Inkscape - Part 80

Last time, I showed you how you can use a fragment identifier in the URL that points to your SVG file in order to make a browser show only a small part of the whole image. With a suitably crafted string, you can precisely identify a rectangular section of the image to display by specifying the x and y coordinates of the top left of the rectangle, together with its width and height. I also demonstrated how you could create named views so that the coordinates are entirely contained within the SVG file and the enclosing web page doesn't need to know the gory details; all it needs to know is the ID of the view to display.

Through this technique, you can effectively turn a single image into a number of named tiles, spread out along the x and y plane. This month, you'll discover that it's possible to perform a similar trick in the z-axis. In other words, you can stack several sub-images on top of each other, and then choose to show only one of them by putting its ID into the fragment identifier.

The starting point for this technique isn't actually SVG, but HTML. The way this trick works relies heavily on inheriting certain behaviours from the HTML and CSS world that can be manipulated into doing something useful in SVG. Let's start with a simple HTML file (shown top right).

Allowing for the requirement to keep magazine listings short, try to imagine that this page has a lot of text in each paragraph, a lot of paragraphs in each section, and more sections than I've included in the sample. If you're trying out the code yourself, make sure to include enough content for you to have to scroll quite a lot to get to the bottom section.

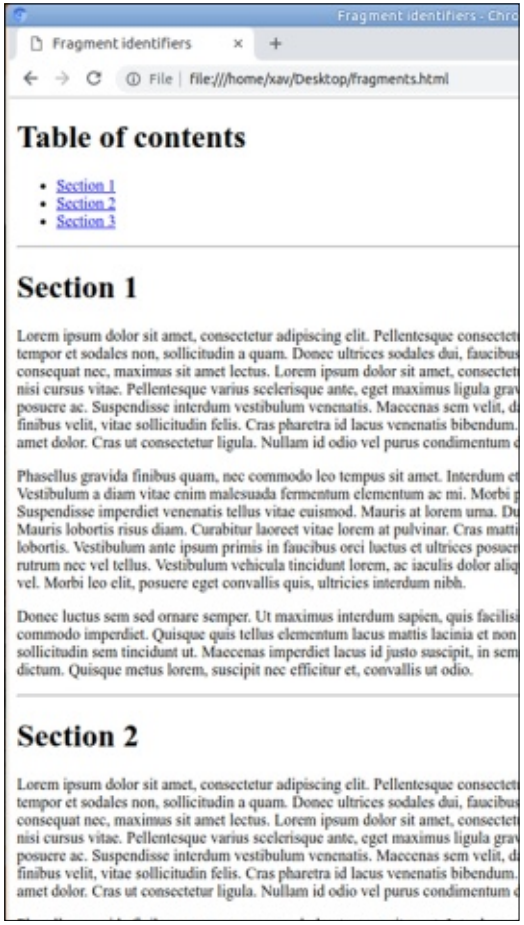
With a lengthy page of text, it would be nice to be able to jump straight to a section via a list of links at the top of the page – a table of contents. To do that, we use the <a> tag in two different forms: around each section to "name" the link target, and in the

```
<!DOCTYPE html>
<head>
      <title>Fragment identifiers</title>
</head>
<body>
      <h1>Section 1</h1>
      <p>Blah, blah, blah…</p>
      <p>More blah, blah, blah…</p>
      <p>Even more blah, blah, blah…</p>
      <hr />

      <h1>Section 2</h1>
      <p>Blah, blah, blah…</p>
      <hr />

      <h1>Section 3</h1>
      <p>Blah, blah, blah…</p>
</body>
```

```
<h1>Table of contents</h1>
<ul>
      <li><a href="#sec1">Section 1</a></li>
      <li><a href="#sec2">Section 2</a></li>
      <li><a href="#sec3">Section 3</a></li>
</ul>
<hr />

<a id="sec1">
      <h1>Section 1</h1>
      <p>…</p>
      <hr />
</a>

<a id="sec2">
      <h1>Section 2</h1>
      <p>…</p>
      <hr />
</a>

<a id="sec3">
      <h1>Section 3</h1>
      <p>…</p>
</a>
```

table of contents to create a link that will jump straight to the named target. Skipping the boilerplate code and the content of the <p> tags we get the code shown on the previous page (bottom right).

A quick trip to a "lorem ipsum" generator to fill out the content, and the web page looks something like this when loaded into a browser:



Clicking the links in the Table of Contents will scroll the browser so that the target of the link is visible on screen. The "href" attribute in those links is essentially just the fragment identifier part of the full URL. Indeed, if we modify the full URL in the browser to append "#sec3" to the end of the URL, the browser will also jump to the third section.

Okay, so appending the ID of an <a> element to the URL will make the page scroll to that target. What's that got to do with SVG? To answer that we need to introduce another part of this puzzle: the CSS ":target" selector. Here's a section of CSS that should be added to our test page:

```
<style>
      :target {
            color: red;
      }
```

This tells the browser to run this little algorithm:

- Does the URL have a fragment identifier?
- If so, does it match a target ID in the page?
- If it does, apply the "color: red" property to the matching element (in addition to the default behaviour of scrolling the element into view).

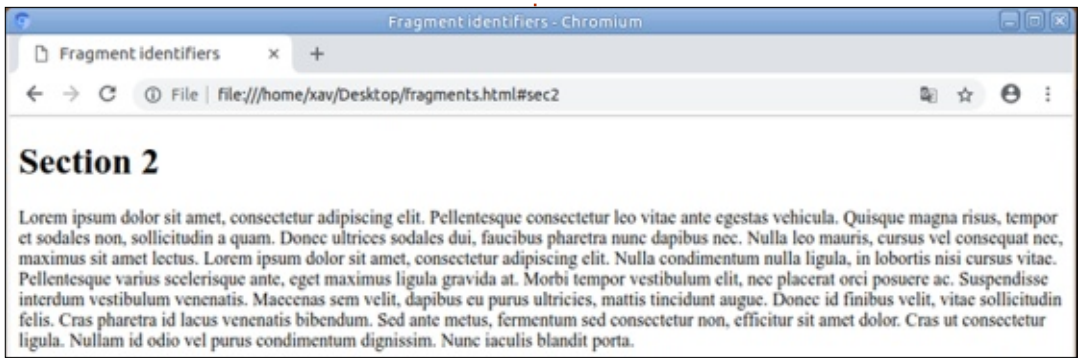
Now, as you click the entries in the table of contents or manipulate the URL by hand, the target section is rendered with red text, rather than black. We've got a way to apply a style to only the element specified in the fragment identifier. Which means we can instead do something like this:

```
<style>
      a { display: none; }

      :target {
            display: initial;
      }
</style>
```

This will hide all the <a> elements, and their descendants, then only show the one that is being targeted by the fragment identifier. As a side-effect, it also hides the links in the Table of Contents. A more fine-grained rule could deal with that, but I just deleted the whole table from my document as it's not needed any more. Now, when manually adding, for example, "#sec2" to the end of the URL we will see only the specified section on screen.

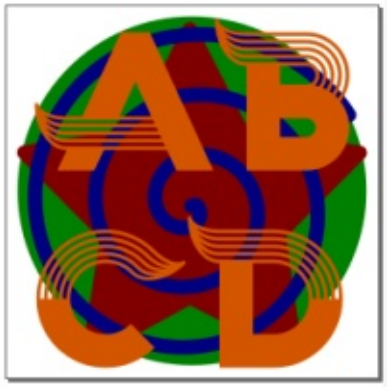
So we've now got a way to show only a single element (and its descendants), based on a label in a fragment identifier. All we need to do is transplant that same logic to the world of SVG, and the browser, using the same CSS rules as it does for an HTML page, will show only the element we've chosen.

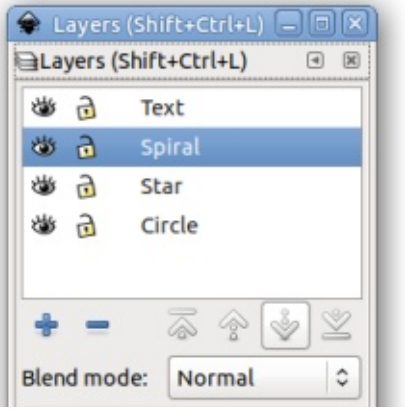
The first thing we'll need, therefore, is a type of element that can act as a container for the content we want to show and hide. In SVG terms, that would be a <g>, which in Inkscape is exposed as a group, but is also used for layers (as noted in previous instalments, layers are just groups with extra metadata). There are minor pros



and cons to each: considering that we need each <g> to have an ID, groups have a small advantage in that the Object Properties dialog (from the context menu) can be used to set the value. Although you can rename layers, doing so doesn't affect their ID, requiring a trip to the XML editor instead.

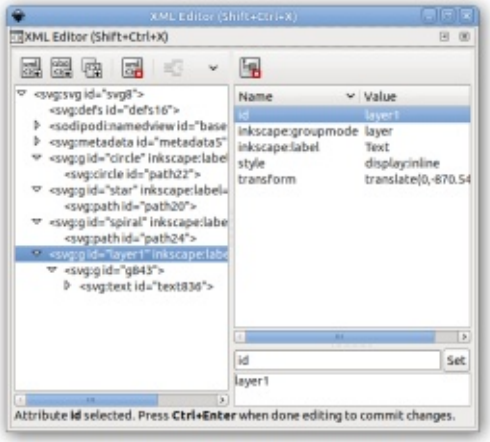
But layers have one big advantage: they're much easier to show and hide when working in Inkscape. Given that this workflow requires each image to be stacked upon the others, this advantage becomes invaluable for anything more than a couple of groups. For the rest of this tutorial, therefore, I'll be using layers. Continuing from last month's instalment, I'm going to stack the same four objects I used previously – a circle, a star, a spiral and some text – putting them into individual layers.





It's worth reiterating at this point that each layer could contain more than a single element – I just didn't want to clutter up the screenshots too much by introducing more content. In practice, each layer would typically be an SVG image in its own right, potentially containing multiple elements, groups, clones and filters. In a more complex situation, you can clearly see how the ability to hide and lock individual layers makes this method generally superior to just using Inkscape groups.

With my layers created, the next step is to open the XML Editor (Edit > XML Editor) and set the IDs for the layers. In the screenshot, you can see that I've already set the IDs for "circle", "star" and "spiral", and I'm about to change "layer1" to "text".



While you've got the XML editor open, it's worth noting what happens when you show and hide a layer. Each layer has a "style" attribute which contains "display: inline" if it's visible, or "display: none" if it's hidden. We'll come back to this later but, for now, just remember that there's a "style" attribute that holds the "display" property directly on each layer.

Once your stack of drawings is complete, and your layer IDs have all been set, you'll need to save the file and open it in a text editor. It's time to add a block of CSS that will turn off the layers, then turn on only the layer specified in the fragment identifier. The placement of the CSS in the file doesn't matter too much, but I usually put it just after the opening <svg …> tag (shown below).

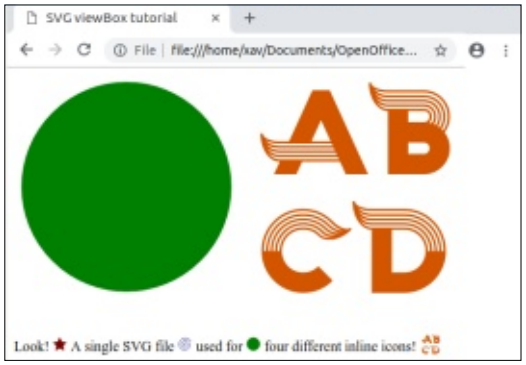
The "svg > g" syntax just means "this rule should affect only <g> elements that are immediate children of an <svg> element", so it doesn't affect any sub-layers. Otherwise, it's pretty similar to the HTML rules we used earlier, but with the addition of "!important"

```
<svg xmlns…
        …
        …>

        <style>
     svg > g {
       display: none !important;
     }


     :target {
       display: initial !important;
     }
   </style>
```

on the properties. This tells the browser that these rules should take precedence over those in the "style" attributes on the layer – without them our new block of code would be ignored. You could avoid them if you manually removed the properties from the style attributes on each layer, but as Inkscape will add them back in again if you ever need to edit the file, it quickly becomes a thankless chore. Sprinkling a few "!important" declarations around isn't particularly good coding practice, but it's a lot more practical.

With our SVG stack set up, we can reference the images with a fragment identifier in the HTML, in a similar manner to the named views we used last time (shown top right).

The resultant web page looks like this:




```
<!DOCTYPE html>

<html>
<head>
  <title>SVG viewBox tutorial</title>
</head>

<body>
  <div>
    <img src="stacks.svg#circle"></img>
    <img src="stacks.svg#text"></img>
  </div>

  <br />

  <div>
    <p>Look! <img src="stacks.svg#star" style="width: 1em;"></img>
    A single SVG file <img src="stacks.svg#spiral" style="width: 1em;"></img> used for
    <img src="stacks.svg#circle" style="width: 1em;"></img> four different inline icons!
    <img src="stacks.svg#text" style="width: 1.5em; vertical-align: bottom;"></img>
    </p>
  </div>
</body>
</html>
```

With SVG stacks, it's also possible to have another layer that sits behind all the others, but which is always visible, regardless of the fragment identifier. Consider a single background layer, with its ID set to "background". This additional CSS selector would ensure that it's always visible:

```
#background {
        display: initial
!important;
}
```

With named views and stacks giving broadly similar results, you might want to know which one is better to use. For many cases, either will work: named views arguably require more work to set up, but provide an advantage of being able to see all the individual images in Inkscape at once. Views also work better if your images are different sizes. Stacks, on the other hand, work well if your images require a single common background, or if it's easier to draw each image by stacking it on top of the previous ones to help with alignment. As is so often the case, the real answer is to try both approaches and see which one works best for you.

Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

Inkscape - Part 81

Over the past few months, we've looked at various tricks and effects that can be achieved when using an SVG file in a web page via the HTML <img> element. But using an SVG like that is subject to various security restrictions imposed by the browser, preventing you from referencing external files (fonts, css, linked bitmaps) or using JavaScript. So now we're going to move on to alternative methods of using SVG in a web page, which offer a little more scope for customisation.

This instalment will look at a little CSS trick for inheriting a color from your HTML page into your SVG. Unfortunately, the laws of CSS scoping mean that this one works only if you inline your SVG code right into your HTML. This requires some care and attention, but isn't really that difficult, as you can largely copy and paste the contents of your SVG file from within a text editor. It does bloat the size of your HTML, but, conversely, you make a saving in not having a separate SVG image to download. Let's start with a basic HTML page (top right).

Now we need some SVG to put into the file. For this demonstration, I created a simple rounded rectangle in Inkscape, then saved it as in "Optimised SVG" format to clear out a lot of the unnecessary metadata and other content. Even then, I was able to manually trim the resulting file a little further, as the namespace declaration isn't required, and I was left with a single <g> wrapping the content which serves no purpose in this case.

One option I did select in the save dialog was to "Convert CSS attributes to XML attributes". This breaks down the otherwise densely filled "style" attribute into a series of individual presentation attributes. You don't have to do that – this technique works just as well with the style attribute, or even a <style> section elsewhere in the XML – but having the styling split into separate attributes makes it a little clearer to explain what's happening.

```
<!DOCTYPE html>

<html>
<head>
  <title>SVG tutorial</title>
</head>

<body>
  <div>
    <!-- SVG goes in here -->
  </div>
</body>
</html>
```

The result of the export, manual trimming, and reformatting to fit the magazine, was the chunk of SVG shown below.

Copying this code and pasting it inside the in the HTML file results, as you might expect, in the browser drawing a rounded rectangle, with a dark red fill. No surprises so far.

The next step is to throw away the "color" attribute entirely: it's not needed in SVG, where we already have fill and stroke colors, but if it's left in place it will affect the CSS cascade and prevent the effect we're trying to achieve.

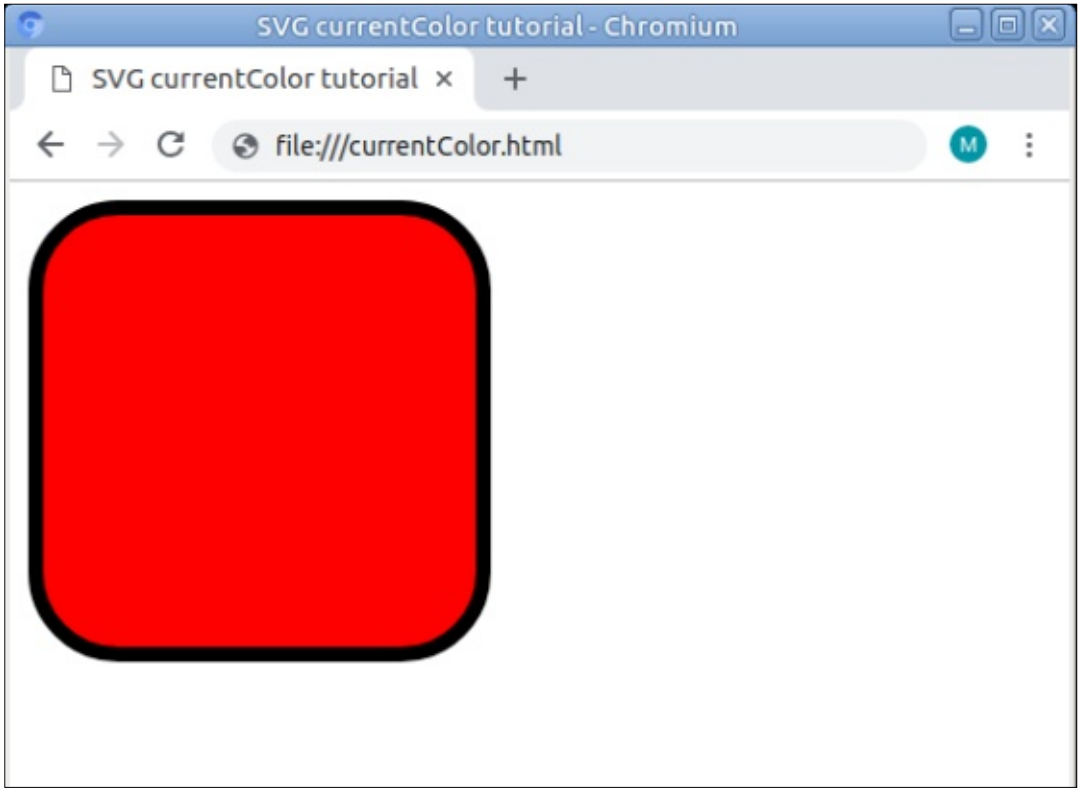
```
<svg width="250" height="250">
  <rect
    x="6" y="6"
    width="239" height="239"
    ry="44"
    color="#000000"
    fill="#800000"
    stroke="#000000"
    stroke-linejoin="bevel"
    stroke-width="8"
    style="paint-order:normal"
  />
</svg>
```

That effect is to get the SVG to use the current font color from the parent HTML page. We'll use it for the fill in this case, so that our rectangle basically becomes a giant color swatch displaying the browser's font color. To do this, we simply have to replace the value of the "fill" attribute with the keyword "currentColor" (note the spelling and capitalisation). Code is shown top right on this page.

Reloading the page will most likely show the rectangle filled with black. What else were you expecting? Black is the default color for text in an HTML page if you haven't styled things differently. But we can change that by setting the CSS "color" property on the <svg> element or, crucially, one of its ancestors. For example, let's change the :

```
<body>

<div style="color: red;">

<svg width="250"
height="250">
    ...
```

Now reloading the page in the browser gives the result shown below left.



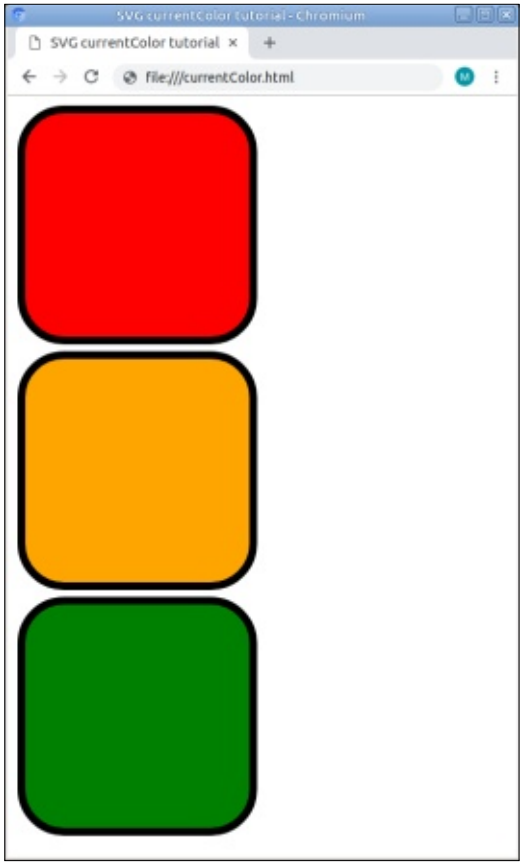
You'd be forgiven for not getting terribly excited by this, but take a step back and think about what you've achieved: you've set a color inside your SVG content based on a CSS value in your HTML. Still not getting it? Let's add an ID to the <rect> element, then create a couple more divs (below).

```
<svg width="250" height="250">
  <rect
    x="6" y="6"
    width="239" height="239"
    ry="44"
    fill="currentColor"
    ...
```

In case you hadn't guessed, the <use> element lets you re-use a snippet of SVG elsewhere, by referencing its ID in the fragment identifier part of the URL in the "href" attribute. In this case, we're referencing an element in the same file, so we don't need the full URL – just the fragment identifier (the ID preceded by a hash). So this code just tells the browser to render three copies of our <rect>,

```
<div style="color: red;">
  <svg width="250" height="250">
    <rect id="rect1"
    ...
   </rect>
  </svg>
</div>


<div style="color: orange;">
  <svg width="250" height="250">
    <use href="#rect1" />
  </svg>
</div>


<div style="color: green;">
  <svg width="250" height="250">
    <use href="#rect1" />
  </svg>
</div>
```

but the CSS in each , combined with the use of currentColor in the SVG, results in some rather cubist looking traffic lights:



Let's put this to a more practical use. How about icons for a website? Here (shown right) I've created four icons, each in a separate layer in Inkscape, and given each layer a descriptive ID. The details of the paths are omitted for brevity. The "color" attributes are removed, and the fill or stroke color set to "currentColor" as necessary. Then the whole SVG block is hidden using CSS in the <svg> element.

Now each individual icon in the set can be displayed on the page via a <use> element, with its size and color set on the SVG element that contains the <use>, or even on a parent element above that – as demonstrated with these couple of sections (next page, top right).

Of course the color could be set at a much higher level on the page, so it needs to be set only once for the whole page – or you could use CSS variables for the same effect. Now a change to a single color will alter all the icons used on your page: you've just created a means of applying a theme. Because the CSS "color" property also affects the text on the page, you can ensure that your icons are kept in sync with the text, whichever theme is selected. As a demonstration, suppose we use a block like this several times (next page, bottom right).

For each copy of this block, we'll change the color values in the . A little extra CSS adds a border that also has its color set to "currentColor", and for some of the copies we'll even put in a "background" property to produce an inverted look. Here's the result of a few minutes of copying, pasting, and editing some CSS – all with only a single copy of the SVG icons, each referenced multiple times:

```
<svg viewBox="0 0 4.23 4.23"
  style="display: none;">
  <g id="plus">
    <path d="…"
      fill="currentColor" />
  </g>

  <g id="cross">
    <g fill="none"
      stroke="currentColor"
      stroke-linejoin="bevel"
      stroke-width=".5">
        <path d="..." />
        <path d="..." />
    </g>
  </g>

  <g id="home">
    <g fill="none"
      stroke="currentColor"
      stroke-linejoin="bevel"
      stroke-width=".5">
        <path d="..." />
        <rect x="1.63" y="2.42"
          width=".98"
          height="1.32" />
    </g>
  </g>

  <g id="tick">
    <path d="..."
      fill="none"
      stroke="currentColor"
      stroke-linejoin="bevel"
      stroke-width=".5" />
  </g>
</svg>
```

I'll confess this trick of using currentColor in SVG is a limited one. The SVG has to be inlined with your HTML, and you can change only a single color. But, with a cleverly designed SVG file, it's possible to give the impression of something more sophisticated – by masking the colored element with a gradient, or using a filter to alter the color, for example. There is one interesting thing to note about this technique: it will work in Internet Explorer right the way back to version 9! If you need to theme some icons on a website, but still need IE support (so no CSS variables), this might be just the trick you need.

Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

```
<div>
  <svg viewBox="0 0 4.23 4.23"
    style="color: red; width: 25px;">
      <use href="#plus" />
  </svg>
</div>

<div style="color: orange; width: 50px;">
  <svg viewBox="0 0 4.23 4.23">
    <use href="#cross" />
  </svg>
</div>
```

```
<div style="color: red;">
  Theme 1:
  <svg viewBox="0 0 4.23 4.23">
    <use href="#plus" />
  </svg>

  <svg viewBox="0 0 4.23 4.23">
    <use href="#cross" />
  </svg>

  <svg viewBox="0 0 4.23 4.23">
    <use href="#home" />
  </svg>

  <svg viewBox="0 0 4.23 4.23">
    <use href="#tick" />
  </svg>
</div>
```

Inkscape - Part 82

Over the past few months, you'd be forgiven for thinking that this column has morphed from an Inkscape tutorial to a more general "SVG in HTML" series. In practice, I've been introducing a little background knowledge before delving into the (limited) JavaScript features that are already present in Inkscape. But that requires just a little more background information about JavaScript itself, and its use in SVG on the web…

JavaScript (JS) is the de facto programming language used in web pages. It's an implementation of a language called ECMAScript, so you might occasionally see that term mentioned. It's nothing to do with the Java programming language – it just shares a similar name thanks to someone in marketing at Netscape many years ago deciding that 'brand awareness' was more important than 'avoiding decades of confusion'.

Within a browser, JavaScript gives you the capability to write code that can modify the page and respond to interactions initiated by the user, or by external actions such as some data being pushed from a server. These triggers are referred to as 'events', and will form the core of the JS code we'll be writing in this series. The basic approach is that we'll use SVG to draw something to the browser window, then attach events to monitor for clicks, mouse movements, keypresses, and so on, each of which then triggers some JS code which can, in turn, modify the SVG.

Because JavaScript can read keypresses from the user, and can talk to a server, it raises security concerns. You could, for example, use SVG to create a beautiful image, but as soon as the mouse moves over it, your JS could redraw the image to look like a legitimate username/password box has opened on the screen. Anything typed into the box could be sent back to your server and used for your own nefarious ends. As a technically aware reader of FCM, you may not be fooled by such an obvious scam, but a huge number of people will happily enter their credentials into such a dialog as an almost Pavlovian response.

To prevent such attacks, browsers limit the ability of SVG to run JavaScript, depending on how the SVG has been included in the page. I've talked about this previously, but it's worth recapping:

SVG in an <img>:
This is how images are usually displayed in a web page, and is used in countless bulletin boards and social media sites. Because anyone can upload any image, there's a huge potential security hole, so JavaScript in SVG is blocked entirely.

SVG as a CSS background-image:
Although less frequently used as a way for users to upload images, the code paths used in browsers are pretty much the same for CSS images as for <img> elements, so the previous rule still applies: no JavaScript.

Inline SVG:
This requires the actual code of the page to be edited, so it's assumed that the work is being done by someone in a trusted position, and therefore the browser allows JavaScript.

SVG in an <object>
This is the W3C standard way to include "foreign" content into a web page – including Flash, Java applets, and other potentially dangerous code. As such, it's always had a more lax set of security rules than <img>, and no sensible website developer allows user-uploaded content to be displayed in an <object>. Therefore it's considered to be something that is added only by someone in a trusted position, and JavaScript is allowed.

SVG in an <iframe>:
Using an <iframe> has a simple syntax, similar to an <img>, but still allows JavaScript like an <object>. I tend to use <object> as that's the officially recommended approach by the W3C, but there are times when an <iframe> is a

better option.

There's one final way to display an SVG image in a browser which doesn't involve embedding it into an HTML file in any way, and that's simply to load the SVG file directly. For an SVG file on your local machine, you can just press CTRL-O and find it in the file selector. For one sent by a web server, the browser's URL field just has to point directly at the SVG image, and the browser will load it in the same manner as if you pointed directly at a PNG or JPEG file…

…except it won't. Not unless the server has been configured correctly. Which is a whole other story of politics and pain in which countless users and developers suffer from an ideological disagreement at a technical level. Brace yourself, this is going to get petty!

Serving an SVG file isn't terribly tricky. Your web server has to be configured to send the right MIME type (a header that tells the browser what sort of file it's receiving), but that's usually a small configuration change. If you've got direct control over the configuration of your server, search online for some appropriate terms (e.g. "Apache SVG MIME"), and you should find suitable instructions. If your server is managed by someone else – such as the typical case of a website hosted by an ISP – first try putting an SVG image onto your site and accessing it, as there's a good chance the configuration has already been done. If the file appears as text, the browser tries to save rather than display it, or there's a message suggesting the browser's treating it as an XML document, you'll need to raise a support request with your host.

Where it gets more complex is with SVGZ files – "compressed SVG" in Inkscape's terms. These are literally just SVG files that have been compressed using the Gzip algorithm, and you can get the same effect by using the gzip program on your Linux box:

```
gzip -k image.svg

mv image.svg.gz image.svgz
```

The first line creates a gzipped version of "image.svg" but doesn't overwrite the original file (due to the -k switch). Gzip defaults to simply appending ".gz" to the filename, so the second line renames the file to the standard ".svgz" (this could also be done directly with the "--suffix" switch to gzip). The resultant file can be directly loaded into Inkscape for further editing – it's indistinguishable from a "compressed SVG" file saved from Inkscape itself. On the surface, SVGZ seems like a great format, as it's much smaller than an equivalent SVG file, but you can still open it in Inkscape, or even convert back and forth from the command-line if you do want to edit the XML content by hand. The problems come when you try to put an SVGZ file online.

The W3C working group that created SVG thought, quite rightly, that defining a compressed form of the format as part of the spec would be a worthwhile addition, especially back in 2001 when storage space and bandwidth were more expensive. Gzipping of content on-the-fly was already a standard feature of the web, so browsers had decompression code in place, making for an obvious choice of algorithm. Unfortunately, this is where an ideological divide took place: rather than treat SVGZ as a format in its own right, the browser vendors opted to natively support only uncompressed SVG.

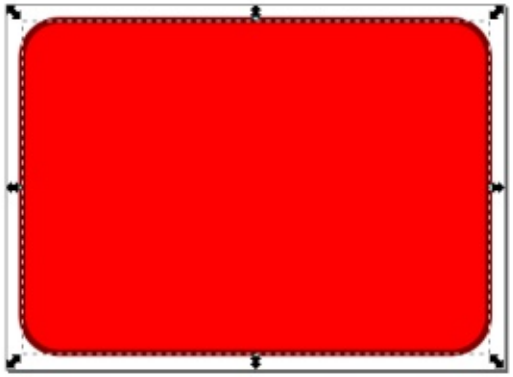
But saying that is like stating that browsers support only uncompressed HTML or CSS. In practice you can send any supported format with on-the-fly Gzip compression, provided your web server correctly sets the "content-encoding" header. This also means that you can send a pre-compressed SVGZ file if you also provide that header – the browser just thinks you've sent an SVG file using on-the-fly compression. Once again, search online for the instructions for your web server, or raise a support request with your ISP if necessary.

The summary, therefore, is that browsers don't really support SVGZ, but with the right server configuration, you can trick them into using those files nevertheless. It also explains why you can't load an SVGZ file directly into your browser from the local filesystem – if the file doesn't come from a web server, there's no "content-encoding" header, and the browser decides to play dumb. This situation could easily be fixed if browsers opted to treat SVGZ as a first class file format, and

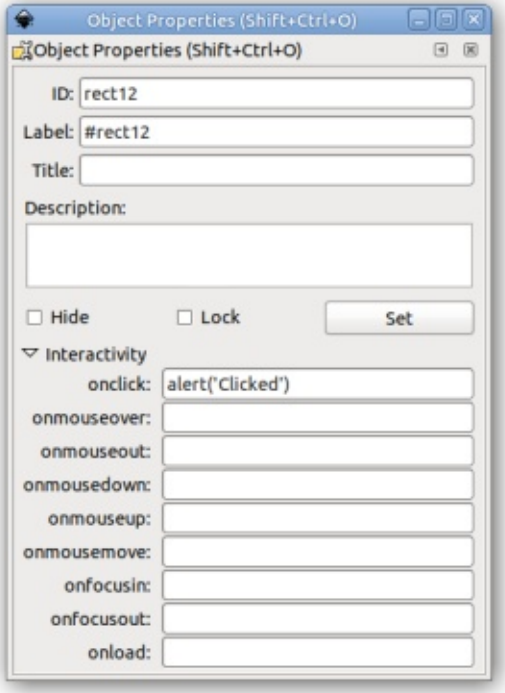
automatically unzip it even in the absence of the header. But as the situation is unlikely to change, I recommend sticking with SVG files and using on-the-fly compression from your web server, rather than trying to work directly with SVGZ files.

Personally, I think the browser vendors are wrong on this one. JPEG images, for example, are essentially just arrays of pixels that are compressed using a "discrete cosine transformation" (DCT) algorithm. Yet browsers don't insist on a "content-encoding: DCT" header to display a JPEG. The philosophical difference between a file that has been compressed using Gzip by the server, and one that has been natively stored in a gzipped format, is a subtle one. But the result is that users suffer from the complexity and confusion of not being able to directly load an SVGZ file into the browser, even though that format has been explicitly sanctioned by the SVG Working Group.

To begin our journey into the world of Inkscape and JavaScript, I'll assume that you are able to load an Inkscape-created SVG file into your web browser, either from a web server or from the local filesystem. Later on, we'll look at some differences that apply when you use <object>, <iframe>, or inline SVG, but, right now, let's keep things self contained in a simple SVG file.



Remember those JavaScript 'events' I spoke of earlier? Let's use Inkscape to add some JS code that listens for a "click" event – the result of the user clicking on an object in our image. Create a new image, draw a simple object, then right-click on it and bring up the



Object Properties dialog. At the bottom of the dialog is a series of fields, all with labels that start with the word "on". If they're not visible, you'll need to click on the "Interactivity" label to expose them. In the "onclick" field, enter the following JavaScript code:

```
alert('Clicked')
```

Save the file and load it into your web browser. You should see the object you drew in Inkscape. Click on it to confirm that the browser presents you with a dialog that contains the word "Clicked".

This type of dialog, referred to as 'an alert', is the simplest form of output from JavaScript. You can display only a single string, and you can't change the layout of the dialog or the label on the button. But writing even this most simplistic of code is a useful first step in any JavaScript application: it proves that Inkscape, your browser, and your web server (if you have one) are all working as expected, and it confirms that your code can respond to mouse clicks, which is a basic requirement for almost any interactive site.

The single line of code you wrote above does one thing: it calls a function named alert() when the user clicks the left mouse button (or taps) on the object to which you attached your code. The function is given a single parameter – a string containing the word "Clicked" – which it displays on the screen in a dialog. Let's see how that code in Inkscape manifests itself in the SVG file. Open the SVG file in a text edito and, towards the bottom of the file, you should find something similar to the code shown on the next page, top right.

You might have a different element than a <rect>, depending on what you drew – and therefore may have other attributes (the "rx" and "ry" attributes govern the roundedness of a rectangle's corners, for example). I've also significantly abbreviated the "style" attribute. But the thing to note is the "onclick" attribute, which contains the JavaScript we typed into the dialog in Inkscape earlier.

It's worth getting familiar with the way that your JS appears in the file. Whilst the single-line text boxes in Inkscape are okay for typing very short amounts of code, if you need something even slightly more substantial, it's often easier to edit the SVG directly. Here's a modified version of my object (with extraneous attributes omitted), to show how you might deal with multiple lines:

With those edits in place and saved, reload your page, and click on your object again. This time you should see a series of three alerts.

```
<rect
        ...
        onclick="alert('clicked');
                alert('A second alert');
                alert('Note the semicolons!');"
        ry="11.49259" />
```

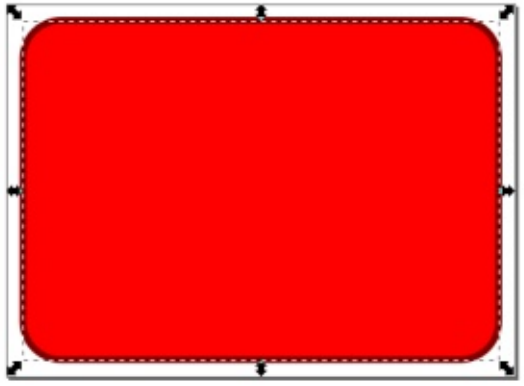
Unfortunately, edits made like this don't reflect well back in the Inkscape UI. Your three lines will be present, but all put onto a single line, and with any white space that you used to align them included in the line. Generally it's easiest to edit code in either a text editor, or in Inkscape, but not to go back and forth between them.

As you've guessed from the Inkscape UI, there are other events you can react to. But, in most cases, using the alert() function will prevent you testing correctly. Consider trying the onmousemove option, which is supposed to fire events continuously as your mouse moves over your object: as soon as your mouse moves over the object you'll get an alert which you'll need to dismiss before you can continue; then another, and another, each time your mouse moves over the object, with you having to manually dismiss each one in turn. Hardly

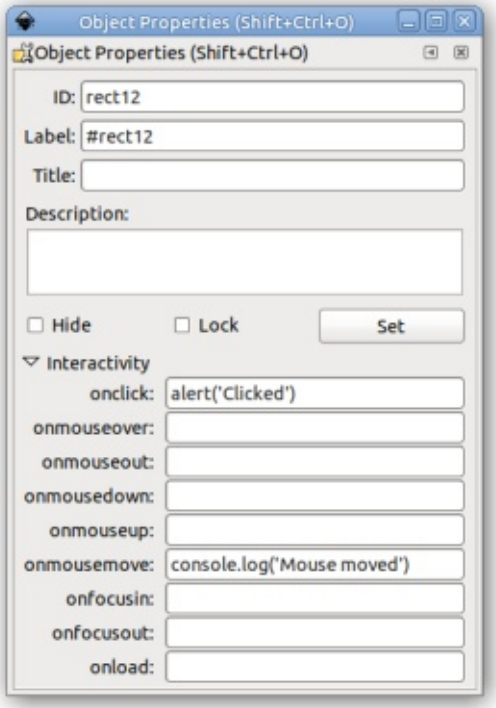
```
<rect
        style="fill:#ff0000;..."
        id="rect12"
        width="142.11905"
        height="101.29762"
        x="14.363094"
        y="19.565475"
        onclick="alert('clicked')"
        ry="11.49259" />
```

the constant stream of events you were interested in.

Back in the dim and distant past, debugging by throwing up alert messages was the de facto way to develop with JavaScript, but, thankfully, the tools have moved on a lot since then. Modern desktop browsers all have a developer toolbox which you can usually open by pressing F12.



There are a variety of tools in here, but the one we're interested in is the console – there should be a tab for it somewhere near the top of the toolbox. In Inkscape try adding a console.log('Mouse moved') call to the onmousemove section of the object properties:



Now, with the file saved and the developer console open, reload your file in the browser. Clicking should throw up an alert, as before, but moving the mouse around over your object should generate a stream of messages in the console. Actually you're likely to only see one message, plus a count to the right of the console indicating how many times the message has been logged. This is a convenience in modern tools to avoid filling your screen with duplicate messages. If you really want to see them streaming by, you can add a random number to your log entry so that each one becomes unique:

```
console.log('Mouse moved',
Math.random())
```

This demonstrates another huge advantage of console.log() over alert() – you can give it multiple parameters, and they don't all have to be strings.

That's a very basic start to adding some interactivity to an Inkscape file. We'll be exploring this topic a lot more over the coming months, so please do try the simple exercises above so that you've got a good basis to build on as we make our events do more interesting things than just printing some text to the screen.

A BLATANT PLUG!

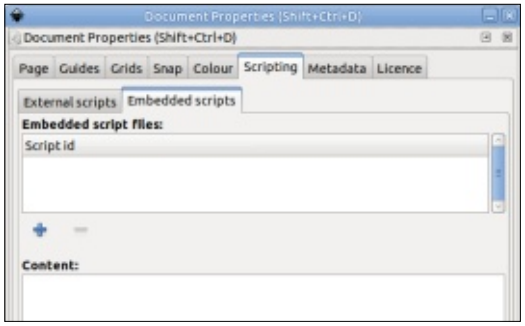
Mark and his colleague Vince have been using Inkscape and MyPaint to create the monthly Elvie cartoon strip, first in Linux Voice, then in Linux Magazine (Linux Pro Magazine in the US), for five years now. To celebrate this anniversary, Mark has written an article in issue #220 of Linux (Pro) Magazine which describes the process they use in some detail. If you're interested in reading about the practicalities of creating a cartoon using FOSS, this issue should still be current by the time FCM#142 comes out, but it's also available to buy as a digital edition from http://www.linux-magazine.com/

Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

Inkscape - Part 83

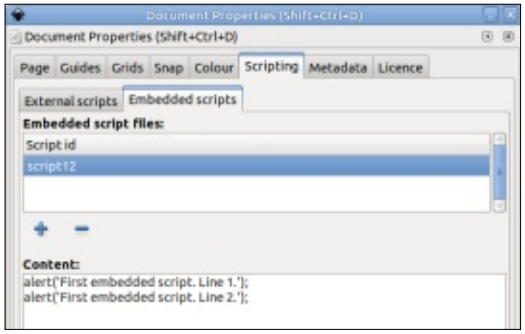
Last time, we looked at some very basic JavaScript to alert or log a message when you click on an object in your drawing, or move the mouse over it. The one-line boxes in the object properties are okay for such short snippets of code, but you wouldn't want to write anything too complex in there. Instead, Inkscape has a couple of other mechanisms for using larger amounts of code in your page: embedded scripts and external scripts.

Both of these features live in the Document Properties dialog, so open that first (File > Document Properties, or CTRL-SHIFT-D), then switch to the "Scripting" tab. Within that area are two other tabs; in this article we're going to look at the second one, "Embedded scripts".



In my opinion, there are a few issues with the UI in this dialog. You might think you can just go ahead and type some JavaScript into the "Content" section, but that won't actually create anything in your file. Instead you must first either select an existing entry from the "Embedded script files" section, or create a new one using the "+" button and then select it. I don't know why the Content section isn't disabled until something is selected, nor why a newly created entry isn't selected by default, but so long as you remember that anything typed into the bottom section will be lost unless there's an entry selected in the top, you'll be okay.

Let's create a new entry by clicking the "+" button, then select it and enter a little JavaScript into



the bottom. We'll just call the alert() function a couple of times at this point.

Note that our alert() calls finish with semicolons, so the JavaScript interpreter knows where one statement ends and the next begins. Save your file, and open it directly in a web browser. You should immediately see two messages appear, even before the content of your document is rendered. JavaScript statements entered like this – outside of any function – are part of the global scope, and are executed as soon as the file is loaded.

Now repeat the process to create a second embedded script file, with similar alert() calls, but the message changed to 'Second embedded script…'. Save your file, and reload it in your web browser (F5). You should see four messages displayed in succession – but, if you read the details, you'll notice that the ones from your second script are displayed first! This is something to be very careful with: the scripts appear in the XML file, and are therefore processed by the browser, in the order they appear in the list, not in the order you created them. In another UI faux-pas, however, it's not possible to re-order the scripts in this dialog.

One way around this problem is to just use a single embedded script, and manually reorder your lines in the Content box. Multiple scripts are all just concatenated together by the browser anyway, so whether you use a single script, or a hundred, it doesn't matter from a JavaScript perspective. Note, however, that the Content box is a little short, and can't be resized (another UI fail). If you want to put a lot of code into your file, then, being able to see only a tiny sliver of it at a time will make it rather difficult to work on.

Usually, the ordering of the scripts isn't too much of a problem, as JavaScript code is typically arranged into functions. The order in which the code executes then depends on the sequence in which the functions are called, not the order they appear in the file. This

also goes some way to explaining why the fields in the Interactivity section of the Object Properties dialog allow only a single line of code: typically they have to make only a single call to execute a separate multi-line function. As an example of this, let's use a function in an embedded script to change the fill color of an object when it's clicked on.

In a new file, create a simple object – a square or circle – with a visible stroke and an obvious fill color. I've stuck with the red rounded rectangle I used last time. Now create a new embedded script file, with the following content:

```
function change_to_blue(elem)
{
  elem.style.fill = 'blue';
}
```

We've created a function named change_to_blue() which takes a single parameter that we've chosen to call "elem" as it represents a single XML element in your file. This parameter will be a reference to the object you click on, and the body of the function just sets the fill color on that object to "blue" (a valid CSS color name). By putting our fill change inside a function we prevent it from running as soon as the page is loaded. Instead we have to explicitly call it from somewhere else in our file. That somewhere else is the "onclick" field of the Object Properties dialog, which I covered last time. To call our function, we simply have to invoke it by name, but we also need to pass a reference to the object you clicked on. JavaScript has a keyword, "this", which means different things in different contexts – but in the case of a simple event handler like this, it gives us the reference we need. Therefore the line to put into the onclick field is this:

```
change_to_blue(this);
```

Save and reload your file, then click on your red object. It should turn to blue. See, interactive SVG isn't so tricky after all!

Try creating more objects, each with a different fill color, but each with the same line in their onclick field. Notice that clicking each one changes the color of only that specific element, thanks to the "this" keyword.

Rather than just set the color to blue, how about creating a toggle between two colors each time the object is clicked. The code's pretty straightforward: we just test to see if the fill color is currently 'blue' and, if so, set it to 'red'. Otherwise we explicitly set it to blue. Here's the code:

```
function change_to_blue(elem)
{
  if (elem.style.fill ===
'blue') {
    elem.style.fill = 'red';
  } else {
    elem.style.fill = 'blue';
  }
}
```

If you're not familiar with JavaScript, be particularly aware of the '===' in the 'if' statement: this triple equals means "are both the value and the type of the variable identical?" It's a more robust check than double equals ("are the values effectively the same, even if the types are different"), and is not the same at all as a single equals, which is used for assigning a value to a variable, not for testing it.

This new code is all well and good, but it would be better still if, instead of simply toggling between blue and red, we toggled between blue and whatever color the object previously had. To do this we need to store the old value of the fill color before we change it to blue, then use that stored value when we turn it back again. Fortunately for us, the "elem" reference that is passed in ("this" on the calling element) is a JavaScript 'Object' (not the same as an object you draw in Inkscape), which can hold additional custom properties. We'll dynamically create a new property, called 'previousFill' to hold the value of the fill just before we change it. Our toggling code becomes this:

```
function change_to_blue(elem)
{
  if (elem.style.fill ===
'blue') {
    elem.style.fill =
elem.previousFill;
  } else {
    elem.previousFill =
elem.style.fill;
    elem.style.fill = 'blue';
  }
}
```

In the "else" section we store the old fill in our 'previousFill' property; in the "if" section we use that value instead of the string "red". Strictly speaking, we should probably also rename the function to toggle_fill() or something similar – but that suggests we could toggle to a color other than blue, which the code doesn't do at

the moment.

Let's extend it a little further so that we can toggle to a different color. By taking an optional second parameter we can let the calling code determine what the toggle color should be, but still fall back to blue as a default. The toggling code becomes this:

```
function toggle_fill(elem,
color) {
  if (color === undefined)
color = 'blue';

  if (elem.style.fill ===
color) {
    elem.style.fill =
elem.previousFill;
  } else {
    elem.previousFill =
elem.style.fill;
    elem.style.fill = color;
  }
}
```

Note that we test the color variable to see if it's the special value 'undefined'. Note that this is a primitive type in JavaScript, like 'Number', 'String' or 'Object', so we're testing to see if color is this special type, not testing to see if it's a string containing the word "undefined". That's why there are no quotes around the word in the code.

Whenever a parameter is missing in a function call, the corresponding value in the receiving function is given a value of 'undefined'. By explicitly testing for this, we can therefore decide what to do if the parameter is omitted – in this case use a default value of 'blue' instead. There are various ways to handle missing and default parameters in JavaScript, but this particular syntax is clear, robust, and works even in older browsers.

With this default value in place, the calling code can be any one of these examples:

```
toggle_fill(this);
toggle_fill(this, undefined);
toggle_fill(this, 'yellow');
toggle_fill(this, 'red');
```

This toggle_fill() function can therefore work with just a single parameter – in which case 'color' is undefined and gets set to 'blue' – or with two parameters. If the second parameter is explicitly set to 'undefined' then it's the same as using just one parameter; otherwise the value will be used to set the fill color. But see how already, with only a short function like this, we've exceeded the number of lines in the Content box in Inkscape. I hope you like coding through a letterbox!

Although this function is a lot more flexible than our original creation, you should be aware that not all valid CSS colors will work in this type of code: rgb() values might get returned by the browser as hexadecimal strings, for example, or the it might change the case, either of which will stop the equality test from working. Code like this, which makes assumptions about data without testing those assumptions rigorously, is fragile and easily broken. But writing this code in a less fragile way isn't easy, and is certainly outside the scope of this tutorial series. For now you can play around with the code just to get a feel for embedding JavaScript into your SVG files. Next time, however, I'll show you how to use CSS classes, in conjunction with JavaScript, to make toggling fill colors (and other styles) far more robust.

Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

Inkscape - Part 84

This month, we're going to look at using CSS classes with your SVG in order to simplify the JavaScript you have to write when you want to change the style of your objects interactively. Note that we're still talking only about changes that can be achieved using CSS styles – fill and stroke colors, line thickness and similar. Changing other aspects of your objects – such as the shape of a path – can't be done simply by using CSS classes.

First of all, what is a CSS class? In short, it's simply a way to group similar objects for styling purposes, by giving them all the same class name. Consider this super simple SVG file, consisting of one red rectangle (top right).

You've seen previously that we can remove the values in the 'style' attribute and put them into a <style> block elsewhere in the document (bottom right).

The <style> block contains CSS declarations, consisting of a 'CSS selector', followed by a number of rules that will be applied to any elements in the file that match the selector. In this case, the selector is just the word 'rect', meaning that the rules applied here will match any <rect> elements in the file. Alternatively, we could use the id of an element, prefixed with a hash character, to make the rule apply to only the single element with that id:

```
<style>
#rect1 {
  fill: #ff0000;
  ...
```

We could also add a 'class' attribute to the element, then use the class name, prefixed with a dot, as the CSS selector (bottom left).

In this case, I've used the class name 'important-thing', since the styling of a red fill with dark red background suggests this might be

```
<svg xmlns="http://www.w3.org/2000/svg">
  <rect
    id="rect1"
    width="150"
    height="100"
    x="15" y="20"
    style="fill: #ff0000;
      stroke:#800000;
      stroke-width:10;"
  />
</svg>
```

```
<svg xmlns="http://www.w3.org/2000/svg">
  <style>
  rect {
    fill: #ff0000;
    stroke:#800000;
    stroke-width:10;
  }
  </style>

  <rect
    id="rect1"
    width="150"
    height="100"
    x="15" y="20"
  />
</svg>
```

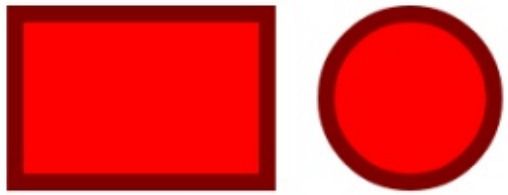
```
<svg xmlns="http://www.w3.org/2000/svg">
  <style>
  .important-thing {
    fill: #ff0000;
    stroke:#800000;
    stroke-width:10;
  }
  </style>

  <rect
    id="rect1"
    class="important-thing"
    width="150"
    height="100"
    x="15" y="20"
  />
</svg>
```

used to indicate important elements. But the class can be anything you like, provided you use only alphanumerics, underscores and hyphens. Note that you can't use spaces in class names, for reasons that will become clear later.

In this example file, we've not really gained much by using classes. The end result is the same whether we use a style attribute, or any of the three CSS selectors we've looked at, because there's only one element in the file that can be affected. But what if we were to add a second element to our file:

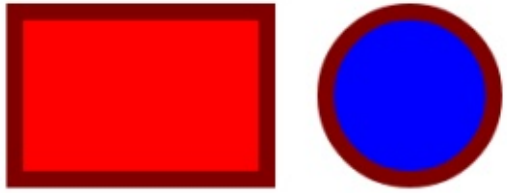
```
<circle
  id="circle1"
  class="important-thing"
  cx="250"
  cy="70"
  r="50"
/>
```

This is a different type of object, so styling it through a <style> using the element selector 'rect' won't work. It has a different id (as it must, because ids have to be unique in an XML document), so an id selector won't work either. But classes don't have to be unique, so we've given this circle the same class as our earlier rectangle. Here's the result, two objects sharing a single style:



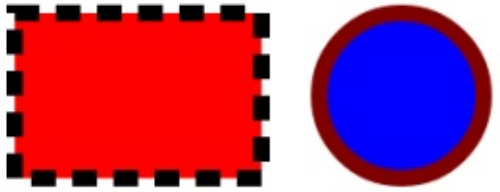
It's important to note that any styles you wish to set using a <style> block must not be present in the XML of the object itself. The inheritance rules of CSS dictate that styles set directly on elements usually take precedence over those set elsewhere. But this does give us the ability to override styles on individual elements. If we want a different fill color on the circle, we can simply set it in the style attribute, but still inherit the stroke width and color via the class:

```
<circle
  id="circle1"
  class="important-thing"
  style="fill: blue;"
  ...
```



You can, of course, have multiple classes defined in the <style> section of a file – but you can also apply multiple classes to a single object, by listing them all in the 'class' attribute, separated by spaces (the reason why class names themselves can't contain spaces). Suppose we were to add another class, then modify the attribute for the rectangle:

```
<style>
  ...previous styles...

 .black-dashes {
    stroke: black;
    stroke-dasharray: 15,10;
  }
</style>

<svg ...>
  <rect
    id="rect1"
    class="important-thing
black-dashes"
    ...
```



As you can see, the 'important-thing' class has been applied, giving the rectangle a red fill, but the 'black-dashes' class has overridden the stroke. It's vital to understand that the 'black-dashes' overrides the other style because it's declared later in the <style> block, not because of the order in which they're put in the 'class' attribute. On the one hand, this is a limitation, in that you can't trivially change the order in which the classes apply by modifying the attribute – but on the other hand it does mean that you can programmatically alter the content of the class attribute more easily, as you don't have to worry about preserving the existing order.

Unfortunately, Inkscape doesn't have any native support for creating and modifying CSS classes. Any edits you make to an element will be applied directly to its own 'style' attribute, or to other attributes directly on the element. Such changes won't delete or change your <style> block, and won't alter the 'class' attribute on the element, so at least Inkscape doesn't completely destroy any manual edits you've applied. But, as we've seen, values set directly on an element will take priority over those applied via a class, so you can easily end up in a situation where your classes no longer appear to have any effect. For this reason, I recommend doing any

work with classes in a text editor, rather than Inkscape, and making those changes as late in the design process as possible. Ideally you won't have to re-open your document in Inkscape at all, but, if you do, take care not to change the styles of any elements that you expect to control using classes – or at least be prepared to re-edit the files in your text editor afterwards.

JavaScript offers a few ways to work with CSS classes, but by far the easiest is the 'classList' property. This has add(), remove(), toggle(), replace() and contains() methods that handle all the corner cases and error handling for you. You can add() a class without having to check if it's already there. You can remove() a class and the code won't throw an error if the class doesn't exist. The toggle() method will add the class if it's missing, or remove it if it's present, which can simplify the code for basic on/off styling. You can use replace(), as the name suggests, to swap one class for another, and contains() does a search of the class attribute to tell you whether or not the name you supply is already present.

Let's finish up by using some of these methods in a new Inkscape drawing. First create a few objects whose classes you wish to alter: for this demo I'll have three buttons to demonstrate some different approaches to the problem of toggling between two states.

After saving, it's safest to close Inkscape before opening the SVG file in a text editor to make the CSS edits. In this case, we first want to add a couple of classes that will contain the styles we wish to switch between. For this demo, each one will contain only a fill color, and I'll add a third class for the stroke. Because the stroke won't be changing when we toggle, I don't really need to add a class for it at all, but it helps to demonstrate that the classList methods work even when there's more than one class applied. Here's the code that gets added after the opening <svg> tag:

```
<svg ...>
  <style>
    .red { fill: #ff0000; }
    .blue { fill: #0000ff; }
    .black-stroke {
      stroke-width: 2;
      stroke: #000000;
  </style>
...
```

Now we need to find each <rect> in the file and remove the fill, stroke and stroke-width properties from the 'style' attribute (or remove the corresponding attributes, if Inkscape has been configured to use presentation attributes rather than CSS styles). If you save the file at this point, and re-open it in Inkscape, you should see that the buttons now have an unset fill and stroke, with a default stroke width of 1. Quit Inkscape without making any changes.

Back in the text editor, add a 'class' attribute to each <rect> so that they use the classes defined above for one of the fill colors, and the stroke. Here's an abridged example of how one of them might look:

```
<rect
  style="display:inline; ..."
  id="rect10"
  class="red black-stroke"
  ...
/>
```

At this point, Inkscape still claims the rectangles have an unset fill and stroke. But we can preview our work by opening the file in a web browser, which honours the class and style work we've done, and displays our buttons with a red background and thick black border. We could add the script directly in the SVG file with a text editor but, since this is an Inkscape column, that's where we're going to add our JavaScript code. Just take care not to change any styles when you open the file.

With the file open in Inkscape, we'll need to add an onclick handler to each element. Our first example is going to be a simple one-liner that just toggles the 'blue' class on and off. Because of the inheritance model in CSS, this will have the effect of overriding the 'red' class, so toggling will switch from red to blue and back. Right-click on the first button, select 'Object Properties' and expand the 'Interactivity' section of the dialog. In the 'onclick' field,

type this:

```
this.classList.toggle('blue')
;
```

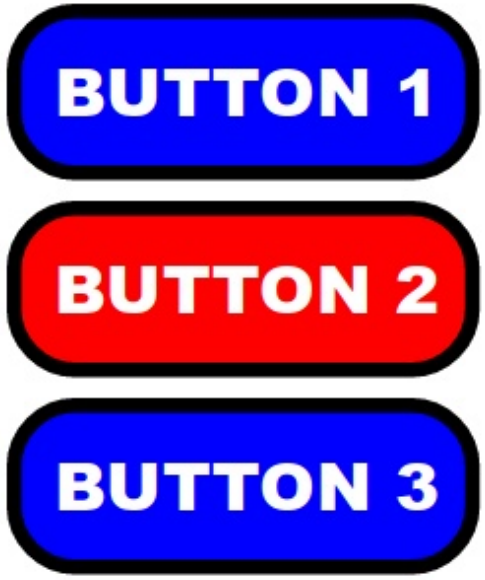
You can save the file, load it in a web browser, and confirm that it works, if you wish. For our second and third options we want more than just a single line of code, so we'll create a pair of functions as embedded scripts. We'll call them 'toggle1' and 'toggle2', and pass 'this' as a handle to the element that was clicked on. Add this line to the 'onclick' field in the object properties for the second button – and add an equivalent for 'toggle2' to the third button:

```
toggle1(this);
```

That's the code to call the functions when the buttons are clicked, but now we need the functions themselves. Open the Document Properties and select the Scripting > Embedded Scripts tab. Click the '+' button to add a new script, select it from the list, and put the toggle1 function in the Content area (top right).

Here (bottom right) is the code for toggle2. You can either add this function after toggle1, or put it into a separate script section by clicking the '+' and selecting the new entry.

Save the file, open it in your browser, and you should be able to click each button to toggle it from red to blue.



Look back over the code, and try to understand the differences between these three approaches. The first is the simplest, and will work in most cases where you just have to turn a class on or off from a single location in the code. The second is more useful if there are other bits of code that could interfere with the content of the class attribute, as the extra check

```
function toggle1(elem) {
  if (elem.classList.contains("blue")) {
    elem.classList.remove("blue");
  } else {
    elem.classList.add("blue");
  }
}
```

```
function toggle2(elem) {
  if (elem.classList.contains("blue")) {
    elem.classList.replace("blue", "red");
  } else {
    elem.classList.replace("red", "blue");
  }
}
```

to confirm if it currently contains the class or not makes it more robust. The third method is rarely used, but might be handy if you need to replace an entire class to avoid inheriting any properties from it. Usually I'd recommend reorganising your classes to avoid this problem, but the replace() method is worth knowing about in case that's not an option.

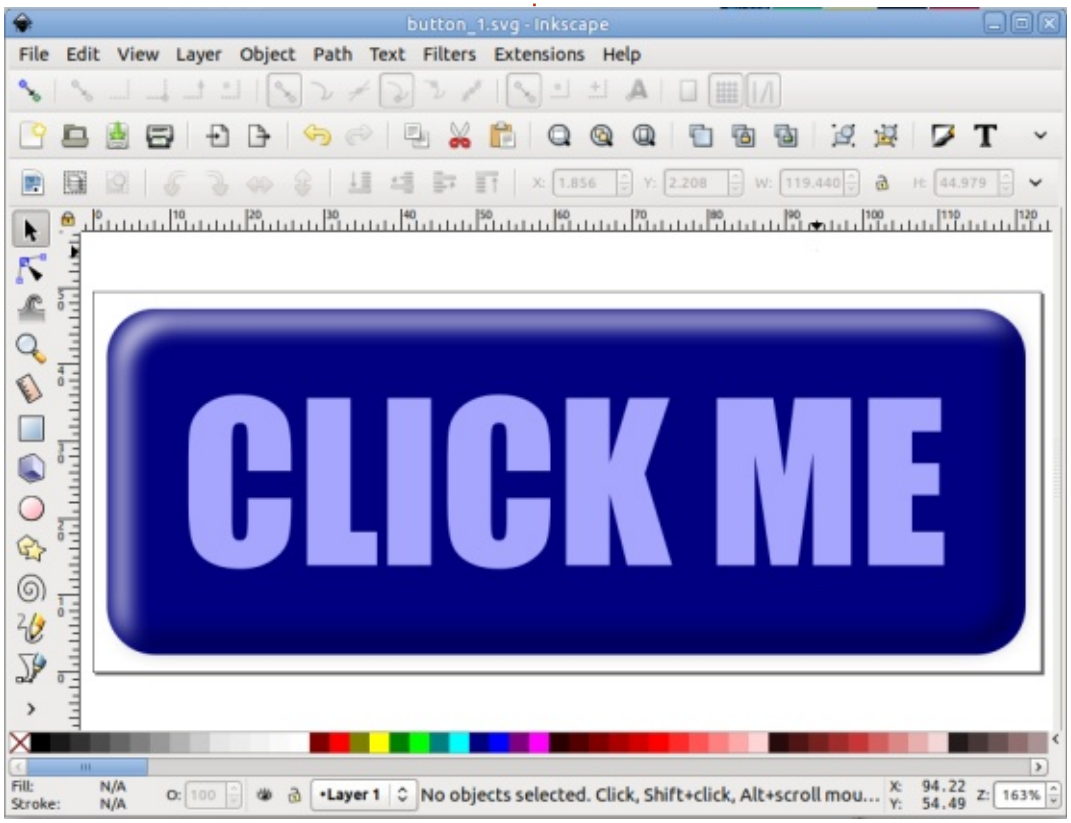
You may notice that clicking on the text, rather than the background, of the button does not cause the class toggle to occur. Furthermore the text is still selectable, which is not something you would usually want in a button. Next time we'll take a look at these issues to see how we can make a click on one object affect a completely different one.

Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

Inkscape - Part 85

A quick recap on where we got to last time: using CSS classes, we were able to toggle the style of a button by clicking on it. But it worked well only for a simple button with no content. If you clicked on the text inside the button, the toggle wouldn't work. And you were still able to select the text, which is less than ideal for a button. What we really want to do is to group together several objects, and have a click on any of them act as a trigger for the button. And, with a little more CSS, we can deal with the selectable text issue, too. So let's begin in Inkscape, by designing a fancier button…

This button is made up of four objects, each of which has a solid fill color so there are no stroke colors to worry about. When



Object	Normal	Clicked
Rounded rectangle	#000080	#800000
Top/left blurred path	#ffffff	#000000
Bottom/right blurred path	#00002e	#ffffff
Text	#a6a6ff	#ffa6a6

toggled, we'd like to change the background and text colors, and give the impression of the button being 'pressed in' by making the top/left object dark and the bottom/right one light. We can therefore draw up a small table showing the type of each object and the colors it will adopt in each state.

We're going to put the entire button into a single group. This isn't strictly necessary, as Inkscape's layers are already SVG group objects, but does make it a little easier to work with if we want to add more than one button (or other objects) to a single layer.

Now it's time to set up the CSS classes. The basic idea is that we will give the outer group a class of 'button', and then toggle an additional 'clicked' class on and off using one of the techniques from last time. Let's start by looking at the structure of the button, as created by Inkscape, but with most of the attributes removed for clarity:

```
<svg>
...
  <g inkscape:label="Layer
1">
    <g id="g972">
      <rect id="rect10" />
      <path id="path31" />
      <path id="path874" />
      <text id="text958">
        <tspan
id="tspan956">CLICK
ME</tspan>
      </text>
    </g>
  </g>
</svg>
```

It's all pretty much as you might expect: a <g> (the layer in Inkscape) containing a <g> (our button) which contains the four objects from the table above. There's a slight oddity in terms of the <text> object containing a <tspan>, but that's just down to the way SVG handles text. If we had multiple lines of text in our button, this might make more sense, as all the lines would then be separate <tspan> elements contained within a single <text>, but, even with only one line, Inkscape still uses a <tspan> even though it's not strictly necessary.

To keep the CSS a little clearer, it's best to change the IDs of some objects, or give them classes to better describe what they do. Otherwise, trying to remember which <path> is which at some point in the future becomes a problem. I tend to use classes for these sorts of labels, so they can be reused in other parts of the file as well. After all, you might want a second button to also have a 'top-left' path, so using that string as an ID would become prohibitive (remember, IDs have to be unique in a document, classes don't).

With the addition of a class for the button group, and one each for the paths, we've basically got this structure:

```
<g class="button">
  <rect />
  <path class="top-left" />
  <path class="bottom-right"
/>
  <text>
    <tspan>CLICK ME</tspan>
  </text>
</g>
```

Now it's time to add our CSS rules. We'll use the immediate child selector (>) to ensure that these rules apply only to elements inside our button, so there's no danger of all the text in the document becoming blue. Compare these rules with the table earlier in this document and you should be able to see what they're doing:

```
<svg>
...
<style>
.button > rect {
  fill: #000080;
}

.button > .top-left {
  fill: #ffffff;
}

.button > .bottom-right {
  fill: #00002e;
}

.button > text {
  fill: #a6a6ff;
}
</style>

<g inkscape:label="Layer 1">
...
</g>
</svg>
```

Don't forget to also remove any explicit fill and color properties in the 'style' attributes of your elements (including the <tspan>), so that the CSS rules you've added aren't overridden. If you load the image into a web browser, it should look the same as the original version in Inkscape. If you're not sure that your styles are working, or that you've removed all the overriding properties on the elements themselves, try temporarily changing the colors in the CSS to other values and confirm that it has an effect when you reload the page.

Take a careful look at each CSS rule to make sure you understand what's happening. Pay particular attention to the difference between an element selector (eg. 'rect') and a class selector (with a dot – eg. '.button'). So, in this case, '.button > rect' (matches any <rect> that is an immediate child of an element with the 'button' class), and '.button > .top-left' (matches any element with the 'top-left' class that is an immediate child of an element with the 'button' class). Spotting the difference between a class, ID, and element can be tricky. The syntax for CSS is excessively terse, and less than obvious if you're not used to it, but it's the language we're stuck with so we'll have to make the most of it.

If everything is working okay at this point, it's time to add another set of rules that will apply when the <g> has both the 'button' and the 'clicked' classes set. In this case, you just have to concatenate the class selectors – but make sure not to add any spaces between them, as that signifies an ancestor-descendent relationship. Yeah, the syntax of CSS really is that terse.

Here's an example of the additional rule for the <rect>, complete with the new color from the table earlier in the article. I'll leave it as an exercise for the reader to create the remaining three CSS rules.

```
.button.clicked > rect {
  fill: #800000;
}
```

You can test your new CSS rules by manually adding an extra

'clicked' class to the group (so that it reads class="button clicked"), saving the file, and reloading the page. Remove the extra class and save again before proceeding.

Open the file in Inkscape and select the group that represents the button. In the Object Properties dialog, expand the 'interactivity' section and add the following one-liner to the 'onclick' field (this should be familiar from last month's column):

```
this.classList.toggle('clicked');
```

Save your file, load it into your browser and, if everything has gone smoothly, you should find that clicking on your button toggles it between the two states.

One advantage of wrapping everything in a group, and applying the code to that outer layer, is that clicks on any part of the button are passed through to the enclosing element. This avoids our previous problem whereby clicks on the text didn't toggle the button. But we still have an issue with the text being selectable. We can address this with the 'pointer-events' CSS rule, which lets us tell the browser that all mouse activity over the text – including clicks and selection – should be ignored. Modify your first set of rules so that the last one looks like this:

```
.button > text {
  fill: #a6a6ff;
  pointer-events: none;
}
```

That deals with the selectability problem, but we can go a step further in making our button seem clickable. By adding a 'cursor' property to the group itself, we can make the mouse pointer change when it moves over the button. Add this rule to the CSS:

```
.button {
  cursor: pointer;
}
```

Save, reload, and move your mouse pointer over the button to see the effect.

Previously, I said I would show you how to make clicks on one element have an effect on a completely different one, but I've run out of space in this article, so that will have to wait until next time. Until then, however, you now know how to use CSS classes to style more complex collections of objects, allowing for the creation of much fancier buttons. You also know how to respond to clicks in such a collection simply by putting the code onto a group that wraps all the content.

Why not spend the next month designing ever more impressive buttons? There are plenty of tutorials online for different styles if you're stuck for inspiration. You don't just have to make them toggle when clicked: a common effect is to apply a class in the 'onmouseover' field, and remove it in the 'onmouseout'. You could even go the whole hog and create a button that has both a mouseover effect and a click effect. And next time, I promise, I will show you how to hook your new buttons up to other elements on your page.

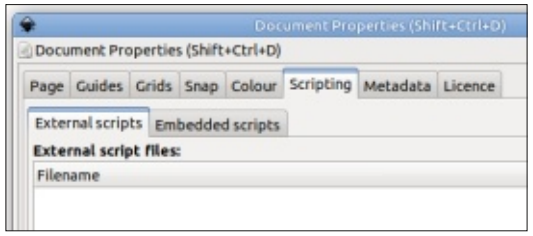
Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

Inkscape - Part 86

If you followed last month's tutorial, you should now have a styled button that toggles between two classes when you click on it. This month, we're going to extend the code behind that button to control other objects on the screen, so if you didn't play along with the previous article, now is the time to go back and give it a try.

We've previously looked at two ways to put JavaScript into an Inkscape file: directly in the fields within the "Interactivity" section of the Object Properties dialog; and via the "Embedded scripts" tab in the "Scripting" section of the Document Properties dialog. You may recall that the latter location also includes a second tab, labelled "External scripts". This month we'll take a look at that tab, and discuss the pros and cons of this approach.

First, load the SVG file for the button that you created last time into Inkscape. Then open the File > Document Properties dialog, and select the "Scripting" tab. Within that section make sure the "External scripts" tab is active.



As with the "Embedded scripts" tab, the UI here is sparse and slightly misleading. At the top is an excessively short area that lists any external script files linked to from your document – it should be empty at the moment. Below that is an unlabelled line, with Plus (+) and Minus (-) buttons after it, the latter being disabled at this point. As the name of the tab suggests, this UI is used to link external scripts – i.e. JavaScript code stored in a separate file – to your SVG file. You have probably also guessed that the Plus button adds a file to the list above, whilst the minus button removes it. What isn't so clear is that the Plus button actually works in three different ways, depending on the content of the unlabelled field, and whether or not your JS file exists yet.

Method 1: If you just click on the Plus button, you'll be presented with a file selector dialog. You can then navigate to the directory of your choice and enter a new filename into the file selector. When you accept the content of the file selector (e.g. by clicking the "Open" button), the full path and name of your file will appear in the list at the top of the dialog. NOTE: This does not actually create the file on-disk, so you'll need to do that manually, opening up the possibility of typos.

Method 2: This is the same as the previous method, except this time you use the file selector to choose a file that already exists. Once again, the full path to the file is put into the list at the top. This has the advantage over the previous method that you can't make a typo with the filename.

Method 3: Type something into the unlabelled box, then click the Plus button. Whatever you typed will be added to the list of files at the top of the dialog. Of course, it only really makes sense to type the name (and possibly path) of a JavaScript file, but this field does no error checking so typos and bad paths are all too easy to introduce.

You might think that the third method is to be avoided – the possibility of making a typo is too high. But, in practice, this is the only method of the three that I recommend using! In just about every case, you will want to keep the JavaScript file close to your SVG file: if not in the same directory, then usually just one level away in a "scripts" or "resources" folder. The third method lets you supply just the filename, or a relative path and filename (eg. 'scripts/button.js'), which will still be valid if you move your SVG and JS files to another machine, or a different location on your hard drive.

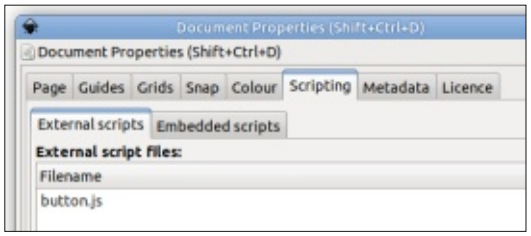
The first two methods, on the other hand, produce absolute paths which will break as soon as the files are moved. And you can't edit the paths within this dialog – you have to modify them via the XML editor. So not only is there a

chance of introducing typos as you edit them, but you have to be comfortable with using the XML editor to do so. Better, I think, to use method 3 and just be careful with what you type.

To keep things simple, let's create a new JavaScript file in the same directory as the main SVG file. Using a text editor, create a new file containing the following line:

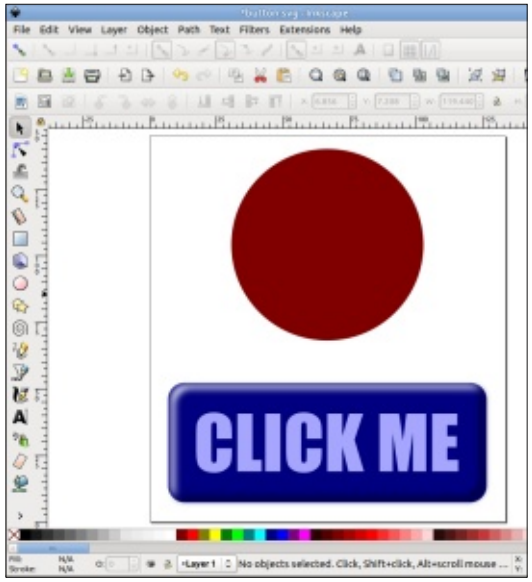
```
alert("Loaded");
```

Save the file to the same directory as your SVG image, with a ".js" extension. In my case the Inkscape file is called "button.svg", so I'll give the JavaScript file the name of "button.js" for clarity, though it's not a requirement for them to share the same name. Back in the Document Properties dialog, I can now type the filename (with no path) into the unlabelled text field, then click the Plus button to add it to the list at the top, so the dialog looks like this:



If everything has been done correctly, loading the SVG file into a browser should result in an alert being shown containing the word "Loaded". That, at least, tells us that the link from SVG to JS files is working. It's better to do a lightweight test like this first, before fleshing out your JavaScript code, to make sure the basics are in place.

We want our button to control another object when it's clicked – which will be a lot easier if we actually have another object to control! In Inkscape, alter the file to add a simple filled circle, making sure to give it a sensible ID via the Object Properties dialog (I called mine "redCircle").



Right-click on the button, and bring up the Object Properties dialog. Within the "Interactivity" section at the bottom, remove any existing code then, in the "onclick" field, add the following JavaScript function call:

```
buttonPressed();
```

This function doesn't actually exist yet: we need to add it to our JS file. Using a text editor, remove the existing alert() and replace it with this:

```
function buttonPressed() {
  alert("Button pressed");
}
```

Save both files, then reload the SVG file in a web browser, confirming that both elements are visible, and that a click on the button shows the new alert() dialog.

Now that we've got a function that runs when the button is clicked, we want to populate it with some code to change the fill color of the circle. Previously, we've changed the fill color of the object being clicked, either by explicitly setting "this.style.fill", or by modifying the classes of the clicked object using "this.classList.toggle()" and similar functions. Altering the style of a different object is essentially the same, except that we no longer use "this" to identify the target for our changes. Instead we need to get a reference to the target object in a different way.

Depending on exactly what you are trying to do, there are various approaches that could be used. But the simplest, at least conceptually, are a pair of methods on the "document" object (which exists implicitly on all XML and HTML documents): querySelector() and querySelectorAll(). The difference between them is that the former returns a single XML node, whereas the latter returns a collection of nodes. A collection is similar in some respects to an array in JS, but doesn't have all the standard array methods, so needs to be treated a little differently. For this example, however, we want to change the style of only a single element, so document.querySelector() will do the job.

Both methods take a single parameter: a string containing a CSS selector. If the selector

matches more than one element then querySelector() just returns the first one. If it matches no elements, the method returns "null". In JavaScript terms, "null" is what's referred to as a "falsey" value – that is, one that evaluates to "false" when you use it in an "if" statement. That makes it pretty simple to write defensive code that won't fall over if your CSS selector doesn't match anything. Let's take a look at an example, by replacing the alert() in your buttonPressed() function (top right).

Save the JS file, then reload the SVG in the browser. With the developer tools open (press F12, if necessary), and the console tab selected, click on the button in your file. You should see the <circle> element logged out – or the string "No circle found!" otherwise. If you saw the latter, it suggests that the CSS query doesn't suit the structure of your file. Here are a few things you can look at to help troubleshoot this:

- Did you remember to put the # in front of the ID in the CSS?
- Does the ID match the one you put on the circle in Inkscape?
- Try changing the selector to match an element (eg. "circle") instead of an ID.
- Open your SVG file in a text editor and search for your ID. Is it definitely present in an ID attribute on the element you expected?

Now that we've got a reference to the circle, it's a pretty simple task to set the style explicitly, or modify its class list. Here's the buttonPressed() function (below) rewritten to directly set the fill color on the circle. Note that I've removed the "else", as we just want the code to fail silently with no side-effects if the CSS selector fails to match anything.

Despite what I said earlier, the querySelector() method doesn't actually return the SVG node, as such, but rather a JavaScript object that references the element in the browser's internal document structure. Usually you can ignore this subtle distinction, but it does mean that we have a JS object in hand, which offers us a few advantages when writing our code. The object has a variety of properties and methods attached to it but we can also attach our own. This can be a useful way to keep track of data that needs to persist outside our function.

```
function buttonPressed() {
  const circle = document.querySelector("#redCircle");
  if (circle) {
    circle.style.fill = "#008000";
  }
}
```

```
function buttonPressed() {
  const circle = document.querySelector("#redCircle");
  if (circle) {
    console.log(circle);
  } else {
    console.log("No circle found!");
  }
}
```

Consider trying to toggle the color in response to the button presses. You could use a CSS class, and call the circle.classList.toggle() method to alternately add and remove it. This is similar to the approach we took last time, except we're referencing our object variable rather than "this". But what if you want to set the fill color on the circle directly, rather than via a class? You could read the value of circle.style.fill back, and test to see what it's currently set to. But there are various ways to define colors in CSS, so you might not get back the format you expect.

A better approach is to create a property on the object that you can refer to each time the function is called. You can test its current value, then set it to something else before your function finishes. You don't need to do anything fancy to create a property like this – the browser will create it as soon as you try to use it – so the code ends up looking like that shown next page, bottom left.

The first time you click the button the "isOn" property doesn't exist. That means the test in the "if" statement fails, and the code in the "else" runs – setting the color to green, and creating the "isOn" property, with a value of "true". The next time you click the button the "if" succeeds, the fill is set to red and the property is set to

"false". Thereafter the color and property will continue to toggle each time you click the button.

One important thing to note as we've been editing this code is that you haven't had to go anywhere near Inkscape since the initial setup. This is one big advantage of using external, linked scripts, as there's less chance of accidentally making changes to your SVG file. You also get all the benefits of using a proper text editor: with any half-decent editor you should get syntax highlighting and other aids, which can help to indicate problems in your code. Plus, being able to work in a proper sized window, rather than the single line of the Object Properties dialog or the letterbox of Inkscape's Embedded Scripts tab, is a huge advantage.

When deploying an SVG file with a linked script, you need to make sure that the script is still accessible to the SVG file once it's on to your web server – typically by ensuring you use a relative path as outlined at the top of this article. You have to make sure you remember to keep your JS file in sync with any changes to your SVG file or its location. But this additional housekeeping is usually more than worth it. Generally, unless you really are writing only a single line of code, or perhaps a single, short function, linking is the way to go.

Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

```
function buttonPressed() {
  const circle = document.querySelector("#redCircle");
  if (circle) {
    if (circle.isOn === true) {
      circle.style.fill = "#800000";
      circle.isOn = false;
    } else {
      circle.style.fill = "#008000";
      circle.isOn = true;
    }
  }
}
```

THE FULL CIRCLE APP FOR UBUNTU TOUCH - UPDATED!

Brian Douglass has updated his FCM app for Ubports Touch devices that will allow you to view current issues, and back issues, and to download and view them on your Ubuntu Touch phone/tablet.

INSTALL

Either search for 'full circle' in the Open Store and click install, or view the URL below on your device and click install to be taken to the store page:
https://uappexplorer.com/app/fullcircle.bhdouglass

HUGE thanks to Brian for this.