**Projects**

**The working behind the scene**

First of all, let’s talk about the setup: first, we have a slide window (in this case, it was created with the div tag). Inside the window, we have another div as the window’s child whose class is called *project-slide-band*. Now this band takes up 1000% of the width of its parent – the slide window, then inside the band we can have less 10 slide divs with each taking 10% of the width of its parent – the slide band, therefore each slide is as wide as its grandparent – the slide window. All the slides will be arranged as an array with no gaps among them. Now with this setup, the following mechanism can take place:

On the bird-eye view, how everything works is that we first have a certain amount of radio input buttons laid out on top of all the slide divs, that is, radio inputs are siblings with the slide divs (this is very important because later we will use the general sibling selector – the tilde mark). This block of slide divs starts with one which contains a class called *first-slide*. Now when one of those radio inputs is checked, the first slide will be dragged towards the left of the slide window so one particular slide after it will be shown through the window; this can happen because in the related CSS file, we have something like this: *#project1-radio2:checked ~ .first-slide {margin-left: 10%;}*. This specific line of code works to let the second slide be shown to us through the slide window because when the second input radio is checked, it sets the left margin of the first slide to 10%. Since all the slide divs are arrayed with no gaps among them inside the slide band, and the slide band’s first 10% slot is overlapped with the slide window, so when the left margin of the slide div is set to 10%, the second slide will be displayed. and since each slide div’s width is the same as the window’s width and all the slides are arrayed width no gaps among them, when the first slide is dragged to the left of the window in the distance equal to the window size, the second slide will be displayed through the window.

Now since we know how the slides can be shown to the user one by one, to make it automatic, we just have to use a timer generated by the setInterval function. Another player in this automation is a number variable named *counter*. This counter variable works with the timer and controls which specific slide’s related radio input is being currently checked. The interval of this timer is set to be 3000, so for every 3 seconds, the counter will increase by one and so the radio inputs will be checked one by one from left to right, therefore user will see a slide change every 3 seconds. Now to manually change to one specific slide, in the JS file we just have to change the counter to the number related to the radio input that the user clicks, and then after that the counter will go up as usual until it hits the end and then it will start all over again.

**Challenge**- adding new projects

It was very difficult to add new projects to the section, so I came up with a new method that made it all much easier. I used this method in the turtle’s website (in fact, it was the main approach in which that project was built). This method was: I used static class to build a component called Slide, and then in a JS file I called the class’s activate method. For this activate method, I passed all the arguments it needed to render the whole project to the section. Then I would just call this method again every time when I needed to add a new project. How I came up with this approach was: I first saw there were many repetitive parts among those blocks of code; and I also noticed there were parts where they differed from one another, so I could just create a static class to contain the whole code and then for those parts in which different projects vary from one another, I just made them parameters for the activate method, so the result was just I could call the activate method for adding a new project, which is so much simpler than having to repeatedly copy and paste an entire block of code and then make some changes.

After I created the static class Slides, the task of adding new projects has become very easy.

One thing to keep in mind about implementing this method: first, just copy and paste the entire original code block, and then try to break the html string down into several different parts managed by methods under the static class; then think of those variables in terms of arguments you pass into a function call or look at them as parameters for their related methods. Don’t try to plan everything ahead in your head, you’re not a genius; always start by doing, then go one step at a time.

Another thought: The main reason that building or working on a piece of software is hard is that it always involves multiple moving parts interconnecting with one another, behind the scene it’s always layers over layers to make just one thing work. So, whenever you see a problem or something that doesn’t work out as the way you expected, don’t try to dig into all these layers, you always try to trace down the problem by starting with the layer that’s closest to where the problem first takes place because we always just add new stuff upon the existing establishment. Take this project for instance, when I was trying to add the static class Slide, I never had to dig into how the animation actually worked, I just identified the key points that made it work, and built the component upon those; later then when I had that problem where I changed the number of image links in the *imgArr* array, and the auto buttons area got really chaotic, I just looked into the method that was responsible for generating those auto buttons, I never had to dig into how the automatic animation worked with the buttons.