
*Computer Science Correspondence School*

*discere domi*

**Lecture 6**

# CISC 131
# Introduction to Programming and Problem Solving
# Spring 2020
# Javascript *setTimeout* Function and Function Closures

**Due: Friday, April 3, 2020**
**Points: none**

## Solution to Problem from Lecture 5

Many of the functions written for the HMTL table element exercise had simple solutions. Shown below are a few of the more challenging functions. You may have implemented these functions differently from that shown here but please study this implementation to help improve your reading and understanding of other people's programs.

```
window.onload = function()
{
document.getElementById("container").innerHTML=createTable(getTableId(),10,20);
 setInterval(updateTable,250);
};

function updateTable()
{
 var cell;
 var col;
 var row;

 row = getRandomInteger ( getTableRowCount()-1 );
 col = getRandomInteger ( getTableColumnCount()-1 );

 cell = getTableCell ( row, col );
 cell.innerHTML = getRandomAlphabetCharacter();

 cell.style.color      = getRandomRGB();
 cell.style.fontFamily = getRandomFontFamily();
 cell.style.fontSize   = getRandomFontSize();
 cell.style.fontStyle  = getRandomFontStyle();
 cell.style.fontWeight = getRandomFontWeight();

 cell.style.textAlign = "center";

 cell.style.backgroundColor = getRandomRGB();

}

function createTable   ( tableId, rows, columns )
{
 var html;
 var i;
 var j;
 var oneRow;

 html = "";

 i=0;
 while ( i<rows )
```

```
 {
  oneRow = "";
  j = 0;
  while ( j < columns )
  {
   oneRow = oneRow +
            createHTMLElement("td", getCellId(i,j), null, null)
   j = j + 1;
  }
  oneRow = createHTMLElement("tr", null, null, oneRow);
  html   = html + oneRow;
  i = i + 1;
 }
 html = createHTMLElement("table", tableId, null, html)
 return html;
}

function getRandomFontStyle()
{
 var i;
 var result;

 i = getRandomInteger ( 2 );
 result = "normal";
 if ( i == 1 ) result = "italic";
 if ( 1 == 2 ) result = "oblique";
 return result;
}

function getRandomFontFamily()
{
 var i;
 var result;

 result = "serif";
 i = getRandomInteger ( 11 );

if ( i ==  0 ) result = '"Lucida Console", Monaco, monospace';
if ( i ==  1 ) result = '"Times New Roman", Times, serif';
if ( i ==  2 ) result = 'Georgia, serif';
if ( i ==  3 ) result = '"Palatino Linotype", "Book Antiqua", Palatino, serif';
if ( i ==  4 ) result = 'Arial, Helvetica, sans-serif';
if ( i ==  5 ) result = '"Arial Black", Gadget, sans-serif';
if ( i ==  6 ) result = '"Comic Sans MS", cursive, sans-serif';
if ( i ==  7 ) result = '"Lucida Sans Unicode", "Lucida Grande", sans-serif';
if ( i ==  8 ) result = 'Tahoma, Geneva, sans-serif';
if ( i ==  9 ) result = '"Trebuchet MS", Helvetica, sans-serif';
if ( i == 10 ) result = 'Verdana, Geneva, sans-serif';
if ( i == 11 ) result = '"Courier New", Courier, monospace';
 return result;
}
```

## Lab Assignment

Many modern programming languages have some method of letting the program *pause* - suspend the execution of statements for a fixed amount of time. This is quite common in object-oriented languages where the program often consists of independent processes that must keep in synchronization with each other. Javascript does not have this kind of capability. It does, however, have two methods in the *Window Object Model* that allow you to schedule when a function should be executed. One is called *setInterval* and the other is called *setTimeout.* The

former calls a function every *n* milliseconds[1] while the later schedules a function to be called *n* milliseconds after the *setTimeout* function is called. You used *setInterval* when you did the *clock* project. *setInterval* calls a function repeatedly. *setTimeout* calls a function only once.

The normal order of statement execution in any programming language is the order in which the statements appear. For example, when a function is called, and after the formal parameters have been created and initialized, the statements in the function are executed one at a time in the order they are listed. Execution of a statement does not begin until the execution of the statement prior to it has been completed.

Some statements will temporarily change this normal order of execution. Loops, conditionals, and function calls all temporarily change this normal order each according to the statement's purpose but the fundamental concept that one statement does not begin execution until the statement before it completes its execution remains. In languages that support a *pause* type statement, there is simply a time delay between the end of the execution of one statement and the start of the execution of the subsequent statement.

Javascript has no statement that is equivalent to a *pause* type statement. Instead, it has a way in which the execution of any function can be scheduled to take place after some fixed time delay. This is done by calling the *window.setTimeout* function. The *window.setTimeout* function has this syntax:

```
window.setTimeout(targetFunction, delayTime)
```
[2]

where the *targetFunction* is the function that is to be executed and *delayTime*, is the integer number of milliseconds after which the *targetFunction* should be called.

Once the *window.setTimeout* function schedules the *targetFunction* for invocation, the statement immediately following the *window.setTimeout* function is then executed. After approximately *delayTime* milliseconds, the *targetFunction* will begin execution. The *targetFunction* can be specified in a number of different ways. The *delayTime* must be an expression that evaluates to a non-negative integer whose value will be interpreted as a number of milliseconds.

In this exercise, you will write a short program that illustrates the effect of using the *window.setTimeout* function.

**Step 1**

Create empty *html* and Javascript files. Hook the *js* file to the *html* file. Make sure the Firefox error console is open. The *html* file should have a single *div* in the *body*. Give this *div* the *id* of *page*.

In the *window.onload* function, declare a variable called *element* and assign it a reference to the *html* element with the *id* of *page*. Test this code to make sure the reference has been established.

Write a function named *timeoutTest* that contains the same statements as the *onload* function: variable declaration of *element*, assignment to *element* of a reference to the *html* element with the *id* of *page*.

Place the following statements in the *onload* function after the existing statements:

```
element.innerHTML = "before call to setTimeout";
```

[1] One millisecond is equal to 1/1000 of a second.

[2] The function returns a value which can be used to later terminate the timeout. We won't show the use of the value here.

```
element.innerHTML =  element.innerHTML + "<br/>after call to setTimeout";
```

The `<br/>` tag is an *html* tag that causes the text following it to be placed on a new line by the browser. Excess whitespace is ignored in *html*, so using an *escape character*, such as the newline character (*\n)*, would have no effect on the display of the text by the browser. *Escape characters* work only where the text will not be processed as *html*, such as in a *window.alert* box.

When you launch the *html* page in the browser, the display should look like this:

```
before call to setTimeout
after call to setTimeout
```

Now modify the *timeoutTest* function by placing the following statement after the other statements in the function:

```
element.innerHTML = element.innerHTML + "<br/>inside the timeoutTest function";
```

Finally, place a call to the *timeoutTest* function between the two *innerHTML* assignment statements in the *onload* function like this:

```
element.innerHTML = "before call to setTimeout";
timeoutTest();
element.innerHTML =  element.innerHTML + "<br\>after call to setTimeout";
```

The text displayed when the *html* page is loaded should look like this:

```
before call to setTimeout
inside the timeoutTest function
after call to setTimeout
```

Given the normal order of statement execution, this output is what you would expect to see. Each statement is executed in the order the statements are listed with the function call temporarily changing that normal order of statement execution.

**Step 2**

To see the effect of using the *window. setTimout* function, change the *onload* function by replacing the call to the *timeoutTest* function with this statement:

```
window.setTimeout("timeoutTest()", 2000);
```

This statement says that the *timeoutTest* function should be called two seconds (2000 milliseconds) after *window.setTimout* statement is executed. When the *html* page is loaded into the browser, two lines of text appear and then, after a two second delay, the final line of text appears. The displayed text should look like this:

```
before call to setTimeout
after call to setTimeout
inside the timeoutTest function
```

As you can see, the *window.setTimeout* function call scheduled the *timeoutTest* function for latter execution, it did not call the function immediately. This can be a very useful tool and is the key to doing some kinds of animation using Javascript.

The actual parameter value of the *targetFunction* formal parameter in a call to the *window.setTimeout* function can be specified in three ways:

- by placing the function call in a *String*, as shown in the above example
- by using a *reference* to the function as shown here:
  `window.setTimeout(timeoutTest, 500); // half second delay`
- by using a Javascript closure

For our purposes, a Javascript *closure* can be thought of as a function definition that is contained inside another function definition. This powerful feature is not commonly supported in other programming languages. Using *closures* is not necessary for most things and its improper use can cause problems, but if you need to send an actual parameter to the *targetFunction* when using *window.setTimeout*, a *closure* is the only mechanism that will work.

**Step 3**

To test out the use of a *closure*, make the following changes to the *timeoutTest* function definition:

- remove the declaration of the variable *element*
- create a formal parameter called *element*.
- remove the statement that assigns *element* a reference to the *html* element whose *id* is *page*

The function should now have only one statement in it: the assignment statement for *element.innerHTML*.

After these changes, a call to the *timeoutTest* function is now defined to require an actual parameter. A *closure* must be used in order to call use this function in a *setTimeout*. The actual parameter that we want to use is the variable *element* that is declared and assigned a value in the *onload* function. In the *unload* function, change the call to *window.setTimeout* to this:

```
window.setTimeout ( function() { timeoutTest(element); },2000 );
```

The *targetFunction* is now a *closure*: an anonymous[3] function whose definition contains a call to the *timeoutTest* function. Load the *html* page in the browser. The test displayed should look just as it did before:

```
before call to setTimeout
after call to setTimeout
inside the timeoutTest function
```

Care must be taken when writing this kind of code, however. The function *timeoutTest* depends on *element* having a value when the function is eventually called. The *window.setTimeout* does not call the function, it only schedules it to be called later. If you change the value of the actual parameter(s) sent to *targetFunction* before *targetFunction* is called, you may experience unexpected results. Here is an example. Change the pertinent statements in the *onload* function to these:

```
element.innerHTML = "before call to setTimeout";
window.setTimeout(function() {timeoutTest(element);},2000);
element.innerHTML =  element.innerHTML + "<br>after call to setTimeout";
element = null;
```

The variable *element* is set to *null* before the call to the *timeoutTest* function is actually made, not just scheduled to be made. When you run this new version, the error console should show that the parameter *element* in the *timeoutTest* function cannot have its inner html set because element no longer references and an element from the *html* page.

Function closures can be also used with the *setInterval* function and the assignment of events such as *onclick*.

[3] An anonymous function is one that has no name. An example is the *window.onload* function.

**Exercise 1**

In this exercise, you will get some practice using function closures and the *setTimeout* function. You will need your *getRandomInteger* and *getRandomRGB* functions.

To begin, create a new *html* file and a new *Javascript* file to go with it. Create an *onload* function in the Javascript and copy the *getRandomRGB* and *getRandomInteger* functions into the Javascript file after the *onload* function. Do the usual testing to ensure they are hooked together. Be sure to have the error console open to check for Javascript errors.

Create a *div* with an *id* of *box* and style it to have a border and be fairly large, say 20em by 20em. Check to be sure it displays correctly in the browser.

In the *onload* function, store a reference to the *box div* in the variable named *element*. Test to ensure you have properly created and stored the reference.

Write a function with this header:

```
function changeColor ( element, timeDelay )
```

The function does not return a value. The function changes the background color of the reference stored in its *element* formal parameter to a random color. For now, the *timeDelay* formal parameter will not be used.

Call the function from the *onload* function passing it the reference stored there and the value 5000 as actual parameters. The result should be that when the page loads the *box div* is assigned a random background color.

In the *onload*, establish an *onclick* behavior with the *box* that calls the *changeColor* function. You will need to use a function closure to do this. Here is a partial example:

```
element.onclick = function() { … };
```

Now when the page loads, clicking on the box should cause its background color to change. Once that is working correctly, modify the *changeColor* function so that its *element* formal parameter no longer has an *onclick* behavior. Now when the page loads, clicking the *box* the first time will change its background color but subsequent clicks will not change its background color.

Now modify the *changeColor* function to have it be scheduled to be called *timeDelay* milliseconds after its background color has been changed. This is done by adding a call to the *window.setTimeout* function that will schedule the execution of the *changeColor* function after a delay of *timeDelay* milliseconds like this:

```
window.setTimeout(function() {changeColor(element, timeDelay);}, timeDelay);
```

In the above statement, the first *timeDelay* is the actual parameter to the *changeColor* function call and the second *timeDelay* is the actual parameter to the *setTimeout* function call. This statement does *not* make a recursive call to *changeColor*. Rather, it schedules the execution of a call to the function to happen after *timeDelay* milliseconds. When the page is opened in the browser, a click of the box will cause it to change color. From that time on, every five seconds (5000 was passed as the actual parameter when the *onload* established the *onclick* behavior) the color will change again.

Finally, modify the *changeColor* function in such a way that it is scheduled to be called in 10% less time than the preceding time it was scheduled. However, the schedule time should never be less than 100 milliseconds. Now when the browser displays the page, clicking on the box will

start the background color to change. The color will gradually change more frequently until it is changing every 100 milliseconds.