Reading: Exploring JavaScript

Exploring JavaScript **Overview**

Learning Objectives

After completing this lesson, you will be able to:

1. Access more built-in functionality of Numbers and Strings
2. Store related values in Objects
3. Use Arrays to store lists of values
4. Create and use Boolean values
5. Automatically repeat code using loops

In the previous lesson, you learned how to assign values to variables and how to create and use functions.

Continuing from those fundamental building blocks, you'll learn about two of JavaScript's *collection types*. Arrays are used to store sequences of values, while Objects are used to store a group of related values.

As you write larger programs, your code will need to run different functions and statements based on certain conditions. (e.g., *Did the user type "yes" or "no"?*, *Has the timer reached 0?*)

Objects

A dictionary is a book of words and their definitions.

JavaScript provides the same idea, but as a data type.

Consider your address book - it has the contact information of your friends, family, and other important people. Here's an example of an entry you might see in an address book:

Key	Value
Name	Alan Turing
Cell	6017576
Birthday	June 23

This is how you would express this as a JavaScript object:

```
const friendInfo = {
    "name": "Alan Turing",
    "cell": "6017576",
    "birthday": "June 23",
};
```

Each **value**, like `"June 23"`, is associated with a specific **key** (`birthday`).

In other words, an Object is a way to group together labeled information.

It lets you create a **mapping** from *keys* to *values*. They are useful for representing information as records.

How do I create an Object?

Looking more closely at our first example:

```
const friendInfo = {
    "name": "Alan Turing",
    "cell": "6017576",
    "birthday": "June 23",
};
```

An object consists of pairs of keys and values inside of curly braces.

Keys and their values are separated by a `:`.

Key-value pairs are separated by a `,`.

An empty object can be created like so:

```
emptyObject = {};
```

What can I use as a key?

Anything valid String can be used as a key.

As a convenience, JavaScript lets you omit the quotes from the keys:

```
const friendInfo = {
    name: "Alan Turing",
    cell: "6017576",
    birthday: "June 23",
};
```

If you omit the quotes, you must use keys that are also valid variable names.

This is ok:

```
const anotherFriend = {
    name: "Grace Hopper",
    rank: "Admiral",
};
```

This is a syntax error:

```
const anotherFriend = {
    name: "Grace Hopper",
    rank: "Admiral",
    Ph.D.: "Earned in 1934",
};
```

Why? *Periods are not valid in a variable name.*

> **Key names are usually lowercased or camelCased**
>
> Most JavaScript developers choose to leave off the quotes. Because non-quoted keys must follow the same rules as variable names, developers also use same capitalization conventions.

What can I use as a value?

Object values can be any valid JavaScript type!

```
const superhero = {
    name: "Wonder Woman",
    alias: "Diana Prince",
    bracelets: 2,
    lassos: 1,
};
```

How do I retrieve value?

There are two ways to retrieve data from an object

- `[<key name with quotes>]` retrieves the value for a key using the **index** syntax
- `.<key name>` retrieves the value for a key using the **dot** syntax

```
const superhero = {
    name: "Wonder Woman",
    alias: "Diana Prince",
    bracelets: 2,
    lassos: 1,
};

hero_name = superhero["name"];
number_of_lassos = superhero.lassos;
```

> **Most developers use dot syntax**
>
> The dot syntax is equivalent to the bracket syntax.
>
> JavaScript developers prefer the dot syntax because it is shorter.

How do I update a value or add a new value?

Use the dot syntax or bracket syntax on the LHS to update a value. This can also be used to add a key/value pair to a dictionary.

```
const superhero = {
    name: "Wonder Woman",
    alias: "Diana Prince",
    bracelets: 2,
    lassos: 1,
};
```

```
// Updating an existing value
superhero.alias = "Princess Diana of Themyscira";

// Equivalent
// superhero["alias"] = "Princess Diana of Themyscira";

// Adding a new value
superhero.hometown = "Themyscira";
```

How do I remove a key?

Use the `delete` keyword to remove a key/value pair from a dictionary.

```
const superhero = {
    name: "Wonder Woman",
    alias: "Diana Prince",
    bracelets: 2,
    lassos: 1,
};

console.log(superhero.lassos);
// 1

delete superhero.lassos;
console.log(superhero.lassos);
// undefined
```

Arrays

As you learned in the previous section, a single Object can store multiple, labeled values. You access a value using the associated key.

Often, you don't need or want labels for each individual value. Instead, you just need a list of values. For example, you don't need to label each item on a grocery list - you just need to know what the items are.

What is an Array?

An Array:

- is a collection of values

- remembers the order of the items in the collection
- lets you access individual values by their position (first, second, third, etc.)

How do I create an Array?

The easiest way to create one is to assign an Array literal to a variable:

```
const todos = ["pet the cat", "go to work", "shop for groceries",
"go home", "feed the cat"];
```

An Array literal is:

- one or more values (or variables)
- separated by commas `,`
- enclosed in square brackets `[]`

How do I access items in an Array?

Here is how JavaScript stores values in our `todos` Array:

index	value
0	"pet the cat"
1	"go to work"
2	"feed the cat"

JavaScript uses an integer *index* to identify values within a single Array. The *first* index is always `0`.

> **TIP**
>
> One way to think of an index is like a "seat number" on a plane, except that the seat numbers are numeric and start at `0`.

You use the index to refer to a specific item in the Array:

```
const todos = ["pet the cat", "go to work", "shop for groceries",
"go home", "feed the cat"];

const firstItem = todos[0];
console.log(firstItem);
// "pet the cat"

const secondItem = todos[1];
console.log(secondItem);
// "go to work"
```

It is not necessary to create a variable. You can index as part of an expression:

```
const todos = ["pet the cat", "go to work", "shop for groceries",
"go home", "feed the cat"];

console.log(todos[0]);
// "pet the cat"

console.log(todos[1]);
// "go to work"
```

What happens if I specify an invalid index?

JavaScript will tell you that there is nothing at that location:

```
const todos = ["pet the cat", "go to work", "shop for groceries",
"go home", "feed the cat"];
console.log(todos[999]);
// undefined
```

> **undefined means "no value"**
>
> JavaScript uses the keyword `undefined` any time space (in memory) was created for a value, but no value was ever assigned.
>
> Here are some situations where `undefined` commonly appears:
>
> - An Array index where there isn't a value
> - A variable that was created, but never assigned
> - A function that does not `return` a value

How do I know how many items are in an Array?

To find out how many items are in an Array (in other words, how "long" an Array is), use the Array's `.length` property:

```
const todos = ["pet the cat", "go to work", "shop for groceries",
"go home", "feed the cat"];
console.log(`I have ${todos.length} things to do.`);
// "I have 5 things to do."
```

How do I add items to an Array?

To add items to an Array, call its `.push()` method. (A *method* is a function that belongs to an object.)

```
const todos = ["pet the cat", "go to work", "shop for groceries",
"go home", "feed the cat"];
console.log(`I have ${todos.length} things to do.`);
// "I have 5 things to do."

todos.push("go to sleep");

console.log(`I have ${todos.length} things to do.`);
// "I have 6 things to do."

console.log(todos[5]);
// "go to sleep"
```

The `.push()` method accepts an argument (a String, in this example) and places it at the end of the Array.

How do I remove items from an Array?

To remove an item from the end of an Array, use its `.pop()` method:

```
const todos = ["pet the cat", "go to work", "shop for groceries",
"go home", "feed the cat"]

const theLastItem = todos.pop();
console.log(theLastItem);
// 'feed the cat'
```

`.pop()` accepts no arguments, and it returns the last item in the Array.

How do I convert an Array of Strings into a single String?

You may want to combine the elements in an Array into a String.

To do that, call the Array's `.join()` method, passing it a String it will place between each item:

```
const todos = ["pet the cat", "go to work", "shop for groceries",
"go home", "feed the cat"];
const todoString = todos.join('! ');
console.log(todoString);
// 'pet the cat! go to work! shop for groceries! go home! feed the
cat'
```

For Strings printed with `console.log()`, you can use special characters ("escape sequences") to alter the formatting. Let's pass the escape sequence for a line break:

```
const todos = ["pet the cat", "go to work", "shop for groceries",
"go home", "feed the cat"];
const todoString = todos.join('!\n');
console.log(todoString);
```

This is the output:

```
pet the cat!
go to work!
shop for groceries!
go home!
feed the cat
```

Notice that the last item has neither the `!` nor the line break.

How do I turn a String into an Array?

To do the opposite of `.join()`, Strings have a method to `.split()` them into an Array of Strings:

```
const todoString = 'pet the cat! go to work! shop for groceries! go
home! feed the cat';
const todos = todoString.split('! ');
console.log(todos);
```

```
// [
//   'pet the cat',
//   'go to work',
//   'shop for groceries',
//   'go home',
//   'feed the cat'
// ]
```

How do I find the index of an item in an Array?

To get the index of an item in an Array, use the `.indexOf()` method:

```
const todos = ["pet the cat", "go to work", "shop for groceries",
"go home", "feed the cat"];
const idx = todos.indexOf('go to work');
console.log(idx);
// 1
```

If an item is not in an Array, `.indexOf()` will return `-1` (which is not a valid index).

```
const todos = ["pet the cat", "go to work", "shop for groceries",
"go home", "feed the cat"];
const idx = todos.indexOf('go to the beach');
console.log(idx);
// -1
```

Note that `.indexOf()` does *exact* matching. In this example,

```
const todos = ["pet the cat", "go to work", "shop for groceries",
"go home", "feed the cat"];
const idx = todos.indexOf('Go to Work');
console.log(idx);
// -1
```

You might be wondering, how does JavaScript decide whether these two Strings are the same? That is the topic of the next section.

Booleans

To make our programs more useful, they should be able to run different statements based on certain conditions.

Imagine a `printTodoCount()` function that will tell you how many todos you have left. If there are no more todos, then it prints the message "All done!"

Here's how you might use this function:

```
const todos = ['pet the cat', 'feed the cat'];
printTodoCount(todos);
// You have 2 things left to do.

todos.pop(); // Remove a todo

printTodoCount(todos);
// You have 1 thing left to do.

todos.pop(); // Remove a todo

printTodoCount(todos);
// All done!
```

Implementing `printTodoCount()`

Here's the skeleton of our function. It accepts an Array of todos and assigns its `.length` property to a variable:

```
function printTodoCount(todoArray) {
  // Find out how many todos are in the array.
  const howMany = todoArray.length;
  // Print different messages based on that number.
}
```

We can compare `howMany` to `0`. If `howMany` is equal to `0`, we'll print `All done!`

```
function printTodoCount(todoArray) {
  // Find out how many todos are in the array.
  const howMany = todoArray.length;
  // Print different messages based on that number.
  if (howMany == 0) {
    console.log('All done!');
  }
}
```

On line 5 is our first `if` statement, which is written in 3 parts:

- the keyword `if`
- a conditional (in this case, a comparison)
- a code block, wrapped in curly braces

What's a conditional?

A conditional is an *expression* that results in a Boolean value. In JavaScript, there are two literal Boolean values:

- `true`
- `false`

Most of the time, you will use some sort of a comparison to create a Boolean value for your `if` statements.

If you look closely at line 5, we're using two equal signs to check if the value of `howMany` is equal to `0`. This pair of symbols is known as the *equality operator*. A common mistake that developers make is to type the assignment operator (a single `=`) instead of the equality operator.

Many JavaScript developers use the *strict equality operator*, which is three equal signs:

```
function printTodoCount(todoArray) {
  // Find out how many todos are in the array.
  const howMany = todoArray.length;
  // Print different messages based on that number.
  if (howMany === 0) {
    console.log('All done!');
  }
}
```

This version is not only easier to see, but it is safer: it compares the types *and* the values.

An `else` branch

If `printTodoCount()` is passed an Array with one or more todos, it should print a different message. Let's update our function:

```
function printTodoCount(todoArray) {
  // Find out how many todos are in the array.
```

```
  const howMany = todoArray.length;
  // Print different messages based on that number.
  if (howMany === 0) {
    console.log('All done!');
  } else {
    console.log(`You have ${howMany} things left to do.`);
  }
}
```

To provide an alternate code block, use the `else` keyword. We *interpolate* the value of `howMany` into our String. This was only possible because we used backtick-quotes.

Technically, our function works correctly, but the output looks a little funny if we pass it an Array that only has one to do:

```
printTodoCount(['correct your grammar']);
// You have 1 things left to do.
```

We need to provide a third code block, but only if `howMany` is equal to `1`.

```
function printTodoCount(todoArray) {
  // Find out how many todos are in the array.
  const howMany = todoArray.length;
  // Print different messages based on that number.
  if (howMany === 0) {
    console.log('All done!');
  } else if (howMany === 1) {
    console.log('You have 1 thing left to do.');
  } else {
    console.log(`You have ${howMany} things left to do.`);
  }
}
```

You can provide multiple alternate code blocks by using this `else if` syntax. Once it finds a `true` conditional, it stops evaluating the remaining `else if` statements.

> **else must come last**
>
> When writing `if` and `else if` statements, you do not have to have an `else` statement.
>
> But if you do, make sure it comes after all of your `else if` statements.

Other kinds of comparisons

Besides equality, you can test if a Number is greater-than or less-than another Number. Here's a function that tells you how busy you are based on your to do list:

```
function busy(todoArray) {
  const howMany = todoArray.length;
  if (howMany > 10) {
    console.log('Too much to do!');
  } else if (howMany > 7) {
    console.log('Pretty busy...');
  } else if (howMany > 3) {
    console.log('Kinda busy...');
  } else if (howMany > 0) {
    console.log('Mostly free');
  } else {
    console.log('Totally free');
  }
}
```

Here is a table of comparison operators:

`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`!=`	Not equal to
`!==`	(Strict) not equal to

Storing booleans

Boolean values can be assigned to variables, like any other value. It can help make your code a little more readable.

Here is a simplified version of our `busy()` function:

```
function busy(todoArray) {
  const howMany = todoArray.length;
  const isBusy = howMany > 5;
  if (isBusy) {
    console.log('You have plenty to do.');
  } else {
    console.log('Time to relax!');
  }
}
```

Line 8 now reads like a sentence in English: "If is busy..."

Negating booleans

You can use the `!` symbol to "flip" any boolean. `!` is the *negation operator*.

```
function busy(todoArray) {
  const howMany = todoArray.length;
  const isBusy = howMany > 5;
  if (!isBusy) {
    console.log('Time to relax!');
  } else {
    console.log('You have plenty to do.');
  }
}
```

You cannot compare Objects or Arrays

Using the equality or strict equality operator, you can reliably compare Integers and Strings.

Comparing Objects and Arrays, even if their contents match, always produces `false`.

```
const obj1 = { name: 'alice' };
const obj2 = { name: 'alice' };

if (obj1 == obj2) {
  console.log('They are equal');
} else {
  console.log('They are not equal');
```

```
}

// They are not equal
```

You would get the same result using the strict equality operator.

When JavaScript checks if two Objects (or Arrays) are equal, it's comparing memory locations. That is, JavaScript is checking if the two Objects are *the same Object*.

The 8 "Falsy" values in JavaScript

The following values are equivalent to a false Boolean value:

`false`	Keyword for a false Boolean value
`0`	The number zero
`-0`	Negative zero
`0n`	The **BigInt (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt)** literal for zero
`""`	An empty string
`null`	Object that represents an invalid value
`undefined`	Keyword indicating that no value was assigned or returned
`NaN`	"Not-A-Number" - the result of an invalid attempt to create a numerical value

All other values are treated as a true Boolean value.

Loops

Functions provide a way to bundle statements together, including `if-else` statements that only run under certain Boolean conditions.

Functions can be called again and again when you want to run a particular bundle of statements, but that requires the developer to call the function manually.

Loops give programmers a way to run a code block repeatedly. A Boolean condition tells the loop when to stop.

while loops

Here is a loop that prints the numbers from 1 to 10:

```
let n = 1;
while (n <= 10) {
  console.log(n);
  n = n + 1;
}
```

A `while` loop is written in 3 parts:

1. Create a "control variable" (`let n = 1;`)
2. Specify the `while` condition using the control variable (`while (n <= 10)`)
3. In the code block, move the control variable closer ending the loop (`n = n + 1;`)

> **Be careful not to write infinite loops!**
>
> An *infinite loop* is a loop that never reaches the end. Here is a version of our program where we comment out the `n = n + 1;`:
>
> ```
> let n = 1;
> while (n <= 10) {
>   console.log(n);
>   //n = n + 1;
> }
> ```
>
> If you run this program, it will print out the number `1` over and over again without stopping. Why? The value of `n` stays at `1` because we never modify it. Because we never modify it, the condition `n <= 10` stays `true` forever.
>
> To stop a program that has an infinite loop, press the `Control` key and the letter `c` on your keyboard at the same time. This sends a signal to stop the program. (Though it usually takes a few seconds to comply.)

> This key combination (usually abbreviated `Ctrl-C`) will "kill" the running program. You can use it with any program running in the Terminal, not just ones written in JavaScript.

Let's make one more modification to our program. Increasing a value by `1` is so common in programming that there is shorthand for it. The first way is to write `n += 1`. This is a combination of addition and assignment. (Naturally, you can use any value; it doesn't have to be `1`.)

The second way is even shorter. It's the *increment* operator:

```
let n = 1;
while (n <= 10) {
  console.log(n);
  //n = n + 1;
  // n += 1;    // addition assignment operator
  n++;          // Increment operator
}
```

Here's a countdown loop. It will use the *decrement* operator to count down by 1.

```
let n = 10;
while (n > 0) {
  console.log(n);
  n--;                  // decrement operator
}
```

A loop's code block can contain any valid JavaScript, including variable declaration, function calls, and `if-else` statements.

This loop only prints even numbers, by checking if the remainder is `0` when dividing by `2`:

```
let n = 1;
while (n <= 10) {
  const isEven = (n % 2) === 0;
  if (isEven) {
    console.log(n);
  }
```

```
  n++;  // Increment outside of the if-statement!!
}
```

Notice that the `n++` *always* runs as part of the loop's code block. A common mistake is to put the increment inside of the `if` statement. The problem is that the value of `n` would get "stuck" on an odd number and never increment again, resulting in an infinite loop.

This last `while` loop brings us back to our to do list. It prints each item in the list.

```
const todos = ["pet the cat", "go to work", "shop for groceries",
"go home", "feed the cat"]

let i = 0;
const howMany = todos.length;
while (i < howMany) {
  console.log(todos[i]);
  i++;
}
```

This example uses a combination of techniques learned during this lesson.

- Lines 3 sets up our control variable
- Line 4 declares a variable for our Array's `.length`
- Line 5 uses our two variables to specify the end condition of the loop
- Line 6 accesses the Array item at a particular index
- Line 7 increments our control variable

The `while` loop is the most general kind of loop and gives you the most control.

for loops

You can think of the next kind of loop as shorthand for a `while` loop. It all the same main parts:

- A control variable
- An end condition
- Code that moves the control variable closer to the end condition

Here is our counting loop using the `for` syntax:

```
for (let n=1; n<=10; n++) {
  console.log(n);
}
```

All three parts have been moved inside the parentheses next to the `for` keyword. The code block only contains the code that you want to repeat.

Here is our countdown as a `for` loop:

```
for (let n=10; n>0; n--) {
  console.log(n);
}
```

Note that the code block (the "body" of the `for` loop) has not changed.

Lastly, this is our to do list printer as a for loop:

```
const todos = ["pet the cat", "go to work", "shop for groceries",
"go home", "feed the cat"];

const howMany = todos.length;
for (let i=0; i<howMany; i++) {
  console.log(todos[i]);
}
```

Summary

In this lesson, you moved beyond variables that hold a single value. You learned how to hold sequences of ordered values using Arrays, access individual items in the Array, and change the contents of Arrays. Likewise, you now know how to use Objects for storing and manipulating multiple, labeled values.

You learned how to create Boolean values by comparing values, and you used Booleans with `if-else` statements to control which code blocks are run by your program.

Finally, you learned two different looping constructs: the `while` loop, and the more compact `for` loop.

In upcoming lessons, you'll learn how to use loops and conditionals to further manipulate the contents of Arrays and Objects. Combined with variables and functions, these tools give you everything you need to write sophisticated algorithms and solve complex problems.

Mutation vs Reassignment

When variables were introduced in the last lesson, they were declared using two different keywords:

- `const` creates a variable that cannot be reassigned
- `let` creates a reassignable variable

In this lesson, we declared our Objects and Arrays using `const`, and we had no problem adding, removing, and updating items. Why not?

Recall that the RHS (Right-Hand Side) of the assignment is evaluated first. That means that the computer makes some space in its memory to store that value.

Assigning a variable means that we are associating a variable name with that memory location:

```
const todos = ['pet the cat'];
```

Reassigning a variable associates the variable name with a *different* memory location:

```
const todos = ['pet the cat'];

// This causes an error!
todos = ['make breakfast'];
//TypeError: Assignment to constant variable.
```

It is considered reassignment even if the RHS *seems* to be the same value:

```
const todos = ['pet the cat'];

// This also causes an error!
todos = ['pet the cat'];
//TypeError: Assignment to constant variable.
```

This happens because line 4 is creating a second array that happens to contain another copy of the string `'pet the cat'`. The two arrays live in the computer's memory in different locations.

Modifying the contents of an Object or Array does *not* change the memory location.

```
const todos = ['pet the cat'];

// OK to remove an item
todos.pop();

// OK to add an item
todos.push('make breakfast');

// OK to update an item
todos[0] = 'go to sleep';
```
