# Cyclic Rules

This is a very simple document that will describe the cyclic rules that may occur with element queries. The examples will use the syntax of the ELQ framework as described in the Problem Formulation document. Also, the CSS presented in this document is written in preprocessor syntax to make the examples more clear, also as described in the problem formulation document.

## Direct cyclic rules

The most straight forward occurances of cyclic rules are when a programmer has specified conflicting width constraints and width updates for an element.

### Example 1

```
<div id="container" elq elq-breakpoints elq-breakpoints-width="300 500">
    <div id="child"></div>
</div>
```

```
#container {
    width: 250px;
}

#container.elq-width-under-300 {
    width: 550px;
}

#container.elq-width-above-500 {
    width: 250px;
}
```

The following will happen for the module:

1. If the initial width of the container is between 300px and 500px, the selector `#container` will be the only matching selector and the width of the container element will be set to 250px. Then next step will be 2. If the container element is under 300px wide, next step will also be 2. If the container element is above 500px wide, next step will be 3.
2. The width of the element is under 300px, and the selector `#container.elq-width-under-300` will now match and since it is more specific than the `#container` selector, the new width of the element will be 550px. Next step will always be 3.
3. The width of the element is above 500px, and the selector `#container.elq-width-`

`above-500` will now match and since it is more specific than the `#container` selector, the new width of the element will be 250px. Next step will always be 2.

Clearly, the browser will be stuck in an infinite layout cycle pending back and forth between case 2 and case 3 (250px and 550px). Depending on CSS implementation, this may or may not cause a rendering engine crash. What could happen is that the rendering engine will execute one layout pass and then evaluate the next set of matched selectors and so on, which will lead to a functioning site but since a relayout is enforced every update, the performance impact will be huge.

## Indirect cyclic rules

Cyclic rules can also occur in a more indirect way. For instance, if the container element will match the width of the child, but the child changes width depending on the parent width, a cycle might occur.

### Example 2

```
<div id="container" elq elq-breakpoints elq-breakpoints-width="300 500">
    <div id="child"></div>
</div>
```

```
#container {
    display: inline-block; //<- Will match the width of the child element.
}

#child {
    width: 250px;
}

#container.elq-width-under-300 #child {
    width: 550px;
}

#container.elq-width-above-500 #child {
    width: 250px;
}
```

What we have here is the same situation as in example 1. What makes this situation trickier is that it is less obvious for numerous reasons:

- The rules of the child element and the rules of the container element might be separated into different parts of the stylesheet (or even different stylesheets). The problem cannot be found without considering both of the rulesets.
- The child element might be another module, and by adding one line to the parent

module (the container) a cycle has appeared.
- The programmer has to be well aware of how the `display` property affects the element and what implications the `inline-block` has to the element.

The examples given so far have been very simple and easily detectable. However, cycles can occur in a much more complex way. As of writing this, it is still unclear to me what would be a good example for a very complex cycle occurance. However, my personal theory is that as factors that creates the cycles increases the harder it gets to detect them (as a programmer). To clarify, let's consider the factors of example 1 and 2:

- **Example 1:** This cycle is very easy to spot (as reading the CSS out loud almost gives away the problem in English). This example only operates on the width of the container element, which is 1 factor to the cycle.
- **Example 2:** This cycle is easy to spot, but not as easy as example 1. The reason is that we cannot know that there is a cycle unless we also consider the container element properties (here the `` `display `` property). So this cycle depends on the width of the child element, the width of the container element and also the display property of the container element. So this example has 3 factors to the cycle.

Let's consider another example which has even more factors:

## Example 3

```
<div id="container" elq elq-breakpoints elq-breakpoints-width="300 500">
    <div id="child">
        When in doubt, mumble.
    </div>
</div>
```

```
#container {
    display: inline-block; //<- Will match the width of the child element.
}

#child {
    font-size: 1.5em;
}

#container.elq-width-under-300 #child {
    font-size: 2em;
}

#container.elq-width-above-500 #child {
    font-size: 1em;
}
```

In this example, it's virtually impossible to tell if the rules are cyclic. So what happens here is that the container element will get the size of the child element, which width depend on the text inside it. The width of the text depends on the font size, which is initially set to 1.5em. The `em` unit is relative to the inherited font size of the element. So basically the width of the container element is dependent on the inherited font size. When the container element is below 300px, the font size is increased to 2em and if the element is above 500px the font size is reduced to 1em. So if 2em results in a font size big enough to make the child element bigger than 500 pixels, and if 1em results in a font size small enough to make the child element smaller than 300 pixels we have a cycle.

Let's list the factors that creates the cycle:

- The width of the container element
- The display type of the container element.
- The font size unit of the child element.
- **The inherited font size of the child element.**
- **The text of the element.**

This adds up to 5 factors. However, the bolded factors are of a far worse type than the ones we have discussed until now. They are impossible to determine at parse time. In this example the text is static but it could be added dynamically. Also, the inherited font size of the child depends on the closest ancestor with a font size property defined. However, since ancestors also can have their font sizes defined in relative units, the dependency tree can go to the root of the document. Further, if no ancestor defines an absolute value for the font size it is up to the browser to default to a size, which is impossible to reason about at parse time. This means that there is no way of telling if the cycle will appear or not without actually running the code. Also, the cycle may appear in different browsers and settings, which makes the cycle even harder to detect.

Far worse is that the previous examples have been of the character of which the reader thinks "why would anyone ever want to do that?", which gives the false hope that even though it is possible to create cycles - developers would never encouter it since "real" code would not be close to the stupid examples given here. With example 3, this is no longer true. Increasing the font size when the layout area is smaller and decreasing the font size when the layout area is bigger is something that is frequently done. Think about how websites adapts to small and big screens. Hint: the font size is usually bigger on the mobile phone than it is on the desktop.

# Conclusion

In short, the programmer needs to be more alert when developing with element queries in order to avoid cycles. One could argue that the programmer still would need to be alert and cunning while styling sites to avoid errors even without element queries. While this is

technically true, the difference is that the damange that programmers may cause due to styling errors are very different. With element queries, it is possible to create infinite cycles that cripples the rendering engine. Without element queries, the worst that could happen due to styling errors is that the website simply doesn't look good. So element queries really put more responsibility to the developer and demands a deeper understanding of CSS. In order to aid and guide the programmer, it is important to be able to catch and handle cyclic rules when implementing element queries.

It is possible to detect the simplest forms of cycles but a robust cycle detection mechanism must be executed at runtime in order to catch the more complicated cycles. Another advantage of detecting the cycles at runtime is that the detection is CSS and implementation agnostic, since the detection system does not need to know all the mechanics of styling - it just needs to know if the browser is stuck in a cycle or not.