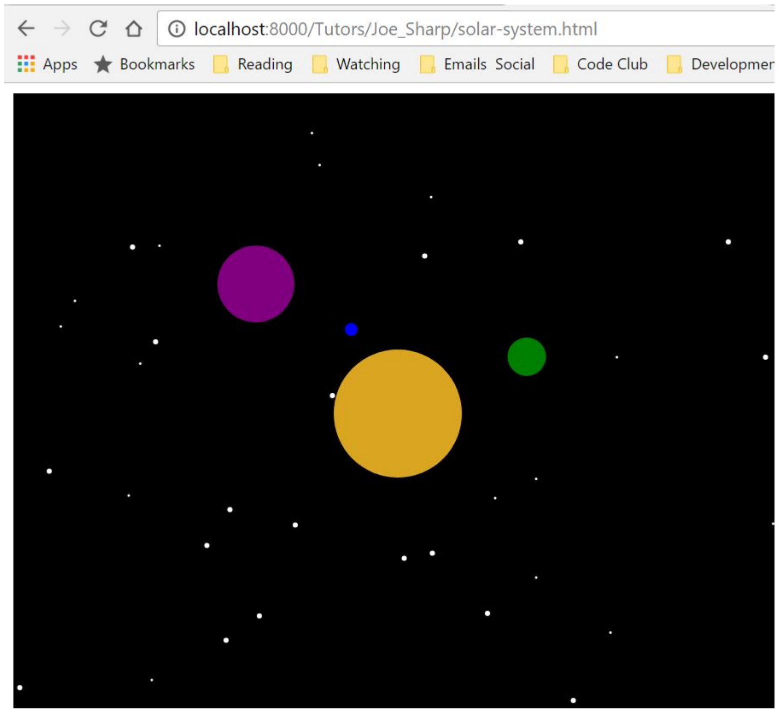
JavaScript Solar System

This project will use HTML5 Scalable Vector Graphics to draw and animate a solar system. Scalable Vector Graphics (SVG) is one of two key methods for generating custom graphics (the other being the canvas system).

This project is split into several stages. The first will use SVG to draw a static image of a Sun, Planets and Stars on a black background. The second stage will use JavaScript to build the scene and animate it.

Eventually it should look something like this.



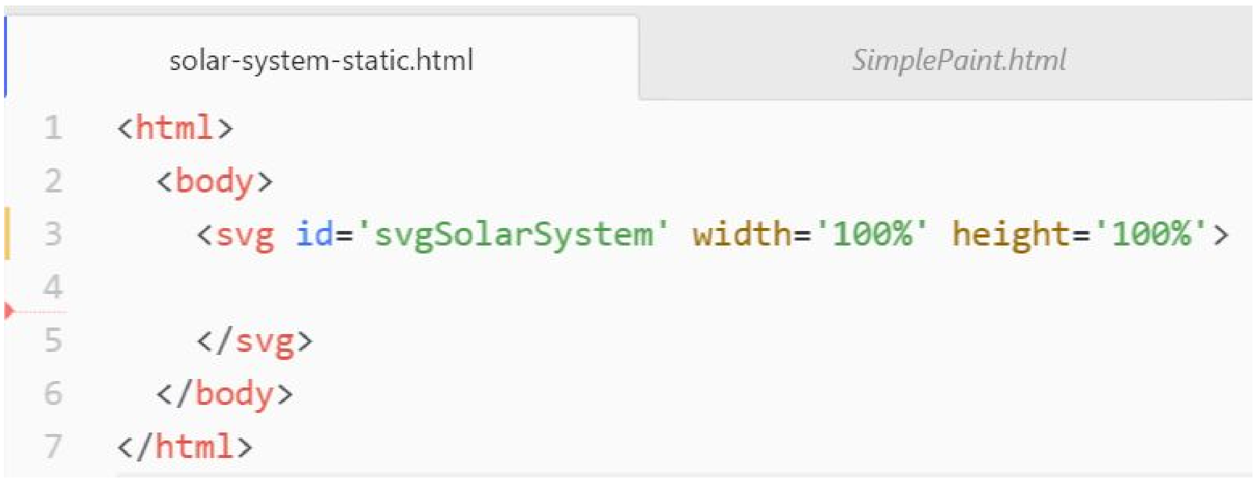
The large yellow blob is the 'sun' and the other coloured blobs are 'planets' orbitting the sun. The white dots are stars.

Phase 1 - Static Scene

Build the SVG

The static version of the page will be a very simple HTML5 document with a single SVG element. This SVG element will contains elements for all the shapes we require in our scene.

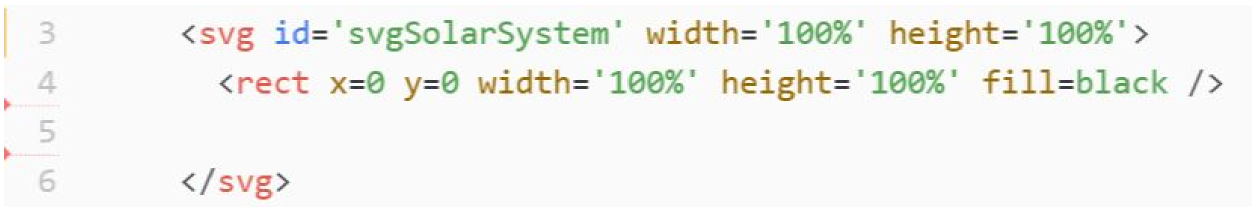
Create a new file called something like **solar-system-static.html** and type the following.



solar-system-static.html | SimplePaint.html

```html
1  <html>
2    <body>
3      <svg id='svgSolarSystem' width='100%' height='100%'>
4
5      </svg>
6    </body>
7  </html>
```

If you open the page in the browser, it will currently show nothing, so let's now add a black rectangle to act as the 'space'. Add the <rect> element shown on line 4 inside our existing <svg> element.



```html
3      <svg id='svgSolarSystem' width='100%' height='100%'>
4        <rect x=0 y=0 width='100%' height='100%' fill=black />
5
6      </svg>
```

Check it Works!

Reload the page in your browser and it should show a big black rectangle that consumes the page.

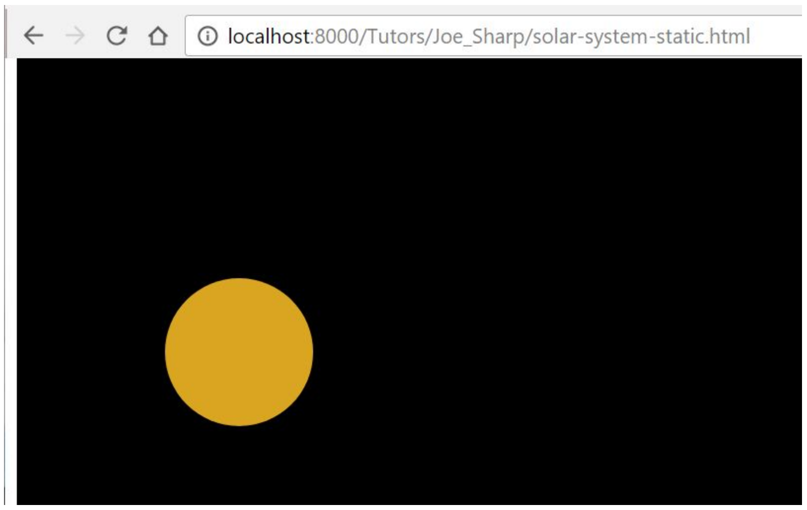
Draw the Sun

Our Sun will be made up of a single large circle on our page. Add the tag shown in line 8 below. **Notice that I have added comments above the shape elements** in order to communicate what they represent in our diagram.

```
 3      <svg id='svgSolarSystem' width='100%' height='100%'>
 4        <!-- Space Background -->
 5        <rect x=0 y=0 width='100%' height='100%' fill=black />
 6
 7        <!-- The Sun -->
 8        <circle cx=150 cy=200 r=50 fill=goldenrod />
 9
10      </svg>
```

Check it Works!

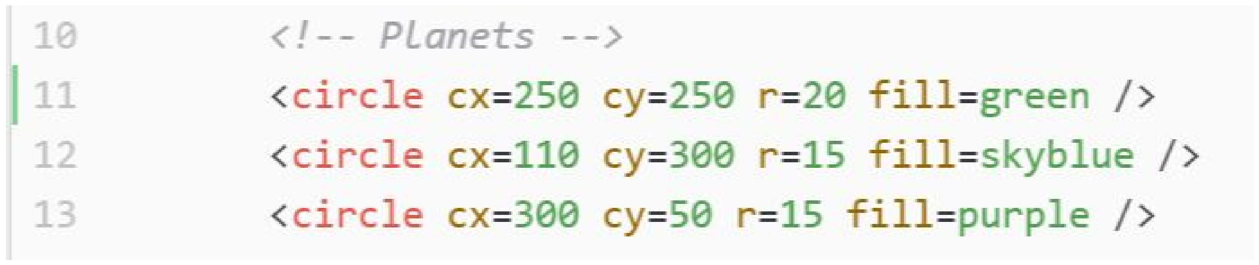
Reload the page and it should look like this. Note that the coordinates system starts with x=0 at the far left, and y=0 at the top of the page. If you increase the value of **y** in the <circle> tag you should find the Sun moves down the page.


localhost:8000/Tutors/Joe_Sharp/solar-system-static.html

Adding further shapes will be as simple as this. The documentation for SVG elements can be found here https://www.w3schools.com/graphics/svg_intro.asp

Draw Some Planets

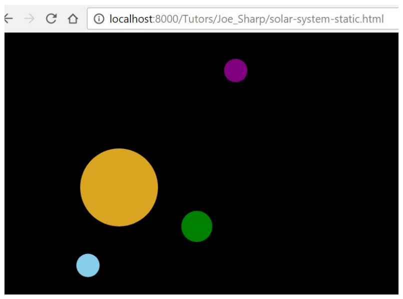
For this we will use the same basic technique as drawing the Sun. I used the following elements for 3 planets but you can tweak the values as you wish.



```
        <!-- Planets -->
        <circle cx=250 cy=250 r=20 fill=green />
        <circle cx=110 cy=300 r=15 fill=skyblue />
        <circle cx=300 cy=50 r=15 fill=purple />
```

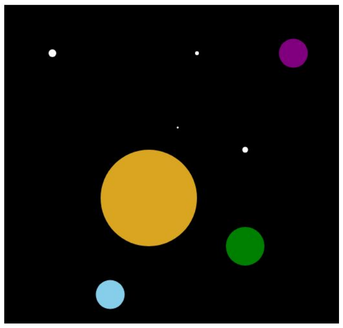
Check it Works!

This gave a result like this.



Draw Some Stars

Last up for the static scene is to add some stars. For this add more circles but set the radius to anywhere between 1 and 4. The colour should be white. I just added 4 stars to mine and it looked like this.



When we draw the scene dynamically, we will generate hundreds of stars using loops.

Check it Works!

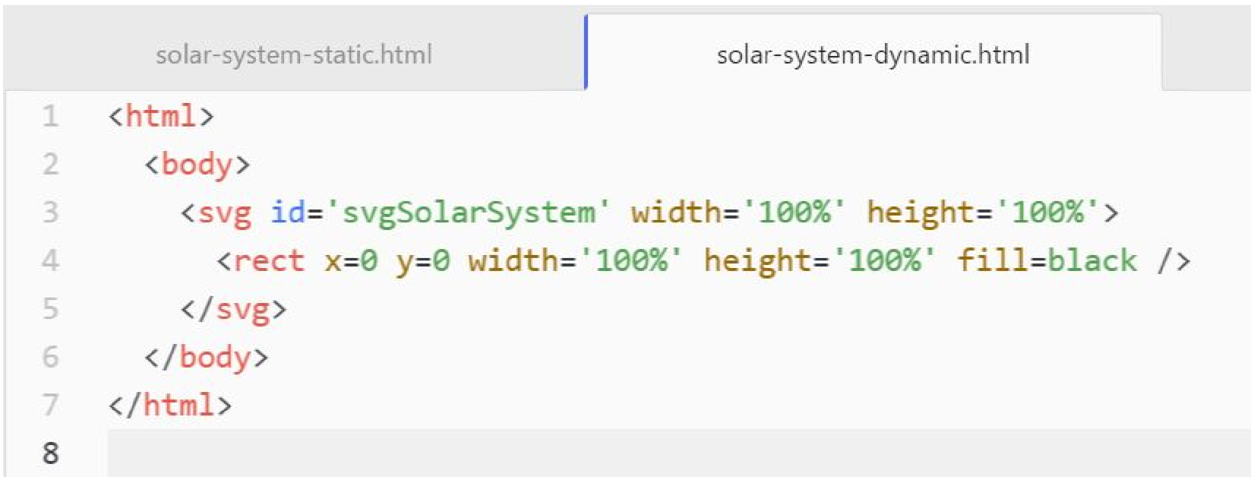
Add your own stars and reload in the browser to make sure they are visible as expected.

Phase 2 - JavaScript Built Scene

In the first phase we handcrafted the contents of the <svg> tag. In this phase we will build the same scene again, but use JavaScript to add the elements.

Create a New File

Create a new file called **solar-system-dynamic.html** and type the following HTML. This looks much like the first few steps of Phase 1. You may want to copy your Phase 1 file and delete all the circles.



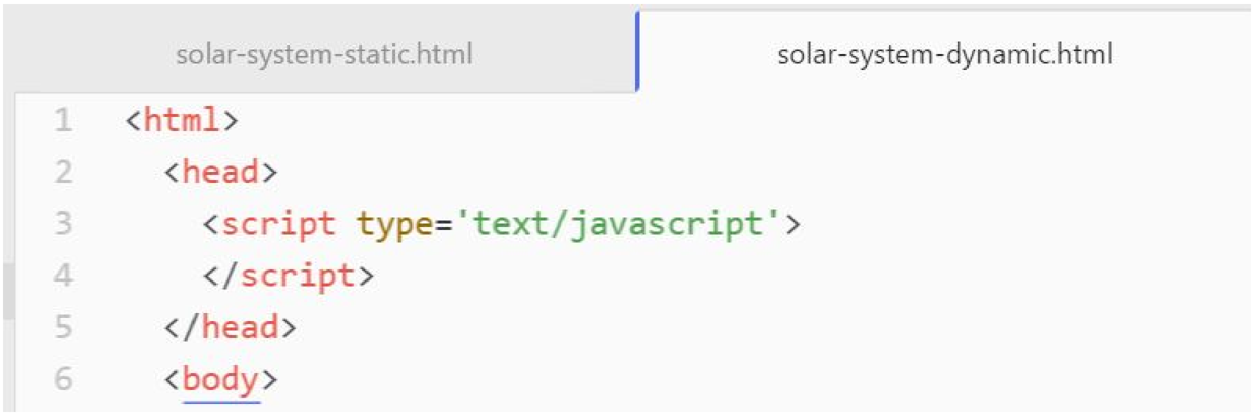
```
<html>
  <body>
    <svg id='svgSolarSystem' width='100%' height='100%'>
      <rect x=0 y=0 width='100%' height='100%' fill=black />
    </svg>
  </body>
</html>
```

Check it Works!

Load this new page in your browser and it should just show a black background.

Add JavaScript

This HTML will contain JavaScript, which we will embed directly in the <head> of the document. So add the <head> and a <script> tag as shown below on lines 2 to 5.



```
1  <html>
2    <head>
3      <script type='text/javascript'>
4      </script>
5    </head>
6    <body>
```

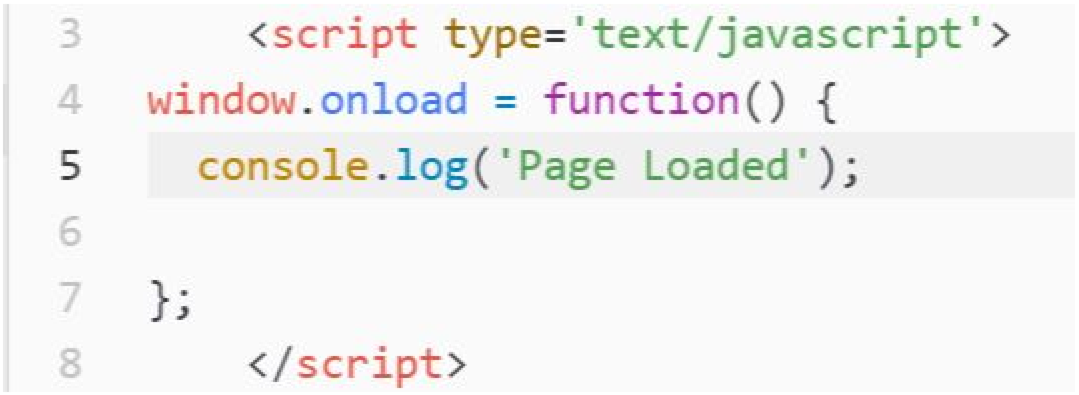
If you open this HTML file in your web browser, it should have the black background (we will leave that as a handcrafted element).

Check it Works!

Open in a browser and perhaps leave open the developer tools. Check that there are no errors in the console. From this point on you will need the developer tools open to make sure your code is working correctly.

Handle Page Load

We need our code to activate after the page has loaded, so that all the elements have been built. For this to work we assign a function to the **window.onload** event and handle everything in there. Add the following code to your script tag. Notice that I have reset the indentation for the JavaScript to prevent endless nesting.



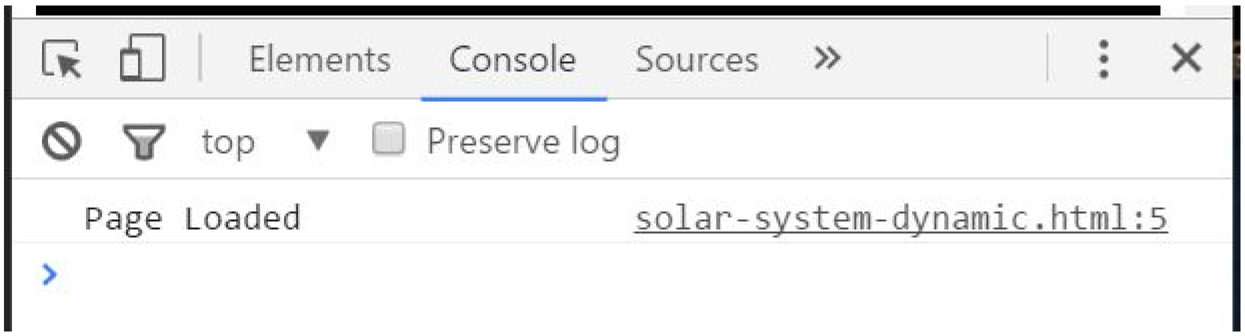
```
        <script type='text/javascript'>
window.onload = function() {
  console.log('Page Loaded');

};
        </script>
```

The console is a great means of debugging your program, so I have put the ‘page loaded’ message in to demonstrate that our function is being called correctly.

Check it Works!

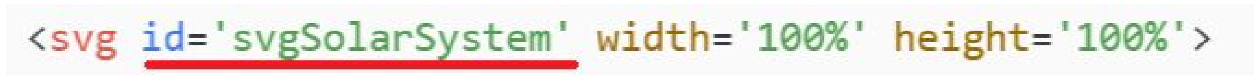
With this code in place, open the Developer Tools in your web browser (CTRL+SHIFT+I in Chrome on Windows) and reload the page. You should see the following message in the console.



```
Page Loaded                    solar-system-dynamic.html:5
>
```

Retrieve Reference to the SVG Element

The first thing this code should do is get a reference to the <svg> element on our page. This is done using the **id** of the tag that we set earlier. Check that your <svg> element has the **id** as shown here.



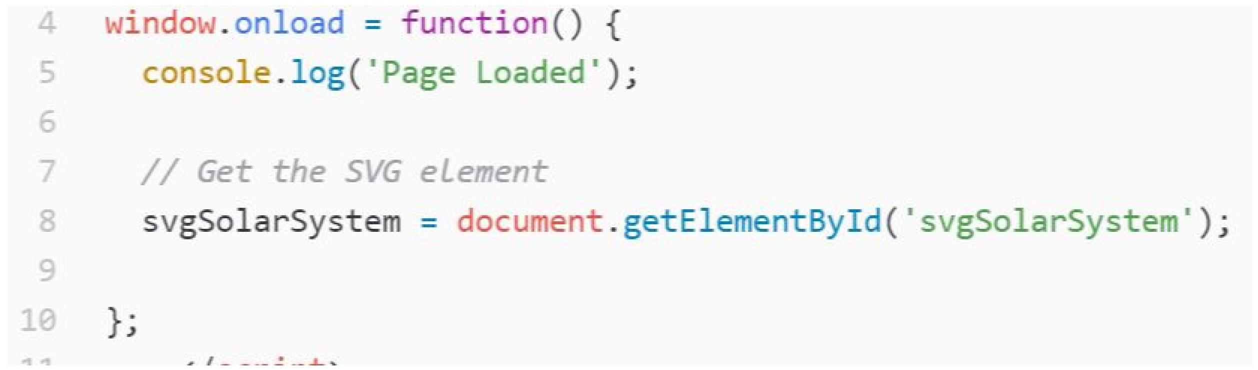
```
<svg id='svgSolarSystem' width='100%' height='100%'>
```

Learning Point!

Giving elements explicit values for **id** allows you to refer to them in your JavaScript. Make sure they are unique with the page!

Handle OnLoad

Then add the following code (line 8) to your onload function.



```
4   window.onload = function() {
5     console.log('Page Loaded');
6
7     // Get the SVG element
8     svgSolarSystem = document.getElementById('svgSolarSystem');
9
10  };
```

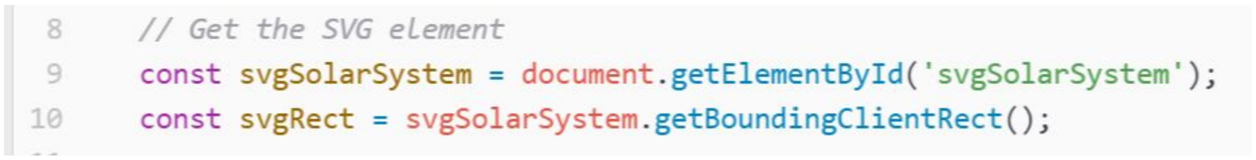
Learning Point!

The window.onload is an example of an **event handler**. These are functions that are called in response to certain events on the page. An event might be a user clicking on something, or even just the mouse moving into its bounds. In this case the page loading is exposed as an event.

Using event handlers is a key part of making sure that multiple parts of code are called in the correct sequence.

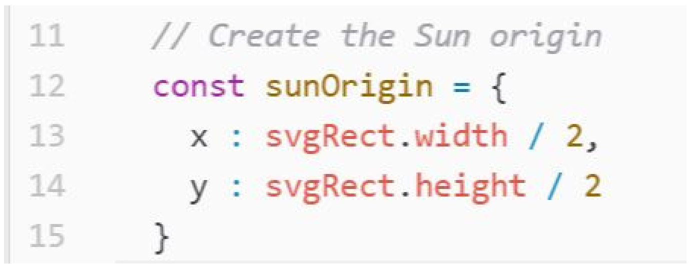
Calculate Sun Origin

We will be using the bounds of the SVG element to place our Sun in the middle of the page. To get the bounding rectangle add line 9.



```
8     // Get the SVG element
9     const svgSolarSystem = document.getElementById('svgSolarSystem');
10    const svgRect = svgSolarSystem.getBoundingClientRect();
```

We will now create an ‘origin’ data object to store the x and y coordinates of our sun.



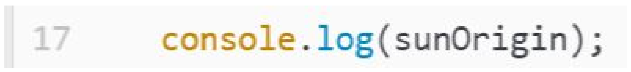
```
11    // Create the Sun origin
12    const sunOrigin = {
13      x : svgRect.width / 2,
14      y : svgRect.height / 2
15    }
```

Learning Point!

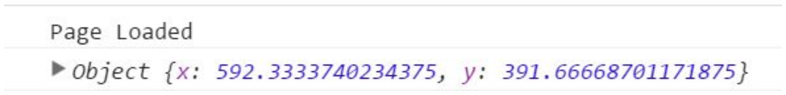
We have created an object in JavaScript by using the curly bracket notation. This object has the properties **x** and **y** which we can then refer to later. This is useful for keeping related bits of data together in a structured way.

Check it Works!

You may wish to print this out using the console to demonstrate that all is working well.



```
17    console.log(sunOrigin);
```



```
Page Loaded
▶ Object {x: 592.3333740234375, y: 391.66668701171875}
```

Create the Sun

In this section we will create the <circle> element for our Sun. To do this we have to use the SVG element namespace. So create the following constant to hold this value.

```
// Required to create new SVG elements
const SVG_NS = 'http://www.w3.org/2000/svg';
```

Now create a new circle element with the following.

```
// Create the Sun element
const sunElement = document.createElementNS(SVG_NS, 'circle');
```

Prepare the Element

As it stands, this element **not appear on your page yet**. We have simply prepared a blank <circle> for us to build and then append when we are ready. I am going to use the **sunOrigin** we calculated earlier and a couple of other values to build up the Sun.

```
// Create the Sun element
const sunElement = document.createElementNS(SVG_NS, 'circle');
sunElement.setAttribute('cx', sunOrigin.x);
sunElement.setAttribute('cy', sunOrigin.y);
sunElement.setAttribute('r', 50);
sunElement.setAttribute('fill', 'goldenrod');
```

Learning Point!

For a <circle> element to appear, it should have the cx, cy and r (radius) attributes populated as a minimum. This step is preparing our valid circle before we add it to the page. After it has been added to the page we can continue to amend the properties and the browser will update the display accordingly.

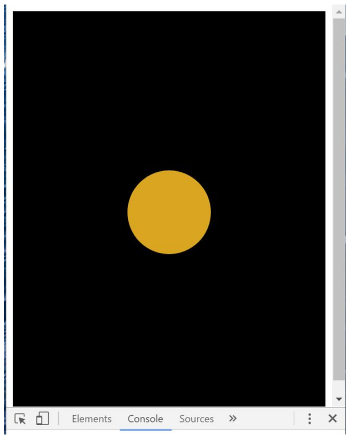
Add the Element

We will add the Sun to our image using the **appendChild** function on the svg element reference we took earlier.

```
svgSolarSystem.appendChild(sunElement);
```

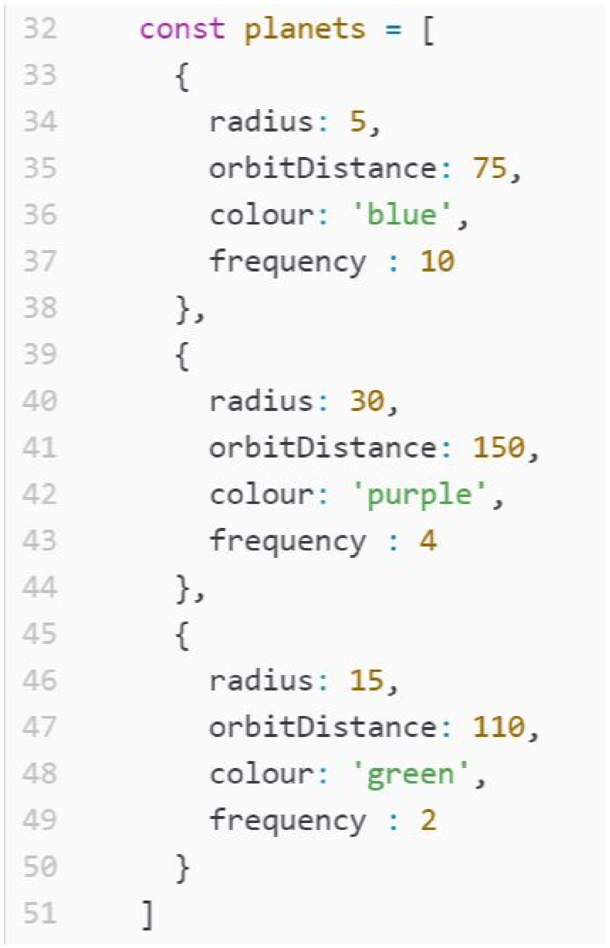
Check it Works!

The image should now contain our Sun, exactly in the middle.



Create the Planets

Our planets will be defined in a JavaScript array of objects. Each object will have the properties required to build our planets. The following code will create this array.



```
32    const planets = [
33      {
34        radius: 5,
35        orbitDistance: 75,
36        colour: 'blue',
37        frequency : 10
38      },
39      {
40        radius: 30,
41        orbitDistance: 150,
42        colour: 'purple',
43        frequency : 4
44      },
45      {
46        radius: 15,
47        orbitDistance: 110,
48        colour: 'green',
49        frequency : 2
50      }
51    ]
```

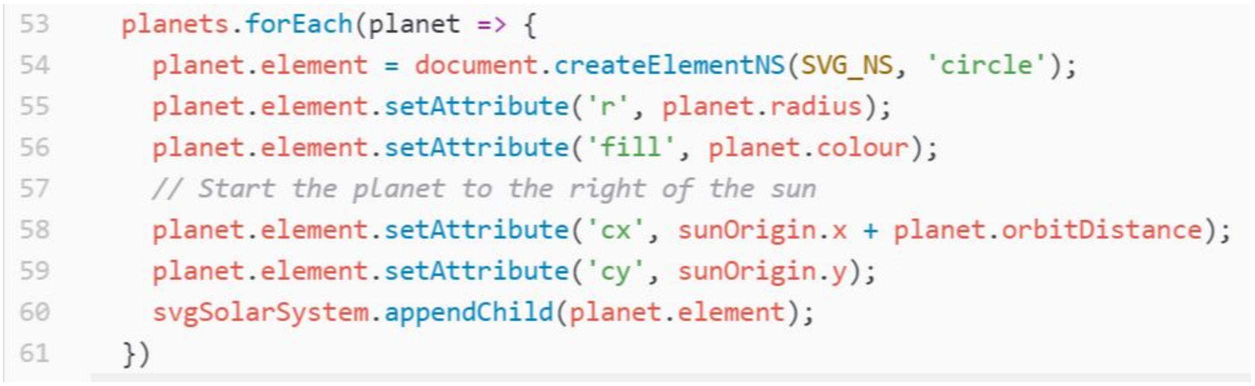
Note that line 29 and 48 contain the square brackets [] that wrap an array. Each object is wrapped in curly brackets {} and separated by commas.

Check it Works!

It would be worth reloading the page and monitoring the console just to check you have typed everything correctly. Feel free to use **console.log** to print the value of **planets**.

Generate an Element For Each Planet

Now we will use JavaScript array **forEach** function to iterate through these objects and create <circle> elements for them. In this code I am creating a property called **element** on the existing **planet** objects we created earlier. We are allowed to do this by JavaScript, it neatly keeps everything together. Later on we will need to refer to this new **element** property when we want to animate the planets.



```
  planets.forEach(planet => {
    planet.element = document.createElementNS(SVG_NS, 'circle');
    planet.element.setAttribute('r', planet.radius);
    planet.element.setAttribute('fill', planet.colour);
    // Start the planet to the right of the sun
    planet.element.setAttribute('cx', sunOrigin.x + planet.orbitDistance);
    planet.element.setAttribute('cy', sunOrigin.y);
    svgSolarSystem.appendChild(planet.element);
  })
```

Learning Point!

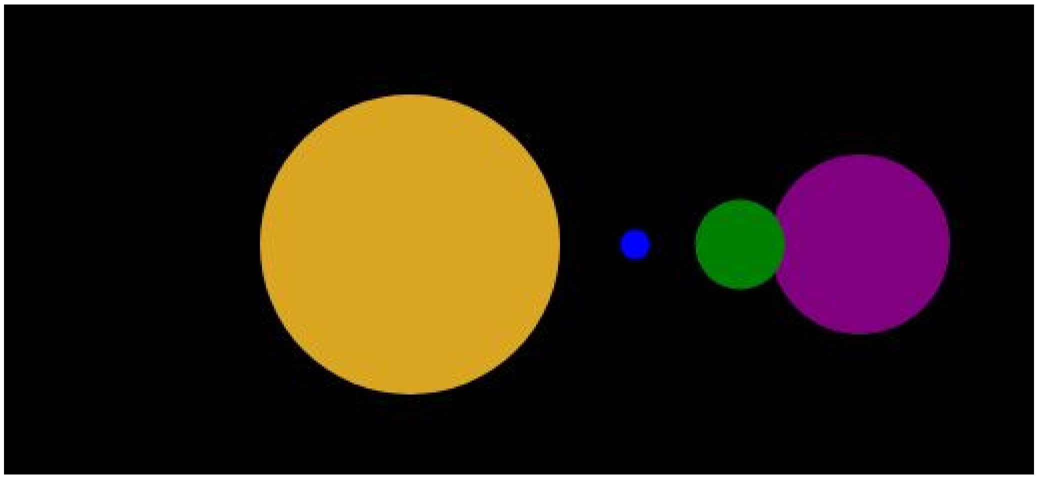
In this forEach call I have defined a function using JavaScripts **lambda** notation. This is the notation that looks like this

```
() => { //code }
```

In this case we have defined a function that accepts a single argument. This function is called for each object in the **planets** array.

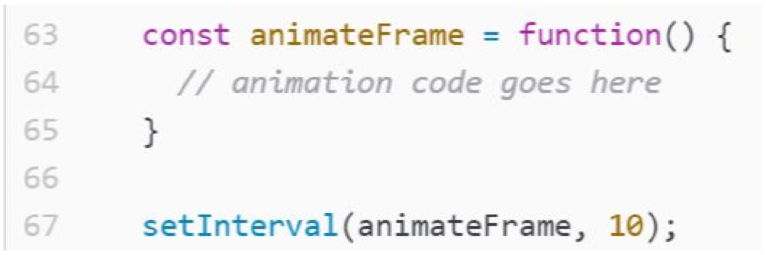
Check it Works!

The result of this code should be something like this. We will animate them in the next step.



Get the Planets in Orbit

We will be using the JavaScript **setInterval** function to animate our creation. First create a function called **animateFrame** which will be called every ten milliseconds. Then add a call to **setInterval** as shown in lines 63 to 67 below.

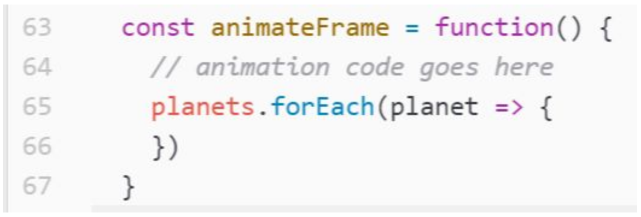


```
63    const animateFrame = function() {
64      // animation code goes here
65    }
66
67    setInterval(animateFrame, 10);
```

Learning Point!

The **animateFrame** function is scoped to this block because we have declared it using the **const** keyword. We could equally have used the **var** keyword. If we missed off **const** or **var**, then the function would be added to the global namespace. This is a risky thing to do as the program gets more complicated so I tend to declare the function as a local variable.

We will be using the **forEach** function on our planet array. The forEach call syntax is the same as it was in a previous step.



```
63    const animateFrame = function() {
64      // animation code goes here
65      planets.forEach(planet => {
66      })
67    }
```

Check it Works!

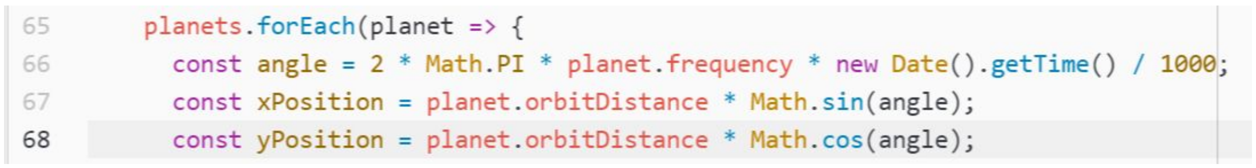
If we reload the page now, nothing will appear different. The JavaScript engine will be calling our function every 10 milliseconds, so now we must put some code in there to animate our planets. Just check the console for any syntax errors.

Calculate Simple Harmonic Motion

Now that we have a function in a loop for handling each planet we must calculate the new x and y positions. For this we will be using the maths of Simple Harmonic Motion, which rely on trigonometry. The formulas are as follows.

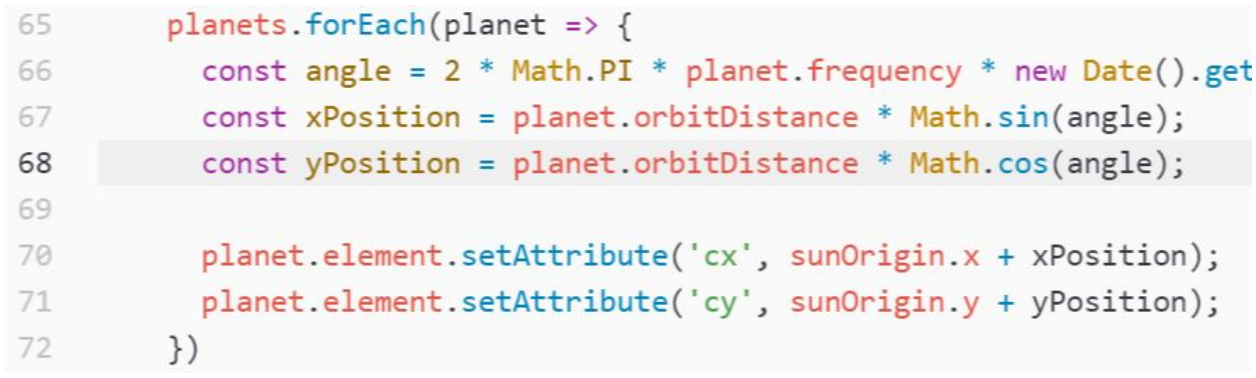
```
x_position = orbit_distance * sin(2 * Pi * frequency * time)
y_position = orbit_distance * cos(2 * Pi * frequency * time)
```

The calculation inside the sin/cos functions is the angle at a given time. We can just do this calculation once as shown.



```
65      planets.forEach(planet => {
66        const angle = 2 * Math.PI * planet.frequency * new Date().getTime() / 1000;
67        const xPosition = planet.orbitDistance * Math.sin(angle);
68        const yPosition = planet.orbitDistance * Math.cos(angle);
```

Now we need to use these x/y values to adjust the centre of our planets <circle> element. This is done as shown on lines 70 and 71.



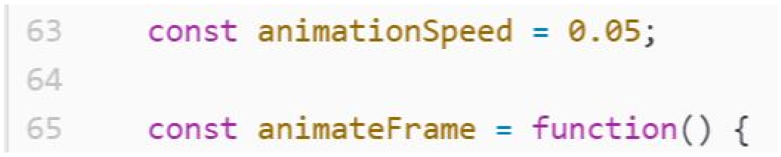
```
65      planets.forEach(planet => {
66        const angle = 2 * Math.PI * planet.frequency * new Date().get
67        const xPosition = planet.orbitDistance * Math.sin(angle);
68        const yPosition = planet.orbitDistance * Math.cos(angle);
69
70        planet.element.setAttribute('cx', sunOrigin.x + xPosition);
71        planet.element.setAttribute('cy', sunOrigin.y + yPosition);
72      })
```

Check it Works!

Load this in your browser and you should find your planets are now rotating around the sun. They are probably going a bit fast though! We will fix that next.

Adjust the Speed

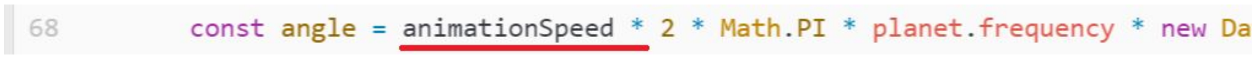
To slow our animation down, we will add a special speed adjustment factor to the animate function and apply it to calculation of **angle**. I have called my adjustment factor **animationSpeed** and define it **above the animateFrame function** as shown.



```
const animationSpeed = 0.05;

const animateFrame = function() {
```

Add it as a factor to your angle calculation as shown.



```
    const angle = animationSpeed * 2 * Math.PI * planet.frequency * new Da
```

Check it Works!

You should now have fully animated orbits. Here are some screenshots of mine.