

Advanced JavaScript Programming

Advanced Techniques

Lesson 1, Activity 2: Optional Function Arguments

When we declare a function in JavaScript, we normally include a list of the arguments the function expects.

Code Sample:

[AdvancedTechniques/Demos/sumAll-1.html](#)

```
---- C O D E   O M I T T E D ----
function sumValues(val1, val2, val3) {
    return val1 + val2 + val3;
}
---- C O D E   O M I T T E D ----
```

But this does not guarantee that our function will always be called with three arguments. It's perfectly valid for someone to call our function passing fewer or more than three arguments.

```
var result1 = sumValues(3, 5, 6, 2, 7);
var result2 = sumValues(12, 20);
```

Both calls will return surprising results (surprising from the caller's perspective).

In the first case, since we are not expecting more than three arguments, the extra values, 2 and 7, will simply be ignored. It's bad because the returned value is probably not what the calling code expected.

It's even worse when we look at the second example. We are passing only two arguments. What happens to `val3`? It will have the value of `undefined`. This will cause the resulting sum to be `NaN`, which is clearly undesirable.

Let's fix our function to deal with these types of situations.

Code Sample:

[AdvancedTechniques/Demos/sumAll-2.html](#)

```
---- C O D E   O M I T T E D ----

function sumValues(val1, val2, val3) {
    if (val1 === undefined) {
        val1 = 0;
    }

    if (val2 === undefined) {
        val2 = 0;
    }

    if (val3 === undefined) {
        val3 = 0;
    }

    return val1 + val2 + val3;
}

var result1 = sumValues(3, 5, 6, 2, 7);
var result2 = sumValues(12, 20);

alert(result1);
alert(result2);
---- C O D E   O M I T T E D ----
```

If we run our example again, we will see that we no longer get `NaN` for the second function call. Instead we get 32, which is probably what the calling code expected.

We now have a pretty robust function that adds three numbers but it still doesn't feel all that useful. Sooner or later we will need to add four or five numbers and we don't want to be updating our function to accept additional parameters. That would be a less than desirable maintenance task. Fortunately, JavaScript can help us with that too.

Every function, when called, has a variable called `arguments`, which is an array of all the arguments passed to the function. Back to our example, the first time we call `sumValues`, the `arguments` array will contain `[3, 5, 6, 2, 7]` and in the second call `[12, 20]`.

What this means is that we can ignore the passed parameters altogether and deal only with the `arguments` array. Let's update our function once again.

Code Sample:

[AdvancedTechniques/Demos/sumAll-3.html](#)

```
---- C O D E   O M I T T E D ----

function sumValues() {
    var sum = 0;
    for (var i = 0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}
```

```
var result1 = sumValues(3, 5, 6, 2, 7);  
var result2 = sumValues(12, 20);  
  
alert(result1);  
alert(result2);  
---- C O D E    O M I T T E D ----
```

Note how we got rid of the parameter list and now we get the values directly from the `arguments` array. When we run our example now we see that the returned values are correct and precisely what was expected by the caller. We now have a function that accepts as many parameters as are thrown at it and will always return the sum of all those arguments.

Lesson 1, Activity 3: Truthy and Falsy

JavaScript, as we already know, has a boolean data type, which has only two possible values: `true` or `false`. Boolean expressions (e.g., `a == b`) also evaluate to a boolean value. But that's not the entire story.

When used in a boolean context (e.g. an if condition) non-boolean expressions are implicitly converted to booleans. This process is called **Type Coercion**. For example, in the expression `if (1) { alert('Hi!') }`, `1` is interpreted as `true`, so the alert will pop up. The value `1` is said to be *truthy*. The value `0`, on the other hand, is said to be *falsy*.

When used in a context that expects a boolean value, any JavaScript expression can be used. See below:

Code Sample:

[AdvancedTechniques/Demos/truthy-falsy.html](#)

```

---- C O D E   O M I T T E D ----

var truthies = [
  "false",
  "0",
  -1,
  "null",
  "undefined",
  "NaN",
  5/0,
  Infinity
];

var foo;
var falsies = [
  null,
  undefined,
  foo,
  0,
  NaN,
  ""
];
---- C O D E   O M I T T E D ----

for (var t=0; t<truthies.length; t++) {
  document.write("<li>" + truthies[t] + " : " + Boolean(truthies[t]) + "</li>");
}
---- C O D E   O M I T T E D ----

for (var f=0; f<falsies.length; f++) {
  document.write("<li>" + falsies[f] + " : " + Boolean(falsies[f]) + "</li>");
}
---- C O D E   O M I T T E D ----

```

Open the above page in a browser and you'll see how the different values get evaluated when implicitly converted to booleans. The table below shows how different values are converted:

Truthy-Falsy Values

Value	Truthy-Falsy	Explanation
0	falsy	0 is falsy.
"0"	truthy	Non-zero-length strings are truthy.
-1	truthy	All numbers but 0 are truthy.
null	falsy	null is falsy.
"null"	truthy	Non-zero-length strings are truthy.
undefined	falsy	undefined is falsy.
"undefined"	truthy	Non-zero-length strings are truthy.
NaN	falsy	NaN is falsy.
"NaN"	truthy	Non-zero-length strings are truthy.
Infinity	truthy	Infinity is a reserved word and is a non-zero number.
5/0	truthy	5/0 evaluates to Infinity.
"" (empty string)	falsy	Zero-length strings are falsy.
"false"	truthy	Non-zero-length strings are truthy.

Type coercion is the reason we have the `===` (triple-equal or strictly equal) comparison operator in JavaScript. The regular equality operator `==` applies type coercion and sometimes your comparisons will not result as expected. Look at the following sample code:

Code Sample:

[AdvancedTechniques/Demos/type-coercion.html](#)

```
---- C O D E   O M I T T E D ----

var num = 0;
if (num == "") {
  alert('Hey, I did not expect to see this.');
```

```
}
if (num === "") {
  alert('This will not be displayed.');
```

```
}
---- C O D E   O M I T T E D ----
```

In the first `if` conditional, we compare two *false* values, and the type coercion will resolve both of them to *false*, causing the result of the comparison to be *true*, which is probably not the original intent of the code.

To detect the type difference (string vs. number) we would need to use the triple equal operator, as shown in the second `if` statement.

Lesson 1, Activity 6: Type Coercion

Duration: 5 to 10 minutes.

In this exercise, you will debug and fix a simple script.

1. Open [AdvancedTechniques/Exercises/type-coercion.html](#) in your editor.
2. Determine what is wrong with the code and fix it.
3. Be sure to test your solution in the browser.

Solution:

[AdvancedTechniques/Solutions/type-coercion.html](#)

```
---- C O D E   O M I T T E D ----  
  
var answer = prompt("What is 10-10?", "");  
if (answer == "0") {  
    alert("Right!");  
} else {  
    alert("Wrong!");  
}  
---- C O D E   O M I T T E D ----
```

This is actually trickier than it looks. In the original file, if the user presses **OK** on the prompt without entering anything, the alert will read "Right!" That's because an empty string is *falsey* and so is 0.

At first, you might think that all you need to do is change the double equals to a triple equals to account for the different types. But if you test that solution, you will find that you always get the "Wrong!" answer, even if you enter "0".

The problem is that the value entered in the prompt is a string and the string "0" does not strictly equal the number 0. That's why you have to change the condition to `answer === "0"`.

Lesson 1, Activity 7: Default Operator

The boolean operators `&&` and `||` also use *truthy* and *falsy* to resolve each of the operands to a boolean value.

From your previous experiences with other programming languages you may be led to believe that the result of a boolean operation is always `true` and `false`. This is not the case in JavaScript.

In JavaScript, the result of a boolean operation is the value of the operand that determined the result of the operation. Let's clarify that with an example:

Code Sample:

[AdvancedTechniques/Demos/default-operator.html](#)

```
---- C O D E   O M I T T E D ----

var a = 0, b = NaN, c = 1, d = "hi";
var result = ( a || b || c || d );
alert("Result: " + result);

result = ( d && c && b && a );
alert("Result: " + result);
---- C O D E   O M I T T E D ----
```

The first boolean expression `(a || b || c || d)` is evaluated from left to right until a conclusion is made. The `||` is the boolean OR operator, which only needs one of the operands to be `true` (truthy) for the operation result to be `true`:

1. `a` is 0, which is *falsy*. The evaluation continues with the remaining operands because *falsy* doesn't determine anything yet. We need one *truthy* value for the whole statement to be `true`.
2. `b` is NaN, also *falsy*. We have the same situation and we need to continue evaluating.
3. `c` is 1, which is *truthy*. We no longer need to continue evaluating the remaining operands because we already know the expression will result `true`.

But here's the catch. Instead of resulting strictly `true`, it will result in the *truthy* value that ended the evaluation, the value of `c` in this case. The message displayed will be "Result: 1".

As you might already expect, the `&&` (boolean AND operator) works in a similar fashion, but with opposite conditions. The AND operator returns the value of the last operand evaluated, which is either the first *falsy* operand or, in the case that all operands are *truthy*, the last operand in the expression. If for the expression `(d && c && b && a)` we follow the same sequence we did for the OR operator we will see that, `d` and `c` are both *truthy* so we need to keep going, then we get to `b`, which is *falsy* and causes the evaluation to stop and return `b`. The message displayed then is "Result: NaN".

You may be reading all this and thinking how can this be of any use. It turns out that this behavior of returning the first conclusive value can be very handy when ensuring that a variable is initialized.

Let's take a look at the function below.

```
function checkLength(text, min, max){
  min = min || 1;
  max = max || 10000;

  if (text.length < min || text.length > max) {
    return false;
  }
  return true;
}
```

The first two lines in this function make sure that `min` and `max` always have a valid value. This allows the function to be called like `checkValue("abc")`. In this case the `min` and `max` parameters will both start with the *undefined* value.

When we reach the line `min = min || 1;` we are simply assigning 1 to `min`, ensuring it overrides the *undefined*. Similarly we assign 1000 to `max`.

If we had passed actual values for these parameters as in `checkLength("abc", 2, 10)` these values would be kept because they are *truthy*.

With this usage of the `||` we are effectively providing default values for these two parameters. That's why this operator, in this context, is also called the *Default Operator*.

The default operator replaces more verbose code like:

```
if (min === undefined) {
  min = 1;
}
```

...becomes simply...

```
min = min || 1;
```

Here is another example of shortening your code using the default operator:

```
var contactInfo;
if (email) {
  contactInfo = email;
```

```

} else if (phone) {
  contactInfo = phone;
} else if (streetAddress) {
  contactInfo = streetAddress;
}

```

...is greatly shortened to...

```

var contactInfo = email || phone || streetAddress;

```

Default Operator Gotcha!

Be careful using the default operator with variables that can accept *falsy* values. The following example illustrates the danger:

Code Sample:

[AdvancedTechniques/Demos/default-operator-gotcha.html](#)

```

---- C O D E   O M I T T E D ----

function calculatePrice(basePrice, tax) {
  basePrice = basePrice || 0;
  tax = tax || .1; //default tax is 10%
  var price = (basePrice * (1 + tax)).toFixed(2);
  alert("The price including tax is " + price);
}

calculatePrice(100, 0); //no tax
---- C O D E   O M I T T E D ----

```

The `calculatePrice()` function uses the default operator to set a default `tax` of `.1`. The problem comes when we pass `0` as the value for `tax`. Our intent is to indicate that there is no tax; however, because `0` is *falsy*, `tax = tax || .1` evaluates to `.1`.

We can fix this by changing `tax = tax || .1` to `tax = (tax === 0) ? 0 : tax || .1`;

Lesson 1, Activity 9: Applying defaults to function parameters

Duration: 5 to 15 minutes.

Here we will revisit an earlier example and use the default operator to handle optional parameters.

1. Open `AdvancedTechniques/Exercises/sumAll-defaults.html` for editing.
2. Edit the existing `sumValues()` to use the default operator instead of the `if` blocks.

Code Sample:

AdvancedTechniques/Exercises/sumAll-defaults.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Sum all numbers, default operator</title>
<script type="text/javascript">
function sumValues(val1, val2, val3) {
  if (val1 === undefined) {
    val1 = 0;
  }

  if (val2 === undefined) {
    val2 = 0;
  }

  if (val3 === undefined) {
    val3 = 0;
  }

  return val1 + val2 + val3;
}

var result1 = sumValues(3, 5, 6, 2, 7);
var result2 = sumValues(12, 20);

alert(result1);
alert(result2);
</script>
</head>
<body>
<p>Nothing to show here.</p>
</body>
</html>
```

Solution:

AdvancedTechniques/Solutions/sumAll-defaults.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Sum all numbers, default operator</title>
<script type="text/javascript">
function sumValues(val1, val2, val3) {
  val1 = val1 || 0;
  val2 = val2 || 0;
  val3 = val3 || 0;

  return val1 + val2 + val3;
}

var result1 = sumValues(3, 5, 6, 2, 7);
var result2 = sumValues(12, 20);

alert(result1);
alert(result2);
</script>
</head>
<body>
<p>Nothing to show here.</p>
</body>
</html>
```

Lesson 1, Activity 10: Functions Passed as Arguments

In JavaScript, functions are first-class objects. Functions aren't just an immutable block of code that can only be invoked. Rather, each function we declare becomes an object, with its own properties and methods, that can be passed around like any other object.

Let's see how we can use functions as parameters to other functions. Consider the following example:

Code Sample:

[AdvancedTechniques/Demos/function-arguments.html](#)

```

---- C O D E   O M I T T E D ----

//let's create an array with 5 values
var values = [5, 2, 11, -7, 1];

//this function simply adds the values of 'a' and 'b' and returns the sum
function add(a, b) {
    return a+b;
}

//this function simply multiplies the value of 'a' times 'b' and returns the result
function multiply(a, b) {
    return a*b;
}

/* the 1st param passed to it is the numbers (in this case, the array of values)
 * the 2nd param passed to it is the initial Value (could be 0, 1, or whatever)
 * the 3rd param passed to it is the operation - which will actually be one of the functions declared above (add or subtract)
 */
function combineAll(nums, initialValue, operation) {
    //let's initialize the local variable 'runningResult' with the initialValue passed to it
    var runningResult = initialValue;
    //loop the nums array
    for (var i=0; i < nums.length; i++) {
        /* for each iteration, call the function (which can be add or multiply) passing the 2 params
         * on the first iteration, runningResult will be the initialValue; after that, it will be the value returned from operation(runningResult, nums
         * nums[i] simply refers to the current number in the iteration; the 1st will be 5, then 2, and so on.
         */
        runningResult = operation(runningResult, nums[i]);
    }
    return runningResult;
}

//notice the 3rd param passes the "add" function - you can pass a function into another function!
var sum = combineAll(values, 0, add);

//notice the 3rd param passes the "multiply" function - you can pass a function into another function!
var product = combineAll(values, 1, multiply);

alert("Sum: " + sum); //should be 12: 5+2+11+(-7)+1
alert("Product: " + product); //should be -770: 5x2x11x-7x1
---- C O D E   O M I T T E D ----

```

You may be wondering what the following function call means: `var sum = combineAll(values, 0, add);`. In this statement we are passing the function `add` as the third parameter of `combineAll`. We are not *invoking* `add` yet, just passing a reference to it. Note that the open and close parenthesis aren't used after `sum`. That should serve as a tipoff that this is not a function *invocation*.

The line that ultimately invokes `add` is:

```
runningResult = operation(runningResult, nums[i]);
```

...which received a reference to `add` in the `operation` parameter. When `operation` is invoked, in reality, it is `add` that is getting called, returning the sum of the two values passed in.

This is a very important technique and the `combineAll` function is often called *reduce*. Take your time to review the code and run the example until you feel comfortable with it. We will be using this capability extensively in the remaining lessons.

Lesson 1, Activity 11: Anonymous Functions

Going back to our previous example, the functions `add` and `multiply` are only referred to once, in each call to `combineAll`. Furthermore, if we stick to that pattern, we will need to create a new function for each new combination behavior we desire (e.g., concatenation) just so we can pass it to `combineAll`. That seems like too much overhead for such a simple thing.

Thankfully, we don't actually need to declare each of these functions. We don't even need to come up with names for them. JavaScript allows us to create functions *on the spot*, any time we need a function that will only be used in that location.

The syntax is rather compact:

```
function (arg1, arg2) {
  //function statements here
}
```

Because functions created this way don't have names, they are aptly called *anonymous functions*.

Let's revisit our previous example and use anonymous functions to replace the single-use functions we declared:

Code Sample:

[AdvancedTechniques/Demos/anon-func-arguments.html](#)

```
---- C O D E   O M I T T E D ----

var values = [5, 2, 11, -7, 1];

function combineAll(nums, initialValue, operation) {
  var runningResult = initialValue;
  for (var i=0; i < nums.length; i++) {
    runningResult = operation(runningResult, nums[i]);
  }
  return runningResult;
}

var sum = combineAll(values, 0, function (a, b) {
  return a+b;
});

var product = combineAll(values, 1, function (a, b) {
  return a*b;
});

var list = combineAll(values, 1, function (a, b) {
  return a + ", " + b;
});

alert("Sum: " + sum);
alert("Product: " + product);
alert("Number List: " + list);
---- C O D E   O M I T T E D ----
```

The highlighted code represents the three anonymous functions. The first two replace where we previously had `add` and `multiply` functions defined. The third anonymous function is used to concatenate the integers into a comma separated list.

Lesson 1, Activity 12: Nested Functions

Since functions in JavaScript are just one more type of object, we can create a function inside another function. These are called *nested functions* or *inner functions*.

The example below shows how to create and use a function inside another one:

Code Sample:

[AdvancedTechniques/Demos/inner-functions.html](#)

```

---- C O D E   O M I T T E D ----

var values = [5, 2, 11, -7, 1];

function combineAll(nums, initialValue, operator) {
  var runningResult = initialValue;
  for (var i=0; i < nums.length; i++) {
    runningResult = calculate(nums[i]);
  }
  return runningResult;

  function calculate(num) {
    switch (operator.toLowerCase()) {
      case "+" :
        return runningResult + num;
      case "x" :
        return runningResult * num;
      case "*" :
        return runningResult * num;
      case "a" :
        return runningResult + ", " + num;
    }
  }
}

var sum = combineAll(values, 0, "+");
var product = combineAll(values, 1, "x");
var list = combineAll(values, 1, "a");

alert("Sum: " + sum);
alert("Product: " + product);
alert("Number List: " + list);
---- C O D E   O M I T T E D ----

```

We will see more important uses of inner functions when we look at private members. For the time being, just notice how `calculate()` has access to `operator` and `runningResult`, which are scoped to the `combineAll()` function.

Lesson 1, Activity 15: Variable Scope

Variables in JavaScript are declared with the `var` keyword and are either *globally* or *locally* scoped.

A variable is globally scoped if any of these conditions are met:

1. It is declared outside of a function.
2. It is used without being declared using the `var` keyword.

A variable is locally scoped if it is declared within a function using the `var` keyword.

Global variables are accessible from anywhere within of your JavaScript code.

Code Sample:

[AdvancedTechniques/Demos/variable-scope.html](#)

```

---- C O D E   O M I T T E D ----

var maxNum; //global
function prepare() {
  var maxNum = 5; //local to function
  //let's forget the "var" in the "for" declaration
  for (i=0; i <= maxNum; ++i) { //no var, so i will be global
    //do something
  }
}

prepare();
alert(maxNum); //undefined
alert(i); //5 (because it is globally scoped)
---- C O D E   O M I T T E D ----

```

Notice that `maxNum` is declared globally but not assigned a value. It is also declared within the `prepare()` function and assigned the value of 5. When we read the `maxNum` value after calling `prepare()`, we see that the value is `undefined`. That's because it's reading the global variable. We can't access the function variable from outside of the function.

We can access the `i` variable though. That's because we assigned a value to it within the function without declaring it using `var`.

JavaScript does not have block scope, so variables are never local to an `if` statement or a `for` loop.

Lesson 1, Activity 16: Observing and Capturing Events

You are probably accustomed to using the HTML event handlers ("on" attributes) to capture events as in the demo below:

Code Sample:

[AdvancedTechniques/Demos/onclick.html](#)

```
---- C O D E   O M I T T E D ----  
  
<ul>  
  <li onclick="document.bgColor='red';">Red</li>  
  <li onclick="document.bgColor='orange';">Orange</li>  
  <li onclick="document.bgColor='green';">Green</li>  
  <li onclick="document.bgColor='blue';">Blue</li>  
</ul>  
---- C O D E   O M I T T E D ----
```

It's better, however, to keep your JavaScript completely separate from your HTML. In a later lesson we'll learn more about unobtrusively attaching events to elements in the DOM to avoid using HTML event handlers.

Lesson 1, Activity 17: The `eval()` Function

The reason we are mentioning `eval()` in this lesson is to acknowledge its existence and to urge you **not** to use it. We will explain why, but first let's explain what it does.

`eval` compiles and executes a string containing JavaScript code. It can be a simple expression like `"1 + 2"` or a long and complex script, with functions and all.

Here's one example that is not too different from what we can find in live sites on the web.

Code Sample:

[AdvancedTechniques/Demos/eval.html](#)

```

---- C O D E   O M I T T E D ----

function getProperty(objectName, propertyName) {
    //expression will be: document.title
    var expression = objectName + "." + propertyName;
    console.log(expression);
    //the eval() function evaluates the expression, so document.title gives us the title of the document
    var propertyValue = eval(expression);
    return propertyValue;
}

var prop = "title"; //assume this was given by the user
alert(getProperty("document", prop));
---- C O D E   O M I T T E D ----

```

This function creates a JavaScript expression by concatenating an object name, with a dot and a property name. Then it uses `eval()` to evaluate that expression.

As we can see `eval()` is a powerful function, but it is also potentially dangerous and incredibly inefficient. It's dangerous because it's typically used to evaluate user entered input, which not always is a safe thing to do. It's inefficient because *each call to `eval()` starts a JavaScript compiler*.

The use of `eval()` normally reveals lack of knowledge from the developer that wrote the script. In the example above, we could have used the `[]` accessor for properties. The same effect would be obtained with `alert(window[prop]);` without having to fire up a compiler just to retrieve the property value.

Remember this: *eval is evil*. Avoid it as much as you can. If you think you need it, maybe it's because you did not learn yet about an alternative way in JavaScript.

Lesson 1, Activity 18: Error Handling

No matter how careful you are, it always seems that errors find their way into your code. Sometimes they are runtime errors caused by unpredicted scenarios. Sometimes the errors are just incorrect behavior of your code, popularly known as *bugs*.

Fortunately, we have tools to deal with either type of problem. In this lesson we will talk about detecting and handling errors, debugging and fixing bugs in our applications.

Runtime Errors

Web browsers are such a hostile environment that it is almost guaranteed that we will constantly deal with runtime errors. Users provide invalid input in ways you didn't think of. New browser versions change their behavior. Ajax calls can fail for any number of reasons.

Many times we cannot prevent runtime errors from happening, but at least we can deal with them in a manner that makes the user experience less traumatic.

Unhandled Errors

Look at this seemingly trivial code sample:

Code Sample:

[AdvancedTechniques/Demos/simple-bug.html](#)

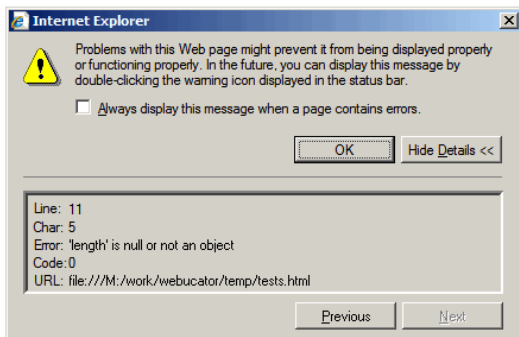
```

---- C O D E   O M I T T E D ----

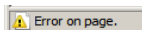
function getInput() {
  var name = prompt('Type your name');
  alert('Your name has ' + name.length + ' letters.');
```

It may not be obvious, but this code has a bug waiting to break free. If the user clicks *Cancel* or presses *Esc* the `prompt()` function will return `null`, which will cause the next line to fail with a null reference error.

If you as a programmer don't take any steps to deal with this error, it will simply be delivered directly to the end user, in the form of an utterly useless browser error message like the one below:



Depending on the user's browser or settings, the error message may be suppressed and only an inconspicuous icon shows up in the status bar:



This can be worse than the error message, leaving users thinking the application is unresponsive.

Globally Handled Errors

The `window` object has an event called `onerror` that is invoked whenever there's an unhandled error on the page.

Code Sample:

[AdvancedTechniques/Demos/simple-bug-onerror.html](#)

```

---- C O D E   O M I T T E D ----

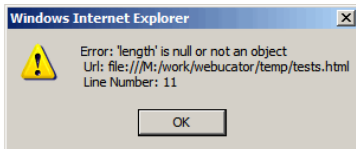
window.onerror = function (message, url, lineNo) {
  alert(
    'Error: ' + message +
    '\n Url: ' + url +
    '\n Line Number: ' + lineNo);
  return true;
}

function getInput() {
  var name = prompt('Type your name');
  alert('Your name has ' + name.length + ' letters.');
```


As you can see, the event will pass three arguments to the invoked function. The first one is the actual error message. The second one is the URL of the file containing the error (useful if the error is in an external .js file.) The last argument is the line number in that file where the error happened.

Returning `true` tells the browser that you have taken care of the problem. If you return `false` instead, the browser will proceed to treat the error as unhandled, showing the error message and the status bar icon.

Here's the message box that we will be showing to the user:



Structured Error Handling

The best way to deal with errors is to detect them as close as possible to where they occur. This will increase the chance that we know what to do with the error. To that effect JavaScript implements structured error handling, via the `try...catch...finally` block, which is also present in many other languages:

```
try {
  //try statements
} catch (error) {
  //catch statements
} finally {
  //finally statements
}
```

The idea is simple. If anything goes wrong in the statements that are inside the `try` block, then the statements in the `catch` block will be executed and the error will be passed into the `error` variable. The `finally` block is optional and, if present, is always executed last, whether or not an error is caught.

Let's fix our example to catch that error:

Code Sample:

[AdvancedTechniques/Demos/simple-bug-try-catch.html](#)

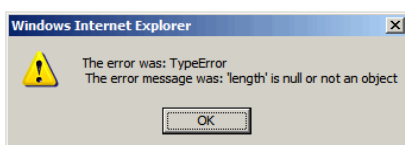
```
---- C O D E   O M I T T E D ----

window.onerror = function (message, url, lineNo) {
  alert(
    'Error: ' + message +
    '\n Url: ' + url +
    '\n Line Number: ' + lineNo);
  return true;
}

function getInput() {
  try {
    var name = window.prompt('Type your name');
    alert('Your name has ' + name.length + ' letters.');
```

The `error` object has two important properties: `name` and `message`. The `message` property contains the same error message that we have seen before. The `name` property contains the kind of error that happened and we can use that to decide if we know what to do with that error.

With that in place, if we reload the page and cancel out of the prompt, we will get the following alert:



It's good programming practice to only handle the error on the spot if you are certain of what it is and if you actually have a way to take care of it (other than just suppressing it altogether.) To better target our error handling code, we will change it to only handle errors named "TypeError", which is the error name that we have identified for this bug.

Code Sample:

AdvancedTechniques/Demos/simple-bug-trv-catch-specific.html

```

---- C O D E   O M I T T E D ----

window.onerror = function (message, url, lineNo) {
  alert(
    'Error: ' + message +
    '\n Url: ' + url +
    '\n Line Number: ' + lineNo);
  return true;
}

function getInput() {
  try {
    var name = window.prompt('Type your name');
    alert('Your name has ' + name.length + ' letters.');
```

Now if a different error happens, which is admittedly unlikely in this simple example, that error will not be handled. The `throw` statement will forward the error as if we never had this `try...catch...finally` block. It is said that the error will *bubble up*.

Throwing Custom Errors

We can use the `throw` statement to throw our own types of errors. The only recommendation is that our error object also has a `name` and `message` properties to be consistent with the built-in error handling.

```

throw {
  name: 'InvalidColorError',
  message: 'The given color is not a valid color value.'
};
```

Lesson 1, Activity 19: The delete Operator

The delete operator is used to delete properties of objects and elements of arrays. Deleting is different from setting the value to null. The former removes the property or element (i.e, makes it undefined) while the latter keeps the property or element, but sets its value to null. The following example illustrates this.

Code Sample:

AdvancedTechniques/Demos/delete.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<script type="text/javascript">
  var colors = [];
  colors["red"] = "#f00";
  colors["green"] = "#060";
  colors["blue"] = "#00f";
  colors["yellow"] = "#ff0";
  colors["orange"] = "#0ff";

  function deleteItem() {
    delete colors["blue"];
    showArray();
    revealArray();
  }

  function setToNull() {
    colors["blue"] = null;
    showArray();
    revealArray();
  }

  function showArray() {
    var output = document.getElementById("colors");
    var strOutput = "<ol>";
    for (var i in colors) {
      strOutput += "<li style='color:" + colors[i] + ";>" + i + "</li>";
    }
    strOutput += "</ol>";
    output.innerHTML=strOutput;
  }

  function revealArray() {
    var colorsCode = document.getElementById("colors-code");
    var strOutput = "&lt;ol&gt;";
    for (var i in colors) {
      strOutput+="\n\t&lt;li style='color:" + colors[i] + ";&gt;" + i + "&lt;/li&gt;";
    }
    strOutput += "\n&lt;/ol&gt;";
    colorsCode.innerHTML=strOutput;
  }
</script>
<title>Delete vs. Setting to NULL</title>
</head>
<body onload="showArray();revealArray();">
<button id="set-to-null" onclick="setToNull();">Set Blue to Null</button>
<button id="del" onclick="deleteItem();">Delete Blue</button>
<output id="colors"></output>
<pre id="colors-code"></pre>
</body>
</html>
```

Don't worry about the showArray() and revealArray() functions yet. Just note that when you click the **Set Blue to Null** button, the array element stays, but becomes an invalid color: null. When you click the **Delete Blue** button, the array element is removed.