📒

Week 2: L3 Readings

The Window Object

Every JS environment has a **global object**.

- **global object** → any variables created in the global scope are properties of this object, and any functions are methods of it.

💡 In a browser environment, the global object is the **window object**

The Browser Object Model

→ collection of properties and methods that contain information about the browser and computer screen.

→ we can find out which browser is being used to view a page (unreliable), dimensions of the screen it is viewed on, and which pages have been visited before the current page.

→ properties and methods are made available through the **window object**.

BOM Only Makes Sense in a Browser Environment

→ This means that other environments (such as Node.js) probably won't have a **window object**, though they still have a global object (Node.js has an object called **global)**

```
// from within the global scope
const global = this;
```

Going Global

Global Variables can be accessed in all parts of the program. They are actual properties of a global object. In a browser environment, the global object is the **window object**. This means that any global variable created is actually a property of the window object.

```
x = 6;  // global variable created
<< 6

window.x // same variable can be accessed as a property of the window object
<< 6

// both variables are exactly the same
window.x === x;
<< true
```

In general, you should refer to global variables without using the `window` object

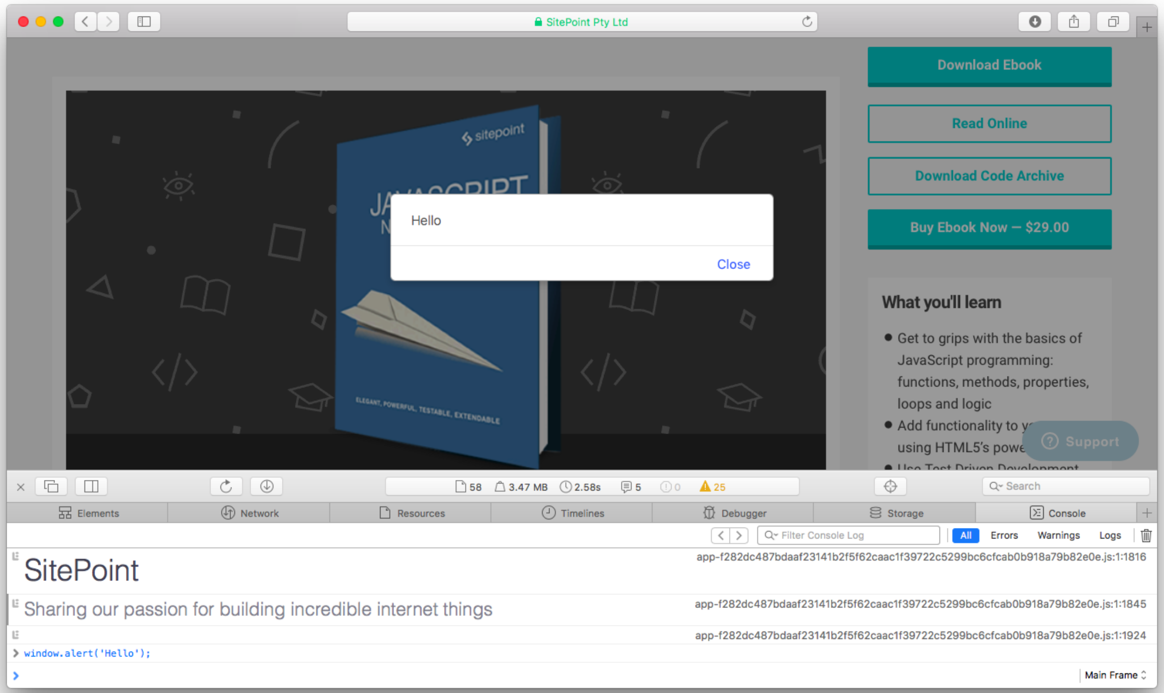
An exception is if you need to check whether a global variable has been defined.

```
if (x) {
// do something
}
```

Dialogs

The `window.alert()` method will pause the execution of the program and display a message in a dialog box. The message is provided as an argument to the method, and `undefined` is always returned:

```
window.alert('Hello');
<< undefined
```



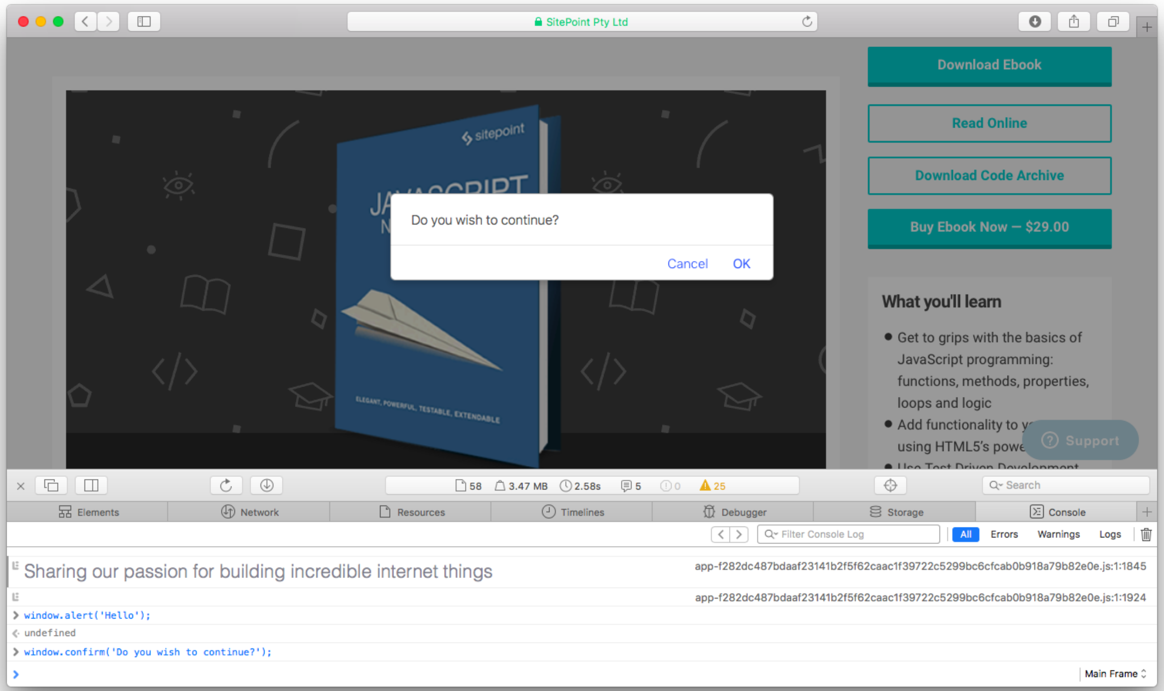
What you'll learn

- Get to grips with the basics of JavaScript programming: functions, methods, properties, loops and logic
- Add functionality to y using HTML5's powe

```
SitePoint                                          app-f282dc487bdaaf23141b2f5f62caac1f39722c5299bc6cfcab0b918a79b82e0e.js:1:1816
Sharing our passion for building incredible internet things      app-f282dc487bdaaf23141b2f5f62caac1f39722c5299bc6cfcab0b918a79b82e0e.js:1:1845
                                                   app-f282dc487bdaaf23141b2f5f62caac1f39722c5299bc6cfcab0b918a79b82e0e.js:1:1924
> window.alert('Hello');
>
```

The `window.confirm()` method will **stop the execution of the program and display a confirmation dialog that shows the message provided as an argument**, and giving the options of OK or Cancel. Returns `true` if the user clicks OK, and `false` if the user clicks Cancel:

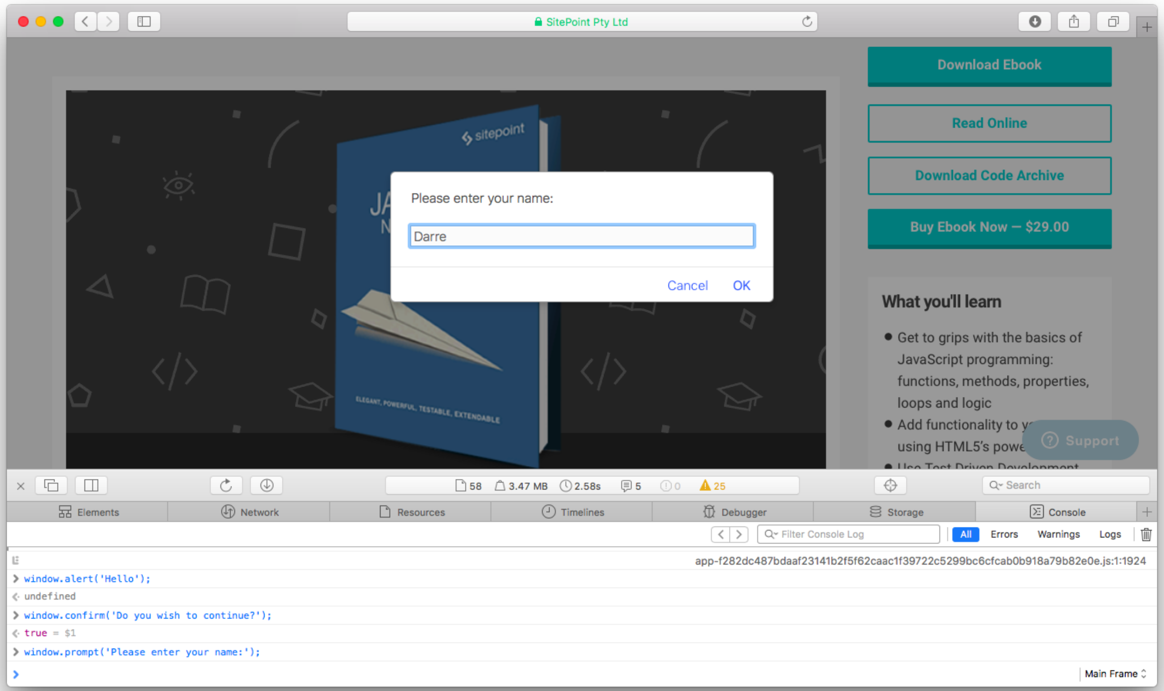
```
window.confirm('Do you wish to continue?');
<< undefined
```



```
Sharing our passion for building incredible internet things    app-f282dc487bdaaf23141b2f5f62caac1f39722c5299bc6cfcab0b918a79b82e0e.js:1:1845
                                                               app-f282dc487bdaaf23141b2f5f62caac1f39722c5299bc6cfcab0b918a79b82e0e.js:1:1924
> window.alert('Hello');
< undefined
> window.confirm('Do you wish to continue?');
```

The `window.prompt()` method will stop the execution of the program. It displays a dialog that shows a message provided as an argument, as well as an input field that allows the user to enter text. This text is then returned as a string when the user clicks OK. If the user clicks Cancel, `null` is returned:

```
window.prompt('Please enter your name:');
```



```
app-f282dc487bdaaf23141b2f5f62caac1f39722c5299bc6cfcab0b918a79b82e0e.js:1:1924
> window.alert('Hello');
< undefined
> window.confirm('Do you wish to continue?');
< true = $1
> window.prompt('Please enter your name:');
>
```

💡 **Use with care!** These methods will stop the execution of a program in its tracks meaning that everything will stop processing at the point the method is called until the user clicks OK or Cancel. This can cause problems if the program needs to process something else at the same time or the program is waiting for a callback function.

Browser Information

The `window` object has a number of properties and methods that provide information about the user's browser.

Which Browser?

The `window` object has a `navigator` property that returns a reference to the `Navigator` object. The `Navigator` object contains information about the browser being used. Its `userAgent` property will return information about the browser and operating system being used.

```
window.navigator.userAgent
<< "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_3) AppleWebKit/602.4.8 (KHTML, like Gecko) Version/10.0.3 Safari/602.4.8"
```

Location, Location, Location

The `window.location` property is an object that contains information about the URL of the current page. It contains a number of properties that provide information about different fragments of the URL.

The `href` property returns the full URL as a string:

```
window.location.href
<< "https://www.sitepoint.com/premium/books/javascript-novice-to-ninja"
```

The `protocol` property returns a string describing the protocol used (such as `http`, `https`, `pop2`, `ftp` etc.)

```
window.location.protocol
<< "https:"
```

The `host` property returns a string describing the domain of the current URL *and* the port number (this is often omitted if the default port 80 is used):

```
window.location.host
<< "www.sitepoint.com"
```

The `hostname` property returns a string describing the domain of the current URL:

```
window.location.hostname
<< "www.sitepoint.com"
```

The `port` property returns a string describing the port number, although it will return an empty string if the port is not explicitly stated in the URL:

```
window.location.port
<< ""
```

The `pathname` property returns a string of the path that follows the domain:

```
window.location.pathname
<< "/premium/books/javascript-novice-to-ninja"
```

The `search` property returns a string that starts with a "?" followed by the query string parameters. It returns an empty string if there are no query string parameters. This is what I get when I search for "JavaScript" on SitePoint:

```
window.location.search
<< "?q=javascript&limit=24&offset=0&page=1&
content_types[]=All&slugs[]=all&states[]=available&order="
```

The `hash` property returns a string that starts with a "#" followed by the fragment identifier. It returns an empty string if there is no fragment identifier:

```
window.location.hash
<< ""
```

The `origin` property returns a string that shows the protocol and domain where the current page originated from. This property is read-only, so cannot be changed:

```
window.location.origin
<< "https://www.sitepoint.com"
```

The `reload()` method can be used to force a reload of the current page. If it's given a parameter of `true`, it will force the browser to reload the page from the server, instead of using a cached page.

The `assign()` method can be used to load another resource from a URL provided as a parameter, for example:

```
window.location.assign('https://www.sitepoint.com/')
```

The `replace()` method is almost the same as the `assign()` method, except the current page will not be stored in the session history, so the user will be unable to navigate back to it using the back button.

The `toString()` method returns a string containing the whole URL:

```
window.location.toString();
<< "https://www.sitepoint.com/javascript/"
```

The Browser History

The `window.history` property can be used to access information about any previously visited pages in the current browser session.

The `window.history.length` property shows how many pages have been visited before arriving at the current page.

The `window.history.go()` method can be used to go to a specific page, where 0 is the current page:

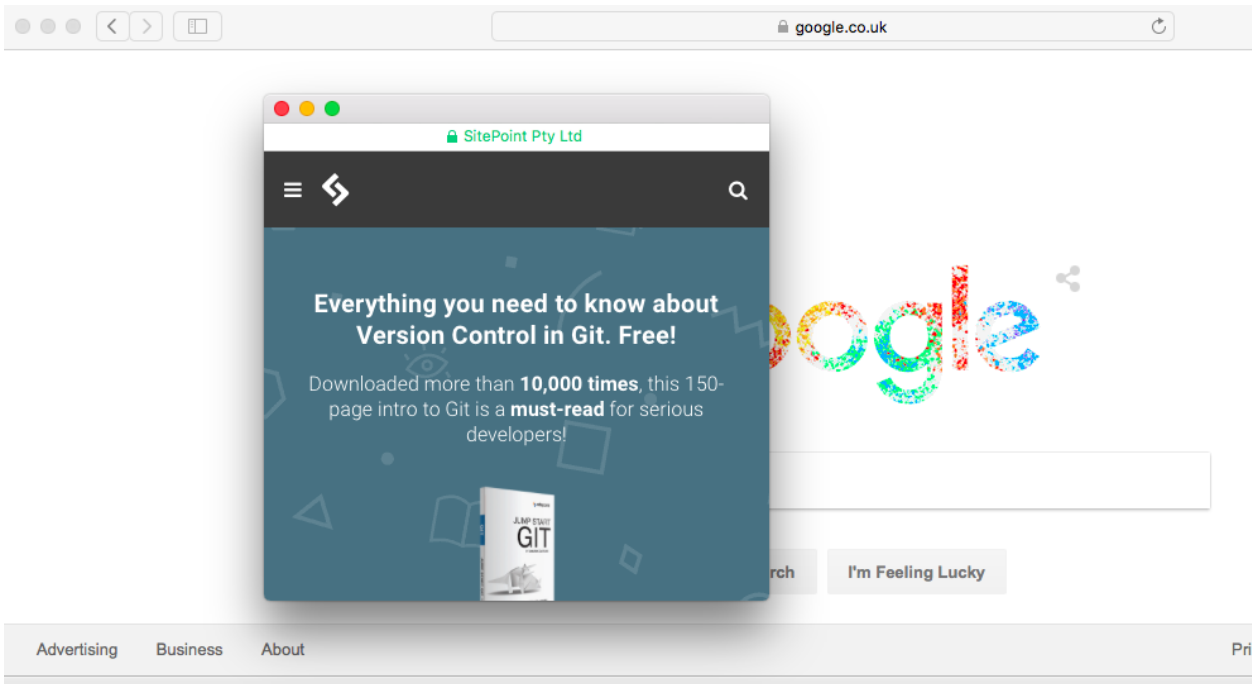
```
window.history.go(1); // goes forward 1 page
window.history.go(0); // reloads the current page
window.history.go(-1); // goes back 1 page
```

There are also the `window.history.forward()` and `window.history.back()` methods that can be used to navigate forwards and backwards by one page respectively, just like using the browser's forward and back buttons.

Controlling Windows

The `window.open()` method takes the URL of the page to be opened as its first parameter, the window title as its second parameter, and a list of attributes as the third parameter.

```
const popup = window.open('https://sitepoint.com','
SitePoint','width=400,height=400,resizable=yes');
```


google.co.uk
SitePoint Pty Ltd
Everything you need to know about Version Control in Git. Free!
Downloaded more than 10,000 times, this 150-page intro to Git is a must-read for serious developers!
I'm Feeling Lucky
Advertising
Business
About

The `close()` method can be used to close a window, assuming you have a reference to it:

```
popup.close();
```

It is also possible to move a window using the `window.moveTo()` method. This takes two parameters that are the X and Y coordinates of the screen that the window is to be moved to:

```
window.moveTo(0,0); // will move the window to the top-left corner of the screen
```

You can resize a window using the `window.resizeTo()` method. This takes two parameters that specify the width and height of the resized window's dimensions:

```
window.resizeTo(600,400);
```

Screen Information

The `window.screen` object contains information about the screen the browser is displayed on. You can find out the height and width of the screen in pixels using the `height` and `width` properties respectively:

```
window.screen.height
<< 1024

window.screen.width
<< 1280
```

The `availHeight` and `availWidth` can be used to find the height and width of the screen, excluding any operating system menus:

```
window.screen.availWidth
<< 1280

window.screen.availHeight
<< 995
```

The `colorDepth` property can be used to find the color bit depth of the user's monitor, although there are few use cases for doing this other than collecting user statistics:

```
window.screen.colorDepth;
<< 24
```

💡 The **Screen object** is more useful for mobile devices. It allows you to do things like turn off the device's screen, detect a change in its orientation or lock it in a specific orientation.

The Document Object

Each `window` object contains a `document` object. This object has properties and methods that deal with the page that has been loaded into the window.

`document.write()`

The `write()` method simply writes a string of text to the page. If a page has already loaded, it will completely replace the current document:

```
document.write('Hello, world!');
```

This would replace the whole document with the string `Hello, world!`

It is possible to include HTML in the string and this will become part of the DOM tree.

```
document.write('<h1>Hello, world!</h1>');
```

The `document.write()` method can also be used within a document inside `<script>` tags to inject a string into the markup. This will not overwrite the rest of the HTML on the page.

```
<h1>
    <script>document.write("Hello, world!")</script>
</h1>
```

💡 The use of `document.write()` is heavily frowned upon as it can only be realistically used by mixing JavaScript within an HTML document.

Cookies

→ small files that are saved locally on a user's computer.

→ A restriction of cookies is that they can only be read by a web page from the same domain that set them.

→ Cookies are also limited to storing up to 4KB of data, although 20 cookies are allowed per domain, which can add up to quite a lot of data.

→ Cookies can be used for personalizing a user's browsing experience, storing user preferences, keeping track of user choices (such as a shopping cart), authentication and tracking users.

→ Cookies take the form of a text file that contain a list of name/value pairs separated by semicolons.

```
"name=Superman; hero=true; city=Metropolis"
```

Creating Cookies

To create a cookie, you assign it to JavaScript's "cookie jar", using the `document.cookie` property:

```
document.cookie = 'name=Superman';
<< "name=Superman"
```

The `document.cookie` property acts like a special type of string. Assigning another cookie to it won't overwrite the entire property, it will just append it to the end of the string. So we can add more cookies by assigning them to `document.cookie`:

```
document.cookie = 'hero=true';
<< "hero=true"

document.cookie = 'city=Metropolis';
<< "city=Metropolis"
```

Changing Cookie Values

A cookie's value can be changed by reassigning it to `document.cookie` using the same name but a different value.

```
document.cookie = 'name=Batman'
<< "name=Batman"
document.cookie = 'city=Gotham'
<< "city=Gotham"
```

Reading Cookies

To see the current contents of the cookie jar, simply enter `document.cookie`:

```
document.cookie:
<< "name=Batman; hero=true; city=Gotham"
```

Cookie Expiry Dates

Cookies are session cookies by default. This means they are deleted once a browser session is finished (when the user closes the browser tab or window).

Cookies can be made persistent — that is, lasting beyond the browser session — by adding `"; expires=date"` to the end of the cookie when it's set, where `date` is a date value in the UTC String format `Day, DD-Mon-YYYY HH:MM:SS GMT`. The following example sets a cookie to expire in one day's time:

```
const expiryDate = new Date();
const tomorrow = expiryDate.getTime() + 1000 * 60 * 60 * 24;
expiryDate.setTime(tomorrow);

document.cookie = `name=Batman; expires=${ expiryDate.toUTCString()}`;
```

An alternative is to set the `max-age` value. This takes a value in seconds

```
document.cookie = 'name=Batman; max-age=86400' // 86400 secs = 1 day
```

The Path and Domain of Cookies

By default, cookies can only be read by pages inside the same directory and domain as the file was set. This is for security reasons so that access to the cookie is limited.

The path can be changed so that any page in the root directory can read the cookie. It's done by adding the string `; path=/` to the end of the cookie when it is set:

```
document.cookie = 'name=Batman; path=/'
```

It's also possible to set the domain by adding `"; domain=domainName"` to the end of the cookie:

```
document.cookie = 'name=Batman; domain=sitepoint.com';
```

Secure Cookies

Adding the string `; secure` to the end of a cookie will ensure it's only transmitted over a secure HTTPS network:

```
document.cookie = 'name=Batman; secure';
```

Deleting Cookies

To remove a cookie, you need to set it to expire at a time in the past:

```
document.cookie = 'name=Batman; expires=Thu, 01 Jan 1970 00:00:01 GMT';
```

Timing Functions

`setTimeout()`

The `window.setTimeout()` method accepts a callback to a function as its first parameter and a number of milliseconds as its second parameter.

```
window.setTimeout( () => alert("Time's Up!"), 3000);
<< 4
```

setInterval()

The `window.setInterval()` method works in a similar way to `window.setTimeout()`, except that it will repeatedly invoke the callback function after every given number of milliseconds.

Use a named function:

```
function chant(){ console.log('Beetlejuice'); }
```

Set up the interval and assign it to a variable:

```
const summon = window.setInterval(chant,1000);
<< 6
```

This should show the message “Beetlejuice” in the console every second (1,000 milliseconds).

To stop this, we can use the `window.clearInterval()`
 method and the variable `repeat` as an argument (this is because the `window.setInterval()` method returns its ID, so this will be assigned to the variable `repeat`):

```
window.clearInterval(summon);
```

💡 Be careful when using a method that uses the `this` keyword with either of these timing methods. The binding of `this` is set to the `window` object, rather than the method’s object, so it can get some unexpected results

Animation

The `setTimeOut()` and `setInterval()` methods can be used to animate elements on a web page.

As an example, let's create a web page that shows a colored square, and make it rotate.

animation folder: index.html, styles.css, main.js

```
<!doctype html>
<html lang='en'>
<head>
<meta charset='utf-8'>
<title>Animation Example</title>
<link rel='stylesheet' href='styles.css'>
</head>
<body>
<div id='square'></div>
<script src='main.js'></script>
</body>
</html>
```

```
#square {
    margin: 100px;
    width: 100px;
    height: 100px;
    background: #d16;
}
```

This code receives a reference to our square div, then sets a variable called `angle` to 0. We then use the `setInterval()` method to increase the value of `angle` by `2` (we also use the `%` operator so that it resets to 0 at 360), then set the `transform` CSS3 property to rotate that number of degrees. The second argument is `1000/60`, which equates to a frame speed of 60 frames per second.

```
const squareElement = document.getElementById('square');
let angle = 0;

setInterval( () => {
    angle = (angle + 2) % 360;
    squareElement.style.transform = `rotate(${angle}deg)`
}, 1000/60);
```

`requestAnimationFrame`

This method of the `window` object works in much the same way as the `window.setInterval()` method, although it has a number of improvements to optimize its performance.

```
const squareElement = document.getElementById('square');
let angle = 0;

function rotate() {
    angle = (angle + 2)%360;
    squareElement.style.transform = `rotate(${angle}deg)`
    window.requestAnimationFrame(rotate);
}

const id = requestAnimationFrame(rotate);
```

Start the animation: call the `requestAnimationFrame()` method, giving the `rotate()` function as an argument.

This will return a unique ID that can be employed to stop the animation using the `window.cancelAnimationFrame()` method:

```
cancelAnimationFrame(id);
```

<template>:The Content Template element

→ The `<template>` HTML element is a mechanism for holding HTML that is not to be rendered immediately when a page is loaded but may be instantiated subsequently during runtime using JavaScript.

→ Think of a template as a content fragment that is being stored for subsequent use in the document.

Attributes

The only standard attributes that the `template` element supports are the global attributes.

Also, the `HTMLTemplateElement` has a standard `content` property (without a corresponding content/markup attribute), which is a read-only `DocumentFragment` containing the DOM subtree which the template represents.

directly using the value of the `content` property could lead to unexpected behavior

Avoiding DocumentFragment pitfall

A `DocumentFragment` is not a valid target for various events, as such it is often preferable to clone or refer to the elements within it.