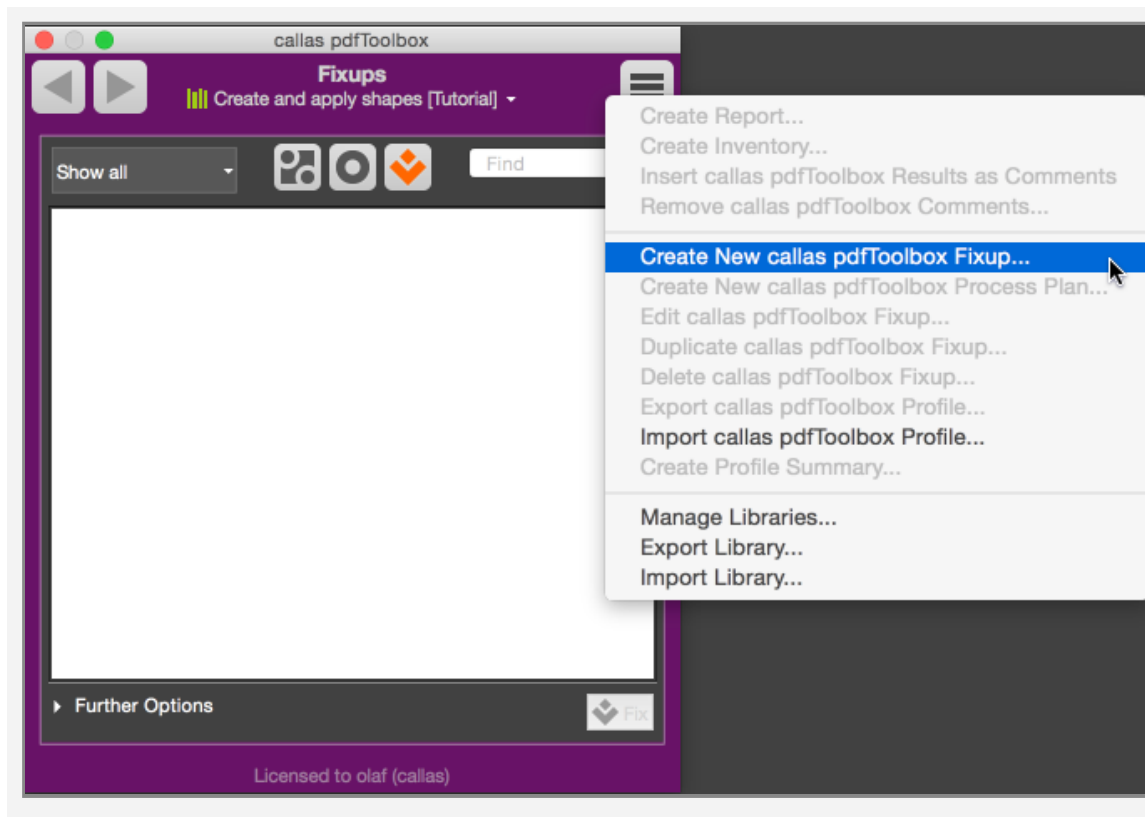
# Defining shapes

Working with shapes means using the "Create and apply shapes" fixup. pdfToolbox 9 comes with a small set of predefined "Create and apply shapes" fixups, which can be found in the "Shapes, Variables, JavaScript, Place content" library. The content below will explain how to create custom "Create and apply shapes" fixups - specifically, how to configure the part that creates shapes. Applying actions to shapes will be explained in the next article.

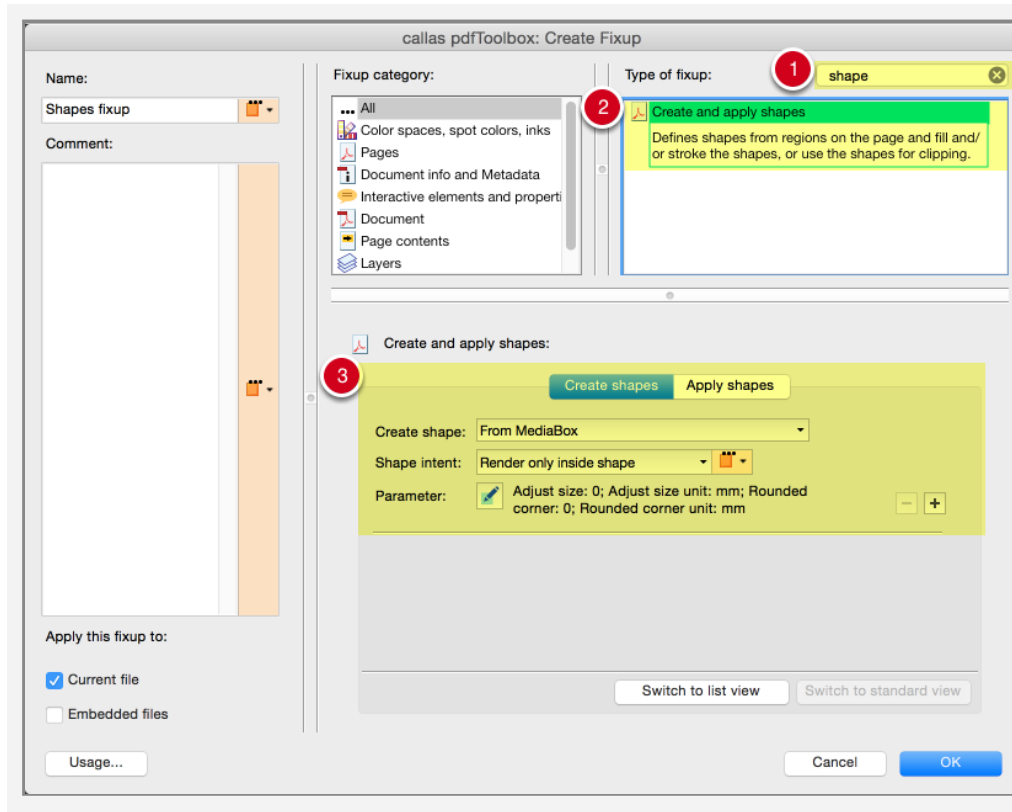
## Creating a Shapes fixup

In order to create a new Shapes fixup, simply create a new fixup, search for "Create and apply shapes" under "Type of fixup" and select "Create and apply shapes"

## Create new fixup



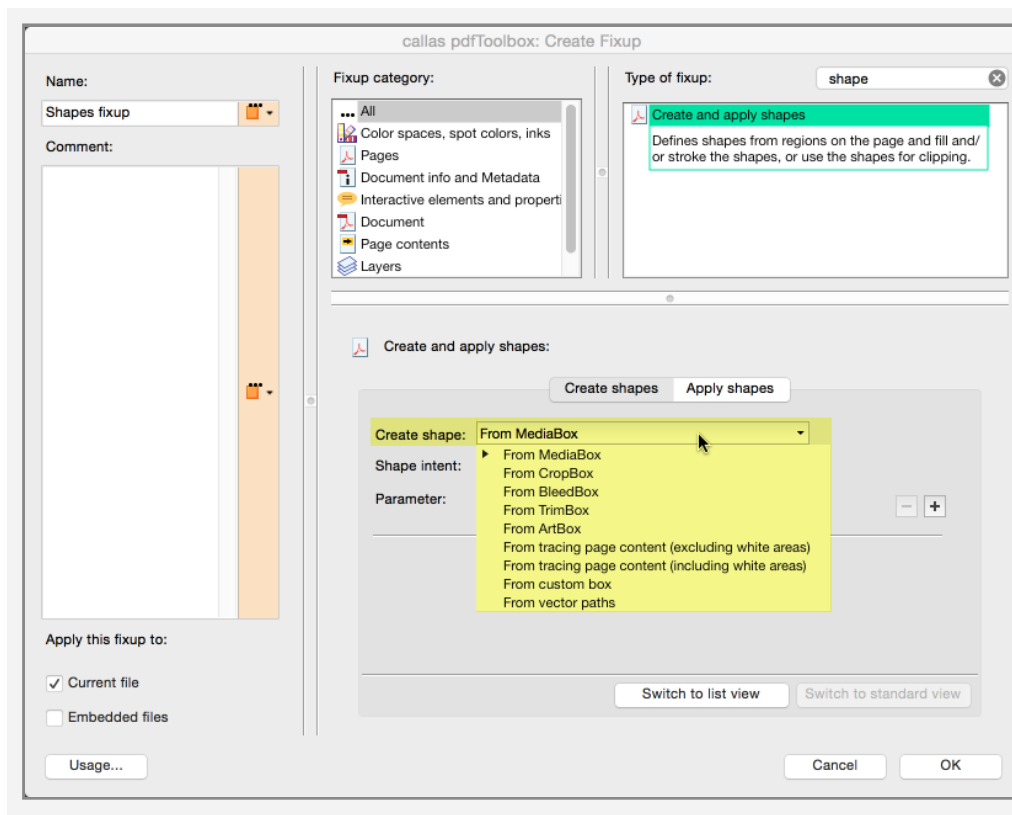
## Setting up new fixup as "Create and apply shapes" fixup



1. enter "shape" in the search field
2. select the "Create and apply shapes" entry in the "Type of fixup" list
3. use the options under "Create shapes" to configure how shapes are to be created

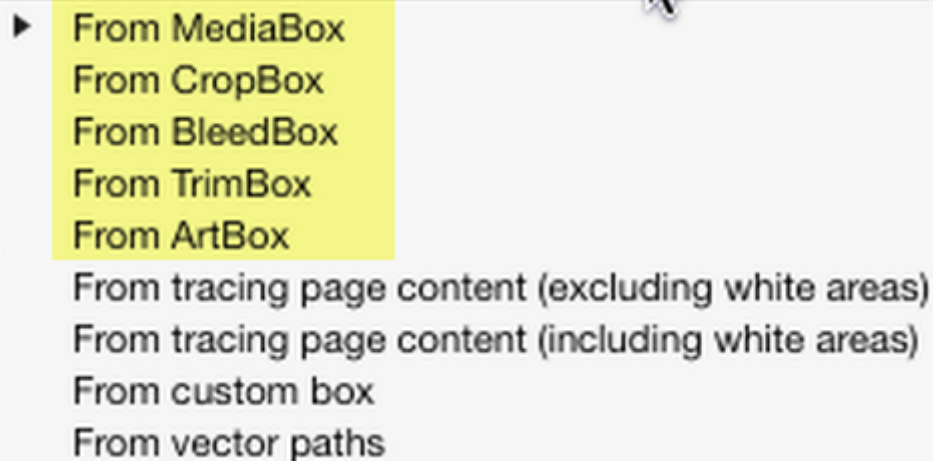


## "Create shape" parameter



There are a number of approaches how one or several shapes can be created. The steps below will discuss each (group of) such approaches.

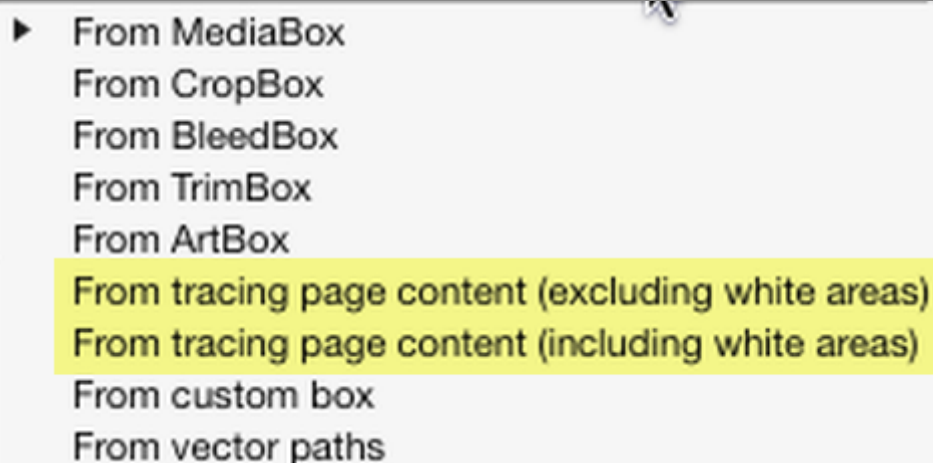
## Discussion of the "Create shape: parameters: MediaBox, CropBox, BleedBox, TrimBox, ArtBox

- 
- ▶ From MediaBox
  - From CropBox
  - From BleedBox
  - From TrimBox
  - From ArtBox
  - From tracing page content (excluding white areas)
  - From tracing page content (including white areas)
  - From custom box
  - From vector paths

The simplest approach is to pick up a page geometry box (i.e. one of MediaBox, CropBox, BleedBox, TrimBox or ArtBox).

[Further below](#) you will find a discussion how to fine tune the use of such a page geometry box.

## Discussion of the "Create shape: parameters: From tracing page content

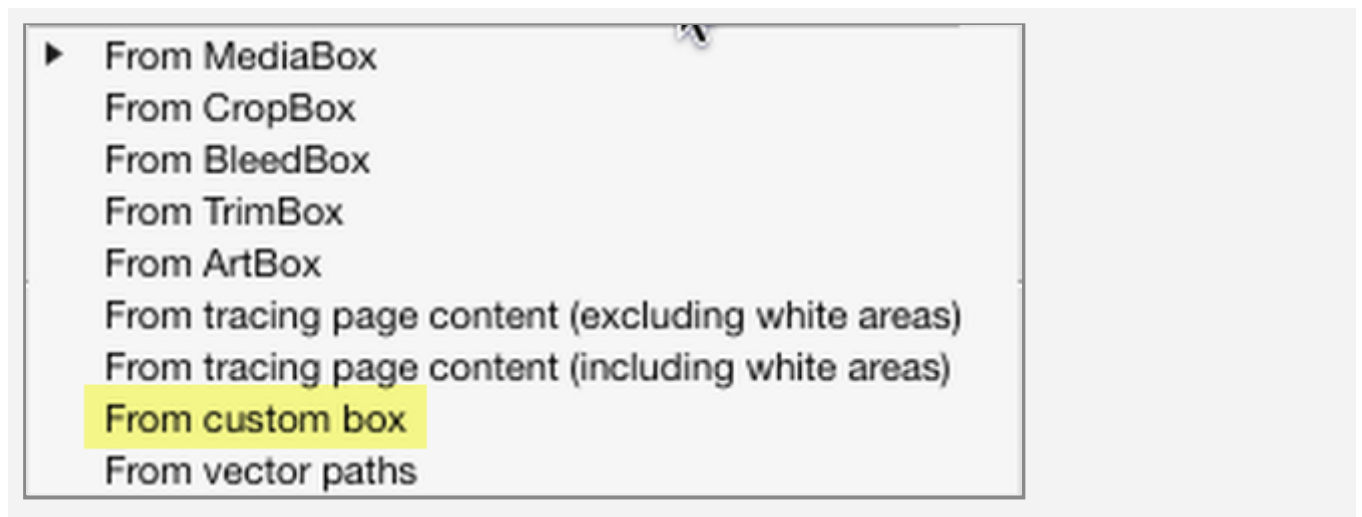
- 
- ▶ From MediaBox
  - From CropBox
  - From BleedBox
  - From TrimBox
  - From ArtBox
  - From tracing page content (excluding white areas)
  - From tracing page content (including white areas)
  - From custom box
  - From vector paths

A more advanced approach is to use the rendered appearance of page content and create an outline around it (and for any 'holes' inside it). Depending on whether white areas (as

opposed to areas that just look white because they are actually transparent and let the white background shine through) are considered part of the rendered page content or not, it is necessary to choose between "From tracing page content (*including* white areas)" and "From tracing page content (*excluding* white areas)".

[Please see below](#) for a discussion of how to adjust parameters for tracing page content.

## Discussion of the "Create shape: parameters: MediaBox, CropBox, BleedBox, TrimBox, ArtBox (Copy) (Copy)"



This approach is similar to the "From MediaBox" etc. approaches but instead of picking up the existing page geometry box, it is necessary to provide the coordinates of the desired box and its position on the page. [Please see below](#) for a discussion of the necessary parameters.

## Discussion of the "Create shape: parameters: MediaBox, CropBox, BleedBox, TrimBox, ArtBox (Copy) (Copy) (Copy)"



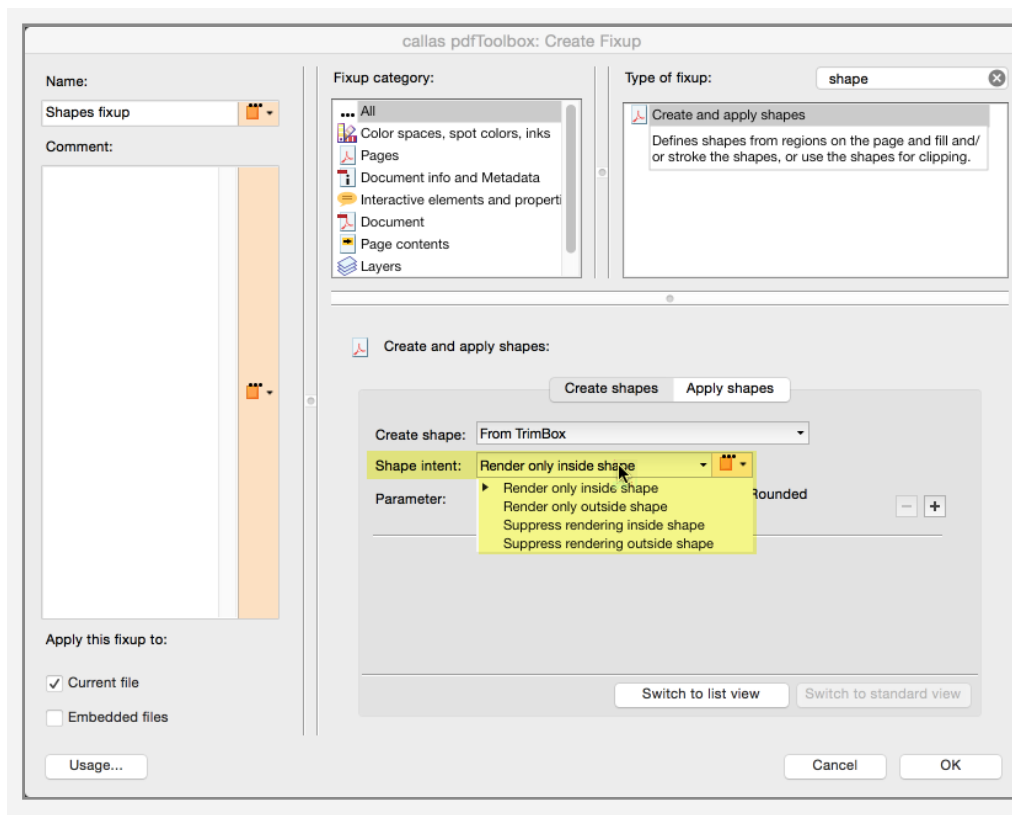
Another way to derive shapes from page content is to use "From vector paths". This will simply pick up existing path objects (and by implication it does not pick up images, image masks, font based text objects, soft masks, or smooth shades) and turn them into shapes.

### Important:

Usually it does not make sense to pick up all path objects on the page, but rather only a small set of path objects, or even just exactly one path object – e.g. a die-line colored in a specific spot color.

[Please see below](#) for discussion of how to use parameters for this approach.

## "Shape intent" parameter



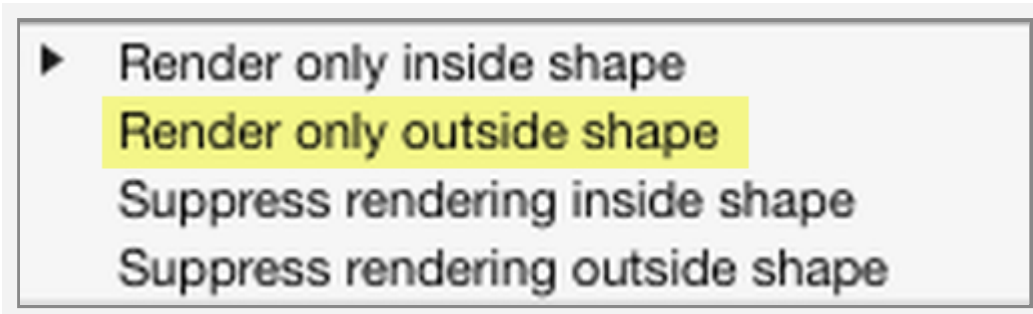
Shapes can be put to use in a number of ways. While aspects such as how to color the area inside a shape or how to stroke its contour are discussed in the next article about applying shapes, some control over the use of each of the individual shapes is already provided in this "Create shape" article.

### Discussion of the "Shape intent" parameter: Render only inside shape

- **Render only inside shape**
- Render only outside shape
- Suppress rendering inside shape
- Suppress rendering outside shape

Where shapes are to be used for filling the area inside them with a certain color, this would be the option to choose. It is important though to understand that where shapes are inside one another, the even odd rule for painting applies – i.e. if a shape consists of two circles where one circle is completely inside the other circle, only the area between the contours of the two circles will be colored, the areas outside the outer circle and the area inside the inner circle would remain uncolored.

## Discussion of the "Shape intent" parameter: Render only outside shape

- 
- ▶ Render only inside shape
  - Render only outside shape
  - Suppress rendering inside shape
  - Suppress rendering outside shape

Where shapes are to be used for filling the outside of an area, but not the area inside it, with a certain color, this would be the option to choose. It is important though to understand that where shapes are inside one another, the even odd rule for painting applies. I.e. in this case if a shape consists of two circles where one circle is completely inside the other circle, the area between the contours of the two circles will not be colored, but the areas outside the outer circle and the area inside the inner circle would be colored.

## Discussion of the "Shape intent" parameter: Suppress rendering inside shape

- ▶ Render only inside shape
- Render only outside shape
- Suppress rendering inside shape
- Suppress rendering outside shape

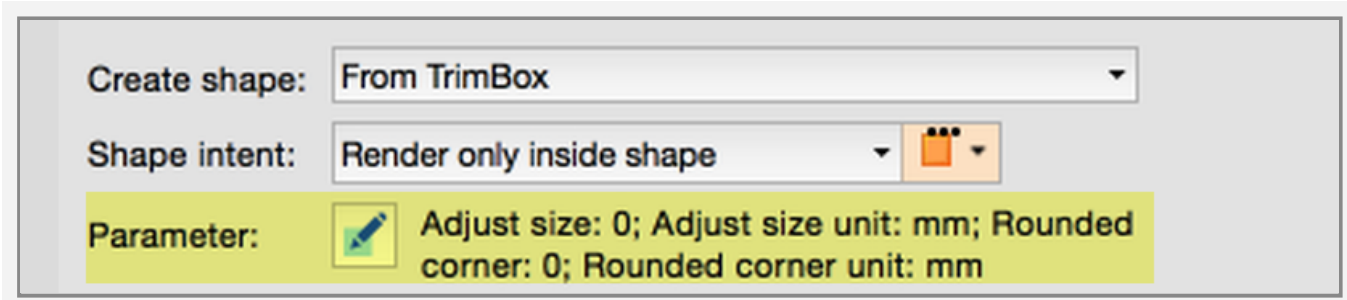
This shape intent ensures that regardless of any of the other shapes and their shape intent the inner area defined by this shape will remain uncolored. This will normally only make sense if this shape is combined with at least one other shape, that creates a colored (or stroked) area.

## Discussion of the "Shape intent" parameter: Suppress rendering outside shape

- ▶ Render only inside shape
- Render only outside shape
- Suppress rendering inside shape
- Suppress rendering outside shape


This shape intent ensures that regardless of any of the other shapes and their shape intent the outside area defined by this shape will remain uncolored. This will normally only make sense if this shape is combined with at least one other shape, that creates a colored (or stroked) area.

## Additional parameters for defining shapes



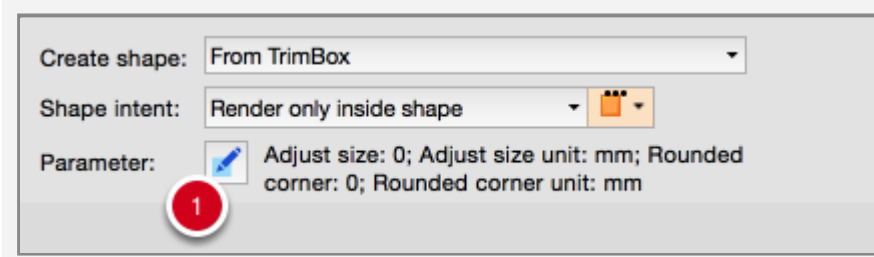
Create shape: From TrimBox

Shape intent: Render only inside shape

Parameter:  Adjust size: 0; Adjust size unit: mm; Rounded corner: 0; Rounded corner unit: mm


The third setting "Parameter" in a shape configuration depends on the actual "Create shape" setting, and will differ substantially between them. The steps below explain the parameters for each of the "Create shape" settings.

### Shapes based on MediaBox, CropBox, BleedBox, TrimBox, ArtBox



Create shape: From TrimBox

Shape intent: Render only inside shape

Parameter:  Adjust size: 0; Adjust size unit: mm; Rounded corner: 0; Rounded corner unit: mm

When the "Create shape" setting is set to any of the page geometry boxes (in this example it is set to "From TrimBox"), the current "Parameter" values are reported.

1. Clicking on the "Edit" button, the "Create shape parameters" dialog will open, offering the applicable parameters for editing.



## Parameters for shapes based on MediaBox, CropBox, BleedBox, TrimBox, ArtBox

Create shapes parameters

1 Adjust size: 0

2 Adjust size unit: mm

3 Rounded corner: 0

4 Rounded corner unit: mm

Cancel OK

1. Adjust size makes it possible to subtract from (negative numbers) or to add (positive numbers) to the page geometry box. For example, entering "-5" will make the resulting rectangular shape smaller by 5 units on each side of the rectangle.
2. "Adjust size unit" makes it possible to pick between point, millimeter and inch.
3. Where it is desired to create a rectangular shape with rounded corners, a number greater than zero needs to be entered in this field. The number defines the radius of the rounded corners.
4. "Rounded corner unit" makes it possible to pick between point, millimeter and inch.

## Shapes based on tracing page content (including or excluding white areas)

Create shape: From tracing page content (excluding white areas)

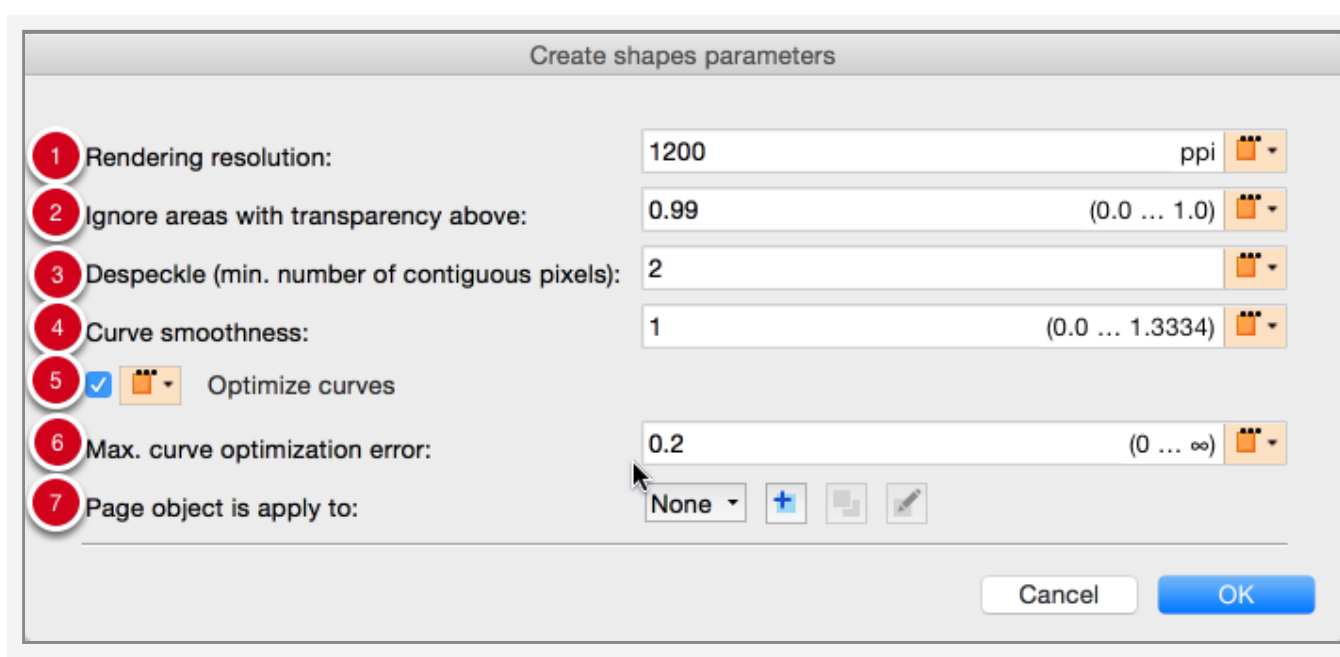
Shape intent: Render only inside shape

Parameter: 1 Adjust size: 0; Adjust size unit: mm; Rounded corner: 0; Rounded corner unit: mm

When the "Create shape" setting is set to one of the two "From tracing page content" options (in this example it is set to "From tracing page content (excluding white areas)"), the current "Parameter" values are reported.

1. Clicking on the "Edit" button, the "Create shape parameters" dialog will open, offering the applicable parameters for editing.

## Parameters for shapes based on tracing page content (including or excluding white areas)



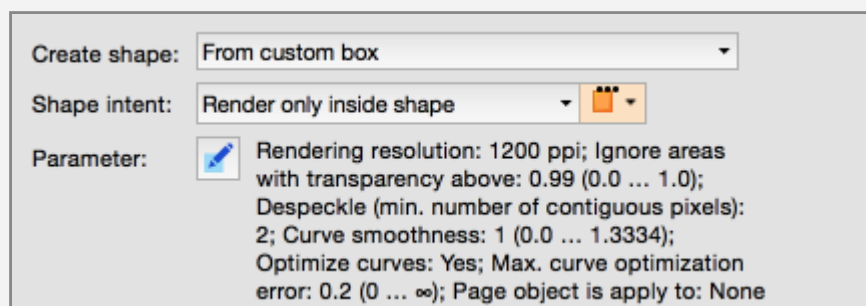
The parameters in this dialog provide some control over the rendering of the page for tracing purposes, and over the tracing as such.

**Important:** The default settings are the result of extensive research, and are considered to be highly suitable for most use cases. Only in the rare case, where there is reason to assume modified values for these parameters will provide more suitable results, should the values be modified.

1. Resolution of the (internally rendered) page image
2. Whether to consider rendered values above this value as transparent
3. Apply despeckling no more than this number of pixels are of the same color and are surrounded on all sides by pixels of the other color.

4. Determines the "curve smoothness" to achieved. A value of zero would lead to a tracing result where the sides of each pixel of the rendered image are followed precisely, leading to a jaggy tracing result (though at 1200 ppi) the jagginess might not be readily noticeable – nonetheless, this lack of smoothness will increase time to process, and will not add any actual quality to the tracing result.
5. Makes it possible to enable or disable curve optimization (see 6.)
6. Maximum delta allowed between a pixel perfect tracing result and the result of curve optimization. A value 0.2 will keep the optimization error at a minimum that will hardly ever be noticeable while still allowing for creation of efficient curves.
7. Makes it possible to create a rendered page only based on the objects found by this filter. For example, if the intention is to create a shape over all vector text objects of spot color "Orange", but not over the rest of the page content, a filter "Is text object using spot color 'Orange'" could be configured and selected, and tracing would happen based on a rendering of the page where the page (temporarily) only shows orange text.

## Shapes based on a custom defined box



When the "Create shape" setting is set to "From custom box", the current "Parameter" values are reported.

1. Clicking on the "Edit" button, the "Create shape parameters" dialog will open, offering the applicable parameters for editing.

## Parameters for shapes based on a custom defined box

The screenshot shows a dialog box titled "Create shapes parameters". It contains 12 numbered red circles on the left, each corresponding to a parameter. The parameters are as follows:

- 1 Relative to: Lower left corner (dropdown menu with an icon)
- 2 of: CropBox (dropdown menu with an icon)
- 3 Horizontal offset: 0 (text input field with an icon)
- 4 Vertical offset: 0 (text input field with an icon)
- 5 Width: (text input field with an icon)
- 6 Height: (text input field with an icon)
- 7 Custom page box unit: mm (dropdown menu with an icon)
- 8 Adjust size: 0 (text input field with an icon)
- 9 Adjust size unit: mm (dropdown menu with an icon)
- 10 Rounded corner: 0 (text input field with an icon)
- 11 Rounded corner unit: mm (dropdown menu with an icon)
- 12 Page object is apply to: None (dropdown menu with icons for fill, stroke, and text)

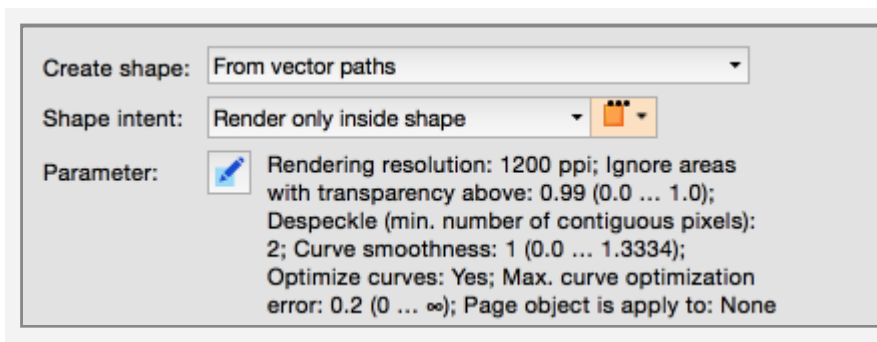
At the bottom right of the dialog are "Cancel" and "OK" buttons.

The parameters for "From custom box" are a combination of the parameters for page geometry boxes, and the parameters needed to describe the size and position of the custom page on the page area:

1. Determines the reference point (relative to the reference rectangle under 2.) starting from which the custom box shall be positioned
2. Determines the reference rectangle from which the custom box shall be positioned
3. Horizontal offset from the reference point where the respective corner of the custom box shall be (e.g. for "Lower right corner" of "CropBox" a positive value would move the lower right corner of the custom box to the right)
4. Vertical offset from the reference point where the respective corner of the custom box shall be (e.g. for "Lower right corner" of "CropBox" a positive value would move the lower right corner of the custom box downwards)
5. Width of the custom box
6. Height of the custom box

7. Measurement unit for the values entered in fields 3. through 6.
8. Adjust size makes it possible to subtract from (negative numbers) or to add (positive numbers) to the page geometry box. For example, entering "-5" will make the resulting rectangular shape smaller by 5 units on each side of the rectangle.
9. "Adjust size unit" makes it possible to pick between point, millimeter and inch.
10. Where it is desired to create a rectangular shape with rounded corners, a number greater than zero needs to be entered in this field. The number defines the radius of the rounded corners.
11. "Rounded corner unit" makes it possible to pick between point, millimeter and inch.
12. Makes it possible to limit the creation (and application) of this shape only to pages where the applicable filter applies. For example, if the filter is set to find objects using a spot color "Lime Green", only for pages that contain at least one object using that spot color will trigger the creation of this custom box.

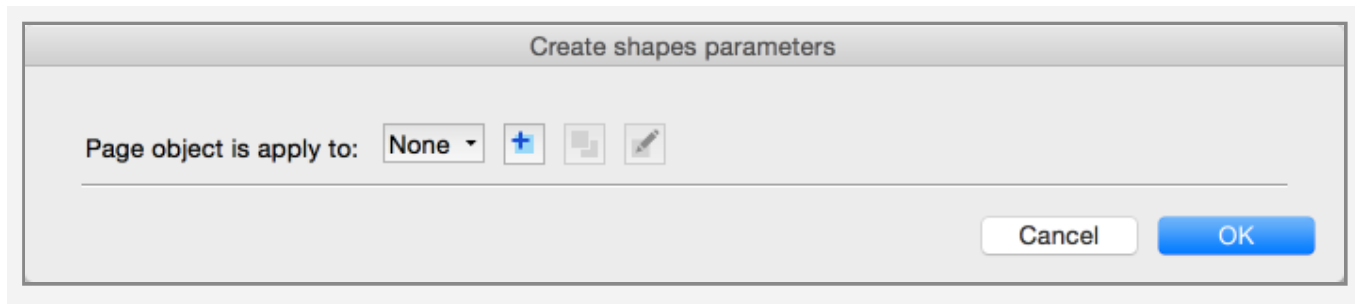
## Shapes based on existing vector paths



When the "Create shape" setting is set to "From vector paths", the current "Parameter" values are reported.

1. Clicking on the "Edit" button, the "Create shape parameters" dialog will open, offering the applicable parameters for editing.

## Parameters for shapes based on existing vector paths



For creation of shapes "From vector paths", only one parameter can be configured: a filter determining which of the vector objects on the respective page to use.

**Important:**

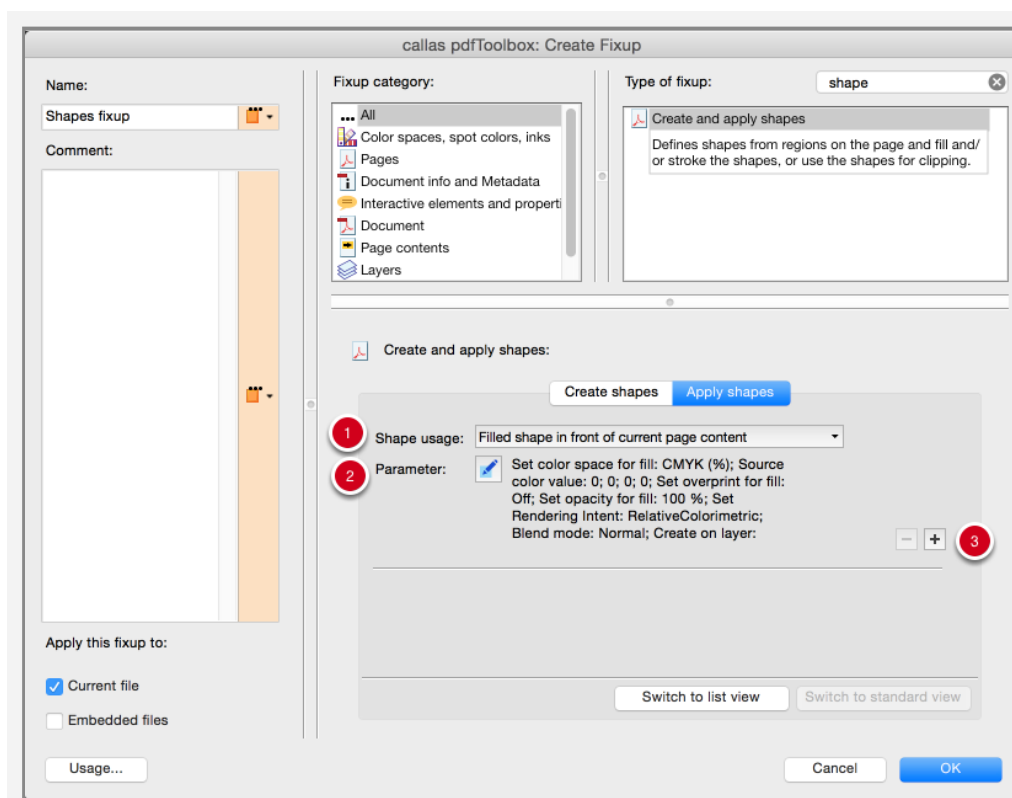
Usually it does not make sense to pick up all path objects on the page, but rather only a small set of path objects, or even just exactly one path object – e.g. a die-line colored in a specific spot color.

# Applying shapes

The "Create and apply shapes" fixup consists of two configuration sections. First, one or more shapes need to be defined under "Create shapes", next an action has to be defined under "Apply shapes" that determines how to make use of these shapes.

This article describes the configuration of the "Apply shapes" section.

## "Apply shapes" parameters

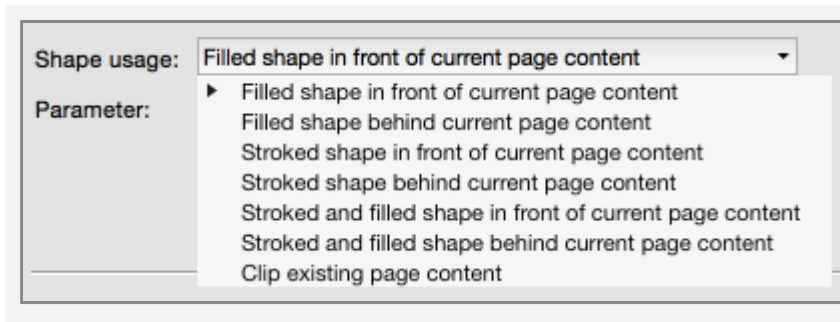


1. The popup menu under "Shape usage" determines what action to apply to the shapes created in the "Create shapes" section.
2. Depending on the chosen "Shape usage", different sets of parameters become available under "Parameter"

- The "+" (plus) button to the right makes it possible to define more than one action on the defined shapes. The "-" (minus) button makes it possible to remove such an action. It is only enabled if there are at least two actions in the list of "Apply shapes" actions.

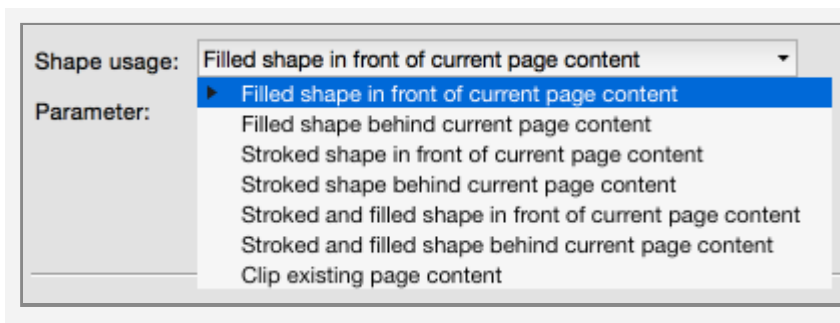
The following steps describe both the "Shape usage" options and the parameters that come with each these options.

## "Apply shape": List of settings



This screenshot lists the currently available "Shape usage" options.

## Filling shapes in front or behind existing page content



Important note:

The description below applies in the exact same fashion to filled shapes created **behind** current page content. The option to create a filled shape behind current page content **will only work as expected in very few cases**, as any current page content that is opaque will hide the filled shape created behind it. If for example the whole page has an opaque white background object, or any other opaque object that fills the

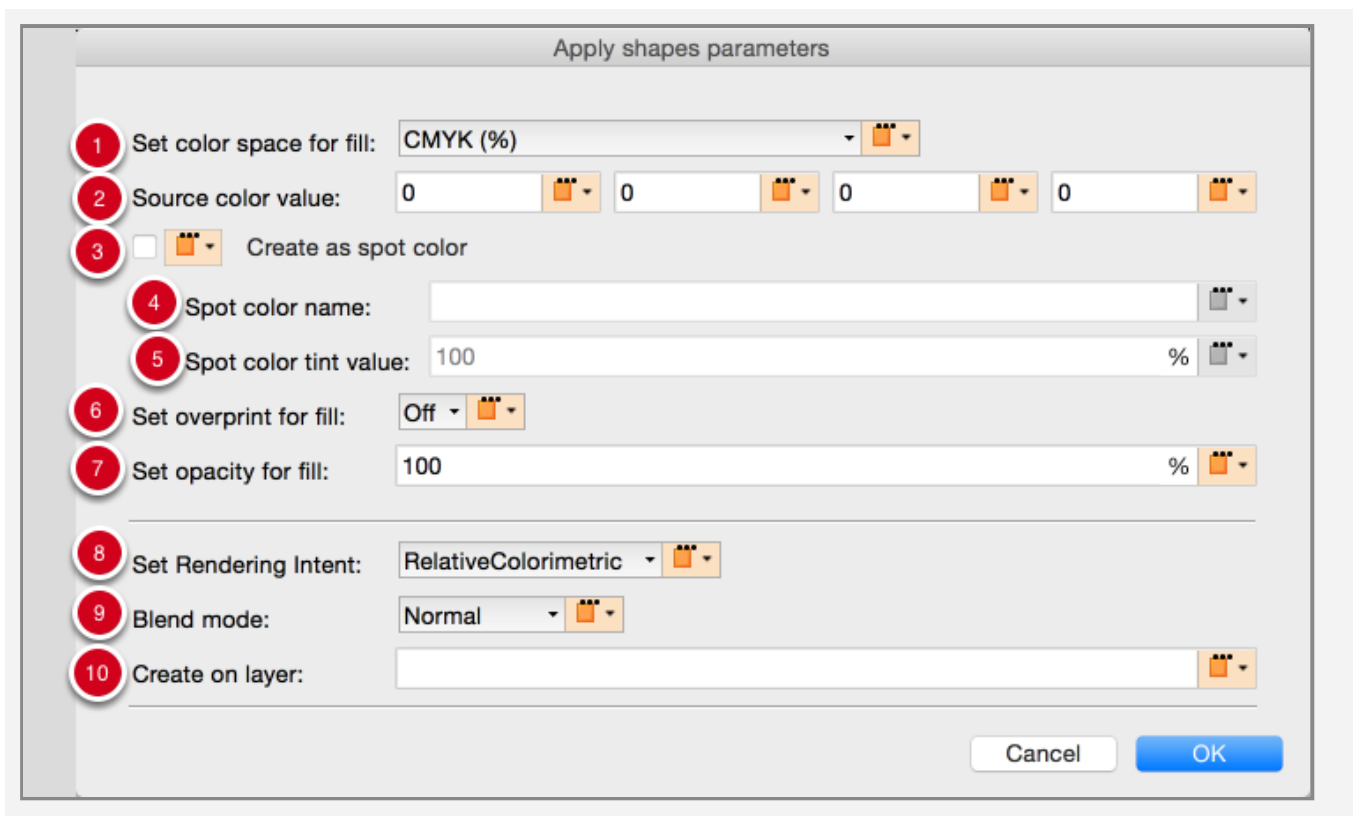


whole page or most of it, the created filled shape will most probably not be visible at all.

The creation of a "Filled shape in front of current page content" means that the defined shape(s) will be inserted into each page as a path object. If the shape consists of several part, it will be created as a path object with as many closed sub-paths. For this path object a fill will be applied as configured under "Parameters".

## Filling shapes in front or behind existing page content

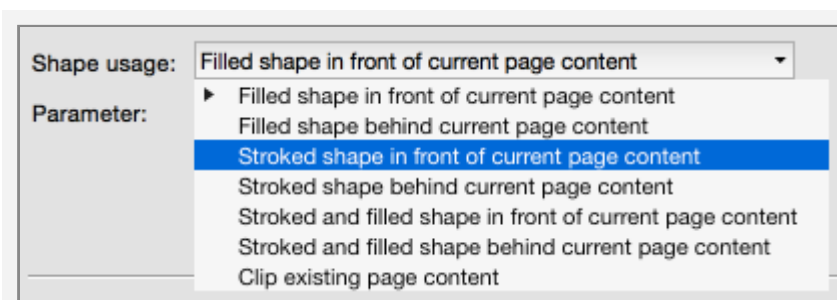
### – Fill parameters



1. Color model to be used for filling the path object; available models are CMYK, RGB, gray, and Lab. For CMYK, RGB and gray, values can be set to be provided in percent (0...100%) or as a number (0.0...1.0). Lab values must always be provided as 0...100 for the L value, and -127...128 for the a and b values.
2. Depending on the chosen color space, one, three or four values have to be provided

3. If this check box is activated, the color used for the fill be create as a spot color, the color model will then be used as the alternate color space for the spot color, and the color values will determine the appearance of a 100% tint value of the spot color
4. Name of the spot color (only enabled if checkbox "Create as spot color" is activated)
5. Tint value to use for filling the path with the spot color (only enabled if checkbox "Create as spot color" is activated)
6. If this checkbox is activated, the fill color will be set to overprint.
7. Sets the opacity for the filled path object, i.e. the degree to which the path object will be transparent or not. A value of 100% means that the object is opaque (not transparent at all), and a value of 0% means the object is fully transparent (which implies that it will not be visible at all).
8. Determines the rendering intent. This will only become relevant when a color conversion is applied at a later stage.
9. Set the transparency blend mode. All 16 blend modes defined in the PDF imaging model are available.
10. If not empty, determines that path object is created on a separate layer, named according to the value in this entry.

## Stroking shapes in front or behind existing page content



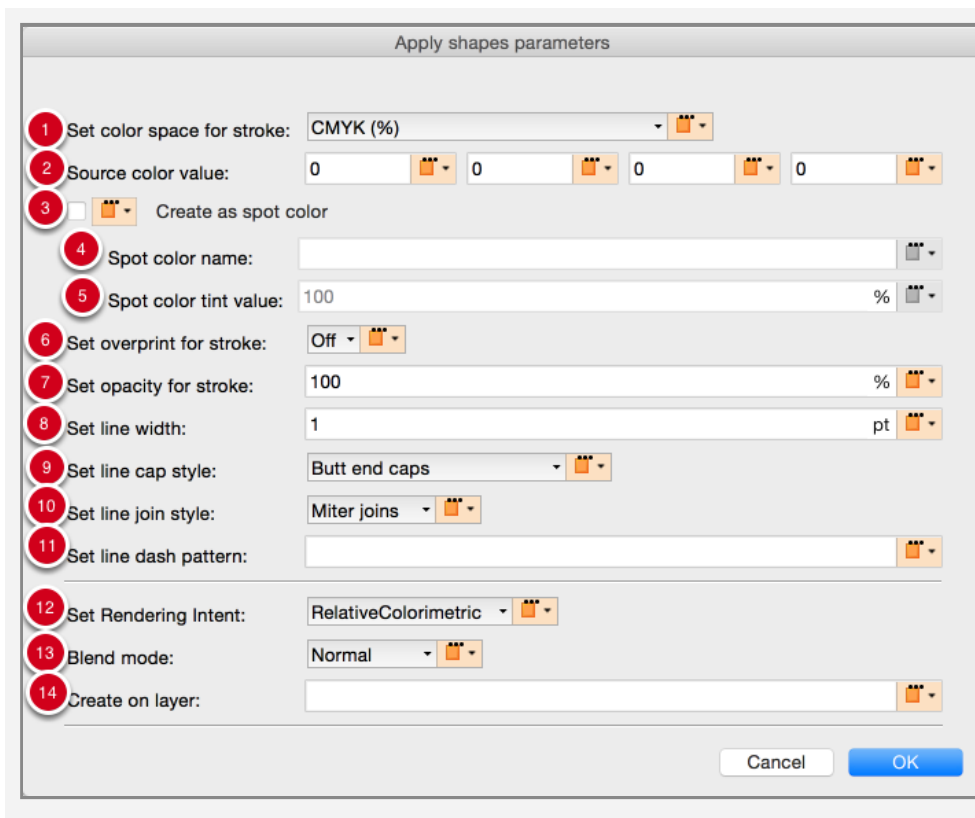
### Important note:

The description below applies in the exact same fashion to stroked shapes created behind current page content. The option to create a stroked shape behind current page content **will only work as expected in very few cases**, as any current

page content that is opaque will hide the stroked shape created behind it. If for example the whole page has an opaque white background object, or any other opaque object that fills the whole page or most of it, the created stroked shape will most probably not be visible at all.

The creation of a "Stroked shape in front of current page content" means that the defined shape(s) will be inserted into each page as a path object. If the shape consists of several part, it will be created as a path object with as many closed sub-paths. For this path object a stroke (also called contour or outline) will be applied as configured under "Parameters".

## Stroking shapes in front or behind existing page content - Stroke parameters



1. Color model to be used for stroking the path object; available models are CMYK, RGB, gray, and Lab. For CMYK, RGB and gray, values can be set to be provided in percent (0...100%) or as a number (0.0...1.0). Lab values must al-

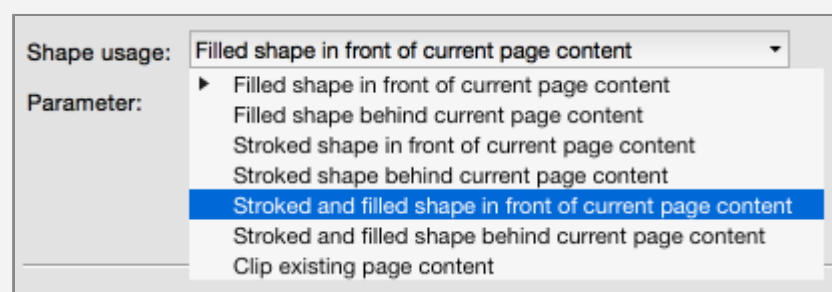
ways be provided as 0...100 for the L value, and -127...128 for the a and b values.

2. Depending on the chosen color space, one, three or four values have to be provided
3. If this check box is activated, the color used for the stroke be create as a spot color, the color model will then be used as the alternate color space for the spot color, and the color values will determine the appearance of a 100% tint value of the spot color
4. Name of the spot color (only enabled if checkbox "Create as spot color" is activated)
5. Tint value to use for stroking the path with the spot color (only enabled if checkbox "Create as spot color" is activated)
6. If this checkbox is activated, the stroke color will be set to overprint.
7. Sets the opacity for the stroked path object, i.e. the degree to which the path object will be transparent or not. A value of 100% means that the object is opaque (not transparent at all), and a value of 0% means the object is fully transparent (which implies that it will not be visible at all).
8. Sets the line width of the stroke in pt.
9. Sets the line cap style for the stroke. This will only have an effect if the stroke is created as a dashed line (see parameter 11). In order to create a dotted line, a suitable line dash pattern needs to be defined, where the part of the dash being painted must have the same length as the line is wide, and the line style must be defined as "Round cap".
10. Sets the line join style This will determine the shape of the line in the corners. Miter joins are ideal for orthogonal corners (e.g. in a rectangle), but can lead to very long pointed corners for corners that are of a sharp angle (this will often be perceived as strange artifacts). For sharp angles it is better to use Bevel join or Round join.
11. Using PDF syntax for dashed lines, this makes it possible to create dashed or dotted lines in many variations. The syntax consists of a sequence of numbers inside one pair of square brackets. Each number determines the length of a line segment that is painted or that is a gap. The first number inside the pair of brackets always defines the length of a painted segment, the second number defines the length of a gap. The third number, defines the length of a painted segment, and so on. Once the sequence of

numbers inside the square brackets have been used up the sequence will start again at the beginning. A simple example would be [3 2] which would lead to a line of painted segments two units long, and gaps between painted segments of a length of 2 units. In order to create an evenly dotted line of 2pt width, use 2 pt for the width of the line, a painted segment length of 0 [sic!] and a gap of 4 units (i.e. a line dash parameter of "[0 4]"), and line cap style of "Round caps". The line join style is irrelevant in this scenario

12. Determines the rendering intent. This will only become relevant when a color conversion is applied at a later stage.
13. Set the transparency blend mode. All 16 blend modes defined in the PDF imaging model are available.
14. If not empty, determines that path object is created on a separate layer, named according to the value in this entry.

## Stroking and filling shapes at the same time in front or behind existing page content



Important note:

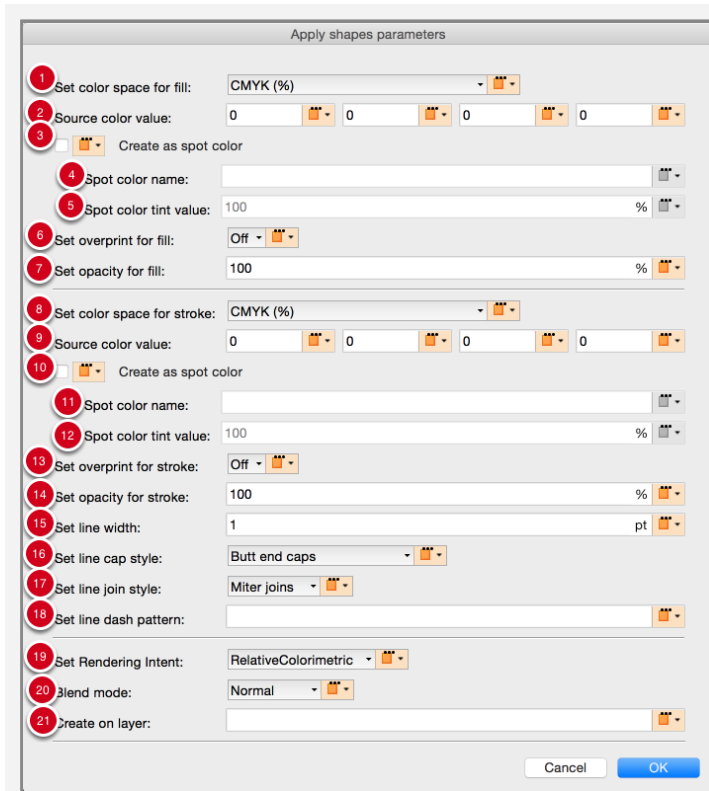
The description below applies in the exact same fashion to stroked and filled shapes created **behind** current page content. The option to create a stroked and filled shape behind current page content **will only work as expected in very few cases**, as any current page content that is opaque will hide the stroked and filled shape created behind it. If for example the whole page has an opaque white background object, or any other opaque object that fills the whole page or most of it, the created stroked and filled shape will most probably not be visible at all.

The creation of a "Stroked and filled shape in front of current page content" means that the defined shape(s) will be inserted into each page as a path object. If the shape consists of several parts, it will be created as a path object with as many closed sub-paths. For this path object a stroke (also called contour or outline) and a fill will be applied as configured under "Parameters". The color for the stroke and the fill can be configured to be different.

According to the PDF imaging model, the fill will always be painted first, followed by the stroke. This implies that the line is guaranteed to always be shown with its indicated line width, half of which will be rendered towards the inside of the path segments, overlaying the fill, with the other half of it being rendered outside of the path segments.

A specific use of this is to define both fill and stroke using the same color definition (preferably a spot color for this to work well), and to use a tint value of 100% for the fill and of 0% for the stroke. If both are set to overprint, they will not affect any content underneath, and the resulting rendered path object will appear to be half of the line width smaller. This can be very useful for example for creating a white background that is slightly smaller than the shape from which it is generated.

## Stroking and filling shapes at the same time in front or behind existing page content - Stroke and fill parameters



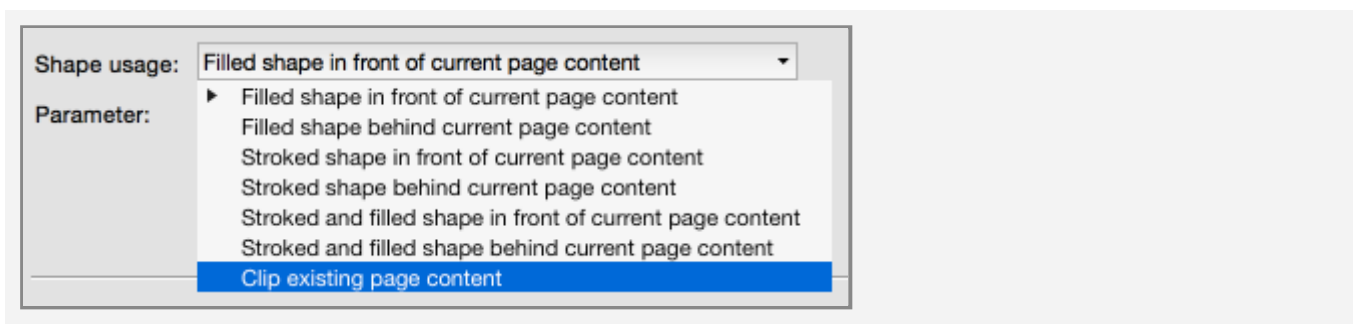
1. Color model to be used for filling the path object; available models are CMYK, RGB, gray, and Lab. For CMYK, RGB and gray, values can be set to be provided in percent (0...100%) or as a number (0.0...1.0). Lab values must always be provided as 0...100 for the L value, and -127...128 for the a and b values.
2. Depending on the chosen color space, one, three or four values have to be provided
3. If this check box is activated, the color used for the fill be create as a spot color, the color model will then be used as the alternate color space for the spot color, and the color values will determine the appearance of a 100% tint value of the spot color
4. Name of the spot color (only enabled if checkbox "Create as spot color" is activated)

5. Tint value to use for filling the path with the spot color (only enabled if checkbox "Create as spot color" is activated)
6. If this checkbox is activated, the fill color will be set to overprint.
7. Sets the opacity for the filled path object, i.e. the degree to which the path object's fill will be transparent or not. A value of 100% means that the object is opaque (not transparent at all), and a value of 0% means the object is fully transparent (which implies that it will not be visible at all).
8. Color model to be used for stroking the path object; available models are CMYK, RGB, gray, and Lab. For CMYK, RGB and gray, values can be set to be provided in percent (0...100%) or as a number (0.0...1.0). Lab values must always be provided as 0...100 for the L value, and -127...128 for the a and b values.
9. Depending on the chosen color space, one, three or four values have to be provided
10. If this check box is activated, the color used for the stroke be created as a spot color, the color model will then be used as the alternate color space for the spot color, and the color values will determine the appearance of a 100% tint value of the spot color
11. Name of the spot color (only enabled if checkbox "Create as spot color" is activated)
12. Tint value to use for stroking the path with the spot color (only enabled if checkbox "Create as spot color" is activated)
13. If this checkbox is activated, the stroke color will be set to overprint.
14. Sets the opacity for the stroked path object, i.e. the degree to which the path object's stroke will be transparent or not. A value of 100% means that the object's stroke is opaque (not transparent at all), and a value of 0% means the object's stroke is fully transparent (which implies that it will not be visible at all).
15. Sets the line width of the stroke in pt.
16. Sets the line cap style for the stroke. This will only have an effect if the stroke is created as a dashed line (see parameter 11). In order to create a dotted line, a suitable line dash pattern needs to be defined, where the part of the dash being painted must have the same length as the line is wide, and the line style must be defined as "Round cap".



17. Sets the line join style This will determine the shape of the line in the corners. Miter joins are ideal for orthogonal corners (e.g. in a rectangle), but can lead to very long pointed corners for corners that are of a sharp angle (this will often be perceived as strange artifacts). For sharp angles it is better to use Bevel join or Round join.
18. Using PDF syntax for dashed lines, this makes it possible to create dashed or dotted lines in many variations. The syntax consists of a sequence of numbers inside one pair of square brackets. Each number determines the length of a line segment that is painted or that is a gap. The first number inside the pair of brackets always defines the length of a painted segment, the second number defines the length of a gap. The third number, defines the length of a painted segment, and so on. Once the sequence of numbers inside the square brackets have been used up the sequence will start again at the beginning. A simple example would be [3 2] which would lead to a line of painted segments two units long, and gaps between painted segments of a length of 2 units. In order to create an evenly dotted line of 2pt width, use 2 pt for the width of the line, a painted segment length of 0 [sic!] and a gap of 4 units (i.e. a line dash parameter of "[0 4]"), and line cap style of "Round caps", The line join style is irrelevant in this scenario
19. Determines the rendering intent. This will only become relevant when a color conversion is applied at a later stage.
20. Set the transparency blend mode. All 16 blend modes defined in the PDF imaging model are available.
21. If not empty, determines that path object is created on a separate layer, named according to the value in this entry.

## Use shape as clipping path for existing page content



This "Shape usage" is a special one that simply uses the defined shape as a clipping path. Depending on whether the shape intent has defined as being "Render only **inside** shape" or "Render only **outside** shape" the clipping path will clip the page content **inside** the path object or the page content **outside** the path object.

There are no configurable parameters for this shape usage.

# "Shapes" features extended in pdfToolbox 9.1

Based on numerous feature requests, the "Shapes" feature, just introduced in pdfToolbox 9.0 in summer 2016, has already been extended in pdfToolbox 9.1, released in November 2016. There are two areas that have been enhanced or added:

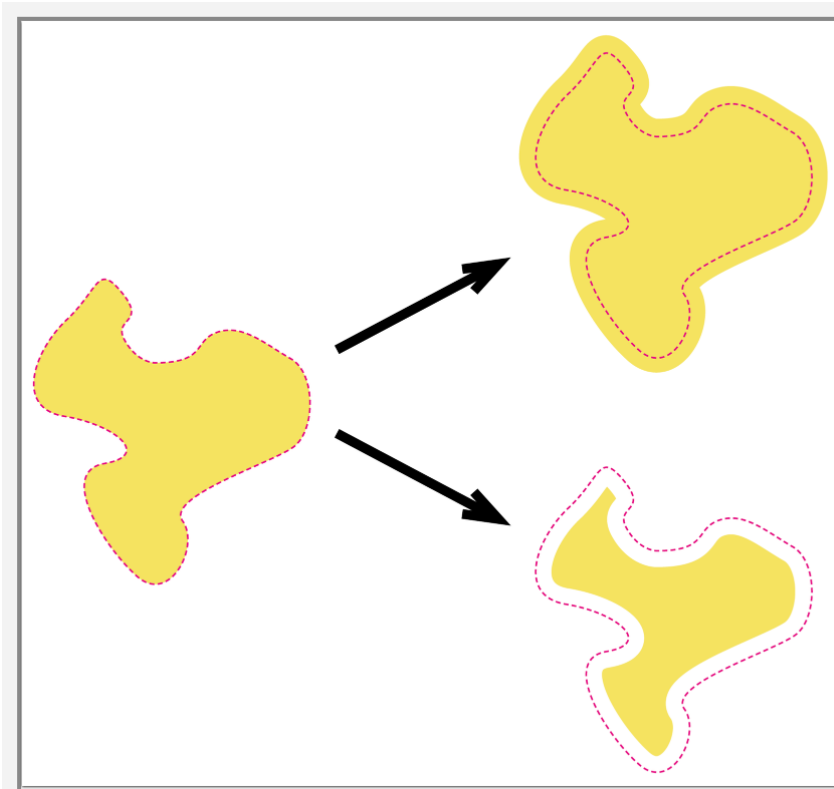
- **enlarging or reducing a shape** by a certain amount now also is available for shapes derived from tracing page content or using existing vector paths (until now, enlarging/reducing was only supported for rectangular shapes based on page geometry boxes or a custom box)
- **"all-inclusive" shape**: it is now possible to merge several nested shapes such that only the outer border of such shapes will be used. For example, for a donut shape, it is sometimes necessary to use the donut shape with the hole in it, and in other cases it is necessary to just use its outer border, i.e. the outer circle, and not take the hole in it into account

## Enlarging or reducing non-rectangular shapes

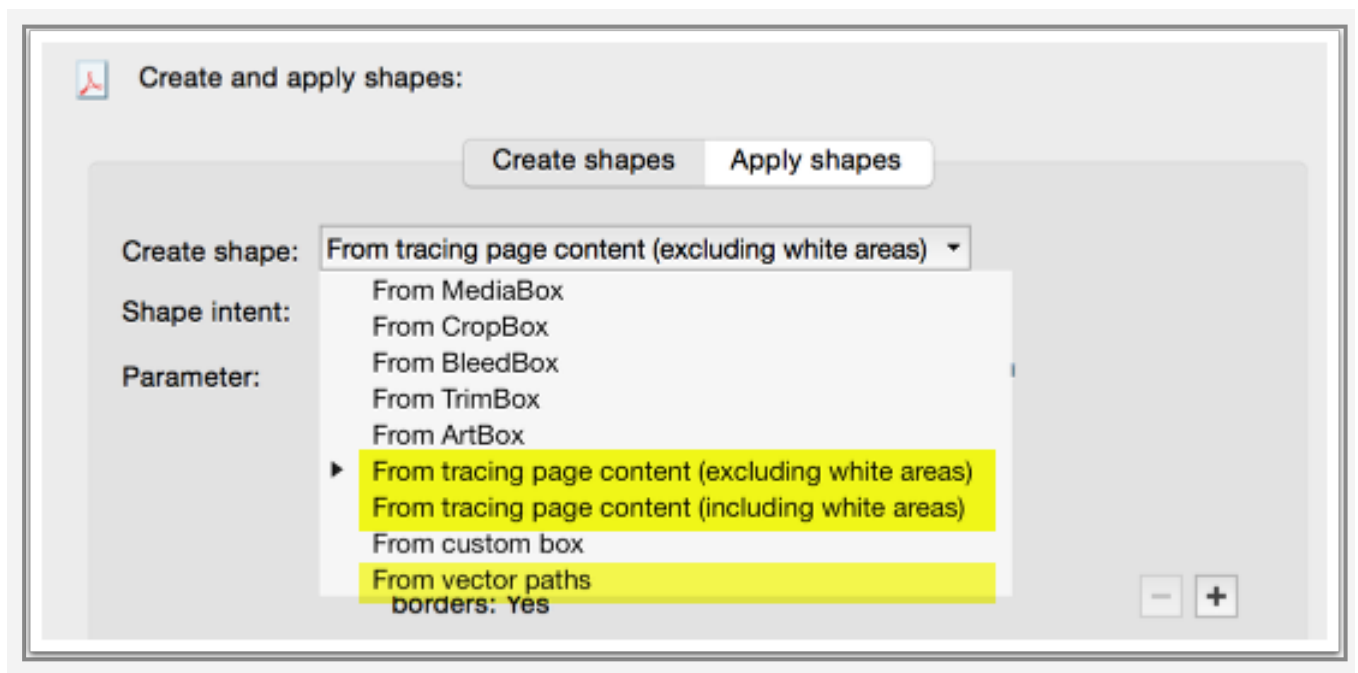
In many cases a shape cannot be directly used as derived from page content. Instead, it needs to be reduced in size by a small amount, or enlarged.

For example, for a white background in label printing on transparent substrate, it could be necessary to print white in all areas of the printed content, but at the same time the white should never become visible itself, e.g. in the case of mis-registration of the colorants during the printing process. Thus it can make sense to reduce the area where white is to be printed by a millimeter or so.

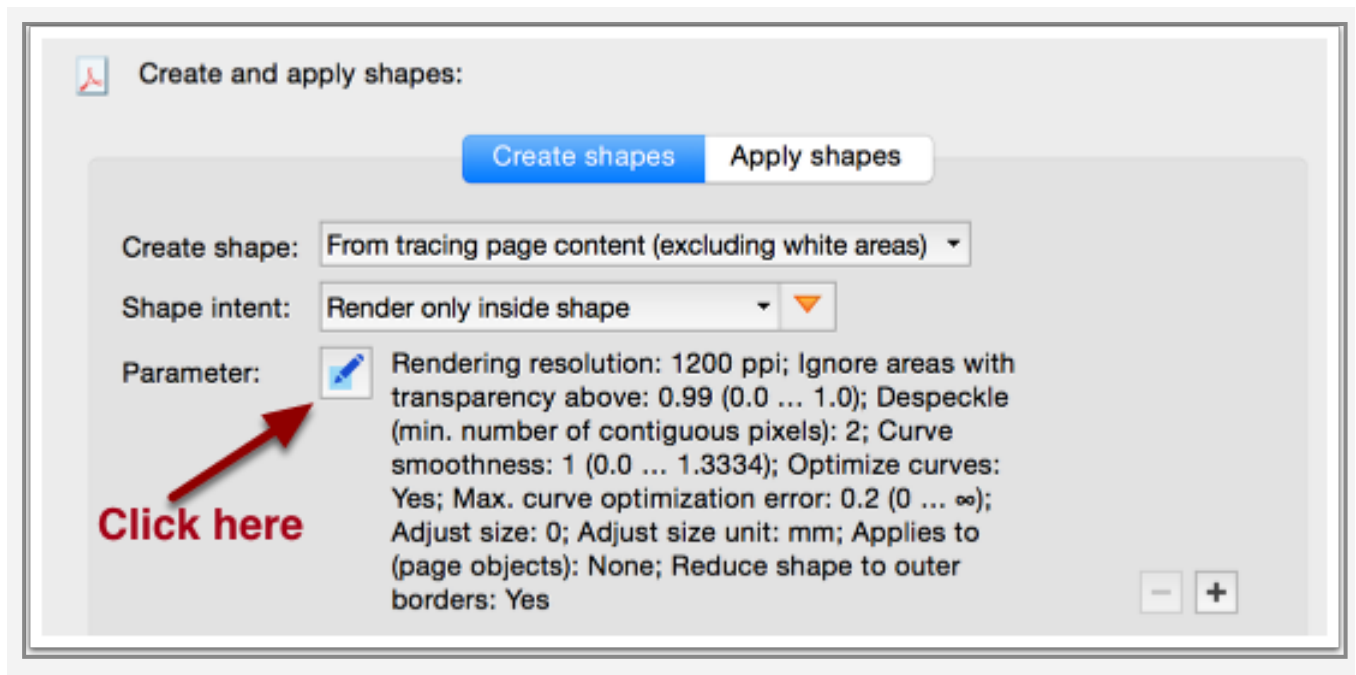
Along the same lines, it can be desirable for varnish to make sure it gets printed in top of all printed page content – and a tiny bit beyond it, to again compensate for possible mis-registration between colorants during the printing process. In this case, the shape would be enlarged by a millimeter or so.



The option to enlarge or reduce a shape's size is now also available for shapes derived from tracing page content or from existing vector paths:



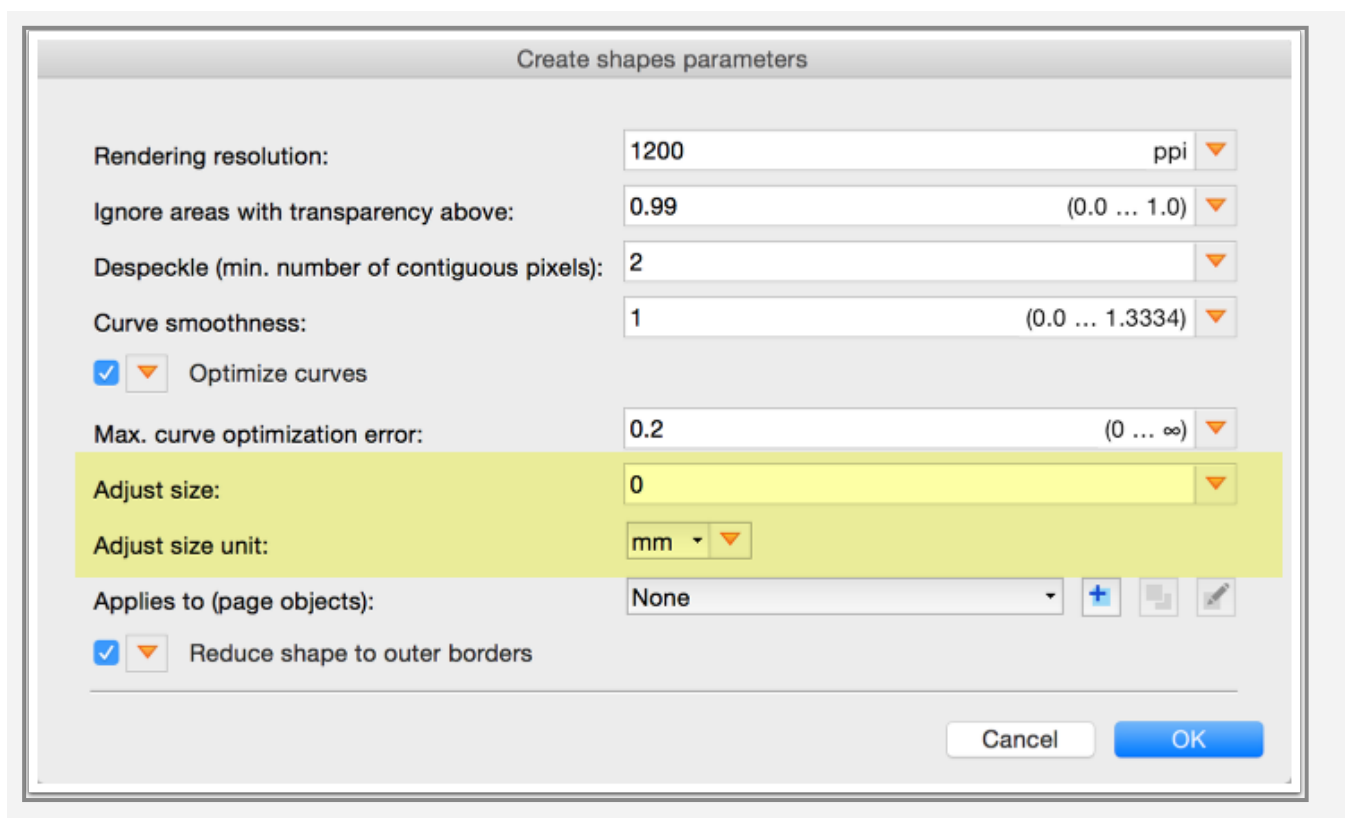
The settings for enlarging or reducing a shape's size can be found inside the "Parameter:" dialog.



The settings for enlarging or reducing a shape's size can be found in the lower half of the "Parameter:" dialog in the form of

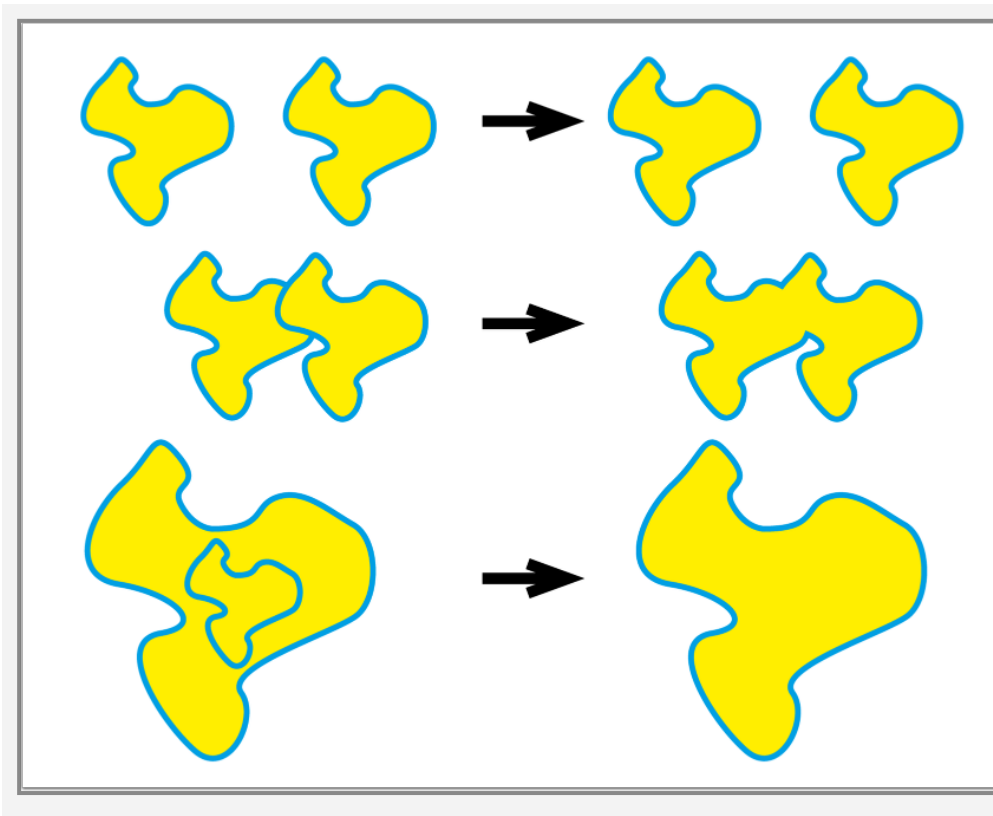
- the "Adjust size" field, and
- the "Adjust size unit" option, supporting "mm" (millimeter), "pt" (point) or "in" (inch) as units

A negative value for "Adjust size" will reduce the shape, whereas a positive value for "Adjust size" will enlarge it.



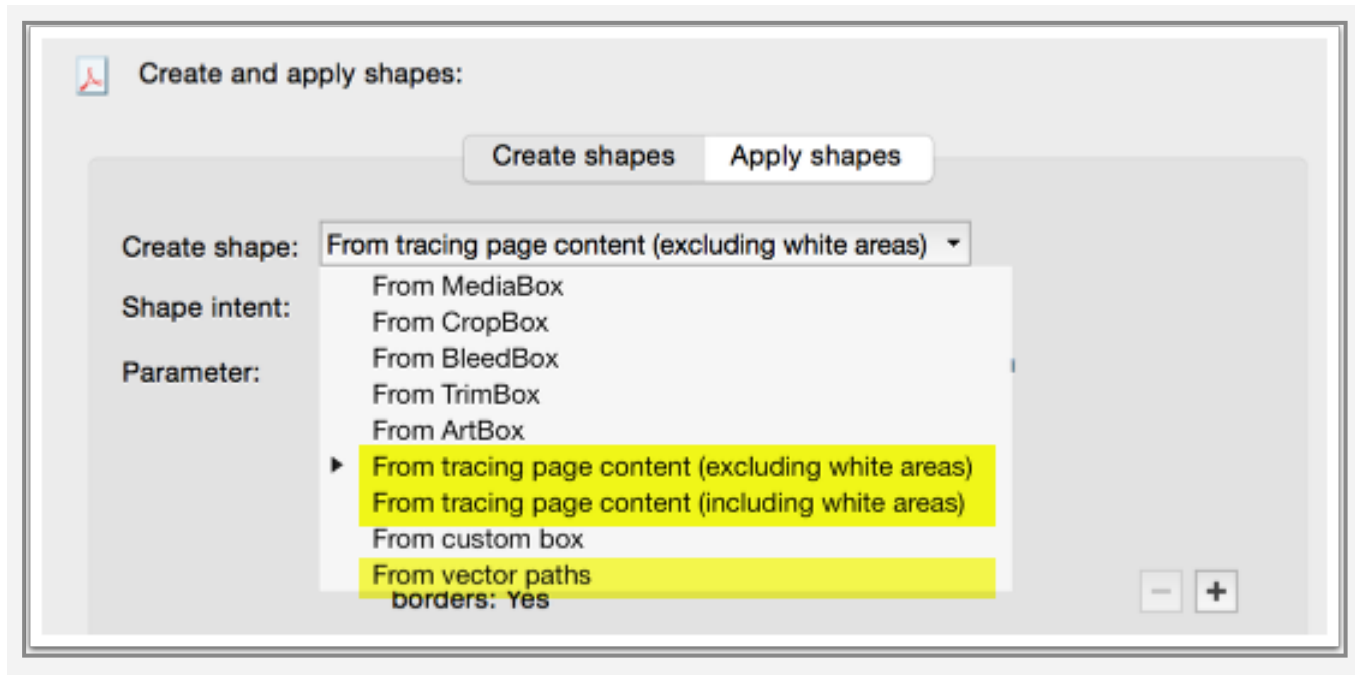
## "All-inclusive" shape: Reduce shape to outer borders

The drawing below illustrates how the new "all-include" option impacts the shape that results from tracing page content or deriving a shape from existing vector paths. The main effect is that any shapes that exist inside other shapes are discarded. Where shapes overlap, the combined area of such overlapping shapes will be used as the actual shape. For shapes that are neither nested nor overlapping, the option has no effect.

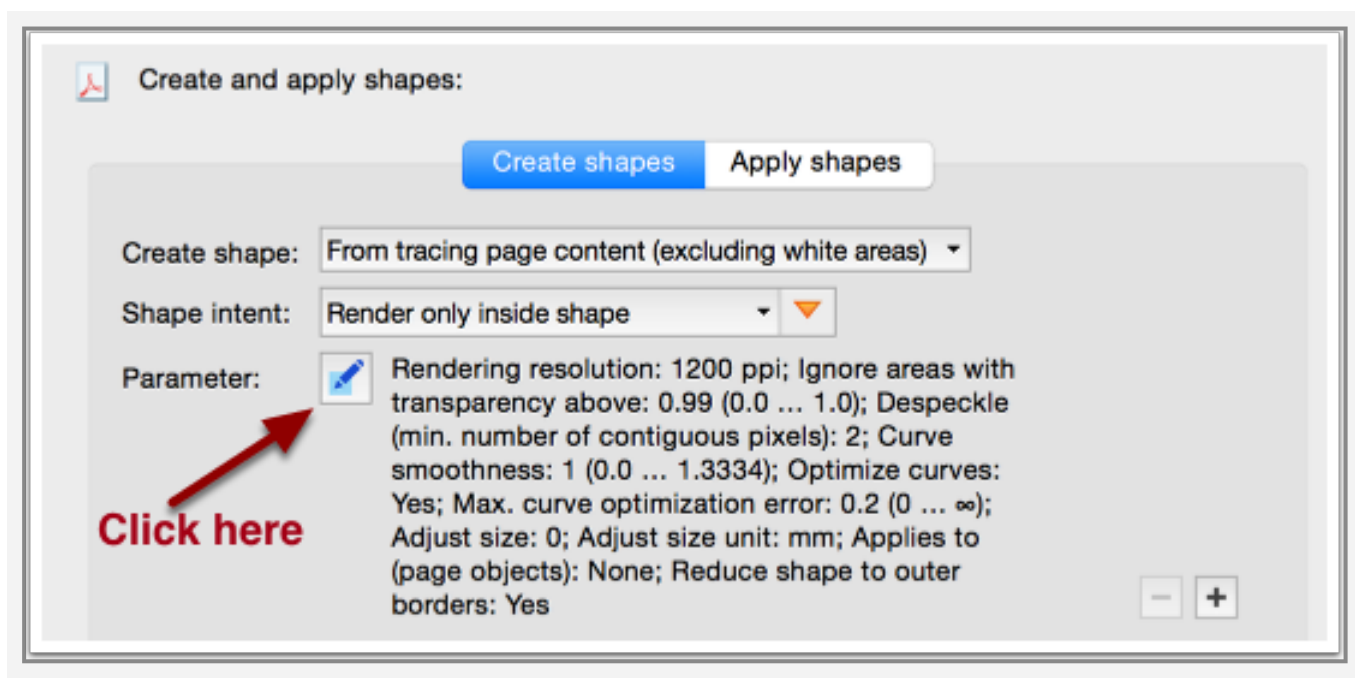


The "Reduce shape to outer borders" is only available for the three "Create shape" variants

- From tracing page content (excluding white area)
- From tracing page content (including white area)
- From vector paths



The "Reduce shape to outer borders" can be found inside the "Parameter:" dialog.



The "Reduce shape to outer borders" can be found at the bottom of the "Parameter:" dialog.



Create shapes parameters

Rendering resolution:	1200	ppi	▼
Ignore areas with transparency above:	0.99	(0.0 ... 1.0)	▼
Despeckle (min. number of contiguous pixels):	2		▼
Curve smoothness:	1	(0.0 ... 1.3334)	▼
<input checked="" type="checkbox"/> <input type="checkbox"/> Optimize curves			
Max. curve optimization error:	0.2	(0 ... ∞)	▼
Adjust size:	0		▼
Adjust size unit:	mm		▼
Applies to (page objects):	None		+ □ ✎
<input checked="" type="checkbox"/> <input type="checkbox"/> Reduce shape to outer borders			

Cancel OK

# Efficiently creating varnish or white background (requires at least v9.1)

In pdfToolbox 9.1, the "Shape" feature has been enhanced. This article shows how to take advantage of the enhancements when creating a varnish or a white background. The two attachments below provide the sample PDF and the pdfToolbox Library containing the fixups used in this article.

## Sample file and pdfTo pdfolbox Library with pre-configured fixups



donuts.pdf



Shapes-\_creating\_varnish\_or\_white\_background.kfpl

## Growing or shrinking a shape

When creating a **partial varnish** it is often desirable to derive the area where varnish shall be applied from actually printed content. At the same time, mostly in order to compensate for less than perfect registration of plates or print heads, it is usually necessary **that the varnish extends slightly, maybe by a millimeter or two, beyond the printed content area**, to ensure that printed content is always varnished. The fact, that some small area where nothing is printed also receives varnish, is typically not considered a problem.

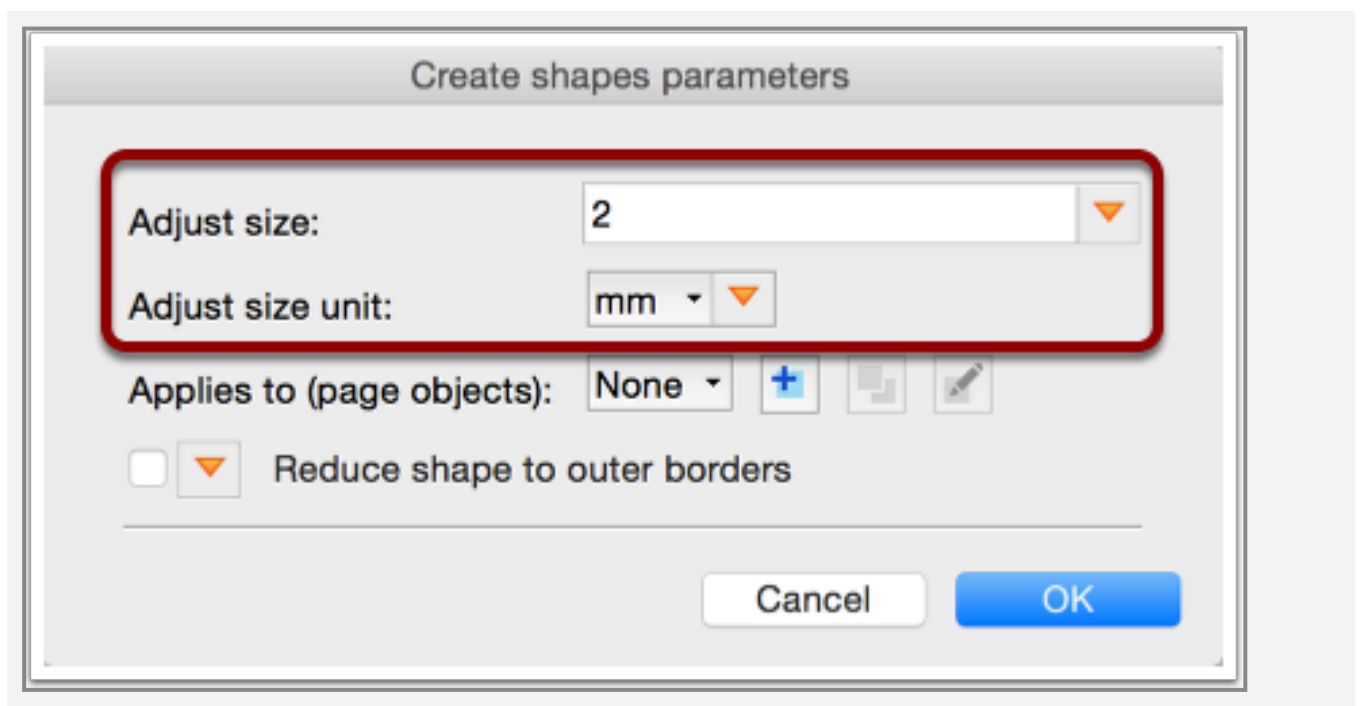
Along the same lines, but usually in the other direction, a white background may have to be created for printing on transparent substrate – but as the background shall not become visible as such, it needs to be shrunk by a little bit to pull back from the border of the print content area.

Both requirements can now be met very easily by the extended "Shape" feature, using the "adjust size" setting. This setting was already available for shapes based on page geome-

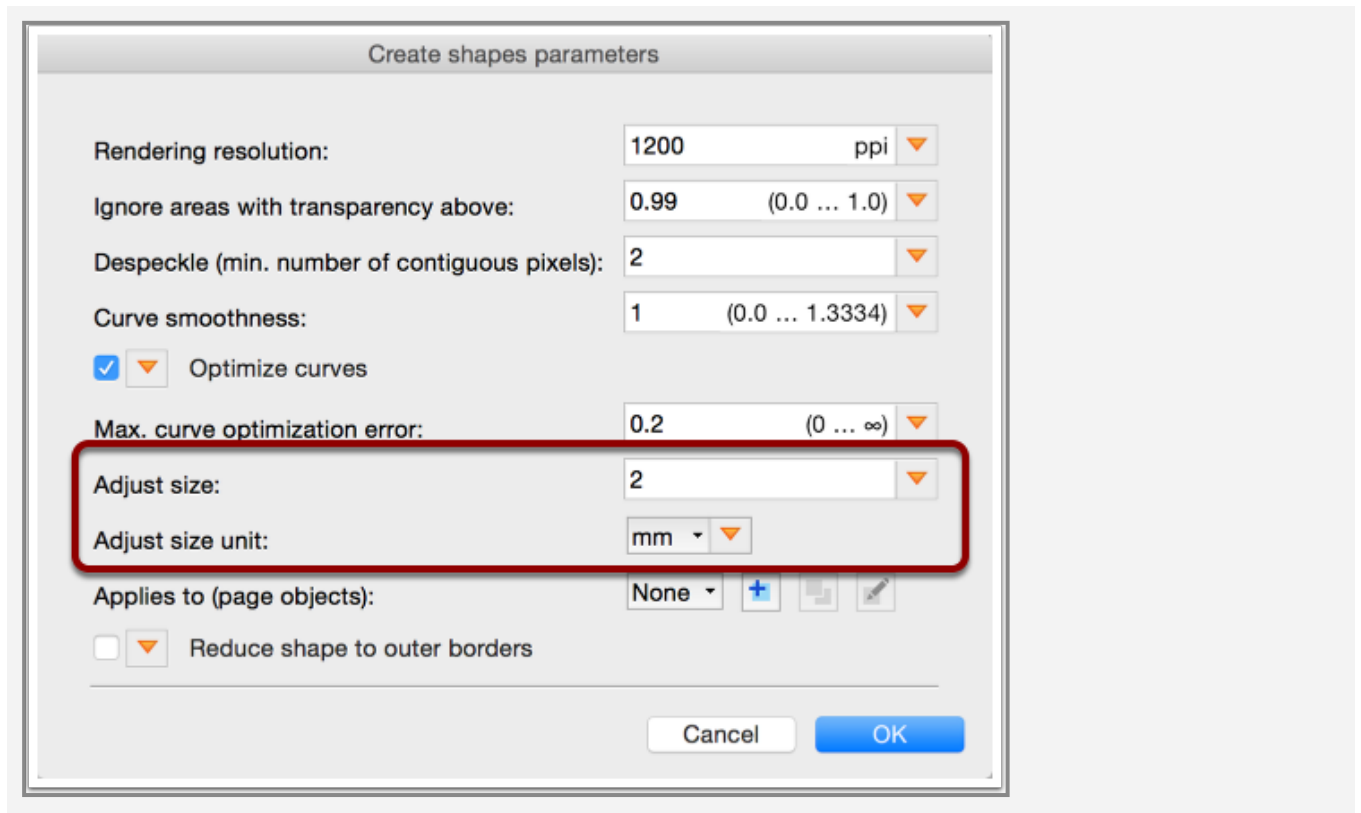
try boxes or a custom box but now can also be used for shapes derived from tracing page content or from existing vector paths.

In addition, a new setting "Reduce shape to outer borders" makes it possible to include 'holes' inside print content areas, which can be very handy for the generation of a white background (see example below). Extending/shrinking shapes and 'reduce to outer border' can be freely combined with each other and any of the other options.

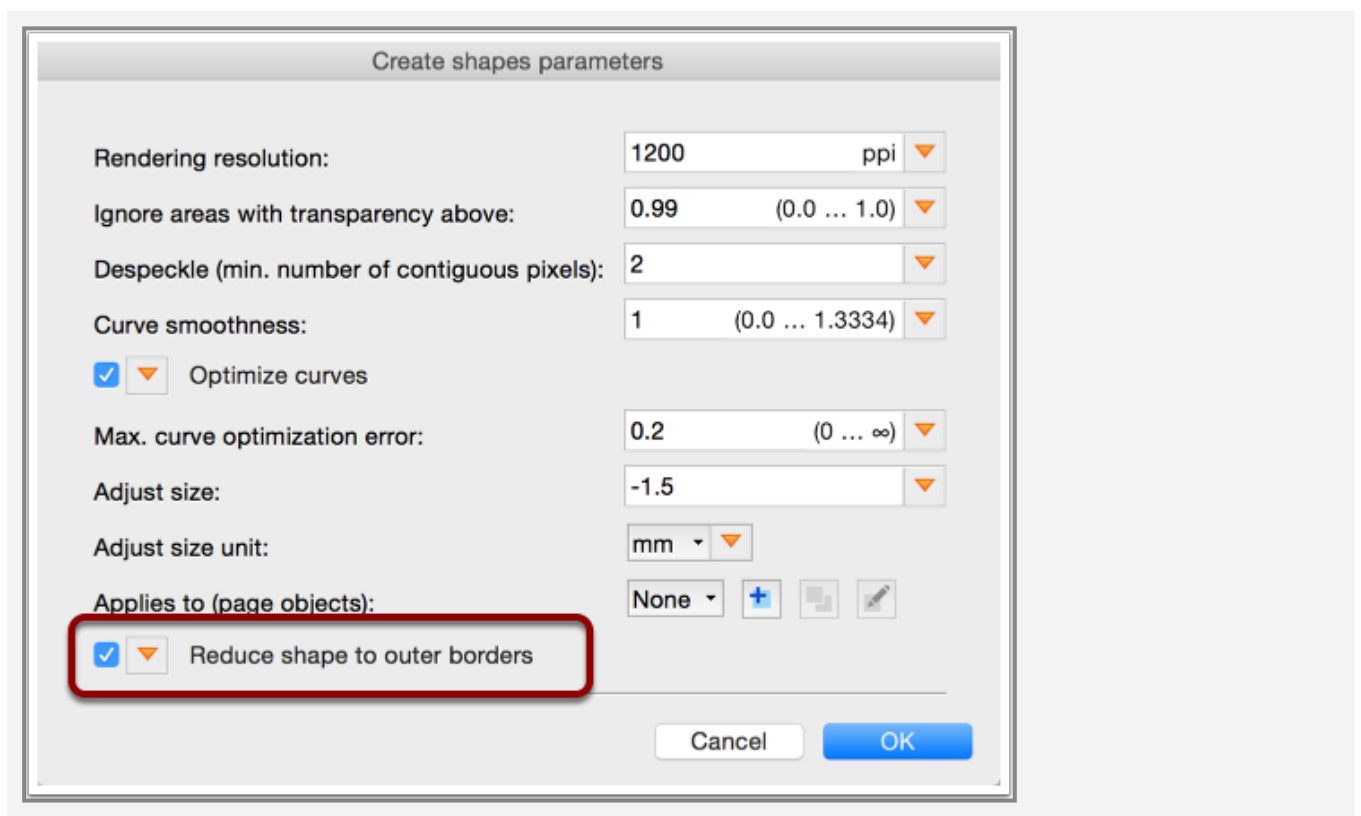
Create shape parameters for "From vector paths" option, with highlighted "Adjust size" settings:



Create shape parameters for "From tracing page content" option, with highlighted "Adjust size" settings:

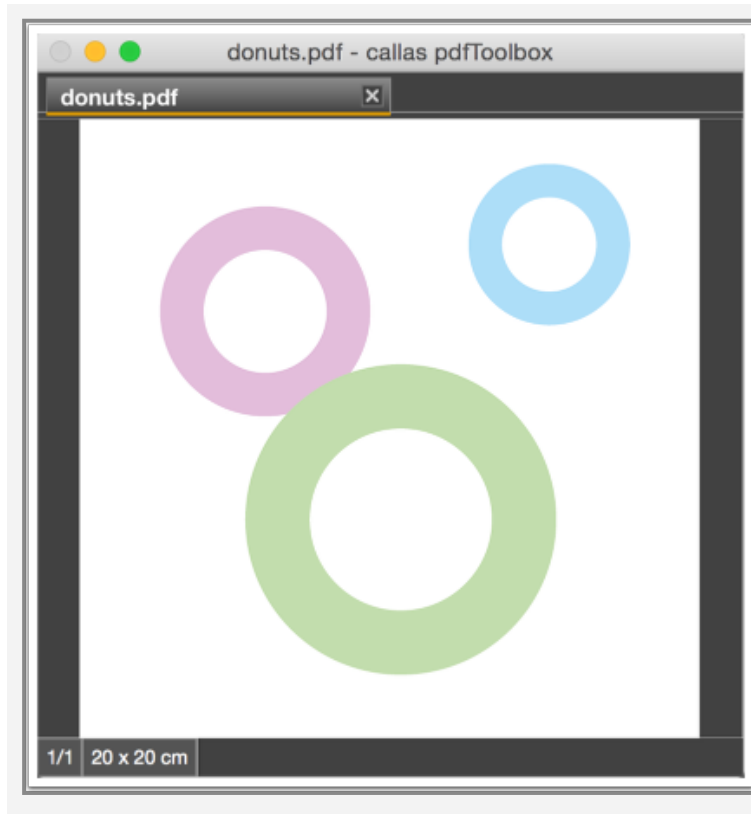


"Reduce shape to outer borders" setting:

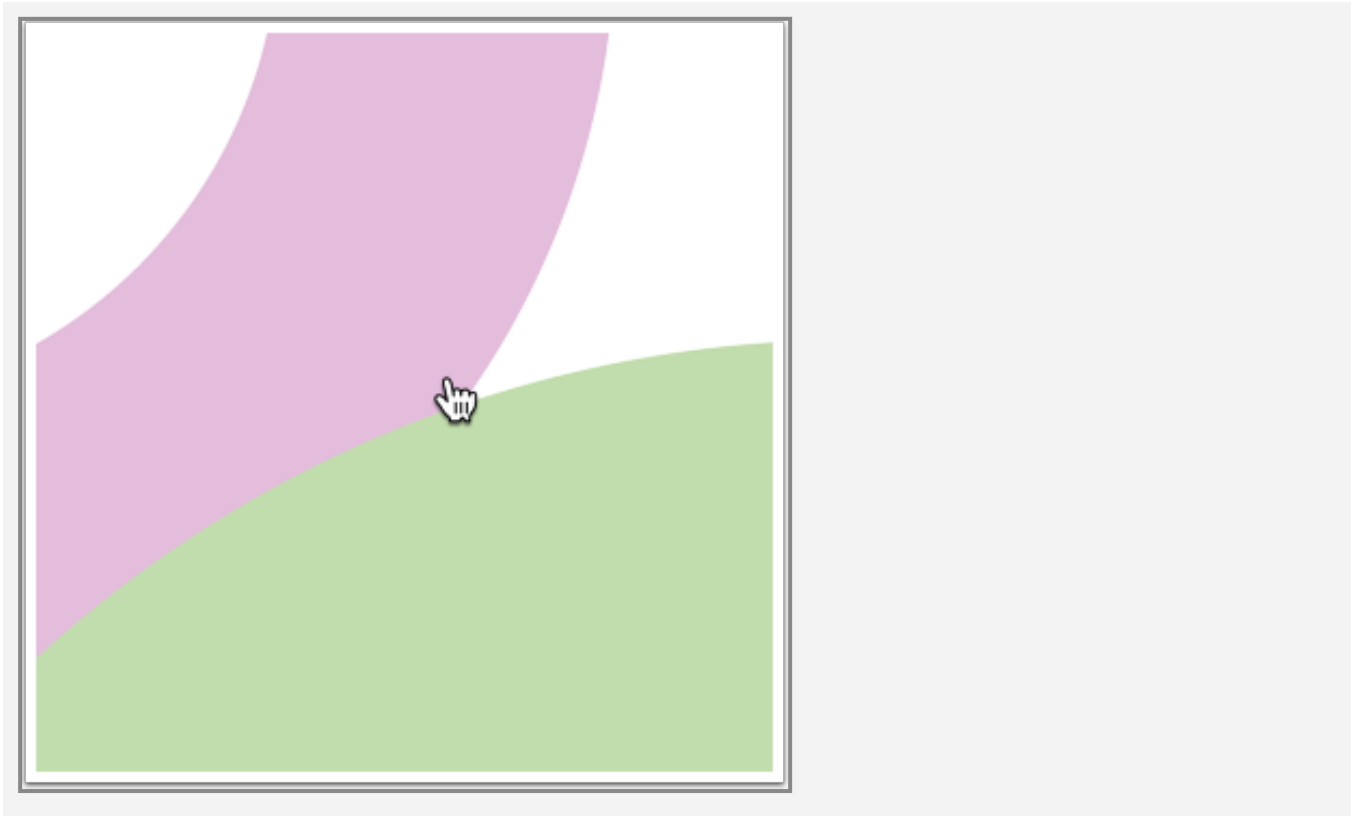


## Example: Extending varnish

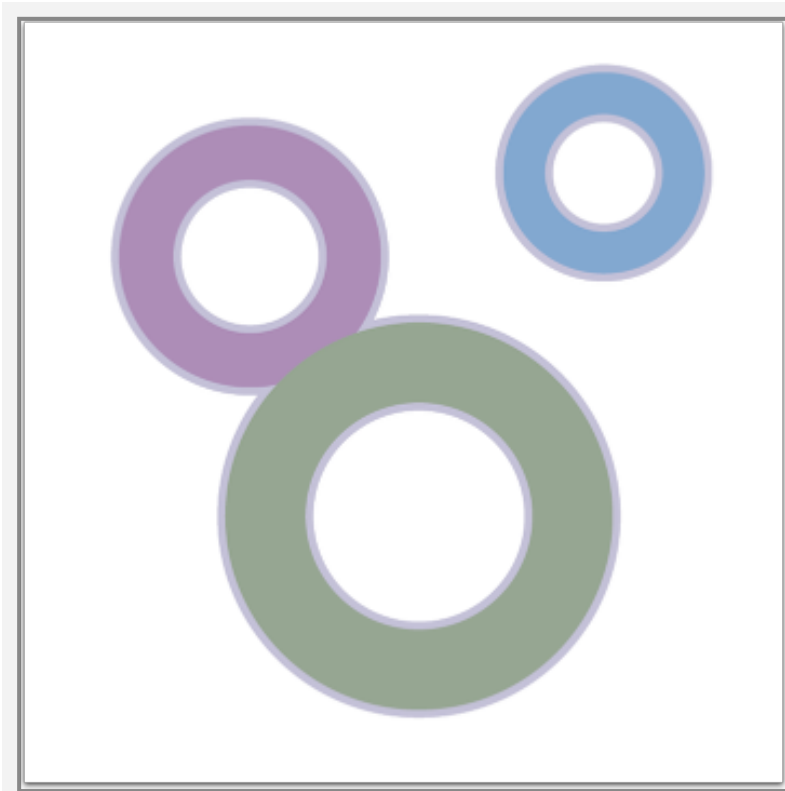
Sample file (see above for download):



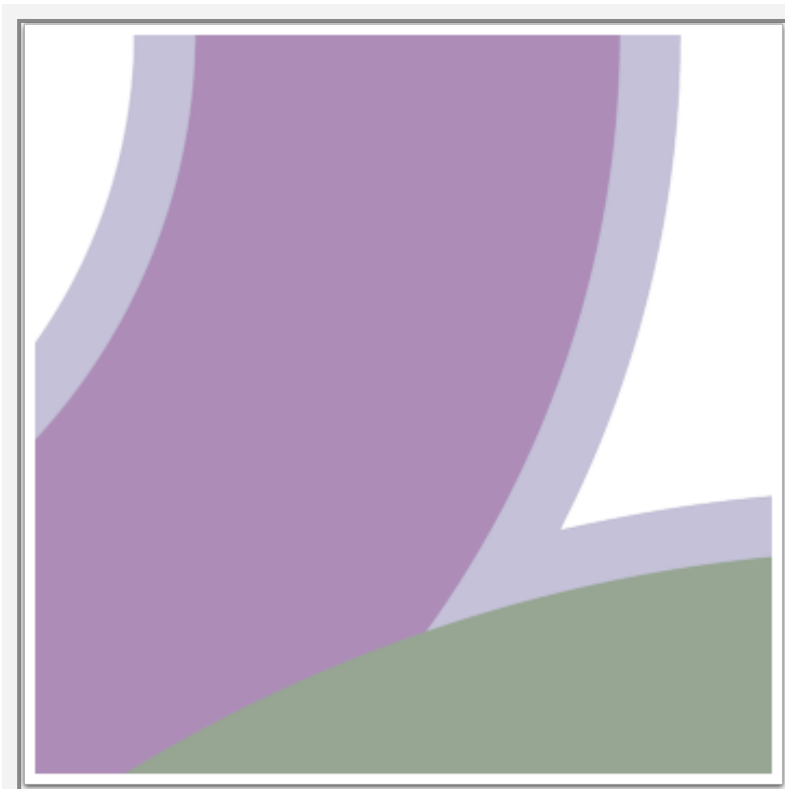
Sample file (see above for download, enlarged detail):



Varnish applied to all page objects based on their vector paths, after applying "Varnish over print objects +2mm (vector)":

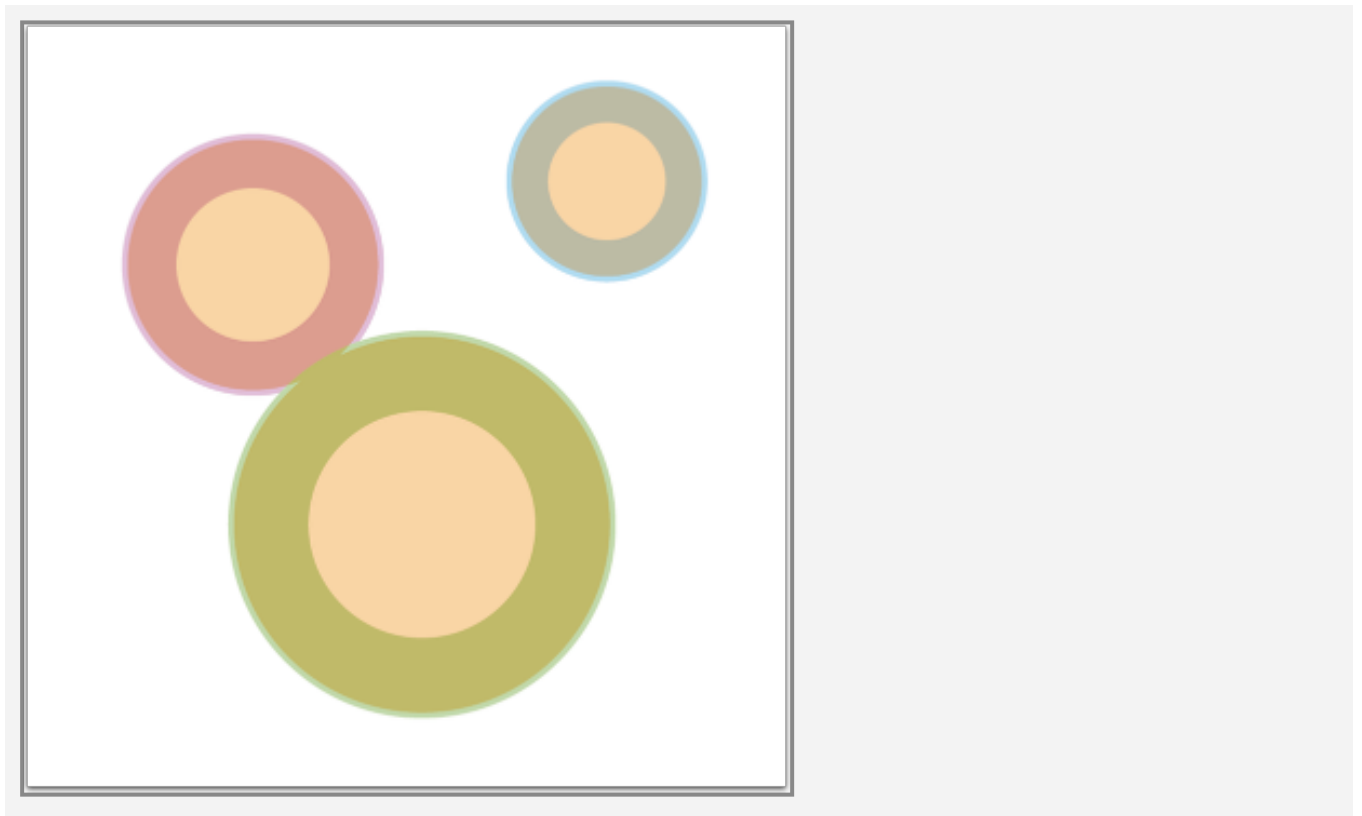


Varnish applied to all page objects based on their vector paths (enlarged detail):



## Example: Shrinking white background, after reducing shape to outer border

White background derived from all page objects based on their vector paths, using only the outer border of all paths, using "White backing under print objects -1.5mm, based on outer border (vector)":



White background derived from all page objects based on their vector paths, using only the outer border of all paths (enlarged detail):



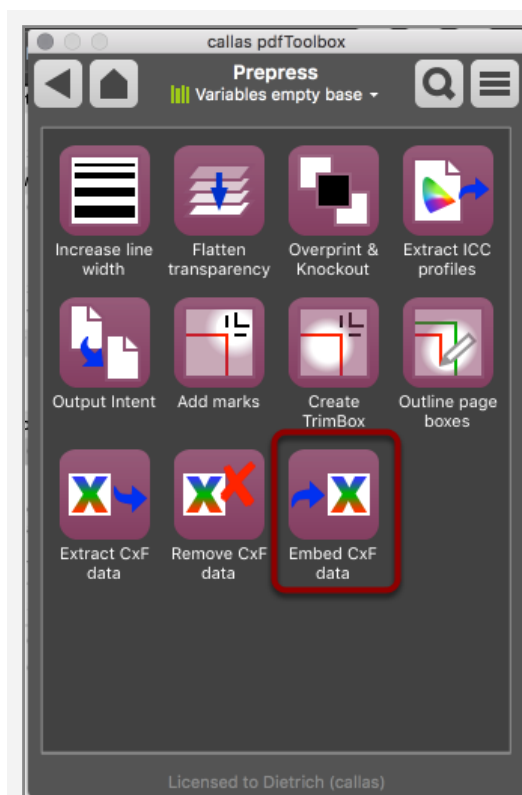


# Spectral color and CxF

# Embed CxF data (import)

In order to import CxF information CxF XML files need to be present in a folder and their name has to be the spot color name that it represents.

## Open Switchboard -> Prepress -> Embed CxF data

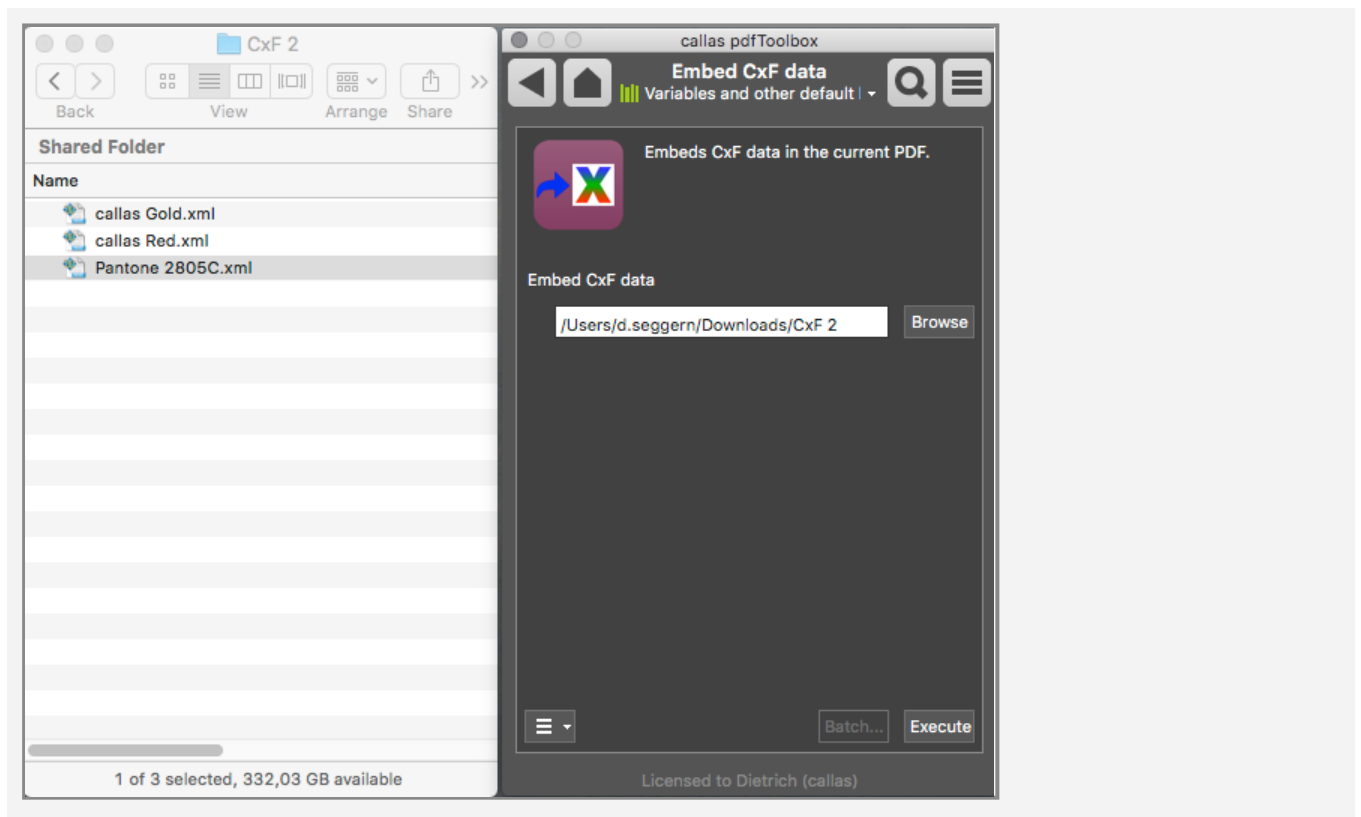


## Open a PDF/X file



The PDF needs to be a PDF/X file or has to have at least an PDF/X Output Intent entry, since the CxF information is being embedded into the Output Intent entry. If there is no Output Intent entry present the "Execute" button in pdfToolbox cannot be hit.

## Select a folder



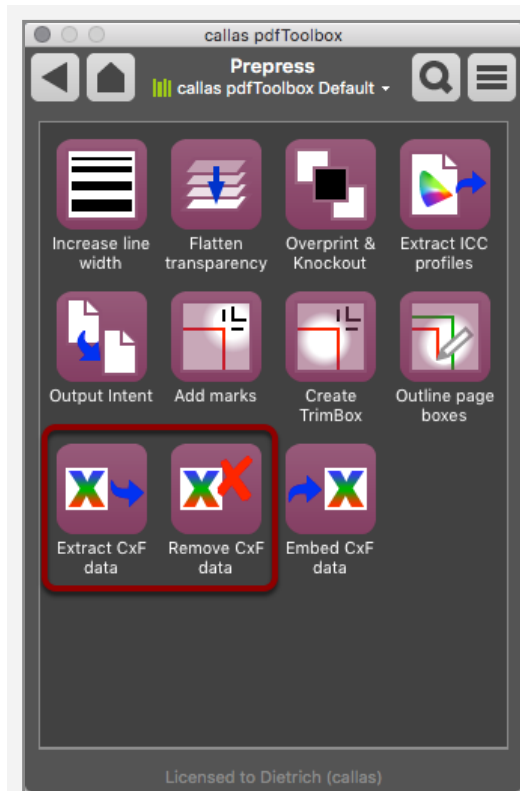
Click on Browse and select a folder that contains CxF XML files. Click on Execute in order to embed the CxF XML files.

**The presence of CxF information in the result PDF is indicated by a CxF button at the bottom of the pdfToolbox window**



# Extract and remove CxF information

**In order to extract or remove CxF information go to Switchboard -> Prepress**



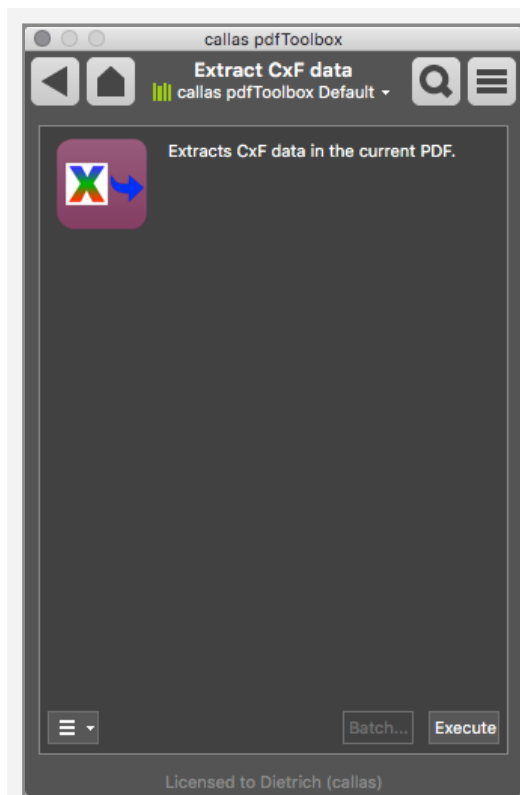
## Open a PDF that has CxF information attached



The presence of CxF information is indicated at the bottom of the pdfToolbox window.

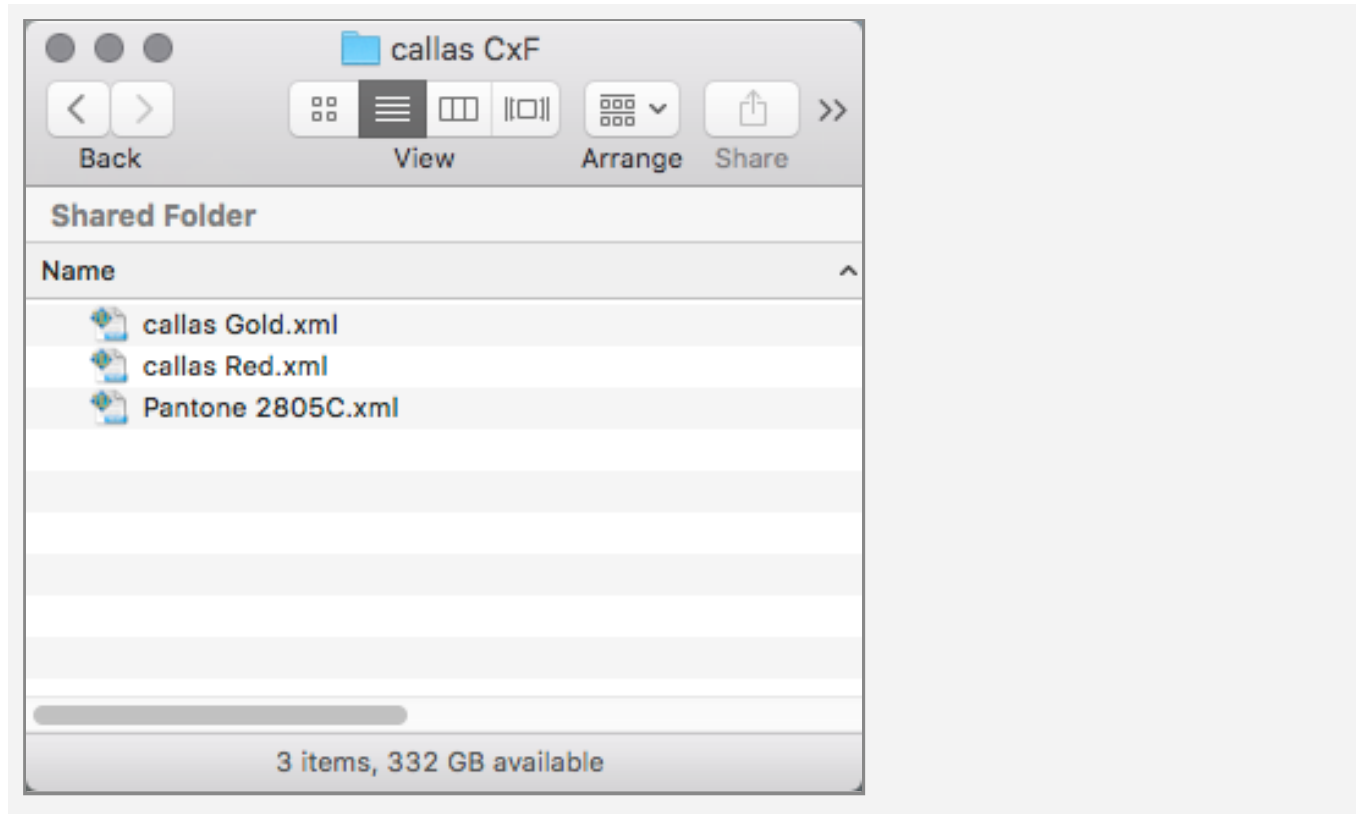


## Extracting CxF data

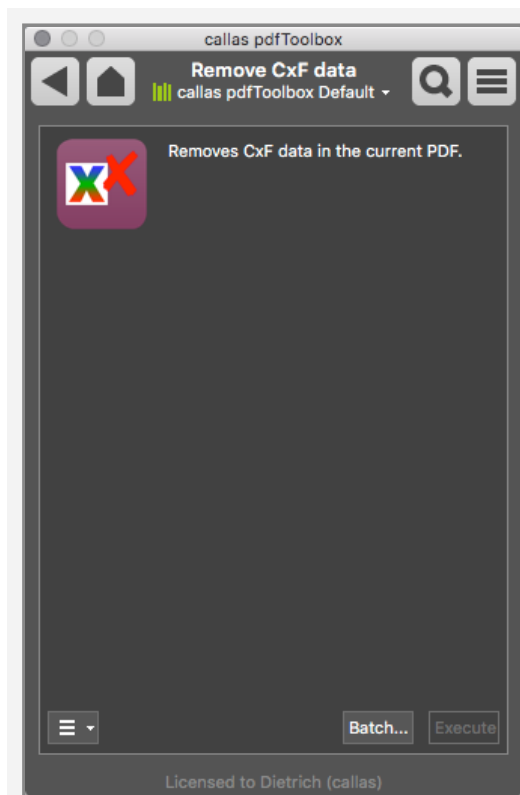


If you extract CxF data you are asked for a folder in your file system. For each CxF information in the PDF an XML file is created in that folder that has the name of the spot color that it represents.

## A folder with CxF information as extracted from a PDF file



## Removing CxF data from a PDF



You will be asked for a location to save the new PDF to.

## The CxF indicator disappears from the pdfToolbox window



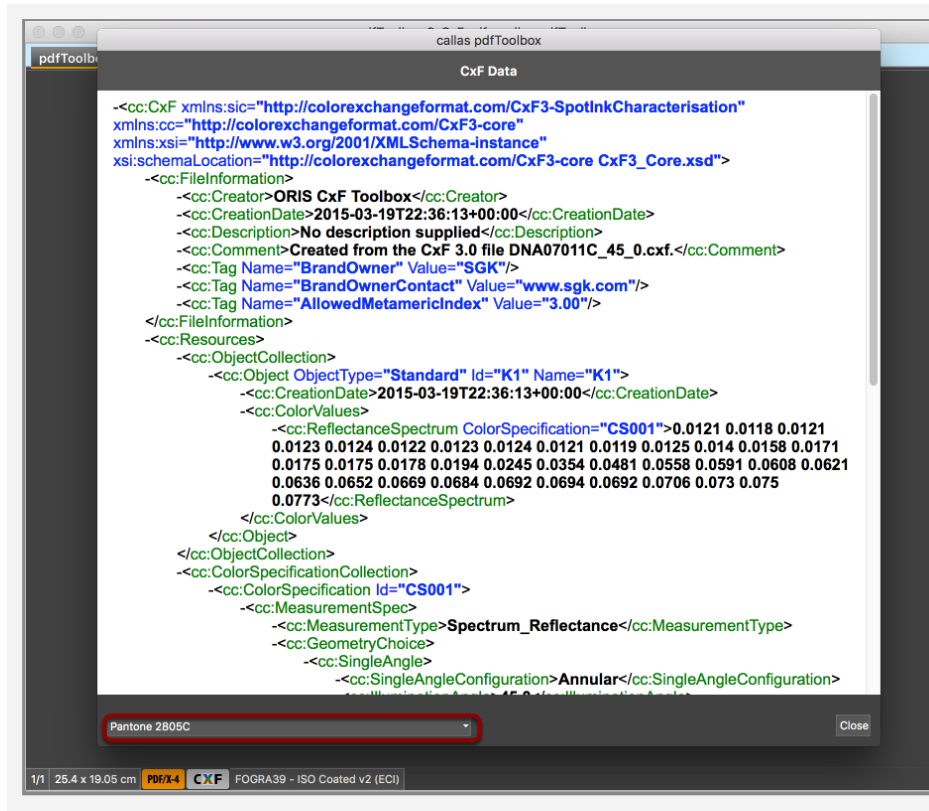
# Analyze CxF information

**Analyzing CxF information in a PDF file is easily possible...**



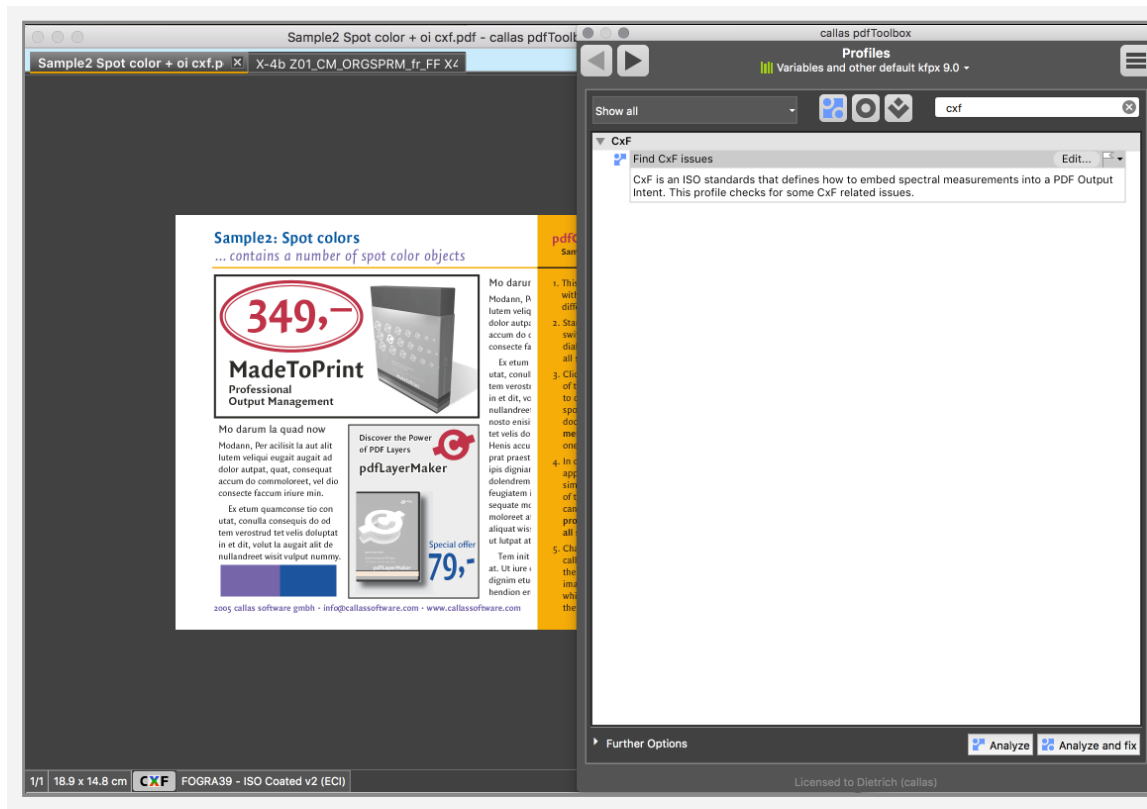
...by clicking on the CxF indicator at the bottom of the window.

## A windows opens that displays the CxF data for the first spot color in its XML structure



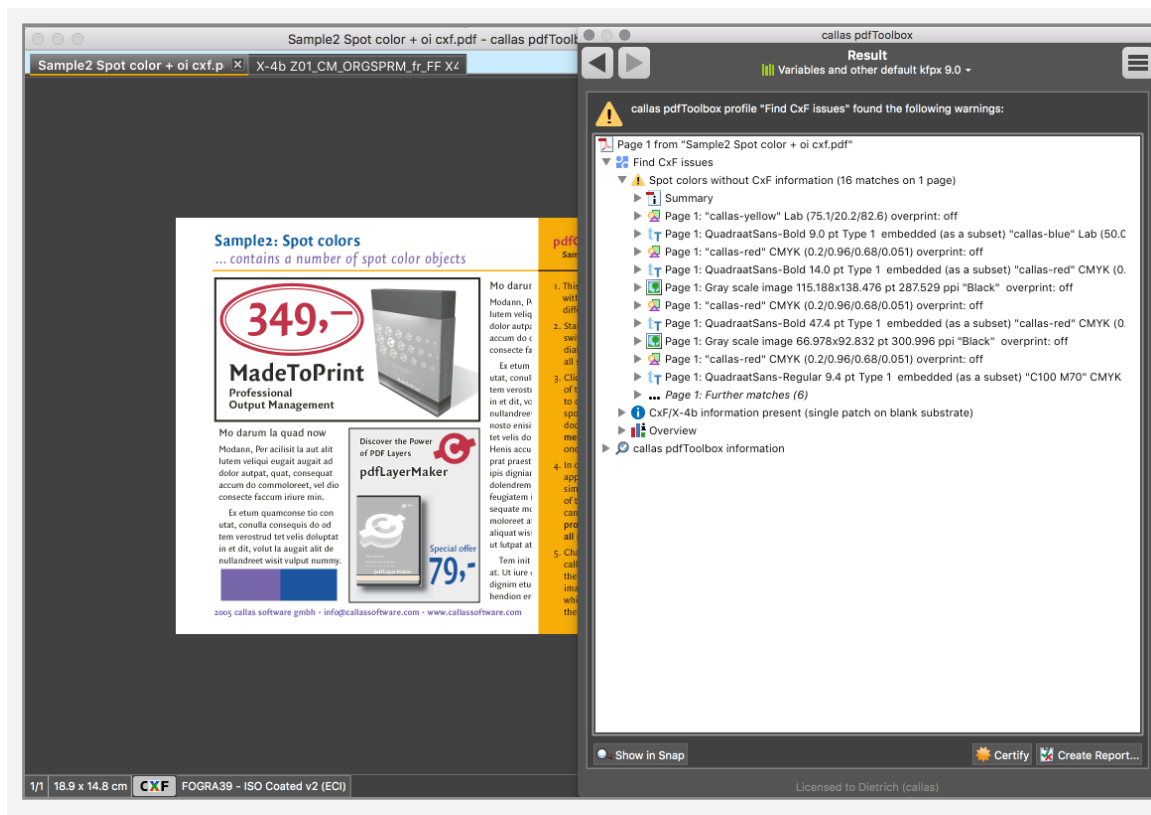
A pop up at the bottom of the windows allows you to select the spot color that you want to see.

**It is also possible to run a profile that checks for various parameters of a PDF in combination with the embedded CxF information**



Open the Profiles window and search for "CxF"

## The CxF information profile "Find CxF issues"...



... reports for example if a PDF uses spot colors for which no CxF information is present.



# Introduction: CxF and spectral data

CxF is an ISO standard that has been published in June 2015 as ISO 17972-4. To be more precise it is a series of standards from which part 4 has been published first. CxF/X-4 covers spot colors, the other standard parts will cover other color, like process colors etc.

CxF stands for Color Exchange Format and allows for embedding spectral data (measurements) into a PDF file. The presence of such data can potentially improve results when colors have to be simulated on a device that does not have that colorant. That obviously makes the most sense for spot colors, e.g. when they have to be printed on digital printer like on an ink jet machine. The same is true when the spot color has to be proofed.

CxF/X-4 defines 3 conformance levels:

- CxF/X-4b - is the least demanding level and requires a measurement for a single solid (100%) spot color patch.
- CxF/X-4a - requires a minimum of 3 measurements (3 spot color patches), recommended are a total of 11 measurements.
- CxF/X-4 - requires that patches on black background have to be measured in addition to the measurements on white background which are the same as for CxF/X-4a . Again 3 measurements are required - on white and black substrate (a total of 6), but recommended are 11 (22). Printing on black background shows how that spot color appears when printed in conjunction with other colors at the same spot of the substrate.

# **New and extended properties**

# New and enhanced Properties in 9.0

To check for number of hits generated by other Properties within the same Check, group "Pages":

- Number of hits in this check

Related to CxF (Spectral data information in PDF), all in group "Output Intents for PDF/X":

- Number of CxF entries
- Number of process colorants without CxF entry
- Number of spot colorants without CxF entry
- Number of colorants without CxF entry
- Number of stray CxF entries
- CxF entry conforms to CxF/X-4 XML schema
- CxF conformance level is CxF/X-4
- CxF conformance level is CxF/X-4a
- CxF conformance level is CxF/X-4b
- CxF entry present for this colorant name
- Spot color is present in CxF and in MixingHints/Solidities

For the groups "Colors", "Output Intents for PDF/X", "Output Intents for PDF/A" and "Output Intents for PDF/E":

- Number of components in ICC profile dictionaries N entry does not match ICC profile

# New and enhanced Properties in 9.1

## New Properties

- Rotation of text:  
To detect rotated text
- Is in custom area  
Can be combined with other Properties to detect if objects are within a defined area or not.

## New Properties related to Processing Steps

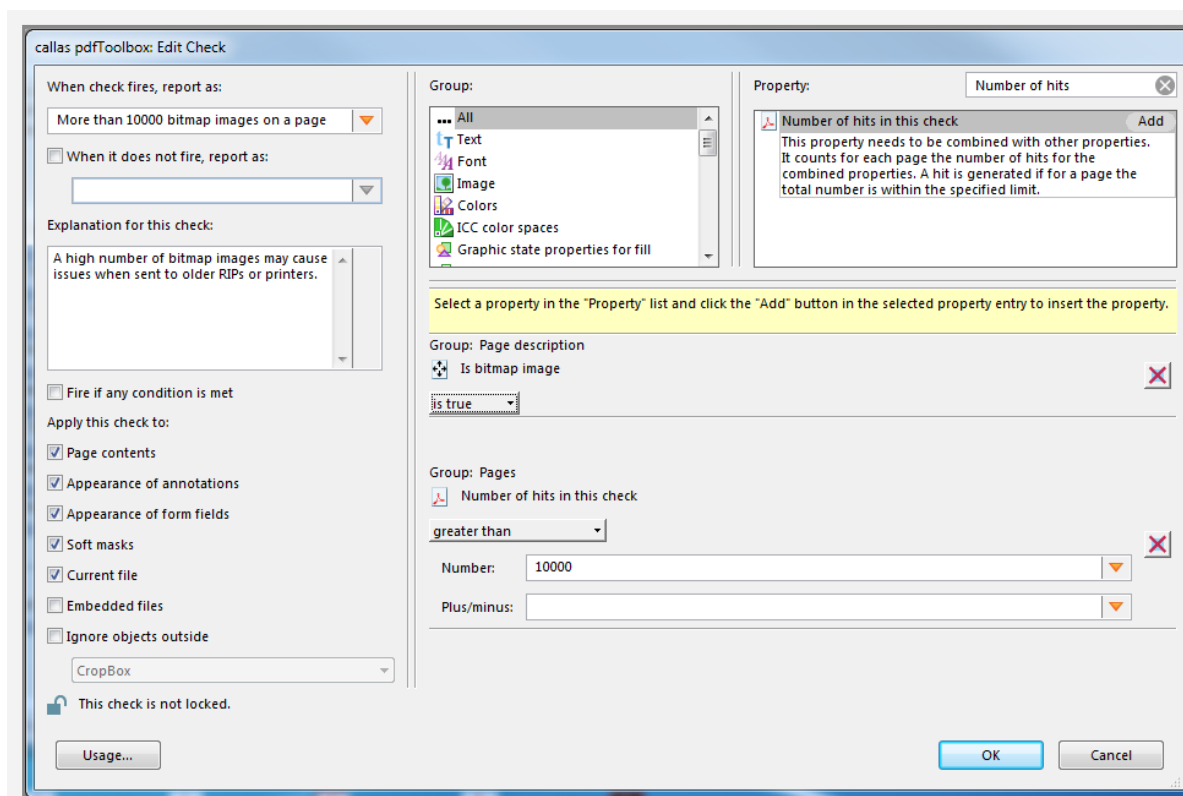
- Processing Steps metadata present
- Same Processing Steps metadata used for more than one layer
- Processing Steps
- Processing Steps metadata uses custom values
- Layer metadata (extended)

# How to use the "Number of hits in the check" property (9.0)

Starting with pdfToolbox 9, a new property can be added to checks to define the number of hits which are needed for the respective Check. "Number of hits in this check" has to be combined with other properties and counts for each page the number of hits for the combined properties. A hit will only be generated for a page if the number of hits matches the defined settings of this property.

This property works object-based and per page, so a combination with e.g. a document or PDF/X property will not give a proper result.

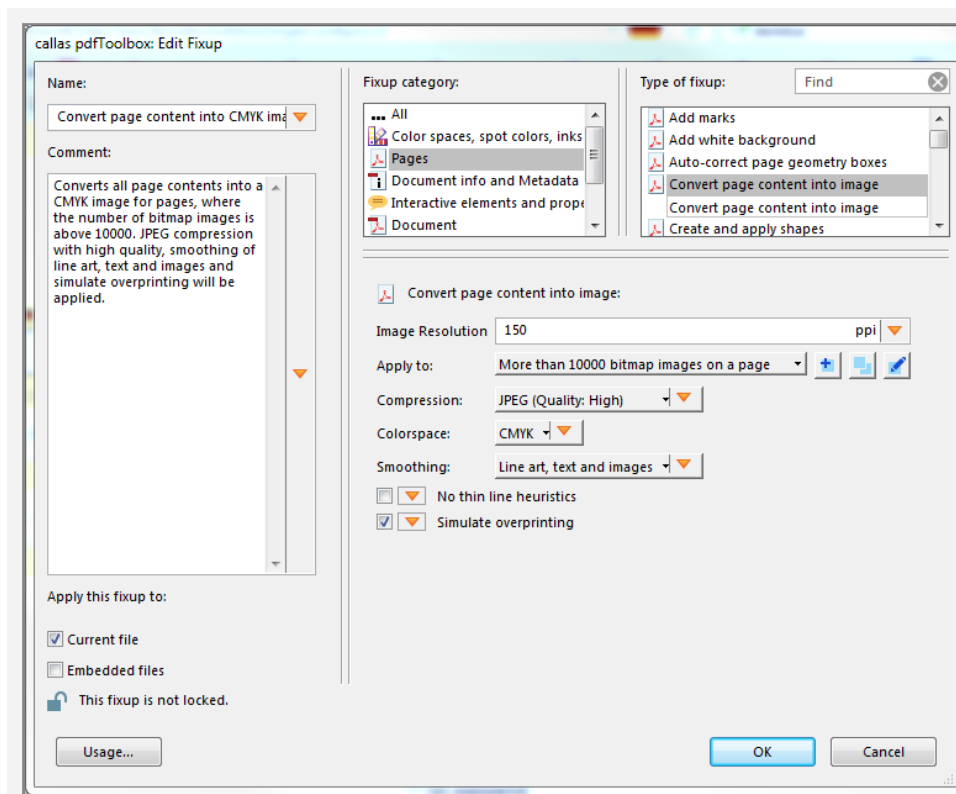
## Only fire, if more than ... objects on a page



For example, the predefined Check "More than 10000 bitmap images on a page" is configured in the way, that it will only fire, if the number of bitmap images is greater than 10000.

Other use case are possible of course: to check if there are less than a defined number of objects on the page (maybe to detect almost empty pages), ...

## Create a Fixup with a such a Check



Such Checks can easily be used in Fixups. For example to convert complex pages into images to prevent problems with an older RIP or printer.

# New and extended Fix-ups

# New and enhanced Fixups in pdfToolbox 9.1

## New Fixups

- Insert empty page  
For adding additional, empty pages at specific positions within the PDF (can be combined with a Check, e.g. for a sequential page number)
- Move objects  
To moves objects (defined by Check) with a defined offset vertically and/or horizontally

Several new or extended Fixups regarding "Processing Steps" e.g. for creating layers or to set special metadata for such layers:

- Put objects on Processing Steps layer
- Modify layer name for Processing Steps layer metadata
- Add Processing Steps layer metadata
- Extended with "Processing Steps" related options:
  - Configure OC-CD
  - Remove layer
  - Set layer default to
  - Set layer initial export state | print state | visibility state
  - Set layer name
  - Set layer state
  - Set layer intent
  - Set layer visibility dependent on a zoom level



## Extended Fixups

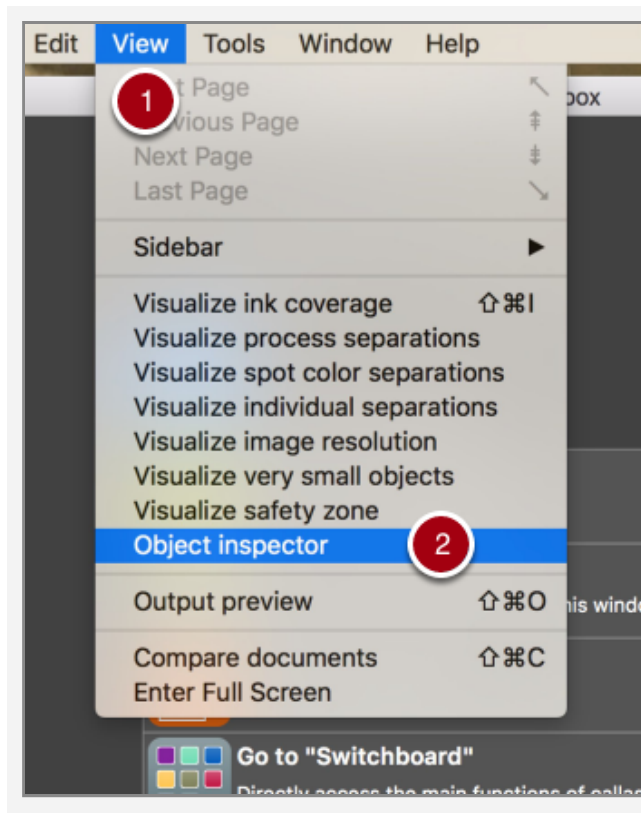
- Flatten transparency:  
Several compression methods added for images, which are created during transparency flattening
- Create and apply shapes:  
New options to reduce or enlarge non-rectangular shapes and to merge overlapping shapes
- Place content on page:  
Support for SVG as input format
- Flip pages:  
Extended with an optional "Apply to" to limit the correction to defined pages

# Wireframe and selective viewing

# Examining page content

In the visualizer section of pdfToolbox Desktop, the Object Inspector allows identifying page content, both viewing which objects form a PDF page, and what their attributes are. Two additional concepts: "Wireframe viewing" and "Object type filtering" have been implemented here.

## Showing the object inspector

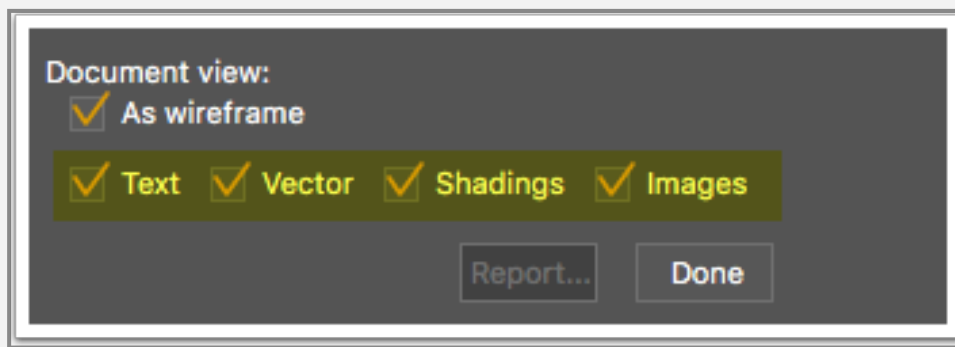


1. Click on the "View" menu
2. Select the "Object inspector" menu item

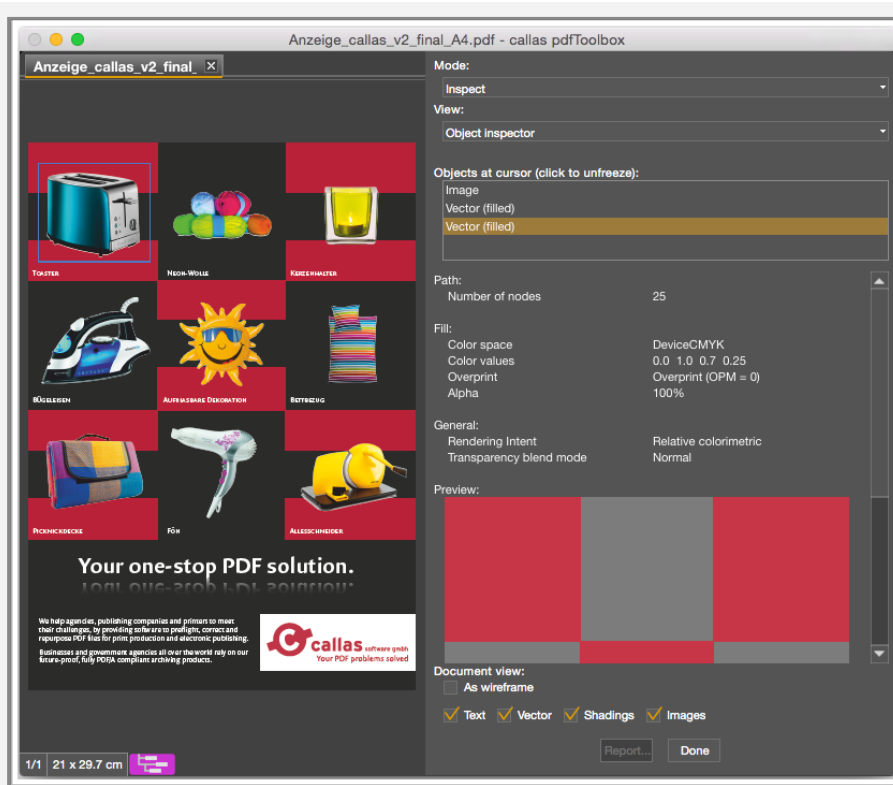
## Object type filtering

By default, the object inspector shows all PDF objects on the page. This can be changed by disabling or enabling the

checkboxes at the bottom of the Object inspector area. Deselecting "Text" for example, will cause all text objects on the page to be hidden. This allows examining whether objects are actually text for example, or allows viewing what is behind other objects (and normally hidden from view).



## Object properties



Every single object can be selected by clicking on it inside the view on the left hand side. A click will fix the selection on the right hand side. Another click will release it.

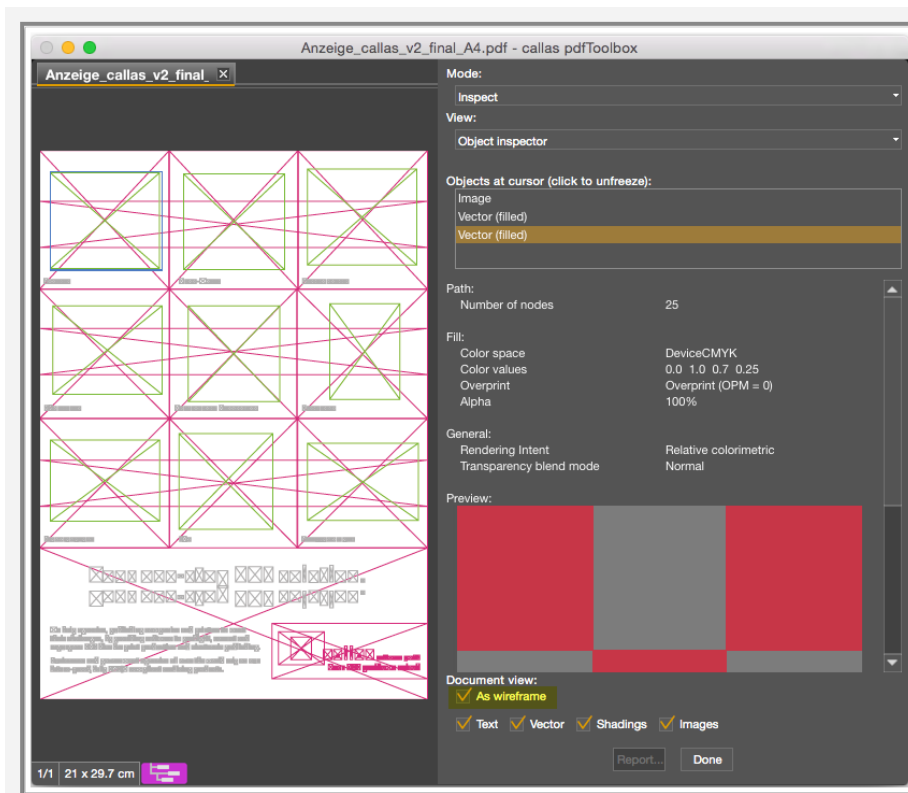
Selected objects are also visible in the preview on the right hand side.

Under "Objects at cursor", you will see a stack of all the objects in the same order as they appear in the PDF.

The topmost object is the topmost in the PDF. You can click at objects below the topmost object to see it's properties.

## Wireframe viewing

At the bottom of the Object inspector area, check the "As Wireframe" checkbox, to show the displayed PDF page as wireframe. In this mode, all objects are shown with different-ly color rectangle outlines. This allows seeing the structure of the page (which objects are on the page, how they are layered etc...).



# Advanced barcode and matrix code features

# Advanced 2D code use cases: Deutsche Post DP Matrix, Data Matrix Industry, rainbow colored QR Code (requires pdfToolbox 9.1)

pdfToolbox 9.1 comes with a number of extended capabilities that make it possible to create barcodes and matrix codes for all kinds of industries and use cases. This article illustrates the possible use of the barcode and matrix code creation in pdfToolbox in the form of several interesting examples.

Note: These examples make use of a couple of advanced features that are not feasible with the "Place barcode" fixup, but require use of the more advanced "Place dynamic content" fixup and custom written HTML and JavaScript.

## pdfToolbox Library with pre-configured example fixups

Please feel free to download and import the pdfToolbox Library provided below:



Barcode\_and\_matrix\_code\_examples.kfpl

After import you will find the examples below in the Fixups area of pdfToolbox.

## Example: Deutsche Post DP Matrix code examples

Deutsche Post DP Matrix 2D codes have to follow very strict specification. It is based upon DataMatrix codes, combined with a number of Deutsche Post specific rules.

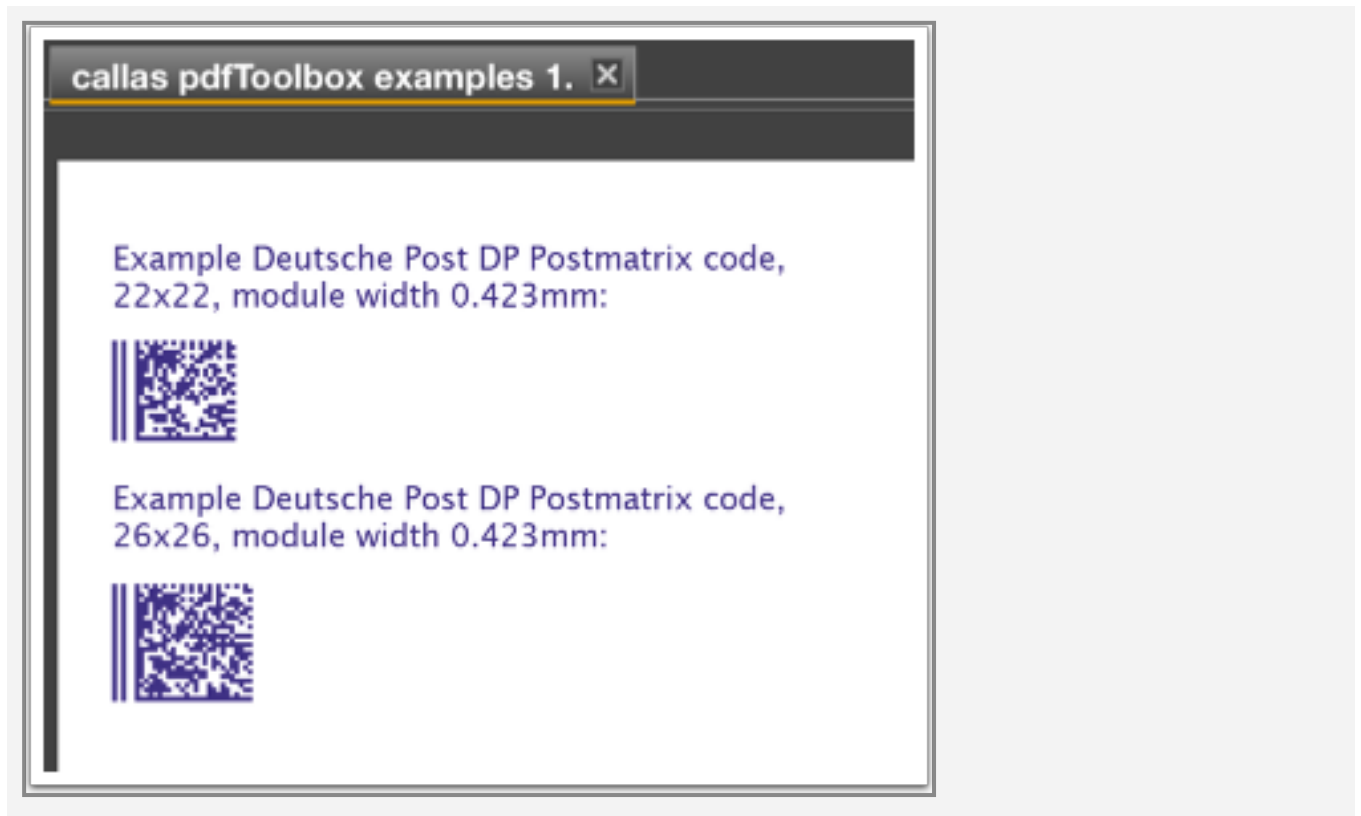
The example fixup provided creates two variants of such a Deutsche Post DP Matrix code in the upper left of every page of the currently open document.

At the core of custom HTML template is the following code:

```
<object class="barcode_object" type="application/barcode" >
  <param name="type" value="Data Matrix">
  <param name="data" value="***insert data!***">
  <param name="modulewidth" value="0.423mm">
  <param name="dm_format" value="PostMatrix">
    <!-- **empty string**, UCCEAN, Industry, _Macro05,
Reader, PostMatrix -->
    <param name="dm_size" value="22x22">
      <!-- **empty string**, 10x10, 12x12, 14x14, 16x16,
18x18,
20x20, 22x22, 24x24, 26x26, 32x32,
36x36, 40x40,
44x44, 48x48, 52x52, 64x64, 72x72,
80x80, 88x88,
96x96, 104x104, 120x120, 132x132,
144x144, 8x18,
8x32, 12x26, 12x36, 16x36,
16x48
-->
    <param name="dm_enforcebinaryencoding" value="false">
      <!-- **false**, true-->
    <param name="dm_rectangular" value="false">
      <!-- **false**, true-->
</object>
```

Applying this code with proper CSS styling will put a Deutsche Post DP Matrix code in the upper left corner on the pages of the currently open PDF :





## Example: DataMatrix Industry 2D code 16x48

DataMatrix codes come in various flavors specific to the sector where they are used. For certain industry uses a sub-type "Industry" exists, that uses a rectangular form not a square form of DataMatrix codes

The example fixup provided creates one variants of 16x48 cells, and places it in the lower left of every page of the currently open document.

At the core of custom HTML template is the following code:

```
<object class="barcode_object" type="application/barcode" >
  <param name="type" value="Data Matrix">
  <param name="data" value="Actual data"> <!-- <== Actual data must go here
-->

  <param name="modulewidth" value="0.25577mm">
  <param name="dm_format" value="Industry">
  <param name="dm_rectangular" value="true">
  <param name="dm_size" value="16x48">

<!--
```

```
<param name="dm_size" value="16x48">  
supported values:  
**empty string**, 10x10, 12x12, 14x14, 16x16, 18x18, 20x20, 22x22, 24x24,  
26x26, 32x32, 36x36, 40x40, 44x44, 48x48, 52x52, 64x64, 72x72, 80x80,  
88x88,  
96x96, 104x104, 120x120, 132x132, 144x144, 8x18, 8x32, 12x26, 12x36,  
16x36, 16x48  
-->  
</object>
```

Applying this code with proper CSS styling will put a DataMatrix Industry 2D code 16x48 in the lower left corner on the pages of the currently open PDF:



## Example: Rainbow colored QR Code with your name

QR codes have many uses. This example illustrates a concept that won't be acceptable when it comes to maximizing readability of codes in an industrial environment, but can still put to good uses in some creative scenarios. Be prepared though to accept that such codes will not conform to any of the ap-

plicable ISO standards. Still they can be scanned surprisingly well with your average smartphone barcode app or any up-to-date 2D code reader.

Note: The technique shown here can also be applied to any other type of barcode and matrix code.

The example fixup provided creates a rainbow color QR code with the name being entered upon executing the fixup.

At the core of custom HTML template is the following code:

```
<object id="barcode_object" type="application/barcode"
      style = "color: #eee; color: -cchip-cmyk(0,0,0,0.1);
      background-color: pink;
      background: linear-gradient(135deg, firebrick, red, orange,
orange, green, blue, indigo, violet); "
>
  <param name="type" value="QR-Code">
  <param name="modulewidth" value="1mm">
  <param name="data" id="id_barcodevalue" value="fill in actual value">
  <param name="swap_foreground_background" value="true">
  <param name="quietzoneleft" value="1">
  <param name="quietzoneright" value="1">
  <param name="quietzonetop" value="1">
  <param name="quietzonebottom" value="1">
  <param name="quietzoneunit" value="X">
</object>
```



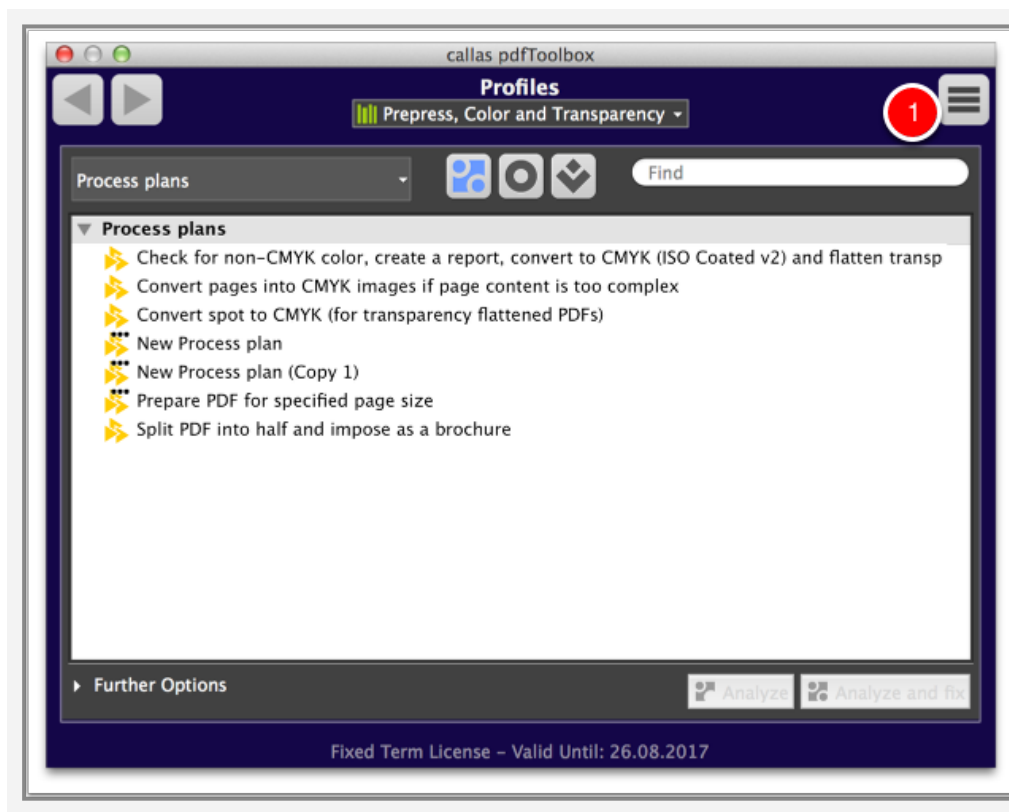
# Debugging of Profiles and Process plans (9.1)

# How to create a detailed log when executing Process Plans (or Profiles, Checks or Fixups)

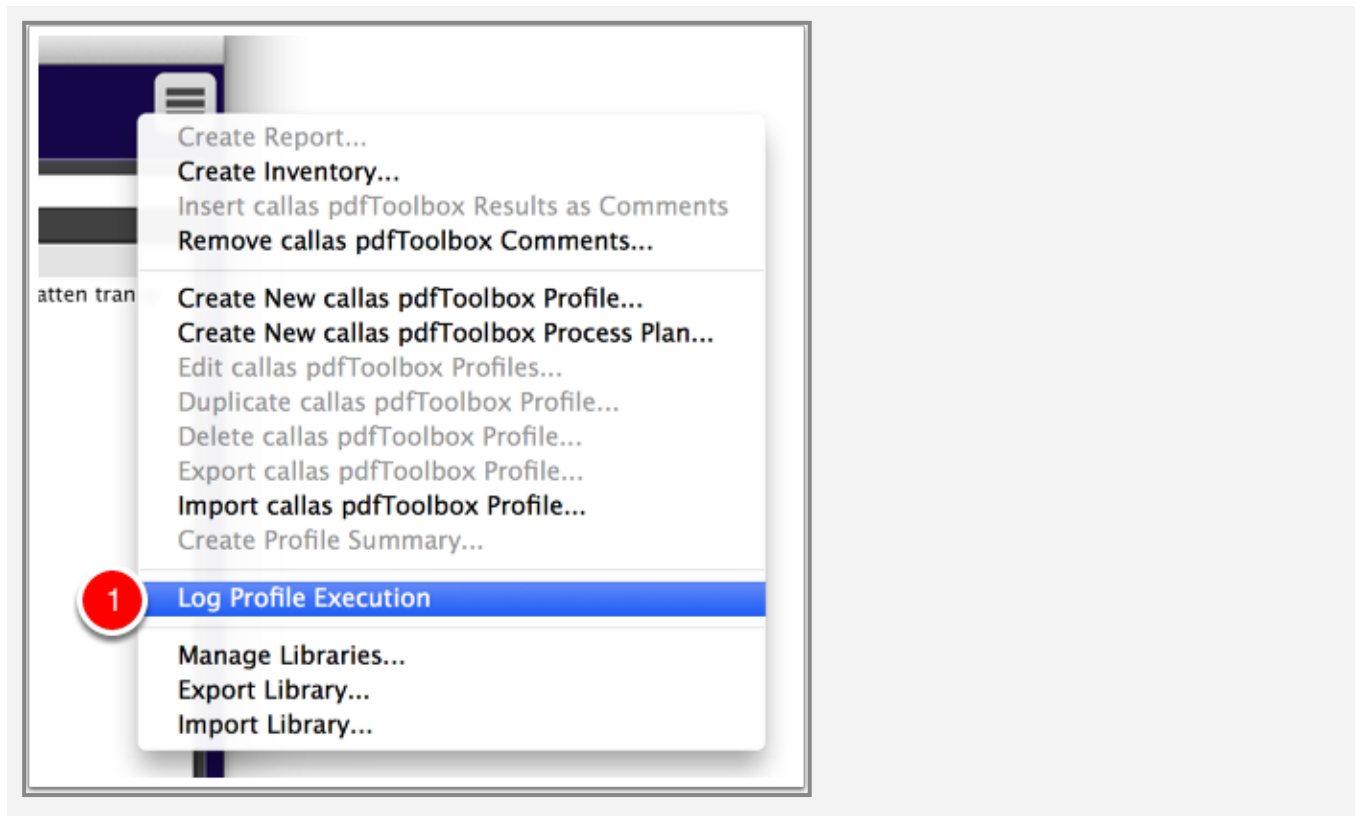
Especially when creating a Process Plan, it can become necessary to inspect intermediate results (such as PDF files in their state in the middle of a Process Plan's processing steps and preferably some details about each step in the form of a log file or similar). This can help to understand and to optimize the configuration of the steps in a Process Plan and their inner workings as a whole.

Note: This logging feature is not only available for Process Plans, but also for Profiles, Checks, and Fixups.

## Activate logging



To activate logging, just click within one of the three sections of the Profile window on the flyout menu button in the upper right corner (1).



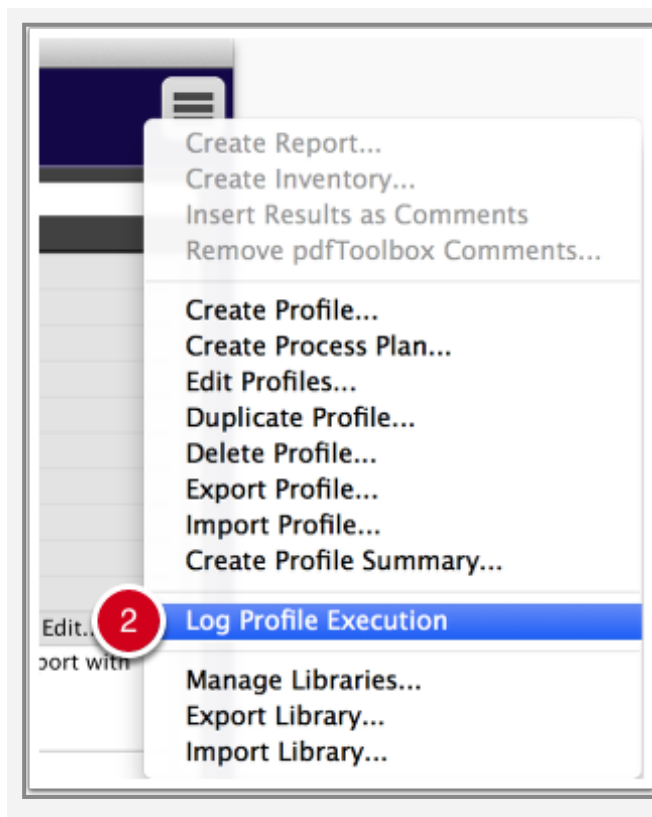
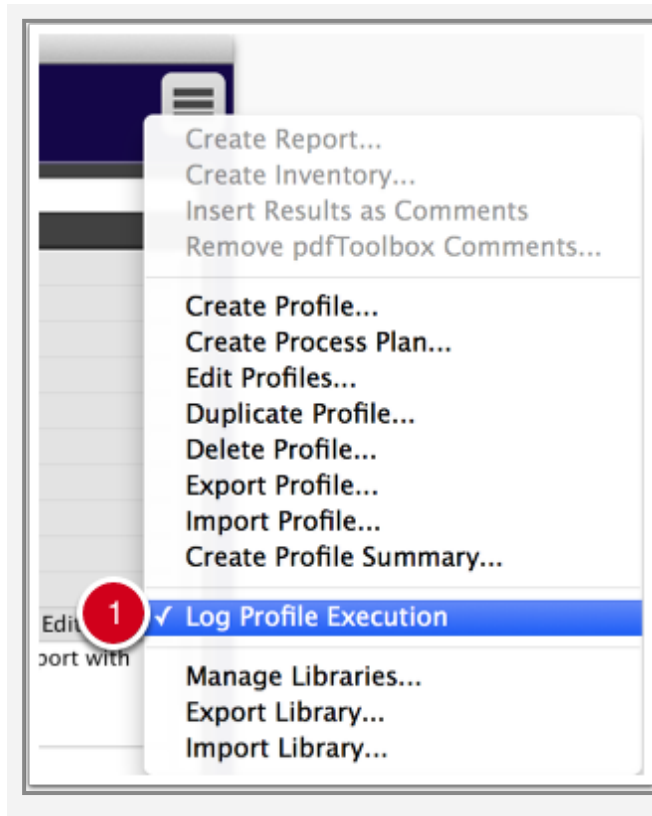
Select the entry "Log Profile Execution" (1).

This option activates the logging of the execution of any Process Plans, Profiles, Checks and Fixups.

A check mark ("✓") in front of the menu item indicates that logging is active.

## Deactivate logging

In order to deactivate logging, simply execute the menu item again (1). The check mark ("✓") will then disappear from the menu item (2).





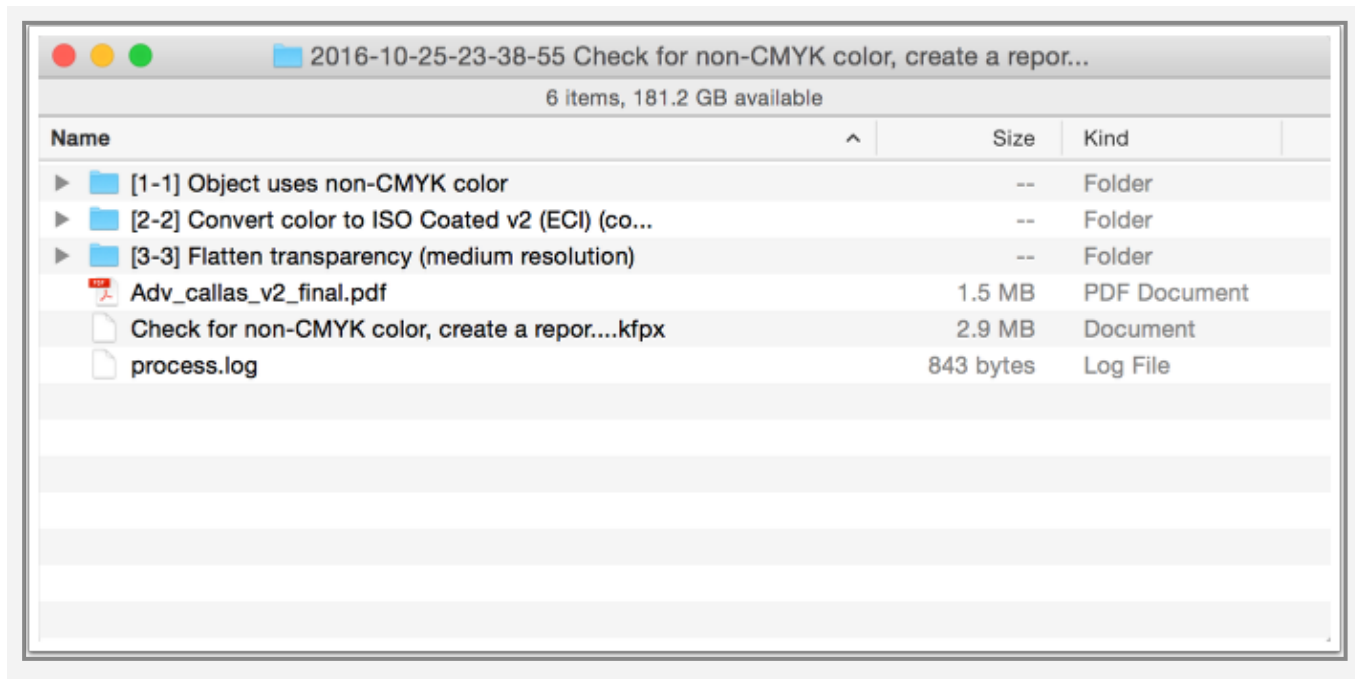
## Execute a Process Plan (or Profile, Check or Fixup)

Execute a Process Plan (or a Profile, Check, or Fixup).



## Explore folder with logging information

After processing the PDF, a window will open in Finder (on Mac OS X) or in the Explorer (on Windows), revealing a folder (having a time stamp at the start of its name) with all the logging data and associated files inside it.



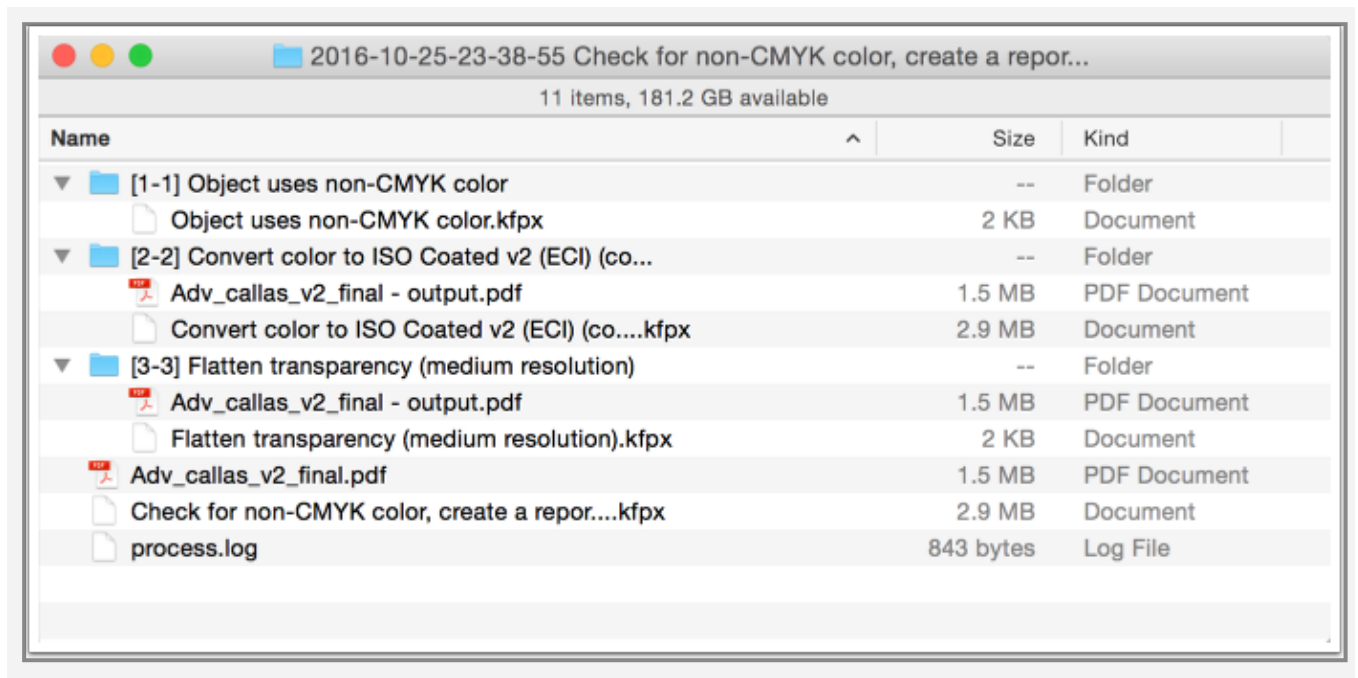
## Structure of the files subfolders in the logging folder

This folder will contain files and subfolders:

- folders with intermediate results (if applicable) for each step in a Process Plan (Profiles, Checks and Fixups only will have one such step) in a folder;
  - the subfolder's name will consist of sequence number and step number in square brackets, followed by the name of the step (or the name of the Profile, Check, or Fixup);
  - the subfolder will also contain the pro-

file, check or  
fixup associ-  
ated with the  
respective  
Process Plan  
step as a "kf-  
px" file

- the original file (in this example "Adv\_callas\_v2\_final.pdf")
- logging information in a file named "process.log"
- a "kfx" file with the whole Process Plan (or Profile, Check or Fixup) that was just executed



The contents of the "process.log" would contain information like shown in the example below:

```
2016-10-25 23:38:55      Check for non-CMYK color, create a report, convert to
CMYK (ISO Coated v2) and flatten transparency
2016-10-25 23:38:55      Input      /Users/olaf/TEMP-DELETE/
Adv_callas_v2_final.pdf
2016-10-25 23:38:55      Starting with step      1
2016-10-25 23:38:55      [1-1]      Object uses non-CMYK color
2016-10-25 23:38:55      Result      No hits
2016-10-25 23:38:55      Continuing with step      2
2016-10-25 23:38:55      [2-2]      Convert color to ISO Coated v2 (ECI)
(convert spot colors to CMYK)
2016-10-25 23:39:04      Result      Success
```

```

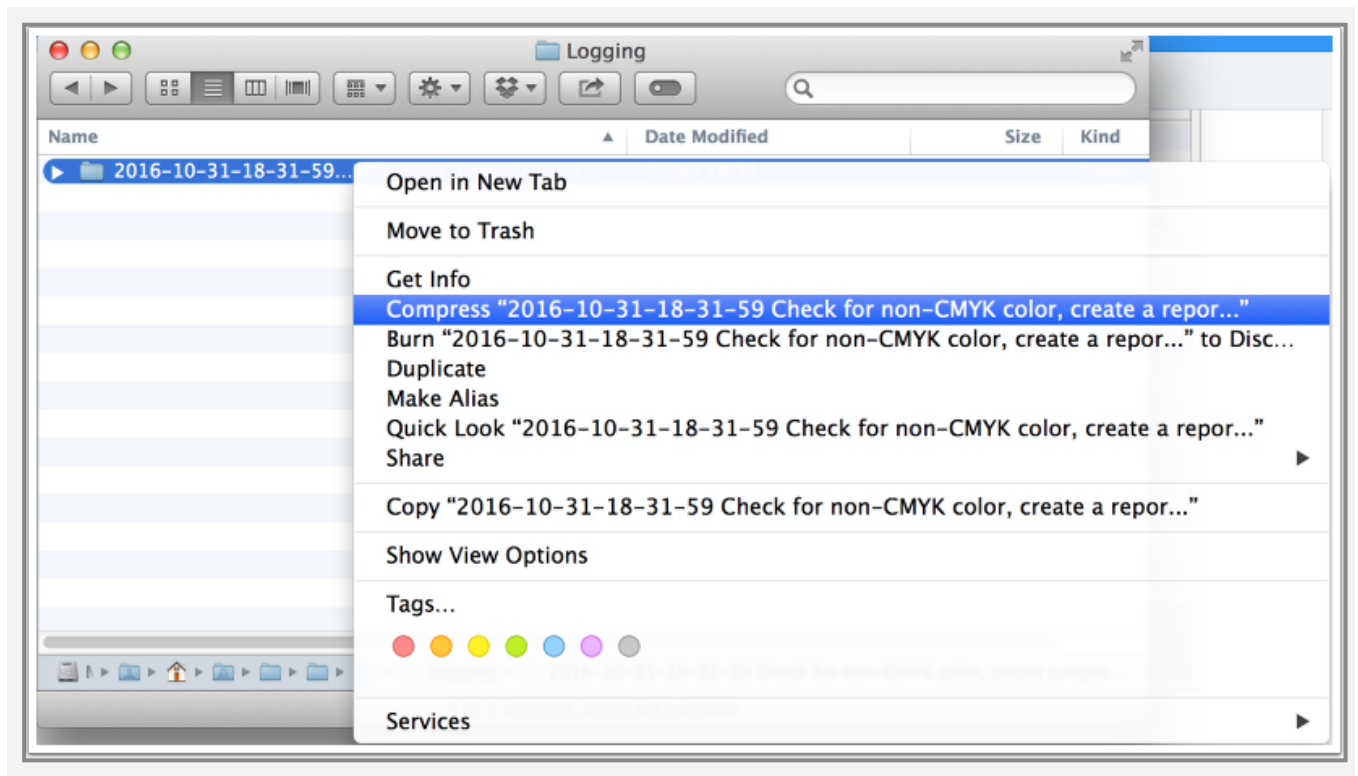
2016-10-25 23:39:04      Modified      /Users/olaf/TEMP-DELETE/
Adv_callas_v2_final - output.pdf
2016-10-25 23:39:04      Continuing with step      3
2016-10-25 23:39:04      [3-3]      Flatten transparency (medium resolution)
2016-10-25 23:39:05      Result      Success
2016-10-25 23:39:05      Modified      /Users/olaf/TEMP-DELETE/
Adv_callas_v2_final - output.pdf
2016-10-25 23:39:05      Terminating
    
```

## Sending the logging information for support cases

When requesting support from the callas support team, you might get asked to send the complete logging package.

This will help us to determine how processing was executed on your computer, and why something may not be working as expected. For easy and safe transfer please compress the entire folder into a ZIP archive by clicking the right mouse button after selecting the respective folder:

- Mac OS X: "Compress..."
- Windows: "Send to..." - "ZIP compressed folder"



# Processing Steps: Overview (9.1)

# Design and more

PDF documents typically are WYSIWYG: what you see is what you get. The PDF document contains those elements that need to be printed and it contains thus "design" elements in a complete and accurate way. However, in many cases, additional information needs to be transmitted from the designer to the printer as well, and that additional information often is added to the PDF document.

## Typical non-design content

It is a bit dangerous to call this additional content "non-design". It is strictly true, because such content will not be reproduced faithfully as is the rest of the design, but it can still influence the final appearance of the PDF document after print. Some examples:

- When designing non-rectangular jobs (such as labels and packaging), a cut line needs to be defined. This cut line indicates how the design is going to be cut to get the final printed piece.
- When designing something that that will be printed on a transparent material, it is often necessary to add an additional white ink layer underneath the rest of the design. This additional white layer is usually included in the PDF document, but it's usually included with a 'fake' (non-white) color appearance so it's visible in the PDF document.
- When creating complex jobs, it is often necessary to add all kinds of additional information to the job, such as job identity, printing and cutting marks, color patches, dimensions and so on. While this information is often critical to the job, it is of course not to be printed.
- Some jobs require special coatings or finishing processes; parts of a job might need to be varnished, may have silver or gold foils applied to them, or require embossing. These special processes are included in the PDF document, again with 'fake' colors to show where they will affect the design.

## Current practices

In workflows where such information is required, these special elements are typically indicated by using spot colors. Elements using a spot color with the name "White" refer to the additional white ink layer. Elements using the name "Varnish" indicate areas of the file to be varnished. Spot colors such as "Legend" or "Registration" or "Marks" may be used for elements that are not print content, but job identification or assistive printing or cutting marks.

## Problems with this approach

This approach with using spot colors raises a number of problems:

- The kind of jobs we are talking about here also typically use a list of spot colors for design elements (elements that do need to be printed in specific brand colors for example). Using spot colors for both design and non-design elements can lead to confusion.
- There is no standardisation on the spot color names used. The spot color used to indicate a die-cut line, may be called "Cut" or "Cutter" or "Die" or "Die-cut" or a variety of other names. On top of that, the design community today is global; a French designer will be likely to use a French name while a Finnish designer will use his own language. This makes it very hard to build any kind of automation for these files as key information can be encoded in a variety of different ways.

# Using metadata for standardisation

Because of the challenges described in the previous article, work of the [Ghent Workgroup](#) lead to the creation of a new ISO standard (ISO 19593) called Processing Steps. In full the standard is called: "Use of PDF to associate processing steps and content data". Content data here obviously refers to the design elements itself, what will be printed. Processing steps refers to this additional "non-design" information stored in the PDF document.

So how does this standard work?

## Use of layers

The PDF standard has a built-in feature called "Optional Content Groups" (OCDs). This is commonly referred to as "layers" though it's important to realise that there are important differences between layers such as you might know them from Adobe Photoshop or Adobe Illustrator and optional content groups. Layers in design application typically reflect stacking order: the objects in the front layer are "on top" of objects in all other layers. This is not the case for optional content groups; PDF documents can contain an optional content group that contains all images in the document, regardless of their stacking order. And moving an element from one optional content group to another, doesn't change it's stacking nor the visual appearance of the document.

These optional content groups are used to gather all elements belonging to a processing step. All vector elements that form the die cut line for example, are placed in an optional content group. Such optional content groups have a name that can be used to easily identify them.

## Attaching metadata to layers

Of course using optional content group names to identify them, would bring us right back to the problem of standardisation; everyone would use their own version of a name... To solve this, the processing steps standard uses metadata attached to the layer for the actual identification. Each layer has two pieces of identifying metadata attached to it:



- **Group**  
Identifies what kind of processing step this is. Possible groups are "Structural", "Dimensions", "Braille", "Legend", "White", "Varnish" and "Positions".
- **Type**  
Identifies the type of processing step in that particular group. In the group "Structural", possible types include "Cutting", "Creasing", "Gluing" and so on.

## Using spot colors in layers

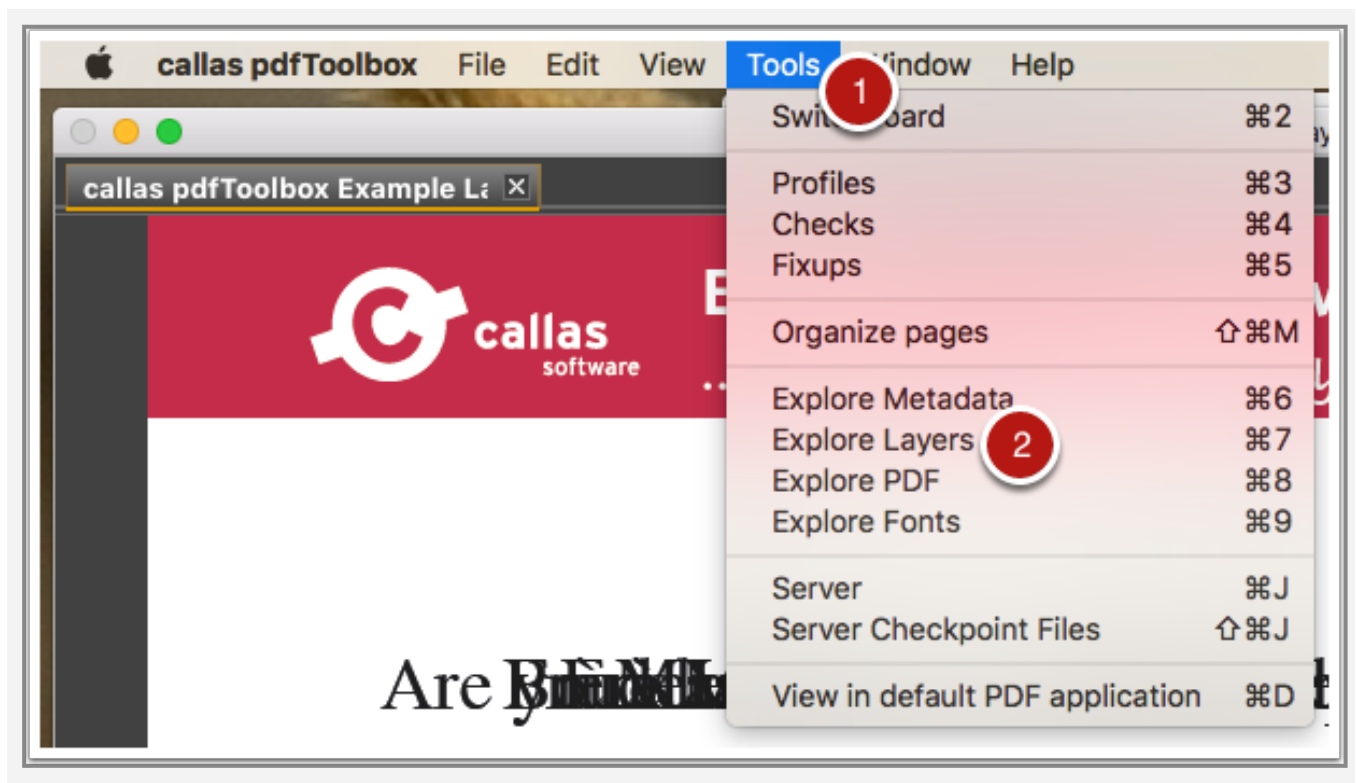
Using layers and metadata associated to layers, solves the standardisation problem for processing steps information. However, the elements that are in such a layer still need to have a color, and it makes the most sense to continue to use spot colors for this.

Because of the layers though, these spot colors can be named whatever the designer wants them to be named. As long as the proper processing steps metadata is used, they can be identified regardless.

# Viewing the layers in a document

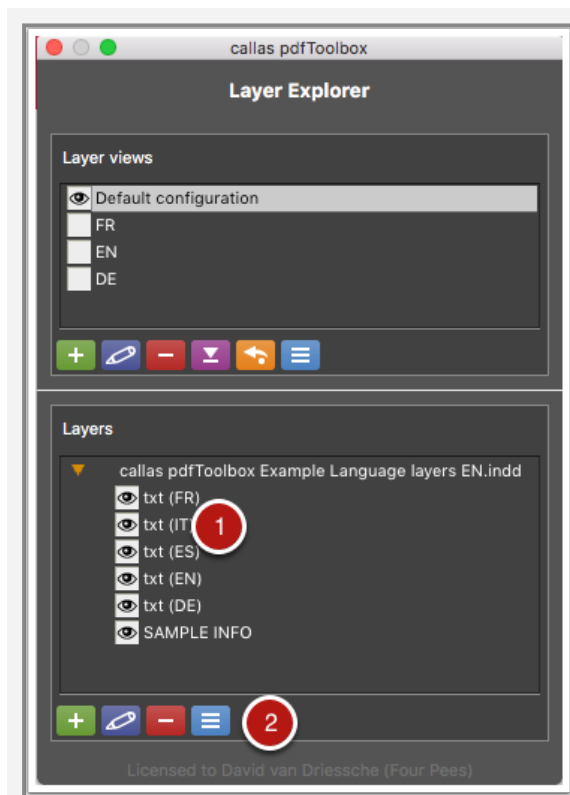
As Adobe Acrobat, pdfToolbox uses the term "Layers" to refer to what is technically called "Optional Content Groups" in the PDF specification. The rest of this article will use the term layers.

## Open the Layer Explorer



1. Use the "Tools" menu.
2. Click on "Explore Layers".

## Work with the layers in the Layer Explorer

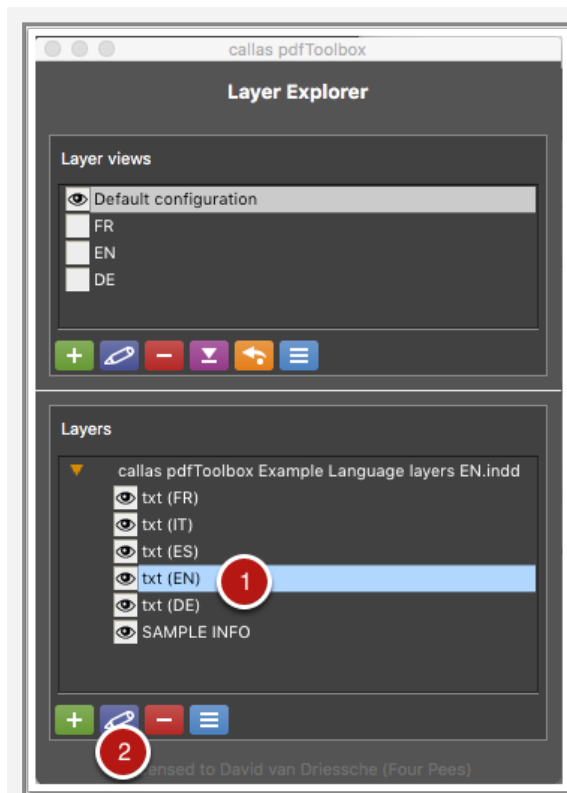


1. The Layer Explorer lists all layers present in the current document. you can switch them on or off (make them visible or invisible) by clicking the little eye icon in front of their name. If processing steps information is available for a layer, it will be listed after the name of the layer.
2. The buttons under the list of layers allow adding, editing or removing a layer.

# Working with processing steps metadata for a layer

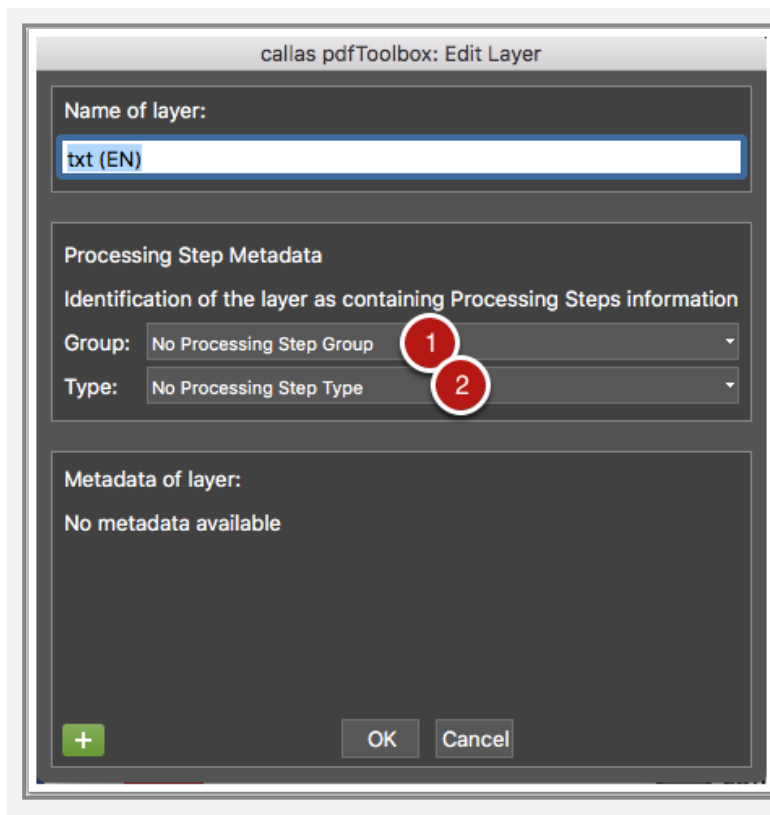
Layers can have regular metadata and processing steps metadata attached to them.

## Accessing layer metadata



1. Click the layer you want to see the metadata of.
2. Click the "edit" button to "Edit layer" dialog window.

## Viewing processing steps metadata



The "Edit layer" dialog window contains three sections with information about the layer:

1. The name of the layer
2. The processing steps information for the layer
3. Additional metadata associated with the layer

Look at the middle section to work with the processing steps information. This section lists:

1. The processing steps group associated with this layer, or "No Processing Steps Group" if no information is available for this layer.
2. The processing steps type associated with this layer, or "No Processing Steps Type" if no information is available for this layer.

## Changing processing steps information

You can change the processing steps group or type by using the pull down menus in the middle section of the "Edit layer" dialog window.

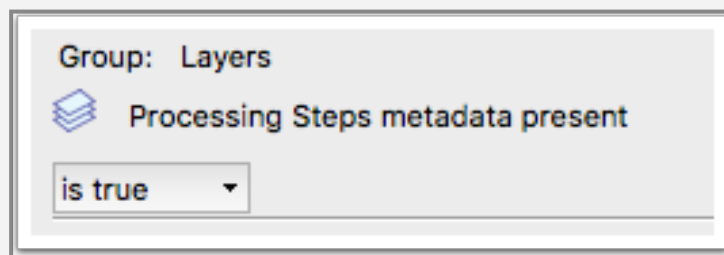
## Deleting processing steps information

Processing steps information can be removed by using the pull down menus in the middle section of the "Edit layer" dialog window. Simply select the top value in both menus ("No Processing Steps Group / Type").

# Checking processing steps information

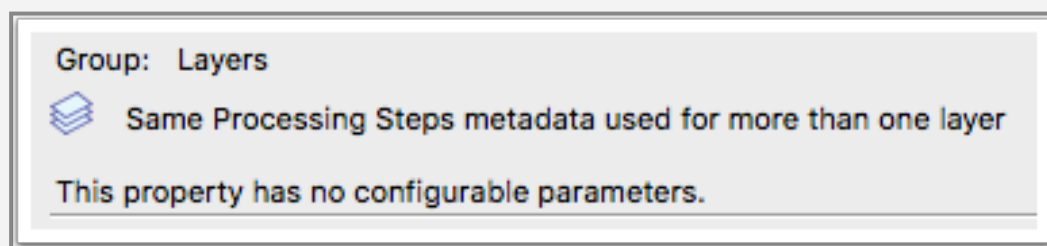
Having correct processing steps information can be important for the functioning of automatic workflows. As such, pdfToolbox implements a number of specific processing steps checks.

## Checking for presence



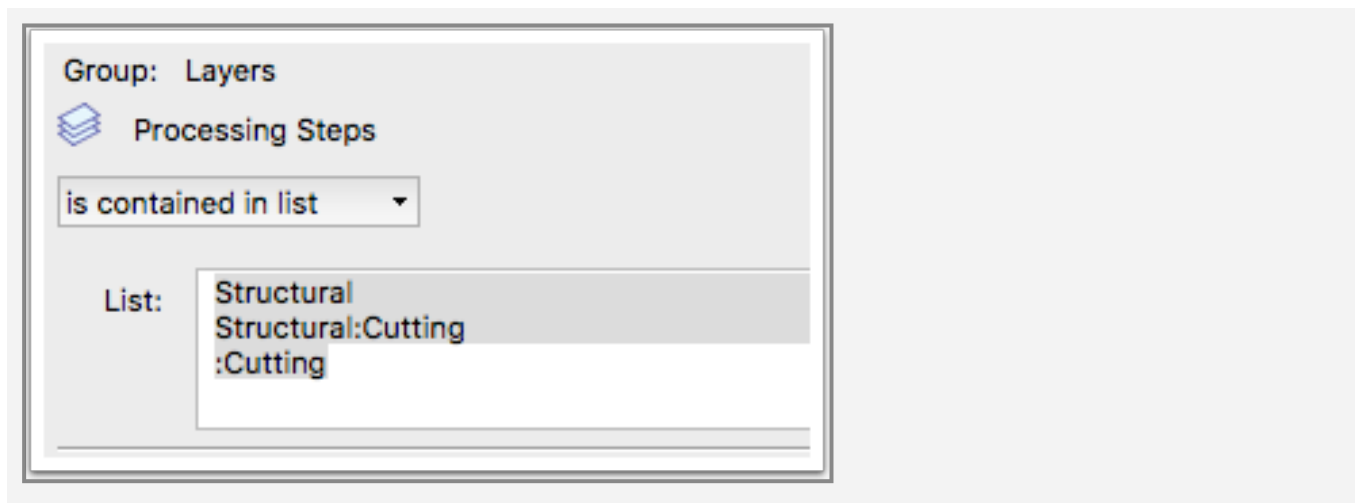
This condition returns true if processing information is present in the document, false if it is not.

## Checking for conflicts



This condition can be used to find out whether the same processing steps information is used for more than one layer. If two layers are marked "Structural" > "Cutting" for example, it makes it harder to figure out which of those two is the actual die-line, and it might indicate other problems with the file or the workflow.

## Identifying layers with specific processing types



This condition is useful to identify specific processing steps layers in a document. Multiple items can be searched for by listing each item on a new line (as in the example above). Each line must have one of three possible formats:

- **<group name>**  
The line contains just the name of a processing steps group, no type is mentioned. This will create a hit for any processing steps layer that has this specific group (regardless of type).
- **<group name>:<type name>**  
The line contains the name of a processing steps group, followed by a colon (':'), followed by the name of a processing steps type. This creates a hit for any layer that has the specified group *and* type.
- **:<type name>**  
The line contains a colon (':'), followed by the name of a processing steps type. This creates a hit for any layer that has the specified type (regardless of group).



## Identifying custom processing steps information

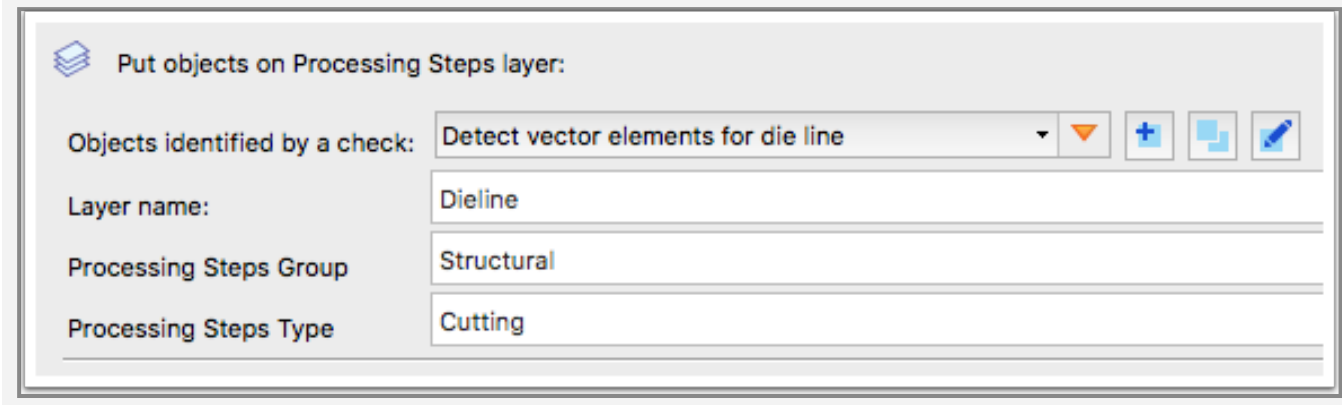


The processing steps standard defines a list of predefined groups and types, but it also allows custom values to be used when none of the predefined values can be used. This condition finds layers where such custom values are used.

# Fixing processing steps data

pdfToolbox can fix a number of common problems with processing steps information and can be used to convert legacy files using spot color identification to processing steps.

## Putting objects on a specific layer

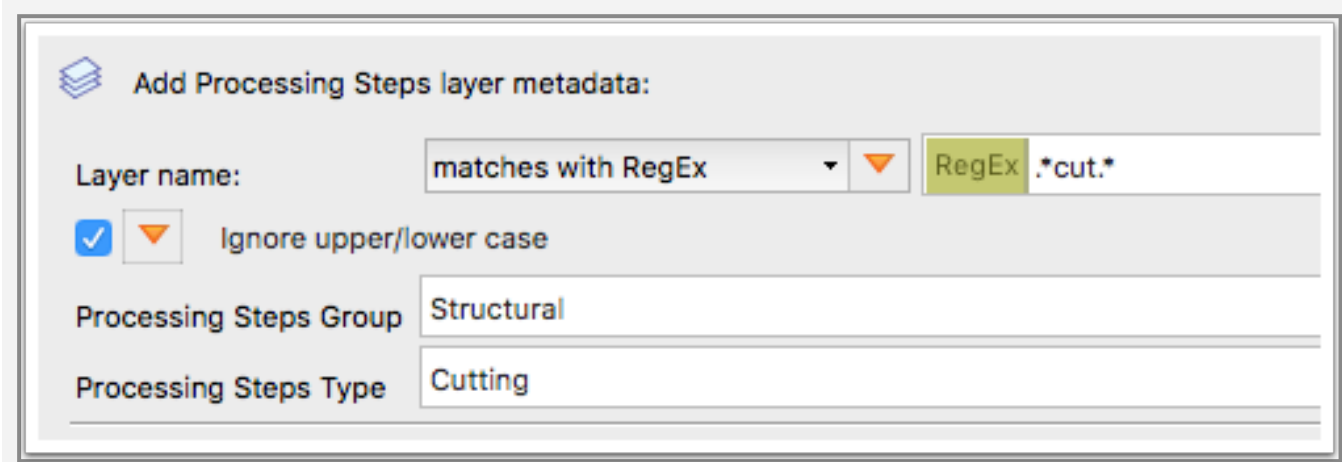


The dialog box is titled "Put objects on Processing Steps layer:". It contains the following fields and controls:

- Objects identified by a check:** A dropdown menu showing "Detect vector elements for die line" with a downward arrow, a plus icon, a square icon, and a pencil icon.
- Layer name:** A text field containing "Dieline".
- Processing Steps Group:** A text field containing "Structural".
- Processing Steps Type:** A text field containing "Cutting".

This fixup identifies objects with a preflight check; those objects are then put on layer identified by its processing steps group and type.

## Adding processing steps information to a layer

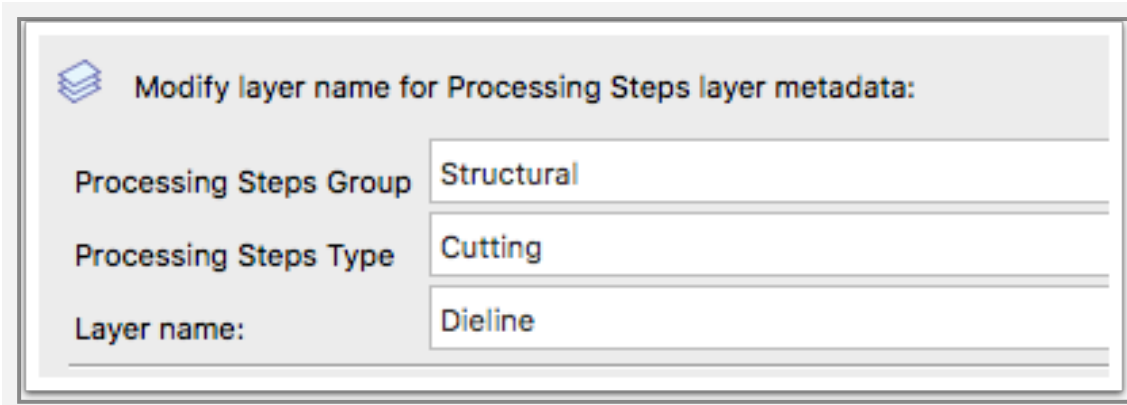


The dialog box is titled "Add Processing Steps layer metadata:". It contains the following fields and controls:

- Layer name:** A dropdown menu showing "matches with RegEx" with a downward arrow, a plus icon, and a square icon. To the right is a text field containing "RegEx .\*cut.\*".
- Ignore upper/lower case:** A checked checkbox with a plus icon and a square icon.
- Processing Steps Group:** A text field containing "Structural".
- Processing Steps Type:** A text field containing "Cutting".

This fixup identifies a layer by name, and then adds specific processing steps information to it.

## Rename layer identified by processing steps information



The screenshot shows a dialog box titled "Modify layer name for Processing Steps layer metadata:". It contains three input fields with the following values:

Field	Value
Processing Steps Group	Structural
Processing Steps Type	Cutting
Layer name:	Dieline

This fixup identifies a layer using the specified processing steps information and changes its name.