Learn web development

Build your own function

Previous Overview: Building blocks Next

With most of the essential theory dealt with in the previous article, this article provides a practical experience — here you'll get some practice with building up your own custom function. Along the way, we'll also explain some further useful details of dealing with functions.

Prerequisites:	Basic computer literacy, a basic understanding of HTML and CSS, JavaScript first steps, Functions — reusable blocks of code.
Objective:	To provide some practice in building a custom function, and explain a few more useful associated details.

Active learning: Let's build a function

The custom function we are going to build will be called `displayMessage()`, and it will display a custom message box to the user over the top of a web site. It will function as a more useful replacement for the browser's built-in alert() function. You've seen this before, but let's just refresh our memories — try the following in your browser's JavaScript console, on any page you like:

```
alert('This is a message');
```

The function takes a single argument — the string that is displayed in the alert box. You can try varying the string to change the message.

The alert function is not really that good: you can alter the message, but you can't easily vary anything else, such as the color, or an icon, or anything else. We'll build one that will prove to be more fun.

> **Note**: This example should work in all modern browsers fine, but the styling might look a bit funny in slightly older browsers. We'd recommend you doing this exercise in a modern browser like Firefox, Opera, or Chrome.

The basic function

To begin with, let's put together a basic function.

> **Note**: For function naming conventions, you should follow the same rules as variable naming conventions. This is fine, as you can tell them apart — function names appear with parentheses after them, and variables don't.

1. We'd like you to start this one off by accessing the function-start.html file and making a local copy. You'll see that the HTML is simple — our body contains just a single button. We've also provided some basic CSS to style the custom message box, and an empty `<script>` element to put our JavaScript in.
2. Next, add the following inside the `<script>` element:

```
function displayMessage() {

}
```

   We start off with the keyword `function`, which means we are defining a function. This is followed by the name we want to give to our function, a set of parentheses, and a set of curly braces. Any parameters we want to give to our function go inside the parentheses, and the code that runs when we call the function goes inside the curly braces.

3. Finally, add the following code inside the curly braces:

```
var html = document.querySelector('html');

var panel = document.createElement('div');
panel.setAttribute('class', 'msgBox');
html.appendChild(panel);

var msg = document.createElement('p');
msg.textContent = 'This is a message box';
panel.appendChild(msg);

var closeBtn = document.createElement('button');
closeBtn.textContent = 'x';
panel.appendChild(closeBtn);

closeBtn.onclick = function() {

```

```
        panel.parentNode.removeChild(panel);
    }
```

This is quite a lot of code to go through, so we'll walk you through it bit by bit.

The first line uses a DOM API function called `document.querySelector()` to select the `<html>` element and store a reference to it in a variable called `html`, so we can do things to it later on:

```
var html = document.querySelector('html');
```

The next section uses another DOM API function called `Document.createElement()` to create a `<div>` element and store a reference to it in a variable called `panel`. This element will be the outer container of our message box.

We then use yet another DOM API function called `Element.setAttribute()` to set a `class` attribute on our panel with a value of `msgBox`. This is to make it easier to style the element — if you look at the CSS on the page, you'll see that the we are using a `.msgBox` class selector to style the message box and its contents.

Finally, we call a DOM function called `Node.appendChild()` on the `html` variable we stored earlier, which nests one element inside the other as a child of it. We specify the panel `<div>` as the child we want to append inside the `<html>` element. We need to do this as the element we created won't just appear on the page on its own — we need to specify where to put it.

```
var panel = document.createElement('div');
panel.setAttribute('class', 'msgBox');
html.appendChild(panel);
```

The next two sections make use of the same `createElement()` and `appendChild()` functions we've already seen to create two new elements — a `<p>` and a `<button>` — and insert them in the page as children of the panel `<div>`. We use their `Node.textContent` property — which represents the text content of an element — to insert a message inside the paragraph, and an 'x' inside the button. This button will be what needs to be clicked/activated when the user wants to close the message box.

```
var msg = document.createElement('p');
msg.textContent = 'This is a message box';
panel.appendChild(msg);

var closeBtn = document.createElement('button');

```

```
    closeBtn.textContent = 'x';
    panel.appendChild(closeBtn);
```

Finally, we use an `GlobalEventHandlers.onclick` event handler to make it so that when the button is clicked, some code is run to delete the whole panel from the page — to close the message box.

Briefly, the `onclick` handler is a property available on the button (or in fact, any element on the page) that can be set to a function to specify what code to run when the button is clicked. You'll learn a lot more about these in our later events article. We are making the `onclick` handler equal to an anonymous function, which contains the code to run when the button is clicked. The line inside the function uses the `Node.removeChild()` DOM API function to specify that we want to remove a specific child element of the HTML element — in this case the panel `<div>`.

```
closeBtn.onclick = function() {
  html.removeChild(panel);
}
```

Basically, this whole block of code is generating a block of HTML that looks like so, and inserting it into the page:

```
<div class="msgBox">
  <p>This is a message box</p>
  <button>x</button>
</div>
```

That was a lot of code to work through — don't worry too much if you don't remember exactly how every bit of it works right now! The main part we want to focus on here is the function's structure and usage, but we wanted to show something interesting for this example.

Calling the function

You've now got your function definition written into your `<script>` element just fine, but it will do nothing as it stands.

1. Try including the following line below your function to call it:

   ```
   displayMessage();
   ```

   This line invokes the function, making it run immediately. When you save your code and reload it in the browser, you'll see the little message box appear immediately, only once. We are only calling it once, after all.

2. Now open your browser developer tools on the example page, go to the JavaScript console and type the line again there, you'll see it appear again! So this is fun — we now have a reusable function that we can call any time we like.

   But we probably want it to appear in response to user and system actions. In a real application, such a message box would probably be called in response to new data being available, or an error having occurred, or the user trying to delete their profile ("are you sure about this?"), or the user adding a new contact and the operation completing successfully ... etc.

   In this demo, we'll get the message box to appear when the user clicks the button.

3. Delete the previous line you added.
4. Next, we'll select the button and store a reference to it in a variable. Add the following line to your code, above the function definition:

   ```
   1 var btn = document.querySelector('button');
   ```

5. Finally, add the following line below the previous one:

   ```
   1 btn.onclick = displayMessage;
   ```

   In a similar way to our `closeBtn.onclick...` line inside the function, here we are calling some code in response to a button being clicked. But in this case, instead of calling an anonymous function containing some code, we are calling our function name directly.
6. Try saving and refreshing the page — now you should see the message box appear when you click the button.

You might be wondering why we haven't included the parentheses after the function name. This is because we don't want to call the function immediately — only after the button has been clicked. If you try changing the line to

```
1 btn.onclick = displayMessage();
```

and saving and reloading, you'll see that the message box appears without the button being clicked! The parentheses in this context are sometimes called the "function invocation operator". You only use them when you want to run the function immediately in the current scope. In the same respect, the code inside the anonymous function is not run immediately, as it is inside the function scope.

If you tried the last experiment, make sure to undo the last change before carrying on.

Improving the function with parameters

As it stands, the function is still not very useful — we don't want to just show the same default message every time. Let's improve our function by adding some parameters, allowing us to call it with some different options.

1. First of all, update the first line of the function:

```
function displayMessage() {
```

   to this:

```
function displayMessage(msgText, msgType) {
```

   Now when we call the function, we can provide two variable values inside the parentheses to specify the message to display in the message box, and the type of message it is.

2. To make use of the first parameter, update the following line inside your function:

```
msg.textContent = 'This is a message box';
```

   to

```
msg.textContent = msgText;
```

3. Last but not least, you now need to update your function call to include some updated message text. Change the following line:

```
btn.onclick = displayMessage;
```

   to this block:

```
btn.onclick = function() {
  displayMessage('Woo, this is a different message!');
};
```

   If we want to specify parameters inside parentheses for the function we are calling, then we can't call it directly — we need to put it inside an anonymous function so that it isn't in the immediate scope and therefore isn't called immediately. Now it will not be called until the button is clicked.

4. Reload and try the code again and you'll see that it still works just fine, except that now you can also vary the message inside the parameter to get different messages displayed in the box!

A more complex parameter

On to the next parameter. This one is going to involve slightly more work — we are going to set it so that depending on what the `msgType` parameter is set to, the function will display a different icon and a

different background color.

1. First of all, download the icons needed for this exercise (warning and chat) from GitHub. Save them in a new folder called `icons` in the same location as your HTML file.

   > **Note**: warning and chat icons found on iconfinder.com, and designed by Nazarrudin Ansyari. Thanks!

2. Next, find the CSS inside your HTML file. We'll make a few changes to make way for the icons. First, update the `.msgBox` width from:

   ```
   width: 200px;
   ```

   to

   ```
   width: 242px;
   ```

3. Next, add the following lines inside the `.msgBox p { ... }` rule:

   ```
   padding-left: 82px;
   background-position: 25px center;
   background-repeat: no-repeat;
   ```

4. Now we need to add code to our `displayMessage()` function to handle displaying the icon. Add the following block just above the closing curly brace (}) of your function:

   ```
   if (msgType === 'warning') {
     msg.style.backgroundImage = 'url(icons/warning.png)';
     panel.style.backgroundColor = 'red';
   } else if (msgType === 'chat') {
     msg.style.backgroundImage = 'url(icons/chat.png)';
     panel.style.backgroundColor = 'aqua';
   } else {
     msg.style.paddingLeft = '20px';
   }
   ```

   Here, if the `msgType` parameter is set as `'warning'`, the warning icon is displayed and the panel's background color is set to red. If it is set to `'chat'`, the chat icon is displayed and the panel's background color is set to aqua blue. If the `msgType` parameter is not set at all (or to something different), then the `else { ... }` part of the code comes into play, and the paragraph is simply given default padding and no icon, with no background panel color set either. This provides a default state if no `msgType` parameter is provided, meaning that it is an optional parameter!

5. Let's test out our updated function, try updating the `displayMessage()` call from this:

```
displayMessage('Woo, this is a different message!');
```

to one of these:

```
displayMessage('Your inbox is almost full — delete some mails', 'warning
displayMessage('Brian: Hi there, how are you today?','chat');
```

You can see how useful our (now not so) little function is becoming.

Note: If you have trouble getting the example to work, feel free to check your code against the finished version on GitHub (see it running live also), or ask us for help.

Conclusion

Congratulations on reaching the end! This article took you through the entire process of building up a practical custom function, which with a bit more work could be transplanted into a real project. In the next article we'll wrap up functions by explaining another essential related concept — return values.

Previous Overview: Building blocks Next