JAVASCRIPT

HANDBOOK

BY

PAPA REACT

JOIN ZERO TO FULL STACK HERO, TO LEARN MORE VISIT: WWW.PAPAREACT.COM

MEET SONNY

ALSO KNOWN AS PAPA REACT

“To truly succeed,
we must get Comfortable with the Uncomfortable”

Sonny Sangha

ssssangha

TABLE OF CONTENTS

WELCOME PAPAFAM!

This JavaScript Handbook will help you quickly learn JavaScript and several of it's important functions and how to use them in certain use cases.

The ideal reader of the book has only a minimal knowledge of JavaScript, has an idea about React, and is looking forward to diving more into the JavaScript and the web developing ecosystem.

I have used JavaScript in ALL of my builds on YouTube which you can watch by clicking https://www.youtube.com/c/SonnySangha. JavaScript is an awesome tool and compliments exceptionally with React!

I hope you enjoy this eBook and learn a ton about JavaScript!

Sonny

PS: If you want to dive into more tech and go further in your developer skills then head over to www.papareact.com to buy the complete Zero to Full Stack Hero course which covers all the latest tech, JavaScript and much more!

INTRODUCTION

CONSOLE.LOG()

The console.log() method is used to log or print messages to the console. It can also be used to print objects and other info.

```
console.log('Hi there!');
// Prints: Hi there!
```

JAVASCRIPT

JavaScript is a programming language that powers the dynamic behavior on most websites. Alongside HTML and CSS, it is a core technology that makes the web run.

```
// Returns a number between 0 and 1
Math.random();
```

METHODS

Methods return information about an object, and are called by appending an instance with a period . , the method name, and parentheses.

```
Math.random();
//     Math is the library
```

LIBRARIES

Libraries contain methods that can be called by appending the library name with a period . , the method name, and a set of parentheses.

```
let amount = 6;
let price = 4.99;
```

NUMBERS

Numbers are a primitive data type. They include the set of all integers and !oating point numbers.

STRING .LENGTH

The .length property of a string returns the number of characters that make up the string.

```
let message = 'good nite';
console.log(message.length);
// Prints: 9

console.log('howdy'.length);
// Prints: 5
```

DATA INSTANCES

When a new piece of data is introduced into a JavaScript program, the program keeps track of it in an instance of that data type. An instance is an individual case of a data type.

BOOLEANS

Booleans are a primitive data type. They can be either true or false .

```
let lateToWork = true;
```

MATH.RANDOM()

The Math.random() function returns a !oating-point, random number in the range from 0 (inclusive) up to but not including 1.

```
console.log(Math.random());
// Prints: 0 - 0.9
```

MATH.FLOOR()

The Math.floor() function returns the largest integer less than or equal to the given number.

```
console.log(Math.floor(5.95));
// Prints: 5
```

SINGLE LINE COMMENTS

In JavaScript, single-line comments are created with two consecutive forward slashes // .

```
// This line will denote a comment
```

NULL

Null is a primitive data type. It represents the intentional absence of value. In code, it is represented as null .

```
let x = null;
```

STRINGS

Strings are a primitive data type. They are any grouping of characters (letters, spaces, numbers, or symbols) surrounded by single quotes ‘ or double quotes “ .

```
let single = 'Wheres my bandit hat?';
let double = "Wheres my bandit hat?";
```

ARITHMETIC OPERATORS

JavaScript supports arithmetic operators for:

- + addition
- - subtraction
- * multiplication
- / division
- % modulo

```
// Addition
5 + 5
// Subtraction
10 - 5
// Multiplication
5 * 10
// Division
10 / 5
// Modulo
10 % 5
```

MULTI-LINE COMMENTS

In JavaScript, multi-line comments are created by surrounding the lines with /* at the beginning and */ at the end. Comments are good ways for a variety of reasons like explaining a code block or indicating some hints, etc.

```
/*
The below configuration must be
changed before deployment.
*/

let baseUrl =
'localhost/taxwebapp/country';
```

REMAINDER / MODULO OPERATOR

The remainder operator, sometimes called modulo, returns the number that remains after the right-

```
// calculates # of weeks in a year,
rounds down to nearest integer
const weeksInYear = Math.floor(365/7);

// calcuates the number of days left
over after 365 is divded by 7
const daysLeftOver = 365 % 7 ;
```

hand number divides into the left-hand number as many times as it evenly can.

```
console.log("A year has " + weeksInYear
+ " weeks and " + daysLeftOver + "
days");
```

ASSIGNMENT OPERATORS

An assignment operator assigns a value to its left operand based on the value of its right operand.
Here are some of them:

- += addition assignment
- -= subtraction assignment
- *= multiplication assignment
- /= division assignment

```
let number = 100;

// Both statements will add 10
number = number + 10;
number += 10;

console.log(number);
// Prints: 120
```

STRING INTERPOLATION

String interpolation is the process of evaluating string literals containing one or more placeholders (expressions, variables, etc).
It can be performed using template literals: text ${expression} text .

```
let age = 7;

// String concatenation
'Tommy is ' + age + ' years old.';

// String interpolation
`Tommy is ${age} years old.`;
```

VARIABLES

Variables are used whenever there's a need to store a piece of data. A variable contains data that can be used in the program elsewhere. Using variables also ensures code re-usability since it can be used to replace the same value in multiple places.

```
const currency = '$';
let userIncome = 85000;

console.log(currency + userIncome + '
is more than the average income.');
// Prints: $85000 is more than the
average income.
```

UNDEFINED

undefined is a primitive JavaScript value that represents lack of de"ned value. Variables that are declared but not initialized to a value will have the value undefined .

```
var a;
console.log(a);
// Prints: undefined
```

LEARN JAVASCRIPT: VARIABLES

A variable is a container for data that is stored in computer memory. It is referenced by a descriptive name that a programmer can call to assign a speci”c value and retrieve it.

```
// examples of variables
let name = "Tammy";
const found = false;
var age = 3;
console.log(name, found, age);
// Tammy, false, 3
```

DECLARING VARIABLES

To declare a variable in JavaScript, any of these three keywords can be used along with a variable name:

- var is used in pre-ES6 versions of JavaScript.
- let is the preferred way to declare a variable when it can be reassigned.
- const is the preferred way to declare a variable with a constant value.

```
var age;
let weight;
const numberOfFingers = 20;
```

TEMPLATE LITERALS

Template literals are strings that allow embedded expressions, ${expression} . While regular strings use single ‘ or double “ quotes, template literals use backticks instead.

```
let name = "Codecademy";
console.log(`Hello, ${name}`);
// Prints: Hello, Codecademy

console.log(`Billy is ${6+8} years
old.`);
// Prints: Billy is 14 years old.
```

LET KEYWORD

let creates a local variable in JavaScript & can be re-assigned. Initialization during the declaration of a let variable is optional. A let variable will contain undefined if nothing is assigned to it.

```
let count;
console.log(count); // Prints:
undefined
count = 10;
console.log(count); // Prints: 10
```

CONST KEYWORD

A constant variable can be declared using the keyword const . It must have an assignment. Any attempt of re-assigning a const variable will result in JavaScript runtime error.

```
const numberOfColumns = 4;
numberOfColumns = 8;
// TypeError: Assignment to constant
variable.
```

STRING CONCATENATION

In JavaScript, multiple strings can be concatenated together using the + operator. In the example, multiple strings and variables containing string values have been concatenated. After execution of the code block, the displayText variable will contain the concatenated string.

```
let service = 'credit card';
let month = 'May 30th';
let displayText = 'Your ' + service  +
' bill is due on ' +  month + '.';

console.log(displayText);
// Prints: Your credit card bill is due
on May 30th.
```

CONDITIONALS

CONTROL FLOW

Control flow is the order in which statements are executed in a program. The default control flow is for statements to be read and executed in order from left-to-right, top-to-bottom in a program file. Control structures such as conditionals (if statements and the like) alter control flow by only executing blocks of code if certain conditions are met. These structures essentially allow a program to make decisions about which code is executed as the program runs.

LOGICAL OPERATOR | |

The logical OR operator || checks two values and returns a boolean. If one or both values are truthy, it returns true . If both values are falsy, it returns false .

A	B	A || B
false	false	false
false	true	true
true	false	true
true	true	true

```
true || false;          // true
10 > 5 || 10 > 20;      // true
false || false;         // false
10 > 100 || 10 > 20;    // false
```

TERNARY OPERATOR

The ternary operator allows for a compact syntax in the case of binary (choosing between two choices) decisions. It accepts a condition followed by a ? operator, and then two expressions separated by a : . If the condition evaluates to truthy, the first expression is executed, otherwise, the second expression is executed.

```
let price = 10.5;
let day = "Monday";

day === "Monday" ? price -= 1.5 : price
+= 1.5;
```

ELSE STATEMENT

An else block can be added to an if block or series of if - else if blocks. The else block will be executed only if the if condition fails.

```
const isTaskCompleted = false;

if (isTaskCompleted) {
  console.log('Task completed');
} else {
  console.log('Task incomplete');
}
```

LOGICAL OPERATOR &&

The logical AND operator && checks two values and returns a boolean. If both values are truthy, then it returns true . If one, or both, of the values is falsy, then it returns false .

```
true && true;       // true
1 > 2 && 2 > 1;     // false
true && false;      // false
4 === 4 && 3 > 1;   // true
```

SWITCH STATEMENT

The switch statements provide a means of checking an expression against multiple case clauses. If a case matches, the code inside that clause is executed.

The case clause should finish with a break keyword.

If no case matches but a default clause is included, the code inside default will be executed.

Note: If break is omitted from the block of a case , the switch statement will continue to check against case values until a break is encountered or the flow is broken.

```
const food = 'salad';

switch (food) {
  case 'oyster':
    console.log('The taste of the sea
🦪');
    break;
  case 'pizza':
    console.log('A delicious pie 🍕');
    break;
  default:
    console.log('Enjoy your meal');
}

// Prints: Enjoy your meal
```

IF STATEMENT

An if statement accepts an expression with a set of parentheses:

- If the expression evaluates to a truthy value, then the code within its code body executes.
- If the expression evaluates to a falsy value, its code body will not execute

```
const isMailSent = true;

if (isMailSent) {
  console.log('Mail sent to
recipient');
}
```

LOGICAL OPERATOR !

The logical NOT operator ! can be used to do one of the following:

- Invert a Boolean value.
- Invert the truthiness of non-Boolean values.

```
let lateToWork = true;
let oppositeValue = !lateToWork;

console.log(oppositeValue);
// Prints: false
```

COMPARISON OPERATORS

Comparison operators are used to comparing two values and return true or false depending on the validity of the comparison:

- === strict equal
- !== strict not equal
- > greater than
- >= greater than or equal
- < less than
- <= less than or equal

```
1 > 3        // false
3 > 1        // true
250 >= 250   // true
1 === 1      // true
1 === 2      // false
1 === '1'    // false
```

ELSE IF CLAUSE

After an initial if block, else if blocks can each check an additional condition. An optional else block can

```
const size = 10;

if (size > 100) {
  console.log('Big');
} else if (size > 20) {
  console.log('Medium');
} else if (size > 4) {
```

be added after the else if block(s) to run by default if none of the conditionals evaluated to truthy.

TRUTHY AND FALSY

In JavaScript, values evaluate to true or false when evaluated as Booleans.

- Values that evaluate to true are known as truthy
- Values that evaluate to false are known as falsy

Falsy values include false , 0 , empty strings, null undefined , and NaN . All other values are truthy.

```
  console.log('Small');
} else {
  console.log('Tiny');
}
// Print: Small
```

FUNCTIONS

ARROW FUNCTIONS (ES6)

Arrow function expressions were introduced in ES6. These expressions are clean and concise. The syntax for an arrow function expression does not require the function keyword and uses a fat arrow => to separate the parameter(s) from the body.

- There are several variations of arrow functions: Arrow functions with a single parameter do not require () around the parameter list.
- Arrow functions with a single expression can use the concise function body which returns the result of the expression without the return keyword.

```
// Arrow function with two arguments
const sum = (firstParam, secondParam)
=> {
  return firstParam + secondParam;
};
console.log(sum(2,5)); // Prints: 7

// Arrow function with no arguments
const printHello = () => {
  console.log('hello');
};
printHello(); // Prints: hello

// Arrow functions with a single
argument
const checkWeight = weight => {
  console.log(`Baggage weight :
${weight} kilograms.`);
};
checkWeight(25); // Prints: Baggage
weight : 25 kilograms.


// Concise arrow functions
const multiply = (a, b) => a * b;
console.log(multiply(2, 30)); //
Prints: 60
```

FUNCTIONS

Functions are one of the fundamental building blocks in JavaScript. A function is a reusable set of statements to perform a task or calculate a value. Functions can be passed one or more values and can return a value at the end of their execution. In order to use a function, you must define it somewhere in the scope where you wish to call it. The example code provided contains a function that takes in 2 values and returns the sum of those numbers.

```
// Defining the function:
function sum(num1, num2) {
  return num1 + num2;
}

// Calling the function:
sum(3, 6); // 9
```

ANONYMOUS FUNCTIONS

Anonymous functions in JavaScript do not have a name property. They can be defined using the function keyword, or as an arrow function. See the code example for the difference between a named function and an anonymous function.

```
// Named function
function rocketToMars() {
  return 'BOOM!';
}

// Anonymous function
const rocketToMars = function() {
  return 'BOOM!';
}
```

FUNCTION EXPRESSIONS

Function expressions create functions inside an expression instead of as a function declaration. They can be anonymous and/or assigned to a variable.

```
const dog = function() {
  return 'Woof!';
}
```

FUNCTION PARAMETERS

Inputs to functions are known as parameters when a function is declared or defined. Parameters are used as variables inside the function body. When the function is called, these parameters will have the value of whatever is passed in as arguments. It is possible to define a function without parameters.

```
// The parameter is name
function sayHello(name) {
  return `Hello, ${name}!`;
}
```

RETURN KEYWORD

Functions return (pass back) values using the return keyword. return ends function execution and returns the specified value to the location where it was called. A common mistake is to forget the return keyword, in which case the function will return undefined by default.

```
// With return
function sum(num1, num2) {
  return num1 + num2;
}

// Without return, so the function
doesn't output the sum
function sum(num1, num2) {
  num1 + num2;
}
```

FUNCTION DECLARATION

Function declarations are used to create named functions. These functions can be called using their declared name. Function declarations are built from:

- The function keyword.
- The function name.
- An optional list of parameters separated by commas enclosed by a set of parentheses () .
- A function body enclosed in a set of curly braces {} .

```
function add(num1, num2) {
  return num1 + num2;
}
```

CALLING FUNCTIONS

Functions can be called, or executed, elsewhere in code using parentheses following the function name. When a function is called, the code inside its function body runs. Arguments are values passed into a function when it is called.

```
// Defining the function
function sum(num1, num2) {
  return num1 + num2;
}

// Calling the function
sum(2, 4); // 6
```

SCOPE

SCOPE

Scope is a concept that refers to where values and functions can be accessed. Various scopes include:
- Global scope (a value/function in the global scope can be used anywhere in the entire program)
- File or module scope (the value/function can only be accessed from within the file)
- Function scope (only visible within the function),
- Code block scope (only visible within a { ... } codeblock)

```
function myFunction() {

  var pizzaName = "Volvo";
  // Code here can use pizzaName

}

// Code here can't use pizzaName
```

BLOCK SCOPED VARIABLES

const and let are block scoped variables, meaning they are only accessible in their block or nested blocks. In the given code block, trying to print the statusMessage using the console.log() method will result in a ReferenceError . It is accessible only inside that if block.

```
const isLoggedIn = true;

if (isLoggedIn == true) {
  const statusMessage = 'User is logged
in.';
}

console.log(statusMessage);

// Uncaught ReferenceError:
statusMessage is not defined
```

GLOBAL VARIABLES

JavaScript variables that are declared outside of blocks or functions can exist in the global scope, which means they are accessible throughout a program. Variables declared outside of smaller block or function scopes are accessible inside those smaller scopes. Note: It is best practice to keep global variables to a minimum.

```
// Variable declared globally
const color = 'blue';

function printColor() {
     console.log(color);
   }

   printColor(); // Prints: blue
```

ARRAYS

PROPERTY .LENGTH

The .length property of a JavaScript array indicates the number of elements the array contains.

```
const numbers = [1, 2, 3, 4];

numbers.length // 4
```

INDEX

Array elements are arranged by index values, starting at 0 as the first element index. Elements can be accessed by their index using the array name, and the index surrounded by square brackets.

```
// Accessing an array element
const myArray = [100, 200, 300];

console.log(myArray[0]); // 100
console.log(myArray[1]); // 200
console.log(myArray[2]); // 300
```

METHOD .PUSH()

The .push() method of JavaScript arrays can be used to add one or more elements to the end of an array. .push() mutates the original array and returns the new length of the array.

```
// Adding a single element:
const cart = ['apple', 'orange'];
cart.push('pear');

// Adding multiple elements:
const numbers = [1, 2];
numbers.push(3, 4, 5);
```

METHOD .POP()

The .pop() method removes the last element from an array and returns that element.

```
const ingredients = ['eggs', 'flour',
'chocolate'];

const poppedIngredient =
ingredients.pop(); // 'chocolate'
console.log(ingredients); // ['eggs',
 'flour']
```

MUTABLE

JavaScript arrays are mutable, meaning that the values they contain can be changed.
Even if they are declared using const , the contents can be manipulated by reassigning internal values or using methods like .push() and .pop() .

```
const names = ['Alice', 'Bob'];

names.push('Carl');
// ['Alice', 'Bob', 'Carl']
```

ARRAYS

Arrays are lists of ordered, stored data. They can hold items that are of any data type. Arrays are created by using square brackets, with individual elements separated by commas.

```
// An array containing numbers
const numberArray = [0, 1, 2, 3];

// An array containing different data
types
const mixedArray = [1, 'chicken',
false];
```

LOOPS

WHILE LOOP

The while loop creates a loop that is executed as long as a specified condition evaluates to true . The loop will continue to run until the condition evaluates to false . The condition is specified before the loop, and usually, some variable is incremented or altered in the while loop body to determine when the loop should stop.

```
while (condition) {
  // code block to be executed
}

let i = 0;

while (i < 5) {
  console.log(i);
  i++;
}
```

REVERSE LOOP

A for loop can iterate “in reverse” by initializing the loop variable to the starting value, testing for when the variable hits the ending value, and decrementing (subtracting from) the loop variable at each iteration.

```
const items = ['apricot', 'banana',
'cherry'];

for (let i = items.length - 1; i >= 0;
i -= 1) {
  console.log(`${i}. ${items[i]}`);
}

// Prints: 2. cherry
// Prints: 1. banana
// Prints: 0. apricot
```

DO...WHILE STATEMENT

A do...while statement creates a loop that executes a block of code once, checks if a condition is true, and then repeats the loop as long as the condition is true. They are used when you want the code to

```
x = 0
i = 0

do {
```

always execute at least once. The loop ends when the condition evaluates to false.

```
  x = x + i;
  console.log(x)
  i++;
} while (i < 5);

// Prints: 0 1 3 6 10
```

FOR LOOP

A for loop declares looping instructions, with three important pieces of information separated by semicolons ; :

- The initialization defines where to begin the loop by declaring (or referencing) the iterator variable
- The stopping condition determines when to stop looping (when the expression evaluates to false)
- The iteration statement updates the iterator each time the loop is completed

```
for (let i = 0; i < 4; i += 1) {
  console.log(i);
};

// Output: 0, 1, 2, 3
```

LOOPING THROUGH ARRAYS

An array's length can be evaluated with the .length property. This is extremely helpful for looping through arrays, as the .length of the array can be used as the stopping condition in the loop.

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

// Output: Every item in the array
```

BREAK KEYWORD

Within a loop, the break keyword may be used to exit the loop immediately, continuing execution after the loop body.
Here, the break keyword is used to exit the loop when i is greater than 5.

```
for (let i = 0; i < 99; i += 1) {
  if (i > 5) {
    break;
  }
  console.log(i)
}

// Output: 0 1 2 3 4 5
```

NESTED FOR LOOP

A nested for loop is when a for loop runs inside another for loop.

The inner loop will run all its iterations for each iteration of the outer loop.

```
for (let outer = 0; outer < 2; outer += 1) {
  for (let inner = 0; inner < 3; inner += 1) {
    console.log(`${outer}-${inner}`);
  }
}

/*
Output:
0-0
0-1
0-2
1-0
1-1
1-2
*/
```

LOOPS

A loop is a programming tool that is used to repeat a set of instructions. Iterate is a generic term that means “to repeat” in the context of loops. A loop will continue to iterate until a specified condition, commonly known as a stopping condition, is met.

ITERATORS

FUNCTIONS ASSIGNED TO VARIABLES

In JavaScript, functions are a data type just as strings, numbers, and arrays are data types. Therefore, functions can be assigned as values to variables, but are different from all other data types because they can be invoked.

```
let plusFive = (number) => {
  return number + 5;
};
// f is assigned the value of plusFive
let f = plusFive;

plusFive(3); // 8
// Since f has a function value, it can
be invoked.
f(9); // 14
```

CALLBACK FUNCTIONS

In JavaScript, a callback function is a function that is passed into another function as an argument. This function can then be invoked during the execution of that higher order function (that it is an argument of).
Since, in JavaScript, functions are objects, functions can be passed as arguments.

```
const isEven = (n) => {
  return n % 2 == 0;
}

let printMsg = (evenFunc, num) => {
  const isNumEven = evenFunc(num);
  console.log(`The number ${num} is an
even number: ${isNumEven}.`)
}

// Pass in isEven as the callback
function
printMsg(isEven, 4);
// Prints: The number 4 is an even
number: True.
```

HIGHER-ORDER FUNCTIONS

In Javascript, functions can be assigned to variables in the same way that strings or arrays can. They can be passed into other functions as parameters or returned from them as well.
A "higher-order function" is a function that accepts functions as parameters and/or returns a function.

JAVASCRIPT FUNCTIONS: FIRST-CLASS OBJECTS

JavaScript functions are first-class objects.
Therefore:

- They have built-in properties and methods, such as the name property and the .toString() method.
- Properties and methods can be added to them.
- They can be passed as arguments and returned from other functions.
- They can be assigned to variables, array elements, and other objects.

```
//Assign a function to a variable
originalFunc
const originalFunc = (num) => { return
num + 2 };

//Re-assign the function to a new
variable newFunc
const newFunc = originalFunc;

//Access the function's name property
newFunc.name; //'originalFunc'

//Return the function's body as a
string
newFunc.toString(); //'(num) => {
return num + 2 }'

//Add our own isMathFunction property
to the function
newFunc.isMathFunction = true;

//Pass the function as an argument
const functionNameLength = (func) => {
return func.name.length };
functionNameLength(originalFunc); //12
//Return the function
const returnFunc = () => { return
newFunc };
returnFunc(); //[Function:
originalFunc]
```

THE .REDUCE() METHOD

The .reduce() method iterates through an array and returns a single value.

In the above code example, the .reduce() method will sum up all the elements of the array. It takes a callback function with two parameters (accumulator, currentValue) as arguments. On each iteration, accumulator is the value returned by the last iteration, and the currentValue is the current element. Optionally, a second argument can be passed which acts as the initial value of the accumulator.

```
const arrayOfNumbers = [1, 2, 3, 4];

const sum =
arrayOfNumbers.reduce((accumulator,
currentValue) => {
  return accumulator + currentValue;
});

console.log(sum); // 10
```

THE .FOREACH() METHOD

The .forEach() method executes a callback function on each of the elements in an array in order.

In the above example code, the callback function containing a console.log() method will be executed 5 times, once for each element.

```
const numbers = [28, 77, 45, 99, 27];

numbers.forEach(number => {
  console.log(number);
});
```

THE .FILTER() METHOD

The .filter() method executes a callback function on each element in an array. The callback function for each of the elements must return either true or false . The returned array is a new array with any elements for which the callback function returns true .

In the above code example, the array filteredArray will contain all the elements of randomNumbers but 4 .

```
const randomNumbers = [4, 11, 42, 14,
39];
const filteredArray =
randomNumbers.filter(n => {
  return n > 5;
});
```

THE .MAP() METHOD

The .map() method executes a callback function on each element in an array. It returns a new array made up of the return values from the callback function.

The original array does not get altered, and the returned array may contain different elements than the original array.

In the example code above, the .map() method is used to add ' joined the contest.' string at the end of each element in the finalParticipants array.

```
const finalParticipants = ['Taylor',
'Donald', 'Don', 'Natasha', 'Bobby'];

// add string after each final
participant
const announcements =
finalParticipants.map(member => {
  return member + ' joined the
contest.';
})

console.log(announcements);
```

OBJECTS

RESTRICTIONS IN NAMING PROPERTIES

JavaScript object key names must adhere to some restrictions to be valid. Key names must either be strings or valid identifier or variable names (i.e. special characters such as - are not allowed in key names that are not strings).

```
// Example of invalid key names
const trainSchedule = {
  platform num: 10, // Invalid because of the space between words.
  40 - 10 + 2: 30, // Expressions cannot be keys.
  +compartment: 'C' // The use of a + sign is invalid unless it is enclosed in quotations.
}
```

DOT NOTATION FOR ACCESSING OBJECT PROPERTIES

Properties of a JavaScript object can be accessed using the dot notation in this manner: object. propertyName . Nested properties of an object can be accessed by chaining key names in the correct order.

```
const apple = {
  color: 'Green',
  price: {
    bulk: '$3/kg',
    smallQty: '$4/kg'
  }
};
console.log(apple.color); // 'Green'
console.log(apple.price.bulk); // '$3/kg'
```

OBJECTS

An object is a built-in data type for storing key-value pairs. Data inside objects are unordered, and the values can be of any type.

ACCESSING NON-EXISTENT JAVASCRIPT PROPERTIES

When trying to access a JavaScript object property that has not been defined yet, the value of undefined will be returned by default.

```
const classElection = {
  date: 'January 12'
};

console.log(classElection.place); //
undefined
```

JAVASCRIPT OBJECTS ARE MUTABLE

JavaScript objects are mutable, meaning their contents can be changed, even when they are declared as const . New properties can be added, and existing property values can be changed or deleted.

It is the reference to the object, bound to the variable, that cannot be changed.

```
const student = {
  name: 'Sheldon',
  score: 100,
  grade: 'A',
}

console.log(student)
// { name: 'Sheldon', score: 100,
grade: 'A' }

delete student.score
student.grade = 'F'
console.log(student)
// { name: 'Sheldon', grade: 'F' }

student = {}
// TypeError: Assignment to constant
variable.
```

JAVASCRIPT FOR...IN LOOP

The JavaScript for...in loop can be used to iterate over the keys of an object. In each iteration, one of

```
let mobile = {
  brand: 'Samsung',
  model: 'Galaxy Note 9'
};

for (let key in mobile) {
```

the properties from the object is assigned to the variable of that loop.

```
  console.log(`${key}:
${mobile[key]}`);
}
```

PROPERTIES AND VALUES OF A JAVASCRIPT OBJECT

A JavaScript object literal is enclosed with curly braces {} . Values are mapped to keys in the object with a colon (:), and the key-value pairs are separated by commas. All the keys are unique, but values are not.

Key-value pairs of an object are also referred to as properties.

```
const classOf2018 = {
  students: 38,
  year: 2018
}
```

DELETE OPERATOR

Once an object is created in JavaScript, it is possible to remove properties from the object using the delete operator. The delete keyword deletes both the value of the property and the property itself from the object. The delete operator only works on properties, not on variables or functions.

```
const person = {
  firstName: "Matilda",
  age: 27,
  hobby: "knitting",
  goal: "learning JavaScript"
};

delete person.hobby; // or delete
person[hobby];

console.log(person);
/*
{
  firstName: "Matilda"
  age: 27
  goal: "learning JavaScript"
}
*/
```

JAVASCRIPT PASSING OBJECTS AS ARGUMENTS

When JavaScript objects are passed as arguments to functions or methods, they are passed by reference, not by value. This means that the object itself (not a copy) is accessible and mutable (can be changed) inside that function.

```
const changeItUp = (num, obj) => {
  num = 7;
  obj.color = 'red';
};

changeItUp(origNum, origObj);

// Will output 8 since integers are
passed by value.
console.log(origNum);

// Will output 'red' since objects are
passed
// by reference and are therefore
mutable.
console.log(origObj.color);
```

JAVASCRIPT OBJECT METHODS

JavaScript objects may have property values that are functions. These are referred to as object methods.

Methods may be defined using anonymous arrow function expressions, or with shorthand method syntax.

Object methods are invoked with the syntax: objectName.methodName(arguments)

```
const engine = {
  // method shorthand, with one
argument
  start(adverb) {
    console.log(`The engine starts up
${adverb}...`);
  },
  // anonymous arrow function
expression with no arguments
  sputter: () => {
    console.log('The engine
sputters...');
  },
};

engine.start('noisily');
engine.sputter();
```

JAVASCRIPT DESTRUCTURING ASSIGNMENT SHORTHAND SYNTAX

The JavaScript destructuring assignment is a shorthand syntax that allows object properties to be extracted into specific variable values.
It uses a pair of curly braces ({}) with property names on the left-hand side of an assignment to extract values from objects. The number of variables can be less than the total properties of an object.

```
const rubiksCubeFacts = {
  possiblePermutations:
'43,252,003,274,489,856,000',
  invented: '1974',
  largestCube: '17x17x17'
};
const {possiblePermutations, invented,
largestCube} = rubiksCubeFacts;
console.log(possiblePermutations); //
'43,252,003,274,489,856,000'
console.log(invented); // '1974'
console.log(largestCube); // '17x17x17'
```

SHORTHAND PROPERTY NAME SYNTAX FOR OBJECT CREATION

The shorthand property name syntax in JavaScript allows creating objects without explicitly specifying the property names (ie. explicitly declaring the value after the key). In this process, an object is created where the property names of that object match variables which already exist in that context. Shorthand property names populate an object with a key matching the identifier and a value matching the identifier's value.

```
const activity = 'Surfing';
const beach = { activity };
console.log(beach); // { activity:
'Surfing' }
```

THIS KEYWORD

The reserved keyword this refers to a method's calling object, and it can be used to access properties belonging to that object.

```
const cat = {
  name: 'Pipey',
  age: 8,
  whatName() {
    return this.name
  }
```

Here, using the this keyword inside the object function to refer to the cat object and access its name property.

```
};

console.log(cat.whatName());
// Output: Pipey
```

JAVASCRIPT FUNCTION THIS

Every JavaScript function or method has a this context. For a function defined inside of an object, this will refer to that object itself. For a function defined outside of an object, this will refer to the global object (window in a browser, global in Node.js).

```
const restaurant = {
  numCustomers: 45,
  seatCapacity: 100,
  availableSeats() {
    // this refers to the restaurant
object
    // and it's used to access its
properties
    return this.seatCapacity -
this.numCustomers;
  }
}
```

JAVASCRIPT ARROW FUNCTION THIS SCOPE

JavaScript arrow functions do not have their own this context, but use the this of the surrounding lexical context. Thus, they are generally a poor choice for writing object methods.
Consider the example code: loggerA is a property that uses arrow notation to define the function. Since data does not exist in the global context, accessing this.data returns undefined . loggerB uses method syntax. Since this refers to the enclosing object, the value of the data property is accessed as expected, returning “abc”.

```
const myObj = {
    data: 'abc',
    loggerA: () => {
console.log(this.data); },
    loggerB() { console.log(this.data);
},
};

myObj.loggerA();     // undefined
myObj.loggerB();     // 'abc'
```

GETTERS AND SETTERS INTERCEPT PROPERTY ACCESS

JavaScript getter and setter methods are helpful in part because they offer a way to intercept property

```
const myCat = {
```

access and assignment, and allow for additional actions to be performed before these changes go into effect.

```
  _name: 'Snickers',
  get name(){
    return this._name
  },
  set name(newName){
    //Verify that newName is a non-
empty string before setting as name
property
    if (typeof newName === 'string' &&
newName.length > 0){
      this._name = newName;
    } else {
      console.log("ERROR: name must be
a non-empty string");
    }
  }
}
```

JAVASCRIPT FACTORY FUNCTIONS

A JavaScript function that returns an object is known as a factory function. Factory functions often accept parameters in order to customize the returned object.

```
// A factory function that accepts
'name',
// 'age', and 'breed' parameters to
return
// a customized dog object.
const dogFactory = (name, age, breed)
=> {
  return {
    name: name,
    age: age,
    breed: breed,
    bark() {
      console.log('Woof!');
    }
  };
};
```

JAVASCRIPT GETTERS AND SETTERS RESTRICTED

JavaScript object properties are not private or protected. Since JavaScript objects are passed by reference, there is no way to fully prevent incorrect interactions with object properties.

One way to implement more restricted interactions with object properties is to use getter and setter methods.

Typically, the internal value is stored as a property with an identifier that matches the getter and setter method names, but begins with an underscore (_).

```
const myCat = {
  _name: 'Dottie',
  get name() {
    return this._name;
  },
  set name(newName) {
    this._name = newName;
  }
};

// Reference invokes the getter
console.log(myCat.name);

// Assignment invokes the setter
myCat.name = 'Yankee';
```

JAVASCRIPT BASICS

Including JavaScript in an HTML Page

```
<script type="text/javascript">
//JS code goes here
</script>
```

Call an External JavaScript File

```
<script src="myscript.js"></script><code></code>
```

INCLUDING COMMENTS

```
//
```

Single line comments

```
/* comment here */
```

Multi-line comments

VARIABLES

VAR, CONST, LET

`var`

The most common variable. Can be reassigned but only accessed within a function. Variables defined with var move to the top when code is executed.

`const`

Cannot be reassigned and not accessible before they appear within the code.

`let`

Similar to const, however, let variable can be reassigned but not re-declared.

DATA TYPES

```
var age = 23
```

Numbers

```
var x
```

Variables

```
var a = "init"
```

Text (strings)

```
var b = 1 + 2 + 3
```

Operations

```
var c = true
```

True or false statements

```
const PI = 3.14
```

Constant numbers

```
var name = {firstName:”John”, lastName:”Doe”}
```

Objects

OBJECTS

```
var person = { firstName:”John”, lastName:”Doe”, age:20,
nationality:”German”
};
```

ARRAYS

```
var fruit = [“Banana”, “Apple”, “Pear”];
```

ARRAY METHODS

`concat()`

Join several arrays into one

`indexOf()`

Returns the first position at which a given element appears in an array

`join()`

Combine elements of an array into a single string and return the string

`lastIndexOf()`

Gives the last position at which a given element appears in an array

`pop()`

Removes the last element of an array

`push()`

Add a new element at the end

`reverse()`

Reverse the order of the elements in an array

`shift()`

Remove the first element of an array

`slice()`

Pulls a copy of a portion of an array into a new array of 4 24

`sort()`

Sorts elements alphabetically

`splice()`

Adds elements in a specified way and position

`toString()`

Converts elements to strings

`unshift()`

Adds a new element to the beginning

`valueOf()`

Returns the primitive value of the specified object

OPERATORS

BASIC OPERATORS

```
+     Addition
-     Subtraction
*     Multiplication
/     Division
(..)  Grouping operator
%     Modulus (remainder)
++    Increment numbers
--    Decrement numbers
```

COMPARISON OPERATORS

```
==    Equal to
===   Equal value and equal type
!=    Not equal
!==   Not equal value or not equal type
>     Greater than
<     Less than
>=    Greater than or equal to
<=    Less than or equal to
?     Ternary operator
```

LOGICAL OPERATORS

```
&&    Logical and
||    Logical or
!     Logical not
```

BITWISE OPERATORS

```
&     AND statement
|     OR statement
```

```
~     NOT
^     XOR
<<    Left shift
>>    Right shift
>>>   Zero fill right shift
```

FUNCTIONS

```
function name(parameter1, parameter2, parameter3) {
// what the function does
}
```

OUTPUTTING DATA

`alert()`

Output data in an alert box in the browser window

`confirm()`

Opens up a yes/no dialog and returns true/false depending on user click

`console.log()`

Writes information to the browser console, good for debugging purposes

`document.write()`

Write directly to the HTML document

`prompt()`

Creates an dialogue for user input

GLOBAL FUNCTIONS

`decodeURI()`

Decodes a Uniform Resource Identifier (URI) created by encodeURI or similar

`decodeURIComponent()`

Decodes a URI component

`encodeURI()`

Encodes a URI into UTF-8

`encodeURIComponent()`

Same but for URI components

`eval()`

Evaluates JavaScript code represented as a string

`isFinite()`

Determines whether a passed value is a finite number

`isNaN()`

Determines whether a value is NaN or not

`Number()`

Returns a number converted from its argument

`parseFloat()`

Parses an argument and returns a floating point number

`parseInt()`

Parses its argument and returns an integer

LOOPS

```
for (before loop; condition for loop; execute after loop) {
// what to do during the loop
}
```

`for`

The most common way to create a loop in Javascript

`while`

Sets up conditions under which a loop executes

`do while`

Similar to the while loop, however, it executes at least once and performs a check at the end to see if the condition is met to execute again

`break`

Used to stop and exit the cycle at certain conditions

`continue`

Skip parts of the cycle if certain conditions are met of 7 24

IF - ELSE STATEMENTS

```
if (condition) {
// what to do if condition is met
} else {
// what to do if condition is not met
}
```

STRINGS

```
var person = "John Doe";
```

ESCAPE CHARACTERS

```
\'   – Single quote
\"   – Double quote
\\   – Backslash
\b   – Backspace
\f   – Form feed
\n   – New line
\r   – Carriage return
\t   – Horizontal tabulator
\v   – Vertical tabulator
```

STRING METHODS

`charAt()`

Returns a character at a specified position inside a string

`charCodeAt()`

Gives you the unicode of character at that position

`concat()`

Concatenates (joins) two or more strings into one

`fromCharCode()Returns a string created from the specified sequence of UTF-16 code units`

`indexOf()`

Provides the position of the first occurrence of a specified text within a string

`lastIndexOf()`

Same as indexOf() but with the last occurrence, searching backwards

`match()`

Retrieves the matches of a string against a search pattern

`replace()`

Find and replace specific text in a string

`search()`

Executes a search for a matching text and returns its position

`slice()`

Extracts a section of a string and returns it as a new string

`split()`

Splits a string object into an array of strings at a specified position

`substr()`

Similar to slice() but extracts a substring depended on a specified number of characters

`substring()`

Also similar to slice() but can't accept negative indices

`toLowerCase()`

Convert strings to lowercase

`toUpperCase()`

Convert strings to uppercase

`valueOf()`

Returns the primitive value (that has no properties or methods) of a string object

REGULAR EXPRESSIONS

PATTERN MODIFIERS

e — Evaluate replacement

i — Perform case-insensitive matching g — Perform global matching

m — Perform multiple line matching s — Treat strings as single line

x — Allow comments and whitespace in pattern U — Non Greedy pattern

BRACKETS

[abc] Find any of the characters between the brackets [^abc] Find any character not in the brackets

[0-9] Used to find any digit from 0 to 9

[A-z] Find any character from uppercase A to lowercase z (a|b|c) Find any of the alternatives separated with |

METACHARACTERS

. — Find a single character, except newline or line terminator

\w — Word character

\W — Non-word character

\d — A digit

\D — A non-digit character

\s — Whitespace character

\S — Non-whitespace character

\b — Find a match at the beginning/end of a word

\B — A match not at the beginning/end of a word

\0 — NUL character

\n — A new line character

\f — Form feed character

\r — Carriage return character

\t — Tab character

\v — Vertical tab character

\xxx — The character specified by an octal number xxx

\xdd — Character specified by a hexadecimal number dd

\uxxxx — The Unicode character specified by a hexadecimal number xxxx

QUANTIFIERS

n+ — Matches any string that contains at least one n

n* — Any string that contains zero or more occurrences of n n? — A string that contains zero or one occurrences of n n{X} — String that contains a sequence of X n's

n{X,Y} — Strings that contains a sequence of X to Y n's

n{X,} — Matches any string that contains a sequence of at least X n's

n$ — Any string with n at the end of it

^n — String with n at the beginning of it

?=n — Any string that is followed by a specific string n

?!n — String that is not followed by a specific string n

NUMBERS AND MATH

NUMBER PROPERTIES

`MAX_VALUE`

The maximum numeric value representable in JavaScript

`MIN_VALUE`

Smallest positive numeric value representable in JavaScript

`NaN`

The "Not-a-Number" value

`NEGATIVE_INFINITY`

The negative Infinity value

`POSITIVE_INFINITY`

Positive Infinity value

NUMBER METHODS

`toExponential()`

Returns a string with a rounded number written as exponential notation

`toFixed()`

Returns the string of a number with a specified number of decimals

`toPrecision()`

String of a number written with a specified length

`toString()`

Returns a number as a string

`valueOf()`

Returns a number as a number

MATH PROPERTIES

```
E          Euler's number
LN2        The natural logarithm of 2
LN10       Natural logarithm of 10
LOG2E      Base 2 logarithm of E
LOG10E     Base 10 logarithm of E
PI         The number PI
SQRT1_2    Square root of 1/2
SQRT2      The square root of 2
```

MATH METHODS

`abs(x)`

Returns the absolute (positive) value of x

`acos(x)`

The arccosine of x, in radians

`asin(x)`

Arcsine of x, in radians

`atan(x)`

The arctangent of x as a numeric value

`atan2(y,x)`

Arctangent of the quotient of its arguments

`ceil(x)`

Value of x rounded up to its nearest integer

`cos(x)`

The cosine of x (x is in radians)

`exp(x)`

Value of Ex

`floor(x)`

The value of x rounded down to its nearest integer

`log(x)`

The natural logarithm (base E) of x

`max(x,y,z,...,n)`

Returns the number with the highest value

`min(x,y,z,...,n)`

Same for the number with the lowest value

`pow(x,y)`

X to the power of y

`random()`

Returns a random number between 0 and 1

`round(x)`

The value of x rounded to its nearest integer

`sin(x)`

The sine of x (x is in radians)

`sqrt(x)`

Square root of x

`tan(x)`

The tangent of an angle

DEALING WITH DATES

SETTING DATES

`Date()`

Creates a new date object with the current date and time

```
Date(2017, 5, 21, 3, 23, 10, 0)
```

Create a custom date object. The numbers represent year, month, day, hour, minutes, seconds, milliseconds. You can omit anything you want except for year and month.

```
Date("2017-06-23")
```

Date declaration as a string

PULLING DATE AND TIME VALUES

`getDate()`

Get the day of the month as a number (1-31)

`getDay()`

The weekday as a number (0-6)

`getFullYear()`

Year as a four digit number (yyyy)

`getHours()`

Get the hour (0-23)

`getMilliseconds()`

The millisecond (0-999)

`getMinutes()`

Get the minute (0-59)

`getMonth()`

Month as a number (0-11)

`getSeconds()`

Get the second (0-59)

`getTime()`

Get the milliseconds since January 1, 1970

`getUTCDate()`

The day (date) of the month in the specified date according to universal time (also available for day, month, fullyear, hours, minutes etc.)

`parse`

Parses a string representation of a date, and returns the number of milliseconds since January 1, 1970

SET PART OF A DATE

`setDate()`

Set the day as a number (1-31)

`setFullYear()`

Sets the year (optionally month and day)

`setHours()`

Set the hour (0-23)

`setMilliseconds()`

Set milliseconds (0-999)

`setMinutes()`

Sets the minutes (0-59)

`setMonth()`

Set the month (0-11)

`setSeconds()`

Sets the seconds (0-59)

`setTime()`

Set the time (milliseconds since January 1, 1970)

`setUTCDate()`

Sets the day of the month for a specified date according to universal time (also available for day, month, fullyear, hours, minutes etc.)

DOM NODE

NODE PROPERTIES

`attributes`

Returns a live collection of all attributes registered to and element

`baseURI`

Provides the absolute base URL of an HTML element

`childNodes`

Gives a collection of an element's child nodes

`firstChild`

Returns the first child node of an element

`lastChild`

The last child node of an element

`nextSibling`

Gives you the next node at the same node tree level

`nodeName`

Returns the name of a node

`nodeType`

Returns the type of a node

`nodeValue`

Sets or returns the value of a node

`ownerDocument`

The top-level document object for this node

`parentNode`

Returns the parent node of an element

`previousSibling`

Returns the node immediately preceding the current one

`textContent`

Sets or returns the textual content of a node and its descendants

NODE METHODS

`appendChild()`

Adds a new child node to an element as the last child node

`cloneNode()`

Clones an HTML element

`compareDocumentPosition()`

Compares the document position of two elements

`getFeature()`

Returns an object which implements the APIs of a specified feature

`hasAttributes()`

Returns true if an element has any attributes, otherwise false

`hasChildNodes()`

Returns true if an element has any child nodes, otherwise false

`insertBefore()`

Inserts a new child node before a specified, existing child node

`isDefaultNamespace()`

Returns true if a specified namespaceURI is the default, otherwise false

`isEqualNode()`

Checks if two elements are equal

`isSameNode()`

Checks if two elements are the same node

`isSupported()`

Returns true if a specified feature is supported on the element

`lookupNamespaceURI()`

Returns the namespaceURI associated with a given node

`lookupPrefix()`

Returns a DOMString containing the prefix for a given namespaceURI, if present

`normalize()`

Joins adjacent text nodes and removes empty text nodes in an element

`removeChild()`

Removes a child node from an element

`replaceChild()`

Replaces a child node in an element

ELEMENT METHODS

`getAttribute()`

Returns the specified attribute value of an element node

`getAttributeNS()`

Returns string value of the attribute with the specified namespace and name

`getAttributeNode()`

Gets the specified attribute node

`getAttributeNodeNS()`

Returns the attribute node for the attribute with the given namespace and name

`getElementsByTagName()`

Provides a collection of all child elements with the specified tag name

`getElementsByTagNameNS()`

Returns a live HTMLCollection of elements with a certain tag name belonging to the given namespace

`hasAttribute()`

Returns true if an element has any attributes, otherwise false

`hasAttributeNS()`

Provides a true/false value indicating whether the current element in a given namespace has the specified attribute

`removeAttribute()`

Removes a specified attribute from an element

`removeAttributeNS()`

Removes the specified attribute from an element within a certain namespace

`removeAttributeNode()`

Takes away a specified attribute node and returns the removed node

`setAttribute()`

Sets or changes the specified attribute to a specified value

`setAttributeNS()`

Adds a new attribute or changes the value of an attribute with the given namespace and name

`setAttributeNode()`

Sets or changes the specified attribute node

`setAttributeNodeNS()`

Adds a new namespaced attribute node to an element

WORKING WITH THE BROWSER

WINDOW PROPERTIES

`closed`

Checks whether a window has been closed or not and returns true or false

`defaultStatus`

Sets or returns the default text in the statusbar of a window

`document`

Returns the document object for the window

`frames`

Returns all <iframe> elements in the current window

`history`

Provides the History object for the window

`innerHeight`

The inner height of a window's content areainnerWidth

The inner width of the content area

`length`

Find out the number of <iframe> elements in the window

`location`

Returns the location object for the window

`name`

Sets or returns the name of a window

`navigator`

Returns the Navigator object for the window

`opener`

Returns a reference to the window that created the window

`outerHeight`

The outer height of a window, including toolbars/ scrollbars

`outerWidth`

The outer width of a window, including toolbars/ scrollbars

`pageXOffset`

Number of pixels the current document has been scrolled horizontally

`pageYOffset`

Number of pixels the document has been scrolled vertically

`parent`

The parent window of the current window

`screen`

Returns the Screen object for the window

`screenLeft`

The horizontal coordinate of the window (relative to screen)

`screenTop`

The vertical coordinate of the window

`screenX`

Same as screenLeft but needed for some browsers

`screenY`

Same as screenTop but needed for some browsers

`self`

Returns the current window

`status`

Sets or returns the text in the statusbar of a window

`top`

Returns the topmost browser window

WINDOW METHODS

`alert()`

Displays an alert box with a message and an OK button

`blur()`

Removes focus from the current window

`clearInterval()`

Clears a timer set with setInterval()

`clearTimeout()`

Clears a timer set with setTimeout()

`close()`

Closes the current window

`confirm()`

Displays a dialogue box with a message and an OK and Cancel button

`focus()`

Sets focus to the current window

`moveBy()`

Moves a window relative to its current position

`moveTo()`

Moves a window to a specified position

`open()`

Opens a new browser window

`print()`

Prints the content of the current window

`prompt()`

Displays a dialogue box that prompts the visitor for input

`resizeBy()`

Resizes the window by the specified number of pixels

`resizeTo()`

Resizes the window to a specified width and height

`scrollBy()`

Scrolls the document by a specified number of pixels

`scrollTo()`

Scrolls the document to specific coordinates

`setInterval()`

Calls a function or evaluates an expression at specified intervals

`setTimeout()`

Calls a function or evaluates an expression after a specified interval

`stop()`

Stops the window from loading

SCREEN PROPERTIES

`availHeight`

Returns the height of the screen (excluding the Windows Taskbar)

`availWidth`

Returns the width of the screen (excluding the Windows Taskbar)

`colorDepth`

Returns the bit depth of the color palette for displaying images

`height`

The total height of the screen

`pixelDepthThe color resolution of the screen in bits per pixel`

`width`

The total width of the screen

EVENTS

MOUSE

`onclick`

The event occurs when the user clicks on an element

`oncontextmenu`

User right-clicks on an element to open a context menu

`ondblclick`

The user double-clicks on an element

`onmousedown`

User presses a mouse button over an element

`onmouseenter`

The pointer moves onto an element

`onmouseleave`

Pointer moves out of an element

`onmousemove`

The pointer is moving while it is over an element

`onmouseover`

When the pointer is moved onto an element or one of its children

`onmouseout`

User moves the mouse pointer out of an element or one of its children

`onmouseup`

The user releases a mouse button while over an element

KEYBOARD

`onkeydown`

When the user is pressing a key down

`onkeypress`

The moment the user starts pressing a key

`onkeyup`

The user releases a key

FRAME

`onabort`

The loading of a media is aborted

`onbeforeunload`

Event occurs before the document is about to be unloaded

`onerror`

An error occurs while loading an external file

`onhashchange`

There have been changes to the anchor part of a URL

`onload`

When an object has loaded

`onpagehide`

The user navigates away from a webpage

`onpageshow`

When the user navigates to a webpage

`onresize`

The document view is resized

`onscroll`

An element's scrollbar is being scrolled

`onunload`

Event occurs when a page has unloaded

FORM

`onblur`

When an element loses focus

`onchange`

The content of a form element changes (for <input>, <select>and <textarea>)

`onfocus`

An element gets focus

`onfocusin`

When an element is about to get focus

`onfocusout`

The element is about to lose focus

`oninput`

User input on an element

`oninvalid`

An element is invalid

`onreset`

A form is reset

`onsearch`

The user writes something in a search field (for <input="search">)

`onselect`

The user selects some text (for <input> and <textarea>)

`onsubmit`

A form is submitted

DRAG

`ondrag`

An element is dragged

`onpaste`

A user pastes content in an element

MEDIA

`onabort`

Media loading is aborted

`oncanplay`

The browser can start playing media (e.g. a file has buffered enough)

`oncanplaythrough`

When browser can play through media without stopping

`ondurationchange`

The duration of the media changes

`onended`

The media has reached its end

`onerror`

Happens when an error occurs while loading an external file

`onseeked`

User is finished moving/skipping to a new position in the media

`onseeking`

The user starts moving/skipping

`onstalled`

The browser is trying to load the media but it is not available

`onsuspend`

Browser is intentionally not loading media

`ontimeupdate`

The playing position has changed (e.g. because of fast forward)

`onvolumechange`

Media volume has changed (including mute)

`onwaiting`

Media paused but expected to resume (for example, buffering)

ANIMATION

`animationend`

A CSS animation is complete

`animationiteration`

CSS animation is repeated

`animationstart`

CSS animation has started

OTHER

`transitionend`

Fired when a CSS transition has completed

`onmessage`

A message is received through the event source

`onoffline`

Browser starts to work offline

`ononline`

The browser starts to work online

`onpopstate`

When the window's history changes

`onshow`

A <menu> element is shown as a context menu

`onstorage`

A Web Storage area is updated

`ontoggle`

The user opens or closes the <details> element

`onwheel`

Mouse wheel rolls up or down over an element

`ontouchcancel`

Screen touch is interrupted

`ontouchend`

User finger is removed from a touch screen

`ontouchmove`

A finger is dragged across the screen

ERRORS

`try`

Lets you define a block of code to test for errors

`catch`

Set up a block of code to execute in case of an error

`throw`

Create custom error messages instead of the standard JavaScript errors

`finally`

Lets you execute code, after try and catch, regardless of the result

ERROR NAME VALUES

`name`

Sets or returns the error name

`message`

Sets or returns an error message in string from

`EvalError`

An error has occurred in the eval() function

`RangeError`

A number is “out of range”

`ReferenceError`

An illegal reference has occurred

`SyntaxError`

A syntax error has occurred

`TypeError`

A type error has occurred

`URIError`

An encodeURI() error has occurred

JOIN ZERO TO FULL STACK HERO TO LEARN MORE,
VISIT WWW.PAPAREACT.COM

Sonny Sangha

ssssangha