JavaScript
Introduction

Student Guide

ISBN: 0-9908404-4-1

Table of Contents

About this Course

Target Student & Prerequisites

This course is intended for students with a good working knowledge of HTML.

How to Use this Book

This book is designed to be used as part of an instructor led course. It contains various lessons that will guide you through learning the course material. The topics are explained including explanations of logical concepts and examples. Also included are exercises that will reinforce the material discussed. The exercises build on each other.

Included with your book are files that you will need to complete the exercises. There are three folders included for each of the twelve lessons, Demos, Exercises and Solutions. Place these folders on your desktop to make it easy to access the files. You can view them in any browser and the editor of your choice.

In some cases, the Demos files are not exactly the same as the code snippet in the book, as the code snippet is intended to be the code to focus on, and the actual HTML file in the Demos folder is a complete working HTML page.

If for some reason, you are not able to complete each exercise, you can find files in each lesson folder to use as your starting point for the exercises. In addition, there are solution files provided for each exercise. Make sure that you save periodically so that you do not lose your work.

At the end of the book, you will find a glossary of the definitions of terms used in this course. In addition, there is an index provided to allow you to quickly find what you are looking for.

.

Lesson 1: Introduction to JavaScript

In this lesson, you will learn about:

1. Versions of JavaScript
2. Scripting Terminology
3. Language Fundamentals
4. Built in Objects
5. User Input Methods

Versions of JavaScript

JavaScript was created by Brendan Eich at Netscape Navigator in 1995. It was developed to interact with HTML pages. It was originally called Mocha, then LiveScript before being renamed JavaScript.

JavaScript is an interpreted scripting language, rather than a compiled programming language.

ECMAScript is the standardized specification of a few similar client-side scripting languages including JavaScript and Microsoft's implementation, JScript. In 1999, edition 3 of ECMAScript added regular expressions and try/catch statements. In 2009, edition 5 added strict mode. ECMAScript edition 6 introduces significantly new syntax, including Python-like features. ECMAScript edition 6 was published in June 2015, but is not yet fully supported by browsers. This course will focus on edition 5.1.

JavaScript has grown more popular in recent years because of its contribution to rich internet applications and AJAX. This has led to an increase in JavaScript libraries such as jQuery for Document Object Model (DOM) scripting and JavaScript Model-View-Controller (MVC) frameworks such as AngularJS and Backbone.js. A main use of JavaScript is to manipulate the DOM through user events, providing immediate interaction and giving the server a break. JavaScript is also used in HTML5 Application Program Interfaces (APIs).

Each browser has its own engine for interpreting JavaScript, which occasionally leads to browser differences in how JavaScript is interpreted. Several JavaScript libraries provide an abstraction layer to deal with browser differences behind the scenes.

JavaScript isn't only used in web browsers. MongoDB and CouchDB are two databases that use JavaScript as their query language. Node.js is a platform that allows JavaScript beyond the browser, and it is even used to create dynamic HTTP servers.

Creating your first JavaScript page

JavaScript can be added to an HTML page in two different ways. It can be added between **<script>** tags that are inside the HTML page, or in a separate file of only JavaScript. Most applications will use a separate JavaScript file. Using a separate file allows the script to be be re-used and updated efficiently by developers. It also allows the browser to cache the script.

Exercise 1-1 Add a script to your page

Complete the exercise below to add a script to your page using tags and a separate file.

1. In an editor, open Lesson 1/Exercises/exercise1-1.html.
2. Add a set of <script> tags in the <head>.
3. Inside the <script> tags, add document.write("Hello!<br>");
4. Save exercise1-1.html.
5. Open the file in your browser.
6. Open exercise1-1.html in your editor.
7. Move only the JavaScript code between the <script> tags to a separate file.
8. Save the file as exercise1-1.js in the Lesson 1/Exercises/js folder.
9. In the exercise1-1.html file, add a src attribute to the <script> tags referencing the external exercise1-1.js file as a relative path.
10. Save exercise1-1.html.
11. Open it in your browser.
12. It should look the same as in step 5.

Scripting Terminology

Before we begin, we need to cover some terms to assist with the learning process.

Expression

In scripting or programming, an expression produces a value. Any value written literally such as a number or string is an expression.

An expression can assign a value to a variable, or just have a value. JavaScript expressions can evaluate to a number, character string or `true/false`.

Below is an example of an arithmetic expression. It evaluates to `3` without an assignment.

```
1 + 2;
```

Variable

A variable captures and holds values. It can be updated.

Operators

Operators are unary, binary or ternary with one, two or three operands. Operators can be different types including arithmetic, string, logical, assignment and comparison. Most JavaScript operators are binary, requiring an operand on either side of the operator.

In this example, the assignment operator (=) is a binary operator and so is the addition (+) operator.

```
x =1 + 2;
```

The operand is the part of the expression being manipulated by the operators. Above, the numbers `1` and `2` and the variable `x` are the operands.

There are a few unary operators with one operand to the left or right of the operator. The minus sign (-) can be used to subtract one number from another in a binary operation, or to negate a value in a unary operation.

```
x = -3; //x is negative 3
```

JavaScript has one ternary operator that has two operators between three operands. It is a shortcut for a `true/false` test and will be demonstrated later.

Statement

A statement can consist of expressions, variables and operators. A *statement* is a complete instruction that contains an expression, to be executed by the web browser. Statements should have semicolons separating them.

```
x = 1 + 2; //statement

x = 3; //statement

1 + 2; //not a statement
```

Program or Script

A program or script is simply a list of statements that can be executed by a browser.

Language Fundamentals

The below section will introduce you to the JavaScript syntax and concepts.

Declaring Variables

In JavaScript, variables are local or global. Local variables are declared with the keyword `var`. Variables that are set without being declared are global by default. When variables are declared within a function, they are local to that function and unavailable outside the function. JavaScript generally has C-style operators that have hierarchy and associativity.

Below is an example that declares the variable `x`. Declaring a variable is not required in JavaScript, but is good practice.

```
var x;
```

The declared variable `x` can be set later in the code.

```
x = 1 + 2;
```

A variable can be set and used without ever declaring it:

```
y = 10 * 2; //no keyword var
```

Or a variable can be set and declared on one line with the keyword `var`:

```
var y = 10 * 2;
```

Best practice is to declare variables at the top of the JavaScript page in alphabetical order. Note that you use the keyword `var` once to declare the variables.

```
var x, y, z;
x = 10;
y = 20;
z = 10;
```

Assigning or Setting Variables

Variables are assigned or set using the equal sign (=). JavaScript has C-style assignment shortcuts: `+=`, `-=`, `*=`, `/=` and `%=`. Below are examples of the various assignment options.

```
//assign the string "hello" to the variable str
str = "hello";
```

```
//concatenate "world" to the variable
str += " world";
//+= is a shortcut for
str = "hello";
str = str + " world";

x = 5;
x *= 10;
//*= is a shortcut for
x = 5;
x = x * 10;
```

Case Sensitive

JavaScript is case-sensitive. Variables, built-in objects, keywords, functions and methods must use the correct case.

JavaScript convention is to use camelCase to name identifiers (variables and functions). JavaScript's camelCase means putting multiple words together and capitalizing all but the first word:

```
var userName = "";
function calculateTotalProfits(){}
```

Hungarian notation is also popular, where the name of the identifier indicates its type and/or intended use.

```
var intAge = 100; //integer value
var strDept = "IT"; //string value
var boolFullTime = true; //Boolean value
var arrEmployeeList = []; //enumerated array
```

Note that using camelCase is not required, and that Hungarian notation does not set the data type, it only indicates the expected data type. In JavaScript, data types can change in context; there is no way to force an identifier to always be a certain data type.

Spacing and Ending Statements

In JavaScript, a semicolon ends a statement. JavaScript is very flexible, so a missing semicolon will usually not cause an error.

Spacing does not matter between tokens (operators and operands). A statement can continue on the next line if broken up between tokens, but a string cannot be broken onto the next line.

Comments

JavaScript uses two C-style comments, the single-line and multi-line block. Comments can be used for debugging or to document code. Single-line comments are two slashes (`//`) and comment out until the end of the line. Multi-line comments are between `/* */`. A best practice is to use multi-line comments only for author and license information.

Data Types

JavaScript has primary/primitive data type, number, string and Boolean. It has two special types, `null` and undefined. It has two composite types, array and object, that can be made up of primitive values or other composite values. There are special types of objects such as functions, `RegExp` and `Date()`. Since JavaScript has dynamic types, a variable can be of different types through a script. The `typeof` operator can help determine a value's data type.

```
var x = 10; //x is a number
var y = x + 20; //30, adds because of type
var x = "User"; //x is a string
var y = x + 1;//User1, concatenates because of type
```

Numeric Variables

In JavaScript, numeric values can be integers (whole numbers) or floats. Large numbers can be created with "e" followed by the exponent of the number.

```
10e2 //10 times 10 to the power of 2 is 1000
```

Integer calculations are precise up to the number 9 quadrillion. Calculations with floats are not always precise, so they should be treated as an approximation.

The `toPrecision()` method formats a number to a specified length. It takes a number of significant digits as its argument and can be used to obtain a more user-friendly number. The argument is the total number of digits before and after the decimal.

`toFixed()` takes a number and converts it to the specified number of digits after the decimal place.

Arithmetic Operators

The arithmetic operators are multiplication (`*`) division (`/`), addition (`+`) and subtraction (`-`). There is also a modulo operator (`%`) that returns the remainder. Multiplication, division and modulo take precedence over addition and subtraction. As in mathematics, operator precedence can be changed with parentheses.

```
x % y //returns remainder of dividing x by y
```

The plus sign (`+`) is overloaded so it can be used for addition *and* string concatenation. If any of the operands are a string, the plus sign acts as concatenation.

Functions

There are three functions for converting strings to a number data type. They can be used to ensure addition rather than concatenation. Each is briefly explained below.

- `parseInt()` will search a string for an integer at the beginning,

```
parseInt("2 apples"); //returns 2
parseInt("2.5 apples"); //returns 2
```

- `parseFloat()` will search a string for a float at the beginning.

```
parseFloat("2.5 apples"); //returns 2.5
parseFloat("4.5%"); //returns 4.5
parseFloat("$2"); //returns NaN
```

- `Number()` will attempt to convert a value to a number. If it cannot, it returns NaN (not a number).

```
Number("2 apples"); //returns NaN
Number(""); //returns 0
```

When the + (plus) appears before a value, the + converts a string to a number like the `Number()` function, and the − operator negates a value.

```
+"100" //returns 100 as a number
x = -20 //x is negative 20
y = -x //y is negative x, 20
```

JavaScript has three special values that fall under the number data type but don't act like other numbers: `Infinity`, `-Infinity` (negative Infinity) and `NaN` ("not a number"). In many languages, division by zero leads to an error, but in JavaScript it returns `Infinity`, `-Infinity` or `NaN`.

String Variables

Strings are for text or a string of characters. A string can contain numbers. A person's phone number or zip code might contain numbers, but the data type will be a string. Strings must always be delimited by quotes, but it doesn't matter whether these are single or double quotes until they are nested. Because of JavaScript's flexibility, a value can be turned into a string by enclosing it in quotes.

A newline can't be added to a string literally with the return key. Characters such as newline or tab can be created using the C-style backslash (\). The backslash must be used inside quotes. The newline and tab will display in `alert()`, `prompt()` and `confirm()` but not in the browser or console.

```
alert("This is one line.\nThis is another line.");
alert("These are \t tab delimited");
```

The + operator concatenates strings, taking two or more strings and making them into one string. Strings must always be in quotes, but identifiers such as variables, object names and array names must not be.

The string data type has a property and several methods that will be explored later.

Exercise 1-2 Number and string data types

This exercise will allow you to explore the data types you have just reviewed. You can complete these using the JavaScript console as a command line.

1. Using the plus sign, type "3" + 3. What was returned and why?
2. Add 0.4 + 0.2. Was the answer what you expected?
3. Try 0.4 + 0.2 with the toPrecision() method. Set num = 0.4 + 0.2, then use num.toPrecision(1).
4. Set num to 100. What is the difference between num.toPrecision(2) and num.toFixed(2)?
5. Use the modulo for 4 % 2, 4 % 3 and 3 % 4. Is the result what you expected? How could you use this?
6. Look at the difference between the Number() and parseInt() functions by passing the string "3 apples" and then by passing empty string ("").
7. Set x = 12 / 0 and see what x returns. What is the data type? Hint: use typeof x. Now do the same with 0 / 0.
8. Use alert() to print "hello world" with a newline in between the two words. Hint: start with alert("hello world").

Boolean Variables

The two values of Boolean type are true and false. Note that "true" or "false" in quotes are strings, not Boolean. The purpose of comparison and logical operators is to return a Boolean value.

Comparison operators are greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), equal to (==), not equal to (`!=`), strictly identical to (===) and not strictly identical to (`!==`).

In an equal to (==) or not equal to (!=) test, operands are tested by value. In an identity test (=== or !==), operands are tested by value and type. Be sure to use == or === for comparison rather than a single equal sign as that is for assignment.

```
4 == "4"; //true, same value
4 === "4"; //false, not same type
```

JavaScript has three logical operators, and (`&&`), or (`||`) and not (`!`). They are often used with comparison operators. All sides must be true for `&&` to return `true`. Only one side must be true for `||` to return `true`. Use the logical not (`!`) operator to reverse logic.

Operator precedence determines in which order operators are evaluated. Comparison operators are before logical operators in the order of operations. The logical and (`&&`) takes precedence over the logical or (`||`), and the logical not (`!`) takes precedence over both. Parentheses are not necessary, but can make code easier to read. Comparison and logical operators are often used in `if` statements.

`&&` and `||` can be used for short-circuit evaluation.

```
//assigns "anonymous" if fName not defined
//fName must be declared for this to work
     var fName;
     user = fName || "anonymous"
```

Other Basic Data Types

JavaScript recognizes three more data types: object, function and undefined.

- Object - Arrays, the value null, literal objects and built-in objects
- Function - JavaScript functions are a data type
- Undefined - Any variable that has been declared but not assigned a value

```
var x;

typeof x; //undefined
```

In contrast to `undefined`, `null` is an assignment value. It is a literal representing an empty value.

```
var y = null;
```

Built-in Objects

JavaScript has a few built-in objects including Array, Math and Date(). Arrays will be covered in more detail later. See below for details on Match and Date.

Math

The Math object is used for calculations that go beyond basic arithmetic. Properties include `PI` (3.141592653589793), and methods include `round()`, `ceil()`, `floor()`, `random()`, `min()` and `max()`.

- `round()` rounds to the closest integer. If the fractional portion is .5 or larger it rounds up to the next integer.
- `ceil()` rounds to the next whole number
- `floor()` always rounds down
- `random()` does not take any arguments, and returns a number between 0 and 1, exclusive of 1
- `Min()` returns the smallest of the numbers in a set of numbers
- `Max()` returns the largest of the numbers in a set of numbers

```
//demo1-1.html
//document.write writes to the web page
  document.write(Math.ceil(4.2));//5
  document.write(Math.floor(4.9));//4
  document.write(Math.round(4.5));//5
//random number 0 – 1 exclusive
//refresh page for new random number
  document.write(Math.random());
```

Date

JavaScript's Date object is based on unix epoch time: January 1, 1970.

`Date()` is a constructor object and the date must be created with the `new` operator. The `Date()` constructor can be empty, take a string, or take one to seven numbers separated by commas. An empty `Date()` constructor returns the current timestamp. To use one of the Date methods, create a Date instance first.

```
//demo1-2.html

var today = new Date();

document.write(today);
```

If you pass one number, the Date() constructor reads it as milliseconds since unix epoch time. Passing two to seven arguments becomes year, month, date, hours, minutes, seconds, milliseconds to create a date. When two or more arguments are passed, the first argument becomes a year, rather than number of millisecond since epoch time.

Useful Date methods include `getFullYear()`, `getDate()`, `getMonth()`, `getDay()`, `getHours()`,`getTime()`, `getMinutes()` and `getSeconds()`.

- `getFullYear()`returns the 4 digit year portion of the date
- `getDate()`returns the day portion of the date – value from 1 to 31
- `getMonth()` returns an index number 0 – 11 representing the month with 0 being January
- `getDay()` returns an index number 0 – 6 representing the day of the week with 0 being Sunday
- `getTime()` method displays the date by number of milliseconds since epoch time. You can use this to find out how many days, weeks or months are between two or more dates.

```
//demo1-3.html

document.write(today.getTime());
```

JavaScript also has the Date methods `setFullYear()`, `setDate()`, `setMonth()`, `setHours()`, `setMinutes()` and `setSeconds()`. `setMonth()` takes a zero-based index number.

```
//demo1-4.html

var birthday = new Date();

birthday.setMonth(11);

document.write(birthday);

//timestamp for December, current date/year
```

The `setTime()` method takes a number of milliseconds since epoch time. It takes negative numbers for dates before epoch time.

```
//demo1-5.html

var prevDate = new Date();

prevDate.setTime(-1000000000000)

document.write(prevDate);

//Sun Apr 24 1938 17:13:20 GMT-0500 (CDT)
```

User input methods

The window object has three methods, `alert()`, `confirm()` and `prompt()`. The last two can be used to get user input. Because they are methods of the window object, they can't be styled; the browser determines how they look.

`alert()` takes one argument, so if there are multiple pieces of data, they must be concatenated before being passed as an argument.

`confirm()` takes one argument and returns a Boolean data type.

`prompt()` takes two arguments, a question and default answer, and returns a string data type.

A user could also input data into a web page by filling out a form, or interact with it through a user event such as clicking. Both of these concepts will be covered in more detail later.

```
//demo1-6.html
alert("Hello!");
prompt("How old are you?", "");
confirm("Are you sure?");
```

Exercise 1-3 The built-in `Date()` object

With this exercise, you will determine the number of days between two dates.

1. Using your editor, create a file named `exercise1-3.js`.
2. Save it in `Lesson 1/Exercises/js`.
3. Use `new Date()` to get the current date.
4. Ask the user for their next birthday.
5. Find out how many days until their birthday. Hint: use `.getTime()` to convert both dates to a common format, then calculate how many milliseconds in a day.
6. Reference the script in `Lesson 1/Exercises/exercise1-3.html` and test.

Lesson 2: JavaScript Conditions

In this lesson, you will learn about:

1 Conditional Constructs

Conditional Constructs

JavaScript supports two types of C-style conditional constructs, `if/else if/else`, and `switch`. The ternary operator, `?:`, provides an `if/else` shortcut.

Any value in JavaScript has a Boolean value that is "truthy" or "falsy." The Boolean `false`, the number `0` and the empty string `""` are falsy, and are equal to each other when compared with the == equality operator but not the === identity operator. `null` and `undefined` are falsy, but are only equal to themselves. `NaN` is falsy, and not equal to anything, including itself.

The `if` Construct

The `if` takes a conditional expression in parentheses and any number of statements in a block designated by curly braces. The statements run if the condition is true.

```
if (condition) {

statement1;

statement2;

}
```

The condition can be any Boolean expression. Any number and combination of comparison and logical operators can be used to construct the condition as long as it evaluates to `true` or `false`. Variables used in a conditional expression must be of string or numeric data type.

If there is only one statement to run if the condition is true, the curly braces can be left off, but it is better practice to use them for clarity.

```
if (condition)

statement;
```

The conditional construct could be on one line.

```
if (condition) { statement; }
```

Testing Multiple Conditions

The `if` construct alone executes statements when the condition is `true`, and does nothing when the condition is `false`. One `else` statement can be added to determine what statements to execute when the `if` condition evaluates to `false`.

```
if (condition) {
statement if condition is true;
} else {
statement if condition is false;
}
```

When more choices are needed, any number of `else if` constructs can be added. The `else if` executes only when the previous `if` or `else if` conditions are not true. An `else` will execute when none of the previous conditions are true. Only one of the following statements will run.

```
if (condition) {
     statement;
} else if (condition) {
     statement;
} else if (condition) {
     statement;
} else {
     statement;
}
```

Exercise 2-1 Using the if construct

This exercise will use the `if` construct to validate input.

1. Using your editor, create a file called `exercise2-1.js`.
2. Save it in `Lesson 2/Exercises/js`.
3. Ask the user for their hours worked and hourly wage. Hint: use `prompt()` twice.
4. Assume employees work 1-80 hours, with an hourly wage from 1 – 100 dollars and that if the user enters a number outside of the range, it has been mistyped. Use an `if` to test that the user entered the correct range, and ask them again if they did not use the correct range.
5. Ask the user if they are a full-time employee. Hint: use a `confirm().` It returns `true` or `false`.
6. Calculate how much they have earned that week. If they are not full-time and did work over 40 hours, they get paid time and a half for overtime.
7. Use `document.write(`*`variable`*`);` to print to the page.
8. Add a reference to the JavaScript file to `Lesson 2/Exercises/exercise2-1.html` and test.

The `switch` Construct

The `switch` construct is used to select among several blocks of code to be executed. Unlike an `if/else if` construct, the `switch` expression (this can be a variable) is evaluated only once.

The value of the expression in the `switch` construct is compared against values for each case. Each `case` value ends with a colon. If the value matches a `case`, a block of code associated with the case is executed. The `default` statement is not required, but is used to specify code to run if there is no match. It is similar to `else`.

The keyword `break;` is used to break out of a switch block. It is used when a `case` match is found and there is no more need for testing. `break;` is not needed for the last statement in a `switch`.

```
switch (expression) {
      case value1:
         statement;
      break;
     case value2:
statement if previous case false;
      break;
      default:
statement if previous case false;
         //break; not necessary at the end
 }
```

Without `break`, different cases will "fall through" and execute the same code. Sometimes this can be used to your advantage.

```
switch (expression) {
          case value1:
          case value2:
             statement to run for both cases;
          break;
          default:
statement to run if previous false;
}


switch (day) {
             case "Saturday":
             case "Sunday":
                msg = "It’s a weekend!";
          break;
          default:
             msg = "It’s a weekday.";
}
```

Exercise 2-2 Using the switch

In this exercise, you will use a `switch` construct.

1. Using your editor, create a file called `exercise2-2.js.`
2. Save the file in `Lesson 2/Exercises/js`.
3. Ask the user for a letter of the alphabet.
4. Use a `switch` to determine whether the letter is a consonant or vowel.
5. Use `document.write()` to print out whether it is a consonant or vowel.
6. Add a reference to the JavaScript file to `Lesson 2/Exercises/exercise2-2.html` and test.

The Conditional (Ternary) Operator

The conditional operator (`?:`) is the only operator in JavaScript that takes three operands (ternary). It is used as a shortcut for the `if/else` construct. The first operand must evaluate to `true` or `false`. The expressions have values of any type.

```
condition ? expression1 : expression2;
```

If *condition* is true, the ternary operator returns *expression1*, the value after the `?`. Otherwise it returns *expression2*, the value after the `:`. The return result can be output, assigned to a variable or printed out.

```
var result = condition ? expression1 : expression2;

document.write(condition ? expression1 : expression2);
```

To execute more than one operation, separate with a comma.

```
condition ? operation1, operation2 : operation3,
operation4;
```

Exercise 2-3 Using the ternary operator

Use the ternary operator as an `if/else` shortcut.

1. Using your editor, create a file called `exercise2-3.js`
2. Save the file in `Lesson 2/Exercises/js`.
3. Ask the user for a price without a dollar sign.
4. If the price is over $100, a 10% discount will be applied.
5. Use the ternary operator to determine whether the discount is applied or not.
6. Use `document.write()` to print out the total price, whether the discount is applied or not.
7. Add a reference to the JavaScript file to `Lesson 2/Exercises/exercise2-3.html` and test.

Lesson 3: JavaScript Loops

In this lesson, you will learn about:

1 JavaScript Loops

2 JavaScript Labels and Branch Statements

JavaScript Loops

JavaScript has three C-style loops, `while`, `do…while` and `for`. Additionally, JavaScript has a `for…in` loop that iterates over the enumerable properties of an object.

Loops are used to test for a particular condition as many times as needed, or repeat code a certain number of times. They are frequently used to iterate over an array or object, or validate form elements.

The while Loop

The `while` loop is a pretest loop, testing a condition before the loop runs. The block of code in a `while` loop is executed as long as the condition is `true`. When the conditional expression returns `false`, the loop stops.

```
while (condition) {
    statements to run if condition is true;
}
```

As with the `if` construct, curly braces can be left off if there is only one statement, but it is better to use them.

If the conditional expression never becomes `false`, the loop can never end, resulting in an *infinite loop*.

```
//demo3-1.html
var x = 0;
while (x < 10) {
    document.write(x);
    document.write("<br>")
    x++; //increment by 1 to end at 10
}
```

When the condition is `false` to start with, the `while` loop doesn't run at all.

```
var x = 10;

while (x < 10) {

document.write(x);

x++;

}
```

Exercise 3-1 Using a `while` loop

Improve your code with `while` loops.

1. Using your editor, create a file named `exercise3-1.js`
2. Save the file in `Lesson 3/Exercises/js`.
3. Assume employees work 1-80 hours, with an hourly wage from 1 – 100 dollars and that if the user enters a number outside of the range, it has been mistyped. Use a `while` to test that the user entered the correct range, and ask them again if they did not use the correct range.
4. Calculate and print their gross pay.
5. Which one is better, an `if` or a `while` and why?
6. Add a reference to the JavaScript file to `Lesson 3/Exercises/exercise3-1.html` and test.

The do…while Loop

The do…while loop is similar to the while loop, except that it is a post-test loop that tests a condition after the loop runs. The block of code in a do…while loop is always executed at least once. Once the conditional expression returns false after the first run, the loop stops.

```
do {

    statements to run if condition is true;

} while (condition)
```

As with the if and while constructs, curly braces can be left off if there is only one statement.

Like the while loop, if the conditional expression never becomes false, the loop is infinite.

```
//demo3-2.html

var x = 0;

do {

  document.write(x);

  x++; //after first run, increment by 1 to reach 10

} while (x < 10);
```

The do…while loop always runs at least once.

```
//demo3-3.html

var x = 10;

//runs at once, before testing

do {

  document.write(x);

  x++;

} while (x < 10);
```

Exercise 3-2 Using a `do…while` loop

Improve your code with `do…while` loops.

1. Open `exercise3-1.js`
2. Save the file as `exercise3-2.js` in `Lesson 3/Exercises/js`.
3. Replace the `while` with a `do…while.`
4. How is the code different from the while loop?
5. Add a reference to the JavaScript file to `Lesson 3/Exercises/exercise3-2.html` and test.

The for Loop

The `for` loop is most similar to the `while` loop. If a condition is initially `true`, it runs a block of statements until the condition is `false`. The `for` loop has three parts in parentheses, separated by two semicolons. The first part is a declaration of a counter variable. The second part is the conditional expression determining whether the loop should run. The last one resets the initial variable, usually incrementing or decrementing the counter. There is no semicolon after the last part of a `for` loop.

```
for (initialize; condition; increment/decrement)

      {

              statements;

      }
```

The curly braces can also be left off the `for` loop if there is only one statement.

Like the other C-style loops, neglecting to alter the initial variable results in an infinite loop.

```
//demo3-4.html

for (var x = 0; x < 10; x++) {

document.write(x)

}
```

Since it is more compact than the `while` or `do…while` loop, the `for` loop is popular for enumerated arrays.

The for...in Loop

The `for…in` loop is used to iterate through an array or object and will be explored in more detail in a later section. The keywords are `for` and `in`. Below, *property* is a variable representing the property of an object, and *object* is the name of the object or array.

The `for…in` loop does not use a numbered counter.

```
for (property in object) {
     document.write(property + object[property]);
      }
```

As with the other constructs, the curly braces can be left off if there is only one statement.

JavaScript Labels and Branch Statements

In JavaScript, `break;` and `continue;` can be used as *branch statements.* They can be used with or without labels. They determine what code the program should execute next. A *label* is any identifier with a colon after it. The branch statements `break;` and `continue;` can be used in a loop to transfer program control.

Using a break; statement

The `break;` statement terminates the loop (or `switch`), jumping to the next statement outside of the loop or `switch`.

```
//demo3-5.html
for (x = 1; x <= 10; x++) {
   if (x % 5 == 0) {
      break; //run until divisible by 5
      }
      document.write(x);
}
```

In a nested loop, the `break;` terminates the current loop.

```
//demo3-6.html
i = 0;
//runs 3 times, breaking at number 4
while (i < 3){
for (x = 1; x <= 10; x++) {
   if (x % 5 == 0) {
      break;
   }
document.write(x);
}
i++;
}
```

`break;` can pass program control to a *label*. In this example, program control is passed to the outer `while` loop when the `for` loop reaches the number 4 because of the `outerLoop:` label created by the developer.

```
//demo3-7.html
i = 0;
//runs once, breaking at number 4
outerLoop: while (i < 3){
for (x = 1; x <= 10; x++) {
   if (x % 5 == 0) {
   break outerLoop; //label
   }
   document.write(x);
 }
 i++;
}
```

Using the continue Statement

The `continue;` statement terminates execution of the loop at the current iteration and continues the loop at the next iteration. Once the `continue;` statement is executed, statements below it are skipped. Below, if numbers are not evenly divisible by 5, jump to the next iteration of the loop. Only numbers divisible by 5 are printed.

```
//demo3-8.html
for (x = 0; x <= 10; x++) {
if (x % 5 != 0) {
continue;
}
document.write(x); //if divisible by 5
}
```

Program control can also be passed to a label with the `continue;` statement.

Exercise 3-3 Using branch statements

Use a branch statement to manipulate program control.

1. Using your editor, create a file named `exercise3-3.js.`
2. Save the file in `Lesson 3/Exercises/js`.
3. Generate a random number between 1 and 10. Hint: use `Math.random()`.
4. Give the user three chances to guess the number. If they guess, give them a "congratulations" message and tell them how many guesses it took.
5. What should you do to the loop if the user guesses before the three chances are up?
6. Give the user the opportunity to quit the guessing game by typing "Q" (case-insensitive).
7. If they don’t guess the correct number in three tries, print the random number.
8. Add a reference to the JavaScript file to `Lesson 3/Exercises/exercise3-3.html` and test.

Lesson 4: Primitive Data Types

In this lesson, you will learn about:

1 The String Data Type

2 The Number Data Type

The String Data Type

A string is a sequence of characters delimited by double or single quotes, used for text representation. A string literal is a primitive data type, like a numeric or Boolean value. Some popular operations on strings include checking their length, concatenating, searching for a character or substring, or extracting a substring.

```
str = "hello world";
```

The primitive type is converted to a string object when necessary, so the variable containing the string literal can use string properties and methods. `new String()` is useful for converting another data type to string, so that you can use string properties and methods. If string properties and methods are used on another data type such as a number, an error is returned.

The `new String()` constructor creates a string object.

```
var x = 100;

var str = new String(x)
```

String Properties

The `length` property of a string returns the length of the string.

```
//demo4-1.html

var email = "username@emaildomain.com";

var len = email.length;

document.write(len);//returns 24
```

A string can be empty, in which case it has a length of `0`.

```
var str = "";

var strlen = str.length; //returns 0
```

String Methods

Below are explanations for some of the string methods that can be used to manipulate string data types.

charAT()

The `charAt()` method takes an index number and returns the character at that index. The index number is an integer between 0 and 1 less than the length of the string. If the index provided is longer than the length of the string, an empty string is returned.

```
//demo4-2.html

var email = "username@emaildomain.com";

var character = email.charAt(0); //returns "u"

document.write(character)
```

indexOf()

The `indexOf()` method takes the character(s) to search for, and an optional starting index number. It returns an index number from 0 to 1 less than the length of the string. If the characters are not found in the string, it returns -1. The search is case sensitive.

```
//demo4-3.html

var index = email.indexOf("a");

document.write(index); //returns 5

index = email.indexOf("a", 6); //11

index = email.indexOf("a", 18); //-1

index = email.indexOf("z"); //-1
```

lastIndexOf()

The `lastIndexOf()` method returns the index number of the last occurrence of the character in the string. The search begins at the end of the string. The search is case sensitive.

```
//demo4-4.html

var lindex = email.lastIndexOf("a"); //17

lindex = email.lastIndexOf("a", 16); //11

document.write(lindex);
```

split()

The `split()` method splits a string into an array of substrings according to a delimiter or separator.

```
//demo4-5.html
var strDays = "Mon, Tues, Wed, Thurs, Fri";
//split by comma and space
var arrDays = strDays.split(", ");
//Mon, Tues, Wed, Thurs, Fri
document.write(arrDays);
```

slice()

The `slice()` method extracts part of a string and returns it. The method takes an index number at which to begin the slice, and an index at which to end the slice as an optional second argument. `slice()` extracts up to but not including the second argument. If the second argument is omitted, `slice()` extracts to the end of the string. If either argument is a negative number, it is treated as string.length – number.

```
//demo4-6.html

//"emaildomain.com"

var strSlice = email.slice(9);

strSlice = email.slice(9, 20); //"emaildomain"
```

trim()

The `trim()` method returns a string with whitespace removed from both ends, without affecting the value of the original string.

```
//demo4-7.html

var user = "     UserName    ";

var trimUser = user.trim();

document.write(trimUser); //"UserName"

document.write(user); //"     UserName    "
```

toLowerCase()

There is a string method for converting a string to lowercase and returning the result, `toLowerCase()`.

```
//demo4-8.html

var user = "UserName";

var lcUser = user.toLowerCase();

//username

document.write(lcUser);
```

toUpperCase()

The string method `toUpperCase()` converts a string to uppercase and returns the result.

```
//demo4-9.html

Var ucUser = user.toUpperCase();

//USERNAME

document.write(ucUser);
```

Exercise 4-1 Using string manipulation methods

Use string methods and logic to validate the data being input.

1. Using your editor, create a file named `exercise4-1.js`.

2. Save the file in `Lesson 4/Exercises/js`.

3. Ask the user for their email address, and use `if…else/if…else` to test.

4. Recall that if you are searching for a letter, it is case-sensitive, but email addresses are not case-sensitive. It is useful to convert the email address to lowercase if you are going to test it against a string.

5. Make sure their entry is not blank but first use the string’s `trim()` method to strip out whitespace.

6. Perform some data validation on the email address.

   a. An email address can’t be valid if it is less than 6 characters long. Test that it is at least 6 characters long.
   b. The email address should have
      i. An @ sign
      ii. No more than one @ sign
      iii. The @ sign in a position other than the first or last character
      iv. No spaces

7. Create an error message for an invalid email address.

8. If the email address is valid, use `document.write()` to print that message. Extract the domain name and print that with the message.

9. Add a reference to the JavaScript file to `Lesson 4/Exercises/exercise4-1.html` and test. For the email validator, test with an invalid email address to make sure you get the errors.

The Number Data Type

We saw some features of the number data type in an earlier lesson. The Number object has a few more properties and functions.

Number Properties

The static property `Number.MAX_VALUE` represents the largest positive number possible in JavaScript. Any number above this is represented as `Infinity`. -`Number.MAX_VALUE` (with a minus sign before it) is the negative number furthest from zero. Any number below it is –`Infinity`.

The smallest positive number closest to zero without being zero is Number`.MIN_VALUE`. Any positive number smaller than this is converted to zero. -`Number.MIN_VALUE` is the smallest negative number closest to zero. Any negative number larger than this is converted to zero.

You must use these properties on the keyword `Number`, not a variable holding a number.

```
//demo4-10.html

document.write(Number.MAX_VALUE); //correct use

document.write(Number.MIN_VALUE); //correct use

x = 10;

x.MAX_VALUE; //incorrect use, returns undefined
```

Number Methods

`toPrecision()` and `toFixed()` are methods of the number object. These were discussed in Lesson 1.

Global Functions Used with Numbers

As seen in Lesson 1, `parseInt()` and `parseFloat()` are global functions that parse a string and return a number. JavaScript has global functions that take an argument of any data type and test whether it is a number.

The global function `isNaN()` tests whether an argument is not a number.

```
isNaN("abc"): //true

isNaN(7): //false
```

The global function `isFinite()` tests whether an argument is finite, and it must be a number to be finite.

```
isFinite("abc"): //false

isFinite(7): //true
```

But `isFinite()` is not a true opposite of `isNaN()`.

```
isNaN(12/0): //false, it is a number

isFinite(12/0); //value is Infinity, so false
```

The only accurate way to determine whether a value is a number is to reverse the logic of `isNaN()`.

```
!isNaN(7); //true
```

Lesson 5: Compound Data Types

In this lesson, you will learn about:

1 Arrays

2 Objects

3 Arrays of Objects

4 Using JSON Syntax to Store Data

Arrays

Numbers, strings and Booleans are primitive data types – a program's basic building blocks. Arrays and objects are composite or compound data types, made up of primitive data types and/or other compound data types.

An array is an enumerated list or collection of data that can hold several values under one name. In JavaScript, an array is a special type of object. In an array, each piece of data has a corresponding numeric index used to access the value. Arrays have one property and several manipulation methods.

Storing Data in Arrays

An array can be created with the literal syntax using square braces (`[]`).

```
var fruits = ["banana", "orange", "apple"];

typeof fruits;//object
```

Arrays are dynamic, so elements can be added or removed.

```
var fruits = []; //declares the array fruits

fruits[0] = "banana";

fruits[1] = "orange";

fruits[2] = "apple";
```

Array Properties

An array's `length` property calculates the number of elements at that point.

```
document.write(fruits.length); //3
```

The `length` property is obtained by adding one to the highest index, not by counting the number of values.

```
var fruits = [];

//non-sequential array with 2 elements

fruits[0] = "banana";

fruits[2] = "apple";
```

```
document.write(fruits.length); //3
```

A specific element can be accessed with an index number. Index numbers range from 0 to array `length` -1.

```
fruits[0]; //first element

fruits[fruits.length-1]; //last element
```

Looping Over an Array

Using index numbers, an array can be looped over with the `while`, `do…while` or `for` loop. These loops work well if the array is sequential. In the example below, if indexes 0, 1 or 2 are missing, `undefined` is printed in their place.

```
for (x = 0; x < 3; x++) {

   document.write(fruits[x]);

}
```

If you don’t know the number of elements in an Array, you can use the length property to determine the number.

```
for (x = 0; x < fruits.length; x++) {
   document.write(fruits[x]);
}
```

The `length` property calculates dynamically, every time it is called. If you are sure the array won’t change, it is more efficient to use the `length` property once, rather than each time the loop runs.

```
var len = fruits.length;
    for (x = 0; x < len; x++) {
        document.write(fruits[x]);
    }
```

The `for…in` loops over every element in the array, even if the array is non-sequential. The keywords are `for` and `in`, with the array name to the right of the `in` and a variable representing the index number to the left.

```
for (x in fruits) {
   //x is index number
   document.write(fruits[x]);
}
```

Exercise 5-1 Holding data in an array

Use arrays to hold data.

1. Using your editor, create a file named `exercise5-1.js.`
2. Save the file in `Lesson 5/Exercises/js`.
3. Make one array that contains the days of the week.
4. *Optional*: make another array that contains the names of the months.
5. Use the `Date()` constructor to get today's date. Get the month, date, year, and day of the week. The date is `getDate()` and the day of the week is `getDay()`.
6. Use the arrays you made to print out the day of the week using words instead of numbers for the month and day of the week. If you did not make a month array, print the correct month number.
7. Add a reference to the file to `Lesson 5/Exercises/exercise5-1.html`
8. Save and test in the browser.

Array Mutator Methods

JavaScript has a few mutator methods that permanently modify an array. The methods return different things, but they all either add to, remove from, or change the order of the array. Below are several methods that you may find useful.

pop()

`pop()` removes the last element from an array and returns it. It does not take any arguments.

```
//demo5-1.html

var lastItem = fruits.pop();

document.write(lastItem); //apple

document.write(fruits); //banana,orange
```

push()

`push()` adds one or more elements to the end of the array and returns the length of the array. It takes one or more arguments.

```
//demo5-2.html

fruits.push("blueberry");

fruits.push("cherry", "grape");
```

reverse()

`reverse()` reverses the order of the elements. It doesn’t take any arguments.

```
//demo5-3.html

//grape,cherry,blueberry,orange,banana

fruits.reverse();
```

shift()

`shift()` removes the first element from an array and returns it. It doesn't take any arguments. The index numbers of the existing elements change.

```
//demo5-4.html

var firstItem = fruits.shift()

document.write(firstItem);//grape

//cherry,blueberry,orange,banana

document.write(fruits);
```

sort()

`sort()` doesn't need any arguments to sort lexically, but we have an issue when we want to sort numerically. Pass a function to the `sort()` method that compares each element as part of a pair and swaps them if they are in the wrong order:

```
//demo5-5.html

var nums = [100, 1, 15, 20, 10, 200];

nums.sort(); //1, 10, 100, 15, 20, 200

//1, 10, 15, 20, 100, 200

nums.sort(function(a, b){return a - b});
```

splice()

`splice()` can add and remove elements from an array. The first two arguments are required. The first argument is the index to start the splice at, the second argument is the number of items to remove.

```
//demo5-6.html

//start index 1, remove next 2

fruits = ["banana", "orange", "apple",
"blueberry", "cherry"];

var rmItems = fruits.splice(1, 2);

document.write(rmItems);//orange,apple

document.write(fruits);//banana,blueberry,cherry
```

```
//demo5-7.html

//second argument higher than remaining number

//of elements – removes rest of elements

var fruits = ["banana", "orange", "blueberry",
"cherry"];
fruits.sort();

document.write(fruits + "<br>");

fruits.splice(1, fruits.length);

document.write(fruits);//banana
```

The third optional argument is a list of one or more items to be inserted into the array at the first argument index number.

```
//demo5-8.html

//start at index 1, add 2 elements, remove

//next 2 elements

var fruits = ["banana", "orange", "apple",
"blueberry", "cherry"];

var rmItems = fruits.splice(1, 2);

fruits.splice(1, 0, "kiwi", "lemon"); // remove
zero, add two

document.write(fruits); //banana, kiwi, lemon,
blueberry, cherry
```

unshift()

`unshift()` adds one or more elements to the front of an array and returns the new length of the array. As with `shift()`, the index numbers of the existing elements change.

```
//demo5-9.html

var fruits = ["banana", "orange", "apple"];

var addItems = fruits.unshift("plum", "pear");

document.write(addItems);//5

document.write(fruits); //plum, pear, banana,
orange, apple
```

Array Accessor Methods

Accessor methods do not modify the array. They return a representation of the original array. Below are some of the common accessor methods used for arrays.

concat()

`concat()` returns a new array consisting of the array on which it is called and the arrays or values provided as arguments.

```
//demo5-10.html

var fruits = ["plum", "pear", "banana", "kiwi",
"lemon"];
var alpha = ["a", "b", "c"];

var nums = [2, 3];

var concatArr = alpha.concat(fruits, nums);

document.write(alpha);//["a", "b", "c"];

document.write(concatArr); //a,b,c,plum,pear,
banana,kiwi,lemon,2,3
```

join()

`join()` takes a delimiter or separator as an optional argument and returns all elements of an array joined as a string. Contrast with the string method `split()`, which splits a string into an array by a delimiter/ separator.

```
//demo5-11.html

fruits = ["plum", "pear", "banana", "kiwi",
"lemon"];

var str1 = fruits.join("|"); //the fruits array
is unchanged

document.write(str1)//plum|pear|banana|kiwi|lemon
```

The delimiter can be more than one character.

```
//demo5-12.html

var fruits = ["plum", "pear", "banana", "kiwi",
"lemon"];

var str1 = fruits.join("|"); //the fruits array
is unchanged

document.write(str1);

var str2 = fruits.join(" > ");

document.write(str2);//plum > pear > banana >
kiwi > lemon
```

When the argument is omitted, the array is joined with commas.

```
//demo5-13.html

var fruits = ["plum", "pear", "banana", "kiwi",
"lemon"];

var str1 = fruits.join("|"); //the fruits array
is unchanged

document.write(str1);

var str2 = fruits.join(" > ");

document.write(str2);

var str3 = fruits.join();//plum,pear,banana,kiwi,
lemon
```

If the argument is empty quotes (""), the elements are joined with no characters between them.

```
//demo5-14.html

var fruits = ["plum", "pear", "banana", "kiwi",
"lemon"];

var str4 = fruits.join("");

document.write(str4);//plumpearbananakiwilemon
```

slice()

`slice()` returns a portion of an array as a new array. Don't confuse it with the mutator method `splice()`, which removes items, adds items, and permanently changes the original array.

The first argument in `slice()` is the index at which to start and an array is extracted up to but not including the second argument. If the second argument is left off, it is extracted to the end of the array. As with strings, `slice()` can take negative numbers.

```
//demo5-15.html

var fruits = ["plum", "pear", "banana", "kiwi",
"lemon"];

var someFruit = fruits.slice(1,2);

document.write(someFruit); //pear

document.write(fruits);//plum,pear,banana,kiwi,
lemon
```

Exercise 5-2 Searching through an array

Use an array, a loop, a `break;` statement and string methods to create a client side search.

1. Using your editor, create a file named `exercise5-2.js.`
2. Save the file in `Lesson 5/Exercises/js`.
3. Make an array of city names. Make sure some of them are multiple words, like New York City.
4. Prompt the user for the name of a city to search for in the array.
5. Loop through the cities array and if you find the city they typed in:
   a. Set a variable called `found` to `true`.
   b. `break;` out of the loop. The loop doesn't need to keep searching.
6. Outside the loop, if the `found` variable is true, use `document.write()` to print that you found the city and print the name.
7. Otherwise, use `document.write()` to print that you did not find the city they typed. Print out what they typed.
8. Add a reference to the script in `Lesson 5/Exercises/exercise5-2.html` and test.
9. Notice that so far, the search expects the full name of the city to be typed in. We want to make it accept partial words (as long as the city name starts with the letters) like "Atl" for Atlanta.

10. In `Lesson 5/Exercises/js/exercise5-2.js`, use the `substring()` method to test if the city in the array starts with the search string's letters. *Hint:* `if (`*`citiesarray`*`[i].substring(0,` *`searchstring`*`.length) ==` *`searchstring`*`)`

11. When printing out the city name you want to print the full name from the array, so assign the searched string that you are going to print out to the full name from the array. *Hint:* *`searchstring`*` = `*`citiesarray`*`[i];`

12. You might have noticed that the search is case sensitive. We want to make it case insensitive. We can use the `toUpperCase()` method to make the whole first word uppercase, but what if the city is multiple words? We can `split()` the city by the space, loop through it, use the `toUpperCase()` method on it and `join()` it back as a string with a space delimiter. *Hint:* *`array`*` = `*`searchstring`*`.split(" ");` before the loop, and *`searchstring`*` = `*`array`*`.join(" ");` after the loop.

13. We don't want the whole word uppercase, just the first letter of each word. We will use string methods to get the first character and make that uppercase. *Hint:* *`array`*`[i].charAt(0).toUpperCase()`, concatenate it with the rest of the word, and reassign to *`array`*`[i]`. The rest of the word can be grabbed with *`array`*`[i].slice(1)`.

14. Actually, they might type in everything uppercase, so we should convert everything but the first letter to lowercase. *Hint: Use* *`array`*`[i].slice(1).toLowerCase()` *instead of just* *`array`*`[i].slice(1)`.

15. Save, and test in the browser.

Multidimensional Arrays

A multidimensional array is an array of arrays.

```
var people = [

["Bob", "Smith"],

["John", "Doe"]

]
```

A two-dimension array's elements are accessed with two sets of square braces where the first set is the "row" number and the second is the "column" number.

```
people[0][0];//"Bob"

people[1][1];//"Doe"
```

A multidimensional array can also be created piece by piece.

```
var people = [[]];

people[0][0] = "Bob";
```

Objects

An object stores multiple values in one variable. The difference between an array and object is that while an array's elements are accessed with numeric indexes, an object's elements are accessed with string keys. Some languages call this an associative array.

Storing Data in an Object

A single object can be created with object literal syntax.

```
var person = {

firstName: "John",

lastName: "Doe",

eyeColor: "green"

}
```

`firstName`, `lastName` and `eyeColor` are properties of the `person` object. Object properties do not need quotes around them unless they include non-alphanumeric characters.

Accessing Data in an Object

There are two ways to access an object's elements, *dot notation* or *bracket notation*. In dot notation, the property name does not have quotes, but in brace notation it does.

```
person.firstName;//John

person["lastName"];//Doe
```

These two notations can also be used to set object elements.

```
person.eyeColor = "blue";

person["age"] = 21;
```

Looping Through Data in an Object

The `for…in` loop is used to loop through objects. The `while`, `do…while` and `for` loop don't work with objects since they don't have numeric indexes. The object name goes to the right of the keyword `in` and a variable for the key goes to the left.

```
for (p in person) {

   document.write(p + ": " + person[p])

 }
```

Creating an Object Method

A custom object method can be created with a function. The keyword `this` refers to the current object.

```
//demo5-16.html
var person = {
firstName: "John",
lastName: "Doe",
fullName: function(){
return this.firstName + " " + this.lastName
}
}
document.write(person.fullName());  //John Doe
```

Creating an Object Constructor

The previous example created a single object. To create a blueprint that can be used to create many objects of one type, use an object constructor function. Convention is that the constructor's first letter is uppercase.

```
//demo5-17.html

function Person (fName, lName) {

   this.firstName = fName;

   this.lastName = lName;

}
```

```
var emp1 = new Person("John", "Doe");

var emp2 = new Person("Robert", "Smith");

//use property name

document.write(emp1.firstName);//John

document.write(emp2["firstName"]);//Robert

document.write(typeof emp1);//object
```

Arrays of Objects

In addition to arrays of arrays, you can create arrays of objects.

```
//demo5-18.html

//array of objects

var employees = [

   {firstName: "John", lastName: "Doe"},

   {firstName: "Robert", lastName: "Smith"}

]

document.write(employees[0]["firstName"]);//John

document.write(employees[0].firstName);//John

//object contains array as value of one property

var employee1 = {

    firstName: "John",

    lastName: "Doe",

    skills: ["HTML", "CSS", "JavaScript"]

}

document.write(employee1["skills"][0]);//HTML

document.write(employee1.skills[0]);//HTML
```

Exercise 5-3 Working with objects

Create an array of objects and access each property.

1. Using your editor, create a file named `exercise5-3.js.`
2. Save the file in `Lesson 5/Exercises/js`.
3. Make a named array of objects. The objects should have properties like firstName, lastName, email and phone.
4. Loop through the array of objects and access all values.
5. Add a reference to the script to `Lesson 5/Exercises/exercise5-3.html`.
6. Save and test in the browser.

Using JSON Syntax to Store Data

JSON stands for **J**ava**S**cript **O**bject **N**otation. It was specified by Douglas Crockford as an alternative to XML for storing and exchanging data. While the X in AJAX stands for XML, it is increasingly being replaced by JSON because of its familiar, lightweight syntax. JSON is language-independent.

While JSON uses JavaScript syntax, it is text only. No variables are allowed, and properties must be quoted. The text-only data format can be read by any programming language.

```
{
"employees" : [
{"firstName": "John", "lastName": "Doe"},
{"firstName": "Robert", "lastName": "Smith"}
              ]
}
```

Below we will explain the two methods that are used for parsing JSON.

JSON.parse()

`JSON.parse()` parses a string containing JSON syntax and returns the value as JSON.

```
//demo5-19.html
//concatenation to break up string
var str = '{ "employees" : [' +
'{"firstName": "John", "lastName": "Doe"},' +
'{"firstName": "Robert", "lastName": "Smith"}' +
     ']}'
//convert string to JSON object
var jObj = JSON.parse(str);
document.write(jObj);//JSON object
```

JSON.stringify()

`JSON.stringify()` takes a JSON object as an argument and converts it to a string.

```
            document.write(JSON.stringify(jObj));
```

Lesson 6: Functions

In this lesson, you will learn about:

1 Declaring and Invoking a Named Function

2 Creating an Anonymous Function Expression

3 Scope

4 Function Arguments

Declaring and Invoking a Named Function

A JavaScript function is a procedure containing a set of statements that performs a task and can be re-used. The code is defined once and can be used many times. The function is declared in the scope in which it is called.

A function definition (or declaration or statement) is created with the keyword `function`, a name for the function, a list of *parameters* for the function in parentheses, and curly braces around the statements that are to run when the function is called. Parameters are initialized to arguments passed in parentheses when the function is invoked.

```
//takes one argument, prints it squared

function square (n) {

document.write(n * n);

}
```

If a function has no parameters, the parentheses are empty. With parameters, a function can be run many times with different arguments to produce different results.

The function can be called above or below the line that defines the function, as long as they are in the same scope. When the function is invoked, it takes *arguments*. The function is not executed until it is called. Later we will see how to let the user call functions with events. The `()` operator invokes the function.

```
square(10);//100
```

Parameters in a function are names listed in the function definition. Arguments are the real values passed to the function. They can be constants or passed with variables. Variables pass *by value*. The original variable has not changed, only its value has been passed to the function.

```
      x = 10;

square(x);//x * x is 100

document.write (x);//x is still 10
```

Returning Values

Returning a value makes the function more flexible. The result can be printed or assigned to a variable. The value is returned back to the calling program.

```
//demo6-1.html

function square (n) {

return n * n;

}

var x = 10;

//do something with return value

document.write(square(x));//100

//assign return result to variable

y = square(x);

document.write(y);//100
```

The return statement makes the function stop executing. That means there can only be one return statement executed per function. Nothing after the return statement will be executed. It is common to use more than one return statement in a function, with a condition that determines which return statement will be executed.

If you must return multiple values, return an array or object.

Exercise 6-1 Writing a function

Create a re-usable piece of code and invoke it.

1. Using your editor, create a file called `exercise6-1.js.`

2. Save the file in `Lesson 6/Exercises/js`.

3. You will make a Fahrenheit to Celsius converter. There are two form fields already in the HTML file, and some code to make one field change another as the numbers are typed. There are also comments with instructions in the code.

    a. The formula for Fahrenheit is `c = (f - 32) * 5 / 9`
    b. The formula for Celsius to Fahrenheit is `f = c * 9 / 5 + 32`
    c. Round the result using `Math.round()` to get a whole number.
    d. Use `document.getElementById("celsius").value` to get and set the value of the Celsius form field (the explanation is in a later lesson).
    e. Use `document.getElementById("fahrenheit").value` to get and set the value of the Fahrenheit form field (the explanation is in a later lesson).
    f. *Optional:* The string argument you pass will be case sensitive. Consider making it case insensitive by converting the argument `toLowerCase()` or `toUpperCase()`.

4. Call the function twice where indicated by comments in the code, passing either "Celsius" or "Fahrenheit."

5. There is already HTML code and a reference to the script file in `Lesson6/Exercises/exercise6-1.html`. The script is pulled in at the bottom of the page so it can use the DOM. We will learn more about the DOM in a later lesson.

Creating an Anonymous Function Expression

An anonymous function or function expression has similar syntax to a named function statement. Some programming languages call it a lambda. It has the keyword `function`, a parameter list and a block, but no function name. The expression can be stored in a variable. Since the variable stores the function expression, the function is called with the variable.

```
var square = function (n) {

document.write(n * n);

}

square (10); //100
```

Invoking a named or anonymous function might look similar, but there is a difference in how the browser loads them into the execution context. Named function declarations load before any code is executed, so a line of code that invokes a function can appear on a line above the function declaration. Anonymous function expressions load when the JavaScript gets to that line of code. An anonymous function must be invoked after the line on which it is defined.

Scope

JavaScript functions themselves have scope, and their variables have scope.

Local and Global Variables

Scope refers to the variables and functions you have access to. JavaScript has variable scope, local and global.

Local Variables

Variables declared within a function are local to the function. JavaScript variables are declared with the keyword `var`, so variables declared with `var` are *local* to their scope, and variables created without the keyword `var` are *global*. Local variables declared inside a function can only be accessed inside the function.

```
function square (n) {

var total = n * n;

}

square(10);
```

```
document.write(total);//undefined
```

Local variables are created when the function starts, and deleted from memory when the function finishes. Because of scope, variables with the same name can be used in different scopes, for example different functions. This is significant because scope makes the function re-usable by ensuring variables don't conflict with other functions.

```
function square (n) {

var total = n * n;

//local variable available only

//inside function

document.write(total);

}

square(10);//100
```

Global Variables

A variable set without the keyword `var` is global, even when defined inside a function. The global variable is available outside the function. A variable created outside a function is global, regardless of whether it is declared with `var`. Any script or function on a web page can access a global variable. Global variables are deleted from memory when the web page is closed.

```
//global when defined outside function

var x = 10;

function square (n) {

//global variable conflicts with others

total = n * n;

}

square(x);

document.write(total);//100
```

Function Scope

JavaScript also has function scope. Functions can be nested, and an inner function can only be called within its parent function.

```
function outer () {
document.write("outer function");
function inner () {
document.write("inner function");
}
//parent must call inner function
inner();
}
//outer function must be called too
outer();
```

Function Arguments

Functions don't specify data types for parameters or arguments, or check the number of arguments that are passed.

Setting Default Arguments

If a function is called with missing arguments, they are set to `undefined`. Arguments can be tested for existence and set to default values if they are missing. If they are passed when the function is called, the default values are overwritten. Being able to set arguments to default values allows you to create optional parameters.

Generally, you must pass `undefined` as the optional argument for the default value to be set.

```
//demo6-2.html
function fullName (fName, mName, lName) {
if (!mName) {
   document.write(fName, " ", lName);
} else {
```

```
document.write(fName, " ", mName, " ", lName);
        }
}
fullName("John", undefined, "Public"); //John Public
```

If the optional parameter is the last one, it can be left off.

```
//demo6-3.html

function fullName (fName, lName, mName) {

if (!mName) {

   document.write(fName, " ", lName);

   } else {

document.write(fName, " ", mName, " ",  lName);

           }

}

fullName("John", "Public");//John Public

//middle name is the last argument

fullName("John", "Public", "Q");//John Q Public
```

Exercise 6-2 Creating a function with optional arguments

Set default values for optional arguments in a function.

1. Using your editor, create a file named `exercise6-2.js.`
2. Save the file in `Lesson 6/Exercises/js`.
3. Create a function that takes three arguments: first name, middle name and last name. Middle name is optional and if they do not provide a middle name, you will display it as "NMN" for "no middle name."
4. Test the function with hard-coded arguments.
5. Reference the file in `Lesson 6/Exercises/exercise6-2.html` and test.

The `arguments` and `this` Properties

When a function is invoked, it has access to its own arguments, its `arguments` object, and its `this` property. `arguments` is an array-like object containing all arguments passed when the function was invoked. `this` is a reference to the object that the function is a method of. It is very helpful when creating objects or working with event handlers. We will see it in use in later lessons.

Looping through the `arguments` object is another way to make arguments optional. In fact, the parameter list in the function definition could be left blank and all arguments could be optional.

```
//demo6-4.html

//no parameter list specified

function sum () {

var x;

var sum = 0;

//keyword arguments is an array

var argLen = arguments.length;

for (x = 0; x < argLen; x++){

     sum += arguments[x];

    }

return sum;

}

document.write(sum(23, 4, 100));//127
```

Exercise 6-3 Using a function's `arguments` object

Create a function that uses the `arguments` object to get the average sum of arguments, and the highest number in the arguments.

1. Using your editor, create a file named `exercise6-3.js.`
2. Save the file in `Lesson 6/Exercises/js`.
3. Create a function that takes any number of numeric arguments and calculates the average of all numbers passed. *Hint:* Use the `arguments` property, it will get all arguments. Get the sum and use it to calculate the average.
4. Add a reference to `Lesson 6/Exercises/exercise6-3.html` and test the function with hard-coded numbers.
5. Create another function that determines the maximum number that was entered. *Hint:* Set the variable for the maximum number to 0 initially, and compare it against each number. If a number in the arguments list is higher than that variable, it is the new maximum.
6. Since all numbers in JavaScript must be above negative Infinity (`-Infinity`), change the initial value of the maximum number variable to that.
7. Test the function with hard-coded numbers.

Lesson 7: Regular Expressions

In this lesson, you will learn about:

1. Constructing a Regular Expression
2. Applying Regular Expression Methods
3. Using Regular Expressions in String Methods
4. Backreferences

Constructing a Regular Expression

Regular expressions are patterns used to match character combinations in strings. Several programming, scripting and markup languages (including XML and HTML5) can implement regular expressions, and the syntax is the same. A JavaScript regular expression is an object. In JavaScript, a regular expression is very useful for validating form data.

A regular expression can be constructed one of two ways. A *regular expression literal* is a pattern between forward slashes. The literal regular expression is compiled when the script is loaded. The regular expression literal provides better performance, if the regular expression is not going to change throughout the script.

```
var re = /abc/
```

The second way is to use the *constructor function* of the RegExp object. The constructor function provides runtime compilation of a regular expression. Use this if the regular expression will be changing, or the pattern is coming from another source, for example user input.

```
var re = RegExp("abc");
```

Creating a Pattern

A pattern, whether a literal or RegExp object, can be as simple as a string of literal characters.

```
//look for the substring the

//matches the, theater, soothe

var re = /the/;
```

A regular expression pattern can be made more powerful by putting together special metacharacters.   Metacharacters are characters with a special meaning.

```
/.the/; //dot is wildcard for a single character
```

The pattern can take quantifiers that further modify the metacharacters.

```
/.+the/; //at least one character before the
```

Backslashes are used to either escape characters and make them literal, or to create metacharacters.

```
/\.html/; //backslash makes dot literal
```

Special Characters

A backslash before a special character indicates that it is interpreted literally. There are also several special characters created within a regular expression by putting a backslash in front of a normal character.

`\d` represents a digit, and `\D` represents a non-digit.

```
//matches 12345-1234

//matches 12345 1234

/\d\d\d\d\d\D\d\d\d\d/;
```

`\s` represents a whitespace character including space, tab and newline. `\S` represents a non-whitespace character.

`\w` represents numbers, letters and underscore. `\W` represents a non-alphanumeric character.

As seen before, the dot is a wildcard for any single character, except the newline character.

Quantifiers

Quantifiers are characters that go after special characters and indicate how many times that type of character appears. The quantifier can indicate a minimum and/or maximum.

```
/\d{5}/;//5 digits

/\d{5,9}/;//5 to 9 digits

/\d{5,}/;//minimum of 5 digits

/\d{,9}/;//maximum of 9 digits
```

There are shortcuts for some quantifiers.

+ is short for one or more and can be written {1,}

`*` is short for zero or more and can be written {0,}

`?` is short for zero or one and can be written {0,1}

```
var re = /\d{5}-?\d{4}/;//has 0 or 1 hyphen
```

Anchors

Anchors are used to indicate the beginning and end of a phrase.

`^` indicates the beginning of a phrase.

```
//starts "the", matches "theater", "the"
/^the/;
```

`$` is used for end of a phrase.

```
//ends "the", matches "soothe", "the"
/the$/;


//matches "the"
/^the$/
```

The phrase boundaries can help quantifiers.

```
//matches 5 digits in a row
//could have more than 5 characters
/\d{5}/;


//only 5 digits
/^\d$/;
```

Ranges

Square braces (`[]`) are used to create a list or range. One of the characters must appear in the string.

`[]` is used for a character set or character class. Matches any one character or character type in braces.

```
/[abc]/;//matches a, b or c

/[abc]+/;//matches 1 or more of a, b, and/or c
```

A range of characters can be specified with a hyphen inside braces.

```
/[0-9]/;//digit, same as \d

/[a-z]/;//case-sensitive, lowercase letter
```

Ranges can also be listed in any order.

```
/[a-zA-Z0-9]/;//letters and numbers

/[a-zA-Z0-9_]/;//same as \w
```

Characters that have a special meaning must be escaped with backslash if they are literal.

```
//letters, numbers, punctuation

/[a-zA-Z0-9!\?\.]/;
```

The braces can take special escaped characters.

```
/[\w\s!\?\.]/;
```

The character set can be negated with `^` *inside square brackets*, not to be confused with the starting string character *outside square brackets*. One character that is not in the list must appear in the string.

```
//any single character but lowercase a, b or c

/[^abc]/;

/[^0-9]/; //any non-digit, same as \D

/[^a-zA-Z0-9]/; //any non-alphanumeric

//starts with non-digit, same as /^\D/

/^[^0-9]/;
```

Modifiers

Modifiers go outside the forward slashes. The `i` flag makes the list case-insensitive and the `g` creates a global search.

```
/[a-z]/i; //case-insensitive

/[a-zA-Z]/;
```

The global modifier is helpful for search and replace, which we will see later.

```
/[a-zA-Z]/g; //globally searches

/[a-zA-Z]/; //finds once
```

Modifiers can be nested.

```
/[a-z]/ig; //case-insensitive global search
```

The pipe (`|`) character is used as an "or" between whole strings to match.

```
//matches bet, get, met or net

//does not match let

/bet|get|met|net/;

/[bgmn]et/; //same matches as above
```

Using Parentheses to Match Substrings

Parentheses create a group. The group can be quantified or used in a *backreference*. A backreference matches the same text as was previously matched in a capturing group, and is explained in more detail later.

```
// s is optional, starts http or https

//forward slashes must be escaped

/^https?:\/\//;

//the whole group of characters is optional

/^(https?:\/\/)?/;
```

Applying Regular Expression Methods

There are two regular expression methods that take strings as arguments for comparison.

Regular Expression Method `exec()`

`exec()` is a regular expression method that searches for a match in a string and returns an array or `null`.

```
//demo7-1.html

var phNum = "abc123";

var re = /^\d{10}$/;

//use ! to test if returns null

if (!re.exec(phNum)){

   document.write("bad phone number");

    } else {

   document.write("good phone number");

}
```

If the match succeeds, `exec()` returns an array and updates properties of the regular expression object. Element zero of the *returned array* contains the entire match. Elements 1 and higher contain submatches within the match. The `index` property of the result returns the index of the first match, and the `input` property returns the original string.

Properties of the *regular expression* include `lastIndex`, the point at which to start the next match. This is zero unless there is a global flag. `ignoreCase` and `global` indicate whether the case-insensitive (`i`) and global (`g`) flags are set. `source` is the text rendition of the pattern.

```
//demo7-2.html

 var re = /quick\s(brown).+(jumps)/ig;

 var str = "The Quick Brown Fox Jumps Over The Lazy
 Dog";

 var result = re.exec(str);

 //whole match

 document.write(result[0], "<br>");

 //Parenthesized match

 document.write(result[1], "<br>");

 //Parenthesized match 2

 document.write(result[2], "<br>");

 //Index number of match

 document.write(result.index, "<br>");

 //Original string

 document.write(result.input, "<br>");

 // Start next match, 25 (0 if global flag not used)

 document.write(re.lastIndex, "<br>");

 //Case-insensitive?

 document.write(re.ignoreCase, "<br>");

 //Global search?

 document.write(re.global, "<br>");

 //Pattern

 document.write(re.source, "<br>");
```

Regular Expression Method `test()`

`test()` is a *regular expression method* that tests for a match in a string and returns `true` or `false`. It is faster than `exec()`.

```
if (!re.test(phNum)){

document.write("bad phone number");

}
```

Using Regular Expressions in String Methods

There are four string methods that take regular expressions as arguments for comparison.

String Method `match()`

`match()` is a string method that searches a string for a match. It returns an array or `null`.

```
//demo7-3.html
var phNum = "(123) 456 - 7890";
var re = /\d+/g;
if (arrMatch = phNum.match(re)){
   document.write("good phone number<br>");
   var phone = "area code " + arrMatch[0] +
   ", prefix " + arrMatch[1] +
   ", last four digits " + arrMatch[2];
    document.write(phone, "<br>");
} else {
document.write("bad phone number<br>");
}
```

String Method `search()`

search() tests for a match in a string and returns the zero-based index of the match or -1 if no match is found.

```
//demo7-4.html

var phNum = "1234567890";

var re = /^\d{10}$/;

if (phNum.search(re) != -1){

   document.write("good phone number");

} else {

   document.write ("bad phone number");

}
```

String Method `replace()`

`replace()` returns a new string with all matches of a pattern replaced by a replacement string. The pattern can be a string or regular expression.

```
//demo7-5.html

var re = /\D/g;

var phNum = "(123) 456-7890";

//replace non-digits with space

cleanedPhone = phNum.replace(re, " ");

//123 456 7890

document.write(cleanedPhone, "<br>");

//original format

document.write(phNum, "<br>");

//strip out non-digits

cleanedPhone = phNum.replace(re, "");
```

```
//1234567890

document.write(cleanedPhone, "<br>");
```

String Method `split()`

`split()` is a string method seen earlier that uses a string literal or regular expression to break a string into an array of substrings.

```
//demo7-6.html

//global modifier g not necessary

var re = /\D/;

var phNum = "(123) 456-7890";

var cleanedPhone = phNum.split(re);

for (i in cleanedPhone) {

document.write(cleanedPhone[i], "<br>");

}
```

`split()` takes a separator character as an argument and an optional second argument, the number of splits to return. The default behavior is to split the string into all possible substrings.

```
var cleanedPhone = phNum.split(re, 2);
```

Exercise 7-1 Using regular expressions

Use regular expressions to qualify data using quantifiers.

1. Using your editor, create a file named `exercise7-1.js.`
2. Save the file in `Lesson 7/Exercises/js`.
3. Ask the user for a U.S. zip code or Canadian postal code.
4. Using one pattern:
    a. The U.S. zip code can be five numbers, nine numbers or five numbers, a hyphen or space and four more digits. It can't be five digits with a hyphen or space without the extra four digits. It can't be more than ten characters total, and it should not allow any characters other than numbers, hyphen or space.

    b. The Canadian postal code has the pattern letter number letter space number letter number. The space is optional. The letters are case-insensitive. *Optional:* The postal code **can't** start with the upper or lower case letters d, f, i, o, q, u, w or z.
5. Use a regular expression method or string method to validate the zip and postal code.
6. Add a reference to the file in `Lesson7/Exercises/exercise7-1.html` and test.

Backreferences

Backreferences can be used inside a pattern or outside a pattern. In either case, the backreferences match a substring in a group in the regular expression, delimited by parentheses. The backreference matches the *actual substring found, not the pattern*.

Groups are created with parentheses, and backreferences refer by number to the group number by position.

Inside a pattern, \1 is a backreference to the first parenthesized group, \2 is a backreference to the second parenthesized group, and so on.

```
//ensures that separation character is repeated

//matches 123-45-6789

//matches 123456789

//doesn't match 12345-6789

//doesn’t match 123-45 6789

/^\d{3}([\-\.\s])?\d{2}\1\d{4}$/;
```

Backreferences outside a pattern are created with `$1` through `$9`, and can only hold nine references to groups. If more references are needed, use array indexes. Backreferences outside a pattern are accessed with `RexExp.$1`, `RegExp.$2` but `$1`, `$2` can be used within the `replace()` method.

```
//demo7-7.html

var reSSN = /^(\d{3})(\d{2})(\d{4})$/;

var strSSN = "123456789";

//hardcode hyphens between group 1, 2 and 3

var cleanSSN = strSSN.replace(reSSN, "$1-$2-$3");

document.write(cleanSSN);//123-45-6789
```

Exercise 7-2 Using advanced regular expressions

Use backreferences in regular expressions.

1. Using your editor, create a file named `exercise7-2.js.`
2. Save the file in `Lesson 7/Exercises/js`.
3. Prompt for a phone number.
4. Strip out non-digits using `replace().`
5. Print the output to see what you have.
6. Use *backreferences outside of the pattern* (`$1`, `$2` etc.) and `replace()` to format the phone number to have parentheses around the area code, a space between the area code and the first three digits, and a hyphen after the first three digits and the next four: (123) 123-1234.
7. Add a reference to the file to `Lesson7/Exercises/exercise7-2.html` and test.
8. *Optional:* Create an HTML tag validator in the same file. Ask for an HTML tag and validate. Note that the browser will interpret any tag following certain rules as valid; it does not have to be a standard tag. The tag should start with a left angle brace (<) with at least one letter. The other characters inside the tag can be anything (Hint: anything but a close angle bracket >). Use a *backreference inside the pattern* (`\1`) to make sure the closing tag matches. Use the pipe (`|`) to also allow a self –closing tag that might or might not have a space. You will probably need parentheses around the closing tag choices. `<a href="index.html">link</a>`, `<input type="text"/>` and `<input type="text" />` (with space) are examples of valid html tags.

Lesson 8: The Browser Object Model

In this lesson, you will be learn about:

1 The Browser Object Model

2 Using a BOM Method to Execute Code

The Browser Object Model

The browser object model (BOM) is all objects exposed by a web browser. The top level of the hierarchy is the `window` object. The `window` object has methods, and properties that are also objects. Because window is the parent object of all other BOM objects, the keyword `window` is often left off in scripts.

Useful window properties include `history`, `location`, `navigator` and `screen`. These will be demonstrated briefly. `console` is another `window` property with its own methods, such as `log()`.

The most commonly used `window` property is `document`. It has its own properties and methods, which will be explored in greater detail in the next lesson.

Window methods include `addEventListener()`, which will be discussed in the lesson on JavaScript Events. `alert()`, `confirm()` and `prompt()` are three window methods that were introduced earlier. Other window methods are `open()` and `close()` for popup windows, and `setTimeout()/clearTimeout()` and `setInterval()/clearInterval()`.

As with the window properties, the keyword `window` is often left off when its methods are used. *This only works for `window` properties and methods.*

```
//both do the same thing

window.prompt("What is your name?", "")

//shorter

prompt("What is your name?", "")
```

The `history` Object

`window.history` is read-only and contains the browser's session history, pages visited in the currently loaded tab. It has its own properties and methods that can be used to get to other pages in the session history. To test the `window.history`, you must go to different URLs in the same tab. Remember that the `window` object can be dropped in your code.

To move backward or forward, as if the back or forward button was pressed:

```
history.back()

history.forward()
```

To move to a specific point in history:

```
history.go(-1);//equivalent of back button

history.go(-2);//go back two

history.go(0);//refresh current page

history.go(1);//equivalent of forward button
```

To find out how many pages are in the session history, use the *read-only* `history.length` property. Some browsers start the history length at 0 while others start it at 1. The maximum number that can be stored in `history.length` is 50.

```
var x = history.length;

document.write(x);
```

The `location` Object

`window.location` is read-only, but a string URL can be assigned to it, changing the URL.

```
alert(location);//displays current URL

//redirects to new location using property

location.href = "http://www.mysite.com";

//redirects to new location using method

location.assign("http://www.mysite.com");

//shortcut

location = "http://www.mysite.com";
```

The location object has methods that can redirect the user to a new page. As seen above, `location.href` and `location.assign()` both redirect. The difference between `location.assign()` and `location.replace()` is that `location.replace()` redirects and removes the current URL from the browser's history so that it is not possible to use the back button to go back to the original page.

```
//redirects, replacing current URL in history

location.replace("http://www.othersite.com");
```

A variety of strings can be assigned to the `href` property of the `location` object, the `assign()` method or the `location` shortcut.

```
//absolute URL

location.href = "http://www.othersite.com";

//relative URL

location.href = "about.html";

//anchor URL

location.href = "#top";
```

`location.reload()` reloads the current page, as if the user had clicked the browser's "reload" button. The page is reloaded from the cache, but it can be set to reload from the server by passing `true`.

```
//reloads page from cache

location.reload();

//reloads page from the server

location.reload(true);
```

`location.hash` can get and set the hash portion of a URL.

```
location = "http://www.mysite.com";

//jumps to anchor #top

location.hash = "top";

//alerts "#top"

alert(location.hash);
```

`location.host` gets or sets the hostname and port number portion of a URL. It can be an IP address. If it is used as a setter, *it only sets the host portion of the URL without changing any other parts*.

```
location = https://www.mysite.com/about.html#top;

alert(location.host);//www.mysite.com

//sets URL to

//https://www.yoursite.com/about.html#top

location.host = www.yoursite.com
```

In contrast, `location.hostname` only gets or sets the hostname without the port.

The read-only `location.origin` property returns the protocol, domain name or IP address, and any port number.

```
location = https://www.mysite.com/about.html#top;

//alerts https://www.mysite.com/about.html#top

alert(location.origin);
```

`location.pathname` gets or sets the path portion of the URL.

```
alert(location.pathname);//about.html
```

`location.port` gets or sets the port portion of a URL, and `location.protocol` gets or sets the protocol.

`location.search` gets or sets the querystring portion of a URL, including the question mark (?). Querystrings are used to pass parameters to a URL, and JavaScript can get those parameters.

The `screen` Object

`window.screen` has several read-only properties.

```
//demo8-1.html

//pixel width of user's screen

document.write(screen.width, "<br>");
```

```
//pixel height of user's screen

document.write(screen.height, "<br>");

//pixel width of user's screen minus taskbar

document.write(screen.availWidth, "<br>");

//pixel height of user's screen minus taskbar

document.write(screen.availHeight, "<br>");

//24 - 32 bit color resolution

document.write(screen.colorDepth, "<br>");

//same as colorDepth on modern computers

document.write(screen.pixelDepth, "<br>");
```

The `console` Object

The `window.console` gives access to the browser's debugging console. It has a few methods. We have been using the `document.write()` method. It takes a list of arguments separated by a comma, or one argument that is concatenated.

`document.write()` automatically puts spaces between arguments separated with a comma. If values are concatenated, a space must be built into the string. The same is true for `alert()`, since it only takes one argument and therefore multiple values must be concatenated.

```
fname = "John";

//"Hello John"

//console log adds space between arguments

console.log("Hello", fname);

//must add a space to the string

alert("Hello" + " " + fname);
```

The `window` Object's `open()` and `close()` Methods

`window.open()` opens a popup window – another instance of a browser window. The method takes four optional arguments. `close()` closes the popup window.

The first argument is a URL to open as a popup. If you do not specify a URL, you can provide content from your main page.

```
//opens help.html in a new tab

open("help.html");
```

The second argument is a name for the popup. The name is used as a target, for example for links to open in the new popup window.

```
//if target = "popup" is used in a link,

//it will open in this window

open("help.html", "popup");
```

The third argument is a list of specifications. `height` and `width` specify the size of the popup, and `left` and `top` specify the position.

```
//all four arguments are optional

//200px wide, positioned 300px to the right

//specification list is in one set of quotes

open("help.html", "", "width=200, left=300");
```

The fourth argument is `true` or `false` (default) and determines whether the new window replaces the current window in the history list.

The return value of the `open()` method is a reference to the newly created window.

```
//demo8-2.html

//doesn't open a particular URL

newWin = open("", "", "width = 200, height = 200, left
= 300");

//uses newWin to put together a page

//this will write to the popup window

newWin.document.write("hello!");
```

Using BOM Methods to Execute Code

JavaScript has two methods of the window object that can be used to execute code at specified times.

setTimeout()/clearTimeout()

`window.setTimeout()` executes code, often a function, *after a certain delay*. It takes the code snippet or *reference to a function* as its first argument, and the delay in milliseconds as a second argument.

```
//demo8-3.html

function showAlert(){

alert("Surprise!");

}

//pass a reference to the function

//capture the timeout number

//runs once, after two seconds

timeoutId = setTimeout(showAlert, 2000);
```

`setTimeout()` returns the number of the timeout. The first time the timeout runs, it returns 1, the second time 2, etc. The number can be captured and passed to `clearTimeout()`.

```
clearTimeout(timeoutID);
```

setInterval()/clearInterval()

`window.setInterval()` executes code *repeatedly, with a delay between each call*. It takes the code snippet or reference to a function as its first argument, and the delay in milliseconds as a second argument.

```
//demo8-4.html

function showAlert(){

alert("Surprise!");

}

//pass a reference to the function
```

```
//capture the interval number

//runs repeatedly, every two seconds

intervalId = setInterval(showAlert, 2000);
```

`setInterval()` returns the number of the interval. The first time `setInterval()` runs, it returns 1, the second time 2, etc. The number can be captured and passed to `clearInterval()`.

```
clearInterval(intervalID);
```

Exercise 8-1 Use window methods to create animation

Use `setInterval()` to make a page update dynamically.

1. In your editor, navigate to the `Lesson 8/Exercises`. Open the file `exercise8-1.html.` You will notice the text "<p id="content"> The Quick Brown Fox Jumps Over the Lazy Dog. </p>". When your program is finished, you should no longer see this text.
2. In the browser, open the the file `exercise8-1.html` located in the `Lesson 8/Solutions` folder. This will give you an idea of what the Exercise should look like when complete. There is a typewriter effect that prints out letters to the browser. `setInterval()` makes the letters print out one by one.
3. In your editor, open the file `Lesson 8/Exercises/js/exercise8-1.js.`
4. Make variables that contain values for:
   a. The words you want to print out.
   b. A counter, set to `0` (zero).
   c. A variable representing the HTML element you want to print to.
   d. The `setInterval()` function that takes a reference to the function you are going to create and one argument, and a speed in milliseconds as a second argument. In the Solutions file, the name of the function is *typewriter* and the speed is 50. Capture the `setInterval()` as a variable so that you can `clearInterval()`. *Hint: var intervalID = setInterval(functionReference, speedNum); and later, clearInterval(intervalID);*
5. Create your function. Inside:
   a. Get the variable of words you want to print and use a string method on it to get one character at a time. Print this to the element that should display the words. *Hint: str.substr(0, counter) where str is your variable of words to print, and counter will increment by one in the function.*
   b. Increment the counter by one.
   c. If the counter is greater than the length of the words string, calling the function.
6. Save the file to `Lesson 8/Exercises/exercise8-1.html` and test.

Lesson 9: The Document Object Model

In this lesson, you will learn about:

1. The Document Object Model API
2. Working with DOM Nodes
3. Manipulating CSS with the DOM

The Document Object Model API

The Document Object Model (DOM) is an Application Programming Interface (API) for HTML, XML and SVG documents. The DOM stores the document in memory as a tree representing node relationships. This defines the way the document can be accessed and traversed in order to manipulate the content, structure and styling.

In the DOM, everything on the web page is a node. The web page itself is a document node. Each HTML element is an element node. Every HTML attribute is an attribute node. Text inside HTML elements is a text node.

Nodes have properties, methods and event handlers. The DOM is not part of the JavaScript language, but it can be accessed with JavaScript and other languages such as Java.

The document in this case is the web page. One HTML document can be rendered in the browser window, or as the HTML source. The DOM provides a third, object-oriented way to represent, store and manipulate content with a programming or scripting language like JavaScript.

The `document` Object

An HTML page is stored as nodes in the DOM, and the `document` object is the root node. It has several frequently used methods for accessing elements on the web page. This is the first step in accessing elements in order to manipulate the structure, content or styling. These methods are so commonly used that DOM libraries such as jQuery have made shortcuts for them. There are also methods for re-structuring elements.

`document.write()` writes a string of text to the document. `document.write()` can take multiple arguments.

```
var fName = "John";

//multiple arguments, including space

document.write("Hello", " ", fName);

//two strings a variable concatenated into one

document.write("Hello" + " " + fName);
```

The `document` Object Access Methods

`document.getElementById()` takes a unique ID as an argument and returns a reference to the element by ID. The ID name is a case-sensitive string. As with all methods, the `getElementById()` method must use the correct case. Note that it will not work as getElementByID().

```
//case-sensitive method and id name

var div1 = document.getElementById("d1");
```

```
<div id="d1"></div>
```

`document.getElementsByName()` returns a live collection of nodes matching the name. `name` is an HTML attribute that unlike `id`, does not have to be unique.

```
var checkboxes = getElementsByName();

<input type="checkbox" name="skills" value="HTML">

<input type="checkbox" name="skills" value="DOM">
```

`document.getElementsByTagName()` returns a live collection of HTML elements matching the tag.

```
//demo9-1.html

//returns a collection

var allParas = document.getElementsByTagName("p");

var num = allParas.length;

alert(num + " paragraphs");
```

`document.getElementsByClassName()` returns an HTML collection of nodes matching the class.

```
//elements that have both col and main class

document.getElementsByClassName("col main");
```

`getElementsByTagName()` and `getElementsByClassName()` can be either called on the `document` object and search the entire document, or on an element, and search descendants of the element.

```
//elements with the class "content",

//descendent of element with id "container"

var el = document.getElementById("container");

el.getElementsByClassName("content");
```

Using Selectors to Access Elements

JavaScript has two methods that take a *selector*. A selector *selects* elements in the DOM by tag name, class name, id name or attribute. Like `getElementsByTagName()` and `getElementsByClassName()`, The methods can be used on the `document` object or an element.

`document.querySelector()` takes a CSS selector and returns the first match.

```
//first paragraph

document.querySelector("p");

//first div that is descendant of p

document.querySelector("p div");

//first div that is direct child of p

document.querySelector("p > div");

//div or paragraph, whichever is found first

document.querySelector("p, div");

//first class "content"

document.querySelector(".content");

//id "container"

document.querySelector("#container");

//first form field with type attribute "text"

document.querySelector("input[type = text]");
```

`document.querySelectorAll()` takes a CSS selector and returns a *NodeList* of all matches.

```
//list of paragraphs

document.querySelectorAll("p");

//divs that are descendants of p

document.querySelectorAll("p div");

//divs that are a direct child of a p

document.querySelectorAll("p > div");

//all divs and paragraphs

document.querySelectorAll("p, div");

//class "content"

document.querySelectorAll(".content");

//id "container" – there should be only one!

document.querySelectorAll("#container");

//form fields whose type attribute is "text"

document.querySelectorAll("input[type=text]");
```

Exercise 9-1 Use DOM and window methods to create animation

Use DOM methods and `setInterval()` to make a page update dynamically.

1. In your browser, open the file `exercise9-1.html` located in `Lesson 9/Solutions`. You will notice that it shows you how many days, hours, minutes and seconds until a date in the future. Inside the file, there is code and CSS displaying the clock.

2. In your editor, open `exercise9-1.js` located in `Lesson 9/Exercises/js`. You will see a `getTimeRemaining()` function that calculates the difference between today and another date, and divides the milliseconds since Unix epoch time into days, hours, minutes and seconds.

3. Add a function called `setClock()` that takes a parameter (the Solutions call it `later`). Inside, set variables for the element on the HTML page holding the days (class="days"), the element on the HTML page holding the hours, the element on the HTML page holding the minutes and the element on the HTML page holding the seconds.

4. *Inside the `setClock()` function*, create a function called `changeClock()` with no parameters.

5. *Inside `changeClock()`:*

   a. Run the `getTimeRemaining()` function on the parameter passed to the outer function, *`setClock()`*, and capture it with a variable such as `time`. (Remember that `changeClock()` itself has no parameters.)

   b. Write the days, hours, minutes and seconds variables to their corresponding HTML elements. For hours, minutes and seconds, you might want to format it with string functions to have a leading zero when it gets down to single digits. This will add a zero at the beginning, but also make sure it is only two digits total: `('0' + seconds).slice(-2);`

   c. Once the `time` variable (the result of the `getTimeRemaining()` function) reaches zero –-this means now and the other date are the same - stop the clock from counting down.

   d. Close the `changeClock()` function.

6. *Outside the `changeClock()` function but inside the `setClock()` function:*

   a. Call the `changeClock()` function once, to get the clock to display initially. You can only call an inner function from its parent.

   b. Under that, call the `changeClock()` function with `setInterval()`, every second. Create an `intervalID` variable to pass to the `clearInterval()` that is back in the `changeClock()` function.

7. Outside both functions:
    a. Set a variable to a date in the future.
    b. Call `setClock()` on the variable.
8. Reference the file to `Lesson 9/Exercises/js/exercise9-1.js.`
9. Save and test with a date in the past, or without `setInterval()`.

Working with DOM Nodes

A `Node` can be of type element, attribute, comment or text. It has several properties and methods.

Creating Nodes with the DOM

The `document` object has several methods that let it create comment, element or text nodes. The new nodes must be positioned with append and insert node methods.

`document.createComment()` takes a string and creates and returns a new comment node dynamically.

```
document.createComment("this is my comment");
```

`document.createElement()` takes a string and creates and returns a new element node dynamically. Libraries such as HTML5Shiv and Modernizr make use of this method so that HTML5 tags can be recognized in older browsers.

```
document.createElement("section");

document.createElement("article");

document.createElement("div");
```

`document.createTextNode()` takes a string and creates and returns a new text node dynamically.

```
document.createTextNode("Hello World!");
```

Using the `Node` Interface Properties

`Node.nodeName` is a read-only property returning the node name as a string. Nodes can be `"#comment"`, `"#text"`, or the element or attribute name.

`Node.nodeType` is a read-only property returning the node type as an integer. 1 is an element node, 3 is a text node and 8 is a comment node. Although the property returns an integer, the constants `Node.ELEMENT_NODE`, `Node.TEXT_NODE` and `Node.COMMENT_NODE` are available to you.

```
//true

if(div1.nodeType == Node.ELEMENT_NODE)
```

`Node.nodeValue` returns a string containing the value of a text or comment node, or the attribute value of an attribute node. It can also be used to set a `nodeValue`.

Numeric Node Type	**Constant**	**Interface**	**Description**
1	ELEMENT_NODE	Element	Represents an element
2	ATTRIBUTE_NODE	Attr	Represents an attribute
3	TEXT_NODE	Text	Represents text content of an element or attribute
4	CDATA_SECTION_NODE	CDATASection	Represents a CDATA section in a document (text that is NOT parsed)
5	ENTITY_REFERENCE_NODE	EntityReference	Represents an entity reference
6	ENTITY_NODE	Entity	Represents an entity
7	PROCESSING_INSTRUCTION_NODE	ProcessingInstruction	Represents a processing instruction

Numeric Node Type	Constant	Interface	Description
8	COMMENT_NODE	Comment	Represents a comment
9	DOCUMENT_NODE	Document	Represents the entire document (the root-node of the DOM tree)
10	DOCUMENT_TYPE_NODE	DocumentType	Provides an interface to the entities defined for the document
11	DOCUMENT_FRAGMENT_NODE	DocumentFragment	Represents a "lightweight" Document object, which can hold a portion of a document
12	NOTATION_NODE	Notation	Represents a notation declared in the DTD

`Node.textContent` returns the `nodeValue` of a text or comment node. For elements, it returns a concatenated string of all child nodes, but excludes comments. Setting `Node.textContent` removes child nodes and replaces them with a text string.

```
//demo9-2.html

<div id = "div1">Hello World</div>

var div1 =  document.getElementById("div1");

var text = div1.textContent;

alert(text); //alerts "Hello World"

div1.textContent = "Goodbye World";

//results in:

<div id = "div1">Goodbye World</div>
```

The `Node` Traversal Properties

`Node.childNodes` is a read-only property that returns a live list of child nodes of the element. `Node.parentNode` is a read-only property returning the parent node of an element.

```
var div1 = document,getElementById("d1");

//can loop through children

var children = div1.childNodes;
```

`Node.firstChild` and `Node.lastChild` are read-only properties that respectively return the first or last child node of the element, and or null if the element is empty. If the element is not empty, the child node will be an element, text or comment node.

```
div1.firstChild;

div1.lastChild;
```

`Node.previousSibling` and `Node.nextSibling` are read-only properties returning the node immediately preceding or following the specified node, or `null` if the current node is the first or last one.

```
<div id = "d1">div one</div>

<div id = "d2">div two</div>

//gets second div

div1.nextSibling();
```

The `Node` Methods

The `Node` interface of the DOM has several methods for inserting, removing or replacing nodes on a web page. The methods can be used with dynamically created nodes. If the node already exists, it gets moved, removed or replaced, along with its child elements, attributes, text and comments. Otherwise, the node must be created with a method.

The exception is if the node is cloned with `Node.cloneNode()`, which takes an optional true argument if a deep copy is to be made. A deep copy includes child and text nodes. In either case, `cloneNode()` copies the element's attributes, but not dynamically added event listeners.

`Node.insertBefore(`*`newnode, reference nod`*`e)` inserts a node before an existing node, rather than within it. It takes two arguments, the *new node* to be inserted and the *reference node* to insert before. The *reference node* must be a child of the *parent node* that uses the `insertBefore()` method.

```
//demo9-3.html

//div1 must be a child of divMain

divMain.insertBefore(para1, div1);

//results in

<div id = "main">

<p id="para1">paragraph</p>

<div id = "div1">div</div>

</div>
```

`Node.appendChild(`*`element`*`)` adds a node to end of the list of children under a parent node.

```
//demo9-4.html

var divMain = document.getElementById("main");

var para1 = document.getElementById("para1");

var div1 = document.getElementById("div1");

div1.appendChild(para1);

//results in

<div id = "main">

   <div id = "div1">div

      <p id="para1">paragraph</p>

   </div>

</div>
```

`Node.removeChild(`*`child`*`)` removes a child node from the DOM and returns the removed node. The removed child exists in memory, but is not part of the DOM. If you want to reuse the child node elsewhere in your code, capture it in a variable. Otherwise, the node gets automatically deleted from memory.

```
//assuming the following structure

<div id = "main">

   <div id = "div1">div

      <p id="para1">paragraph</p>

   </div>

</div>

var divMain = document.getElementById("main");

var para1 = document.getElementById("para1");

var div1 = document.getElementById("div1");
```

```
//para1 is not a direct child, so does nothing
divMain.removeChild(para1);
//removes div1 and children, including para1
divMain.removeChild(div1);
//now:
<div id = "main"></div>
```

`Node.replaceChild(`*`newchild, oldchild`*`)` replaces a child with another element.

```
//assuming the following structure
          <p id="para1">paragraph</p>
          <div id = "main">
   <div id = "div1">div</div>
</div>
var divMain = document.getElementById("main");
var para1 = document.getElementById("para1");
var div1 = document.getElementById("div1");
divMain.replaceChild(para1, div1);
//yields
<div id = "main">
   <p id="para1">paragraph</p>
</div>
```

Emulating Methods

There is no native "insertAfter" or "prepend" method in JavaScript, but they can be emulated.

An "insertAfter" can be emulated by combining an `insertBefore()` with `nextSibling`. This works even if `div1` is the last child and does not have a `nextSibling`.

```
            divMain.insertBefore(para1, div1.nextSibling);
```

`insertBefore()` can be combined with `parent.firstChild` to create a "prepend."

```
            divMain.insertBefore(para1,  divMain.firstChild);
```

Test Methods

`Node.contains(`*`otherNode`*`)` searches the parent node for another node and returns a Boolean value indicating whether *`otherNode`* is a descendant of the *`parentNode`*.

```
            //given the following structure

            <div id = "main">

              <section>

                  <p id="para1">paragraph</p>

               <section>

            </div>

            //true

            divMain.contains(para1)

            //false

            divMain.contains(div1)
```

`Node.hasChildNodes()` returns a Boolean value, `true` if the node has child nodes.

```
            //true

            divMain.hasChildNodes();
```

The `Element` Interface

`Element` differs from `Node` in that it specifically represents an element. It has properties and methods that pertains to elements in general.

`Element.attributes` is a read-only property that returns a `NamedNodeMap`, a live collection of key-value pairs of all attribute nodes of that element.

`Element.hasAttribute(attName)` returns a Boolean indicating whether the element has the specified attribute, and `Element.hasAttributes()` returns a Boolean determining whether the element has *any* attributes.

```
<img src="logo.png" alt="Company Logo" id="img1">

var img1 = document.getElementById("img1");

var attrs = img1.attributes

//loop through attributes
```

`Element.children` differs from `Node.childNodes`. *children* are elements, *childNodes* are any node.

`Element.id` returns or sets the element's unique `id` attribute. Use the correct case to get it.

`Element.innerHTML` gets or sets the element's descendants. When used to get, it includes HTML tags. When used to set, it overwrites any previous content.

`Element.getAttribute(`*`attName`*`)` gets the value of the specified attribute.

`Element.setAttribute(`*`name, value`*`)` changes the value of an existing attribute with that name, or adds a new attribute. `Element.removeAttribute(`*`name`*`)` removes an attribute's name/value pair.

`Element.remove()` removes the current element from its place in the DOM tree.

Manipulating CSS with the DOM

Within JavaScript, you can set classes or the style attribute for elements. `class` is an HTML attribute, `className` is the JavaScript equivalent property to set or get a class. The word "class" conflicts with programming languages that are used to manipulated the DOM. The `className` references a class from CSS. If a `class` attribute already exists in the HTML element, this adds another one.

```
//CSS

.highlight{

}

divMain.className = "highlight";

//yields

<div id = "divMain" class="highlight"></div>

var cName = divMain.className;
```

`window.getComputedStyle(`*`element`*`)` is a read-only property that gets the computed style of the element argument from an external or embedded CSS stylesheet. The computed style is all CSS property values applied to an element.

`element.style` gets or sets the equivalent of the HTML's *`style`* attribute. This has the highest priority, over embedded stylsheets and external stylesheets, in the CSS cascade. Setting the `style` property does not overwrite other styles assigned to the element. Using `element.setAttribute("style", "properties")`, overwrites the whole style attribute.

`style` properties that are hyphenated in CSS (`margin-top`) are written as camelCase in JavaScript (`marginTop`).

```
//demo9-5.html

divMain.style.backgroundColor = "gray";

divMain.style.color = "red";

var bg = divMain.style.backgroundColor;

//overwrites color and background color

divMain.setAttribute("style", "color: navy");
```

Exercise 9-2 Manipulate CSS with the DOM

Use DOM scripting to set CSS classes dynamically and create an interactive menu.

1. In the browser, open the file `exercise9-2.html` located in `Lesson 9/Solutions.` When you click on the three bars to the right of the page, a menu bar slides out, and the three bars turn into an X. When you click on the X, the menu bar disappears. This effect is mostly created by CSS, so we will use JavaScript to set CSS classes dynamically.

2. In your editor, open `exercise9-2.js` located in `Lesson 9/Exercises/js`.

3. Get the DOM elements with `class="effects"` and `class = "bt-nav"` as variables.

4. Set the `className` property of the element that has `class="effects"` (that you set a variable for) to a class in the stylesheet called `"open".`

5. Attach an event handler to the `class = "bt-nav"` element that you set a variable for. The code looks like: `btnNav.addEventListener("click", function() { }).` The menu bar will change when this element is clicked. Events are covered in the next lesson.

6. Inside the `addEventListener` code, test if the current node's parent's `className` is `"open."` If so, set to `"close."` Write an else if the current node's parent's `className` is `"close."` If so, set to `"open."` The menubar will toggle between being displayed and not. *Hint: This gets the current element's parent's className* *`this.parentNode.className`*

7. Add a reference to the file to `Lesson 9/Exercises/exercise9-2.html` and test.

Lesson 10: Event Handling

In this lesson, you will learn about:

1 The `event` Object

2 Event Handling

3 Types of Events

4 Event Methods

The `event` Object

The `event` object represents any DOM event. Categories include mouse events, keyboard events, clipboard events, window events and form events. While events start "on-" in HTML, the "on-" is often dropped within JavaScript.

There are a few ways to register an event handler. This is an old way that should not be used because it is obsolete.

```
<form id = "frm" onsubmit="validator()">
```

A better way is to register the event within JavaScript, once the DOM is loaded. This can be done with traditional event registration, which works in all browsers, or `addEventListener()` which is more flexible. We will explore them in more detail on the next pages.

Event Handling

Interactive web pages are created by having the user run code, usually functions, through events.

Traditional Event Registration

We want to assign the function itself to `onsubmit`. We can pass a *reference* to the function by dropping the parentheses:

```
function validate(){

//code

}

frm = document.getElementById("frm");

frm.onsubmit = validate;
```

Or we can assign an anonymous function expression to the event handler:

```
frm.onsubmit = function (){

//code

};
```

Additional assignments to `onsubmit` would overwrite any existing event handlers.

Registering With `addEventListener()`

The `addEventListener()` method registers the listener on the target it gets called on. The target can be an element, the document or the window.

`addEventListener()` takes the event (drop the "on-" prefix) as its first argument, and a callback function as its second argument. The callback function is a reference to a function to be run on the event. To pass parameters to the callback function, the second argument should be an anonymous function expression.

```
//demo10-1.html

<table id="tbl1">

table
```

```
<tr>

<td id="td1">table column</td>

</tr>

</table>

var tbl1 = document.getElementById("tbl1");

var td1 = document.getElementById("td1");

           function outerMsg(){

              alert("you clicked on the table!");

}

           function innerMsg(){

              alert("you clicked on the column!");

}

//clicking table runs table event

tbl1.addEventListener("click", outerMsg);

//clicking td bubbles

//runs td then parent table event

td1.addEventListener("click", innerMsg);
```

This method allows for multiple handlers for one event.

```
//demo10-2.html

td1.addEventListener("click", innerMsg);

td1.addEventListener("click", otherMsg);

td1.addEventListener("mouseout", newMsg);
```

`addEventListener()` has a third optional Boolean argument. `true` indicates that the user wants the events to occur in the capturing rather than the bubbling phase. Bubbling, the default, states that if there is an event on a parent and child, the child event happens then the parent event. Capturing states that the parent event happens, then the child event.

```
//clicking td captures

//runs table then child td event

tbl1.addEventListener("click", outerMsg, true);

td1.addEventListener("click", innerMsg);
```

The default value of the third argument is `false`, but sometimes it is passed explicitly for older browsers.

```
tbl1.addEventListener("click", callback, false);

tbl1.addEventListener("click", function(){

//code

}, false);
```

IE 6 – 8 support `attachEvent()` with the "on-" prefix instead. Here is a cross-browser solution:

```
if(tbl1.addEventListener){

   tbl1.addEventListener("click", callback);

   }

 else {

tbl1.attachEvent("onclick", callback);

 }
```

Within the handler, `this` references the element on which the event was fired.

```
//demo10-3.html

//this is the object calling the function

function changeColor(){

this.style.backgroundColor = "navy";

}

td1.addEventListener("click", changeColor)

//using this in an anonymous function

td1.addEventListener("click", function(){

this.style.backgroundColor = "navy";

});
```

Removing the Event Listener

You might want to stop an event from happening once it has occurred a certain number of times. `removeEventListener()` will remove an event handler that has been attached with `addEventListener()`. It has the same arguments, but the function must not be anonymous.

IE 6 – 8 support `detachEvent()` with the "on-" prefix instead of `removeEventListener()`.

```
if(tbl1.removeEventListener){

tbl1.removeEventListener("click", callback);

}

else {

tbl1.detachEvent("onclick", callback);

}
```

Types of Events

Mouse Events

Mouse events include `click`, `dblclick`, `mousedown` and `mouseup`. `click` is a combination of `mousedown` and `mouseup` events.

Other mouse events are `mousemove` (when the mouse moves while over an element) and `mouseover/mouseout`.

`mouseenter/mouseleave` are similar to `mouseover/mouseout` but do not bubble up through the DOM.

```
tbl1.addEventListener("mouseout", function(){

        alert("table");

    });

//moving the mouse out of the td runs both

td1.addEventListener("mouseout", function(){

        alert("column");

    });

tbl1.addEventListener("mouseleave", function(){

        alert("table");

    });

//moving the mouse out of the td runs only td

td1.addEventListener("mouseleave", function(){

        alert("column");

    });
```

Keyboard Events

Keyboard events include `keyup`, `keypress` and `keydown`. `keydown` happens when the user first presses a key and `keyup` when the user lets go of a key. The `keypress` event happens when a key is pressed that results in a character value.

In this example, we want to capture the character once it has been typed. For testing purposes, we don't want the form to autocomplete:

```
//demo10-4.html

<form autocomplete="off">

<input type="text" id="fName">

</form>

Hello <span id="sp1"></span>Hello <span
id="sp1"></span>

sp1 = document.getElementById('sp1');

fName = document.getElementById('fName');

function showChar(){

sp1.textContent = fName.value

}

fName.addEventListener("keyup", showChar);
```

`keydown` and `keyup` both represent keys, while `keypress` represents *only input characters*.

The keyboard event object also returns Boolean for `altKey`, `ctrlKey` and `shiftKey`.

```
//demo10-5.html

fld.addEventListener("keydown", function(){

//true if shift key pressed

//false if other key pressed

alert(event.shiftKey);

})
```

Clipboard Events

Some browsers support the clipboard events `cut`, `copy` and `paste`. This can be useful for disabling cut copy or paste for certain fields, for example confirming an email address.

```
//demo10-6.html

<form id="frm">

Confirm email: <input type="text" id="email">

</form>

function disable(){

   alert("no cutting, copying or pasting");

   event.preventDefault();

}

email = document.getElementById('email');

email.addEventListener("cut", disable);

email.addEventListener("copy", disable);

email.addEventListener("paste", disable);
```

Window Events

The `window` object has events for `load` and `unload`. The `load` event is very important when working with the DOM, as we want to make sure the DOM has loaded before accessing elements. Previous examples resolved this by putting the script tags below the HTML content we were accessing.

```
//demo10-7.html

window.addEventListener("load", function(){

//get div element once we are sure it's //loaded

divMain = document.getElementById("divMain");

});
```

Form Events

Form events are those that happen on form fields and are good for invoking form validation methods. There are a few events specific to forms:

`focus` and `blur` mean that the user enters or leaves a field. When the `blur` event happens, validation code can be run on the field.

```
//not necessary to use preventDefault

fName.addEventListener("blur", function(){

if(fName == ""){

alert("Name field is empty");

}

})
```

The `change` event happens when the value of a field, such as a pulldown menu, has changed.

`select` happens when the user selects the text in a field.

`reset` happens when the reset button is clicked, and `submit` happens when the form is submitted. The form can be submitted when the user clicks a submit button, but `submit` is an event on the form object, not a button.

```
//preventDefault prevents form submission

function validate(){

if(fName == ""){

alert("Name field is empty");

event.preventDefault();

}

}

//frm is form

frm.addEventListener("submit", validate);
```

Forms often use other events, like `click` for a checkbox, radio button or other button.

```
//sbm is submit button

sbm.addEventListener("click", validate);
```

Since a form gets user input, `cut`, `copy` and `paste` events might also be used on a form field.

Event Methods

The event object has methods to stop bubbling or default behavior, which can be useful in event handling.

stopPropagation();

DOM elements are often nested, so when an event such as a click happens to a child element, it also happens to its parent. An event *bubbles* from the child tag, triggering its parents in nested order.

Clicking the child `tr` or `td` would trigger the `table` event. As you may recall, the `td` or `tr` also had an event handler, that would run as well.

```
<table id="tbl1">

  table

  <tr>

  <td id="td1">table column</td>

  </tr>

  </table>

  var tbl1 = document.getElementById("tbl1");

  var td1 = document.getElementById("td1");

//when table is clicked on

tbl1.addEventListener("click",
function(){alert("table")})

//both events happen when td is clicked on

td1.addEventListener("click", function(){alert("td")})
```

The event object has a `stopPropagation()` method to cancel the bubbling phase, so that if the user clicks on the `td`, *and the* `td` *has an event associated with it,* only that event will execute. The `table` would have to be clicked on directly for its event to run. If the `td` does not have an event associated with it, the table event runs regardless of whether `stopPropagation()` is used.

```
tbl1.addEventListener("click", function(){

//event if table clicked

alert("table");

});

td1.addEventListener("click", function(){

alert("td");

//don't bubble to parent

//only td event happens if td is clicked

event.stopPropagation();

});
```

`event.stopImmediatePropagation()` stops other events from executing *on that element* as well as on the parent.

preventDefault()

The `event` object has a method that stops the default behavior of the element. It can stop form submission if the form is not valid, or stop a link from going to another page if it should load content through AJAX instead.

```
<form id="frmContact" action="success.html">

   <input type="text" id="fName">

   <input type="submit">

</form>

var frm = document.getElementById("frmContact");

var fName = document.getElementById("fName");

//test when form submit button pressed

frm.addEventListener("submit", function(){
    if(fName.value == ""){

         alert("Name field is empty");

         //don't submit form if field empty

         event.preventDefault();

    }

    })
```

`preventDefault()` does not stop bubbling. Both `event.preventDefault()` and `event.stopPropagation()` should be used to prevent default behavior and bubbling.

Exercise 10-1 Using events

Create a read more/read less link with the DOM, CSS and events.

1. In the browser, open file `exercise10-1.html` located in `Lesson 10/Solutions`. When you click on the "Read more" link, you will see another paragraph. Then you can click on "Read less" and the second paragraph will disappear.

2. In your editor, open `exercise10-1.html` located in `Lesson10/Exercises.` Note how the page is laid out with element IDs. There are classes called `moreless`, `moreless-show`, `moreless-hide`, `showLink` and `hideLink`.

3. In your editor, open `exercise10-1.js` located in `Lesson 10/Exercises/js`.

4. Write a function called `showHide()`. Inside

    a. Using JavaScript to dynamically set CSS: if the element whose id is `"moreless-show"` is not set to `display: none`, set it to `display: none`, and set the element whose id is `"moreless"` to `display: block`. *Hint:* *`document.getElementById('moreless-show').style.display != 'none'`*

    b. Using an else if, if the element whose id is `"moreless-show"` is set to `display: block`, set the element whose id is `"moreless"` to `display: none`.

5. Close the function. Outside the function

    a. Add an `addEventListener()` for the `window load` event. Inside the callback function it takes

        i. Get the element with `class="showLink."` Add a click event listener to that element. When it is clicked on, run the `showHide()` function and stop the default behavior of a link.

        ii. Do the same for the element with `class=hideLink.` Get the element and add and event listener to it for click that runs the `showHide()` function and stops the default behavior of a link.

        iii. Close the `addEventListener()` for the `window load` event.

6. Save, add a reference to the file to `Lesson 10/Exercises/exercise10-1.html`, and test.

Lesson 11: Working with forms

In this lesson, you will learn about:

1 Validating Form Data with JavaScript

2 Data Validation with Regular Expressions

Validating Form Data with JavaScript

An HTML form is the most common way to get input from the user. The data is often sent to a database to be stored, requiring server-side scripting. The HTML `<form>` tag will reference the server-side script and the method with which to send data. When the user clicks a "submit" button, the data is sent to that script.

```
<form method="post" action="proc.php" id="frm">
```

Although the data should ultimately be validated on the server, client-side validation or data qualification is helpful. It gives the server a break, and it makes the page more responsive by letting the user know right away when there is a problem.

If the data is not valid according to our script, we want to prevent form submission and give the user a chance to correct it. Form validation gets data with the DOM, uses events to call functions, and tests with string and regular expression methods.

The form and its elements should have IDs so that they can be uniquely identified and accessed.

```
var frm = document.getElementById("frm");
```

There are several ways to get form elements as a collection that you can loop through. The form has an `elements` collection:

```
var els = frm.elements;

//can also be chained together

document.getElementById("frm").elements;
```

You can use a DOM method:

```
var els = document.getElementsByTagName("input");
```

Or a CSS attribute selector:

```
var els = document.querySelectorAll("input");

//specify the form

var els = document.querySelectorAll("#frm input");

//only input type="text"

var els = document.querySelectorAll("[type=text]");
```

Validation tests include:

- Checking that required fields are not empty
- Making sure one of a set of radio buttons or pulldown options has been selected
- Checking that input is in the correct format
- Compare two fields to make sure they match
- Checking for an accurate numeric range

Form Events

The form object has the events `onsubmit` and `onreset` that can process return values. If they are set to `false`, the submit or reset will not be performed. The `true` and `false` values should be acquired dynamically, of course!

`confirm()` returns `true` if the user clicks the "Ok" button and `false` if the user clicks the "cancel" button. `return` captures the `true` or `false`. If `false`, this code is the equivalent of `onreset = false` and does not reset the form. `true` is automatically returned if `false` is not.

```
function check(){

var yn = confirm("Do you want to clear?");

if(yn == false){

      event.preventDefault();

      }

      }

frm.addEventListener("reset", check);
```

The `onsubmit` event will capture `true` or `false` from a function that determines if the form data is valid. The form will not submit if `onsubmit = false`.

```
function validate(){

fName = document,getElementById("fName");

if(fName.value == ""){

   alert("The name field is empty");

   event.preventDefault();

}

}

frm.addEventListener("submit", validate);
```

Checking for Empty Fields

JavaScript has access to the form and elements' attributes. Form attributes include `method`, `action`, `name` and `id`. Useful element attributes include `name`, `id`, `type` and `value`.

Below, we loop through all text elements. We use `value` to access the contents of the field and test if they are empty, and `id` to identify the missing fields in our error message. This code should go in a function that is called when the form is submitted. It is very important to get the `value`, because you want to test a string, not an object. Once we have the string, we can use string methods like `trim()`.

What if the user types only a space into the firstname field? Technically, that field is not empty, but it doesn't have meaningful content either. Our validating function will strip out leading and trailing spaces and analyze what is left.

```
var els = document.querySelectorAll("[type=text]");

   for (i = 0; i < els.length; i++){

      if(els[i].value.trim() == ""){

         alert(els[i].id + " is required!");

         event.preventDefault();

}

}
```

In this example, we test only one field, when the user leaves the field (when the `onblur` event takes place). Remember that form events include `onfocus` and `onblur`. Form elements also have a `focus()` method that puts the cursor into a field, and `blur()` that takes the cursor out. We use the keyword `this` to refer to the current field. If the user leaves the required field without typing anything in, we will show them an error message and put their cursor back in the field.

```
function validateField(){

     if(this.value.trim() == ""){

        alert("this field is required");

        this.focus();

     }

}

fName.addEventListener("blur", validateField);
```

Testing for String Length

Strings and arrays have a `length` property.

```
var user = document.getElementById("user");

if (user.value.length < 5 || user.value.length > 8)
{

alert("Must be 5 to 8 characters.");

user.focus();

}
```

Using string methods, we can restrict the length of a comments field. We use the `substring()` method to grab the first 100 characters and put them in the `textarea`:

```
//demo11-1.html
<form id="frm">
<textarea id="txt" rows="7" cols="35"></textarea>
<input type="submit">
</form>
var txt = document.getElementById("txt");
function shortText(){
var numChars = txt.value.length;
if(numChars > 100){
var short = txt.value.substring(0,100);
alert("Character limit exceeded.");
event.preventDefault();
txt.value = short;
}
}
frm.addEventListener("submit", shortText);
```

Making Sure a Selection is Made

Radio Buttons

Radio buttons are grouped together, and only one can be selected if they have the same `name`. The only way to unselect one is to select another. For form validation, we can loop through the set of radio buttons until we find the one that has been selected.

In this example, the for loop stops running once a radio button is checked. If none are checked, it "falls through" to the error message. This is to avoid having the error message display for every unchecked radio button.

```
//demo11-2.html

<input type="radio" name="rdo" value="UPS">UPS

<input type="radio" name="rdo"
value="FedEx">FedEx

var rdo =
document.querySelectorAll("[type=radio]");

function checkRdo(){

   for(i = 0; i < rdo.length; i++){

      //if a radio button is checked

      if(rdo[i].checked == true){

         return;//leave function

      }

   }

alert("Please choose one.");

event.preventDefault();

}
```

Pull Down Menu

In HTML, the `<select>` element has child `<option>` elements. In JavaScript, `select` has an `options` collection and a `selectedIndex` property that is zero-based. A pulldown menu with five choices has indexes from 0 to 4. When a new selection is made, it triggers the `onchange` event. In this example, we provide a blank selection, then test if that is the `selectedIndex`.

```
//demo11-3.html
<select id="ship">
<option value="">Select One
<option value="UPS">UPS
<option value="FedEx">FedEx
</select>
var sel = document.getElementById("ship");
function checkSel(){
if(sel.selectedIndex == 0){
        alert("Please choose one.");
        event.preventDefault();
}
}
frm.addEventListener("submit", checkSel);
```

Testing whether the user has checked a box

Checkboxes and radio buttons have a `checked` property that is `true` or `false`. We can use it to see if they checked a checkbox. Checkboxes can be toggled on and off.

```
  //demo11-4.html
  <input type="checkbox" id="chk">
  var chk = document.getElementById("chk");
  function check(){
  if (chk.checked == false) {
  alert("please read our terms!");
  event.preventDefault();
  }
}
frm.addEventListener("submit", check);
```

Data Validation with Regular Expressions

Many fields on a contact form use common patterns for data. A U.S. phone number should have ten numbers with optional parentheses, spaces and hyphens.

```
//demo11-5.html

Phone: <input type="text" id="phone">

var phone = document.getElementById("phone");

function check(){

rePhone=/^\(?\d{3}\)?\s?\d{3}-?\d{4}$/;

if(!rePhone.test(phone.value)){

alert("bad format");

event.preventDefault();

}

}

frm.addEventListener("submit", check);
```

A U.S. zip code should have 5 to 9 numbers with an optional space or hyphen. Use one of the regular expression or string methods to implement them.

```
//demo11-6.html
Zip: <input type="text" id="zip">
var zip = document.getElementById("zip");
function check(){
reZip=/^\d{5}|\d{5}-\d{4}$/;
if(!reZip.test(zip.value)){
alert("bad format");
event.preventDefault();
zip.focus()
}
}
frm.addEventListener("submit", check);
```

Exercise 11-1 Validating a form

Commented [A1]: In the solution file if you test it while completely empty, all errors are returned including a field being empty and invalid. Telling me it's empty makes sense, don't tell me it's invalid until I type something. There aren't two things wrong at the same time, only one.

Validate a form using the DOM, events, functions, string tests and regular expressions.

1. In the browser, open file `exercise11-1.html` located in `Lesson 11/Solutions`. Test what happens when you don't fill it out, and when you fill it out correctly. Look at it in the editor and see the IDs in the HTML tags.

2. In your editor, open `Lesson 11/Exercises/js/exercise11-1.js`. We are going to write some functions testing different things, and call them when the form is submitted.

3. First, create two variables at the top of the page, but don't set them. One should be called `msg` and one should be called `name`.

4. Create a function called `checkRequired()` that takes a parameter `fld`. It will check if `fld` is empty. If so, it will create a message that says that `name` is required. Use the `+=` operator so we can build the error message string, and add a newline (`\n`) at the end of the string, so we can display the complete set of error messages on different lines in an `alert()`. `name` is the global variable that will be populated later with the field name we run the function on. *Hint:* *`msg += name + " is required\n";`*

5. Create a function called `checkLength()` that takes three parameters, `fld`, `min` and `max`. Inside:

   a. Get the `fld` parametr's `value`.

   b. Test whether the value is between `min` and `max`.

   c. If not build on the `msg` variable with `+=` and add a string that says name must be between `min` and `max`.

6. Create a function called `madeSelection()` that takes `fld` as a parameter. Inside:

   a. Test whether `fld` is of `type` select. *Hint:* *`fld.type=="select-one"`*

   b. Also test if the `selectedIndex` of `fld` is zero. selectedIndex is a property of a pulldown menu, representing the index number of the selection. We made our first selection (index 0) say "Select One," so if that is still the `selectedIndex`, they haven't chosen anything. Both of these tests can be in one `if`.

   c. Add to `msg` that this field is required.

7. Create a function called `checkLetters()` that takes `fld` as a parameter. Inside:

   a. Get the `value` of `fld`.

   b. Set a regular expression variable to check for letters only.

   c. If the regular expression doesn't match the field's value, use `msg` to add to the error message.

8. Create a function called `checkZip()` that takes `fld` as a parameter and does the same as above, for a zipcode regular expression.

9. Create a function called `checkAlphaNumeric()` that takes `fld` as a parameter and does the same as above, for a regular expression that allows letter and numbers only.

10. Create a function called `checkEmail()` that takes `fld` as a parameter and does the same as above, for an email regular expression.

11. Now, we will create another function called `validateForm()` This one uses `frm` as a parameter to represent the form, and calls the other functions on it.

    a. Declare a local variable called `fld`.

    b. Set `msg` to an empty string.

    c. Loop through the form elements (`frm` is the form). In the loop:

       i. Set `fld = frm.elements[i];` (each element).

       ii. Set `name = fld.name.substring(3);` This takes the "txt" or "ddl" prefix off of the ID of the element so it says "Interests" instead of "ddlInterests" in the error message.

       iii. If the field type is text *Hint:* *`if(fld.type == "text")`* then call the `checkRequired()` function on `fld`. Inside this `if`:

          1. If `name` is `FirstName`, run the `checkLetters(fld)` function.

          2. If `name` is `UserName`, run the `checkLength(fld, min, max)` (supplying numbers for `min` and `max`) and `checkAlphaNumeric(fld)` functions.

          3. If `name` is `Zip`, run the `checkZip(fld)` function

          4. If `name` is `Email`, run the `checkEmail(fld)` function

       iv. `else if` field `type` is `"select-one"`, run `madeSelection(fld).`

       v. `else if` field `type` is `"submit,"` don't run any functions, use `continue` to skip this field. We are looping through all fields, but have no testing for this field.

    d. When that set of `if/else if` is finished, close it and do another test: if `msg` is empty, `return true`, if not, alert the `msg` we concatenated and prevent the default behavior. We are going to call the form validator on form submission, so we don't want the form to submit if there are any errors.

    e. Finally, close the `validateForm(frm)` function.

12. On page load, get the form by id, and on form submission, run the `validateForm()` function on it.

13. Add a reference to the file to `Lesson 11/Exercises/exercise11-1.html` and test.

Commented [A2]: When or where did we discuss the form element type property.

Lesson 12: Debugging and best practices

In this lesson, you will learn about:

1. JavaScript Coding Styles
2. JavaScript Best Practices
3. Custom Error Handling

JavaScript Coding Styles

Most JavaScript code should be in a separate file with the extension `.js`, so that it can re-used, tested and developed more easily. The same coding conventions should be used for all projects to make it easier to read and maintain.

Variable and Statement Coding Styles

While HTML is case insensitive, most web servers such as Apache and Unix treat *file names* as case sensitive. If the file is called "logo.png" and our DOM script tries to access "Logo.png" with a different case, it will fail. The convention is to use lowercase in HTML for consistency and readability.

Use camelCase for identifier names (variables and functions).

```
var firstName = "John";

var lastName = "Doe";

function calcTotal(){}
```

Although it isn't required, it is more readable to put spaces around operators and after commas.

```
var z = x + y;

var cities = ["Atlanta", "Boston", "Chicago"];
```

Although JavaScript does not throw an error for a missing semicolon if a new statement starts on the next line, you should always put semicolons on the end of each statement.

```
var fullName = firstName + " " + lastName;
```

When assigning an object, the semicolon goes at the end of the curly braces. This is different from the curly braces in a block statement, which should not have a semicolon after them.

The opening curly braces goes on the same line as the object name.

Use a colon and space between the property name and value.

String *values* (not property names) must have quotes around them.

Numeric values should not have quotes.

In JSON, the property names must also have quotes around them.

Comma separate property/value pairs, but there is no comma after the last one.

The closing brace goes on a new line, with no leading spaces.

```
var person = {

firstName: "Jane",

lastName: "Doe"

};
```

If a JavaScript statement must be broken for readability, it should be done between operators, especially a comma if possible.

```
var cities = ["Atlanta", "Boston",

           "Chicago"];
```

Block Coding Styles

The block statement, where several statements can be grouped together with a semicolon, has several conventions. The opening curly brace should be at the end of the first line, with a space before it. The closing curly braces should be on its own line with no spaces before it. The statements inside the block should have semicolons, but the block itself should not.

```
//spaces before parentheses
//space before curly opening brace
if (hour < 12) {
alert("Good Morning!");
}
else {
alert("Good Afternoon!");
} //no semicolon here
```

It is a general convention to use four spaces to indent code blocks. Avoid using tabs, because they are not interpreted consistently by different editors. Generally, there should *not* be a space before the parentheses unless it is an anonymous function.

```
function calcTotal(){

   //indenting with four spaces

              return (price * tax) + price;

           }

           //put a space before parentheses in an
           //anonymous function

           function () {

              //do this

           }
```

JavaScript Best Practices

Although JavaScript doesn't require it, always declare variables. Declaring a variable inside a function makes it local to the function; this is preferable to a global variable.

```
//bad, undeclared variable

x = 1;

//declared variable

var x = 1;
```

Variables should be declared at the top of the page, to make it easier to find any variable declarations.

```
//declares multiple variables in one statement

var firstName, lastName, intAge;
```

It is even better to initialize variables when declaring them, and at least hint at the intended data type. Use string, numeric, array and object literals for code efficiency. Note the indentation for readability.

```
//can still be separated by comma

var firstName = "",

      fullPrice = 0,

      myArr = [],

      myObj = {};
```

JavaScript is loosely typed, so if you want to perform type as well as value comparison, use the identity === operator rather than the comparison == operators. The comparison == operator tries to convert to matching types.

```
//type conversion makes these true
0 == ""; //true
1 == true; //true
//but they are not identical
0 === ""; //false
1 === true; //false
```

The assignment operator is one equal sign. A common mistake is to use it instead of the comparison or identity operator.

```
var x = 1;

//false, works as expected

if (x == 2) {}

//true, assigns 2 to x and tests whether x is true

if (x=2) {}
```

Optimizing Code Performance

Avoid making JavaScript select or calculate something repeatedly, if it isn't expected to change.

Remember that the length property will tell you how many elements are in an array, and that JavaScript arrays are dynamic. The number of elements could increase or decrease within the code. In this example, JavaScript calculates the array length each time the loop runs.

```
var arr = [];

for (i = 0; i < arr.length; i++) {}
```

If you know the number of elements in an array will remain the same, it is much more efficient to get the length once, and use that number in the loop. In this case, we calculate the number of elements in the array only once.

```
var arr = [];

var len = arr.length;

for (i = 0; i < len; i++) {}
```

While JavaScript's ability to work with the HTML DOM makes web pages very interactive, accessing the DOM is a relatively slow and "expensive" operation. If you expect to access the same DOM object more than once in a script, store it in a variable to re-use.

```
//get once

var div = document.getElementById("divMain");

//use multiple times

div.innerHTML = "Hi";

div.style.color = "blue";
```

In the past, there was a convention to put scripts in the `head` tag of an HTML page to get them out of the way. Browser code is single-threaded, so HTML in the body won't load until the scripts have loaded. The user won't see anything in the browser during that time. As scripts have become longer and it has become popular to use several libraries, it is better to put scripts at the bottom of the page. This way the browser loads the HTML first, and the user can see the page while the scripts are loading. You should use the `window.onload` event to make sure the page has loaded before using any DOM scripting.

```
//instead of

<head>

   <script src="longScript.js"></script>

</head>

<body>

   content, images

</body>


//user sees page while script is loading

           <body>

   content, images

   <script src="longScript.js"></script>

</body>
```

The `"use strict";` Directive

JavaScript is very flexible, and won't generally throw errors for bad syntax. `"use strict"` is a literal expression, ignored by older versions of JavaScript (for example, < IE 10). If the `"use strict"` declaration is at the beginning of a JavaScript file, it has global scope, and all code executes in strict mode. While this is normally valid code, strict mode would cause an error, as it requires all variables to be declared.

```
x = 10;

person = {

firstName: "John",

lastName: "Doe"

};
```

The following forces JavaScript to throw an error for mistyped variables, which are undeclared, instead of making them into a new variable. Since a global `"use strict"` requires variables inside a function to be declared, it effectively requires them to be local.

```
//demo12-1.html
"use strict";
var x = 10;
var person = {
firstName: "John",
lastName: "Doe"
};
for (var i = 0; i < 10; i++) {
   document.write(i);
      }
            function calcTotal(){
               x = 20; //ok because x declared above
               y = 10; //global use strict causes error
               var z = 20; //valid
      }
```

If the "use strict" declaration is inside a function, it has local scope and only applies to the code inside.

```
//demo12-2.html

x = 10; //not in strict mode, this is valid

function calcTotal(){

   "use strict"; //in scope of function

   y = 20; //not valid

   var z = 30; //valid

}
```

The JavaScript `debugger;` Statement

A browser's error console will show syntax errors, but logical errors are difficult to find. One approach is to write to `document.write()`. You can only see the result if you go to the browser's JavaScript console.

```
var x = 10;

alert(x);
```

If there is an error, JavaScript will not run any code. You might want to set breakpoints and run a little bit of code at a time. Breakpoints force JavaScript to stop executing and allow you to check JavaScript values. You can restart the script from within the debugger.

JavaScript has a `debugger;` statement that sets a breakpoint on the line it is on. When the code is viewed in the browser, the `debugger;` statement has no effect. When the code is run with the debugging tools activated, it sets a breakpoint and provides an arrow to click on to restart code. You can use the `debugger;` statement to set breakpoints as many times in the code as you need.

Without debugging tools activated, this alerts 30. In debugger mode, this stops below each variable assignment and waits for an arrow to be pressed to restart the code.

```
//demo12-3.html

var x = 10;

debugger;

var y = 20;

debugger;

alert(x + y);
```

Custom Error Handling

JavaScript has `try`, `catch`, `throw` and `finally` statements that let you customize error handling. `try` takes a block of code to test, `catch` lets you handle the error, `throw` lets you create custom errors, and `finally` lets you execute code regardless of the result of the `try` and `catch`. Combinations of flow control with these four statements are: `try/catch`, `try/finally`, or `try/catch/finally`. The `throw` statement that overwrites the built-in error goes in the `try` block.

Let's say we misspell a method. JavaScript will output an error to the console. We can `catch` the error and redirect it to the page instead of the console. The built-in error is available to the `catch` statement. Whatever variable is passed to `catch` represents the error, in this case `error`. The error has a message property.

```
//demo12-4.html
      try {
         alerts("Welcome guest!");
      }
      catch(error) {
document.getElementById("demo").innerHTML = error.message;
}
```

We can raise an error or throw an exception and create our own error message.

```
//demo12-5.html
var x = prompt("Enter a number", "");
      try {
  //as many tests and throws here as needed
    if (x == "")
      {
  throw "empty";
            }
  }
      catch(error) {
         alert(error); //now the custom error
               }
            finally {
         //clean-up code happens regardless
      }
```

Exercise 12-1 Throwing a custom error

Throw a custom error for a form field.

1. In the browser, open `exercise12-1.html.` Test the form field when it is empty, not a number, and a number under 5 or greater than 100.
2. In your editor, open `Lesson 12/Exercises/js/exercise12-1.js.` We are going to *throw* a set of custom errors for the field.
3. Create a function called `checkAge()`. Inside:
    a. Declare variables `message` and `age`.
    b. Set the message variable from the area on the page with `id="message"` so we can output it.
    c. Set the `age` variable from the age form field.
    d. Use `try` to set a series of if tests.
        i. If the `age` field is empty, `throw` an error stating that it is empty.
        ii. If the `age` field is not a number (*Hint: `isNaN()`*), `throw` an error stating that.
        iii. If the `age` field is below five, `throw` an error stating that it is too low.
        iv. If the `age` field is over 100, `throw` an error stating that it is too high.
    e. Catch the error (*Hint: use `catch(err)` and then print out `err`*). Write the message to the appropriate field, and stop the form from submitting if there is an error.
4. On page load, get the form element, and call the `checkAge()` function when it is submitted.
5. Add a reference to the file to `Lesson 12/Exercises/exercise12-1.html`, save and test.

Index

Glossary

Array

Collection of values with enumerated index.

DOM

The Document Object Model, the representation of an HTML or XML page in memory.

Event

A thing that happens to an HTML element, especially invoked by the user, such as click or mouseover.

Flow Control

Generating control of a script with logic. Includes if/else if/else, switch, ternary operations, loops and branch statements.

Function

Re-usable subroutine.

Object

Set of key: value pairs where the key is a string.

Regular Expression

A pattern created to match data.

Variable

Holds a value and can be changed.