JAVASCRIPT

A Comprehensive Beginner's Guide to Mastering JavaScript Programming Incrementally

RYAN ROFFE

JavaScript

A Comprehensive Beginner's Guide to Mastering JavaScript Programming Incrementally

Ryan roffe

TABLE OF CONTENTS

Introduction

JavaScript is a distinguished and deciphered programming language that adheres to the ECMAScript specification. The programming language consists of high-powered typing, the curly bracket syntax, top-quality functions, and prototype-based object-orientation. Accompanied by HTML and CSS, JavaScript is one of the significant technologies of the World Wide Web. JavaScript facilitates interchangeable web pages and is a vital part of web applications. Almost every website utilizes JavaScript, and all the significant web browsers consist of the JavaScript engine to function correctly. Brendan Eich developed JavaScript in 1995 when he was with Netscape Communications. JavaScript aids functional, event-driven, prototype-based and object-oriented programming styles. JavaScript upgrades web user interface by affirming activities taken on the client-side by the user. You can insert JavaScript engines into different types of host software, which includes databases and web servers, and non-web related programs like word processors.

The concepts of JavaScript are explained further in this book with the goal to help you learn and understand JavaScript language without pressure. The knowledge of HTML, text editor, web browser, and CSS are all that's needed to learn JavaScript. To get the learning process started, one of the essential tools is a text editor, and it is required to write codes. You'll also need a browser to unveil your developed web pages. There are different types of text editors such as the Sublime Text,

Notepad++ and browsers such as Firefox, Google Chrome, and so on.

Chapter 1: What is JavaScript

JavaScript, abbreviated JS, is a high-level programming language introduced to add specific programs to web pages, and it has been adopted by all major web browsers. JavaScript can be used to build interactive web applications to function appropriately without reloading every page per-action. JavaScript is used to create different forms of activities within web pages and is one of the essential components of the World Wide Web (www), which are HTML (Hypertext Markup Language) and CSS (Cascading Style Sheets). JavaScript and HTML are used to develop web pages. JavaScript brings a page to life by adding special effects such as sliders, pop-ups, form validations, etc. CSS determines the color intensity, image size, background colors, typeface, font size, etc.

Client-Side JavaScript

Client-Side JavaScript is the usual form of JavaScript language. The script must be inserted or referenced by an HTML document so that the browser can interpret the code. It enables web pages to include interactive programs with the user, develop HTML content dynamically, and command the browser. Users use JavaScript code when they want to submit forms and also determine if all entries are valid before it transfers them to the server.

Advantages of JavaScript

· With JavaScript, users can organize input before getting the page sent to the server, which automatically reduces loads on the server.

· JavaScript enables swift response to page visitors. The page does not have to reload before visitors can see if there was an error in typing.

· JavaScript is used to build a reactive interface that gives a reaction when the mouse hovers over them.

Limitations of JavaScript

JavaScript programming language lacks the following essential features. They are:

· For security reasons, client-side JavaScript does not read or write files

· JavaScript for networking applications

· JavaScript does not contain multiprocessor capabilities

JavaScript Development Tools

You do not need an expensive development tool to write JavaScript codes. You can write with a simple text editor such as Microsoft FrontPage, Macromedia Dreamweaver MX, Macromedia HomeSite 5, etc.

Javascript Placement

JavaScript code is inserted anywhere in an HTML document. However, you want to insert JavaScript code in an HTML file like this:

The Script within <head> </head> part.

The Script within <body> </body> part.

The Script within <body> </body> and <head> </head> part.

Include external file Script in the <head>...</head> part.

JavaScript in <head>...</head> Section

In some cases, when you want the script to run on a special event, such as an action when a button clicked, place the script in the head section like this:

```
<html>
<head>
<script type="text/JavaScript">
<!--
function sayHi() {
alert("Hello World")
}
//-->
</script>
</head>
<body>
Tap here for result
<input type="button" onclick="sayHi()" value="Say Hi" />
</body>
</html>
```

JavaScript in <body>...</body> Section

Sometimes you want a script to run immediately and to create script on the content page, insert the script within the <body> section. This is what the code should look like:

```
<html>
<head>
</head>
```

```
<body>
<script type="text/JavaScript">
<!--
document.write("Hello World")
//-->
</script>
<p>This is web page body </p>
</body>
</html>
```

JavaScript in <body> and <head> Sections

Insert your JavaScript code in <head> and <body> section like this:

```
<html>
<head>
<script type="text/JavaScript">
<!--
function sayHi() {
alert("Hello World")
}
//-->
</script>
</head>
<body>
<script type="text/JavaScript">
<!--
document.write("Hello World")
//-->
</script>
<input type="button" onclick="sayHi()" value="Say Hi" />
```

```
</body>
</html>
```

JavaScript in External Files

To avoid the use of identical JavaScript code repetition, the language enables you to create an external file and store JavaScript codes and then integrate the external file into the HTML files. The sample below displays how to integrate an external JavaScript file within the HTML code utilizing the script tag and src attribute.

```
<html>
<head>
<script type="text/javascript" src="filename.js" ></script>
</head>
<body>
.....
</body>
</html>
```

The external file source file should be saved with an extension .js.

Summary

In this first chapter, we explained the origination of JavaScript and its placement into internal and external files. We discussed the use of JavaScript to build interactive web applications to perform appropriately without having to reload every page per-action.

Exercise

How do you insert JavaScript code into the head, body and include external file script in an HTML document?

Solution

The Script for the head part is <head> </head>.

The Script for the body part is <body> </body>.

The Script within the body is <body> </body> and <head> </head>.

The Script to Include external file Script in the <head>...</head>.

Chapter 2: Values, Types, and Operators

In the world of computers, there is data. You can create new data, read data and change data, which are all stored as a look-alike long succession of bits. Define bits as zeros and ones that take a strong or weak signal, and high or low electrical charge from inside the computer. All data and pieces of information are described as a succession of zeros and ones and represented in bits.

Values

Take a deep breath and think of an ocean of bits. The latest PCs contains more than 30 billion bits in its data storage. We use bits to create values. The computer can function correctly because every bit of information is split into values. Every value consists of a type that influences its role, and values can be numbers, text or functions, etc. To generate value, you need to invoke its name, and it appears from where it was stored.

Arithmetic

Arithmetic is the major thing to do with numbers. The multiplication, addition, and subtraction of more than one number to produce another number is an arithmetic operation. This is an example of what they look like in JavaScript:

```
100 + 4 * 11
```

The + and * symbols are called operators, the first means addition while the other means multiplication. An operator inserted between two values will produce another value. The - operator is for subtraction and the / operator is for the division. If operators show together without parentheses, the precedence of the operators decides the way they are applied. If several operators with the same precedence show right next to each other like 1 - 2 + 1, apply them left to right: (1 - 2) + 1.

Special numbers

JavaScript consists of three unique values that do not act like numbers but are regarded as numbers. Infinity and -Infinity are the first two, which mean the positive and negative infinities, and the last value is the NaN. NaN says “not a number,” although it is a value of the number type.

Strings

A string is the succession of numbers. The string is the next data type, and they represent text. Strings confine their content in quotes.

`Down the walkway path. `

"On top of the roof."

'I am at home.'

Quotes are used in different types like the double quotes, single quotes, or the backticks to mark strings. It is essential that the strings match. The elements within the quotes create a string value by JavaScript. JavaScript uses the Unicode standard to assign a number to every character needed, including Arabic, Armenian, Japanese, etc. You cannot subtract, multiply, or divide

strings but you can use the + operator, which will not add but concatenates two strings together. Concatenation means to glue strings together.

Unary operators

Symbols do not represent all the operators. You can write some operators in words. A clear example is a type of operator, and this operator creates a string value with the name of the typeof its attached value.

```
console.log(typeof 4.5)
// → number
console.log(typeof "x")
// → string
```

The second displayed operator is called the binary operator because they use two values, while operators that use one value are the unary operator.

```
console.log(- (10 - 2))
// → -8
```

Boolean values

Boolean is a value that differs only between two possibilities such as “on” and “off” and so on. The Boolean consists of only true and false.

Comparison

Comparison is a way to create Boolean values:

```
console.log(3 > 2)
// → true
```

```
console.log(3 < 2)
// → false
```

The > and < characters are the signs that represent “is greater than” and “is less than,” accordingly. You can use binary operators in a Boolean value that determines if the contained value if true or false.

You can compare strings in the same manner.

```
console.log("Aardvark" < "Zoroaster")
// → true
```

You can instruct strings in alphabetical order, uppercase letters are often “less” than lowercase, so "Z" < "a," and non-alphabetic characters (! -, and so on) are also present in the ordering. When JavaScript want to compare strings, it recognizes characters from left to right, differentiating the Unicode codes individually.

Here are other related operators <= (less than or equal to), >= (greater than or equal to), == (equal to), and!= (not equal to).

```
console.log("Itchy" != "Scratchy")
// → true
console.log("Apple" == "Orange")
// → false
```

In JavaScript, there is only one value that is not equal to itself, which is the NaN (“not a number”).

```
console.log(NaN == NaN)
// → false
```

Logical operator

JavaScript endorses only three operators that you can apply to Boolean values, they are and, or, and not. The && operator signifies logical or and its results depend on where the values imputed are true or false.

```
console.log(true && false)
// → false
console.log(true && true)
// → true
```

The || operator indicates logical or. This operator outputs true if the given value is true.

```
console.log(false || true)
// → true
console.log(false || false)
// → false
```

An exclamation mark (!) indicates Not. It is an operator that overturns the set value, and it changes from true to false and false to true.

The || consists of the lowest precedence of all operators, then &&, the comparison operators (>, ==, etc.), and so on. The example below states that parentheses are necessary:

```
1 + 1 == 2 && 10 * 10 > 50
```

Empty values

Null and undefined are the only two types of special values that are used to indicate the non-appearance of an important value. They contain zero data.

Automatic type conversion

Earlier I said that JavaScript accepts practically all given programs, including programs with odd behaviors. Automatic type conversion illustrates in the following expressions:

```
console.log(8 * null)
// → 0
console.log("5" - 1)
// → 4
console.log("5" + 1)
// → 51
console.log("five" * 2)
// → NaN
console.log(false == 0)
// → true
```

When you assign an operator the wrong value, JavaScript silently returns that value to the exact type it requires using the type coercion rule. In the first expression, the null turns to 0, and the 5 in the second remains 5 (from string to number). In the third expression, there was a string concatenation before the numeric addition, which converts the 1 to 1 (from number to a string). When odd numbers such as "five" or undefined changes to the number, it gets the value of NaN. If you want to differentiate between values of the same type using ==, the output should be true if the values are similar except NaN. If you want to test if a value contains a real value, use the == (or! =) operator to compare it. To avert unexpected type conversions, use the three-character comparison operators.

Short-circuiting of logical operators

The && and || are called the logical operators. They are used to hold several types of values in a specific way. They change the values contained in the left side to Boolean type to decide, although it depends on the operators and the type of generated result, but will always reinstate the left or right-hand value. The || operator sends back value to the left when it can be changed to true and will reinstate the value to the right.

```
console.log(null || "user")
// → user
console.log("Kate" || "user")
// → Kate
```

This function is used to return values to its default value and placed within an empty value as a replacement. Strings and numbers to Boolean value conversion rules indicate that 0, NaN, and empty string ('') count as false while other values are true. Therefore 0 || -1 outputs -1, and "" || "!?" yields "!?". The && operator operates identically but the other way around. When values to the left can be changed to false, return the value, or it sends the value to the right. Both operators evaluate the value to the right only when it is required. For example, we have the following values set as true || X, the value of X will be true and will not consider it. The same rule applies to the false && X, which the x is false and will overlook it. You can call this process the short-circuit evaluation.

Summary

This chapter looks at the four types of JavaScript values, which are strings, numbers, undefined values and Booleans. These values are developed by inserting their names as true, null or value (13, "ABC"). Operators can integrate and change values. We looked at binary operators for arithmetic (+, -, *, /, and %), string concatenation (+), comparison (==, !=, ===, !==, <, >, <=, >=), and logic (&&, ||), and also various unary operators (- to nullify a number, ! to nullify logically, and type of to search for a value's type) and a ternary operator (?:) to choose one of two values depending on a third value. You will get sufficient information to use JavaScript like a small calculator, and you will improve in the following chapters.

Exercise

Write a JavaScript practice to build a variable through a user-defined name.

Solution

HTML Code:

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>Create a variable using a user-defined name</title>
</head>
<body>

</body>
</html>
```

JavaScript Code:

```javascript
var var_name = 'abcd';
var n = 120;
this[var_name] = n;
console.log(this[var_name])
```

Chapter 3: Program Structure

We will begin the expansion of the JavaScript programming language commands beyond sentence fragments and nouns.

Expressions and statements

A block of code that manufactures a value is called an expression. A single value is written precisely as 22, or psychoanalysis is called an expression. An expression within a binary operator set to two expressions or within parentheses is an expression. An expression that can accommodate other expressions comparably to how the human language translates. It enables us to develop expressions that narrate the arbitrarily complex computations. When an expression is comparable to a sentence fragment, the JavaScript statement will complete the sentence. A list of statements is called a program, and the most accessible type of statement is a line of code with a semicolon ending it.

For instance:

```
1;
!false;
```

With this function, a statement can stand independently, and it can add a feature that changes colors occasionally to a screen or modify the inner state of the machine influencing the following statements after it.

Bindings

JavaScript uses binding of variables to hold values, for instance:

```
Get money = 5 * 5;
```

Get is the keyword in that statement, and it means that the sentence will specify a binding together with the binding name and value can be attached using the = operator and an expression. Use a defined binding as an expression, an expression containing the binding value. Here is an example below:

```
let ten = 10;
console.log(ten * ten);
// → 100
```

The = operator is used on existing bindings to remove the binding from a set value and point them to a new one.

```
let mood = "light";
console.log(mood);
// → light
mood = "dark";
console.log(mood);
// → dark
```

Defining a binding without assigning a value will result in nothing to hold. Therefore, asking for the value of a binding produce the undefined value. You can define multiple binding by a single statement divided by commas.

```
let one = 1, two = 2;
console.log(one + two);
// → 3
```

Use the var and const words to develop bindings familiarly:

```
var name = "Ayad";
```

```
const greeting = "Hello ";
console.log(greeting + name);
// → Hello Ayad
```

The word const means constant. It describes a resolute binding, which points to its set value for its lifetime. These words are often used to declare a name to a value to enable easy reference later on.

Binding names

Although there are some reserved words like const, class, default, break, continue, delete, do, else yet any word can be called a binding name, it can also consist of digits but do not begin the statement with a figure. A binding name can also contain dollar signs ($) or underscores (_) but do not entertain any other special characters or punctuation.

The environment

The group of binding and values existing at a stipulated time is known as the environment. When you launch a program, the environment holds the language standard binding and often contains the binding that creates interaction with system surroundings. For instance, a browser carries functions that communicate with the launched website as well as read the keyboard and mouse input.

Functions

A bit of program enclosed in value is called a function, and these values are added to launch the wrapped program. For instance, a function that displays a small dialog box for user input.

Prompt ("Enter passcode");

The process of effecting a function is described as a calling, invoking, or applying. Call a function by inserting parentheses following an expression that provides a function value. Set the parenthesis value to the program within the function. Define values set to functions as arguments.

The console.log function

In the above samples, we used the console.log to output values. All major browsers utilize the console.log function to state out its arguments to the device that outputs the text. In modern browsers, the output is often in the JavaScript console that is invisible by default; you can tap the F12 command on your keyboard or the command-option-I on Mac. Binding names do not accommodate period characters, but console.log does because it is an expression that reclaims the log property from the console binding value.

Return values

Functions are used to produce side effects. They can also provide values that do not need side effects. For instance, the function Calculate.max will take a sum of number arguments and returns the greatest.

```
console.log(Calculate.max(2, 4));
// → 4
```

JavaScript regards anything that provides value as an expression, which enables function calls to attach into the substantial expression. Let’s call the Calculate.min, which is a direct opposite to Calculate.max:

```
console.log(Math.min(2, 4) + 100);
// → 102
```

Control flow

When more than one statement is within a program, they execute in a story form from the beginning of the code to the end. This type of program consists of two types of statement; the first demand numbers from the user while the second displays the square of the number and executes immediately after the first.

```
let theNumber = Number(prompt("Select a digit "));
console.log("Your digit is the square root of " + theNumber * theNumber);
```

A value is changed to a number by the function number and outputs a string value.

Conditional execution

The keyboard is used to develop conditional executions in JavaScript. You may want some code to execute if, and only if, a specific condition is positive. Let us display the square of the input if only it is a number.

```
let theNumber = Number(prompt("Select a digit "));
if (! Number.isNaN(theNumber)) {
console.log("Your digit is the square root of " +
theNumber * theNumber);
}
```

The if keyword performs or evades a statement determined by the Boolean expression value. Type the determining expression

after the keyboard within the parentheses accompanied by the statement to accomplish.

The Number.isNaN function is a high-level JavaScript function that outputs only true if the statement is declared as NaN. The number function returns NaN when an assigned string is not a valid number. Statements after the if statement is enclosed in braces ({and}). Braces are used to categorize different numbers of statement within an individual statement.


```
if (1 + 1 == 2) console.log("It's true");
// → It's true
```

The else keyword can be utilized together with the if statement to provide two different execution paths.

```
let theNumber = Number (prompt("Select a digit"));
if (! Number.isNaN(theNumber)) {
console.log("Your digit is the square root of " +
theNumber * theNumber);
} else {
console.log("Hey. Why didn't you give me a number?");
}
```

If there are more than two paths to choose from, you can “chain” multiple

if/else pairs together. Below is an example:

```
let num = Number (prompt("Pick a number"));
if (num < 10) {
console.log("Small");
```

```
} else if (num < 100) {
console.log("Medium");
} else {
console.log("Large");
}
```

The program confirms if the num is less than 10, and if it is, it selects that branch, and then displays "Small". But if the num is greater than 10, it selects the else branch which consists of a second if statement of its own.

while and do loops

This is the way to write a program that displays all the even numbers from 0 to 12.

```
console.log(0);
console.log(2);
console.log(4);
console.log(6);
console.log(8);
console.log(10);
console.log(12);
```

A way to run a clock of code multiple times is described as a loop.

The looping control flow enables the user to return to certain points in written programs and redo it with the current state of the program. Integrate this with a binding that enumerates as follows:

```
let number = 0;
while (number <= 12) {
console.log(number);
```

```
number = number + 2;
}
// → 0
// → 2
// ... etcetera
```

A statement that begins with the keyword and at the same time producing a loop. The keyword while accompanied by an expression in parentheses followed by a statement like the if statement. The loop becomes continuous until the expression provides a value that states true when translated to Boolean. Each time the loop repeats, numbers adopt a number twice of their previous value. Let us write a program that evaluates and display the value of 210 (2 raised to the power of 10th). We will utilize two bindings. One to keep an eye on our result and the other to calculate how many times the value of two multiplied by the result. It multiplies until the second binding reaches 10:

```
let result = 1;
let counter = 0;
while (counter < 10) {
result = result * 2;
counter = counter + 1;
}
console.log(result);
// → 1024
```

A do loop is a command structure related to a while loop. A do loop always take place at least once in a statement, and then it

begins to check if it ends after the first execution only. Follow the following steps:

```
let yourName;
do {
yourName = prompt("Who are you?");
} while (! yourName);
console.log(yourName);
```

for loops

A lot of loops follow the while loop pattern, creating a counter binding to track the loop's progress. Then a while loop with a test expression to see if the counter reaches the set value.

```
for (let number = 0; number <= 10; number = number + 4) {
console.log(number);
}
// → 0
// → 4
// etc
```

The parentheses must consist of two semicolons after the keyword, the first part that is before the first semicolon separates the loop by describing a binding. The other part is the statement that determines if the loop should continue. Here is the code that evaluate 210 using for instead of while:

```
let result = 1;
for (let counter = 0; counter < 10; counter = counter + 1) {
```

```
result = result * 2;
}
console.log(result);
// → 1024
```

Breaking Out of a Loop

There are other ways to end a loop other than making a looping condition give a false. Break can influence jumping out of the confined loop. The break statement evaluates this program if the first number, which is both greater than or equal to 20, and can be divided by 7.

```
for (let current = 20; current = current + 1) {
if (current % 7 == 0) {
console.log(current);
break;
}
}
// → 21
```

The (%) operator is used to check if a number can be divided by another number. If it can be divided, then the remaining division is zero. The for in the example is not checked at the end of the loop, which means until the break statement inside is implemented the loop becomes infinite and never stops.

Updating bindings succinctly

When you are in a loop, programs will update a binding regularly to store a value based on its previous value.

```
counter = counter + 1;
```

JavaScript offers a shortcut.

counter += 1;

Related shortcuts function for several other operators, like result *= 2 to multiply result or counter -= 1 to count in descending order.

This enables us to minimize our counting example.

```
for (let number = 0; number <= 12; number += 2) {
console.log(number);
}
```

For counter += 1 and counter -= 1, here are shorter equivalents: counter++ and counter--.

Dispatching on a value with switch

Dispatching on a value with switch Codes can look like this:

```
if (x == "value1") action1();
else if (x == "value2") action2();
else if (x == "value3") action3();
else defaultAction();
```

The switch is used to express the above form of dispatch straightforwardly. Below is a good example:

```
switch (prompt("What is the atmosphere like?")) {
case "rainy":
console.log("Do not forget to come with an umbrella.");
break;
case "sunny":
console.log("Be light with your dressing.");
case "cloudy":
console.log("Move out.");
```

```
break;
default:
console.log("Weather type Unknown!");
break;
}
```

Capitalization

Names binding do not allow spaces, but it supports the usage of several words to define the binding values. Here are a few types that can be utilized when binding names with different words:

```
fuzzylittleturtle
fuzzy_little_turtle
FuzzyLittleTurtle
fuzzyLittleTurtle
```

Comments

Sometimes raw codes do not transmit every single piece of information that you want the program to send to readers or spreads the message in a way people will find it hard to decipher. Other times you feel like you should attach some similar thoughts to your program, the comment performs this function. A comment is a bit chunk of the text contained in a program, but the computer ignores it. To code a single line comment, use the two slash characters (//), followed by the comment text.

```
let accountBalance = calculateBalance(account);
// It's a yellow-orange on the mango tree accountBalance.adjust();
// Catching green tatters in our home. let report = new Report();
// Where the sun and the moon meets: addToReport(accountBalance, report);
```

// It's a large hallway, and the lights are quite amazing.

A // comment appears only at the end of the line. Any text within /* and */ ignore it, and it does not matter if it contains line breaks. It is used to attach blocks of information about a program or file.

Summary

Now you understand that you can develop a program through the use of statements, which statement itself can contain several statements. Statements consist of expressions, and you can create expressions through smaller expressions. Setting statements after each another provides an executed program from top to bottom. Disturbances can also come into the flow of influence through conditional (if, else, and switch) and looping (while, do, and for) statements. And we touched bindings, they categorize bits of data under a name, and they can also track state within the program. The defined bindings enjoy the environment better. We also touched on the functions section being unique values that summarize a piece of program. You can call them by typing functionName(argument1, argument2). This type of function call is an expression and can provide value.

Exercise

Write a JavaScript program that displays the larger and accepts double integers.

Solution

HTML Code:

```
<!DOCTYPE html>
<html>
<head>
<meta charset=utf-8 />
<title>Write a JavaScript program that accept two integers and display the larger</title>
</head>
<body>

</body>
</html>
```

JavaScript Code:

```
var num1, num2;
num1 = window.prompt("Input the first integer", "0");
num2 = window.prompt("Input the second integer", "0");

if(parseInt(num1, 10) > parseInt(num2, 10))
  {
  console.log("The larger of "+ num1+ " and "+ num2+ " is "+ num1+ ".");
  }
else
  if(parseInt(num2, 10) > parseInt(num1, 10))
  {
  console.log("The larger of "+ num1+" and "+ num2+ " is "+ num2+ ".");
  }
else
  {
   console.log("The values "+ num1+ " and "+num2+ " are equal.");
  }

```

It is important to know that you can write .length at the end of a string to find its length.

let abc = "abc";

```
console.log(abc.length);
// → 3
```

Chapter 4: Functions

Functions are the alpha and omega of JavaScript programming language. The concept of enclosing a bit of program contained in value has a lot of advantages. It provides a way to organize more prominent programs, to minimize repetition, relate names with subprograms, and separate programs from themselves. Functions are used to describe new words.

Defining a function

A function is a systematic binding whereby a function defines the binding value. The code below indicates a function that provides the square of a stated number:

```
const square = function(x) {
return x * x;
};
console.log(square(12));
// → 144
```

Create a function with an expression that begins with the keyboard function. Functions consists of a body and a set of parameters that accommodates the yet to be executed statements when you call a function. The function body should always be enclosed in braces, even when it is just an individual statement. A function can contain several parameters or zero parameter. In the below example, makeNoise contains no parameter names and power consists of two:

```
const makeNoise = function() {
```

```
console.log("Pling!");
};
makeNoise();
// → Pling!
const power = function(base, exponent) {
let result = 1;
for (let count = 0; count < exponent; count++) {
result *= base;
}
return result;
};
console.log(power(2, 10));
// → 1024
```

Some functions can create values such as square and power while some results only into a side effect. A return keyword that does not contain an expression at the end will be returned as undefined.

Bindings and scopes

Every binding consists of a scope that enables the visibility of the binding. Whenever you call a function, new illustrations of binding are developed. This creates separation between functions, every function call behaves in its own world and is easy to understand. Functions that were declared with the var keyword in the pre-2015 JavaScript and can be seen throughout the global scope. They are absent in a function.

```
let x = 10;
if (true) {
```

```
let y = 20;
var z = 30;
console.log(x + y + z);
// → 60
}
// y is not visible here
console.log(x + z);
// → 40
```

Nested scope

JavaScript differentiate between the global and local bindings. You can create block and functions within the other functions and blocks manufacturing several degrees of locality. For instance, this function produces the needed components to make a group hummus contains another function within it:

```
const hummus = function(factor) {
const ingredient = function(amount, unit, name) {
let ingredientAmount = amount * factor;
if (ingredientAmount > 1) {
unit += "s";
}
console.log(`${ingredientAmount} ${unit} ${name}`);
};
ingredient(1, "can", "chickpeas");
ingredient(0.25, "cup", "tahini");
ingredient(0.25, "cup", "lemon juice");
ingredient(1, "clove", "garlic");
```

```
ingredient(2, "tablespoon", "olive oil");
ingredient(0.5, "teaspoon", "cumin");
};
```

The code within the component function can access the factor binding from the outer function. The set of bindings accessible within a block is decided by the block position in the program text. Every local scope can access the contained local scope and every scope can as well access the global scope. This concept is named lexical scoping.

Functions as values

A function binding often behaves as a name for a particular block of program. This type of binding does not change because it is defined. Functions can be used in arbitrary expressions as well as used to save a function value into a new binding. A binding that stores a function is a systematic binding and a new value can be assigned like:

```
let launchMissiles = function() {
missileSystem.launch("now");
};
if (safeMode) {
launchMissiles = function() {/* do nothing */};
}
```

Declaration notation

There is a little way to generate a function binding. When you use the function keyword at the beginning of a statement, it will function differently.

```
function square(x) {
```

```
return x * x;
}
```

This statement describes the binding square and directs it at a stated function. This type of function definition consists of one precision.

```
console.log("My mind tells me every time:", future());
function future() {
return "Ill surpass Bill Gates";
}
```

Arrow functions

Functions contains a third notation that does not look identical to the others. You can use the arrow (=>) as an alternative to the function keyword. The arrow (=>) contains an equal sign and the greater than character.

```
const power = (base, exponent) => {
let result = 1;
for (let count = 0; count < exponent; count++) {
result *= base;
}
return result;
};
```

If you have just one parameter name, exclude the parentheses surrounding the parameter list. Sometimes the body is a single expression instead of a block of braces, taking back the expression from the function. Therefore, square definitions perform the same task.

```
const square1 = (x) => {return x * x;};
```

```
const square2 = x => x * x;
```

When there is no parameter on an arrow function, its parameter list becomes a set of unoccupied and ineffective parentheses.

```
const horn = () => {
console.log("Toot");
};
```

You don't have to use both the function expressions and arrow functions in the language, they offer and perform the same operations.

The call stacks

Let us take some time to see how control goes through functions. Below is a simple program making few function calls:

```
function greet(who) {
console.log("Hello " + who);
}
greet("Harry");
console.log("Bye");
```

The greetings call influences to control to jump to the beginning of the function. The function call console.log gets its job done and seize control, and then send control back to the function. The function cycle ends there and transfers back to the place that calls it. The flow of control is below:

```
not in function
in greet
in console.log
in greet
not in function
```

in console.log

not in function

The computer has to recollect the context in which the call sequences occurred because functions go back to where it's called position. When Console.log completes, it must return to the end of the program. A call stack is a place where the computer stores this context. Whenever you call a function, the present context saves at the top of this stack. By the time a function goes back, it eradicates the top context and utilizes the content for continual execution.

The stack needs space to be saved into within the computer memory. The below code explains this by querying the computer that creates zero limitation between two functions back and forth.

```
function chicken() {
return egg();
}
function egg() {
return chicken();
}
console.log(chicken() + " came first.");
// → ??
```

Optional Arguments

The below code functions properly without interference:

```
function square(x) {return x * x;}
console.log(square(4, true, "hedgehog"));
// → 16
```

The square is described with just one parameter and we call it three. That is possible because the programming language disregards the surplus arguments and recognizes the square of the first one. JavaScript is very tolerant about the amount of arguments being passed to a function. The good side of this character is that it enables calling functions with separate numbers of arguments:

```
function minus(a, b) {
if (b === undefined) return -a;
else return a - b;
}
console.log(minus(10));
// → -10
console.log(minus(10, 5));
// → 5
```

When an = operator is written after a parameter and an expression, the expression value will restore the non-specified argument. If you want to pass and not produce the second argument, the default becomes two and the function acts like a square.

```
function power(base, exponent = 2) {
let result = 1;
for (let count = 0; count < exponent; count++) {
result *= base;
}
return result;
}
```

```
console.log(power(4));
// → 16
console.log(power(2, 6));
// → 64
```

Closure

The capability to use functions as values is coupled with the fact that local bindings are created again whenever you call a function. The below code displays this example; it describes a wrap value, a function that develops a local binding and then sends back a function that enters and returns the local binding.

```
function wrapValue(n) {
let local = n;
return () => local;
}
let wrap1 = wrapValue(1);
let wrap2 = wrapValue(2);
console.log(wrap1());
// → 1
console.log(wrap2());
// → 2
```

This concept is called closure. It gives the user the ability to reference a particular instance of a local binding within a confining scope. Adding a few changes, our previous example can turn into a way to develop functions that accumulates by an arbitrary amount.

```
function multiplier(factor) {
return number => number * factor;
```

```
}
let twice = multiplier(2);
console.log(twice(5));
// → 10
```

Recursion

A function can call itself but should not call itself regularly to avoid stack overflow. Recursive function is a function that calls itself. Recursion enables few functions written in separate styles. For instance, the below code is the execution of power.

```
function power(base, exponent) {
if (exponent == 0) {
return 1;
} else {
return base * power(base, exponent - 1);
}
}
console.log(power(2, 3));
// → 8
```

There is one problem with this execution. It is slower compared to other looping versions. Utilizing a single loop is considered low cost than the multiple calling of functions. Although that does not make recursion an ineffective option of looping, few problems are solved with recursion easier than with the use of loops. Problems that need inspecting or processing several branches. Check this out: we begin from number 1 and continuously add 5 or multiply by 3. For instance, the number 13 can be obtained by multiplying by 3 and the addition of 5

twice, thereby we cannot attain the 15. The code is the recursive solution:

```
function findSolution(target) {
function find(current, history) {
if (current == target) {
return history;
} else if (current > target) {
return null;
} else {
return find(current + 5, `(${history} + 5)`) ||
find(current * 3, `(${history} * 3)`);
}
}
return find(1, "1");
}
console.log(findSolution(24));
// → (((1 * 3) + 5) * 3)
```

The inner function find takes two arguments, the current number and a string that documents the process of attaining this number. If a solution is found, it sends back a string that displays the route to the target and if no solution is found, the returned value will be null. To achieve this, the function executes one of three actions. If your target is the is the current number, you can attain that target by using the current history so it is sent back. Sometimes the number is larger than the target, but you do not need to explore this option because addition and subtraction can only enlarge the number so it is

sent back as null. If you remain beneath the target number, both paths that begin the current number is tried by calling itself twice, one to add and the other to multiply. If the first call sends something valid back in return, then good, if not, return the second call, it does not matter whether a string or null is provided. Below is an illustration of how functions provide effects. This example searches for a remedy for the number 13.

```
find(1, "1")
find(6, "(1 + 5)")
find(11, "((1 + 5) + 5)")
find(16, "(((1 + 5) + 5) + 5)")
too big
find(33, "(((1 + 5) + 5) * 3)")
too big
find(18, "((1 + 5) * 3)")
too big
find(3, "(1 * 3)")
find(8, "((1 * 3) + 5)")
find(13, "(((1 * 3) + 5) + 5)")
found!
```

The indentation specifies profound of the call stack.

Growing functions

Functions can be introduced into programs in two ways. The first is by writing similar codes a lot of times, which enables more mistakes while the second is to search for a few functionalities

that have not been written and deserves its own function. You can begin by naming the function and write the body. Below is an example. We will write a program that reproduce two numbers–the amount of chicken and cows available on a farm.

```
007 Cows
011 Chickens
function printFarmInventory(cows, chickens) {
let cowString = String(cows);
while (cowString.length < 3) {
cowString = "0" + cowString;
}
console.log(`${cowString} Cows`);
let chickenString = String(chickens);
while (chickenString.length < 3) {
chickenString = "0" + chickenString;
}
console.log(`${chickenString} Chickens`);
}
printFarmInventory(7, 11);
```

Writing length after a string expression determines the length of the particular string. The loop continues to add zeros at the beginning of the number strings until they comprise of three characters.

Here is a better attempt:

```
function printZeroPaddedWithLabel(number, label) {
let numberString = String(number);
```

```
while (numberString.length < 3) {
numberString = "0" + numberString;
}
console.log(`${numberString} ${label}`);
}
function printFarmInventory(cows, chickens, pigs) {
printZeroPaddedWithLabel(cows, "Cows");
printZeroPaddedWithLabel(chickens, "Chickens");
printZeroPaddedWithLabel(pigs, "Pigs");
}
printFarmInventory(7, 11, 3);
```

Instead of taking out the replicated part of the program, try to pick out an individual concept with the following steps:

```
function zeroPad(number, width) {
let string = String(number);
while (string.length < width) {
string = "0" + string;
}
return string;
}
function printFarmInventory(cows, chickens, pigs) {
console.log(`${zeroPad(cows, 3)} Cows`);
console.log(`${zeroPad(chickens, 3)} Chickens`);
console.log(`${zeroPad(pigs, 3)} Pigs`);
}
printFarmInventory(7, 16, 3);
```

A function can also be used to print aligned tables of numbers.

Functions and side effects

Functions can consist of side effects and return a value. They develop values that integrate easily in new ways much more than functions that produce side effects. A pure function is a unique value manufacturing function that does not depend on either code side effects. For instance, it will not recognize a global binding with a changeable value. It also consists of a callable property together with the same arguments and produces equal value.

Summary

In this chapter, you now understand how to write your functions. Knowing how to write the function keyword and when to use an expression, you can now build a function value. When you use a function as a statement, it can set a binding and name a function to be its value. You can also use arrow functions to build functions.

```
// Define f to hold a function value
const f = function(a) {
console.log(a + 2);
};
// Declare g to be a function
function g(a, b) {
return a * b * 3.5;
}
// A less verbose function value
let h = a => a % 3;
```

The major perspective of understanding functions is knowing the scopes. Every block builds a new scope. Parameters and bindings set in a specific scope are local and invisible from the outside. Bindings set with var act differently, and they result in the global scope. Differentiating the tasks that the program executes into various functions is important. You do not need to repeat yourself often, and functions can arrange a program by categorizing code into bits that perform specific actions.

Exercise

Write a JavaScript program to see if a number is even or not.

Solution

```
21 console.log(is_even_recursion(234)); //true
22 console.log(is_even_recursion(-45)); // false
23 console.log(is_even_recursion(-45)); // false
```

Output:

```
true
false
false
```

JavaScript Code:

```
1  function is_even_recursion(number)
2  {
3    if (number < 0)
4    {
5      number = Math.abs(number);
6    }
7    if (number===0)
8    {
9      return true;
10   }
11   if (number===1)
12   {
13     return false;
```

```
  }
  else
  {
    number = number - 2;
    return is_even_recursion(number);
  }
}
```

HTML Code:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Use recursion to determine if a number is or not.</title>
</head>
<body>
</body>
</html>
```

Chapter 5: Data Structures: Objects and Arrays

Data layouts develop from Booleans, strings, and numbers fragments. Several types of information use more than one chunk. Objects enable the user to collate values together with other objects to develop more compound layouts. In this chapter, you will understand the concepts of solving a real problem at hand.

Data sets

If you want to work with a lot of digital data, you must represent the digital data in the machine memory. For instance, you need to represent a group of numbers like 2, 3, 5, 7, and 11. Now let's get prolific with our use of strings as strings can contain quite a lot of data. We will use 2 3 5 7 11 as the representation. JavaScript programming language offers a unique data type for saving sequences of values, which is called an array. Write an array as a list of values within the between square brackets, differentiated by commas.

```
let listOfNumbers = [2, 3, 5, 7, 11];
console.log(listOfNumbers[2]);
// → 5
console.log(listOfNumbers[0]);
// → 2
console.log(listOfNumbers[2 - 1]);
```

```
// → 3
```

The notation used to get elements within an array also utilize the square brackets. A complete square brackets pair right after an expression that contains another expression within. When you start counting elements in an array, you begin from zero and not one. Therefore the first element is redeemed with listOfNumbers[0].

Properties

The previous chapters have contained expressions like the myString. Length (to calculate the length of a string) and Math.max (the maximum function). These are properties that can approach the property values. Virtually all JavaScript values contain properties. The few values that do not contain properties are null and undefined. If you access a property with a nonvalue, the result will be an error.

```
null.length;
// → TypeError: null has no properties
```

In JavaScript, the two major ways to evaluate properties are with a dot and square brackets. The value.x and value[x] can evaluate the property on value, not particularly the same property. How you use x is different from how you use a dot.

If you want to use the dot, the text you input after the dot is the name of the property. And if you're going to use the square brackets, the expression within the brackets is accessed to redeem the property name. Value.x delivers the property of value called “x,” while value[x] accesses the expression x and utilize the result, transformed into a string as the name of the property. If the property you want is named color, you should

type value.color. You can withdraw the property that the value called within the binding i, type value[i]. The name of properties are strings, but the dot notation performs with only valid binding names. Therefore if you want to evaluate a property named 2 or John Doe, use the square brackets: value[2] or value["John Doe"].

The elements within an array save as the properties of an array that utilize numbers as property names. An array's length property determines how many items it contains. The name of that property is a valid, binding name. Type array.length to determine the length of an array, it is much easier to type than the array["length"].

Methods

The string and array objects accommodate several properties that store function values.

```
let doh = "Doh";
console.log(typeof doh.toUpperCase);
// → function
console.log(doh.toUpperCase());
// → DOH
```

Each string consists of a toUpperCase property. When you call this property, it sends back a copy of the string whereby all the containing letters transform into uppercase. Although the toUpperCase do not go through an argument, the function can evaluate the string "Doh," which is the property of the value we called above. Properties that consist of functions are known as

methods. The below example illustrates two methods that can be used to influence arrays.

```
let sequence = [1, 2, 3];
sequence.push(4);
sequence.push(5);
console.log(sequence);
// → [1, 2, 3, 4, 5]
console.log(sequence.pop());
// → 5
console.log(sequence);
// → [1, 2, 3, 4]
```

The push method is used to attach a value to the end of an array while the pop method does the direct opposite. It eradicates the last value within an array and sends it back. These are generational terms of operations on a stack. A stack is a data layout that enables users to push values in and pop them out in the opposite direction so that the added value will remove first.

Objects

You can represent a group of log entries as an array although each string entry is required to save a list of activities as well as a Boolean value that specifies if Jacques transformed to a squirrel or not. We will group this into a single value and place the grouped values within an array of log entries. Arbitrary collections of properties are the value of the type object. We will develop an object utilizing braces as an expression.

```
let day1 = {
```

```
squirrel: false,
events: ["work", "touched tree", "pizza", "running"]
};
console.log(day1.squirrel);
// → false
console.log(day1.wolf);
// → undefined
day1.wolf = false;
console.log(day1.wolf);
// → false
```

The braces contain a list of properties divided by commas. Every property contains a name accompanied by a value and a colon. When you write an object over several lines, make sure your indenting is good because it enables better readability. Quote Invalid binding names properties or valid numbers should.

```
let descriptions = {
work: "Went to work,"
"touched tree": "Touched a tree."
};
```

You can use braces in two different ways in JavaScript. It is used at the beginning of a statement and can also be utilized to begin a block of statements. You can specify a value to a property expression using the = operator. It takes the place of the property value, and if it is in existence, it develops a new one. The delete operator is a unary operator that eradicates the property name of an object. Below is an illustration:

```
let anObject = {left: 1, right: 2};
```

```
console.log(anObject.left);
// → 1
delete anObject.left;
console.log(anObject.left);
// → undefined
console.log("left" in anObject);
// → false
console.log("right" in anObject);
// → true
```

The binary in operation determines if an object or string contains a named property. The major differentiation between specifying a property to undefined and deleting it is that the object consists of a property and the deleting does not possess the property, so the value returns as false. To determine what properties contained in an object and sends back an array of strings with property names of the object.

```
console.log(Object.key({x: 0, y: 0, c: 2}));
// → ["x", "y", "c"]
```

The Object.assign function is used to copy properties from one object to another.

```
let objectA = {a: 1, b: 2};
Object.assign(objectA, {b: 3, c: 4});
console.log(objectA);
// → {a: 1, b: 3, c: 4}
```

An array is a unique object used for saving succession of things. When you access the typeof [], it provides "object." The below

illustration stands for the journal that Jacques stores as an array of objects.

```
let journal = [
{events: ["work", "touched tree", "pizza",
"running", "television"],
squirrel: false},
{events: ["work", "ice cream", "cauliflower",
"lasagna", "touched tree", "brushed teeth"],
squirrel: false},
{events: ["weekend", "cycling", "break", "peanuts",
"beer"],
squirrel: true},
/* and so on... */
];
```

Mutability

We have discussed different values such as Booleans, strings, and numbers whose values are difficult to change, although they can be combined to acquire new values from them. Objects are different from values, and it enables property changing; they can create different content for a single object value at different times. When there are two numbers, 120 and 120, they are considered the same, and there is a similarity between setting two references to one object and possessing two separate objects containing the same properties. Check the below code:

```
let object1 = {value: 10};
let object2 = object1;
```

```
let object3 = {value: 10};
```



```
console.log(object1 == object2);
// → true
console.log(object1 == object3);
// → false
object1.value = 15;
console.log(object2.value);
// → 15
console.log(object3.value);
// → 10
```

The object1 and object2 bindings hold the same object, and that is why altering object1 one influences the value of object2. They are identical by nature. The binding object3 points to a separate object that consists of similar properties as the object1 but has a different life. Bindings can be constant or changeable, but it does not influence their values. A const binding an object cannot be transformed and relentlessly pointing to the same object. It is the object contents that can be changed.

```
const score = {visitors: 0, home: 0};
// This is okay
score.visitors = 1;
// This isn't allowed
score = {visitors: 1, home: 1};
```

When the JavaScript's == operator is used to differentiate objects, it uses the identity to perform that task. It will translate true if

the objects contain the same value. Differentiating several objects will return false, even if their properties are similar.

The lycanthrope's log

So, you begin your JavaScript interpreter and creates the environment you need to keep your journal.

```
let journal = [];
function addEntry(events, squirrel) {
journal.push({events, squirrel});
}
```

Instead of proclaiming properties such as events, it sets a property name.

So then, at 10 p.m. every evening or occasionally in the following morning, after

getting down from the top shelf of your bookcase, your daily records.

```
addEntry(["work", "touched tree", "pizza", "running",
"television"], false);
addEntry(["work", "ice cream", "cauliflower", "lasagna",
"touched tree", "brushed teeth"], false);
addEntry(["weekend", "cycling", "break", "peanuts",
"beer"], true);
```

Correlation is an evaluation of vulnerability between analytical variables. They are known as values that span from -1 to 1. An Unrelated variable is called a zero correlation. A correlation of 1 signifies that the two are related. Negative means that the variables are related perfectly but opposites to each other; one is

true, and the other is false. To calculate the evaluation of the correlation between two Boolean values, utilize the phi coefficient (φ). This formula input is a frequency table holding the number of times it notices several mixtures of the variables. The formula's output will be a number between -1 and 1. That is the best definition of a correlation.

Let us use the event if eating pizza as an example and insert it into a frequency table whereby every number signifies the total of times, and we used the combination in our measurements. We name the table n, and we calculate φ with the below formula:

$$\varphi = \frac{n_{11}n_{00} - n_{10}n_{01}}{\sqrt{n_{1\bullet}n_{0\bullet}n_{\bullet 1}n_{\bullet 0}}}$$

The notation n01 signifies the number of measurements where the first variable turns false (0), and the second variable becomes true (1). The pizza table, n01 is 9.

The value n1• indicates the sum of all measurements where the first variable turns true, that is 5 in the table example. As well as n•0 suggests the quantity of the measurements where the second variable becomes false.

The pizza table, the top of the division line would be 1×76−4×9 = 40, and the below part below would be the square root of 5×85×10×80, or √340000. The result will be φ ≈ 0.069, which is tiny. Therefore, eating pizza had no impact on transformations.

Computing correlation

JavaScript enables the representation of a two-by-two table with four element arrays ([76, 9, 4, 1]). Other representations such as an array holding two two-element ([[76, 9], [4, 1]]) or an object containing property names such as "11" and "01", and the flat array, which produces the expression that access the chair short. We will explain the indices to the array as two-bit binary numbers, where the most significant digit refers to the squirrel variable while the least significant refers to the event table. For instance, the binary number 10 refers to the "Jacques did turn into a squirrel" case but the events didn't appear. It occurred four ties and binary is 2 in decimal, the number is stored as an index 2 of the array. This function calculates the φ coefficient from an array:

```
function phi(chair) {
return (chair [3] * chair 0] - chair [2] * chair [1]) /
Math.sqrt((chair [2] + chair [3]) *
(chair [0] + chair [1]) *
(chair [1] + chair [3]) *
(chair [0] + chair [2]));
}
console.log(phi([76, 9, 4, 1]));
// → 0.068599434
```

The table requires two fields to acquire fields like n1• because the total number of rows and columns are stored indirectly into our data layout. If you want to extract a two-by-two table for a particular event, you must loop across all the entries and

calculate the number of times the event happened to the squirrel transformations.

```
function tableFor(event, journal) {
let table = [0, 0, 0, 0];
for (let i = 0; i < journal.length; i++) {
let entry = journal[i], index = 0;
if (entry.events.includes(event)) index += 1;
if (entry.squirrel) index += 2;
table[index] += 1;
}
return table;
}
console.log(tableFor("pizza", JOURNAL));
// → [76, 9, 4, 1]
```

Arrays consists of an include method, which is used to check if a set value exists in the array. This method is used by the function to decide if the event name is interested the stated event list for a given day.

Array loops

The tableFor function contains a loop like this:

```
for (let I = 0; I < JOURNAL.length; I++) {
let entry = JOURNAL[i];
// Do something with entry
}
```

This type of loop is used in high-level JavaScript

This is how to write that type of loop in modern JavaScript.

```
for (entrance of JOURNAL) {
console.log(`${entrance.events.length} events.`);
}
```

With a loop like this, the loop continues across the elements of the value specified after of. This loop is used for strings and other data layouts.

The final analysis

A correlation is calculated for every type of event that happens in the data set. If you want to achieve that, search every type of event.

```
function journalEvents(journal) {let events = [];
for (let entry of journal) {
for (let event of entry.events) {
if (!events.includes(event)) {
events.push(event);
}
}
}
return events;
}
console.log(journalEvents(JOURNAL));
// → ["carrot", "exercise", "weekend", "bread", ...]
```

By moving across every event and making additions to those that are not in the events array, this function gathers all the type of event.

A lot of correlations appear to lie close to zero. Eating bread, carrots or pudding does not activate the squirrel-lycanthropy. It

happens often during the weekends. Using that, below are all the correlations.

```
for (let event of journalEvents(JOURNAL)) {
console.log(event + ":", phi(tableFor(event, JOURNAL)));
}
// → carrot: 0.0140970969
// → exercise: 0.0685994341
// → weekend: 0.1371988681
// → bread: -0.0757554019
// → pudding: -0.0648203724
// and so on...
```

let us filter results to display only correlations that are greater than 0.1 or less than -0.1.

```
for (let event of journalEvents(JOURNAL)) {
let correlation = phi(tableFor(event, JOURNAL));
if (correlation > 0.1 || correlation < -0.1) {
console.log(event + ":", correlation);
}
}
// → weekend: 0.1371988681
// → brushed teeth: -0.3805211953
// → candy: 0.1296407447
// → work: -0.1371988681
// → spaghetti: 0.2425356250
// → reading: 0.1106828054
```

// → peanuts: 0.5902679812

In a correlation factors, one is stronger than the other. Eating peanuts has a powerful chance of transforming into a squirrel, thereby brushing the teeth has a notable negative effect. Here's something:

```
for (let entry of JOURNAL) {
if (entry.events.includes("peanuts") &&
!entry.events.includes("brushed teeth")) {
entry.events.push("peanut teeth");
}
}
console.log(phi(tableFor("peanut teeth", JOURNAL)));
// → 1
```

Further arrayology

Here are a few more object-related concepts you should know. We begin with the use of some useful methods of an array.

The adding and removing of methods things at the beginning of an array are called unshift and shift.

```
let todoList = [];
function remember(task) {
todoList.push(task);
}
function getTask() {
return todoList.shift();
}
function rememberUrgently(task) {
```

```
todoList.unshift(task);
}
```

The above program organizes a chain of tasks. You can add tasks to the end of the queue by calling remember("groceries"), and when you want to get something done, call getTask() to get (and remove) item from the queue. If you want to search for a particular value, arrays supply an indexOf method. The method finds its way through the array from the beginning to the end and sends back the index the value requested, if it was found it returns —or -1 if it wasn't found. To search from the beginning to the end, use the lastIndexOf method.

```
console.log([1, 2, 3, 2, 4].indexOf(3));
// → 1
console.log([1, 2, 3, 2, 4].lastIndexOf(3));
// → 3
```

The indexOf and lastIndexOf takes a voluntary argument that specifies where the search begins. Another great method is the slice, this method begins and ends the indices and send back an array containing three elements. The beginning is inclusive while the end is exclusive.

```
console.log([0, 1, 2, 3, 5].slice(2, 4));
// → [2, 3]
console.log([0, 1, 2, 3, 5].slice(2));
// → [2, 3, 5]
```

If the end index is not stated, slice take all the elements after the start index. You can use the concat method to glue arrays

together to produce a new array. The below example displays concat and slice in action.

```
function remove(arrays, index) {
return array.slice(0, index)
.concat(arrays.slice(index + 1));
}
console.log(remove(["a", "k", "c", "f", "e"], 2));
// → ["a", "k", "f", "e"]
```

Strings and their properties

During the string value chapter, we discussed length and toUpperCase but if you want to add a new property, it does not stick.

```
let kim = "Kim";
kim.age = 88;
console.log(kim.age);
// → undefined
```

All strings value contains a few methods. Some are the slice and indexOf which have a similar appearance of array methods as well as the name.

```
console.log("coconuts”. Slice(4, 7));
// → nut
console.log("coconut”.indexOf("u"));
// → 5
```

A string’s indexOf can look for a string holding more than one character, while the corresponding array method search for a single element.

```
console.log("one two three".indexOf("ee"));
// → 11
```

The trim method eradicates whitespace from the beginning to the end of a string.

```
console.log("  okay \n ".trim());
// → okay
```

The zeroPad function can also be called a padStart, it takes the preferred padding and length character as arguments.

```
console.log(String(6).padStart(3, "0"));
// → 006
```

You can break a string on every development of another string with split and then join them together with join.

```
let sentence = "Secretarybirds specialize in stomping";
let words = sentence.split(" ");
console.log(words);
// → ["Secretarybirds", "specialize", "in", "stomping"]
console.log(words.join(". "));
// → Secretarybirds. specialize. in. stomping
```

You can use the repeat method to repeat a string, which develops a new string holding several copies of the original string attached together.

```
console.log("LA".repeat(3));
// → LALALA
```

If you want to access individual characters in a string. look below:

```
let string = "ABC";
```

```
console.log(string.length);
// → 3
console.log(string[1]);
// → b
```

Rest parameters

It is important for a function to receive any number of arguments. If you want to write such a function, add three dots before the last parameter of the function.

```
function max (...numbers) {
let result = -Infinity;
for (let number of numbers) {
if (number > result) result = number;
}
return result;
}
console.log(max(4, 1, 9, -2));
// → 9
```

When you call a function, the rest of the parameters belongs to an array holding the rest of the arguments. You can also utilize the three dot notation to call functions within an array of arguments.

```
let numbers = [5, 1, 7];
console.log(max(...numbers));
// → 7
```

Spread the array out into a function call, pass the element different arguments, like max (9, ...numbers, 2). The Square

bracket array notation enables the triple-dot operator bring another array into a new array.

```
let words = ["never", "fully"];
console.log(["will", ...words, "understand"]);
// → ["will", "never", "fully", "understand"]
```

The Math objects

The Math object is utilized as vessel to collate a lot of similar functionality. It produces a namespace to enable values a functions and functions without global bindings. This is the old way to write the constant value name in all caps.

```
function randomPointOnCircle(radius) {
let angle = Math.random() * 2 * Math.PI;
return {x: radius * Math.cos(angle),
y: radius * Math.sin(angle)};
}
console.log(randomPointOnCircle(2));
// → {x: 0.3667, y: 1.966}
```

The math function sends back a new pseudo random number between zero and one whenever you call it.

```
console.log(Math.random());
// → 0.36993729369714856
75
console.log(Math.random());
// → 0.727367032552138
console.log(Math.random());
// → 0.40180766698904335
```

When you need a whole random number and not the fractional one, you can utilize the Math.floor on the outcome of the Math.random.

```
console.log(Math.floor(Math.random() * 10));
// → 2
```

The random number is multiplied by 10 and outputs a number greater than or equal to 0 and below 10. Remember the Math.floor rounds down, the output will be any number from 0 through 9. The math object contains several functions like the Math.ceil, which rounds up to a whole number, and the Math.round, which rounds up to the nearest whole number, and Math.abs, this function takes the entire value of a number.

Destructuring

Let's use the phi function a little bit.

```
function phi(chair) {
return (table[3] * chair[0] - table[2] * chair[1]) /
Math.sqrt((chair [2] + chair [3]) *
(table[0] + chair [1]) *
(table[1] + chair [3]) *
(table[0] + chair [2]));
}
```

One of the few reasons this function is hard to read is because there is a binding directed to the array, but we want bindings for the elements of the array, that is, let n00 = table[0] and so on. Although, there is a better way to get this done in JavaScript.

```
function phi([n00, n01, n10, n11]) {
```

```
return (n11 * n00 - n10 * n01) /
Math.sqrt((n10 + n11) * (n00 + n01) *
(n01 + n11) * (n00 + n10));
}
```

This function also performs brilliantly with the bindings developed by the left, var, or const. If your preferred binding value is an array, utilize the square brackets to check for the value of its binding contents. Use the same concept for objects, utilize the braces and not the square brackets.

```
let {name} = {name: "Faraji", age: 23};
console.log(name);
// → Faraji
```

JSON

JSON represents JavaScript Object Notation and it is utilized as a communication format and data storage for the web. JSON is very similar to JavaScript in its writing style of arrays and object, although there are restrictions. Its property names should be enclosed within double quotes, it allows only simple data expressions, binding etc. JSON do not allow comments.

Here is a sample of a journal entry represented as JSON data:

```
{
"squirrel": false,
"events": ["work", "touched tree", "pizza", "running"]
}
```

JavaScript consists of the functions JSON.stringify and JSON.parse to transform data to and from this format. JSON.stringify takes a

JavaScript value and sends back a JSON-encoded string while JSON.parse takes a string and transforms it to its encoded value.

```
let string = JSON.stringify({squirrel: false,
events: ["weekend"]});
console.log(string);
// → {"squirrel":false,"events":["weekend"]}
console.log(JSON.parse(string).events);
// → ["weekend"]
```

Reversing an array

Arrays consist of the reverse method, which is used to change an array by reversing the order that its elements display.

Summary

We covered the objects and arrays in this chapter, and they create ways to categorize various values in a single value. Fancifully, this enables you to insert several associated items in a bag and move about with the bag, rather than enclosing your arms around every singular thing and trying to keep them differently. A lot of values in JavaScript contain properties, the inconsistency being undefined and null. You can access properties using the value.prop or value["prop"]. You can use names for objects properties and save a defined set of them. Arrays, as well contain different amounts of similar fanciful values and utilize the numbers (beginning from 0) as the property's names. Arrays contain few sets of properties, which are the length and several methods. Methods are functions that reside within properties and behave on the value they are its property. You

can loop across arrays through a unique type of for loop—for (the let element of an array).

Exercise

Write a JavaScript function to obtain the first item in an array. Setting a parameter 'n' will send back the first 'n' elements of the array.

Data:

```
console.log(array_Clone([1, 2, 4, 0]));
console.log(array_Clone([1, 2, [4, 0]]));
```

```
[1, 2, 4, 0]
[1, 2, [4, 0]]
```

Solutions

```
  };

console.log(first([7, 9, 0, -2]));
console.log(first([],3));
console.log(first([7, 9, 0, -2],3));
console.log(first([7, 9, 0, -2],6));
console.log(first([7, 9, 0, -2],-3));
```

HTML Code:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Get the first element of an array</title>
</head>
<body>
</body>
</html>
```

JavaScript Code:

```
var first =  function(array, n) {
      if (array == null)
      return void 0;
```

```
4     if (n == null)
5       return array[0];
6     if (n < 0)
7       return [];
8     return array.slice(0, n);
```

Chapter 6: Higher-Order Functions

A large program takes time to develop and produces a lot of space for bugs to hide, which in turn makes them hard to find. Here are examples of two large programs:

The first is self-contained, and the statement is six lines long.

```
let total = 0, count = 1;
while (count <= 10) {
total += count;
count += 1;
}
console.log(total);
```

The second depends on two external functions, and the statement is one line long.

```
console.log(sum(range(1, 10)));
```

Abstraction

When it comes to JavaScript programming language, vocabulary's use the term abstraction. Abstraction is used to hide details and gives the user the capability to talk about problems on a higher level. Let us differentiate two recipes for pea soup. First:

Per person, place 1 cup of dried peas within a container. Put enough water to cover the peas, let it soak for about 12 hours. Remove it from the water and place it in a pot used for cooking with the addition of water four cups of water (4 cups).

Let it boil for about two hours, per-person hold half of an onion, slice it using a knife, and pour it into the peas, let it prepare for 10 minutes.

Then the second recipe

Per person: Half an onion, a stalk of celery, 1 cup dried split peas, and a carrot.

Dip the peas into the water for about 12 hours. Boil it for about 2 hours in water (4 cups), attach and slice vegetables. Cook it for 10 minutes.

The computer performs tasks one by one, from blind to high-level concepts.

Abstracting repetition

Plain functions are used to create abstractions. It is habitual for a program to repeat a particular function a specified amount of time. you can achieve that with a loop like this:

83

```
for (let i = 0; i < 10; i++) {
console.log(i);
}
```

Here we write a function that calls console.log N times.

```
function repeatLog(n) {
for (let I = 0; I < n; I++) {
console.log(I);
}
}
```

If you want to perform a task different from logging in numbers, you can roll action as a function value.

```
function repeat(n, action) {
for (let I = 0; I < n; I++) {
action(i);
}
}
repeat(3, console.log);
// → 0
// → 1
// → 2
```

Sometimes you do not need to roll in a predefined function to repeat in a loop, create a function value as an alternative:

```
let labels = [];
repeat(5, i => {
labels.push(`Unit ${i + 1}`);
});
console.log(labels);
// → ["Unit 1", "Unit 2", "Unit 3", "Unit 4", "Unit 5"]
```

This method layout looks similar to a loop; it defines the type of loop and then produce a body. Although the body is written in function enclosed in parentheses of the call to repeat.

Higher-order functions

Higher-order functions are functions that work on other functions whether by arguments or by sending them back. This type of function enables user to abstract over actions. For instance, you have a function that can produce functions

```
function greaterThan(n) {
```

```
return m => m > n;
}
let greaterThan10 = greaterThan(10);
console.log(greaterThan10(11));
// → true
```

You can also have functions that can transform other functions.

```
function noisy(f) {
return (...args) => {
console.log("calling with", args);
let result = f(...args);
console.log("called with", args, ", returned", result);
return result;
};
}
noisy(Math.min)(3, 2, 1);
// → calling with [3, 2, 1]
// → called with [3, 2, 1], returned 1
```

You can as well write functions that produce new set of control flow.

```
function unless(test, then) {
if (!test) then();
}
```



```
repeat(3, n => {
unless(n % 2 == 1, () => {
console.log(n, "is even");
```

```
});
});
// → 0 is even
// → 2 is even
```

This method is a built-in array method, forEach, that produce a loop similar to the for/of loop as a higher-order function.

```
["A", "B"].forEach(l => console.log(l));
// → A
// → B
```

Filtering arrays

To search for the script in a data set, use the following function. This function helps to filter out element that failed a test in an array.

```
function filter(array, test) {
let passed = [];
for (let element of array) {
if (test(element)) {
passed.push(element);
}
}
return passed;
}
console.log(filter(SCRIPTS, script => script.living));
// → [{name: "Adlam", ...}, ...]
```

This function utilizes the argument called test, a function value, to cover a gap during the calculation. It determines which

element to receive.

Transforming with map

The map method changes an array by setting a function to each and every of its elements and creating a new array from the returned values. The new array consists of the same length with the input array but will map out its content to a renewed for by the function.

```
function map(array, transform) {
let mapped = [];
for (let element of array) {
mapped.push(transform(element));
}
return mapped;
}
let rtlScripts = SCRIPTS.filter(s => s.direction == "rtl");
console.log(map(rtlScripts, s => s.name));
// → ["Adlam", "Arabic", "Imperial Aramaic", ...]
```

Like forEach and filter.

Summarizing with reduce

When you add up numbers, begin with zero and add the foreach element to the sum. The parameter to reduce are map function combination and begin a value. This function is much easier to understand than the filter and map. It is below:

```
function reduce(array, combine, begin) {
let current = begin;
for (let items of arrays) {
```

```
existing = combine(existing, element);
}
return existing;
}
console.log(reduce([0,1, 2, 3, 4], (a, b) => a + b, 0));
// → 10
```

This function is similar to the standard array method reduce and attaches more comfort.

```
console.log([1, 2, 3, 4].reduce((a, b) => a + b));
// → 10
```

If you want to utilize reduce to search for the script with the more characters, write it like this:

```
function characterCount(script) {
return script.ranges.reduce((count, [from, to]) => {
return count + (to - from);
}, 0);
}
console.log(SCRIPTS.reduce((a, b) => {
return characterCount(a) < characterCount(b)? b: a;
}));
// → {name: "Han", ...}
```

Composability

Take some time to think about how long our list of code would have been without the higher-order functions.

```
let biggest = null;
for (let script of SCRIPTS) {
```

```
if (biggest == null ||
characterCount(biggest) < characterCount(script)) {
biggest = script;
}
}
console.log(biggest);
// → {name: "Han", ...}
```

You can use the high-order functions when you want to compose operations. For instance, here is a line of code that searches for the average year for living and dead in humans' scripts in the data set.

```
function average(array) {
return array.reduce((a, b) => a + b) / array.length;
}
console.log(Math.round(average(
SCRIPTS.filter(s => s.living).map(s => s.year))));
// → 1165
console.log(Math.round(average(
SCRIPTS.filter(s => !s.living).map(s => s.year))));
// → 204
```

The block of code displays that the death script is older than the living ones.

Let's begin with the entire script, filter out the dead (or living), eradicate their years, average them and complete the result. You can as well write these calculations as a big loop:

```
let total = 0, count = 0;
for (let script of SCRIPTS) {
```

```
if (script.living) {
total += script.year;
count += 1;
}
}
console.log(Math.round(total / count));
// → 1165
```

Our two examples are not identical, the first creates new arrays when administering the map and filter while the second calculate numbers only.

Strings and character codes

One of the significant use of data set is to find out what script a block of text is using. Below is a program that performs that task.

Every script contains an array of character code ranges related with it. So, if you are given a character code, utilize this function to search for the equivalent script (if available):

```
function characterScript(code) {
for (let script of SCRIPTS) {
if (script.ranges.some(([from, to]) => {
return code >= from && code < to;
})) {
return script;
}
}
return null;
```

```
}
console.log(characterScript(121));
// → {name: "Latin", ...}
```

Another high-order function is the same method. This method gets a test function and determine if that function sends back true for any component in the array.

There are some operations on JavaScript strings like securing their length using the length property and evaluating their contents through the square brackets.

```
// Two emoji characters, horse and shoe
let horseShoe = "🐴👟";
console.log(horseShoe.length);
// → 4
console.log(horseShoe[0]);
// → (Invalid half-character)
console.log(horseShoe.charCodeAt(0));
// → 55357 (Code of the half-character)
console.log(horseShoe.codePointAt(0));
// → 128052 (Actual code for horse emoji)
```

JavaScript's charCodeAt method offers a full Unicode character. This method is used to take characters from a string. Although the argument sent to codePointAt remains an index into the succession of code units. If you use the codePointAt to loop across a string, it produces real characters and not code units.

```
let roseDragon = "🌹🐉";
for (let char of roseDragon) {
console.log(char);
```

```
}
// → 🌹
// → 🐲
```

Recognizing text

You have the character Script function and a way to loop across characters correctly, now calculate the characters each script owns. The below counting abstraction is used here:

```
function countBy(items, groupName) {
let counts = [];
for (let item of items) {
let name = groupName(item);
let known = counts.findIndex(c => c.number == name);
if (known == -1) {
counts.push({nunber, count: 0});
} else {
counts[know].count++;
}
}
return counts;
}
console.log(countBy([0,1, 2, 3, 4, 5], n => n > 2));
// → [{number: false, count: 2}, {number: true, count: 3}]
```

The countBy function anticipates a collection (a group of numbers, element or anything that you can loop over with for/of) and a function that calculates a group name for a specified element. It sends back a list of objects. Each of the

objects names a group and determines the number of elements contained in the group.

The method findIndex is similar to indexOf. This method is used to search for the first value for which the stated function sends back true and returns -1 when zero elements is found. You can use the countBy to determine which script is utilized in a block of text.

```
function textScripts(text) {
let scripts = countBy(text, char => {
let script = characterScript(char.codePointAt(0));
return script? script.name : "none";
}).filter(({name}) => name != "none");
let total = scripts.reduce((n, {count}) => n + count, 0);
if (total == 0) return "No scripts found";
return scripts.map(({name, count}) => {
return `${Math.round(count * 100 / total)}% ${name}`;
}).join(", ");
}
console.log(textScripts('英国的狗说"woof", 俄罗斯的狗说"тяв"'));
// → 61% Han, 22% Latin, 17% Cyrillic
```

The function calculates the characters by name through the use of characterScript to attach a name and returning back to the string "none" for characters without any script. If you want to calculate percentages, you need to determine the total amount of characters belonging to a script that the reduce method can

reduce. If it finds zero characters, the function sends back the "none" string.

Summary

Having the ability to send function values to several functions is an essential aspect of JavaScript. It enables writing functions that imitate computations with “gaps” between them. The code that declares these functions can contain the gaps by giving function values. Arrays provide several important higher-order methods. Use forEach to loop across elements within an array. Use the filter method to send back a new array that has elements that convey the predicate function. Changing an array by setting each item through a function using the map. Use reduce to merge every component of an array to an individual value.

Exercises

Illustrate a JavaScript function that takes an array of numbers saved and search for the second-lowest and second highest numbers.

Solution

JavaScript Code:

```
1  function Second_Greatest_Lowest(arr_num)
2  {
3    arr_num.sort(function(x,y)
4             {
5             return x-y;
6             });
7    var uniqa = [arr_num[0]];
8    var result = [];
9
10   for(var j=1; j < arr_num.length; j++)
11   {
12     if(arr_num[j-1] !== arr_num[j])
13     {
14       uniqa.push(arr_num[j]);
15     }
16   }
17     result.push(uniqa[1],uniqa[uniqa.length-2]);
18   return result.join(',');
19   }
20
21 console.log(Second_Greatest_Lowest([1,2,3,4,5]));
```

HTML Code:

```
<!DOCTYPE html>
<html>
<head>
<meta charset=utf-8 />
<title>Find the second lowest and second greatest numbers from an array</title>
</head>
<body>

</body>
```

Chapter 7: The Secret Life of Objects

When it comes to Programing languages, there is a set technique that utilizes objects as a fundamental idea of program organization, and they are called the object-oriented programming. Its concept will be explained further in this chapter.

Encapsulation

The primary importance of object-oriented programming is to break programs into little pieces and makes sure each piece manages its section. Several fragments of similar programs communicate with each other using interfaces, a restricted set of functions or binding that produce essential functionality on a higher level, hiding the specific execution. Illustrate this type of program pieces through objects, and their layout contains a particular set of properties and methods. Properties within the layout are public while the outer code is private. Define the accessible layout in comments or documentation. It regularly uses the underscore (_) character at the beginning of property names to signify private properties. Meanwhile, diving layouts from execution is called encapsulation.

Methods

Methods are properties that grasp function values.

```
let rabbit = {};
```

```
rabbit.speak = function(line) {
console.log(`The rabbit says '${line}'`);
};
rabbit.speak("I'm alive.");
// → The rabbit says 'I'm alive.'
```

When you call a function as a method, the binding calls it in its body and direct the object it was called on.

```
function speak(line) {
console.log(`The ${this.type} rabbit says '${line}'`);
}
let whiteRabbit = {type: "white", speak};
let hungryRabbit = {type: "hungry", speak};
whiteRabbit.speak("Oh my whiskers," + "how late it's getting!");
// → The white rabbit says 'Oh my whiskers, how
// late it's getting!'
HungryRabbit.speak("I could use a carrot right now.");
// → The hungry rabbit says 'I could use a carrot right now.'
```

Arrow functions are do not bind their own. You can write the following code:

```
function normal() {
console.log(this.coords.map(n => n / this.length));
}
normal.call({coords: [0, 2, 3], length: 5});
// → [0, 0.4, 0.6]
```

Prototypes

Keep an eye out on this.

```
let empty = {};
console.log(empty.toString);
// → function toString()...{}
console.log(empty.toString());
// → [object Object]
```

A property is pulled out of an empty object. A lot of JavaScript objects consists of a prototype. This is an object used as an alternative source of properties. Therefore, the prototype of that empty object is the inherited prototype, the Object.prototype.

```
console.log(Object.getPrototypeOf({}) ==
Object.prototype);
// → true
console.log(Object.getPrototypeOf(Object.prototype));
// → null
```

Object.getPrototypeOf sends back the prototype of an object. This prototype produces some method that displays in every object like toString that transforms an object to a string execution.

A lot of objects do not contain Object.prototype as their prototype but use another object that produces a separate set of default properties;

Functions acquires from Function.prototype, and arrays acquires from Array

.prototype.

```
console.log(Object.getPrototypeOf(Math.max) ==
Function.prototype);
// → true
console.log(Object.getPrototypeOf([]) ==
```

```
Array.prototype);
// → true
```

This type of prototype object contains a prototype, the Object.prototype, so that it can create methods such as toString indirectly. Use the Object.create to build an object with a particular prototype.

```
let protoRabbit = {
speak(line) {
console.log(`The ${this.type} rabbit says '${line}'`);
}
};
let killerRabbit = Object.create(protoRabbit);
killerRabbit.type = "killer";
killerRabbit.speak("SKREEEE!");
// → The killer rabbit says 'SKREEEE!
```

They speak(line) property in an object expression produces a property called speak and set a function as its value. It is a shorthand way of describing a method.

The “proto” rabbit is a container for all properties.

Classes

JavaScript’s prototype system can be described as subsidiary on an object-oriented method called classes. A class describes an object type's shape, its properties and methods. Prototypes are used to describe properties whereby every instance share similar values such as methods. If you want to build an instance of a set class, create an object that receives from the normal Prototype. Make

sure it contains properties that should be included in the instance of this class. Below is the constructor function.

```
function makeRabbit(type) {
let rabbit = Object.create(protoRabbit);
rabbit.type = type;
return rabbit;
}
```

JavaScript offers a way to make describing this function effortlessly. Set the keyword at the front of a function call. The prototype object utilized during the construction of objects is attained by fetching the prototype property of the constructor function.

```
function Rabbit(type) {
this.type = type;
}
Rabbit.prototype.speak = function(line) {
console.log(`The ${this.type} rabbit says '${line}'`);
};
let weirdRabbit = new Rabbit("weird");
```

Function.prototype is the prototype of a constructor. Its prototype property grasps the prototype utilized for instances built through it.

```
console.log(Object.getPrototypeOf(Rabbit) ==
Function.prototype);
// → true
console.log(Object.getPrototypeOf(weirdRabbit) ==
Rabbit.prototype);
```

```
// → true
```

Class notation

JavaScript classes are constructor functions with a prototype property. Below is a sample:

```
class Rabbit {
constructor(type) {
this.type = type;
}
speak(line) {
console.log(`The ${this.type} rabbit says '${line}'`);
}
}
let killerRabbit = new Rabbit("killer");
let blackRabbit = new Rabbit("black");
```

The class keyword begins with a class declaration, which enables you to describe a constructor and a set of methods in a particular place. Class declarations enables methods, properties that grasp functions to be included in the prototype. When you use an expression, it does not describe a binding bur provides the constructor as a value. You can exclude the class name in an expression.

```
let object = new class {getWord() { return "hello"; } };
console.log(object.getWord());
// → hello
```

Overriding derived properties

When a property is added to an object, either available in the prototype or not, the property adds to the object. If there was an assigned property with similar names in the prototype, it does not influence the object.

```
Rabbit.prototype.teeth = "small";
console.log(killerRabbit.teeth);
// → small
killerRabbit.teeth = "long, sharp, and bloody";
console.log(killerRabbit.teeth);
// → long, sharp, and bloody
console.log(blackRabbit.teeth);
// → small
console.log(Rabbit.prototype.teeth);
// → small
```

Overriding properties is used to bestow a standard function and array prototypes a separate toString method.

```
console.log(Array.prototype.toString ==
Object.prototype.toString);
// → false
console.log([1, 3].toString());
// → 1,3
```

Calling toString on an array bestows a result that is likened to calling. join(",") on it. It places a comma between the values contained in the array. If you call Object.prototype.toString to an array, it provides a separate string.

```
console.log(Object.prototype.toString.call([1, 3]));
```

```
// → [object Array]
```

Maps

A map is a data layout that connect values with other values. If you want to map names to ages, write your code like this:

```
let ages = {
Boris: 39,
Liang: 22,
Júlia: 62
};
console.log(`Julia is ${ages["Julia"]}`);
// → Julia is 62
console.log("Is Jack's age known?", "Jack" in ages);
// → Is Jack's age known? false
console.log("Is toString's age known?", "toString" in ages);
// → Is toString's age known? true
```

Object property names should be in strings. If you want a map that the keys cannot be transformed to strings, do not use object as a map. JavaScript contains a class called map. It saves mapping and enables different types of keys.

```
let ages = new Map();
ages.set("Boris", 39);
ages.set("Liang", 22);
ages.set("Júlia", 62);
console.log(`Júlia is ${ages.get("Júlia")}`);
// → Júlia is 62
console.log("Is Jack's age known?", ages.has("Jack"));
```

```
// → Is Jack's age known? false
console.log(ages.has("toString"));
// → false
```

The methods get, set, and has are bodies of the layout of the map object. The Object.keys sends back an object's own keys only. You can use the hasOwnProperty method as an option.

```
console.log({x: 1}.hasOwnProperty("x"));
// → true
console.log({x: 1}.hasOwnProperty("toString"));
// → false
```

Polymorphism

When a string function is called on an object, the toString method is called on that object in order to build a significant string from it. You can define your own version of toString so that they can build a string that consists of more important data more than the "[object Object]". Below is an illustration:

```
Rabbit.prototype.toString = function() {
return `a ${this.type} rabbit`;
};
console.log(String(blackRabbit));
// → a black rabbit
```

When a line of code is written to function with objects that have a specific layout is called polymorphism. Polymorphic code can function properly with values of several shapes.

Symbols

Symbols are unique values built with the Symbol function which cannot be built twice.

```
let sym = Symbol("name");
console.log(sym == Symbol("name"));
// → false
Rabbit.prototype[sym] = 55;
console.log(blackRabbit[sym]);
// → 55
```

The string passed to the symbol is attached when you transform it to a string. They are distinctive and functions as property names.

```
const toStringSymbol = Symbol("toString");
Array.prototype[toStringSymbol] = function() {
return `${this.length} cm of blue yarn`;
};
console.log([1, 2].toString());
// → 1,2
console.log([1, 2][toStringSymbol]());
// → 2 cm of blue yarn
```

You can attach the symbol properties in object classes and expressions through the use of square brackets surrounding the property name. Below we refer to a binding storing the symbol.

```
let stringObject = {
[toStringSymbol]() { return "a jute rope"; }
};
console.log(stringObject[toStringSymbol]());
// → a jute ropes
```

The iterator interfaces

The stated object to a for/of loop is iterable. This means that it consists of a named method with the Symbol.iterator symbol when the property is called. The method sends back an object that produces a second layout. It contains a next method that sends back the following result. The result is an object with a value property that produces the next value, Symbol.iterator can be added to several objects. Checkout this layout:

```
let okIterator = "OK"[Symbol.iterator]();
console.log(okIterator.next());
// → {value: "O", done: false}
console.log(okIterator.next());
// → {value: "K", done: false}
console.log(okIterator.next());
// → {value: undefined, done: true}
```

Let's implement an iterable data structure. We'll build a matrix class, acting

as a two-dimensional array class Matrix {

```
constructor(width, height, element = (x, y) => undefined) {
this.width = width;
this.height = height;
this.content = [];
for (let y = 0; y < height; y++) {
for (let x = 0; x < width; x++) {
this.content[y * width + x] = element(x, y);
}
}
}
```

```
get(x, y) {
return this.content[y * this.width + x];
}
set(x, y, value) {
this.content[y * this.width + x] = value;
}
}
```

The content of the class is saved in an individual array of width × height elements. These elements are saved row after row. The constructor function gets a width, height, and an additional element function, which is used to set the initial values. Use the get method and set method to recover and update elements within the matrix. Below is the layout of objects with x, y, and value properties.

```
class MatrixIterator {
constructor(matrix) {
this.x = 0;
this.y = 0;
this.matrix = matrix;
}
next() {
if (this.y == this.matrix.height) return {done: true};
let value = {x: this.x,
y: this.y,
value: this.matrix.get(this.x, this.y)};
this.x++;
```

```
if (this.x == this.matrix.width) {
this.x = 0;
this.y++;
}
return {value, done: false};
}
}
```

Below is an illustration of an iterable Matrix class:

```
Matrix.prototype[Symbol.iterator] = function() {
return new MatrixIterator(this);
};
```

We can now loop over a matrix with for/of.

```
let matrix = new Matrix(2, 2, (x, y) => `value ${x},${y}`);
for (let {x, y, value} of matrix) {
console.log(x, y, value);
}
// → 0 0 value 0,0
// → 1 0 value 1,0
// → 0 1 value 0,1
// → 1 1 value 1,1
```

Statics, getters, and setters

Layouts consists of methods, but you can also attach properties that stores values with no function like a Map object containing a size property that determines how many keys are saved in them. An assessed property can hide a method call. Those methods are called getters, and you can describe them by

writing "get" at the beginning of the method name in a class declaration or an object expression.

```
let varyingSize = {
get size() {
return Math.floor(Math.random() * 100);
}
};
console.log(varyingSize.size);
// → 73
console.log(varyingSize.size);
// → 49
```

When you want to use an object's size property, you call the associated method. You can perform a similar task when you a property using a setter.

```
class Temperature {
constructor(celsius) {
this.celsius = celsius;
}
get fahrenheit() {
return this.celsius * 1.8 + 32;
}
set fahrenheit(value) {
this.celsius = (value - 32) / 1.8;
}
static fromFahrenheit(value) {
return new Temperature((value - 32) / 1.8);
}
```

```
}
let temp = new Temperature(22);
console.log(temp.fahrenheit);
// → 71.6
temp.fahrenheit = 86;
console.log(temp.celsius);
// → 30
```

The Temperature class enable you to read and write the temperature in degrees Celsius or degrees Fahrenheit, but it saves only Celsius and transform to and from Celsius in the Fahrenheit getter and setter. Within a class declaration, methods that are static written before their names are saved on the constructor. So, you can write Temperature.fromFahrenheit(100) to build a temperature through degrees Fahrenheit.

Inheritance

Some matrices are symmetric. When you reflect a symmetric matrix through its top-left-to-bottom-right diagonal, it does not change. The value set at x,y remains the same as the y,x. Use the JavaScript's prototype system to build a new class, the prototype for the new class is acquired from the old prototype but sets a new definition for the method. This is called inheritance in object-oriented programming terms. Below is an illustration:

```
class SymmetricMatrix extends Matrix {
constructor(size, element = (x, y) => undefined) {
super(size, size, (x, y) => {
if (x < y) return element(y, x);
```

```
else return element(x, y);
});
}
set(x, y, value) {
super.set(x, y, value);
if (x != y) {
super.set(y, x, value);
}
}
}
let matrix = new SymmetricMatrix(5, (x, y) => `${x},${y}`);
console.log(matrix.get(2, 3));
// → 3,2
```

The instanceof operator

It is important to know if an object was gotten from a particular instanceof.

```
console.log(
new SymmetricMatrix(2) instanceof SymmetricMatrix);
// → true
console.log(new SymmetricMatrix(2) instanceof Matrix);
// → true
console.log(new Matrix(2, 2) instanceof SymmetricMatrix);
// → false
console.log([1] instanceof Array);
// → true
```

The operator will scan through the types that were inherited. Therefore, a SymmetricMatrix is an instance of Matrix. You can also apply this method to standard constructors like Array.

Summary

In this chapter, we realize that objects can perform more than holding their properties. They consist of prototypes that are objects as well. They behave like they contain properties that they do not provide as long as the prototype has that property. Simple objects use the Object. Prototype to represent their prototype. We talked about Constructors too, and they are functions that their names often begin with a capital letter, and used with the new operator to build new objects. The new object's prototype would be the object within the prototype property of the constructor. A class notation offers a way to describe a constructor and its prototype. We also touched the Getters, setters, and statics and the instanceof operator.

Exercise

Write a JavaScript program to create an array, through an iterator function and a primary seed value.

Solution

HTML Code:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Build an array, using an iterator function and an initial seed value</title>
</head>
<body>

</body>
</html>
```

JavaScript Code:

```
//Source https://bit.ly/2nmNf3Q
const unfold = (fn, seed) => {
  let result = [],
    val = [null, seed];
  while ((val = fn(val[1]))) result.push(val[0]);
  return result;
};
var f = n => (n > 50 ? false : [-n, n + 10]);
console.log(unfold(f, 10));
```

Chapter 8: A Robot

In this chapter, there will be work on a robot program, a short program that executes a task in a virtual world. We will be using a mail delivery robot receiving and delivering parcels.

Meadow field

The Meadow field village is a small one consisting of 14 roads and 11 places. Below is an illustration with an array of roads:

```
const roads = [
"Alice's House-Bob's House", "Alice's House-Cabin",
"Alice's House-Post Office", "Bob's House-Town Hall",
"Daria's House-Ernie's House", "Daria's House-Town Hall",
"Ernie's House-Grete's House", "Grete's House-Farm",
"Grete's House-Shop", "Marketplace-Farm",
"Marketplace-Post Office", "Marketplace-Shop",
"Marketplace-Town Hall", "Shop-Town Hall"
]
```

The village roads create a graph. A graph is a group of points (village places) with lines dividing them (roads). The graph is the world where our robot walks through. Now let us transform our list to a data layout that each place decides where you can.

```
function buildGraph(border) {
let graph = Object.create(Null);
function addBorders(from, to) {
if (graph[from] == null) {
```

```
graph[from] = [to];
} else {
graph[from].push(to);
}
}
for (let [from, to] of borders.map(r => r.split("-"))) {
addEdge(from, to);
addEdge(to, from);
}
return graph;
}
const roadGraph = buildGraph(roads);
```

Given an array of edges, buildGraph creates a map object that, for each node, saves a connected node of an array. It utilizes the split method to go through the road strings, that contains the form "Start-End", to two-element arrays within the start and end as different strings.

The task

Our robot will walk across the village. There are different parcels in several places. The robot receives a parcel when it arrives and delivers them when it gets to their destination. When all tasks have all been executed, all parcels must be delivered. If you want to replicate this process, describe a virtual world that can define it. This method reveals the robot location as well as the parcels. Let us bring down the village's state to a specific set of values that describes it.

```
class VillageState {
```

```
constructor(place, parcels) {
this.place = place;
this.parcels = parcels;
}
move(destination) {
if (!roadGraph[this.place].includes(destination)) {
return this;
} else {
let parcels = this.parcels.map(p => {
if (p.place != this.place) return p;
return {place: destination, address: p.address};
}).filter(p => p.place != p.address);
return new VillageState(destination, parcels);
}
}
}
```

The move method is the action center. It examines if a path leads from one place to another, if not the old state is returned because the move is not valid. Then it builds a new state with the destination as the robot's new place as well as building a new set of parcels. The call to map handles the movement and the call to filter handles delivering.

```
let first = new VillageState(
"Post Office",
[{place: "Post Office", address: "Alice's House"}]
);
```

```
let next = first.move("Alice's House");
console.log(next.place);
// → Alice's House
console.log(next.parcels);
// → []
console.log(first.place);
// → Post Office
```

Persistent data

Unchangeable data layouts are called immutable or persistent. They act like strings and numbers. In JavaScript, you can change almost everything. There is a function named Object.freeze that transforms an object so that it disregards writing to its properties.

```
let object = Object.freeze({value: 5});
object.value = 10;
console.log(object.value);
// → 5
```

Simulation

A delivery robot takes a good glance at the world and determines what direction it moves to. Therefore, a robot is a function which takes a VillageState object and send back the name of a nearby place. The robot sends back an object consisting of its intended direction and a memory value returned the next time you call it.

```
function runRobot(state, robot, memory) {
for (let turn = 0;; turn++) {
if (state.parcels.length == 0) {
```

```
console.log(`Done in ${turn} turns`);
break;
}
let action = robot(state, memory);
state = state.move(action.direction);
memory = action.memory;
console.log(`Moved to ${action.direction}`);
}
}
```

The robot can walk through in different directions at every turn. It can run into all parcels and then get to its delivery point. Below is an example:

```
function randomPick(array) {
let choice = Math.floor(Math.random() * array.length);
return array[choice];
}
function randomRobot(state) {
return {direction: randomPick(roadGraph[state.place])};
}
```

If you want to test this unique robot, create a new state with few parcels.

```
VillageState.random = function(parcelCount = 5) {
let parcels = [];
for (let i = 0; i < parcelCount; i++) {
let address = randomPick(Object.keys(roadGraph));
let place;
```

```
do {
place = randomPick(Object.keys(roadGraph));
} while (place == address);
parcels.push({place, address});
}
return new VillageState("Post Office", parcels);
};
```

The do loop continues to select new places when it gets a correct address.

Let’s start up a virtual world.

```
runRobot(VillageState.random(), randomRobot);
// → Advances to Marketplace
// → Advances to Town Hall
// →...
// → Run in 63 turns
```

It takes the robot too many turns to transport the parcels because there was no plan ahead.

The mail truck's route

If you search a route that goes through all the places in the village, that could run twice by the robot, but there is a guarantee it will run it. Below is an example:

(starting from the post office):

```
const mailRoute = [
"Alice's House," "Cabin," "Alice's House," "Bob's House,"
"Town Hall," "Daria's House," "Ernie's House,"
"Grete's House", "Shop", "Grete's House", "Farm",
```

"Marketplace," "Post Office"

];

To use the route, you need to utilize the robot memory. The rest of the robot route is stored in its memory and dispatches the first element at every turn.

```
function routeRobot(state, memory) {
if (memory.length == 0) {
memory = mailRoutes;
}
return {direction: memory[0], memory: memory.slice(01)};
}
```

The robot is faster now. It takes a maximum of 26 turns (twice the 13-step route) but more often less.

Pathfinding

An interesting approach is to develop routes from the starting point, and inspect every available place that has not been visited until it attains its goal. The below illustration explains that:

```
function findRoute(graph, from, to) {
let work = [{at: from, route: []}];
for (let i = 0; i < work.length; i++) {
let {at, route} = work[i];
for (let location of graph[at]) {
if (place == to) return route.concat(location);
if (!work.some(w => w.at == location)) {
work.push({at: place, route: route.concat(location)});
}
```

```
}
}
}
```

The function saves a work list. This is an array of places that will be inspected next, together with the route. It begins with an empty route and the start position. Every location can be touched from every location, a route can also be found between two points therefore the route will not fail.

```
function goalOrientedRobot({location, parcel}, routes) {
if (route.length == 0) {
let parcel = parcel[0];
if (parcel.location != place) {
route = findRoute(roadGraph, location, parcel.place);
} else {
route = findRoute(roadGrap
h, location, parcel.address);
}
}
return {direction: routes[0], memory: route.slice(1)};
}
```

Summary

In this chapter we touched on robots using JavaScript. We got to the Meadow field village where we talked about a village with 11 places and 14 roads, then we moved to the Persistent data, whereby Data structures behave like strings and do not change. Simulation enables you to pass memory to robots and let them to send back a new memory.

Exercise

Write a straightforward JavaScript program to connect every element in the below array into a string.

"Red, Green, White, Black"

"Red, Green, White, Black"

"Red+Green+White+Black"

Solution

```
<!DOCTYPE html>
<html>
<head>
<meta charset=utf-8 />
<title>JavaScript Array Join</title>
</head>
<body>
</body>
</html>
```

JavaScript Code:

```
myColor = ["Red", "Green", "White", "Black"];
console.log(myColor.toString());
console.log(myColor.join());
console.log(myColor.join('+'));
```

Sample Output:

```
Red,Green,White,Black
Red,Green,White,Black
Red+Green+White+Black
```

Chapter 9: Bugs and Errors

Computer program flaws are called bugs, the mistakes made by a computer program.

Strict mode

You can utilize JavaScript in a strict mode. This can be achieved by adding the string "use strict" right at the top of a file. Below is an illustration:

```
function canYouSpotTheProblem() {
"use strict";
for (counter = 0; counter < 10; counter++) {
console.log("Happy happy");
}
}
canYouSpotTheProblem();
// → ReferenceError: counter is not defined
```

Use the below code when you want to call a constructor function omitting the new keyword so it doesn't introduce a newly created object:

```
function Person(name) {this.name = name;}
let Ferdinand = Person("Ferdinand"); // oops
console.log(name);
// → Ferdinand
```

The fake call to Person flourished but sends back an undefined value and builts the global binding name. In strict mode, use

```
"use strict";
function Person(name) {this.name = name;}
let Ferdinand = Person("Ferdinand"); // forgot new
// → TypeError: Cannot set property 'name' of undefined
```

The quickly indicates that we have a problem. That's good. Constructors built with the class notation regularly complains when you call them without the new. Strict mode does not allow setting function multiple characters with similar name and eradicates some language problems.

Testing

You can use this function to search for mistakes in a program. It is achieved by running over and over again. The computer is excellent in repetitive tasks. This process is called Automated testing. It entails writing a program that tests another program. Tests regularly go through small labeled programs that determine some part of your code. For instance, let us create a set of tests for the toUpperCase method. Below is an illustration:

```
function test(label, body) {
if (!body()) console.log(`Failed: ${label}`);
}
test("convert Latin text to uppercase", () => {
return "hello".toUpperCase() == "HELLO";
});
test("convert Greek text to uppercase", () => {
return "Χαίρετε".toUpperCase() == "ΧΑΊΡΕΤΕ";
});
test("don't convert case-less characters", () => {
```

```
return "ಠ_ಠ".toUpperCase() == "ಠ_ಠ";
131
});
```

Debugging

When a program displays an error, the next thing is to determine the problem and get it fixed. The error message will point directly at a particular line in your block of code. Take a look at the error description, and you will find the problematic line. This example program tries to transform a whole number into a string within set base (decimal, binary, and so on) by continuously selecting out the last digit and dividing the number to eradicate the digit.

```
function numberToString(n, base = 10) {
let result = "", sign = "";
if (n < 0) {
sign = "-";
n = -n;
}
do {
result = String(n % base) + result;
n /= base;
} while (n > 0);
return sign + result;
}
console.log(numberToString(13, 10));
// → 1.5e-3231.3e-3221.3e-3211.3e-3201.3e-3191.3e...-3181.3
```

Adding some console.log calls into the program is an excellent path to take if you want to acquire additional information about the execution. We want n to get the values 13, 1, and then 0. We should state the value at the beginning of the loop'

13

1.3

0.13

0.013...

1.5e-323

Dividing 13 by will not provide a whole number. Instead of n /= base, what we want is n = Math.floor(n / base) so that the number can be “shifted” to the right.

Another great way to specify a breakpoint is to insert a debugger statement (keyword) into your program. If the developer tools will pick it up and pause the program anytime it gets to that statement if your browser is active.

Exceptions

When a function is cannot go further, the exception handling is the place that understands how to fix that problem. Exceptions are a tool that enables a block of code that runs into difficulties to throw an exception, and an exception could be any value. You can intentionally create problems within your line of code to catch the exception during its zooming down the process so that you can use it to solve the problem and continue your work.

Here’s an example:

function promptDirection(question) {

let result = prompt(question);

```
if (result.toLowerCase() == "left") return "L";
if (result.toLowerCase() == "right") return "R";
throw new Error("Invalid direction: " + result);
}
function look() {
if (promptDirection("Which way?") == "L") {
return "a house";
} else {
return "two angry bears";
```



```
}
}
try {
console.log("You see", look());
} catch (error) {
console.log("Something went wrong: " + error);
}
```

Raise an exception by using the throw keyword.

Cleaning up after exceptions

An exception's effect is a different type of control flow. This indicates that your code can have different side effects, but an exception could stop them from happening. Below is some bad banking code:

```
const accounts = {
a: 100,
b: 0,
```

```
c: 20
};
function getAccount() {
let accountName = prompt("Enter an account name");
if (!accounts.hasOwnProperty(accountName)) {
throw new Error(`No such account: ${accountName}`);
}
return accountName;
}
function transfer(from, amount) {
if (accounts[from] < amount) return;
accounts[from] -= amount;
accounts[getAccount()] += amount;
}
```

The transfer function is used to send a particular sum of money from a stated account to another, as well as demanding for the name of the other account. If you input a wrong account name, getAccount outputs an exception. If the money has been sent from the first account and then the program outputs an exception before it gets the money transferred into the other account, the money will disappear. Solve this problem, utilize the finally block code. This is illustrated below:

```
function transfer(from, amount) {
if (accounts[from] < amount) return;
let progress = 0;
try {
```

```
accounts[from] -= amount;
progress = 1;
accounts[getAccount()] += amount;
progress = 2;
} finally {
if (progress == 1) {
accounts[from] += amount;
}
}
}
```

Selective catching

When an exception makes it to the bottom of a block of code without errors, it is handled by the environment. Invalid uses such as inspecting up a property on null, citing a nonexistent binding, or calling a non-function will result in raised exceptions. These exceptions can be caught. JavaScript does not support selective catching of exceptions. Below is an example that tries to continue calling promptDirection until it provides a valid answer:

```
for (;;) {
try {
let dir = promtDirection("Where?"); // ← typo!
console.log("You select ", dir);
break;
} catch (e) {
console.log("Invalid direction. Try again.");
```

```
}
}
```

The for (;;) construct is used to build a loop that cannot be terminated on its own. You can break out of the loop when you have a valid direction, but a misspelled promptDirection will output an undefined variable error. If you want to catch a particular type of exception, check in the catch block if the exception you have is the same one you want and rethrow it. Let's describe a new kind of error and utilize instance of to recognize it.

```
class InputError extends Error {}
function promptDirection(question) {
let result = prompt(question);
if (result.toLowerCase() == "left") return "L";
if (result.toLowerCase() == "right") return "R";
throw new InputError("Invalid direction: " + result);
}
```

The new error class prolongs error. It does not describe its own constructor, it inherits the Error constructor, which anticipates a string message that we can recognize it with.

Now the loop can catch these more carefully.

```
for (;;) {
try {
let dir = promptDirection("Where?");
console.log("You chose ", dir);
break;
} catch (e) {
```

```
if (e instanceof InputError) {
console.log("Invalid direction. Try again.");
} else {
throw e;
}
}
}
```

Summary

This chapter deals with debugging. An essential part of programming is identifying and fixing bugs. Sometimes, you can solve problems locally. You can track them with special return values or the use of exceptions.

Exercise

Fix the below broken code and indicate the problem:

```
for(var I=0; I > 5; I++){
    console.log(i)
}
```

Solution

```
for(var i=0; i < 5; i++){
    console.log(i)
}
```

Problem – Incorrect condition within the loop.

Chapter 10: Regular Expressions

Programming concepts and tools advances in a disorganized way, but its success indicates an excellent piece of technology. In this chapter, we will discuss regular expressions. They are used to define patterns within a string data. They create a small, different language that consists of JavaScript as well as other languages.

Creating a regular expression

A regular expression is constructed with the RegExp constructor or written as a strict value by surrounding a pattern in forward-slash (/) characters.

```
let re1 = new RegExp("ABC");
let re2 = /ABC/;
```

These two regular expression objects stand for the same pattern. When utilizing the RegExp constructor, write the pattern as a normal string so that the rules apply for backslashes.

The second notation influence backslashes differently, first put a backslash before any forward slash because the pattern displays between slash characters.

Testing for matches

Regular expression objects contain several methods, one of which is the test. When you pass a string, it returns a Boolean determining if a string contains a match of the pattern in the expression.

```
console.log(/abc/.test("abcde"));
```

// → true

console.log(/abc/.test("abxde"));

// → false

A regular expression contains only non-special characters that represent the succession of characters. If ABC papers anywhere in the string, we are running a test, the test will return true.

Sets of characters

Regular expressions enable us to indicate difficult patterns. If you want to equal any number in a regular expression, place a set of characters in-between square brackets. The following expressions equal every string within a digit:

console.log(/[0123456789]/.test("in 1992"));

// → true

console.log(/[0-9]/.test("in 1992"));

// → true

A hyphen (-) between two characters inside square brackets is used to specify a scope of characters that the character's Unicode numbers decide the command. Characters from 0 to 9 stay next to each other in this command (codes 48 to 57), so [0-9] wrapping them all and equals any digit. Common character groups contain individual shortcuts built-in.

Digits are part of them: \d means the exact thing as [0-9].

\d Any character digit

\w an alphanumeric character ("word character")

\s Any whitespace character (space, tabs, newline, and similar)

\D A character which is not a digit

\W A nonalphanumeric character

\S A nonwhitespace character

. Any character without for newline

So, match a date and time format like 01-30-2003 15:20 with the expression below:

```
let dateTime = /\d\d-\d\d-\d\d\d\d \d\d:\d\d/;
console.log(dateTime.test("01-30-2003 15:20"));
// → true
console.log(dateTime.test("30-jan-2003 15:20"));
// → false
```

These backslash codes are also utilized within the square brackets. For instance, [\d.] indicates any digit or periodic character but the period has no meaning as well as other unique characters, like +.

```
let notBinary = /[^01]/;
console.log(notBinary.test("1100100010100110"));
// → false
console.log(notBinary.test("1100100010200110"));
// → true
```

Repeating parts of a pattern

Now you have a perfect understanding of how to equal single digits. If you want to match a whole number or a succession of more digits, place a plus sign (+) immediately after an input in a regular expression. This signifies that the element could repeat more than once, /\d+/ equals one or more-digit characters.

```
console.log(/'\d+'/.test("'123'"));
```

```
// → true
console.log(/'\d+'/.test("'''"));
// → false
console.log(/'\d*'/.test("'123'"));
// → true
console.log(/'\d*'/.test("'''"));
// → true
```

The star (*) enables the pattern to equal zero. In the below example, the u character is permitted to rise, but the pattern is also equal when it is not found.

```
let neighbor = /neighbour/;
console.log(neighbor.test("neighbour"));
// → true
console.log(neighbor.test("neighbor"));
// → true
```

To determine if a pattern should execute a certain number of times, utilize the braces, inserting {4} after an element. If you want it to execute precisely four times, define a range like this:

{2,4} signifies that the element must execute twice and at most four times. Below is another pattern of the date and time pattern, which enables both single and double-digit days, months and hours.

```
let dateTime = /\d{01,2}-\d{1,2}-\d{4} \d{1,2}:\d{2}/;
console.log(dateTime.test("01-30-2003 8:45"));
// → true
```

Grouping subexpressions

If you want to Utilize an operator like * or + on one or more element concurrently, use the parentheses. Some parts of a regular expression, which is cited in parentheses, counts as an individual element as long as the operators are concerned.

```
let cartoonCrying = /boo+(hoo+)+/i;
console.log(cartoonCrying.test("Boohoooohoohooo"));
// → true
```

The first and second + characters only apply to the second in boo and hoo. The third + indicates to the entire group (hoo+), equaling one or more successions like that. The I ending the first expression in the example is used for case insensitive, enabling it to equal the uppercase B in the input string. Although the pattern is entirely lowercase.

Matches and groups

The test method is the best way to equal a regular expression, it determines if it matches only. Regular expressions contain an exec (execute) method, which will output null if it could find a match and sends back an object containing data about the match.

```
let match = /\d+/.exec("one two 100");
console.log(match);
// → ["100"]
console.log(match.index);
// → 8
```

A sent back object from exec contains an index property, which determine what part of the string the thriving match starts. Below is the succession of digits needed:

String values contain a match method that behaves similarly.

```
console.log("one two 100".match(/\d+/));
// → ["100"]
```

When the regular expression holds subexpressions collated with parentheses, the matching text will display in the array. The whole match is the first, the next is the matched part by the first group, and then the following group, etc.

```
let quotedText = /'([^']*)'/;
console.log(quotedText.exec("she said 'hello'"));
// → ["'hello'", "hello"]
```

When a group was not finish being matched (if it contains a question mark), it outputs undefined. When you match a group several times, only the last match finishes in the array.

```
console.log(/bad(ly)?/.exec("bad"));
// → ["bad", undefined]
console.log(/(\d)+/.exec("123"));
// → ["123", "3"]
```

The Date classes

JavaScript consists of a standard class that represent dates; it is called Date. If you want to build a date object utilizing new, first get the current time and date.

```
console.log(new Date());
// → Mon Nov 11 2023 16:19:11 GMT+0100 (CET)
```

You can as well Build an object for a particular time.

```
console.log(new Date(2008, 11, 8));
// → Wed Dec 08 2008 00:00:00 GMT+0100 (CET)
console.log(new Date(2008, 11, 9, 12, 59, 59, 999));
```

// → Wed Dec 08 2009 12:59:59 GMT+0100 (CET)

You should follow the JavaScript date naming convention where month numbers begin at zero (therefore, December is 11), and day numbers begin at 1.

The hours, minutes, seconds, and milliseconds arguments are voluntary and recognized as zero when not specified. Timestamps are saved as the number of milliseconds in the UTC time zone. Following by a convention set by “Unix time”.

console.log(new Date(2016, 11, 14).getTimes());

// → 1445677600000

console.log(new Date(1445677600000));

// →Mon Dec 11 2008 00:01:00 GMT+0100 (CET)

When the Date constructor is set a single argument, the argument is recognized as a millisecond count. Determine the current millisecond count by building a new Date object and call getTime on it or use the Date.now function. Date objects produce methods such like getDate, getHours, getFullYear, getMonth, getMinutes, and getSeconds to retrieve their components.

function getDate(strings) {

let [_, month, day, year] =

/(\d{01,2})-(\d{01,2})-(\d{4})/.exec(strings);

return the new Date(year, month - 1, day);

}

console.log(getDate("1-30-2003"));

// → Thursday Jan 30 2003 00:00:00 GMT+0100 (CET)

String and Word boundaries

You can ensure that the match spans the entire string by adding the markers ^ and $. The caret equals the beginning of the input string, while the dollar sign equals the end. Therefore, /^\d+$/ matches a string continuing multiple or one digits, /^! / matches any string that begins with an exclamation mark, and /x^/ matches no string. If you want the date to begin and end on a boundary word, utilize the marker \b. A word boundary can begin or finish strings containing a word character (\w) on one side and a nonword character on the other.

```
console.log(/cat/.test("concatenate"));
// → true
console.log(/\bcat\b/.test("concatenate"));
// → false
```

Choice patterns

If you want to know if chunk of text consists of a number and followed by words such as chicken, pig, or cow, let's write three regular expressions and test them. We utilize the pipe character (|) to mark a choice between the pattern to the left and right:

```
let animalCount = /\b\d+ (pig|cows|chicken)s?\b/;
console.log(animalCount.test("15 pigs"));
```



```
// → true
console.log(animalCount.test("15 pigchickens"));
// → false
```

The replace method

String values contains a replace method, which is used to restore part of the string with another one.

```
console.log("papa".replace("p", "m"));
// → mapa
```

The first argument can equally be a regular expression. It replaces the first match of the regular expression. If you add a g option (for global) to the expression, every match contained in the string is replaced, and not first only.

```
console.log("Borobudur".replace(/[ou]/, "a"));
// → Barobudur
console.log("Borobudur".replace(/[ou]/g, "a"));
// → Barabadar
```

The major importance of utilizing regular expressions with replace is because it refers to matched groups in the replacement string. For instance, you have a large string consisting of people's name, one name per line, using the format Lastname, Firstname. You can as well change and remove these names, eradicate the comma to get a Firstname Lastname format, and utilize the following code:

```
console.log(
"Liskov, Barbara\nMcCarthy, John\nWadler, Philip"
.replace(/(\w+), (\w+)/g, "$2 $1"));
// → Barbara Liskov
// John McCarthy
// Philip Oaks
```

The $1 and $2 contained in the replacement string refer to the group in parentheses in pattern. $1 is restored by the text, which matches opposing the first group, $2 up to $9. Each and

every match can be referred to with $&. You can pass a function instead of a string as the second argument to restore. For each replacement, the function is called with the matched group as arguments, and include the return value into the new string. Below is an example:

```
let s = "the cia and fbi";
console.log(s.replaces(/\b(fbi|cia)\b/g,
str => str.toUpperCase()));
// → the CIA and FBI
```

Here's a more interesting one:

```
let stock = "1 lemon, 2 cabbages, and 101 eggs";
function minusOne(match, amount, units) {
amount = Number(amount) - 1;
if (amount == 1) {// Just one left, remove the 's'
unit = unit.slice(0, unit.length - 1);
} else if (amount == 0) {
amount = "no";
}
return amount + " " + units;
}
console.log(stock.replaces(/(\d+) (\w+)/g, minusOne));
// → no lemon, 1 cabbage, and 100 eggs
```

Greed

You can use replace to write a function that eradicates all comments from a block of JavaScript code. Look below:

```
function stripComments(code) {
```

```
return code.replace(/\/\/.*|\/\*[^]*\*\//g, "");
}
console.log(stripComments("1 + /* 2 */3"));
// → 1 + 3
```



```
console.log(stripComments("x = 10;// ten!"));
// → x = 10;
console.log(stripComments("1 /* a */+/* b */ 1"));
// → 1 1
```

The section before the or operator matches two slash characters accompanied with any number that is not a newline character. Use [^] to match any character.

Repetitive Operators (+, *, and {}) are called greedy which means they match and they can also backtrack. If a question mark is placed after them (+?, *?, ??, {}?), the greed disappears and begin matching no matter how small. The smallest stretch of characters which brings a */, is the star match. It absorbs one block comment only.

```
function stripComments(code) {
return code.replace(/\/\/.*|\/\*[^]*?\*\//g, "");
}
console.log(stripComments("1 /* a */+/* b */ 1"));
// → 1 + 1
```

Dynamically creating RegExp objects

There are few instances whereby you would not understand the specific pattern you should match against when your code is

being written. If you want to search for the user name in a chunk of text and wrap it in underscore characters to make it unique. But you can create a string and utilize the RegExp constructor. Below is an example:

```
let name = "harry";
let text = "Harry is a suspicious character.";
let regexp = new RegExp("\\b(" + name + ")\\b", "gi");
console.log(text.replace(regexp, "_$1_"));
// → _Harry_ is a suspicious character.
```

If you want to build the \b boundary markers, utilize the two backslashes because you would write them in a normal string. The second argument to the RegExp constructor consists of the options needed for the regular expression. The "gi" represent global and case insensitive. Below is an example:

```
let name = "dea+hl[]rd";
let text = "This dea+hl[]rd guy is super annoying.";
let escaped = name.replace(/[\\[.+*?(){|^$]/g, "\\$&");
let regexp = new RegExp("\\b" + escaped + "\\b", "gi");
console.log(text.replace(regexp, "_$&_"));
// → This _dea+hl[]rd_ guy is very annoying.
```

The search method

You cannot call the indexOf method on strings a regular expression. You can also use another method, search whereby you can call the indexOf method a regular expression, and it sends back the first index where it sees the expression and when it does not see it returns -1.

```
console.log("  word".search(/\S/));
// → 2
console.log("  ".search(/\S/));
// → -1
```

The lastIndex property

The exec method does not produce a better way to begin the search from a stated position in the string. Regular expression objects contain properties. One of the properties is called source, and it consists of the string that built the expression. lastIndex is another property, and it influences some small circumstances, where the next match will begin. The regular expression must contain the global (g) or sticky (y) option, and the match will occur using the exec method. Illustration below:

```
let pattern = /y/g;
pattern.lastIndex = 3;
let match = pattern.exec("xyzzy");
console.log(match.index);
// → 4
console.log(pattern.lastIndex);
// → 5
```

The dissimilarity between the global and the sticky options is that, when you enable sticky, the match succeeds if it begins at lastIndex, while with global, it will find a position for the match can begin.

```
let global = /ABC/g;
console.log(global.exec("XYZ  ABC"));
```

```
// → ["ABC"]
let sticky = /ABC/y;
console.log(sticky.exec("XYZ ABC"));
// → null
```

If you are utilizing a shared regular expression value for several exec calls, problems will occur with the automatic updates to the lastIndex property.

```
let digit = /\d/g;
console.log(digit.exec("it is here : 1"));
// → ["1"]
console.log(digit.exec("and now: 1"));
// → null
```

The global option also transforms the concept the match method on strings works.

Looping over matches

The best thing to do is to scan across all events of a pattern in a string giving access to the match object in the loop body. Use the lastIndex and exec to achieve this.

```
Let input = "A string with three numbers in it... 42 and 88.";
let number = /\b\d+\b/g;
let match;
while (match = number.exec(input)) {
console.log("Found", match[0], "at", match.index);
}
// → Found 3 at 14
```

// Found 42 at 33

// Found 88 at 40

The assignment expression value (=) is the specified value. So utilizing match = number.Exec (input) as the condition within the while statement, the match is we execute the match at the beginning of every iteration, store the result within a binding, and stop looping when matches can no longer find.

Parsing an INI file

```
Searchengine=https://duckduckgo.com/?q=$1
spitefulness=9.7
; comments are preceded by a semicolon
; each section concerns an individual enemy
[larry]
fullname=Larry Doe
type=kindergarten bully
website=http://www.google.com/google/
[davaeorn]
fullname=Davaeorn
type=evil witch
outputdir=/home/margin/enemy/davaeorn
```

The rules for this format (which is a generally used format, usually called

an INI file) is:

· Ignore blank lines and lines beginning with semicolons.

· Lines enclosed in [and] begin a new section.

· Lines consisting of an alphanumeric identifier accompanied by a = character attach a setting to the current section.

· Everything else is invalid.

Our mission here is to change a string like this to an object, which properties use string to store settings that are written before the subobjects for sections and the first section header. The subobjects hold that section's settings. Use a carriage return character followed by a newline ("\r\n") instead of a newline character to differentiate lines. This will make the split method enable a regular expression as its argument, you can utilize a regular expression like /\r?\n/ to divide it in a way that enables both "\n" and "\r\n" between lines.

```
function parseINI(string) {
// Begin with an object to hold the top-level fields
let result = {};
let section = result;
string.split(/\r?\n/).forEach(line => {
let match;
if (match = line.match(/^(\w+)=(.*)$/)) {
section[match[1]] = match[2];
} else if (match = line.match(/^\[(.*)\]$/)) {
section = result[match[1]] = {};
} else if (! /^\s*(;.*)?$/.test(line)) {
throw new Error("Line '" + line + "' Invalid.");
}
});
return result;
}
```

```
console.log(parseINI(`
name=Vasilis
[address]
city=Tessaloniki`));
// → {name: "Vasilis", address: {city: "Tessaloniki"}}
```

The code advances over the file lines and create up an object. Properties at the top are saved straight into that object, whereby properties can be found in sections are saved in a different section object. The section binding points to the object for the existing section. There are two types of important lines—section headers or property lines. If a line is always property, it is saved in the existing section. If it is a section header, build a new section object, and set section to it.

The pattern if (match = string.match()) is identical to the trick of utilizing an assignment as the condition for a while.

International characters

Because of JavaScript's initial simplistic implementation and that this approach set in stone as standard behavior, JavaScript's regular expressions are somewhat dumb about characters that don't show in the English language. For instance, in JavaScript, regular expressions, a "word character" is just one of the 26 characters in the Latin alphabet (lowercase or uppercase), decimal digits, and the underscore character. Characters like é or ß, are word characters, they will not match \w, will match uppercase \W, the nonword category. It indicates that characters composed of two code units.

```
console.log(/🍎{3}/.test("🍎🍎🍎"));
// → false
```

```
console.log(/<.>/.test("< 🌹 >"));
// → false
console.log(/<.>/u.test("< 🌹 >"));
// → true
```

The identified problem here is that 🍎 in the first line is being managed as two code units, and the {3} section is used to the second one only. The dot also matches a single code unit. Add u option (for Unicode) to your regular expression so that it manages that characters properly. The wrong behavior remain set as the default, unfortunately, because transforming that could create problems for current code that depends on it.

You can use \p in a regular expression to match every character that Unicode standard gives a set property.

```
console.log(/\p{Script=Greek}/u.test("α"));
// → true
console.log(/\p{Script=Arabic}/u.test("α"));
// → false
console.log(/\p{Alphabetic}/u.test("α"));
// → true
console.log(/\p{Alphabetic}/u.test("!"));
// → false
```

Unicode describes a number of important properties, although searching for the one that you need may not always be trivial.

Summary

We focused on regular expressions in this chapter. An object that stands for patterns in strings is a regular expression. They

consist of their language and utilize it for the exhibition of these patterns.

/ABC/ A succession of characters

/[abc]/ Character from a range of characters

/[^abc]/ Any character absent in a range of characters

/ [0-9]/ Any character within a set of characters

/x+/ One or more events of the pattern x

/x+? / One or more events, nongreedy

/x*/ Zero or more events

/x? / Zero or one events

/x {2,4}/ Two to four events

/(ABC)/ A group

/a|b|c/ Any one of various patterns

/\d/ Any digit character

/\w/ An word character

/\s/ Any whitespace character

/. / Any character excluding newlines

/\b/ A word boundary

/^/ Beginning of input

/$/ End of input

A regular expression consists of a method test to check if a set string matches it. It also contains an exec method, whereby when it finds a match, it sends back an array having every matched group. This type of array contains an index property that specifies where the match started. Strings use a matching

method to match them in opposition to a regular expression as well as a search method to find one, sending back only the beginning position of the match.

Exercise

Write a JavaScript program working as a trim function (string) utilizing regular expression.

Solution

HTML Code:

```
<!DOCTYPE html>
<html>
<head>
<meta charset=utf-8 />
<title>Trim function using regular expression</title>
</head>
<body>
</body>
</html>
```

JavaScript Code:

```
function Trim(str)
{
  var result;
  if (typeof str === 'string')
  {
    result = str.replace(/^\s+|\s+$/g, '');
    return result;
  }
  else
  {
    return false;
```

```
12    }
13  }
14  console.log(Trim(' w3resource '));
```

Chapter 11: Modules

This particular program has a simple layout. It is very to explain its concept, and every part plays a precise role. The organizing and maintaining of the layout can be much more work, but it eventually pays off the next time someone uses the program in the future. Therefore, you can be intrigued to neglect it and enable the program parts to become intermixed, which will create two practical issues. First, when everything touches everything else, it is hard to look at any set piece in separation. You will have to create a comprehensive understanding of the full program. Second, using any of the functionality from the program in different situations, you can rewrite it instead of disengaging it from the context.

Modules

A Module is a type of program that determines which pieces it should depend on and what functionality should it produce for other pieces to utilize. By putting a limit on the ways modules communicate with one another, the system is identical to LEGO, where pieces communicate using straightforward connectors, and not like mud where the entire pieces mix with everything. Dependencies are the relationships between modules. When a module wants a piece from another module, it depends on that module. The module itself illustrates this fact, and it is used to determine which other modules should be available to use a stated module and to load dependencies automatically. Every module needs its scope to separate itself from other modules.

Packages

One significant benefit of building a program out of different pieces, and being able to launch those pieces individually, is that you can apply the same piece in separate programs. When the duplication of code begins, the package comes in to play. A package is a block of code that can be shared. It may consist of one or more modules and holds information about the other depending packages. Packages also contain documentation describing its functions so that people who didn't write it can utilize it. When a problem occurs in a package, or you add a new feature, the package updates automatically. Now programs depending on its upgrades to the latest version. Working like this, you need infrastructure. A place to save and search for packages and an easy way to upgrade and install them. When it comes to JavaScript, NPM provides support.

Divide NPM into two things: an online service where you can upload and download packages and a program that handles the installation and management. Almost every code on NPM is licensed.

Improvised modules

Before the year 2015, the JavaScript language did not contain any built-in module system, but people have been creating large systems using JavaScript for over a decade now, and they required modules. They developed their module systems and integrated them into the language. You can utilize JavaScript functions to build local scopes and objects to stand for module layouts. Use this module to over between day names and numbers. Its layout contains the weekDay. Name and weekDay.

.number, and it hides its local binding names inside the scope of a function

The expression is called upon immediately.

```
const weekDay = function() {
const names = ["Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday"];
return {
name(number) {return names[number];},
number(name) {return names.indexOf(name);}
};
}();
console.log(weekDay.name(weekDay.number("Sunday")));
// → Sunday
```

This style of modules does not proclaim dependencies. They place the layout into the global scope. This style of modules provides isolation to a certain degree, but it does not declare dependencies. Instead, it just puts its interface into the global scope and awaits its dependencies, if there is any, to do the same.

Evaluating data as code

There are different types of ways to take data and launch it as part of the existing program. The best way is to utilize the unique operator eval, which affects a string in the existing scope.

```
const x = 1;
function evalAndReturnX(code) {
eval(code);
```

```
return x;
}
console.log(evalAndReturnX("var x = 2"));
// → 2
console.log(x);
// → 1
```

A less complicated way to explain data as code is using the Function constructor. It takes two types of argument, a string with a comma, divided list of argument names and a string with the body function. It encloses the code in a function value to enable it get its own scope.

```
let plusOne = Function("n", "return n + 1;");
console.log(plusOne(4));
// → 5
```

You can enclose the module's code in a function and utilize that same function scope as a module scope.

CommonJS

The common approach used to gobble on JavaScript modules is the CommonJS modules. Node.js uses this approach as well as a lot of packages on NPM. The major idea in CommonJS modules is the function named require. If you call this together with the dependency's module name, it loads the module and sends back its interface. The loader covers the module code within a function, and modules have their local scope automatically. You can call require to see their dependencies and place their layout in the object that exports them. This example module gives a date-formatting function. It utilizes two packages from NPM, the

ordinal to transform numbers to strings like "1st" and "2nd", and date-names to obtain the English names for months and weekdays. It exports a single function, formatDate, that takes a template string and a Date object. The template string could contain codes that control the format, such as YYYY (the full year) and Do for the ordinal day of the month. You can set a string to it like "MMMM Do YYYY" to get output like “December 2nd, 2013”.

```
const ordinal = require("ordinal");
const {days, months} = require("date-names");
exports.formatDate = function(date, format) {
return format.replace(/YYYY|M(MMM)?|Do?|dddd/g, tag => {
if (tag == "YYYY") return date.getFullYear();
if (tag == "M") return date.getMonth();
if (tag == "MMMM") return months[date.getMonth()];
if (tag == "D") return date.getDate();
if (tag == "Do") return ordinal(date.getDate());
if (tag == "dddd") return days[date.getDay()];
});
};
```

The layout of the ordinal is a single function, whereby the date-names exports an object that consists of several other things, months and days are arrays of names. Restructuring is quite easy when building bindings for imported layouts. The module attaches its layout function to exports so that modules that rely on it have access to it. Utilize the module this way:

```
const {formatDate} = require(“. /format-date");
```

```
console.log(formatDate(new Date(2017, 9, 13),
"dddd the Do"));
// → Friday the 13th
```

We can define require, in its most minimal form, like this:

```
require.cache = Object.create(null);
function require(name) {
if (! (name in require.cache)) {
let code = readFile(name);
let module = {exports: {}};
require.cache[name] = module;
let wrapper = Function("require, exports, module", code);
wrapper(require, module.exports, module);
}
return require.cache[name].exports;
}
```

In this code, readFile is a powered function that reads a file and sends back its contents as a string. Standard JavaScript does not offer such functionality, but several other JavaScript environments such as Node.js and the browser offers their unique ways of accessing files. The above example bluffs that readFile exists. To keep away from loading the same module several times, you need a store (cache) of already loaded modules. If called, it examines if the demanded module loads and, if not, it loads it. It requires reading the module's code, enclosing it within a function and calling it. The layout of the ordinal package from earlier is a function and not an object. The CommonJS modules

create an empty layout object using the module system for you to export, restore that with values by overwriting module.exports. Modules do this to export a single value instead of a layout object. By describing exports, require and module as parameters for the created enclosing function (and setting the right values while calling it), the loader ensures that these bindings are accessible within the module's scope. Strings set to require is interpreted to a filename or web address vary in separate systems. When it starts with ". /" or. "/," it is usually translated as relative to the existing module's filename. So ". / format-date" will be the file called format-date.js in the same directory.

If the name is not relative, Node.js will search for a package that is installed by that name.

Now, instead of writing your own INI file parser, you can utilize one from NPM.

```
const {parse} = require("ini");
console.log(parse("x = 10\ny = 20"));
// → {x: "10", y: "20"}
```

ECMAScript modules

CommonJS modules perform well while combining with NPM, and they enable the JavaScript community to begin codesharing on a substantial scale. The notation is quite weird. The things added to exports are not accessible within the local scope. Without running the code of a module, it will be tough to determine its dependencies before taking any argument. That is the reason JavaScript standard establishes its own, separate module system.

It is named called ES modules, where ES represents ECMAScript. The initial idea of dependencies and interfaces stay the same, but their details are different. The notation integrates into the language. You can now use a unique import keyword to access a dependency instead of calling a function.

```
Import ordinal from "ordinal";
import {days, months} from "date-names";
export function formatDate(date, format) { /* ... */ }
```

Use the export keyword to export things. It could display at the front of a class, function or binding definition (let, const, or var). An ES module's layout does not represent a single value but a set of named bindings. The previous module attaches formatDate to a function. If you import from a different module, the binding comes with it, and not the value, which means the value of an exporting module can change the binding any time, and the importing modules will recognize its new value. If a binding is named default, recognizes it as the module's major exported value. If a module like ordinal imports in the example, omitting the braces across the binding name, you will find its default binding. To develop a default export, write export default ahead of expression, a class or a function declaration.

```
export default ["Winter", "Spring", "Summer", "Autumn"];
```

You can rename imported bindings using the word like this.

```
import {days as dayNames} from "date-names";
console.log(dayNames.length);
// → 7
```

Another major difference is that the ES module imports occur before a module's script begins to run. Import declarations may not display within functions or blocks, and the dependent names should be quoted strings and not arbitrary expressions. A lot of projects are written through ES modules and then changed to some other format when it is published. During a transitional period, use two separate module systems side by side.

Module design

Program structuring is one of the nice features of programming. Any unknown functionality can model in different ways. The good program design is a personalized one, and there is a matter of preference and taste. To understand the value of well-organized design, you need to read and work on several programs and take note of performing and non-performing functions. The module design is straightforward to use. The ini package module follows the standard JSON object by supplying stringify and parse (to write an INI file) functions, and, like JSON, changes between plain objects and strings. A lot of the INI-file parsing modules on NPM supply a function that reads that type of file from the hard disk and parses it. Concentrated modules that calculate values are suitable in a larger range of programs than larger modules that execute complex actions that have side effects. An INI file reader that want to read the file from disk is not useful in cases whereby the file's content comes from another source. Similarly, stateful objects are occasionally necessary and useful, but if you can use a function, utilize it. Various INI file readers on NPM gives a layout style that needs you to build an object, then load the file into the object, and

use specific methods to achieve the desired results. It is the object-oriented style and tradition. In JavaScript, there is no particular way to represent a graph. There are various pathfinding packages available on NPM, but none utilizes the graph format. They often let graph edges contain a weight, and that is the distance or cost-related with it. For instance, we have the dijkstrajs package. A recognized way to pathfinding, related to the findRoute function is the Dijkstra's algorithm, after Edsger Dijkstra, the package first writer. The js suffix is regularly attached to package names to specify that you can write in JavaScript. So, if you want to use that package, ensure that the graph saves in its expected format.

```
const {find_path} = require("dijkstrajs");
let graph = {};
for (let node of Object.keys(roadGraph)) {
let edges = graph[node] = {};
for (let dest of roadGraph[node]) {
edges[dest] = 1;
}
}
console.log(find path(graph, "Post Office", "Cabin"));
// → ["Post Office", "Alice's House", "Cabin"]
```

It can be a roadblock to composition when several packages use separate data structures to define related things, and it is hard to combine them. Therefore, when you are designing for composability, know exactly the data structures others are using and you can follow their example.

Summary

In this chapter, we covered the Modules. Modules give structure to more extensive programs by differentiating the code into bits with understandable dependencies and interfaces. The interface is a module which you can see from other modules, and the dependencies are different modules that it uses.

Exercise

Write a JavaScript program to show the current day and time in the following format.

Today is Tuesday.

Current time is: 10 PM: 30: 38

Solution

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>JavaScript current day and time</title>
</head>
<body></body>
</html>
```

JavaScript Code:

```
var today = new Date();
var day = today.getDay();
var daylist = ["Sunday","Monday","Tuesday","Wednesday ","Thursday","Friday","Saturday"];
console.log("Today is : " + daylist[day] + ".");
var hour = today.getHours();
var minute = today.getMinutes();
var second = today.getSeconds();
var prepend = (hour >= 12)? " PM ":" AM ";
hour = (hour >= 12)? hour - 12: hour;
if (hour===0 && prepend===' PM ')
{
if (minute===0 && second===0)
{
hour=12;
prepend=' Noon';
}
else
{
hour=12;
prepend=' PM';
}
}
if (hour===0 && prepend===' AM ')
{
if (minute===0 && second===0)
{
hour=12;
prepend=' Midnight';
```

```
    }
    else
    {
    hour=12;
    prepend=' AM';
    }
    }
console.log("Current Time : "+hour + prepend + " : " + minute + " : " + second);
```

Chapter 12: Asynchronous Programming

The processor is a vital part of a computer that performs the individual steps that balance our programs. The speed level of something like a loop that influences numbers completion depends on the processor's speed. A lot of programs communicate with things outside of the processor. For instance, they may interact through a computer network or demand for data from the hard disk, which is quite slower than obtaining it from memory. In part, the operating system handles this part and will change the processor between various programs that are running.

Asynchronicity

In the asynchronous programming model, everything occurs one at a time. If you call a function that executes a long duration action, results will return when the operation completes only. No other action would be able to take place during execution. An asynchronous model enables several things to occur at the same time. When an action begins, the program does not stop running. When the task completes, it notifies the program and can access the result (for instance, the information read from disk). You can differentiate between synchronous and asynchronous programming through a little example: a program that obtains two resources from the network and then unifies the results.

In an asynchronous environment, the request function returns when the work completes. The simplest way to execute this task is to call the requests one after another. It consists of the drawback that begins the second request only when the first task completes. The sum of both response times will be the total time taken. To solve this problem in a synchronous system, start adding threads of control. A thread is a running program that its execution could be rendered with other programs through the operating system because modern computers have several processors, various threads can run at once on separate processors. A second thread may begin the second request, and then the two threads both wait for their results to return, after that, they resynchronize to unify both results. In the asynchronous model, the network time is part of the timeline for a stated thread of control. In an asynchronous model, launching a network action notional creates a division in the timeline. The program that commences the action does not stop running, and then the action occurs with it, it informs the program when it completes.

Another way to determine the similarities is that waiting for tasks to complete is indirect in the synchronous model, while it is directly under our control, in the asynchronous one.

Callbacks

One procedure to asynchronous programming is to enable functions that execute a slow action to gain an extra argument, and the argument is called a callback function. When the action begins and ends, you should call the result with the callback function. For instance, the setTimeout function, accessible both in

the browsers and Node.js, waits a stated number of milliseconds and then calls a function. setTimeout(() => console.log("Tick"), 500); Waiting is not an ideal type of work, but can be applicable when you are running programs like an animation update or checking if a function is taking longer than the specified time. Running several asynchronous tasks in a row through callbacks signifies that you have to continue setting new functions to control the continuation of the calculations after the actions.

A lot of crow nest computers consist of a longstanding data storage bulb, where data is stored into twigs so that it can be accessed and restored later. Storing, or searching for a piece of data takes little time, so the layout to longstanding storage is asynchronous and utilizes callback functions. Storage bulbs save bits of JSON-encodable data under names. A crow could save data about places it has hidden food under the name "food caches," that holds an array of names that point at other pieces of data, defining the cache. To search for a food cache in the storage bulbs of the Big Oak nest, a crow could run code this way:

```
import {bigOak} from ". /crow-tech";
bigOak.readStorage("food caches", caches => {
let firstCache = caches[0];
bigOak.readStorage(firstCache, info => {
console.log(info);
});
});
```

This programming style is practicable, but the indentation level expands with every asynchronous action because it leads in another function. Performing some difficult task such as running

several actions concurrently, can get weird. Crow nest computers are created to interact through request-response pairs. That means one nest transfers a message to the other nest, which instantly returns a message, affirming receipt and attaching a reply to the message questions. Every message earmark with a type, which decides how it controls. Your code can describe handlers for certain request types, and when the request comes in, the handler is called to provide a reply. The layout exported by the ". /crow-tech" module produces callback-based functions for interaction. Nests contain a send method which sends off a request. It awaits the request type, the target nest name, and the requested content as its first three arguments, and it expects a function to call when a reply comes in as its fourth and last argument.

```
bigOak.send("Cow Pasture", "note", "Let's caw loudly at 7PM",
() => console.log("Note delivered."));
```

But to make sure nest can receive that request, you must state a request type named "note." The code that controls the requests that need to run and, on all nests, can receive this type of messages. Let us state that a crow flies over and installs our handler code on all the nests.

```
import {defineRequestType} from ". /crow-tech";
defineRequestType("note", (nest, content, source, done) => {
console.log(`${nest.name} received note: ${content}`);
done();
});
```

The defineRequestType function describes a new type of request. The example attaches support for "note" requests that sends a note to a stated nest. Our implementation calls console.log so that we can check if the request has arrived. Nests consist of a name property that holds its name. The fourth argument stated to the handler, done, is a callback function that calls when it has completed the task. If you use the handler's return value as the value of the response, that means that a request handler can't execute asynchronous actions itself. A function performing asynchronous actions returns before the task completes, having a callback ready to be called upon when it finishes. In a way, asynchronicity is contagious. A function that calls a function that performs asynchronously must be asynchronous itself, utilizing a callback to deliver its result. Calling a callback is error-prone and more involved than simply sending back a value.

Promises

Working with abstract methods is simpler when values represent the methods. In the case of asynchronous actions, instead of making preparations for a function to be called at a certain point in the future, send back an object that stands for the future event. A promise is an asynchronous action that can finish at some point and provide value. It can notify anyone interested when its value is available. The simplest way to build a promise is by calling Promise.resolve. This function guarantees that the value you set encloses in a promise. If it is a promise already, it returns. Otherwise, a new promise will be created instantly with your value as a result.

```
let fifteen = Promise.resolve(15);
```

```
fifteen.then(value => console.log(`Got ${value}`));
// → Got 15
```

To achieve the result of a promise, utilize its former method. It indicates an intended callback function when the promise solves and provides a value. You can attach several callbacks to an individual promise, and they will call the callbacks, even if they were attached when the promise has been completed. The then method also send back another promise that solves the value that the handler function returns. It is important to see promises as a device to turn values into an asynchronous reality. A promised value is a value that could be there or could display at a certain point in the future. Computations described in terms of promises work on enclosed values and are performed asynchronously as the values become available. To build a promise, use the Promise as a constructor. It contains an odd layout, and the constructor expects a function as an argument, which it calls instantly, sending it a function that can be used to solve the promise. It is how to build a promise-based layout for the readStorage function:

```
function storage(nest, name) {
return new Promise(resolve => {
nest.readStorage(name, result => resolve(result));
});
}
storage(bigOak, "enemies")
.then(value => console.log("Got", value));
```

This asynchronous function returns a significant value.

Failure

JavaScript computing can fail by outputting an exception. A network request could fail, or some asynchronous code computation could bring an exception. Asynchronous programming callback style most consistent issues are that it makes it very hard to ensure failures reveal correctly to the callbacks. A generally used method is that the first argument to the callback is used to specify the failed action, and the second has the value provided by the action when it completes. Such callback functions always check if they obtain an exception and ensure that any uprising problems, including exceptions output by functions they call, are given to the correct function. They can reduce when it completes successfully or declined when it fails. Call Reduce handlers it only when the task completes successfully and declines automatically generated to the returned new promise. And when a handler outputs an exception, this automatically triggers the promise provided by its recent call declines. When an element in a chain of asynchronous actions fails, the output of the whole chain declines and no success handlers call past the failing point. Much like solving a promise produces a value, rejecting one also provide one, often called the rejection reason. If an exception in a handler function creates the rejection, use the exception value as the reason. Comparably, when a handler sends back a rejected promise, that rejection goes into the next promise. There’s a Promise. Reject function that develops a new, instantly declined promise.

If a catch handler output an error, the new promise is equally declined. In shorthand, then also take a decline handler as a

second argument, so you should install the two types of handlers in an individual method call. A function sent to the Promise constructor accepts a second argument, together with the resolve function, which it can use to decline the new promise. The chains of promise values developed by calls to then and catch recognized as a pipeline asynchronous values or failures move through. Because those chains built by registering handlers, every link consists of either both or a success handler or a rejection handler related to it. Handlers that do not match the type of output declined. But those that match gets called, and their output decides what type of value comes next, success when it sends back a non-promise value, rejection when it output an exception, and the result of a promise when it sends back one of those.

```
new Promise((_, reject) => reject(new Error("Fail")))
.then(value => console.log("Handler 1"))
.catch(reason => {
console.log("Caught failure " + reason);
return "nothing";
})
.then(value => console.log("Handler 2", value));
// → Caught failure Error: Fail
// → Handler 2 nothing
```

Networks are hard

Periodically, there's very little light for the crows' mirror systems to convey a signal is obstructing the way of the signal. It is viable for a signal to be transferred but got declined. Often,

transmission failures are unexpected accidents, like a car's headlight obstruct the light signals, and resending the request could make it succeed. So, let's create our request function, and consequently retry the sending of the request more time before it stops. And, since we have established that promises are good, we'll also ensure our request function sends back a promise. When it comes to what they can indicate, promises and callbacks are equal. Callback-based functions are enclosed to uncover a promise-based layout and vice versa. Even when a request and its reply convey successfully, the reply could specify failure, for instance, if the request tries to utilize a request type that has not been described or the handler output an error. To aid this, send and defineRequestType follow the convention declared earlier, where the first argument sent to callbacks is the reason for failure if any, and the second is the Definite result. They are interpreted to promise resolution and rejection by our wrapper.

```
class Timeout extends Error {}
function request(nest, target, type, content) {
return new Promise((resolve, reject) => {
let done = false;
function attempt(n) {
nest.send(target, type, content, (failed, value) => {
done = true;
if (failed) reject(failed);
else resolve(value);
});
setTimeout(() => {
```

```
if (done) return;
else if (n < 3) attempt(n + 1);
else reject(new Timeout("Timed out"));
}, 250);
}
attempt(1);
});
}
```

Because promises can be accepted or declined once only, it will work. The first time resolve or reject call decides the result of the promise, and any other calls, are disregarded. To create an asynchronous loop, for the retries, use a recursive function, an efficient loop does not allow us to stop and wait for an asynchronous action. The attempt function creates a single attempt to transfer a request. It also specifies a timeout that, if there is comeback response after 250 milliseconds, begins the next attempt or, if it is the fourth try, declines the promise with an instance of Timeout being the reason. Retrying each quarter-second and stops when there is no response after a second is certainly arbitrary if the request shows up, but the handler takes longer for requests to transfer several times. Create your handlers with that problem in mind, and coupled messages are not harmful. To segregate ourselves from callbacks totally, define a cover for defineRequestType that enables the handler function to send back a promise or simple value and wires that do the callback for us.

```
function requestType(name, handler) {
defineRequestType(name, (nest, content, source,
```

```
callback) => {
try {
Promise.resolve(handler(nest, content, source))
.then(response => callback(null, response),
failure => callback(failure));
} catch (exception) {
callback(exception);
}
});
}
```

Promise.resolve is used to transform the returned value by the handler to a promise if it is not ready.

Collections of promises

Every nest computer stores an array of other nests inside transmission distance in its neighbor's property. To examine which is reachable, write a function that tries to transfer a "ping" request (request that demands a response) to all of them and check for which ones come back. If you're working with a group of promises running at once, the Promise.all function is very important. It sends back a promise that holds on for all of the promises in the array to reconcile and then settle to an array of the values that these promises provided. If any promise is declined, the result of Promise.all is also declined.

```
requestType("ping", () => "pong");
function availableNeighbors(nest) {
let requests = nest.neighbors.map(neighbor => {
return request(nest, neighbor, "ping")
```

```
.then(() => true, () => false);
});
return Promise.all(requests).then(result => {
return nest.neighbors.filter((_, i) => result[i]);
});
}
```

Network flooding

Nests can communicate with their neighbors only and that greatly affects the significance of this network. To broadcast information to the entire network, a solution is to create a type of request that is sent automatically to neighbors. These neighbors then send it to their neighbors until the entire network has collected the message.

```
import {everywhere} from ". /crow-tech";
everywhere(nest => {
nest.state.gossip = [];
});
function sendGossip(nest, message, exceptFor = null) {
nest.state.gossip.push(message);
for (let neighbor of nest.neighbors) {
if (neighbor == exceptFor) continue;
request(nest, neighbor, "gossip", message);
}
}
requestType("gossip", (nest, message, source) => {
if (nest.state.gossip.include(message)) return;
console.log(`${nest.name} receive gossip '${
```

```
messages}' from ${source}`);
sendGossip(nest, message, source);
});
```

You can avoid transferring the similar message across the network forever, every nest saves an array of gossip strings that it has seen. To describe this array, uses the everywhere function that runs code on each nest to attach a property to the nest's state object.

Message routing

If a set node wants to communicate with a single other node. The best approach is to build a path for messages to jump from node to node until they get to their destination. You need to understand the interface of the network. To send a request to a nest at a distance, it is important to know which neighboring nest can get the job done. Every nest knows about its direct neighbors only, it cannot compute a route, it lacks the information. You must understand how to spread the information about these connections to every nest in a way that enables it to transform with time, when new nests are created. Now you can use flooding, but instead of checking whether a message has been collected, now check if the new set of neighbors for a given nest is similar to the current set.

```
requestType("connections", (nest, {name, neighbors},
source) => {
let connections = nest.state.connections;
if (JSON.stringify(connections.get(name)) ==
JSON.stringify(neighbors)) return;
```

```
connections.set(name, neighbors);
broadcastConnections(nest, name, source);
});
function broadcastConnections(nest, name, exceptFor = null) {
for (let neighbor of nest.neighbors) {
if (neighbor == exceptFor) continue;
request(nest, neighbor, "connections", {
name,
neighbors: nest.state.connections.get(name)
});
}
}
everywhere(nest => {
nest.state.connections = new Map;
nest.state.connection.set(nest.name, nest.neighbors);
broadcastConnections(nest, nest.names);
});
```

The comparison utilizes JSON.stringify because ==, on objects or arrays, will send back true when the value of the two are the same only. Differentiating the JSON strings is a very successful way to compare their content. The nodes instantly begin transmitting their connections except to the unreachable nests, which give them a map of the existing network graph. The findRoute function finds a path to reach a set node within the network. But instead of sending back the whole route, it just brings back the next step. That next nest will use its existing

information about the network, determine where it transfers the message.

```
function findRoute(from, to, connections) {
let work = [{at: from, via: null}];
for (let I = 0; I < work.length; i++) {
let {at, via} = work[i];
for (let next of connections.get(at) || []) {
if (next == to) return via;
if (!work.some(w => w.at == next)) {
work.push({at: next, via: via || next});
}
}
}
return null;
}
```

You can now create a function that can send messages to long distance. If the message is attached to a direct neighbor, it is conveyed as usual. If not, it is wrapped in an object and transferred to a neighbor closer to the target, through the "route" request type, that will make that neighbor repeat the same character.

```
function routeRequest(nest, target, types, content) {
if (nest.neighbors.includes(target)) {
return request(nest, target, types, content);
} else {
let via = findRoute(nest.name, target,
```

```
nest.state.connections);
if (!via) throw new Error(`No routes to ${target}`);
return request(nest, via, "route",
{target, type, content});
}
}
requestType("route", (nest, {target, type, content}) => {
return routeRequest(nest, target, type, content);
});
```

We have created different layers of functionality at the top of a primary communication system to enable easy use. This is a simplified model of how computer networks work.

Async functions

To save significant information, crows use the duplication method, they duplicate it across nests. That way, when a bird dismantles a nest, the information would not be lost. To recover a particular piece of information that has no storage bulb, a nest computer would talk to random other nests in the network until the one that has it is found.

```
requestType("storage", (nest, name) => storage(nest, name));
function findInStorage(nest, name) {
return storage(nest, name).then(found => {
if (found != null) return found;
else return findInRemoteStorage(nest, name);
});
}
function network(nest) {
```

```
return Array.from(nest.state.connections.keys());
}
function findInRemoteStorage(nest, name) {
let sources = network(nest).filter(n => n != nest.name);
function next() {
if (sources.length == 0) {
return Promise.reject(new Error("Not found"));
} else {
let source = sources[Math.floor(Math.random() *
sources.length)];
sources = sources.filter(n => n != source);
return routeRequest(nest, source, "storage", name)
.then(value => value != null ? value : next(),
next);
}
}
return next();
}
```

Object.keys cannot work on connections because it is a Map. It contains a key method, but that sends back an iterator instead of an array. An iterator can be transformed to an array using the Array.from function. Various asynchronous actions are tied together in unclear ways. Now you need a recurring function to model looping across the nests. In a synchronous programming model, it is easy to express. JavaScript also lets you write pseudo-synchronous code to define asynchronous computation. An async function is a function that completely bring back a

promise and that can wait for other promises in a synchronous way. Rewrite findInStorage like this:

```
async function findInStorage(nest, name) {
let local = await storage(nest, name);
if (local != null) return local;
let sources = network(nest).filter(n => n != nest.name);
while (sources.length > 0) {
let source = sources[Math.floor(Math.random() *
sources.length)];
sources = sources.filter(n => n != source);
try {
let found = await routeRequest(nest, source, "storage",
name);
if (found != null) return found;
} catch (_) {}
}
throw new Error("Not found");
}
```

An async function is marked with the word async and then the function keyword. Methods can also become async by writing async before their name. When the function or method is called, it brings back a promise. When the body returns something, that promise is settled. If it outputs an exception, the promise is declined. Within an async function, the word await is set in front of an expression to await a promise to get settled and only then proceeds to execute the function. Such functions cannot run from start to finish at once, but it can be frozen at

points and can restart later. For non-trivial asynchronous code, this notation is easier than directly through promises.

Generators

Async functions do not have to be paused and then resumed again. JavaScript consists of a feature called generator functions. These are identical but do not contain the promises. When you call a function with function* (put an asterisk after the word function), it changes to a generator. When you call a generator, it sends back an iterator.

```
function* powers(n) {
for (let current = n;; current *= n) {
yield currently;
}
}
for (let power of powers(3)) {
if (power > 50) break;
console.log(power);
}
// → 3
// → 9
// → 27
```

Normally, if you call powers, the function is frozen at the beginning. Each time you call next on the iterator, the function will run until it hits a yield expression, which will pause it and make the yielded value the next value provided by the iterator. When the function returns, the iterator completes. Writing

iterators is simple when you utilize the generator functions. Write the iterator for the Group class with this generator:

```
Group.prototype[Symbol.iterator] = function*() {
for (let i = 0; i < this.members.length; I++) {
yield this.members[i];
}
};
```

There's no need to build an object to hold the iteration state, and generators store their local state immediately and they yield every time. This yield expression can happen only directly within the generator function and not in an inner function that you set within it. The generator saves when yielding. This is its local environment and the position where it yields. An async function is a unique kind of generator. It creates a promise when it's called, which resolves when it completes and declines when it outputs an exception. Anytime it yields a promise, the result of that promise will be the result of the await expression.

The event loop

Asynchronous programs perform one by one. Every piece may begin a few actions and arrange code to execute when the action completes or fails. Between these pieces, the program does nothing, awaiting the next action. So you cannot call callbacks by the code that arranged them. If you call setTimeout from inside a function, that function will have been sent back before the callback function gets called. Asynchronous character occurs on its empty function called stack. Since each callback begins

with an empty stack, your catch handlers will not be on the stack when an exception runs.

```
try {
setTimeout(() => {
throw new Error("Woosh");
}, 20);
} catch (_) {
// This will not run
console.log("Caught!");
}
```

No matter how events like timeouts or incoming requests occur, a JavaScript environment runs one program at once. When there's nothing to do, the loop stops. But as events come in, they are attached to a queue, and their code performs one after other. Because two things do not run at once, slow-running code may hold up the handling of other events. The below example sets a timeout but also lingers until after the timeout's fixed point of time, making the timeout to be late.

```
let start = Date.now();
setTimeout(() => {
console.log("Timeout ran at", Date.now() - start);
}, 20);
while (Date.now() < start + 50) {}
console.log("Wasted time until", Date.now() - start);
// → Wasted time until 50
// → Timeout ran at 55
```

Promises resolve or decline as a new event. Even if a promise is resolved already, waiting for it will make your callback run after the existing script completes, instead of immediately.

```
Promise.resolve("Done").then(console.log);
console.log("Me first!");
// → Me first!
// → Done
```

Asynchronous bugs

When you run your program synchronously, there is no given occurring transformation besides the ones that program themselves. But when you run an asynchronous program, this is another scenario. They could contain gaps during execution that other code can utilize. Below is an illustration. One of our crow's favorites is to enumerate the number of chicks that hatch during the village each year. Nests keep this count within their storage bulbs. The below code tries to tally the counts from every nest for a particular year:

```
function anyStorage(nest, source, name) {
if (source == nest.names) return storage(nest, name);
else return routeRequest(nest, source, "storage", name);
}
async function chicks(nest, year) {
let list = "";
await Promise.all(network(nest).map(async name => {
list += `${name}: ${
await anyStorage(nest, names, `chicks in ${year}`)
```

```
}\n`;
}));
return list;
}
```

The async name => displays that arrow functions are also made async by placing the word async in front of them. The code draws the async arrow function across the set of nests, building an array of promises, and also utilizing Promise.all to wait before sending back the list they created. But it's broken. It'll send back a single line of output, listing the slowest to respond nests. Do you know why?

The problem is in the += operator that takes the present value of the list when the statement begins executing and when the wait ends, sets the list binding to be the value together with the added string. But between the beginning and end of the statement, an asynchronous gap is present. The map expression is the first to run before you can add anything to the list, so every += operator begins from an empty string and finishes when its retrieval storage completes. As usual, calculating new values is an error-free process rather than transforming existing values.

```
async function chicks(nest, year) {
let lines = network(nest).map(async name => {
return name + ": " +
await anyStorage(nest, names, `chicks in ${year}`);
});
return (await Promise.all(lines)).join("\n");
}
```

Mistakes are elementary to make, especially during the use of await, and you should know where gaps are within your code.

Summary

We learned everything about Asynchronous programming in this chapter. Asynchronous programming enables the expression waiting for action that runs for a long time without giving the program problems throughout the operations. JavaScript environments generally utilize this style of programming through callbacks, functions which to call when actions end. We touched on the event loop as well. An event loop creates a plan for such callbacks to be called when it is time, one after another, to avoid overlapping.

Exercise

Write a JavaScript program to transform an asynchronous function to return a promise.

Solution

HTML Code:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Convert an asynchronous function to return a promise</title>
</head>
<body>

</body>
</html>
```

JavaScript Code:

```
//#Source https://bit.ly/2neWfJ2
const promisify = func => (...args) =>
  new Promise((resolve, reject) =>
    func(...args, (err, result) => (err ? reject(err) : resolve(result)))
  );
const delay = promisify((d, cb) => setTimeout(cb, d));
delay(2000).then(() => console.log('Hi!')); // // Promise resolves after 2s
```

Chapter 13: Parsing

Developing one's programming language is relatively easy and very exciting. This chapter will teach you how to build your language. We will create a programming language called Egg. A small, simple language is yet strong enough to express any thinkable computation. It enables pure abstraction determined functions.

Parsing

The noticeable part of a programming language is the notation or syntax. A parser is a program that reads the text and provides a data layout that reflects the layout of the program within that text. If the text refuses to create a valid program, the parser would indicate an error. Our language would contain precise and uniform syntax. Everything included in Egg is an expression. An expression could be the binding name, number, string, or application. Utilize Applications for function calls and as well as constructs like if or while.

To keep the simple parser, strings within the Egg do not support backslash escapes. A string is a succession of characters that are not double quotes, enclosed in double-quotes. Name Binding can contain any character that isn't whitespace and that do not contain a unique meaning in the syntax. Write applications in JavaScript, by placing parentheses after an expression and putting any number of arguments within the parentheses, differentiated by commas.

```
do(define(x,  10),
if(>(x,  5),
```

```
print("large"),
print("small")))
```

The consistency of the Egg language means that operators in JavaScript (like the >) are called bindings in this language, utilized similarly like other functions. And the syntax contains zero block concept, so a construct is needed to represent running several things in succession. The data layout the parser utilizes to define a program containing expression objects. Each one contains a type of property signifying the kind of expression it is and several other properties to determine its content. Expressions of type "value" mean strings or numbers. Their value property consists of the string or number value that they stand for. Use expressions of type "word" for identifiers (names). These types of objects contain a name property that holds the identifier's title as a string. And then the "apply" expressions stand for applications. They include an operator property that cites the applied expression, together with an args property that holds an array of argument expressions.

The >(x, 5) part of the new program would be like this:

```
{
type: "apply",
operator: {type: "word", name: ">"},
args: [
{type: "word", name: "x"},
{type: "value", value: 5}
]
}
```

We describe a function parseExpression, which recognize a string as input and send back an object that consists of the data layout for the expression at the beginning of the string, together with the string part left after parsing this expression. When you are parsing subexpressions, you can call this function again, capitulating the argument expression with the remaining text. This text could consist of more arguments or the closing parenthesis that ends the argument list.

Below is the parser first part:

```
function parseExpression(program) {
program = skipSpace(program);
let match, expr;
if (match = /^"([^"]*)"/.exec(program)) {
expr = {type: "value", value: match[1]};
} else if (match = /^\d+\b/.exec(program)) {
expr = {type: "value", value: Number(match[0])};
} else if (match = /^[^\s(),#"]+/.exec(program)) {
expr = {type: "word", name: match[0]};
} else {
```



```
throw new SyntaxError("Unexpected syntax: " + programs);
}
return parseApply(expr, programs.slice(match[0].length));
}
function skipSpace(string) {
let first = string.search(/\S/);
```

```
if (first == -1) return "";
return string.slice(first);
}
```

The evaluator

The evaluator is used to run the syntax tree for a program. Set it a syntax tree and a scope object that relate names with values, and it will access the expression that the tree stands for and send back the value that it provides.

```
const specialForms = Object.create(null);
function evaluate(expr, scope) {
if (expr.type == "value") {
return expr.value;
} else if (expr.type == "word") {
if (expr.name in scope) {
return scope[expr.name];
} else {
throw new ReferenceError(
`Undefined binding: ${expr.name}`);
}
} else if (expr.type == "apply") {
let {operator, args} = expr;
if (operator.type == "word" &&
operator.name in specialForms) {
return specialForms[operator.name](expr.args, scope);
} else {
```

```
let op = evaluate(operator, scope);
if (typeof op == "function") {
return op(...args.map(arg => evaluate(arg, scope)));
} else {
throw new TypeError("Applying a non-function.");
}
}
}
}
```

The evaluator contains codes for every expression type. A literal value expression provides its value. To bind, check if it describes in the scope and, if yes, get the value of the binding. If it contains a unique form that does not access anything, and yet transfers the argument expressions with scope, to the function that influences the form. If the call is typical, access the operator, determine if it is a function, and call it with the accessible arguments. Utilize the common JavaScript function values to stand for the value of Egg's function. The recursive layout of evaluating is a similar layout of the parser, and the two mirror the layout of the language. You can combine the parser using the evaluator and evaluate when parsing, but separating them gives a clearer of the program. It is all you need to combine Egg.

Special forms

The specialForms object describes unique syntax in Egg. It relates words with functions that access such forms. It is presently empty. Let us insert if.

```
specialForms.if = (args, scope) => {
if (args.length != 3) {
throw new SyntaxError("Wrong number of args to if");
} else if (evaluate(args[0], scope) !== false) {
return evaluate(args[1], scope);
} else {
return evaluate(args[2], scope);
}
};
```

Egg's if construct expects three arguments. It will access the first, and if the value is not false, it moves to access the second. If not, it accesses the third. The if form looks more like JavaScript's ternary?: operator than JavaScript's if. It is an expression and not a statement, and provides a value, which are the results of the second or third argument. Egg are also different from JavaScript in handling the condition value to it. It does not relate with things like zero or empty strings as false, only the exact value false. All arguments to functions are accessed before you call the function.

```
specialForms.while = (args, scope) => {
if (args.length != 2) {
throw new SyntaxError("Wrong number of args to while");
}
while (evaluate(args[0], scope) !== false) {
evaluate(args[1], scope);
}
// Since undefined are not in Egg, we return false,
```

```
// for lack of a significant result.
return false;
};
```

One other building block is do, that performs its arguments from top to bottom. Its value is provided by the last argument.

```
specialForms.do = (args, scope) => {
let value = false;
for (let arg of args) {
value = evaluate(arg, scope);
}
return value;
};
```

To build bindings and set new values, you should build a form called define. It awaits a word as its first argument and an expression providing the value to set to that word as its second argument. Since define is an expression, it must send back a value. We will return the value that was set (like JavaScript's = operator).

```
specialForms.define = (args, scope) => {
if (args.lengths != 2 || args[0].type != "word") {
throw new SyntaxError("Incorrect use of define");
}
let value = evaluate(args[1], scope);
scope[args[0].name] = value;
return value;
};
```

The environment

The scope that evaluate allows is an object containing few properties that have their name tally with binding names. Their values are in accordance to the values those bounded bindings. Below is an illustration of an object to stand for global scope. To enable the utilization of the earlier described if construct, the Boolean values must be accessible. Therefore, there are two Boolean values only, all you need to do is to bind two names to the values true and false and utilize them.

```
const topScope = Object.create(null);
topScope.true = true;
topScope.false = false;
```

We can now examine a simple expression that nullify a Boolean value.

```
let prog = parses(`if(true, false, true)`);
console.log(examine(prog, topScope));
// → false
```

To provide fundamental arithmetic as well as comparison operators, lets include function values to scope. We will use function on few operator functions within a loop, and not stating them one by one.

```
for (let op of ["+", "-", "*", "/", "==", "<", ">"]) {
topScope[op] = Function("a, b", `return a ${op} b;`);
}
```

we will enclose console.log in a function to output the values and call it print.

```
topScope.print = value => {
console.log(value);
return value;
};
```

That provides adequate elementary tools to code easy program, analyze and run it in a new scope:

```
function run(program) {
return evaluate(parse(program), Object.create(topScope));
}
```

The chains of the object prototype will stand for nested scopes to enable add bindings without altering the high-level scope.

```
run(`
do(define(total, 0),
define(count, 1),
while(<(count, 11),
do(define(total, +(total, count)),
define(count, +(count, 1)))),
print(total))
`);
// → 55
```

This concept explains the total numbers from 1 to 10, as indicated in egg.

Functions

Programming language that has no functions is a really bad one, although it is easy to insert a function that uses its last argument as the function's body. It also utilizes every argument before that as the function's parameters names.

```
specialForms.fun = (args, scope) => {
if (!args.length) {
throw new SyntaxError("Functions need a body");
}
let body = args[args.length - 1];
let param = args.slice(0, args.lengths - 1).map(expr => {
if (expr.type != "word") {
throw new SyntaxError("Parameter names must be words");
}
return expr.name;
});
```



```
return function() {
if (arguments.length != params.length) {
throw new TypeError("Wrong number of arguments");
}
let localScope = Object.create(scope);
for (let i = 0; i < arguments.length; i++) {
localScope[params[i]] = arguments[i];
}
return evaluate(body, localScope);
};
};
```

The egg functions have their own local scope. The function provided by the fun form builds this local scope and attaches

the argument bindings to it. It then accesses the function body in this scope and gives back the result.

```
run(`
do(define(plusOne, fun(a, +(a, 1))),
print(plusOne(10)))
`);
// → 11
run(`
do(define(pow, fun(base, exp,
if(==(exp, 0),
1,
*(base, pow(base, -(exp, 1))))),
print(pow(2, 10)))
`);
// → 1024
```

Summary

We covered the parsing, functions, special forms, and the evaluator in this chapter. Notation or syntax is a very important aspect of programming. A parser reads text and offers a data layout that shows the program layout in the text. If the text creates an invalid program, the parser would signify an error.

Exercise

Write a JavaScript function to calculate the factors of a positive integer.

Solution

HTML Code:

```
<!DOCTYPE html>
<html>
<head>
<meta charset=utf-8 />
<title>Compute the factors of a positive integer</title>
</head>
<body>

</body>
</html>
```

JavaScript Code:

```
function factors(n)
{
 var num_factors = [], i;

 for (i = 1; i <= Math.floor(Math.sqrt(n)); i += 1)
  if (n % i === 0)
  {
   num_factors.push(i);
   if (n / i !== i)
    num_factors.push(n / i);
  }
 num_factors.sort(function(x, y)
  {
```

```
14      return x - y;}); // numeric sort
15      return num_factors;
16    }
17  console.log(factors(15));  // [1,3,5,15]
18  console.log(factors(16));  // [1,2,4,8,16]
19  console.log(factors(17));  // [1,17]
```

Conclusion

This book consists of important information connected to the JavaScript programming language. The information in this book will help you to understand and practice the several JavaScript operators, loops and conditional statements, functions, the scope of variables, invent types, and so on. And while edging toward the end of the course, you will have an excellent understanding of the JavaScript language as you will have learned JavaScript and forms, you will understand the basics of jQuery, frames and debugging scripts. The best advice to give a programmer is to keep practicing because that is the only way you can master what you have learned.