Javascript Timing Events

The setInterval() Method

The setInterval() method will wait a specified number of milliseconds, and then execute a specified function, and it will continue to execute the function, once at every given time-interval.

Syntax

```
window.setInterval("javascript function", milliseconds);
```

The window.setInterval() method can be written without the window prefix.

The first parameter of setInterval() should be a function.

The second parameter indicates the length of the time-intervals between each execution.

Note: There are 1000 milliseconds in one second.

Example

Alert "hello" every 3 seconds:

```
setInterval(function () {alert("Hello")}, 3000);
```

The example show you how the setInterval() method works, but it is not very likely that you want to alert a message every 3 seconds.

Below is an example that will display the current time. The setInterval() method is used to execute the function once every 1 second, just like a digital watch.

Example

Display the current time:

```
var myVar=setInterval(function () {myTimer()}, 1000);


function myTimer() {
    var d = new Date();
      document.getElementById("demo").innerHTML = d.toLocaleTimeString();
}
```

How to Stop the Execution?

The clearInterval() method is used to stop further executions of the function specified in the setInterval() method.

Syntax

```
window.clearInterval(intervalVariable)
```

The window.clearInterval() method can be written without the window prefix.

To be able to use the clearInterval() method, you must use a global variable when creating the interval method:

```
myVar=setInterval("javascript function", milliseconds);
```

Then you will be able to stop the execution by calling the clearInterval() method.

Example

Same example as above, but we have added a "Stop time" button:

```
<p id="demo"></p>

<button onclick="clearInterval(myVar)">Stop time</button>

<script>
var myVar = setInterval(function () {myTimer()}, 1000);
function myTimer() {
      var d = new Date();
      document.getElementById("demo").innerHTML = d.toLocaleTimeString();
}
</script>
```

The setTimeout() Method

Syntax

```
window.setTimeout("javascript function", milliseconds);
```

The window.setTimeout() method can be written without the window prefix.

The setTimeout() method will wait the specified number of milliseconds, and then execute the specified function.

The first parameter of setTimeout() should be a function.

The second parameter indicates how many milliseconds, from now, you want to execute the first parameter.

Example

Click a button. Wait 3 seconds. The page will alert "Hello":

```
<button onclick = "setTimeout(function(){alert('Hello')},3000)">Try it</button>
```

How to Stop the Execution?

The clearTimeout() method is used to stop the execution of the function specified in the setTimeout() method.

Syntax

`window.clearTimeout(timeoutVariable)`

The window.clearTimeout() method can be written without the window prefix.

To be able to use the clearTimeout() method, you must use a global variable when creating the timeout method:

`myVar=setTimeout("javascript function", milliseconds);`

Then, if the function has not already been executed, you will be able to stop the execution by calling the clearTimeout() method.

Same example as above, but with an added "Stop" button:

```
<button onclick="myVar=setTimeout(function(){alert('Hello')},3000)">Try it</button>

<button onclick="clearTimeout(myVar)">Try it</button>
```

JavaScript Errors - Throw and Try to Catch

The try statement lets you test a block of code for errors.

The catch statement lets you handle the error.

The throw statement lets you create custom errors.

The finally statement lets you execute code, after try and catch, regardless of the result.

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
try {
      adddlert("Welcome guest!");
}
catch(err) {
      document.getElementById("demo").innerHTML = err.message;
}
</script>

</body>
</html>
```

JavaScript try and catch

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The JavaScript statements try and catch come in pairs:

```
try {
    Block of code to try
}
catch(err) {
    Block of code to handle errors
}
```

JavaScript Throws Errors

When an error occurs, JavaScript will normally stop, and generate an error message.

The technical term for this is: JavaScript will throw an error.

The throw Statement

The throw statement allows you to create a custom error.

The technical term for this is: throw an exception.

The exception can be a JavaScript String, a Number, a Boolean or an Object:

```
throw "Too big";    // throw a text

throw 500;          // throw a number
```

If you use throw together with try and catch, you can control program flow and generate custom error messages.

Input Validation Example

```
<!DOCTYPE html>
<html>
<body>

<p>Please input a number between 5 and 10:</p>

<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>
<p id="message"></p>

<script>
function myFunction() {
    var message, x;
    message = document.getElementById("message");
    message.innerHTML = "";
    x = document.getElementById("demo").value;
    try {
        x = Number(x);
        if(x == "") throw "empty";
        if(isNaN(x)) throw "not a number";
        if(x > 10) throw "too high";
        if(x < 5) throw "too low";
    }
    catch(err) {
        message.innerHTML = "Input is " + err;
    }
}
</script>

</body>
</html>
```

The finally Statement

The finally statement lets you execute code, after try and catch, regardless of the result:

```
try {
    Block of code to try
}
catch(err) {
    Block of code to handle errors
}
finally {
    Block of code to be executed regardless of the try / catch result
}
```

Example

```
function myFunction() {
    var message, x;
      message = document.getElementById("message");
      message.innerHTML = "";
    x = document.getElementById("demo").value;
    try {
      x = Number(x);
        if(x == "") throw "is empty";
        if(isNaN(x)) throw "is not a number";
        if(x > 10) throw "is too high";
        if(x < 5) throw "is too low";
      }
    catch(err) {
      message.innerHTML = "Error: " + err + ".";
      }
    finally {
          document.getElementById("demo").value = "";
      }
}
```

Another very cool shortcut notation is the ternary notation for conditions. So, instead of the following...

```
var direction;
if(x < 200){
  direction = 1;
} else {
  direction = -1;
}
```

... You could write a shorter version using the ternary notation:

```
var direction = x < 200 ? 1 : -1;
```

The `true` case of the condition is after the question mark, and the other case follows the colon.

7. FUNCTIONS CAN EXECUTE THEMSELVES

There's no denying it:

```
(function() { alert('hello'); })(); //alerts 'hello'
```

The syntax is simple enough: we declare a function and immediately call it just as we call other functions, with `()` syntax. You might wonder why we would do this. It seems like a contradiction in terms: a function normally contains code that we want to execute later, not now, otherwise we wouldn't have put the code in a function.

One good use of self-executing functions (SEFs) is to **bind the current values of variables** for use inside delayed code, such as callbacks to events, timeouts and intervals. Here is the problem:

```
var someVar = 'hello';
setTimeout(function() { alert(someVar); }, 1000);
var someVar = 'goodbye';
```

Newbies in forums invariably ask why the `alert` in the `timeout` says `goodbye` and not `hello`. The answer is that the `timeout` callback function is precisely that—a callback—so it doesn't evaluate the value of `someVar` until it runs. And by then, `someVar` has long since been overwritten by `goodbye`.

SEFs provide a solution to this problem. Instead of specifying the timeout callback implicitly as we do above, we return it from an SEF, into which we pass the current value of `someVar` as arguments. Effectively, this means **we pass in and isolate the current value of `someVar`, protecting it from whatever happens to the actual variable `someVar` thereafter**. This is like taking a photo of a car before you respray it; the photo will not update with the resprayed color; it will forever show the color of the car at the time the photo was taken.

```
var someVar = 'hello';
setTimeout((function(someVar) {
    return function()  { alert(someVar); }
})(someVar), 1000);
var someVar = 'goodbye';
```

This time, it alerts `hello`, as desired, because it is alerting the isolated version of `someVar` (i.e. the function argument, *not* the outer variable).

Summary

The `do...while` statement creates a loop that executes a specified statement until the test condition evaluates to false. The condition is evaluated after executing the statement, resulting in the specified statement executing at least once.

Syntax

```
do
   statement
while (condition);
```

`statement`

A statement that is executed at least once and is re-executed each time the condition evaluates to true. To execute multiple statements within the loop, use a `block` statement (`{ ... }`) to group those statements.

`condition`

An expression evaluated after each pass through the loop. If `condition` evaluates to true, the `statement` is re-executed. When `condition` evaluates to false, control passes to the statement following the `do...while`.

Examples

Example: Using `do...while`

In the following example, the `do...while` loop iterates at least once and reiterates until `i` is no longer less than 5.

```
var i = 0;
do {
   i += 1;
   console.log(i);
} while (i < 5);
```

Syntax

```
while (condition) {
  statement
}
```

`condition`

An expression evaluated before each pass through the loop. If this condition evaluates to true, `statement` is executed. When condition evaluates to false, execution continues with the statement after the `while` loop.

`statement`

A statement that is executed as long as the condition evaluates to true. To execute multiple statements within the loop, use a block statement (`{ ... }`) to group those statements.

Examples

The following `while` loop iterates as long as `n` is less than three.

```
var n = 0;
var x = 0;

while (n < 3) {
  n++;
  x += n;
}
```

Each iteration, the loop increments `n` and adds it to `x`. Therefore, `x` and `n` take on the following values:

- After the first pass: `n` = 1 and `x` = 1
- After the second pass: `n` = 2 and `x` = 3
- After the third pass: `n` = 3 and `x` = 6

After completing the third pass, the condition `n` < 3 is no longer true, so the loop terminates.

Example: Using `for`

The following `for` statement starts by declaring the variable `i` and initializing it to `0`. It checks that `i` is less than nine, performs the two succeeding statements, and increments `i` by 1 after each pass through the loop.

```
for (var i = 0; i < 9; i++) {
   console.log(i);
   // more statements
}
```

Example: Optional `for` expressions

All three expressions in the head of the `for` loop are optional.

For example, in the *initialization* block it is not required to initialize variables:

```
var i = 0;
for (; i < 9; i++) {
    console.log(i);
    // more statements
}
```

Like the *initialization* block, the *condition* block is also optional. If you are omitting this expression, you must make sure to break the loop in the body in order to not create an infinite loop.

```
for (var i = 0;; i++) {
   console.log(i);
   if (i > 3) break;
   // more statements
}
```