Understanding timers: setTimeout and setInterval

Ilya Kantor

Browser provides a built-in scheduler which allows to setup function calls for execution after given period of time.

The syntax is:
var timerId = setTimeout(func|code, delay)

func|code
: Function variable or the string of code to execute.

delay
: The delay in microseconds, 1000 microseconds = 1 second.

The execution will occur after the given delay.

For example, the code below calls alert('hi') after one second:

```
1 function f() {
2   alert('Hi')
3 }
4 setTimeout(f, 1000)
```

If the first argument is a string, the interpreter creates an anonymous function at place from it.

So, this call works the same.

```
1 setTimeout("alert('Hi')", 1000)
```

it causes problems with minificators. It's here mainly for compatibility with older JavaScript code.

Use anonymous functions instead:

```
1 setTimeout(function() { alert('Hi') }, 1000)
```

The returned timerId can be used to cancel the action.

The syntax is: clearTimeout(timerId).

In the example below the timeout is set and then cleared, so nothing happens.

```
1 var timerId = setTimeout(function() { alert(1) }, 1000)
2
3 clearTimeout(timerId)
```

The setInterval(func|code, delay) method has same features as setTimeout.

It schedules the repeating execution after every delay microseconds, which can be stopped by clearInterval call.

The following example will give you an alert every 2 seconds until you press stop. Run it to see in action.

```
1 <input type="button" onclick="clearInterval(timer)" value="stop">
2
3 <script>
4   var i = 1
5   var timer = setInterval(function() { alert(i++) }, 2000)
6 </script>
```

Create the clock red-green-blue clock as given below :

The starting code is here: tutorial/advanced/timing/clock-interval-src/index.html.

Solution

First, the HTML/CSS structure.

```
<div id="clock">
    <span id="hour">hh</span>:<span id="min">mm</span>:<span
id="sec">ss</span>
</div>
```

Every component is split aside for better styling and independent update.

The execution will use setInterval(update, 1000):

```
01 var timerId // current timer if started
```

```
02
03 function clockStart() {
04   if (timerId) return
05
06   timerId = setInterval(update, 1000)
07   update()  // (*)
08 }
09
10 function clockStop() {
11   clearInterval(timerId)
12   timerId = null
13 }
```

Note that update is not only scheduled, but called right now, so the user doesn't have to wait 1000 ms for the start.

And finally, the updating function:

```
01 function update() {
02   var date = new Date()
03
04   var hours = date.getHours()
05   if (hours < 10) hours = '0'+hours
06   document.getElementById('hour').innerHTML = hours
07
08   var minutes = date.getMinutes()
09   if (minutes < 10) minutes = '0'+minutes
10   document.getElementById('min').innerHTML = minutes
11
12   var seconds = date.getSeconds()
13   if (seconds < 10) seconds = '0'+seconds
14   document.getElementById('sec').innerHTML = seconds
15 }
```

setInterval does not promise any real execution delay. So every update creates a brand new Date object and uses it to update the clock.

The full code of the solution is at tutorial/advanced/timing/clock-interval/index.html.

Try adding a button with alert:

```
<input type="button" onclick="alert(123)" value="Demo alert"/>
```

When you press it, the interpreter hangs. Note how only the nearest setInterval execution gets queued, folllowing ones are ignored.

Create the clock red-green-blue clock as given below :

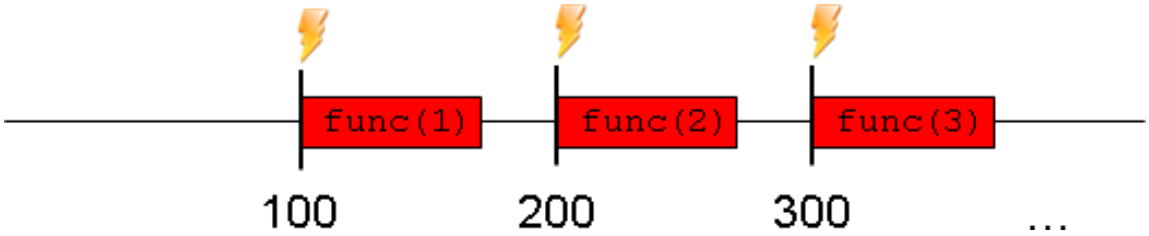
The starting code is here: tutorial/advanced/timing/clock-timeout-src/index.html.

Open solution

setInterval(func, delay) tries to execute the func every delay ms.

Let's take a setInterval(function() { func(i++) }, 100) as an example. It executes func every 100 ms, increasing the counter every run.

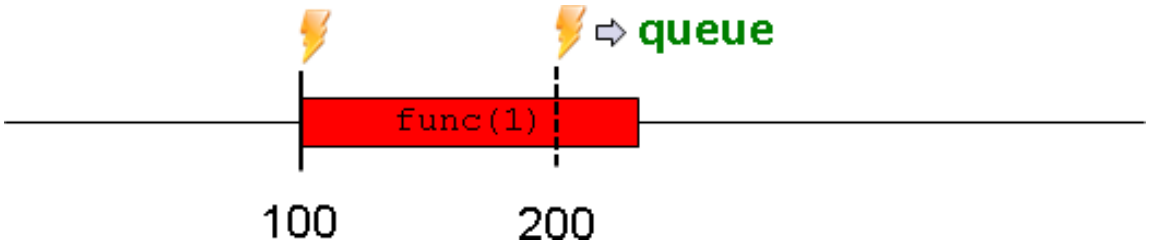
In the picture below, the red brick is func execution time. The time between bricks the time between executions is actually less than the delay:



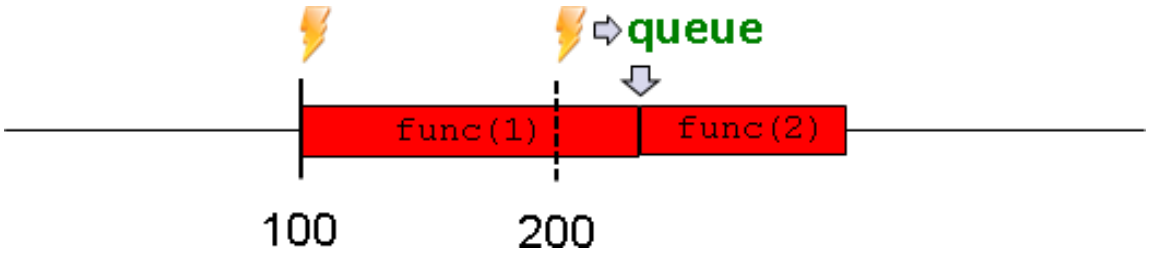
Interesting things start to happen when func takes more than the delay

The picture below shows an example of a long-running function.

The execution of setInterval gets queued and runs immediately when possible.

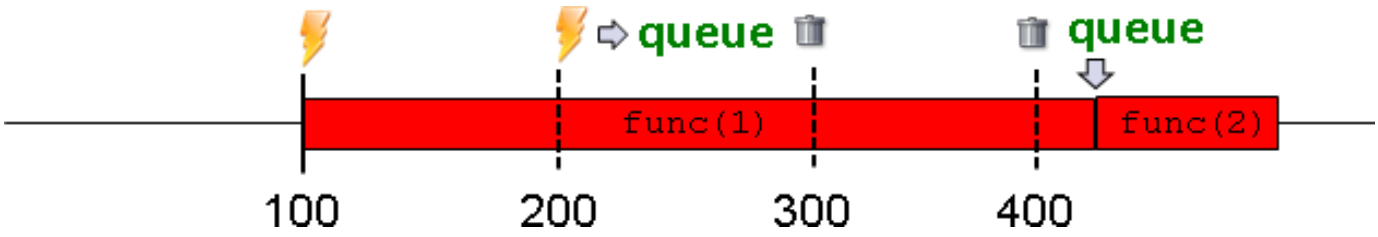


The real time between executions here is much more than delay ms.



There are cases when func takes more than scheduled runs.

On the picture below, setInterval tries to execute at 200 ms and queues the run. At 300 ms and 400 ms it wakes up again, but does nothing.



Only one execution can be queued.

Let's see how that affects real code. Run the example and await for an alert. Note that while the alert is shown, JavaScript execution is blocked, so we have a long-running function. Wait a while and press OK.

```
<input type="button" onclick="clearInterval(timer)" value="stop">

<script>
  var i = 1
  timer = setInterval(function() { alert(i++) }, 2000)
</script>
```

1. The browser runs the function every 2 seconds
2. - the execution gets blocked and remains blocked while the alert is shown.
3. If you wait long enough, the browser queues next execution and ignores those after it.
4. - the queued execution triggers immediately.
5. The next execution triggers with shorter delay. That's because scheduler wakes up each 2000ms. So, if the alert is released at 3500ms, then the next run is in 500ms.

setInterval(func, delay) does not guarantee a given delay between executions.

There are cases when the real delay is more or less than given.

In fact, it doesn't guarantee that there be any delay at all.

When we need a fixed delay, rescheduling with setTimeout is used.

Here is the example which gives alert every two seconds with setTimeout instead of setInterval:

```
<input type="button" onclick="clearTImeout(timer)" value="stop">

<script>
  var i = 1
  var timer = setTimeout(function() {
    alert(i++)
    timer = setTimeout(arguments.callee, 2000)
  }, 2000)
```

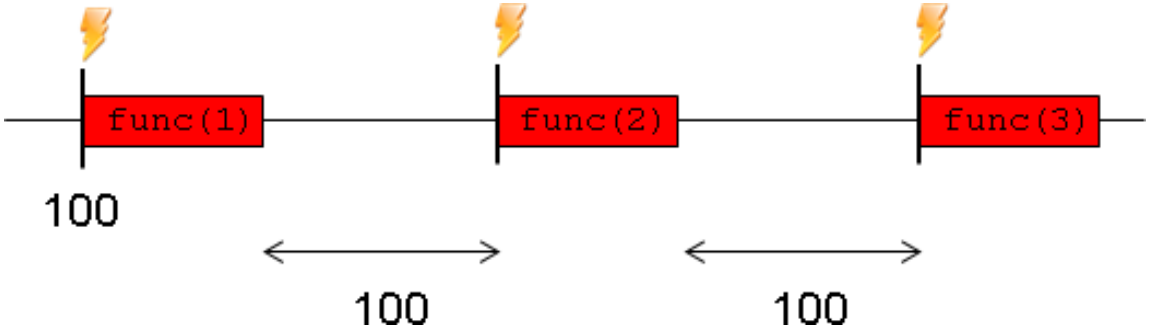
```
9 </script>
```

The trick is to reschedule the new execution every call.

The example below demonstrates the syntax for a named function:

```
01 <input type="button" onclick="clearTImeout(timer)" value="stop">
02
03 <script>
04   var i = 1
05
06   function func() {
07     alert(i++)
08     timer = setTimeout(func, 2000)
09   }
10   var timer = setTimeout(func, 2000)
11 </script>
```

The execution timeline will have fixed delays between executions (picture for 100 ms):



The timer resolution is limited. Actually, the minimal timer tick varies between 1ms and 15ms for modern browsers and can be larger for older ones.

If the timer resolution is 10, then there is no difference between setTimeout(..,1) and setTimeout(..,9).

Let's see that in the next example. It runs several timers, from 2 to 20. Each timer increases the length of the corresponding DIV.

Run it in different browsers and notice that for most of them, several first DIVs animate identically. That's exactly because .

```
01 <style> div { height: 18px; margin: 1px; background-
   color:green; } </style>
02
03 <input type="button" value="Click to stop">
04
05 <script>
06 onload = function() {
```

```
07   for(var i=0; i<=20; i+=2) {
08     var div = document.createElement('div')
09     div.innerHTML = i
10     document.body.appendChild(div)
11     animateDiv(div, i)
12   }
13 }
14
15 function animateDiv(div, speed) {
16   var timer = setInterval(function() {
17     div.style.width = (parseInt(div.style.width||0)+2)%400 +
   'px'
18   }, speed)
19
20   var stop = document.getElementsByTagName('input')[0]
21   var prev = stop.onclick
22   stop.onclick = function() {
23     clearInterval(timer)
24     prev && prev()
25   }
26 }
27 </script>
```

Behavior of setTimeout and setInterval with 0 delay is slightly different.

- In Opera, setTimeout(.. ,0), is same as setTimeout(.. ,10). It will execute often than setTimeout(.. ,2).
- In Internet Explorer, zero delay setInterval(.., 0) doesn't work. Changing the delay from 0 to 1 or using setTimeout helps.

Open the animation with `setTimeout`

In real world animations, it is not recommended to have many setInterval/setTimeout at the same time. They eat CPU.

It is much more preferred that a single setInterval/setTimeout call manages animation for multiple elements.

The button below should change it's value to 'OK', but it doesn't work. Why?

```
1 <input type="button"
2   onclick="setTimeout(function() { this.value='OK' }, 100)"
3   value="Click me"
4 >
```

The reason is . ,
or this=undefined in ES5 strict mode.

The reference is usually passed through closure. The following is fine:

```
<input id="click-ok" type="button" value="Click me">
<script>
  document.getElementById('click-ok').onclick = function() {
    var self = this
    setTimeout(function() { self.value='OK' }, 100)
  }
</script>
```

In object-oriented code, a scheduled method call may not work:

```
function User(login) {
  this.login = login
  this.sayHi = function() {
    alert(this.login)
  }
}

var user = new User('John')

setTimeout(user.sayHi, 1000)
```

The user.sayHi indeed executes, but . It takes the bare function and schedules it.

To put it clear, these two setTimeout do the same:

```
setTimeout(user.sayHi, 1000)

var f = user.sayHi
setTimeout(f, 1000)
```

There are two ways to solve the problem. The first is to create an intermediate function:

```
function User(name) {
  this.name = name
  this.sayHi = function() {
    alert(this.name)
  }
}

var user = new User('John')

setTimeout(function() {
  user.sayHi()
```

```
12 }, 1000)
```

The second way is to bind sayHi to the object by referencing it through closure instead of using this:

```
01 function User(name) {
02   this.name = name
03
04   var self = this
05
06   this.sayHi = function() {
07     alert(self.name)
08   }
09 }
10
11 var user = new User('John')
12
13 setTimeout(user.sayHi, 1000)
```