**Chapter 3 JavaScript Basics**

*Last updated: 2021-08-24 14:54:58*

**3.1 Introduction**

So far we learned how to define the contents and style of web pages, with HTML (Chapter [1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\html.html#html)) and CSS (Chapter [2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\css.html#css)), respectively. We now move to the third major web technology, **JavaScript**. Being a programming language, JavaScript is fundamentally different from HTML and CSS, which are essentially data formats. Using HTML and CSS we can instruct the browser to display the given contents, with particular styling. Using a programming language, such as JavaScript, we can give the browser (or other computing environments) a much wider variety of instructions. For example, we can instruct the computer to perform mathematical calculations, to show or hide contents in response to user input, to periodically load up-to-date contents from an external source, and so on.

JavaScript is the most important of the technologies presented in this book, and we will use it extensively throughout the later chapters to build interactive web maps. Chapter [3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#javascript-basics) (this chapter) introduces the basic principles of the JavaScript language and the practical way of experimenting with it, using the JavaScript console built into web browsers (Figure [3.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fig:chrome-js-console)). All of the examples we will see in this chapter work “in isolation,” not affecting web page content and style. Therefore, while reading this chapter, the reader can temporarily put aside what we learned in Chapters [1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\html.html#html)–[2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\css.html#css). Then, in Chapter [4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-interactivity.html#javascript-interactivity), we will go back to HTML and CSS and see how to associate our JavaScript code with the web page contents, and how to modify that contents in response to user input. Finally, in Chapters [6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#leaflet)–[13](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\collaborative-mapping.html#collaborative-mapping), we will apply these techniques in the context of web mapping, using JavaScript to create customized web maps and to control their behavior.

**3.2 What is JavaScript?**

[**JavaScript**](https://en.wikipedia.org/wiki/JavaScript) is a programming language, mostly used to define the interactive behavior of web pages. JavaScript allows you to make web pages more interactive by accessing and modifying the contents and styling in a web page while it is being viewed in the browser. JavaScript, along with HTML and CSS, forms the foundation of modern web browsers and the internet.

JavaScript is considered, unofficially, to be the most popular programming language in the world (Figure [0.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\index.html#fig:stackoverflow-survey)). Numerous open-source mapping and data visualization libraries are written in JavaScript, including **Leaflet**, which we use in this book, **OpenLayers**, **D3**, and many others. Many commercial services of web mapping also provide a JavaScript API for building web maps with their tools, such as **Google Maps JavaScript API**, **Mapbox GL JS** and **ArcGIS API for JavaScript** (more on that in Section [6.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#alternatives-to-leaflet)).

**3.3 Client-side vs. server-side**

When talking about programming for the web, one of the major concepts is the distinction between **server-side** and **client-side**.

* The **server** is the location on the web that serves your website to the rest of the world.
* The **client** is the computer that is accessing that website, requesting information from the server.

In server-side programming, we are writing scripts that run on the *server*. In client-side programming, we are writing scripts that run on the *client*, that is—in the browser. This book focuses on *client-side* programming, though we will say a few words on server-side programming in Chapter [5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\web-servers-1.html#web-servers-1). JavaScript can be used in both server-side[19](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn19) and client-side programming, but is primarily a client-side language, working in the browser on the client computer.

As we will see later on, there are two fundamental ways for using JavaScript on the client-side:

* Executing scripts when the web page is being **loaded**, such as scripts that instruct the browser to download data from an external location and display them on the page
* Executing scripts in response to user **input** or interaction with the page, such as performing a calculation and showing the result when the user clicks on a button

When JavaScript code is executed in a web page, it can do many different types of things, such as modifying the contents of the page, changing the appearance of content, sending information to a server, or getting information from the server.

**3.4 The JavaScript console**

When loading a web page which is associated with one or more scripts, the JavaScript code is automatically *interpreted* and executed by the browser. We can also manually interact with the browser’s interpreter or [engine](https://en.wikipedia.org/wiki/JavaScript_engine) using the JavaScript **console**, also known as the command line, which is built into all modern web browsers. The console is basically a way to type code into a computer and get the computer’s answer back. This is useful for experimenting with JavaScript code, such as quickly testing short JavaScript code snippets. It is exactly what we are going to do in this chapter.

* Open a web browser.
* In Chrome: open the JavaScript console by pressing **Ctrl+Shift+J**, or by opening the developer tools with **F12**, then clicking on the **Console** tab.
* Type 5+5 and press **Enter**.
* You have just typed a JavaScript expression, which was interpreted and the returned value 10 was printed.
* The expression sent to the interpreter is marked with the > symbol. The returned value being printed is marked with the <- symbol (Figure [3.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fig:chrome-js-console)).
* Use **Ctrl+L** to clear the console.

Figure [3.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fig:chrome-js-console) shows a screenshot of the JavaScript console with the entered expression 5+5 and the response 10.

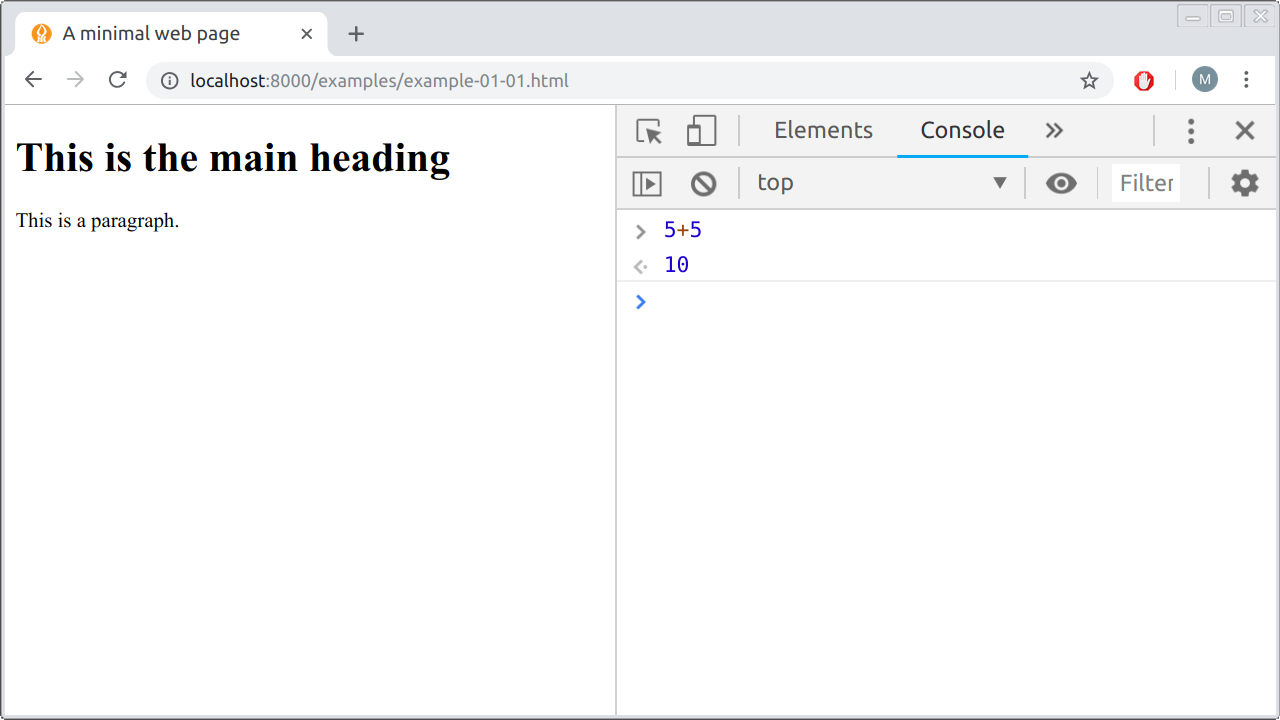


FIGURE 3.1: The JavaScript console in Chrome

In this chapter, we will be using the console to get familiar with the JavaScript language. We will be typing expressions and examining the computer’s responses. In the following Chapters [4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-interactivity.html#javascript-interactivity)–[13](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\collaborative-mapping.html#collaborative-mapping), instead of typing our JavaScript code, we will mostly load the code using the <script> element (Section [1.6.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\html.html#styling-and-scripts)), either from embedded scripts or from external files. Nevertheless, the console is still going to be useful to interactively explore function behavior or currently loaded objects, or to interactively [**debug**](https://en.wikipedia.org/wiki/Debugging) our scripts when they contain errors.

**3.5 Assignment**

**Assignment** to variables is one of the most important concepts in programming, since it lets us keep previously calculated results in memory for further processing. **Variables** are containers that hold data values, simple or complex, that can be referred to later in your code.

There are several ways to define variables in JavaScript, using the following keywords:

* var
* let
* const

The traditional way has been using the keyword var. The newer method—using the keyword let—is more similar to the “usual” behavior in other programming languages. For example, variables defined with let have block-scope, are inaccessible before the declaration, and cannot be re-defined. For these reasons, in this book we are going to use let. You should be aware of var as well, because it is still commonly used and you are going to see it often in other scripts, programming forums, etc.

The third keyword, const, is similar to let but used to define *constant* variables, which cannot be modified later on in our code. We are going to use let throughout the book to keep things simple. Keep in mind that using const instead let is recommended for variables that are kept constant throughout the program, to increase code clarity and minimize potential mistakes.

To **define** a variable, the let keyword is followed by the variable name. For example, the following two expressions define the variables x and y:

**let** x;

**let** y;

Note that each JavaScript expression should end with ;. Ending statements with semicolon is not strictly required, since statement ending can also be automatically determined by the browser, but it is highly recommended to use ; in order to reduce ambiguity.

Values can be **assigned** to variables using the assignment operator =. Assignment can be made along with variable definition:

**let** x = 5;

**let** y = 6;

or later on, in separate expressions, after the variables have already been defined:

x = 5;

y = 6;

Either way, we have now defined two variables x and y. The values contained in these two variables are 5 and 6, respectively. In your JavaScript console, to see a current value of a variable, type its name and hit **Enter**. The current value will be returned and printed:

x; *// Returns 5*

The above code uses a JavaScript **comment**. We already learned how to write HTML comments (Section [1.5.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\html.html#html-comments)) and CSS comments (Section [2.8.2.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\css.html#color-background-color)) in previous chapters. In JavaScript:

* Single line comments start with //
* Single or multi-line comments start with /\* and end with \*/ (like in CSS)

Note that you cannot **re-define** an existing variable defined with let, but you can assign a **new value** into an existing variable. For example, the following pair of expressions raises an *error*[20](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn20):

**let** x = 5;

**let** x = 6; *// Uncaught SyntaxError: Identifier 'x' has already been declared*

while the following is perfectly fine:

**let** x = 5;

x = 6;

There are several short-hand assignment operators, in addition to the basic assignment operator =. For example, another commonly used assignment operator is the **addition assignment** operator +=, which adds a value to a variable. The following expression uses the addition assignment operator +=:

x += 10;

This is basically a shortcut for the following assignment[21](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn21):

x = x + 10;

**3.6 Data types**

**3.6.1 Overview.**In the examples in Section [3.5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#assignment) we defined variables holding *numeric* data, such as the numbers 5 and 6. JavaScript variables can also hold other **data types**. JavaScript data types are divided into two groups: **primitive** data types and **objects**.

Primitive data types include the following:

* **String**—Text characters that are delimited by quotes, for example: "Hello" or 'Hello'
* **Number**—Integer or floating-point value, for example: 5 and -10.2
* **Boolean**—Logical values, true or false
* **Undefined**—Specifies that a variable is declared but has no defined value, undefined
* **Null**—Specifies an intentional absence of any object value, null

Objects are basically collections of the primitive data types and/or other objects:

* **Array**—Multiple variables in a single variable, for example: ["Saab", "Volvo", "BMW"]
* **Object**—Collections of name:value pairs, for example: {type:"Fiat", model:"500", color:"white"}

The data types in JavaScript are summarized in Table [3.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#tab:js-data-types). In the following Sections [3.6.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#primitive-data-types)–[3.6.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#checking-type-of-variables) we go over the data types in JavaScript one by one.

| TABLE 3.1: JavaScript data types | | |
| --- | --- | --- |
| **Group** | **Data type** | **Example** |
| Primitive | String | "Hello" |
|  | Number | 5 |
|  | Boolean | true |
|  | Undefined | undefined |
|  | Null | null |
| Objects | Array | ["Saab", "Volvo"] |
|  | Object | {type: "Fiat", model: "500"} |

**3.6.2 Primitive data types**

**3.6.2.1 String.Strings** are collections of text characters, inside single (') or double (") quotes[22](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn22). For example, the following expressions define two variables that contain strings, s1 and s2:

**let** s1 = 'Hello';

**let** s2 = "World";

Strings can be **concatenated** using the + operator:

s1 + s2; *// Returns "HelloWorld"*

s1 + " " + s2; *// Returns "Hello World"*

The += operator works with strings too. It is useful to sequentially construct strings with HTML code. For example, in the following code section we construct HTML code for a <ul> (unordered list) element (Section [1.6.8.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\html.html#html-ul-ol-li)) with two internal <li> elements (list items) using the += operator, assigned into a string named list:

**let** list = "";

list += "<ul>";

list += "<li>Apples</li>";

list += "<li>Oranges</li>";

list += "</ul>";

We are going to revisit this technique later on, when constructing HTML code using loops (Section [3.10.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#loops)).

**3.6.2.2 Number.Numbers** can be integers or decimals. The usual **arithmetic operators**, +, -, \*, and /, can be used with numeric variables:

**let** x = 5, y = 6, z;

z = x + y; *// Returns 11*

In this example, we defined two variables x and y and assigned their values (5 and 6). We also defined a third variable z without assigning a value. Note how we defined x, y, and z in the same expression, separated by commas—this saves typing the let keyword three times. In the last expression we calculated x+y and assigned the result of that calculation into z.

JavaScript also has **increment** ++ and **decrement** -- operators. The increment operator ++ adds one to the current number. The decrement operator -- subtracts one from the current number. For example:

**let** x = 5;

x++; *// Same as: x=x+1; the value of x is now 6*

x--; *// Same as: x=x-1; the value of x is now 5*

When running the last two expressions, you will see 5 and 6 (not 6 and 5) printed in the console, respectively. This is because the increment ++ and decrement -- expressions return the current value, before modifying it[23](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn23).

**3.6.2.3 Boolean.Boolean** (logical) values are either true or false. For example:

**let** found = **true**;

**let** lost = **false**;

In practice, boolean values are usually created as the result of **comparison operators**. For example:

9 >= 10; *// Returns false*

11 > 10; *// Returns true*

JavaScript comparison operators are summarized in Table [3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#tab:comparison-operators).

| TABLE 3.2: JavaScript comparison operators | |
| --- | --- |
| **Operator** | **Meaning** |
| == | Equal to |
| === | Equal value and equal type |
| != | Not equal |
| !== | Not equal value or not equal type |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |

You may have noticed there are two versions of the equality (==, ===) and inequality (!=, !==) comparison operators (Table [3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#tab:comparison-operators)). The difference between them is that the former ones (==, !=) consider just the value, while the latter ones (===, !==) consider both value and type. What do the terms *equal value* and *equal type* mean? Consider the following example, where 5 and "5" have equal value (5) but unequal type (number vs. string). Comparing these two values gives different results when using the == and === operators. This is because == considers just the value, while === is more strict and considers both value and type:

"5" == 5; *// Returns true*

"5" === 5; *// Returns false*

Since the conversion rules of the == and != operators are complicated and unmemorable, it is not advised to use them. Instead, the === and !== operators, with their expected and strict behaviors, are recommended ([Crockford 2008](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#ref-crockford2008javascript)).

Boolean values can be combined with **logical operators**:

* &&—**AND** operator, true only if both values are true
* ||—**OR** operator, true if at least one of the values is true[24](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn24)
* !—**NOT** operator, true if value is false

For example:

**let** x = 6;

**let** y = 3;

x < 10 && y > 1; *// Returns true*

x == 5 || y == 5; *// Returns false*

!(x == 5 || y == 5); *// Returns true*

Boolean values are commonly used for flow control, which we learn about later on (Section [3.10](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#flow-control)).

**3.6.2.4 Undefined.** (undefined) means that a variable has been declared but has not been assigned with a value yet. For example:

**let** x;

x; *// Returns undefined*

**3.6.2.5 Null.** (null) is another special data type, representing lack of a value. For example:

**let** x = **null**;

**3.6.3 Objects**

**3.6.3.1 Array.**JavaScript **arrays** are used to store an ordered set of values in a single variable. An array can be created by putting zero or more values inside a pair of square brackets ([ and ]), separating the values with commas (,). For example:

**let** a = []; *// Empty array*

**let** b = [1, 3, 5]; *// Array with three elements*

Array items can be accessed using the brackets ([) along with a numeric **index**. Note that index position in JavaScript starts at 0. For example:

b[0]; *// Returns 1*

b[1]; *// Returns 3*

b[2]; *// Returns 5*

An array can be composed of different types of values, though this is rarely useful:

**let** c = ["hello", 10, **undefined**];

An array element can also be an array, or an object (Section [3.6.3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#objects-1) below). For example, an array of arrays can be thought of as a **multi-dimensional** array:

**let** d = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];

To access an element inside a multi-dimensional array we can combine several array access operations:

d[2][1]; *// Returns 8*

In the last expression, we used d[2] to access the element in position 2 of d, which is [7, 8, 9]. This is actually the *third* element of d, since, as you remember, JavaScript index positions start at 0. Then, we used d[2][1] to access the *second* element of [7, 8, 9], which gives us the final value of 8[25](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn25).

**3.6.3.2 Objects.**JavaScript **objects** are collections of named values[26](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn26). An object can be defined using curly brackets ({ and }). Inside the brackets, there is a list of name: value pairs, separated by commas (,). For example:

**let** person = {

firstname: "John",

lastname: "Smith",

age: 50,

eyecolor: "blue"

};

The above expression defines an object named person. This object is composed of four named values, separated by commas. Each named value is composed of a name (such as firstname) and a value (such as "John"), separated by a colon (:).

The named values are also called object **properties**. For example, the above object has four properties, named firstname, lastname, age, and eyecolor. The property values can be of any data type, including primitive data types (as in the above example, "John", "Smith", 50, "blue"), but also arrays and objects. Property values can also be functions, in which case they are referred to as *methods* (Section [3.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#methods)).

Objects are fundamental to JavaScript, and almost everything we work with in JavaScript is an object. The rationale of an object is to bind related data and/or functionality into a single collection. The collection usually consists of several variables and functions, which are called properties and methods when they are inside objects, respectively.

A JavaScript object is comparable to a real-life object. For example, a car can be thought of as an object (Figure [3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fig:js-object)). The car has properties like weight and color, which are set to certain values, and it has methods, like starting and stopping.

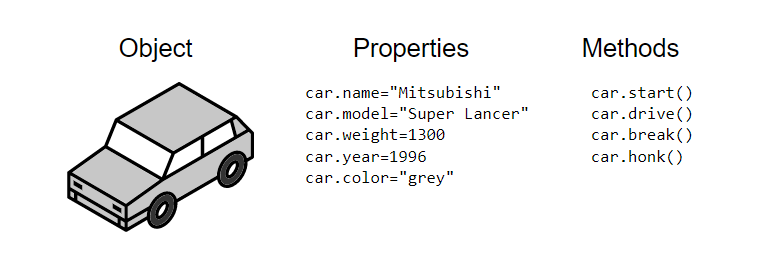


FIGURE 3.2: A JavaScript object has properties and methods, just like a real-life object

Object properties can be accessed using either of the following two methods:

* The **dot** notation (.)
* The **bracket** notation ([)

In both cases, we need to specify the object name and the property/method name. For example, getting the person properties using the dot notation looks like this:

person.firstname; *// Returns "John"*

person.age; *// Returns 50*

person.firstname + " is " + person.age + " years old.";

* What do you think will be returned by the third expression in the above code section?
* Create the persons object in the console and run the expression to check your answer.

The same can be accomplished using the bracket notation, as follows:

person["firstname"]; *// Returns "John"*

person["age"]; *// Returns 50*

person["firstname"] + " is " + person["age"] + " years old.";

When using the bracket notation, property names are specified as strings, in quotes. This means the dot notation is shorter to type, but the bracket notation is useful when the property name is specified with a variable:

**let** property = "firstname";

person[property]; *// Returns "John"*

person.property; *// Returns undefined*

The first expression works, returning the firstname property value "John". The second expression doesn’t work, returning undefined, since it looks for a property named property which doesn’t exist.

As we already mentioned, an object property value can be another object, or an array, rather than a primitive data type. For example, we could arrange the information in the person object in a different way:

**let** person = {

name: {firstname: "John", lastname: "Smith"},

age: 50,

eyecolor: "blue"

};

Instead of having individual firstname and lastname properties, we now have a name property which is an *internal* object, containing the firstname and lastname properties. To reach an internal object within another object, the dot notation (or the bracket notation) can be repeated several times in the same expression:

person.name.firstname; *// Returns "John"*

This is typical syntax that you will see a lot in JavaScript code:

object1.object2.property;

* Note the **auto-complete** functionality, which can make it easier to interactively construct this kind of expression in the console.
* For example, create the person object in the console, then start typing person. or person.name. to observe the auto-complete suggestions.

Object properties can also be modified via assignment. For example, we can change the person name from "John" to "Joe" as follows[27](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn27):

person.name.firstname = "Joe";

person.name.firstname; *// Returns "Joe"*

**3.6.4 Checking type of variables.**The typeof operator can always be used to query the type of variable we are working with. The following expressions demonstrate the use of typeof on primitive data types (Section [3.6.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#primitive-data-types)):

**typeof** "a"; *// Returns "string"*

**typeof** 1; *// Returns "number"*

**typeof** **false**; *// Returns "boolean"*

**typeof** **undefined**; *// Returns "undefined"*

**typeof** **null**; *// Returns "object" (!)*

Note that typeof returns "object" when applied to null, even though null is a primitive data type. This behavior is considered to be a [bug](http://2ality.com/2013/10/typeof-null.html) in the JavaScript language, kept for legacy reasons.

Arrays and objects are collectively classified as *objects* (Section [3.6.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#objects-data-type)):

**typeof** [1,5]; *// Returns "object"*

**typeof** {a:1}; *// Returns "object"*

**3.7 Functions**

A **function** is a block of code designed to perform a particular task. If different parts of our JavaScript code need to perform the same task, we do not need to repeat the same code block multiple times. Instead, we *define* a function once, then *call* it multiple times, whenever necessary.

A function is *defined* with:

* The function keyword
* A function name of our choice, such as multiply
* Parameter names separated by commas, inside parentheses (( and )), such as (a, b)
* The code to be executed by the function, curly brackets ({ and }), such as {return a \* b;}

The function code may contain one or more expressions. One or more of those can contain the return keyword, followed by a value the function **returns** when executed. When a return expression is reached the function stops, and the value after return is returned by the function. In case there is no return expression in the function code, the function returns undefined.

For example, the following expression **defines** a function named multiply. The multiply function has two **parameters**, a and b. The function returns the result of a multiplied by b.

*// Function definition*

**function** multiply(a, b) {

**return** a \* b;

}

Once the function is defined, you can execute it with specific **arguments**. This is known as a function **call**. For example, the following expression is a function call of the multiply function, returning 20:

*// Function call*

multiply(4, 5);

Note the distinction between parameters and arguments. Function *parameters* are the names listed in the function definition. Function *arguments* are the real values passed to the function when it is called. In the above example, the parameters of the multiply function are a and b. The arguments in the above function call of multiply were 4 and 5.

A function does not necessarily have any parameters at all, and it does not necessarily return any value. For example, here is a function that has no parameters and does not return any value[28](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn28):

**function** greeting() {

console.log("Hello!");

}

The code in the greeting function uses the console.log function, which we have not met until now. The console.log function prints text into the console.

greeting(); *// Prints "Hello!" in the console*

The console.log function is very helpful when experimenting with JavaScript code, and we will use it often. For example, when running a long script we may wish the current value of a variable to be printed out, to monitor its change through our program.

To keep the returned value of a function for future use, we use assignment[29](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn29):

**let** x;

x = multiply(4, 5);

* What is the value of x after executing the above two expressions?

**3.8 Methods**

**3.8.1 What are methods?**When discussing objects (Section [3.6.3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#objects-1) above), we mentioned that an object property can also be a function, in which case it is also known as a **method**. For example, a method may be defined when creating an object, in case one of the properties is assigned with a function definition:

**let** objectName = {

...,

methodName: **function**() { expressions; }, *// Method definition*

...

}

Note that, in case of methods, the function is defined a little differently—there is no function name between the function keyword and the list of parameters (). This type of function is called an **anonymous function**. Once a method is defined, it can be accessed just like any other property:

objectName.methodName(); *// Method access*

The parentheses () at the end of the last expressions imply we are making a function call for the method. For example, let us define a car object (Figure [3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fig:js-object)), which has several properties with primitive data types, and one method named start:

**let** car = {

name: "Mitsubishi",

model: "Super Lancer",

weight: 1300,

year: 1996,

color: "grey",

start: **function**() { console.log("Car started!"); }

};

Generally, methods define the actions that can be performed on objects. Using the car as an analogy, the methods might be *start*, *drive*, *brake*, and *stop*. These are actions that the car can perform. For example, *starting* the car can be done like this:

car.start(); *// Prints "Car started!"*

As a more realistic example, think of an object representing a layer in an interactive web map (Section [6.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\leaflet.html#adding-vector-layers)). The layer object may have properties, such as the layer geometry, symbology, etc. The layer object may also have methods, such as methods for adding it to or removing it from a given map, adding popup labels on top of it, returning its data in a different format, and so on.

* Add another method to the car object, named stop, which prints a "Car stopped!" message in the console.
* Try using the new stop method with car.stop().

**3.8.2 Array methods.**In JavaScript, primitive data types and arrays also have several predefined properties and methods, even though they are not strictly objects[30](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn30). In this section, we cover three useful properties and methods of arrays: .length, .pop, and .push. Another array method .forEach, related to loops, will be introduced later when we discuss loops (Section [3.10.3](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#loops)).

The .length property of an array gives the number of elements that it has:

**let** a = [1, 7, [3, 4]];

a.length; *// Returns 3*

a[2].length; *// Returns 2*

* In the above code section:
  + Why does the first expression return 3?
  + Why does the second expression return 2?

The .pop and .push methods can be used to remove or to add an element at the end of an array, respectively. The .pop method removes the last element from an array. For example:

**let** fruits = ["Orange", "Banana", "Mango", "Apple"];

fruits.pop(); *// Removes the last element from fruits*

Note that the .pop method, like many other JavaScript methods, modifies the array *itself*, rather than creating and returning a modified copy of it. The returned value by the .pop method is in fact the item which was removed. In this case, the returned value is "Apple". In other words, when executing the above expressions the value "Apple" is printed, while the new value of fruits becomes ["Orange", "Banana", "Mango"].

The .push method adds a new element at the end of an array:

**let** fruits = ["Orange", "Banana", "Mango", "Apple"];

fruits.push("Pineapple"); *// Adds a new element to fruits*

Again, note that the .push method modifies the original array, which means that fruits now has length of 5 (including "Pineapple" at the end), after the above expressions are executed. The returned value of the .push method is the new array length. In this example, the returned value is 5.

**3.9 Scope**

Variable [**scope**](https://en.wikipedia.org/wiki/Scope_(computer_science)) is the region of a computer program where that variable is accessible or “visible”:

* Variables defined with let inside a **block**—a code section encompassed in curly brackets ({})—are only accessible in the block where they are defined.
* A variable defined with let outside of any block is known as a **global** variable. Global variables are accessible anywhere in the program. That is, all expressions in the same script, inside or outside of any code block, can access global variables.

In the following code section, carName is a global variable. It can be used inside and outside of the code block of the function named myFunction:

**let** carName = "Suzuki";

*// Code here can use carName*

**function** myFunction() {

*// Code here can use carName*

}

Here carName is confined to the code block of myFunction. It can only be used inside the code block of myFunction, where it was defined[31](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn31):

*// Code here \*cannot\* use carName*

**function** myFunction() {

**let** carName = "Suzuki";

*// Code here can use carName*

}

In general, it is not recommended to create global variables unless you intend to, since they can conflict with other variables having the same name in the global environment. For example, trying to re-define a variable that was already defined with let will result in an error. Local variables, on the other hand, can share the same name in several different blocks without resulting in a conflict, since each variable is confined to its own block.

**3.10 Flow control**

**3.10.1 What is flow control?**By default, all expressions in our script are executed in the given order, top to bottom. This behavior can be modified using **flow control** expressions. There are two types of flow control expressions: conditionals and loops.

* **Conditionals** are used to condition code execution based on different criteria.
* **Loops** are used to execute a block of code a number of times.

**3.10.2 Conditionals.**The if **conditional** is the most commonly used conditional. It is used to specify that a block of JavaScript code will be executed if a condition is true. An if conditional is defined as follows:

**if**(condition) {

*// Code to be executed if the condition is true*

}

For example, the following conditional sets greeting to "Good day" if the value of hour is less than 18:

**if**(hour < 18) {

greeting = "Good day";

}

The else statement can be added to specify an *alternative* block of code, to be executed if the condition is false:

**if**(condition) {

*// Code to be executed if the condition is true*

} **else** {

*// Code to be executed if the condition is false*

}

For example, the following conditional sets greeting to "Good day" if hour is less than 18, or to "Good evening" *otherwise*:

**if**(hour < 18) {

greeting = "Good day";

} **else** {

greeting = "Good evening";

}

**Decision trees** of any complexity can be created by combining numerous if else expressions. For example, the following set of conditionals defines that if time is less than 10:00, set greeting to "Good morning", if not, but time is less than 18:00, set greeting to "Good day", otherwise set greeting to "Good evening":

**if**(hour < 10) {

greeting = "Good morning";

} **else** {

**if**(hour < 18) {

greeting = "Good day";

} **else** {

greeting = "Good evening";

}

}

This type of code may be used to create a customized greeting on a website, for instance. As another example, think of the way you can **toggle** layers on and off in a web map by using an if statement to see whether the layer is currently visible. If visible is true, hide the layer, and vice versa.

Conditionals can also be used in map symbology, as we will see later on (Section [8.4](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\symbology-and-interactivity.html#varying-style)). Consider the following example of a function that determines color based on an attribute. The function uses conditionals to return a color based on the “party” attribute: "red" for Republican, "blue" for Democrat, and "grey" for anything else:

**function** states\_color(p) {

**if**(p === "Republican") **return** "red";

**if**(p === "Democrat") **return** "blue";

**return** "grey";

}

Note that in the above example, the brackets ({}) around conditional code blocks are omitted to make the code shorter, which is legal when code blocks consist of a single expression. Also note that the code does not use else. Instead it relies on the fact that when return is encountered the function terminates. That way, when p is equal to either "Republican" or "Democrat" the funtion returns "red" or "blue", respectively, and terminates. Otherwise, the last expression is executed, and the function return the “default” value "grey"[32](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn32).

* Define the states\_color function, by copying the above code and pasting it in the console.
* Try running the states\_color function three times, each time with a different argument, to see how you can get all three possible returned values: "red", "blue", or "grey".

**3.10.3 Loops**

**3.10.3.1 Standard for loop syntax.Loops** are used to execute a piece of code a number of times. JavaScript supports several types of loops. The most useful kind of loop within the scope of this book is the for loop.

The **standard** for loop has the following syntax:

**for**(statement 1; statement 2; statement 3) {

*// Code block to be executed*

}

where:

* **Statement 1** is executed once, before the loop starts.
* **Statement 2** defines the condition to keep running the loop.
* **Statement 3** is executed in each “round,” after the code block is executed.

In practice, *statement 1* usually initializes a counter variable, such as let i=0. *Statement 2* contains a condition referring to i, such as i<5. *Statement 3* increments i (Section [3.6.2.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#number)), such as i++. For example:

**let** text = "";

**for**(**let** i = 0; i < 5; i++) {

text += "The number is " + i + "<br>";

}

Let’s try to locate the for loop components in the above example:

* **Statement 1** sets a variable before the loop starts (let i=0).
* **Statement 2** defines the condition for the loop to keep running (i<5 must be true, i.e., i must be less than five).
* **Statement 3** increases the value of i (i++) each time the code block in the loop has been executed, using the increment operator ++ (Section [3.6.2.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#number)).

As a result, the code block is executed five times, with i values of 0, 1, 2, 3, and 4. A loop is thus an alternative to repetitive code. For example, the above for loop is an alternative to the following code section, which gives exactly the same result *without* using a loop:

**let** text = "";

text += "The number is " + 0 + "<br>";

text += "The number is " + 1 + "<br>";

text += "The number is " + 2 + "<br>";

text += "The number is " + 3 + "<br>";

text += "The number is " + 4 + "<br>";

* Execute the for loop from the above code section, then print the value of the text variable to see the effect of the loop.

Here is another example of a for loop. In this example, the code will be executed 1000 times, since i gets the values of 0, 1, 2, etc., up to 999.

**for**(**let** i = 0; i < 1000; i++) {

*// Code here will run 1000 times*

}

**3.10.3.2 Iterative for loop syntax.**In addition to the standard for loop syntax shown above, there is a special syntax to iterate over elements of arrays or objects. In this **iterative** syntax, instead of having three statements, we define the loop with just one statement:

**for**(**let** i **in** object) {

*// Code block to be executed*

}

where i is a variable name (of our choice) and object is the object which we iterate over. In each iteration, i is set to another object element name. In case object is an array, i gets the array index values: "0", "1", "2", and so on. In case object is an object, i gets the property names. For example:

**let** obj = {a: 12, b: 13, c: 14};

**for**(**let** i **in** obj) {

console.log(i + " " + obj[i]);

}

This loop runs three times, once for each property of the object obj. Each time, the i variable gets the next property name of obj. Inside the code block, the current property name (i) and property value (obj[i]) are being printed, so that the following output appears in the console:

a 12

b 13

c 14

For the next for loop example, we define an object named directory:

**let** directory = {

musicians: [

{firstname: "Chuck", lastname: "Berry"},

{firstname: "Ray", lastname: "Charles"},

{firstname: "Buddy", lastname: "Holly"}

]

};

The directory object has just one property, named musicians, which is an array. Each element in the directory.musicians array is an object, with firstname and lastname properties. Using a for loop we can go over the musicians array, printing the full name of each musician[33](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn33):

**for**(**let** i **in** directory.musicians) {

**let** musician = directory.musicians[i];

console.log(musician.firstname + " " + musician.lastname);

}

This time, since directory.musicians is an array, i gets the array indices "0", "1", and "2". These indices are used to select the current item in the array, with directory.musicians[i]. As a result, all of the musician names printed in the console[34](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn34):

Chuck Berry

Ray Charles

Buddy Holly

**3.10.3.3 The .forEach array method.**The .forEach array method (Section [3.8.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#array-methods)) can be used as a cleaner and shorter alternative to for loops (Section [3.10.3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#iterative-for-loop-syntax)) in case we wish to go over the array and apply a function on each element.

The .forEach method accepts a function that will be applied on each element.The function passed to forEach, also takes one argument: the contents of the array element. The parameter name of the internal function is for us to choose, though the name element is a common convention. We could choose any different name. The important point is that the parameter refers to the element contents which we can do something with in the function body. Here is a small example of the .forEach array method:

**let** a = [52, 97, 104, 20];

a.forEach(**function**(element) {

console.log(element);

});

In the first expression, we define an array named a. In the second expression, we are applying an (anonymous) function on each element of a, using .forEach. The function takes one argument, element. The function code includes just a console.log function call to print element. As a result, all elements of a are printed in the console:

52

97

104

20

As another example, the following code is the forEach version of the last code example from Section [3.10.3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#iterative-for-loop-syntax) for printing musician names:

directory.musicians.forEach(**function**(element) {

console.log(element.firstname + " " + element.lastname);

});

Here is another example with the same data. In this case, we “extract” the musician names into a new array named musicians, instead of just printing them into the console. First, musicians is initialized as an empty array. Then, in each iteration, the current musician name is appended to the musicians array using the .push method (Section [3.8.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#array-methods)):

**let** musicians = [];

directory.musicians.forEach(**function**(element) {

musicians.push(element.firstname + " " + element.lastname);

});

* Run the above code in the console, after difining the directory object.
* Examine the contents of the resulting musicians array.

Note how the code using .forEach is shorter and avoids the need to declare a counter i, compared to for loops (Section [3.10.3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#iterative-for-loop-syntax)). Most loops we are going to use in this book will be iterations over arrays, in which case we are going to use .forEach for its simplicity. In other cases, we will resort to for.

**3.11 JavaScript Object Notation (JSON)**

**3.11.1 JSON.**[**JavaScript Object Notation (JSON)**](https://en.wikipedia.org/wiki/JSON) ([Bassett 2015](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#ref-bassett2015introduction)) is a data format closely related to JavaScript objects. It is a **plain text** format, which means that a JSON instance is practically a character string, which can be saved in a plain text file (usually with the .json file extension), in a database, or in computer memory[35](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn35).

JSON and JavaScript objects are very similar and easily interchangeable. For that reason, JSON is the most commonly used format for exchanging data between the server and the client in web applications. The principal difference between JSON and JavaScript objects is as follows:

* A JavaScript **object** (Section [3.6.3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#objects-1)) is a data type in the JavaScript environment. A JavaScript object does not make sense outside of the JavaScript environment.
* A **JSON** instance is a plain text string, formatted in a certain way according to the JSON [standard](https://json.org/). A JSON string is thus not limited to the JavaScript environment. It can exist in another programming language, or simply stored inside a plain text file or a database.

For example, we already saw (Section [3.6.3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#objects-1)) that a JavaScript object can be created using curly brackets ({}), in an expression such as the following one:

**let** obj = {a: 1, b: 2};

The corresponding JSON string can be defined, and stored in a variable, as follows:

**let** json = '{"a": 1, "b": 2}';

A notable difference between how the above two variables are defined is that in a JSON string the property names must be enclosed in quotes: "a" and "b" (it’s possible to enclose property names in quotes when defining an object too, but it’s not mandatory). The JSON standard requires double quotes (") around property names, therefore we need to enclose the entire string with single quotes (').

To make a JSON string useful within the JavaScript environment, the JSON string can be **parsed** to produce an object. The parsing process, **JSON→object**, is done with the JSON.parse function. The JSON.parse function is used to convert a JSON string (e.g., coming from a server) to a JavaScript object:

JSON.parse(json); *// Returns an object*

The opposite conversion, **object→JSON**, is done with JSON.stringify:

JSON.stringify(obj); *// Returns a JSON string*

The JSON.stringify function is commonly used when sending an object from the JavaScript environment to a server. Since, like we said, JavaScript objects cannot exist outside of the JavaScript environment, we need to convert the object to a string with JSON.stringify before the data are sent elsewhere. For example, we are going to use JSON.stringify in Chapter [13](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\collaborative-mapping.html#collaborative-mapping) when sending spatial layers to permanent storage in a database (Section [13.6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\collaborative-mapping.html#sending-features-to-the-database)).

Finally, keep in mind that JSON is limited in that it cannot store undefined values or functions. JSON *can* store all of the other JavaScript data types (Section [3.6.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#overview-data-types))[36](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn36), namely:

* Strings
* Numbers
* Booleans
* null
* Arrays
* Objects

**3.11.2 GeoJSON.**[**GeoJSON**](https://en.wikipedia.org/wiki/GeoJSON) is a spatial vector layer format based on JSON. Since GeoJSON is a special case of JSON, it can be processed in the JavaScript environment using the same methods as any other JSON string, such as JSON.parse and JSON.stringify (Section [3.11.1](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#javascript-json)). For this reason, GeoJSON is the most common data format for exchanging spatial (vector) data on the web.

Here is an example of a GeoJSON string:

{

"type": "Feature",

"geometry": {

"type": "Point",

"coordinates": [125.6, 10.1]

},

"properties": {

"name": "Dinagat Islands"

}

}

This particular GeoJSON string represents a point layer with one attribute called name. The layer has just one feature, a point at coordinates [125.6, 10.1]. Don’t worry about the details of the GeoJSON format at this stage. We will cover the syntax of the GeoJSON format in Chapter [7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\geojson-1.html#geojson-1).

A variable containing the corresponding GeoJSON string can be created in a JavaScript environment with the following expression:

**let** pnt = '{' +

'"type": "Feature",' +

'"geometry": {' +

'"type": "Point",' +

'"coordinates": [125.6, 10.1]' +

'},' +

'"properties": {' +

'"name": "Dinagat Islands"' +

'}' +

'}';

Note that the string is defined piece by piece, concatenating with the + operator, to make the code more readable. We could likewise define pnt in a single line, using a long string and without needing +.

GeoJSON can be parsed with JSON.parse just like any other JSON:

pnt = JSON.parse(pnt); *// Returns an object*

Now that pnt is a JavaScript object, we can access its contents using the dot (or bracket) notation just like with any other object (Section [3.6.3.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#objects-1)):

pnt.type; *// Returns "Feature"*

pnt.geometry.coordinates; *// Returns [125.6, 10.1]*

pnt.geometry.coordinates[0]; *// Returns 125.6*

pnt.properties; *// Returns {name: "Dinagat Islands"}*

Going back to a GeoJSON string is done with JSON.stringify:

JSON.stringify(pnt); *// Returns a string*

The above expression gives the following result[37](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn37):

{"type":"Feature","geometry":{"type":"Point","coordinates":[**...**]}

Note that JSON.stringify takes further arguments to control the formatting of the resulting string. For example, JSON.stringify(pnt, null, 4) will create the following indented, multi-line string—much like the one we began with. This is much easier to read than the default one-line result shown above:

{

"type": "Feature",

"geometry": {

"type": "Point",

"coordinates": [

125.6,

10.1

]

},

"properties": {

"name": "Dinagat Islands"

}

}

**3.12 Exercise.**Write JavaScript code, as follows.

* The code starts with defining two arrays:
  + The *first* array A represents a single two-dimensional coordinate, such as: [100, 200].
  + The *second* array B contains several coordinates, as internal arrays of length 2, such as: [[0, 3], [0, 4], [1, 5], [2, 5]].
* Then, the code creates a *new* array, where the individual coordinate A is attached to each of the coordinates in B, so that we get an array of coordinate pairs, such as: [[[100, 200], [0, 3]], [[100, 200], [0, 4]], [[100, 200], [1, 5]], [[100, 200], [2, 5]]][38](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fn38).
* Hint: create an empty array result, then run a loop that goes over the second array B, each time “pushing” (Section [3.8.2](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#array-methods)) the A element plus the current B element into result.
* Solution to this exercise, and most of the other excercises in the book, can be found in Section [C](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\list-of-exercise-solutions.html#list-of-exercise-solutions).

**References**

Bassett, Lindsay. 2015. *Introduction to JavaScript Object Notation: a to-the-point guide to JSON*. Sebastopol, CA, USA: O’Reilly Media, Inc.

Crockford, Douglas. 2008. *JavaScript: The Good Parts*. Sebastopol, CA, USA: O’Reilly Media, Inc.

1. **Node.js** (<https://nodejs.org/en/>) is a prominent example of a *server-side* environment for executing JavaScript code.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref19)
2. At the time of writing, the console inside the developer tools in Chrome *does* allow redefining variables defined with let, which makes code testing easier (<https://stackoverflow.com/questions/62911715/redeclaration-with-var-works-but-not-with-let-in-chrome-snippets#answer-62912023>). Elsewhere—e.g., in scripts executed on page load—variables defined with let cannot be redefined.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref20)
3. More information on +=, and other assignment operators, can be found in the *Assignment Operators* reference (<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators>) by Mozilla.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref21)
4. For more information on strings, see the *Strings* reference (<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Strings>) by Mozilla.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref22)
5. For more information on numbers, see the *Numbers and Operators* reference (<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Math>) by Mozilla.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref23)
6. Note the distinction between && and &, and between || and |. The latter, & and |, are known as *bitwise operators* and are not intended for combining comparison expressions.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref24)
7. For more information on arrays, see the *Arrays* article (<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Arrays>) by Mozilla.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref25)
8. JavaScript objects are comparable, for example, to *dictionaries* in Python or to *lists* in R.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref26)
9. For more information on objects, see the *Objects* reference (<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics>) by Mozilla.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref27)
10. As mentioned above, a function with no return statement returns the default value of undefined.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref28)
11. For more information on functions, see the *Functions* article (<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Functions>) by Mozilla.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref29)
12. The mechanism which makes this happen is beyond the scope of this book.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref30)
13. If you assign a value to a variable that has not been declared, it will automatically be defined as a global variable using the var keyword, regardless of *where* the assignment was made. This is not recommended as it leads to ambiguity in our code, and we will not use this approach.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref31)
14. For more information on conditionals, see the *Making decisions in your code—conditionals* article (<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/conditionals>) by Mozilla.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref32)
15. Note that it is possible to redefine a local variable inside the code block of a for loop, since a new separate variable is created in each “round” of the loop (<https://stackoverflow.com/questions/34952429/javascript-variable-declaration-within-loop>).[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref33)
16. For more information on loops, see the *Looping code* article (<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Looping_code>) by Mozilla.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref34)
17. Other well-known plain text data formats are, for instance, *Comma-Separated Values (CSV)* (<https://en.wikipedia.org/wiki/Comma-separated_values>) and *Extensible Markup Language (XML)* (<https://en.wikipedia.org/wiki/XML>).[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref35)
18. For more details on the difference between a JavaScript object and a JSON string, check out the *StackOverflow question* on this matter (<https://stackoverflow.com/questions/6489783/whats-the-difference-between-javascript-object-and-json-object>).[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref36)
19. The last part of the string is omitted and replaced with [...] to fit on the page.[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref37)
20. The resulting array can be useful for drawing line segments between a single point A and a second set of points B (Figure [11.8](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\spatial-queries-1.html#fig:example-11-04)).[↩︎](file:///C:\Users\User\Downloads\Telegram%20Desktop\Web_Gis\Web_Gis\js\javascript-basics.html#fnref38)