Chapter 30: Intervals and Timeouts

Section 30.1: Recursive setTimeout

To repeat a function indefinitely, `setTimeout` can be called recursively:

```
function repeatingFunc() {
    console.log("It's been 5 seconds. Execute the function again.");
    setTimeout(repeatingFunc, 5000);
}

setTimeout(repeatingFunc, 5000);
```

Unlike `setInterval`, this ensures that the function will execute even if the function's running time is longer than the specified delay. However, it does not guarantee a regular interval between function executions. This behaviour also varies because an exception before the recursive call to `setTimeout` will prevent it from repeating again, while `setInterval` would repeat indefinitely regardless of exceptions.

Section 30.2: Intervals

```
function waitFunc(){
    console.log("This will be logged every 5 seconds");
}

window.setInterval(waitFunc,5000);
```

Section 30.3: Intervals

Standard

You don't need to create the variable, but it's a good practice as you can use that variable with clearInterval to stop the currently running interval.

```
var int = setInterval("doSomething()", 5000 ); /* 5 seconds */
var int = setInterval(doSomething, 5000 ); /* same thing, no quotes, no parens */
```

If you need to pass parameters to the doSomething function, you can pass them as additional parameters beyond the first two to setInterval.

Without overlapping

setInterval, as above, will run every 5 seconds (or whatever you set it to) no matter what. Even if the function doSomething takes long than 5 seconds to run. That can create issues. If you just want to make sure there is that pause in between runnings of doSomething, you can do this:

```
(function(){

   doSomething();

   setTimeout(arguments.callee, 5000);

})()
```

Section 30.4: Removing intervals

`window.setInterval()` returns an `IntervalID`, which can be used to stop that interval from continuing to run. To do this, store the return value of `window.setInterval()` in a variable and call `clearInterval()` with that variable as the only argument:

```
function waitFunc(){
    console.log("This will be logged every 5 seconds");
}

var interval = window.setInterval(waitFunc,5000);

window.setTimeout(function(){
    clearInterval(interval);
},32000);
```

This will log `This will be logged every 5 seconds` every 5 seconds, but will stop it after 32 seconds. So it will log the message 6 times.

Section 30.5: Removing timeouts

`window.setTimout()` returns a `TimeoutID`, which can be used to stop that timeout from running. To do this, store the return value of `window.setTimeout()` in a variable and call `clearTimeout()` with that variable as the only argument:

```
function waitFunc(){
    console.log("This will not be logged after 5 seconds");
}
function stopFunc(){
    clearTimeout(timeout);
}

var timeout = window.setTimeout(waitFunc,5000);
window.setTimeout(stopFunc,3000);
```

This will not log the message because the timer is stopped after 3 seconds.

Section 30.6: setTimeout, order of operations, clearTimeout

setTimeout

- Executes a function, after waiting a specified number of milliseconds.
- used to delay the execution of a function.

Syntax : `setTimeout(function, milliseconds)` or `window.setTimeout(function, milliseconds)`

Example : This example outputs "hello" to the console after 1 second. The second parameter is in milliseconds, so 1000 = 1 sec, 250 = 0.25 sec, etc.

```
setTimeout(function() {
    console.log('hello');
}, 1000);
```

Problems with setTimeout

if you're using the `setTimeout` method in a for loop **:**

```
for (i = 0; i < 3; ++i) {
  setTimeout(function(){
    console.log(i);
  }, 500);
}
```

This will output the value `3` `three` times, which is not correct.

Workaround of this problem :

```
for (i = 0; i < 3; ++i) {
  setTimeout(function(j){
    console.log(i);
  }(i), 1000);
}
```

It will output the value `0`,`1`,`2`. Here, we're passing the `i` into the function as a parameter(`j`).

Order of operations

Additionally though, due to the fact that JavaScript is single threaded and uses a global event loop, `setTimeout` can be used to add an item to the end of the execution queue by calling `setTimeout` with zero delay. For example:

```
setTimeout(function() {
    console.log('world');
}, 0);

console.log('hello');
```

Will actually output:

```
hello
world
```

Also, zero milliseconds here does not mean the function inside the setTimeout will execute immediately. It will take slightly more than that depending upon the items to be executed remaining in the execution queue. This one is just pushed to the end of the queue.

Cancelling a timeout

clearTimeout() : stops the execution of the function specified in `setTimeout()`

Syntax : clearTimeout(timeoutVariable) or window.clearTimeout(timeoutVariable)

Example :

```
var timeout = setTimeout(function() {
    console.log('hello');
}, 1000);

clearTimeout(timeout); // The timeout will no longer be executed
```