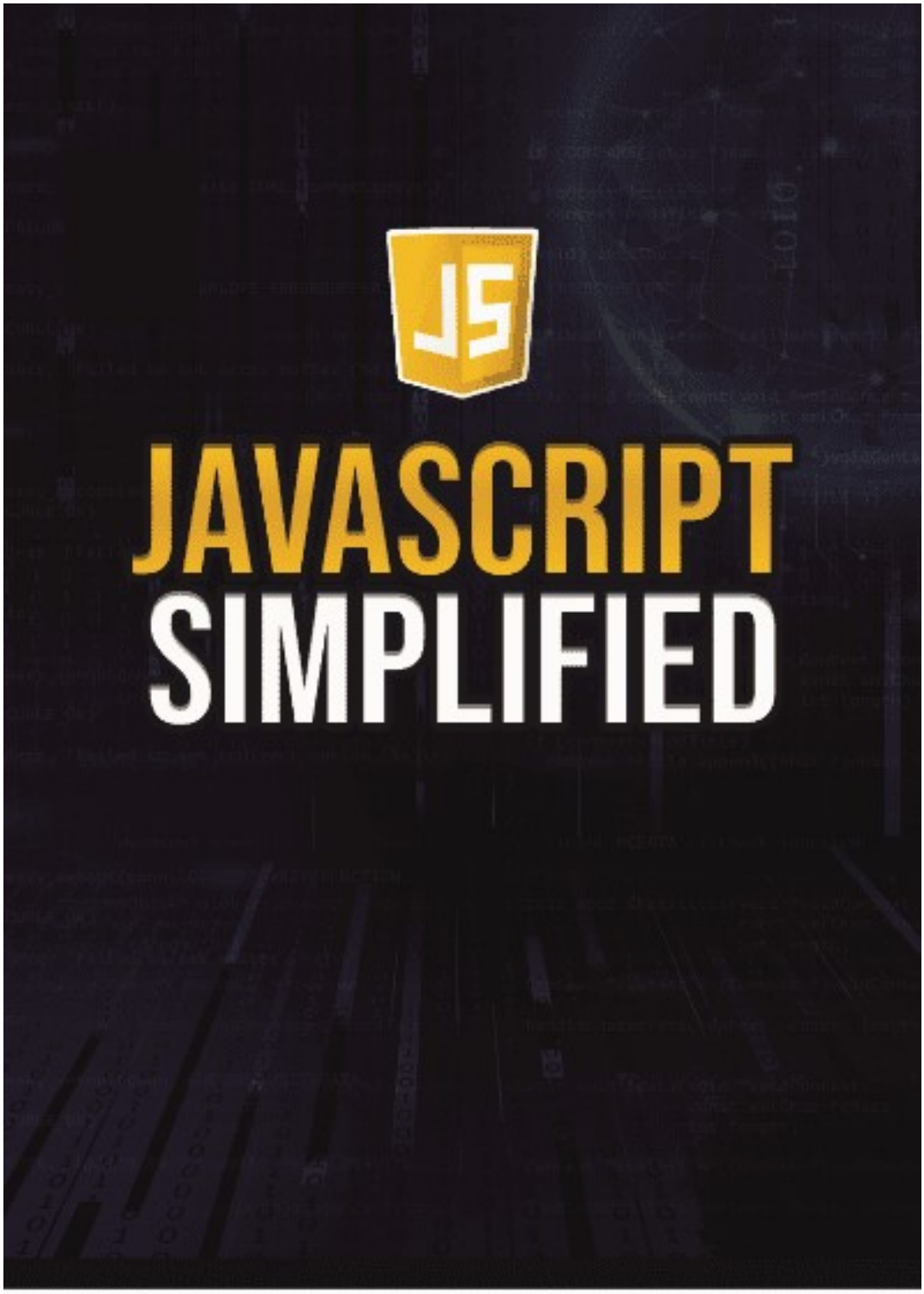

Taye Abidakun

DEDICATION

I dedicate this book to my first teachers, that is, my parents. My parents were teachers before they retired from the teaching profession. They made me love teaching. I want to make programming simple for many students, the same way my father made mathematics simple for me. One day, I hope to master the art of teaching like my father.

ACKNOWLEDGEMENT

Glory to God for making this book a reality. I appreciate my parents, Mr. and Mrs. Abidakun for their motivations and inspirations.

Special thanks to all staffs and students of SQI college of ICT for the privilege to learn, serve and develop at the great institution. I found a mentor in the CEO of SQI, Dr. Adeyemi Aderinto.

Thanks to my friends that went through this book in developmental stage to help edit and correct it. Teslim, Janet, Beyond and Favour. I appreciate you.

PREFACE

I started writing this book as an instructor at SQI, driven by the personal need to record the simplicity with which I teach JavaScript to my students in class and help them reconnect to these class moments, but **JavaScript Simplified** is more than a chronicle or a tutorial. The greater reason for this book is to help a lot more people trying to learn web development get comfortable creating web applications with JavaScript, getting beyond just learning syntaxes from programming lessons to confidently knowing how they work, when and where to use different programming concepts.

This book was written based on my classroom experiences teaching JavaScript. It contains some of my students frequently asked questions and the solutions I provide when asked such questions.

This book will be of great help to JavaScript instructors and students especially those into frontend development. The only prerequisite to this book is the knowledge of HTML.

For beginners, I will recommend that you follow the order of the chapters in this textbook.

To get the best out of this book, please code along.

Have a great time as you read the simplest textbook on JavaScript.

Taye Abidakun

tayeabidakun@gmail.com

March 20, 2022.

TABLE OF CONTENT

INTRODUCTION

Why JavaScript?

With the knowledge of HTML and CSS, you can build a responsive webpage, but there is more to frontend than a responsive webpage. JavaScript allows you to build an interactive webpage.

What does this mean?

Look at this scenario: You are bored at home and you need something to entertain yourself. With your phone with you, you have two options.

Option 1: Watch movie on your phone.

Option 2: Play game on your phone.

If you choose option 1, then you play the video and watch what others have acted for you. Your action does not determine the story. The button you press on your phone will not affect the story line. Yours is to watch and follow the storyline. It is a storyline that does not depend on your inputs.

If you choose option 2, then your action will determine the next event in the game. How you play will determine whether you will win or lose. If you are playing a football tournament, the way you play will determine whether you will qualify for the next round or not. It means, as a user, you can control some of the events. You aren't just a spectator.

With the knowledge HTML and CSS, you can provide the first option for your users, that is, you can create a webpage where users can come, view what you put there but not more. A static webpage.

With JavaScript, you will be able to provide the second option, you will be able to create a page that interacts with your users. A page that carries out some actions and displays results based on users input. An interactive webpage.

Question: What is the difference between a static webpage and an interactive webpage?

Answer: Users can relate with an interactive webpage but not with a static webpage.

Now that you know the reason for learning this, let's answer the question: What is JavaScript?

According to developer.mozilla.org "JavaScript is a scripting or programming language that allows you to implement complex features on web pages". Before moving further, let's discuss what a programming language is, as this will help understand what is discussed in this book better. A programming language is used to give instructions to a computer. Programming languages have some similarities with our day-to-day languages like English, Yoruba, French, etc.

One of these similarities is “keyword”. Keywords are words with predefined meanings.

These keywords can be likened to verbs in English language. They are used to give instruction to someone. When you say them, the person receiving the instruction immediately knows what to do. Go, Come, Sleep, Jump, Run etc. are keywords in English language. When you tell someone to come, the person immediately knows that he/she should approach you. When you tell someone to go, the person immediately knows that he/she is to move away from you. Keywords also work this way in programming languages. When you type some words, your computer immediately knows what to do. To learn a programming language, then you need to know some of the keywords in the language.

“Oh, I need to start learning a whole lot of things” you are already thinking.

Here is a question for you: Do you know all the verbs in English language? No. So you don’t need to learn all the keywords in a programming language, you only need to learn few of them for a start. Relating this to English language, you need to learn the “sleep”, “eat”, “come”, “go”, “run” etc. of a programming language (those verbs used on a daily basis) and ignore the “disembogue”, “impignorate” etc. (Not frequently used verbs).

Still on keywords.

While “come” means “to move closer” in English, “come” is a jargon in French. A French speaker will be lost when you say “come”. In order to make a French speaker move closer to you, then you go ahead and search for what “come” means in French, then say it to him/her. Same way, a keyword in one programming language maybe a jargon in another programming language. To code in JavaScript, you need to be familiar with some of the keywords in JavaScript. In this book, I will be showing you some of the keywords in JavaScript and how to use them.

You also need to know JavaScript inbuilt methods (also called inbuilt commands). Like keywords, inbuilt methods also have predefined meanings.

Now that you know what JavaScript is, let’s take it a step further and write our first JavaScript code.

Question: What is JavaScript?

Answer: A programming language used to give instructions to our webpages.

One way to get the best out of this textbook is to code along. Happy reading and coding.

SECTION ONE

JAVASCRIPT BASICS

CHAPTER ONE

WRITING YOUR FIRST JAVASCRIPT CODES.

Getting started

Fig 1.1

Fig 1.1 is a HTML code. Let's add JavaScript codes to it. As seen in the image above, the name of this file is 'index.html'. It means the extension of the file is '.html'. If we run this file on the browser, the browser naturally assumes that we only have HTML codes in this file because of this '.html' extension.

How do we write JavaScript codes in a HTML file? By opening a script tag as shown below

index.html

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <script type="text/javascript">
        // JavaScript codes will be here.
    </script>
</body>
</html>
```

Fig 1.2

All JavaScript codes will be within the script tag. The script tag tells the browser “Oh browser, JavaScript codes are here”. If we run this page on the browser, we won’t see anything on the page. I will type a number in my JavaScript code.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <script type="text/javascript">
        5
    </script>
</body>
</html>
```

Fig 1.3

If I run this code on the browser, I will not see anything on the browser, this is because, I am yet to instruct my code to display the number I typed. The browser will not automatically display whatever I type. I have to instruct the browser to display it before it does. The

browser does not know if I want my users to see what I typed or not, therefore it will not automatically display what I typed until I ask it to do so.

The next question is: How do I instruct my browser to display something?

To do that, I will use an inbuilt command. Remember I told you inbuilt commands have predefined meanings, that is, whenever the browser sees any of these inbuilt commands, it automatically knows what to do. The inbuilt command to display something on the browser is 'alert'.

This is how it looks like:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title></title>
5  </head>
6  <body>
7      <script type="text/javascript">
8          alert(5);
9      </script>
10 </body>
11 </html>
```

Fig 1.4

To display something, write "alert" and put whatever you want to display/alert in a parenthesis just after writing the "alert" as displayed in fig 1.4.

Run this on the browser, and you will have this result.

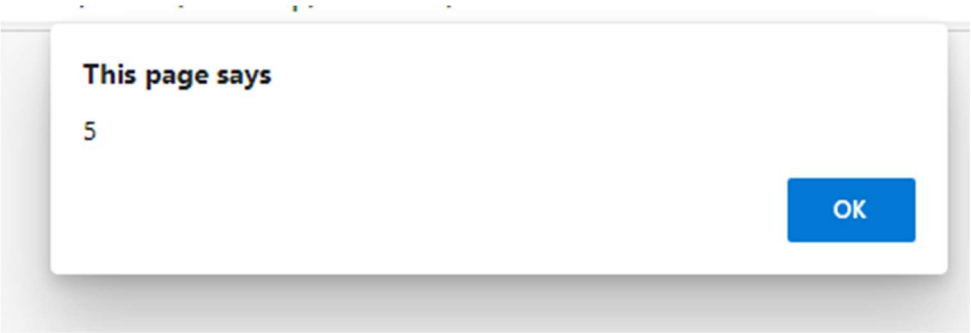

Fig 1.5

Can you see a dialog box that displays 5? Immediately the browser sees “alert”, it creates a dialog box and put the value within the parenthesis, right inside the dialog box. This dialog box is not the same as CSS modal, this is created by JavaScript, not CSS.

Let’s move forward. JavaScript has the ability to do some arithmetic, let’s do this together.

I will write some basic mathematics operation within the alert keyword, and you will see JavaScript evaluate it for me.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <script type="text/javascript">
        alert(6 * 9);
    </script>
</body>
</html>
```

Fig 1.6

If I run the code above on the browser, I will get 54.

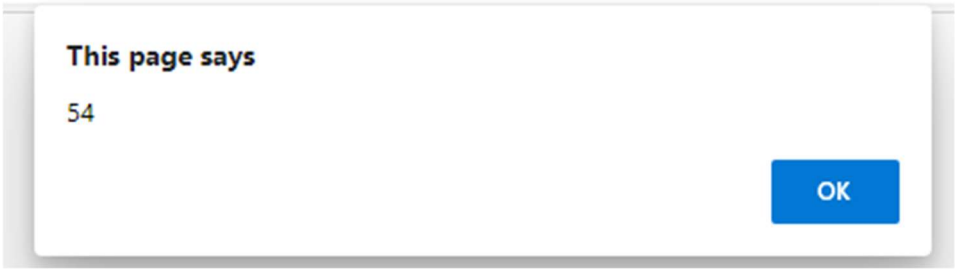

Fig 1.7.

We have 54 as the result because JavaScript has the ability to evaluate mathematical operations.

Ensure you code along. Run it on your browser and see the result yourself. Don’t just read, code along.

Whenever JavaScript sees a mathematical expression, it evaluates it and gives the result.

Problem 1.1

What will be the out of the codes below?

1. alert(3 -1);
2. alert(4-7);
3. alert(7 * 5);
4. alert(4+1+12);
5. alert(3 + 4 * 6);

Make sure you give the answers before moving forward.

Answer:

1. 2;
2. -3
3. 35
4. 17
5. 27

The last question gave an answer of 27, this shows that JavaScript follows BODMAS principle.

You might have these questions running through your mind:

Question 1.1

a. You have been talking about JavaScript without linking it up with HTML, does that mean JavaScript and HTML are two distinct entities without any relationship?
b. Is JavaScript only meant to solve mathematical problems?

I will answer the first question now while the second question will be answered in chapter four.
For the first question, JavaScript and HTML are meant to work together. Let's connect our JavaScript to HTML.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <div id="result" style="background-color: red; color: white"></div>
    <script type="text/javascript">
        alert(5);
    </script>
</body>
</html>
```

Fig 1.8

If I run this code on the browser, I will see a dialog box like before.

Fig 1.9

Nothing different, what I'm about to do now is: Instead of displaying the 5 in a dialog box, I will display it in a div. All I need to do is to call on the id of the div and place the number inside the div.
According to the code above, the id of the div I created is "result" (you can use any id of choice).
To do this, I will run the code below

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <div id="result" style="background-color: red; color: white"></div>
    <script type="text/javascript">
        result.innerText = 5;
    </script>
</body>
</html>
```

Fig 1.10

Here is the result on the browser.

Fig 1.11

Explanation:

To display a JavaScript value in your HTML, create a container in your HTML (I created div, you can create a span or use other tags), give the container an id of your choice (I gave mine an id of "result"), then set the innerText to the value of your choice(I set the innerText to 5,you can set it to any value). As its name implies, innerText refers to the text inside a container.

In summary, I called on a container using the id I gave the container and used the innerText command to set the text inside the container to a value of my choice.

Example
Create a span, give it an id of "answer", use JavaScript to set the text inside it to 20.

Answer:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <span id="answer"></span>
    <script type="text/javascript">
        answer.innerText = 20;
    </script>
</body>
</html>
```

Fig 1.12

Problem 1.2:

1. Create a paragraph tag in HTML, give it an id of your choice, use JavaScript to set the text inside it to 3;
2. Create a button, give it an id of your choice, use JavaScript to set the text inside the button to the product of 3 and 5 (which is 15);

Solution:

1.
```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <p id="myParagraph"></p>
    <script type="text/javascript">
        myParagraph.innerText = 3;
    </script>
</body>
</html>
```

Fig 1.13

2.
```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <button id="calculate"></button>
    <script type="text/javascript">
        calculate.innerText = 3 * 5;
    </script>
</body>
</html>
```

Fig 1.14

In the solution to the first and second questions, I used id of “myParagraph” and “calculate” respectively, you can use any id of your choice, you will still get the same result

Problem 1.3: Create an input tag, give it any id of your choice and set the innerText to 10;

Solution:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <input id="inp">
    <script type="text/javascript">
        inp.innerText = 10;
    </script>
</body>
</html>
```

Fig 1.15

Run the code on the browser and check the result.

Fig 1.16

Oops! Why is it not displaying 10 as the innerText of the input tag? The innerText worked well when we used it with button, div, span but it seems not to work now. What is going on?

Editable and uneditable containers

I group all containers into two: editable containers and uneditable containers. By default, you cannot change the content of uneditable containers on the browser, while the opposite is true for editable containers. Are you confused with that statement? Here is a clearer explanation:

If I have a div, and I set the content of the div to ‘I am a boy’ in my HTML code. Can a user visiting my page change the content of the div? No, the user can only read whatever I put in my div.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <div>I am a boy</div>
</body>
</html>
```

Fig 1.17

Here is the result on the browser:

I am a boy

Fig 1.18

Visitors of a website cannot change the content of a div, except they have access to the source code, therefore, I called a div an uneditable container.

Let us look at another scenario:

If I have an input, and I set the content of the input to ‘I am a boy’ using HTML. Can a user visiting my page change the content of the input? Yes, by default, visitors of a page can change the value of an input when they visit the page.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <input value="I am a boy">
</body>
</html>
```

Fig 1.19

As a visitor, I can type inside the input on the browser and change the content to the input to anything I want:

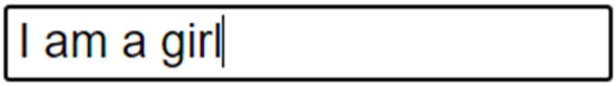

Fig 1.20

As shown above, I have typed inside the input and I have changed the content of the input from “I am a boy” to “I am a girl”, therefore an input is an editable container.

List of some editable containers (HTML tags)

- input
- textarea
- select

List of some uneditable containers (HTML tags)

- div
- span
- h1-h6
- footer
- p

As a rule, use **innerText** to set the content of uneditable containers and use **value** to set the content of editable containers.

Now back to the problem at problem 1.3, we want to set the content of an input to 10.

To do that, create an input, give it an id of your choice and set the value to 10.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <input id="inp">
    <script type="text/javascript">
        inp.value = 10;
    </script>
</body>
</html>
```

Fig 1.21

Run it on the browser

Fig 1.22

Now, we can see 10 in the input. I have explained the first question at question 1.1, JavaScript and HTML are meant to work together.

We have succeeded in using our JavaScript code to set the content of our HTML tags.

The second question in problem 1.1 will be explained in chapter four.

CHAPTER TWO

VARIABLES

Need for a variable.

Problem 2.1:

Create a div, a span and an input, give the three elements different ids (Note: An Id is meant for only one container, no two containers should share the same id);

Use JavaScript to set the content of the three containers to the result of: 2 + 3 * 6 – 7

Here is the solution:

```
<body>
    <div id="display"></div>
    <span id="result"></span>
    <input id="inp">
    <script type="text/javascript">
        display.innerText = 2 + 3 * 6-7;
        result.innerText = 2 + 3 * 6-7;
        inp.value = 2 + 3 * 6-7;
    </script>
</body>
```

Fig 2.1

I had to write “2 + 3 * 6-7” three times because we want to display the result three times.

Let’s continue, add two more divs, using JavaScript, multiply the result of “2 + 3 * 6-7” by 2 and place it inside one of the divs (so that you will have 26 in the div). Multiply the result of “2 + 3 * 6-7” by 5 and place it inside the second newly created div (so that you will have 65 in the div).

Here is the solution:

```
 6  <body>
 7      <div id="display"></div>
 8      <span id="result"></span>
 9      <input id="inp">
10
11      <!-- The newly added containers -->
12      <div id="first"></div>
13      <div id="second"></div>
14      <script type="text/javascript">
15          display.innerText = 2 + 3 * 6-7;
16          result.innerText = 2 + 3 * 6-7;
17          inp.value = 2 + 3 * 6-7;
18          first.innerText = (2 + 3 * 6-7) * 2;
19          second.innerText = (2 + 3 * 6-7) * 5;
20      </script>
21  </body>
```

Fig 2.2

Again, we had to type "(2 +3 * 6-7) * 2" and "(2 +3 * 6-7) * 5".

There is a better solution. Since we keep reusing "(2 +3 * 6-7)", we can save it in a container, then call on the value in the container each time we need it. These containers that hold values are known as **variables**. There are three containers available in JavaScript, they are: var, let and const. I will only explain "var" in this chapter and explain the remaining two under ES6 section.

A variable is a container that holds value(s). When you keep reusing a value (as in the fig 2.2), it is better to keep it in a container. Saving values in containers makes it easy for us to reuse values.

Let us first consider the process of creating a variable before using it to make the codes in fig 2.2 better.

Creating a variable:

```
7    <script type="text/javascript">
8        var a = 5;
9        alert(a);
10   </script>
```

Fig 2.3

Explanation:

”var” is a keyword in JavaScript keyword, meaning JavaScript automatically understands it immediately it sees it. Whenever your browser sees “var”, it creates a container for you. After creating a container, you **must** label the container. Whatever you type after the ‘var’ is the label you are giving to the container. In fig 2.3, I typed “a” after var, it means I am labeling my container “a”. You can give your container any label of your choice (I will write on those labels that are not allowed later in this chapter). After labeling a container, the next is to insert a value into the container.

To insert a value into the container, you use the equal sign “=”. Whatever is at the right hand side of the equal sign “=” will be inserted into the container. As in the case above, I inserted 5 into the container.

Point to note: All containers must be labeled.

Problem 2.2: Create a container, label it “num” and insert 12 in the container. Then alert “num”

Solution:

```
<script type="text/javascript">
    var num = 12;
    alert(num);

</script>
```

Fig 2.4

You will see 12 on the browser. Whenever you call on a variable, the value inside that variable will be placed at that point.

Problem 2.3: Create a variable labeled “n”, set 3 as the value of n, then place the product of n and 2 in a div (such that the div will eventually contain 6).

Solution:

```
 6  <body>
 7      <div id="result"></div>
 8      <script type="text/javascript">
 9          var num = 3;
10          result.innerText = num * 2;
11      </script>
12  </body>
```

Fig 2.5

Problem 2.4:

Go back to fig 2.2, create a variable named “arithmetic”, with a value of 2 + 3 * 6 -7, then use “arithmetic” so that you get the same result that we got initially on the browser without rewriting 2+3*6-7.

Solution:

```
 6  <body>
 7      <div id="display"></div>
 8      <span id="result"></span>
 9      <input id="inp">
10
11      <!-- The newly added containers -->
12      <div id="first"></div>
13      <div id="second"></div>
14      <script type="text/javascript">
15          var arithmetic = 2 + 3 * 6 - 7;
16          display.innerText = arithmetic;
17          result.innerText = arithmetic;
18          inp.value = arithmetic;
19          first.innerText = arithmetic * 2;
20          second.innerText = arithmetic * 5;
21      </script>
22  </body>
```

Fig 2.6

As seen above, we now have a cleaner code. Instead of using 2+3 * 6-7 everytime, all we have to do is to call on “arithmetic” since “arithmetic” holds the value 2+3 * 6-7.

Changing the value in a variable:

What will I get if I run the code below on the browser? a. 2, b. 4, c. 42. d. 24

```
16 <script type="text/javascript">
17      var num = 4;
18      num = 2;
19      alert(num);
20 </script>
```

Fig 2.7

Answer: a. 2

Explanation:

JavaScript reads codes line by line. This mean it will read line 1 before line 2, then to line 3,till it is done. On a given line, JavaScript reads codes from left to right.

On line 17 in the code above, JavaScript will see “var” and will immediatley create a container for us. Then it will see “num” and label the container “num”. Then it sees “=” and knows you are about to place a value inside the container. Then it will see 4 and will immediately place 4 in the container. Lastly on that line, it will see ‘;’ and knows you have terminated the statement. It moves to line 18.

On line 18, immediately it sees “num” without seeing “var” before it, JavaScript knows you are not creating a new container. Remember I told you JavaScript will create a container immediately it sees “var”, but because there is no “var” here, it will not create a new container. Instead, it will assume you have a container somewhere that you have already created. Therefore, the “num” on line 18 is only a reference to the “num” that was created initially. Then, it sees the equal sign “=”, it knows you are trying to place a value in your container. Then, it will see 2. The 2 will override whatever value it initially meets in the container.

Whenever you place a value in a container, it overrides whatever value it meets there.

Then it sees ‘;’ and terminate the statement, then it goes to the next line.

On line 19, it sees the alert command and alert the value of “num”. Since 2 has overriden 4, it displays 2.

Problem 2.5: What will be the output of these codes on the browser?

```
16 <script type="text/javascript">
17     var num = 4;
18     alert(num);
19     num = 2;
20     alert(num);
21 </script>
```

Fig 2.9

a. 2 only.
b. I will first display 2, then 4.
c. I will first display 4, then 2.
d. 4 only.

Answer: c. It will first display 4, then 2.

Explanation: Remember, JavaScript reads codes from top to the bottom. On line 17, it will create a container and label it num, then insert 4 in the container. On line 18, it will alert the value in num (which is 4). On line 19, it will replace the value in num with 2. On line 20, it will alert 2 since we now have 2 as the value of num.

Let us look at another scenario

Problem 2.6: What will I get if I run the code below on the browser?

a. 2 only.
b. I will first display 2, then 4.
c. I will first display 4, then 2.
d. 4 only.

```
16 <script type="text/javascript">
17     var num = 4;
18     alert(num);
19     var num = 2;
20     alert(num);
21 </script>
```

Fig 2.10

Answer: c. It will first display 4, then 2.

As you can see, the answer is the same as that of fig 2.9. Although the two blocks of codes produce the same results, they are not the same.

Explanation: On line 17, I created a container called num and inserted 4 in it. On line 18, I displayed the value of num using an alert. On line 19, I created another container called num and I inserted 2 in the new container. JavaScript will create a new container on line 19 because of the var keyword. The num on line 17 and on line 19 are not the same. They are two separate containers.

On line 20, I asked JavaScript to display num. We now have two containers, both labeled num. JavaScript will be confused on which to display. To solve the problem, it will use its "instinct", it will say to itself "for this developer to create another container and labeled it num, then the new container must be more valuable to him/her than the first container."

JavaScript will then display the value inside the second container (the newly created container).

This is the reason we got 2 on line 20. It then means that the container created on line 17 will be useless after line 19 (since there is a new container with the same name).

Point to note: When you have two or more containers/variables bearing the same name, JavaScript uses the newly created container making the previously created ones redundant. Avoid giving the same name to more than one container.

It is important that you understand the explanation here as it will be needed when we get to functions.

Before leaving this part, let us look at the rules that guide variable declaration.

Rules that guide variable declaration

The label given to our containers must follow these rules:

a. A variable name must not be a number. Example, writing: var 3 = 7; is not allowed.
b. A variable name must be a single word (It must not contain whitespaces). Example, writing var my number = 12; is not allowed. The space between "my" and "number" will be seen as an error.
c. The only two special characters allowed to use while writing a variable name are: $ (dollar sign) and _(underscore). Example, var @num = 10; is not allowed, while var _num = 10; is allowed.
d. A variable name must not start with a number; Example, var 4num = 12; is not allowed, while var num4 = 12; is allowed.
e. A variable name cannot be a keyword. Example, var var = 2; is not allowed, while var num = 2; is allowed. This is because var is a keyword while num is not a keyword. Click on this link to check all the keywords in JavaScript.
f. Variables are case sensitive. var name = "Taye"; and var Name = "Taye"; are not the same.

CHAPTER THREE

NAMING CONVENTIONS

Reason for naming conventions

A variable is a container that holds value. It is a good practice to make the label of the container explains the value it holds. This is what I mean: Let us assume I have 5 chairs, so I asked three developers to create a variable to hold the number of chairs I have.

The code below shows the response of the three developers

```
<script type="text/javascript">
    var a = 5; //first developer//
    var chairs = 5; //second developer//
    var number of chairs owned by Taye = 5; //third developer//

</script>
```

Fig 3.1

Which of these is the best: a. First developer b. Second developer c. Third developer

Let us analyze their responses one after the other.

The first developer labeled his variable “a”. If he is collaborating with other developers, they will find it difficult using the variable, because it is hard for anyone to think and know that “a” stands for “number of chairs”. “Oh! what if he is working on the project alone?” you may ask. This can still cause problem to the developer. If this developer goes back to his code after 1 year, he may find it difficult to know the meaning of “a”.

The second developer labeled her variable ‘chairs’. This is more explanatory than that of the first developer but the third developer’s answer is the most explanatory. It is easy to understand. If the third developer goes back to the codes after a year or two, it will be easy to remember the purpose of the variable. Therefore, always make the label of your variable explains the value it holds.

Ironically, the codes of the first and second developers will run smoothly while the third developer’s won’t. Though it is easy to understand, it violates the rules of variable declaration. Remember, part of the rules of variable declaration is that, a variable should not have whitespaces.

Let us make a little adjustment to the code of the third developer so that it no longer violates the rules of variable declaration.

```
<script type="text/javascript">
    var a = 5; //first developer//
    var chairs = 5; //second developer//
    var numberofchairsownedbyTaye = 5; //third developer//

</script>
```

Fig 3.2

I have removed all the whitespaces and we now have valid variable name.

But we may encounter some words that may imply something else when the whitespaces are removed.

```
<script type="text/javascript">
    var go on = true; //before space is removed
    var goon = true; //after space is removed
</script>
```

Fig 3.3

In the codes above, I have a variable name called “go on”, on removing the spaces, I have “goon” which means something else. We are faced with two problems.

The first is that, we want to make our variable name explains the value it holds. The second is that variable names must not have whitespaces. Meanwhile, removal of whitespaces will create a new word with an entirely different meaning.

This is where the naming conventions come in.

Some naming conventions

I will be mentioning three naming conventions.

- Pascal case (also known as upper camel case).
- Camel case (also known as lower camel case).
- Snake case

1. Pascal case: In this model, you start each word with an upper case.
   Let’s rewrite our variable name using Pascal casing.

```
 7      <script type="text/javascript">
 8          var go on = true; //before space is removed
 9          var GoOn = true; //Pascal case
10      </script>
```

Fig 3.4

Uppercase signifies the beginning of a new word. With this, it will be easy to read.
Let us rewrite our first example using Pascal casing:

```
7    <script type="text/javascript">
8        var number of chairs owned by Taye = 5; //before space is removed
9        var NumberOfChairsOwnedByTaye = 5; //Pascal case
10   </script>
```

Fig 3.5

2. Camel case: This is similar to Pascal case except that you start the first word with a lowercase while every other word begins with an upper case.
   Here are the camel case versions of the variables created above.

```
<script type="text/javascript">
    var goOn = true; //camel case
    var numberOfChairsOwnedByTaye = 5; //camel case
</script>
```

Fig 3.6

3. Snake case: To write in snake case, you separate each word from the next using an underscore since JavaScript allows you to use underscore in naming variables.

```
<script type="text/javascript">
    var go_on = true; //snake case
    var number_of_chairs_owned_by_taye = 5; //snake case
</script>
```

Fig 3.7

You can use any of these systems that you are most comfortable with but in this book, I will be using the camel case. Most developers in the JavaScript community use camel case and I will advise that you stick to this.

Problem 3.1

Write these words in Pascal case, camel case and snake case:

1. Age of student
2. Highest mountain in the world
3. Longest word in dictionary

Answer:

1. Age of student
   a. Pascal case: AgeOfStudent
   b. Camel case: ageOfStudent
   c. Snake case: age_of_student
2. Highest mountain in the world

    a. Pascal case: HighestMountainInTheWorld
    b. Camel case: highestMountainInTheWorld
    c. Snake case: highest_mountain_in_the_world
3. Longest word in dictionary
    a. Pascal case: LongestWordInDictionary
    b. Camel case: longestWordInDictionary
    c. Snake case: longest_word_in_dictionary

CHAPTER FOUR

DATATYPES

Introduction

So far, we have been dealing mainly with numbers, does that mean that JavaScript can only deal with numbers? Is JavaScript only meant to solve mathematical problems?

This brings us to data types.

What are datatypes?

Let me use this analogy

There are seven main classes of foods which are carbohydrates, proteins, vitamins, fats, minerals, fibre and water. When we eat, the body checks if the substance eaten can be processed as a food (that is, is it among these 7 classes of foods?). If it can, then the body sends the food through the digestive tract. In the digestive tract, the body then goes deeper to check the specific content of the food (whether it is carbohydrate or protein, or minerals, or water, or vitamins or fibre). The check is necessary since the way the digestive tract processes proteins is different from the way it processes water.

If I consume rice, the body will ask itself, "is this item among the classes of foods?". Then it answers itself with a yes (if no, the body says "oh, this is not processable by the digestive tract"). The body asks itself another question, "since the consumed item is a food, which class of food does the consumed item belongs to". The body checks and answers itself "carbohydrates". The body then says "Oh, let me begin the digestion from the mouth since it is carbohydrates".

Let us go back to JavaScript.

Classes of foods = datatypes

Food = data to be processed

Body = JavaScript

When we try to process data, JavaScript will ask itself, "is this data among the data types?" If it answers itself with "Yes", it means the data is processable by JavaScript. If it answers itself with "no", it will throw an error.

If the first question returns "Yes", then it asks itself, "which class does this data belongs to among the classes of data (datatypes)?" Then it will do some background checks and returns an answer to itself. Finally, it will process the data the way its class (datatype) should be processed.

With the explanation above, it means datatypes do two things:

1. It tells JavaScript if a data is processable.

2. It tells JavaScript how to process a data.

Datatypes in JavaScript

Before we proceed, let me explain why we are learning this.

The datatype of a given data tells JavaScript how to process the data. In order words, the datatype determines the things JavaScript can do to a data. When you do something to a data, it is known as an operation. It therefore means that the datatype of a data determines the kind of operation JavaScript can perform on the data.

In this book, I will be explaining 5 datatypes in JavaScript.

1. Number
2. Boolean
3. String
4. Undefined
5. Object
    i. Null
    ii. Array
    iii. Object

There is an inbuilt command in JavaScript to check the datatype of any value. The command is: typeof.

1. Number: In JavaScript, a number can either be a whole number or a floating number (a number that contains decimal point).
   It means: 5, 7, 109, 8.6, 52.06 are all classified as numbers in JavaScript.

```
<script type="text/javascript">
    alert(typeof(50));
</script>
```

Fig 4.1

If you run this on the browser, you will get:

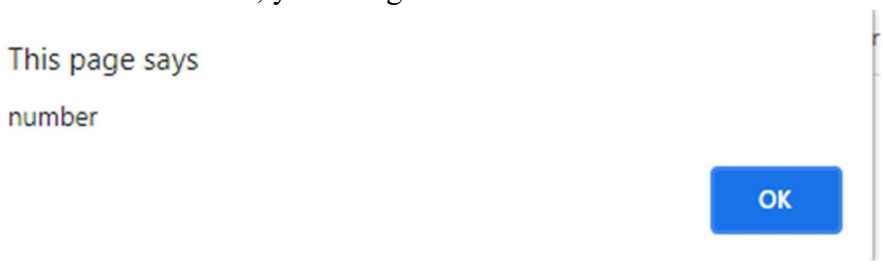

Fig 4.2

Run these codes on your browser:

1. alert(typeof(3.21));
2. alert(typeof(123.456));
3. alert(typeof(32.1));

Each of the codes above will give you “number”. It means JavaScript sees them as numbers.

What will be the output of these codes?

```
<script type="text/javascript">
    var myNumber = 4;
    alert(typeof(myNumber));
</script>
```

Fig 4.3

Answer: number

Explanation: **A variable automatically takes the datatype of the value it holds.**

2. Boolean: A Boolean has only two values. It is either true or false.

```
<script type="text/javascript">
    var goOn = true;
    alert(typeof(goOn));
</script>
```

Fig 4.4

If we run this on the browser, we will get:

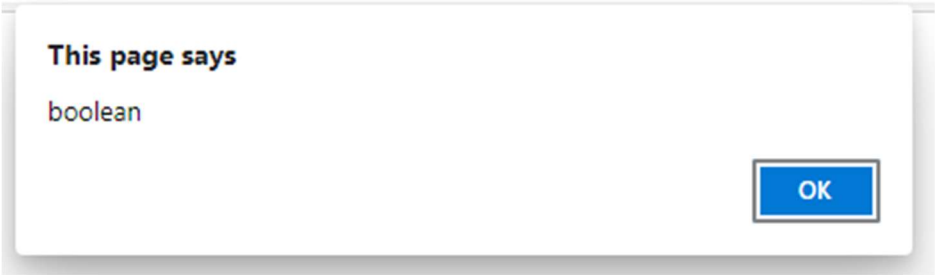

Fig 4.5

Again, a boolean is a value that is either true or false.

3. String: To understand what a string is, let us consider the codes below.

```
 7  <body>
 8
 9
10  <div id="result"></div>
11  <script type="text/javascript">
12      var name = taye;
13      result.innerText = name;
14  </script>
15  </body>
```

Fig 4.6

When we run this code on the browser, “taye” is expected to be placed inside the div, but somehow, nothing is displayed on the browser.

On line 10, I created a div with the id of “result”. On line 12, I created a variable with a name of “name” and I inserted “taye” into the variable.

Why isn’t it working? To understand what is going on, consider the code below and run it on the browser

```
 9  <script type="text/javascript">
10      alert(a);
11  </script>
```

Fig 4.7

You will see nothing. This is because something is wrong with the codes. What do you think is wrong with the codes in fig 4.7?

Answer: We asked JavaScript to alert “a” but JavaScript does not understand the meaning of “a”.

It does not know what “a” means. Immediately JavaScript sees alert(a), it goes to its brain, and ask itself, “Is there any container labeled ‘a’?” If the answer is yes, then it will display the value in the container labeled “a” but if it is no, then it will throw an error.

Since there is no container labeled “a”, then it throws an error and we see nothing on the screen.

Note: Remember, in JavaScript, we use the ‘var’ keyword to create containers.

Here is another example:

```
 9 <script type="text/javascript">
10     var a = 5;
11     alert(a);
12 </script>
```

Fig 4.8

When we run this on this on the browser, we will see 5 because we have created a container labeled “a” on line 10, then we used alert to display the value in container labeled “a” on line 11. If we wrote line 11, without line 10 (that is, if we alert “a” without declaring variable “a”), it will not work because JavaScript will not understand the meaning of “a”.

When JavaScript sees a text that is not a keyword, it immediately assumes it is the name of a container. It goes to its memory to see if it is a variable name. If it is not, then it goes to HTML to see if it is the id of a container (div, span, input etc.). If it is not, it throws an error and nothing is displayed on the screen.

Let us go back to the initial issue:

```
 7 <body>
 8
 9
10 <div id="result"></div>
11 <script type="text/javascript">
12     var name = taye;
13     result.innerText = name;
14 </script>
15 </body>
```

Fig 4.9

Immediately JavaScript sees the text “taye”. It goes to its memory to see if there is any variable labeled “taye”. Since we didn’t create any variable with the name “taye”, so JavaScript will see nothing. Then it says to itself, “oh, since I can’t find any variable (container in JavaScript) labeled “taye”, let me check HTML to see if I will find any container labeled “taye”, that is, elements (div, span, input) with the id of ‘taye’”.

Unfortunately, there's none in HTML. So JavaScript will throw an error.

How do I say this to JavaScript "My beloved JavaScript, I don't want you to go to your memory searching for the container labeled "taye" because there is no container labeled 'taye'".

In order words, how do we tell JavaScript, not to see "taye" as the label of a container but instead, we want it to see "taye" as a text?

We simply put it in quotes. By putting a text in quotes, you are telling JavaScript that, it is a text and not the label of a container. So, it will take it for what it is -a text- and will not bother itself looking for the container with that label.

```
6  <body>
7      <div id="result"></div>
8      <script type="text/javascript">
9          var name = "taye";
10         result.innerText = name;
11     </script>
12 </body>
```

Fig 4.10

If we run it on the browser, we will see the text "taye" on the screen as displayed below.

taye

Fig 4.11

You can use single quote or double quotes, whichever you are more comfortable with.

Anything you place within quotation marks is known as a string.

```
<body>


<div id="result"></div>
<script type="text/javascript">
    var name = "taye";
    result.innerText = typeof(name);
</script>
</body>
```

Fig 4.12

Result on the browser:

string

Fig 4.13

Make sure you code along and you also run the codes on your browser.

Problem 4.1

In the codes below, what will be the output of lines 9, 10, 11 and 12 respectively.

```
 7      <script type="text/javascript">
 8          var num = "54";
 9          alert(typeof(96));
10          alert(typeof("true"));
11          alert(typeof(num));
12          alert(typeof(false));
13      </script>
```

Fig 4.14

Answer: number, string, string, boolean.

Explanation: Line 9 will output “number” because 96 is a number. I explained earlier that whole numbers and floating point numbers are both classified under data type of number.

Line 10 will output “string” because the text is within quotation marks. Anything within quotation marks is regarded as a string.

Line 11 will also output “string”. Remember I said a variable takes the datatype of the value it holds. Variable “num” holds a value of “54”, the “54“ is in quotes which makes it a string. Therefore, the datatype of num is string.

Line 12 will give an output of boolean. A boolean is anything that is either true or false.

4. **Undefined:** Undefined and null are similar in usage even though they are of different datatypes. While undefined is a datatype, null is an object.

   I will explain the two at this point due to their similarities.

   Both undefined and null are used to denote emptiness.

   To explain the difference between the two, let us consider this: You want to construct a building, then you got a contractor to construct the building for you. The contractor hired some people and they helped you construct the building. Some months after the completion, you visited the building.

   Here is my question for you: Is the building empty?

   Answer: The building is empty in some sense and not empty is another sense. This sounds confusing, so I will explain better.

   We can say it is empty because you have not consciously placed any item in the building. We can say it is not empty because it contains leftover of cements used for construction and maybe weeds because it is left untouched for a while.

   At this point, it is at the state of **undefined**. Undefined means you have not consciously placed any value inside a container.

   To understand what null means in JavaScript, let us go back to the example of the building.

   On getting to the building, you decided to get cleaners to help clean up everywhere and remove everything inside the building. When the cleaners are done, we can then say the building is at the state of null.

   It means you have consciously removed everything present in a container.

   I hope the analogy above explains the difference between undefined and null.

   Let me explain the difference using another example:

   You rented an apartment. After two years, you packed out of the apartment.

Here is my question for you: Is the apartment empty?

Answer: The apartment is empty in some sense and not empty is another sense.

It is empty because you have packed all your loads from the apartment. It is not empty because you may have papers etc. that are no longer needed, left in the apartment.

At this point, it is at the state of **undefined**. When your landlord or his/her agent gets there and decided to clean up everywhere, wash up the apartment and get it empty indeed, then it is at the state of **null**.

The two analogies I used up here show that there are different ways to set the value of a variable to undefined (1. To construct a building and decide not to consciously put anything inside the building. 2. To pack out of an apartment. These are just two of many ways.), but there is only one way to set the value of a variable to null (to clean it up). This sentence is very important.

Problem 4.2: In your own way, explain the difference between undefined and null.

Now that we have seen the difference between undefined and null in theory, let us go to the code editor to write come codes.

```
<script type="text/javascript">
    var a;
    alert(a);
</script>
```

Fig 4.15

When we run the code above on the browser we will get:

Fig 4.16

The value of "a" is undefined because we created the container labeled "a" but we did not consciously place any value in the container. It therefore automatically assumes a value of undefined.

If we check the datatype of variable "a", we will also get "undefined".

Problem 4.3:

Create a span element in your HTML, then display the datatype of variable “a” inside the span element.

Solution:

```
<body>
<span id="display"></span>
<script type="text/javascript">
    var a;
    display.innerText = typeof(a);
</script>
</body>
```

Fig 4.17

If we run it on the browser, then we have:

undefined

Fig 4.18

Let us also look at **null**.

To set the value of a variable to null, you have to assign null to the variable.

```
<script type="text/javascript">
    var a = null;
    alert(a);
</script>
```

Fig 4.19

If we run this on the browser, we will have:

Fig 4.20

Problem 4.4:

Create an input element in your HTML, then display the datatype of variable “a” inside the input element.

Solution:

```
<body>
<input type="" id="display" nam
<script type="text/javascript">
    var a = null;
    display.value = typeof(a);
</script>
</body>
```

Fig 4.21

If we run this on the browser, we will have:

Fig 4.22

Remember I told you the datatype of **null** is **object**, this is the reason we have “object” as the answer.

I choose to explain **null** while explaining **undefined** due to their similarities.

Before leaving this section, I need to explain this.

Although the value of a variable is automatically set as undefined when created without a value, you can also consciously set the value of a variable to undefined.

Let us see it together.

```
<script type="text/javascript">
    var a = undefined;
    alert(a);
</script>
```

Fig 4.23

And we have this on the browser:

Fig 4.24

This means:

var a;

is the same as:

var a = undefined;

The value of "a" will be set to undefined in both instances. You can use whichever you are more comfortable with.

5. Object: This datatype allows you store collections of data.
   Under this datatype, we have:
   a. null
   b. Object
   c. Array

   Don't be confused, object is an example of the datatypes under object. While I have explained null, the remaining two (object and array) will not be explained in this chapter. Chapter ten is dedicated for object while array is explained in chapter eleven.

CHAPTER FIVE

OUTPUT COMMANDS

Output commands

We use output commands to display things on the browser. We have seen three of the output commands (alert, innerText and value), this section is to discuss two other important output commands that will be frequently used in this book. Before discussing the two, let us have a quick revision of the output commands we have learnt.

1. alert(): This is used to display information in a dialog box.
2. innerText: This is used to display information in uneditable containers
3. value: This is used to display information in editable containers.

The two other output commands I will be discussing here are:

1. innerHTML
2. console.log()

1. innerHTML: This is very similar to innerText except that innerHTML understands HTML codes while innerText does not.

Similarities of innerText and innerHTML

```
<body>
    <div id="res"></div>
    <div id="disp"></div>
<script type="text/javascript">
    res.innerText = "hello";
    disp.innerHTML = "hi";
</script>
</body>
```

Fig 5.1

If we run this on the browser, we will get:

hello
hi

As seen above, they produced the same result.

Difference between innerText and innerHTML

```
<body>
    <div id="res"></div>
    <div id="disp"></div>
<script type="text/javascript">
    res.innerText = "<h1>hello</h1>";
    disp.innerHTML = "<h1>hi</h1>";
</script>
</body>
```

Fig 5.2

If we run this on the browser, we will get:

<h1>hello</h1>

hi

Fig 5.3

As seen above, the innerText sees everything as a text while the innerHTML understands HTML codes.

2. console.log: This command is used to log things to the browser's console. Before explaining how console.log works, let us first understand what the browser's console is.
   The browser's console is usually used for debugging purpose. If you don't understand the last sentence, here is a better explanation.

   Debugging is the act of identifying and removing errors from your codes.

   You wrote some blocks of codes, ran it on the browser but it refused to work as expected. You went back to your codes, looked around for what you did wrong but you couldn't see anything. You became confused, without knowing what to do.
   A friend came, tapped your shoulder, asked you what the problem is. You narrated the incident to him. He looked at your codes and pointed his finger to the particular line where you made a syntax error. You fixed it, ran it on the browser and it worked perfectly. You are consoled. You are happy.

   That friend that came to tap your shoulder and pointed to the line where you had a syntax error is the browser's console.

The browser's console will point to the line of error whenever you have a syntax error in your JavaScript code. Aside that, you will need console for other debugging purposes.

How to open the browser's console: Open your browser, then right-click. Can you see a menu similar to the one in fig 5.4? Select "inspect".

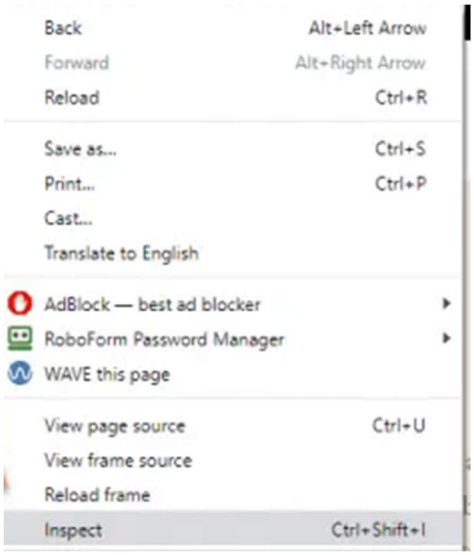

Image source

Fig 5.4

A window will be opened either at the left or at the bottom of your browser. On that window, you will see tabs labeled 'Elements', 'Console', 'Network', 'Application' etc.

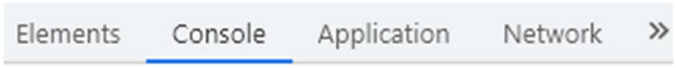

Fig 5.5

Our concern is the console tab. Click on the console tab to open it. Congratulations, you have successfully opened the browser's console.

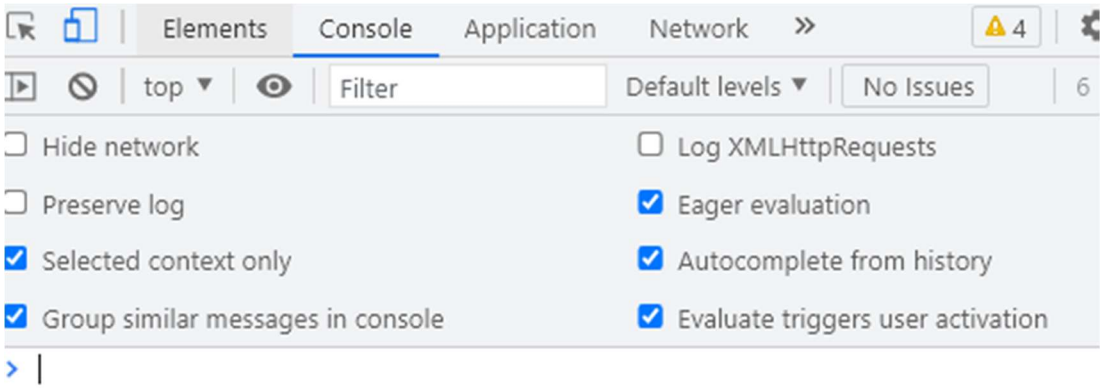

Fig 5.6

We will talk more about it uses later in this section. You can now close the browser's console. You can also open the browser's console by pressing the shortcut: ctrl + shift + i. Then click on 'console'.

Using the console

If we run the codes below on the browser, we won't see anything:

```
 9  <script type="text/javascript">
10      var num = 5;
11      aler(num);
12  </script>
```

Fig 5.7

The reason is because there is nothing called "aler" in JavaScript. I just committed an error. We can easily figure this out because we have just two lines of codes. Imagine we have a thousand lines of code. It will be difficult to figure out.

Let us check the console.

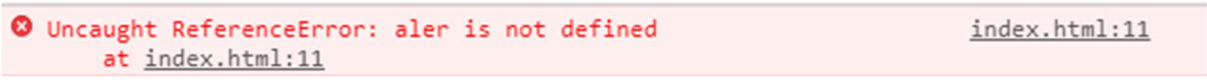

Fig 5.8

The console has helped us, telling us the type of error and the exact line with the error. We can now go ahead to fix it. When your JavaScript code doesn't work as expected, the first place to check is the browser's console. It will help you debug faster.

Now that you know what a browser's console is and how it works, let's us take a look at what **console.log** is and how it works.

The console only helps you identify syntax error, not logic error. Sometimes, your code won't work as expected, you will check the console and you won't see any error. It means the problem you have with your code is not syntax error but logic error. So, you go back to your code and check, line by line.

We know that when we have syntax error in our JavaScript codes, it will be reported to the console, but sometimes, you (as a developer) want to send some values to the console to debug logic errors.

Look at this scenario: I created a variable in my code on line 1, like this "var a = 4". On line 50, I had "result.innerText = a;". On running this on the browser, I suddenly saw 10 instead of 4 within the div with the id of 'result'. As expected, I was confused, and asked myself "What went wrong? What did I do wrong? How did the value of variable 'a' changed from 4 to 10?".

Then the debugging starts. What I can do is to go to any line of codes between line 1 and 50 to display the value of "a". Let's say I displayed the value of "a" on line 30 and I got 4, it means everything works fine between line 1 and 30 and the problem is somewhere between line 30 and line 50. With that, I can go to line 40 to display the value of "a". Let's say I got 10 on line 40, it means the problem happened before line 40. It means the problem is between line 30 and line 40. I can go to line 35 to display the value of "a". I can continue like that till I get the exact line causing the problem. Then I fix the bug.

When you are in a situation like this (debugging, trying to identify the exact line with error in your code), a convenient way of displaying the value of 'a' (or any variable that isn't working as expected) is to use the console.log. It is a very good tool for debugging.

Let us see how it works

```
 9 <script type="text/javascript">
10     var num = 5;
11     console.log(num);
12     console.log("hello");
13 </script>
```

Fig 5.9

When we run it on the browser, we will see '5' and 'hello' in the browser's console together with the lines displaying them.

I console.log "num" on line 11 and console.log "hello" on line 12. So, we will see this result in the browser's console as shown below.

```
5                                                    index.html:11
hello                                                index.html:12
```

Fig 5.10

Summary: The console is used for debugging purpose. When you have a bug, the console should be the first place to check to help identify the bug, and know if it is a syntax error. You can also log values to the console by using console.log while trying to debug a logic error.

Read more on error handling in chapter twenty-two.

CHAPTER SIX

JAVASCRIPT OPERATORS PART 1.

Operators are tools we use to perform operations on JavaScript items.

What is done = operations.

What we use = operators.

Items we perform the operation on = operands.

Let us redefine this: We use operators to perform operations on operands.

Problem 6.1: A carpenter used a hammer to drive a nail into a wood. Identify the operator, the operation and the operands.

Answer

The operation: Nailing a wood.

The operator: Hammer

The operands: Nail, wood.

The carpenter used a hammer to nail a wood.

Note: The operator is not transformed at the end of the process. Only the operands are transformed. You can see the operators as enzymes or catalysts that facilitate the process but they are not transformed at the end of the process. The work of an operator is to facilitate a process. To make things happen.

For instance, if I want to add 2 and 7 together, then I use the plus sign (+) to make it happen. 2+7 = 9.

The "+" here made the addition possible. It is the operator. It operated on 2 and 7, so they are the operands.

Let us dive into some of the operators we have in JavaScript.

1. Arithmetic operators
2. Assignment operators
3. String operator (Concatenator)
4. Conditional operators
5. Logical Operators
6. Ternary operator

Only the first two are discussed in this chapter. Others are discussed in subsequent chapters.

1. **Arithmetic operators**

These operators are used to perform arithmetic operations. The operations they perform include some of the operations that we are familiar with in basic mathematics: addition, subtraction, multiplication, division.

a. Addition: This operator is used when you need to add two numbers together. It is denoted with a plus sign (+).

```
<script type="text/javascript">
    alert(3 + 7);
</script>
```

Fig 6.1

When we run this on the browser, we will 10. It means the plus sign added 3 and 7 together just like it is in mathematics.

b. Subtraction: This is used to subtract one value from another. It is denoted with hypen (-). e.g 5-2 = 3; The hyphen between 5 and 2 performs the operation.
c. Multiplication: This is used to multiply two numerical values. It is denoted with the asterisk (*). E.g 4 * 3 = 12; The asterisk sign between 4 and 3 multiplied the two numbers together.
d. Division: This is used to divide two numerical values. It is denoted with forward slash (/). E.g 15/3 = 5; The forward slash between 15 and 3 divided 15 by 3. This is the reason we got 5 as the result.

As seen above, the +, -, /, * all work the same way they do in basic mathematics. Let us look at some other arithmetic operators.

a. Modulo: This is the remainder after division. It is denoted with the percentage sign.
   E.g 8 % 2 = 0.
   Explanation: If we divide 8 by 2, we will get 4 and there will be no remainder after the division, so the result is 0.
   Problem 6.1
   Give the answer to the questions below.
   i. 15%3 = ?
   ii. 17 % 5 = ?
   iii. 7 % 4 = ?
   iv. 4 % 6 = ?

   Solution

   i. 0.

   Explanation: If we divide 15 by 3, we will get 5 and there will be no remainder after the division, so the result is 0.

ii. 2.

Explanation: If we divide 17 by 5, we will get 3, remainder 2. Since modulo only deals with the remainder after division, the result will be the remainder (2).

iii. 3.

Explanation: If we divide 7 by 4, we will get 1, remainder 3. Since modulo only deals with the remainder after division, the result will be the remainder (3).

iv. 4.

Explanation: In this case, the denominator is greater than the numerator, 4 divided by 6 will give 0. Since we did not divide the numerator, the numerator will remain intact and we will get 4 as the result.

Tip: Whenever the denominator is greater than the numerator, the modulo will always give the numerator.

b. Increment: This operator is both an arithmetic operator and an assignment operator.
This operator will be explained under the assignment operators.

2. **Assignment operators**

This is used to assign a value to a variable. They include: =, ++, --, +=, -=, *=, /=

Let us look at these syntaxes one after the other

i. The equal sign (=): This is used to assign a value to a variable.

```
13  <script type="text/javascript">
14  var a = 5;
15  alert(a);
16  </script>
```

Fig 6.2

If we run this code on the browser, we will get 5. On line 14, we created a container (variable), labeled it ‘a’, then we used the equal sign (=) to insert 5 into ‘a’. Inserting a value into a variable is the same as assigning a value to the variable, thus, the equal sign (=) is an assignment operator.

ii. Increment (++): This is used to increase the value of a number by 1. It is denoted by ++.

There are two types of increment, they are:

a) Pre-increment
b) Post-increment

a) Pre-increment: This increases a value by 1 and updates the variable with the new value. It is written by placing the sign (++) before the variable you want to increment.

```
13  <script type="text/javascript">
14  var a = 5;
15  ++a;
16  alert(a);
17  </script>
```

Fig 6.3

On line 14, I created variable labeled “a”, assigned 5 to variable “a”. On line 15, I used pre-increment. Take note of the sign. I placed two plus signs before the variable I wanted to increment. This will add 1 to the value of variable “a” and assign it back to variable “a”. Therefore, line 16 will alert 6.

If you are confused, let us take a look at the difference between pre-increment and the addition operator we looked at earlier.

Let’s consider these four sets of codes together.

Case 1:

What will be the outputs on line 13 and 14 in fig 6.4?

```
10  <script type="text/javascript">
11      var a = 5;
12      var b = a + 1;
13      console.log(a);
14      console.log(b);
15  </script>
```

Fig 6.4

Answer:

In the browser's console, we got 5 on line 13 and 6 on line 13.

```
5                                                    index.html:13
6                                                    index.html:14
```

Fig 6.5

Explanation: Variable "a" was created on line 11 and 5 was inserted into the container.

On line 12, I created another container labeled b. Then I inserted the value of a+1(5 + 1) into container b. On line 13, I displayed the value of variable a (5). On line 14, I displayed the value of variable b (6).

Case 2:

What will be the output on line 12 and line 13?

```
10  <script type="text/javascript">
11      var a = 5;
12      console.log(a+1);
13      console.log(a);
14  </script>
```

Fig 6.6

Answer:

```
6                                                    index.html:12
5                                                    index.html:13
```

Fig 6.7

In the browser's console, we got 6 on line 12 and 5 on line 13.

Explanation: I created a container and labeled it 'a' on line 11, I also assigned 5 to variable "a" on line 11.

On line 12, I added 1 to the value of variable a and displayed the result (Note: I only added 1 to the value of variable a, I did not insert it into any container, the value of variable "a" is still 5). I displayed the value of variable a on line 13.

Are you still confused? Let us consider the remaining two cases and things will become clearer.

Case 3:

What will be the output of the code below?

```
10 <script type="text/javascript">
11     var a = 5;
12     a + 1;
13     console.log(a);
14 </script>
```

Fig 6.8

Answer:

```
5                                                    index.html:13
```

Fig 6.9

Explanation: I created a container and labeled it "a" on line 11, I also assigned 5 to variable "a" on line 11. On line 12, I added 1 to the value of variable "a" but I did not display the value. I displayed the value of variable "a" on line 13 (Note: The value of variable "a" is still 5. Although line 12 added 1 to variable "a", variable "a", the result of the addition is not inserted into variable "a" since there is no equal sign on line 12).

Are you confused? Let us take a look at case 4 and you will understand better.

Case 4:

```
10  <script type="text/javascript">
11      var a = 5;
12      a = a + 1;
13      console.log(a);
14  </script>
```

Fig 6.10

Answer:

```
6                                                    index.html:13
```

Fig 6.11

Explanation:
Variable “a” was created on line 11 and 5 was inserted into the container.
On line 12, I added 1 to the value of variable “a” and inserted it back into variable “a” using the equal sign, so the value of variable “a” will no longer be 5 but now 6. (compare this with case 3 where variable “a” was not updated with the new value). On line 13, I displayed the value of variable “a” (6).

A shorter way of writing case 4 is:

```
10  <script type="text/javascript">
11      var a = 5;
12      ++a;
13      console.log(a);
14  </script>
```

Fig 6.12

The codes in fig 6.10 and the codes in fig 6.12 work the same way. Fig 6.12 is just a shorter version of fig 6.10.

```
6                                                    index.html:13
```

Fig 6.13

The ++a will do two things, firstly, it will add 1 to the value of variable “a”, secondly, it will update the value of variable “a” with the new value. This makes it different from the addition operator (which only adds a value without updating the variable with the new value).

In summary, pre-increment adds 1 to the value of a variable and update the variable with the new value.

b) Post-increment: This increases a value by 1 and update the variable with the new value. It is written by placing ++ after the variable we want to increment. It works like pre-increment.

```
10 <script type="text/javascript">
11     var a = 5;
12     a++;
13     console.log(a);
14 </script>
```

Fig 6.14

This will add 1 to the value of variable a and update the value of variable "a" with the new value (just like pre-increment).

Result:

```
6                                                    index.html:13
```

Fig 6.15

As seen above, pre-increment and post-increment seem to work the same way. Let us look at the difference between the two.

Difference between pre-increment and post-increment

Pre-increment increases the value of a variable by 1 and makes the new value available for use immediately (at that spot). Post increment increases the value of a variable by 1 and makes the new value available for use the next time the variable is called.

Consider the codes below

Case 1:

```
10 <script type="text/javascript">
11     var a = 5;
12     console.log(++a);
13     console.log(a);
14 </script>
```

Fig 6.16

Run the codes above on the browser.

Result:

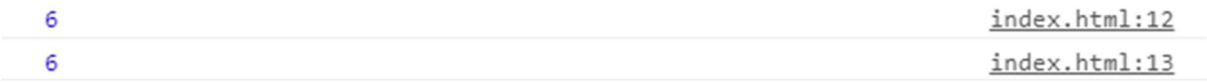

Fig 6.17

On line 12, I added 1 to the value of variable “a” using the pre-increment. I displayed the value of variable “a” on the same line (using console.log). On line 13, I displayed the value of variable “a”.

I got 6 on line 12 because the pre-increment added one to the value of variable “a” and immediately made the updated value available for use.

I got 6 on line 13 because line 12 has increased the value of variable a by 1.

Case 2:

```
10 <script type="text/javascript">
11     var a = 5;
12     console.log(a++);
13     console.log(a);
14 </script>
```

Fig 6.18

Let us run the codes above on the browser.

Result:

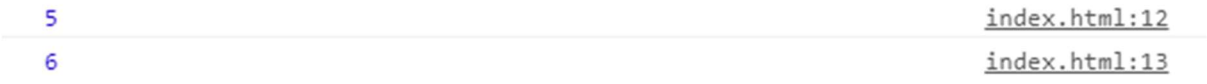

Fig 6.19

On line 12, I added 1 to the value of variable “a” using post-increment. I displayed the value of variable “a” on the same line (using console.log). On line 13, I displayed the value of variable “a”.

I got 5 on line 12 because the post-increment added 1 to the value of variable “a”, but does not make the new value available for use at that spot. The new value is made available the next time we use variable “a”. I got 6 on line 13 because line 12 added 1 to the value of variable “a” and made the new value available the next time variable “a” is used.

Problem 6.2

What will be the output of line 12 and line 13 in each of the images below?

1.

```
10  <script type="text/javascript">
11      var a = 5;
12      console.log(++a + ++a);
13      console.log(a);
14  </script>
```

Fig 6.20

2.

```
10  <script type="text/javascript">
11      var a = 5;
12      console.log(a++ + a++);
13      console.log(a);
14  </script>
```

Fig 6.21

3. .

```
10  <script type="text/javascript">
11      var a = 5;
12      console.log(a++ + ++a);
13      console.log(a);
14  </script>
```

Fig 6.22

4.

```
10 <script type="text/javascript">
11     var a = 5;
12     console.log(++a + a++);
13     console.log(a);
14 </script>
```

Fig 6.23

Solution

1. Fig 6.20 will output 13 on line 12 and output 7 on line 13.
   Explanation: Let us analyze the lines one after the other.
   Line 11: I created variable "a" on line 11 and set 5 as the value of the variable.
   Line 12: We have "++a + ++a" on this line. Let us divide this into three parts.
   i. The first ++a: This will add 1 to the value of variable "a" and make the new value available for use immediately (pre-increment). At that spot, the value of variable "a" will change from 5 to 6.
   ii. The plus sign at the center: This is to add the two ++a together. Its work is to perform addition operation.
   iii. The second ++a: This will add 1 to the value of variable "a" and make the new value available for use immediately (pre-increment). Since the first ++a has changed the value of variable "a" from 5 to 6, this will add 1 to 6 and make the new value available for use immediately. The value of variable "a" will change from 6 to 7.
   The expression on line 12 will then be interpreted as console.log(6 + 7). This is the reason we got 13 as the output of line 12.

   Line 13: We displayed the value of variable "a". Remember that the second ++a on line 12 changed the value of variable a from 6 to 7, so we will get 7 as the answer on line 13.

2. Fig 6.21 will output 11 on line 12 and output 7 on line 13.
   Explanation: Let us analyze the lines one after the other.
   Line 11: I created variable "a" on line 11 and set 5 as the value of the variable.
   Line 12: We have "a++ + a++" on this line. Let us divide this into three parts.
   i. The first a++: This will add 1 to the value of variable "a" and make the new value available for use the next time we use the variable (post-increment). Although the value of variable "a" has increased from 5 to 6, the new value is not available at that spot. The value of variable "a" at that spot is still 5.
   ii. The plus sign at the center: Its work is to perform addition operation.

iii. The second a++: This is the next time we are using variable "a" after the first a++ increased the value of variable "a" from 5 to 6, so we will have access to the updated value (6). This second a++ will also increase the value of variable "a" from 6 to 7 but the new value will not be available at that spot but will be made available the next time we use variable "a". The expression on line 12 will be interpreted as console.log(5+6) which gives 11.

Line 13: We displayed the value of variable "a". Since the second a++ on line 12 increased the value of "a" by 1, line 13 will display 7.

3. Fig 6.22 will output 12 on line 12 and output 7 on line 13.
Explanation: Let us analyze the lines one after the other.
Line 11: I created variable "a" on line 11 and set 5 as the value of the variable.
Line 12: We have "a++ + ++a" on this line. Let us divide this into three parts.
   i. a++: This will add 1 to the value of variable "a" and make the new value available for use the next time we use the variable (post-increment). Although the value of variable "a" has increased from 5 to 6, the new value is not available at that spot. This will give 5 even though the value of variable "a" has increased to 6, this is because the new value is not made available at that spot.
   ii. The plus sign at the center: Its work is to perform addition operation.
   iii. ++a: This is the next time we are using variable "a" after the a++ increased the value of variable "a" from 5 to 6, so we will have access to the updated value (6). This ++a will also increase the value of variable "a" from 6 to 7 and make the new value available at that spot (pre-increment). The expression on line 12 will be interpreted as console.log(5+7) which gives 12.

   Line 13: We displayed the value of variable "a". Since the ++a on line 12 increased the value of "a" by 1, line 13 will display 7.

4. Fig 6.23 will output 12 on line 12 and output 7 on line 13.
Explanation: Let us analyze the lines one after the other.
Line 11: I created variable "a" on line 11 and set 5 as the value of the variable.
Line 12: We have "++a + a++" on this line. Let us divide this into three parts.
   i. ++a: This will add 1 to the value of variable "a" and make the new value available for use immediately (pre-increment). At that spot, the value of variable "a" will change from 5 to 6.
   ii. The plus sign at the center: Its work is to perform addition operation
   iii. a++: This will increase the value of variable "a" from 6 to 7 but the new value will not be available at that spot but will be made available the next time we use variable "a". The expression on line 12 will be interpreted as console.log(6+6) which gives 12.

   Line 13: We displayed the value of variable "a". Since the a++ on line 12 increased the value of "a" by 1, line 13 will display 7.

Side note: We also have pre-decrement and post-decrement. --a subtracts 1 from variable "a" and make the new value available for use immediately while a— subtracts 1 and make the new value available for use the next time the variable is called.

Other assignment operators (*=, -=, /=, %=):

a. +=
To understand this, let us quickly revise what we have done.

Problem 6.3:

Step 1: Create a variable labeled num and set 12 as the value of num.

Step 2: Add 1 to the value of num.

Step 3: Change the value of variable num to a new value so that the value of num is now 13 instead of 12.

Step 4: Log the value of num to the console so that we will see 13 in the console.

Solution:

```
11 <script type="text/javascript">
12     var num = 12;
13     num++;
14     console.log(num);
15
16 </script>
```

Fig 6.24

Did you get it right?

If no, kindly go back to the pre-increment and the post-increment section.

Here is another task to do.

Problem 6.4:

Step 1: Create a variable labeled num and set 12 as the value of num.

Step 2: Add 2 to the value of num.

Step 3: Change the value of variable num to a new value so that the value of num is now 14 instead of 12.

Step 4: Send the value of num to the console so that we will see 14 in the console.

Wrong solution:

```
11 <script type="text/javascript">
12       var num = 12;
13       num+++;
14       console.log(num);
15
16 </script>
```

Fig 6.25

You might be tempted to write num+++ in order to increment num by 2. If you thought in this direction at first, congratulations, you are on your way to becoming a great programmer. That is how you should think as a programmer. Afterall, we used num++ to add 1, why can't we use num+++ to add 2 and then use num++++ to add 3?

On checking this in the browser's console, we will see an error. While ++ and -- are valid syntax, +++ is not. The increment and decrement are only meant to add or remove 1. Let us then look at a valid solution.

Solution:

```
11 <script type="text/javascript">
12       var num = 12;
13       num=num+2;
14       console.log(num);
15
16 </script>
```

Fig 6.26

On checking the browser's console we will get:

```
14                                                    index.html:14
```

Fig 6.27

Explanation:

I created and inserted 12 into variable num on line 12. On line 13, I added 2 to the value of num and set the result as the new value of num. On line 14, I log the value of num to the console.

Another way of writing he code above is this:

```
11  <script type="text/javascript">
12      var num = 12;
13      num+=2;
14      console.log(num);
15
16  </script>
```

Fig 6.28

I have made line 13 shorter.

The line 13 in figure 6.28 will work the same as line 13 in figure 6.26. When the code sees num+=2, it will pick the initial value of num, add 2 to it and set the result as the new value of num.

I call this, the shorthand assignment.

Problem 6.5:

Create a variable "a", set the value of variable "a" to 7, add 5 to the value of variable "a" and set the result as the value of variable "a". Log "a" to the browser's console so that 12 is logged to the console.

Solution:

```
11  <script type="text/javascript">
12      var a = 7;
13      a = a +5;
14      console.log(a);
15  </script>
```

Fig 6.29

Solution using the shorthand assignment:

```
11 <script type="text/javascript">
12     var a = 7;
13     a +=5;
14     console.log(a);
15 </script>
```

Fig 6.30

Line 13 in the code above will get the initial value of variable a (7), add 5 to it (to get 12), and set the result (12) as the value of variable "a".

Problem 6.6:

Create a variable b, set the value of variable b to 10, subtract 4 from the value of variable b and set the result as the value of variable b.

Log the result to the browser's console so that you will see 6 in the console.

Solution:

```
11 <script type="text/javascript">
12     var b = 10;
13     b = b - 4;
14     console.log(b);
15 </script>
```

Fig 6.31

Using the shorthand assignment:

```
11 <script type="text/javascript">
12     var b = 10;
13     b -=4;
14     console.log(b);
15 </script>
```

Fig 6.32

Line 13 will get the initial value of b, subtract 4 from the value and set the result as the new value of b.

Problem 6.7

Using the shorthand methods, solve the problems below.

1. Create a variable val, set the value of variable val to 12, multiply the variable by 4 and set the result as the value of variable val. Log the result to the browser's console so that you will see 48 in the console.
   Solution

```
11 <script type="text/javascript">
12     var val = 12;
13     val*=4;
14     console.log(val);
15 </script>
```

Fig 6.33

2. Create a variable val, set the value of variable val to 15, divide the variable by 3 and set the result as the value of variable val.
   Log the result to the browser's console so that you will see 5 in the console.
   Solution:

```
11 <script type="text/javascript">
12     var val = 15;
13     val/=3;
14     console.log(val);
15 </script>
```

Fig 6.34

3. Create a variable val, set the value of variable val to 10, divide the variable by 4 and set the result as the modulo of the result (remainder after division).
   Log the result to the browser's console so that you will see 2 in the console.

Solution:

```
11 <script type="text/javascript">
12     var val = 10;
13     val%=4;
14     console.log(val);
15 </script>
```

Fig 6.35

Now that we have looked at the assignment operators, let us move to the next operator.

CHAPTER SEVEN
JAVASCRIPT OPERATORS PART 2

String Operator (Concatenator)

Before reading this chapter, it is highly recommended that you go back to chapter four and read the string section. This will help you understand it better.

Concatenation means to weld things together.
This is used to join:

a. A string to a variable
b. A string to another string
c. A variable to another variable.

A. A string to a variable.

If I create a variable "name" as shown below.

```
<script type="text/javascript">
    var name = "Abidakun Taye";
</script>
```

Fig 7.1

How do I log a result that look like this –My name is Abidakun Taye- to the console?

I want to join "My name is " to the value of variable name and log everything to the console.

Let us do this together.

Step 1:

Since I want to log "My name is Abidakun Taye" to the console, I can go about it like this.

```
<script type="text/javascript">
    var name = "Abidakun Taye";
    console.log("My name is Abidakun Taye");
</script>
```

Fig 7.2

On checking the console, I will see this.

```
My name is Abidakun Taye                                    index.html:11
```

Fig 7.3

This is the result we want to produce.

The problem with this approach is that: var name on line 10 is useless and it is doing nohing.

If I change the value of variable “name” on line 10 to something else, line 11 will still display, “My name is Abidakun Taye”.

As shown below.

```
 9 <script type="text/javascript">
10     var name = "Kehinde Idowu";
11     console.log("My name is Abidakun Taye");
12 </script>
```

Fig 7.4

On checking the console, we will see:

```
My name is Abidakun Taye                                    index.html:11
```

Fig 7.5

This is not the result I want. Instead, I want to get (My name is (whatever the value of variable name is)).

As in the code above, I want to get “My name is Kehinde Idowu” not “My name is Abidakun Taye” since the value of variable “name” is Kehinde Idowu.

Step 2:

```
 9 <script type="text/javascript">
10     var name = "Kehinde Idowu";
11     console.log("My name is name");
12 </script>
```

Fig 7.6

On checking the console, we will see:

```
My name is name                                             index.html:11
```

Fig 7.7

Since “My name is name” is within a quote, you are instructing JavaScript that it is a string and it should not bother itself looking for any container. (Refer to chapter four if you do not understand this statement).

This isn’t what we want. Instead, we want JavaScript to treat “My name is “ as a string and add it to the value inside the container labeled “name”.

What this means is that “My name is” will be in quotes so that it can be treated as a string while the second name will not be in quotes so it can be treated as a variable.

Check it out below.
Step 3:

```
 9 <script type="text/javascript">
10     var name = "Kehinde Idowu";
11     console.log("My name is " name);
12 </script>
```

Fig 7.8

With this now, the name outside the quote will be treated as a variable and the value of variable name will be placed at that point.

Let us check the result in the console.

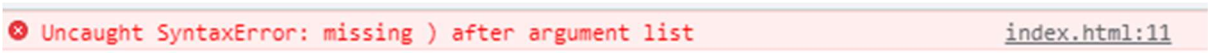

Fig 7.9

We have this error. Let us fix this error.

I will give you a principle: You don’t just jump from string to a variable.

Whenever you want to join a variable to a string, you need to weld them together.

To weld them together, you use a string operator (also known as concatenator).

The string operator in JavaScript is the plus sign (+).

Step 4:

```
 9 <script type="text/javascript">
10     var name = "Kehinde Idowu";
11     console.log("My name is " + name);
12 </script>
```

Fig 7.10

The plus sign is joining “My name is “ with the value of variable name.

On checking in console, we will see:

Fig 7.11

This is the result we want to get.

A question for you before we drop this particular question:

What is wrong with the codes below?

```
 9 <script type="text/javascript">
10     var name = "Kehinde Idowu";
11     console.log(My name is name);
12 </script>
```

Fig 7.12

Answer:

On line 11, when JavaScript sees “My”, and it isn’t in quotes, it will assume there is a container labeled “My” and it will search for it. Since there is no container labeled “My”, it will throw an error.

Well, we have a container labeled “name”, so JavaScript will not have issue with that.

Again, JavaScript will have issue with “is” since there is no container labeled “is”.

This is the reason we used the solution in step 4.

I have done mine, it’s time to do yours.

Problem 7.1

Create two variables.

Step 1. Create a variable “dept” and insert your department’s name into it.

Step 2: Create a variable “school” and let it hold the name of your institution.

Step 3: log a result that looks like this to the console (I am studying (the value of variable department) at (the value of value of variable school).

Solution:

```
<script type="text/javascript">
    var dept = "software engineering";
    var school = "SQI";
    console.log("I am studying "+ dept + " at "+ school);
</script>
```

Fig 7.13

Explanation: We put “I am studying” in quotes because we want it to be read as a string. “dept” should not be in quotes because we want JavaScript to pick the value of variable dept. We can’t jump from string to variable, so there is a need to use the plus sign to join the string “I am studying” to variable dept.

We also want JavaScript to pick “at” as a string since there is no variable with the name “at”, so we wrap it in quotes. Also, we can’t jump from variable to string, so we join them together using a string operator. Finally, we want JavaScript to pick the value of variable “school” so it must not be in quotes. Again, we can’t jump from string to variable so we join them together using a string operator.

2. Variable to variable:
A string operator can also be used to join two variables together.

Example

```
<script type="text/javascript">
    var name = "CR";
    var num = 7;
    console.log(name+num);
</script>
```

Fig 7.14

You don't just place two variables beside each other, an operator has to link them together. In the code above, the plus sign (+) is welding them together, therefore, CR7 is logged to the console.

3. String to string:

A string operator is also needed to join a string with another string as shown below.

```
 9  <script type="text/javascript">
10      console.log("Hello"+" "+"World");
11  </script>
```

Fig 7.15

In the code above, the concatenator is used to join the text "Hello" with an empty space. The empty space is then joined with the text "World" with the help of the concatenator.

Note: A string operator is also known as concatenator.

The empty space is to ensure there is a space between the two words while displaying them on the screen.

The problem with the dual functionalities of the plus sign:

Under the arithmetic operator section, we discussed that the plus sign is used to perform an addition. Under this section, we explained that the plus sign is used for concatenation.

To understand this problem better. Let us take a look at other arithmetic operators.

What will I get if I run the code below on the browser?

```
 9  <script type="text/javascript">
10      var b = "3";
11      console.log(7-b);
12  </script>
```

Fig 7.16

a. 4 b. Error. C. Undefined d. 73

Answer: 4.

Explanation: 7-3 = 4.

Let us take a closer look to address some things.

The datatype of variable b is string (it is not a number because the value of variable b is placed within quotes). On line 11, I tried subtracting a string (“3”) from a number (7), and somehow, it was successful.

The minus sign (-) only does one thing in JavaScript. Whenever JavaScript sees it, it immediately assume you are trying to subtract a value from another. When JavaScript gets to line 11, it will see console.log(7-“3”).

Then it says to itself: “This person is trying to subtract a string from a number, should I tell him/her that it isn’t possible? Oh! Let me give him/her a lifeline by going an extra mile before deciding on what to do. I will check the value within the quotes and see if it is something I can subtract from 7. Oh! It is 3, I will subtract it from 7 and give out the result.”

Let us look at another scenario.

```
9  <script type="text/javascript">
10      var b = "Taye";
11      console.log(7-b);
12  </script>
```

Fig 7.17

When we run this on the browser, we will get:

```
NaN                                                    index.html:11
```

Fig 7.18

NaN stands for “Not a Number”.

Explanation:

On line 11, JavaScript will say to itself

“This person is trying to subtract a string from a number, should I tell him/her that it isn’t possible? Oh! Let me give him/her a lifeline by going an extra mile before deciding on what to do. I will check the value within the quotes and see if it is something I can subtract from 7. Oh! It is a text, There is no way I can subtract that from 7. Let me inform the programmer that what he/she is trying to subtract from 7 is not a number.”

One more scenario before we leave this part:

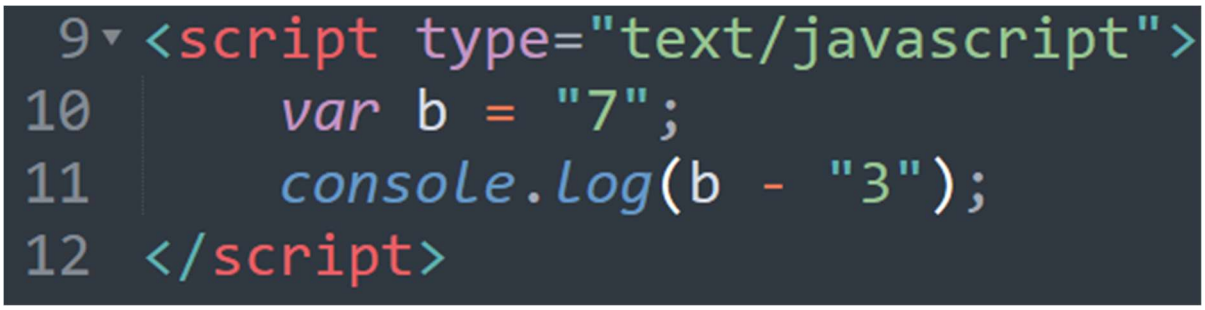

Fig 7.19

The code above will log 4 to the console.

Explanation:

JavaScript says to itself: “This person is trying to subtract two strings, should I tell him/her that it isn’t possible? Wait a minute, let me check the values within the quotes and see if I can subtract them. Oh! What a clown! You placed two numbers in quotes and think I won’t know? You think I won’t be able to subtract them… Ah Ah ah. I got you. I will subtract them successfully”.

I hope this is clear.

Other arithmetic operators will behave this way except the plus sign (+). This is because the plus sign performs two functions in JavaScript (addition and concatenation).

The plus sign (+):

When it sees numbers, it performs addition, when it sees a string, it performs concatenation.

Example:

First case (Two numbers):

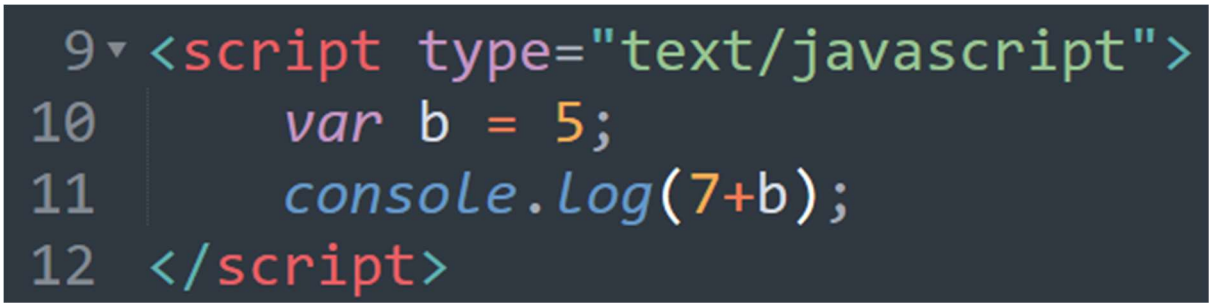

Fig 7.20

12 will be logged to the console. 7 +5 = 12. The two operands belong to the datatype of Number. When the plus sign is placed between two numbers, it performs addition.

Second case (A number and a string):

```
<script type="text/javascript">
    var b = "Taye";
    console.log(7+b);
</script>
```

Fig 7.21

When the plus sign is placed between a number and a string, it concatenates them together, so we have this result.

```
7Taye                                                    index.html:11
```

Fig 7.22

Another example:

```
<script type="text/javascript">
    var b = "3";
    console.log(7+b);
</script>
```

Fig 7.23

Since the datatype of variable b is string, it concatenates 3 with 7 so we get this result on the browser:

```
73                                                       index.html:11
```

Fig 7.24

Third case (two stings):

The plus sign concatenates two strings together as shown below

```
 9  <script type="text/javascript">
10      var b = "Taye";
11      console.log("Abidakun "+b);
12  </script>
```

Fig 7.25

Result:

```
Abidakun Taye                                                    index.html:11
```

Fig 7.26

Another Example:

```
 9  <script type="text/javascript">
10      var b = "5";
11      console.log("7"+b);
12  </script>
```

Fig 7.27

Result:

```
75                                                               index.html:11
```

Fig 7.28

Problem 7.2

What will I get if I run the codes below on the browser?

```
 9  <script type="text/javascript">
10      var b = "James";
11      console.log(b* 5);
12  </script>
```

Fig 7.29

Answer: NaN

Remember I said every other arithmetic operator would behave like the minus sign since they only perform one function.

Problem 7.3

What will be the result of the code below?

```
<script type="text/javascript">
    var b = "10";
    console.log(b* 5);
</script>
```

Fig 7.30

Answer: 50.

Again, it will behave like the minus sign.

Problem 7.4 What will be the output of the codes below?

```
<body>
    <input type="number" value="5" id="myInp">
    <script type="text/javascript">
        console.log(myInp.value * 3);
    </script>
</body>
```

Fig 7.31

15. correct. The codes above will log 15 to the console.

Explanation: On line 9, we got the value of “myInp” (5) and multiplied it by 3, the result was logged to the console.

Problem 7.5 What will be the output of the codes below?

```
6  <body>
7      <input type="number" value="5" id="myInp">
8      <script type="text/javascript">
9          console.log(myInp.value + 3);
10     </script>
11 </body>
```

Fig 7.32

8? Wrong. Try again.

53? Correct.

Explanation: On line 9, I got the value of myInp (5) and added 3 to it using the plus sign.

It is important to know that JavaScript will always treat any value gotten using any of these three commands (value, innerText, innerHTML) as a string.

The myInp.value on line 9 will return 5, JavaScript treats the returned 5 as a string, not as a number even though I set the type attribute of the input to number on line 7.

Let's verify this together.

```
6  <body>
7      <input type="number" value="5" id="myInp">
8      <script type="text/javascript">
9          // console.log(myInp.value + 3);
10         console.log(typeof(myInp.value));
11     </script>
12 </body>
```

Fig 7.33

Here is the output

```
string                                                    index.html:10
```

Fig 7.34

It returns a string as shown above. It then concatenates string 5 to 3. Which gives 53.

Before we leave the string operator, let us look at one important thing.

Point of concatenation

Problem 7.6

What will be the result of the code below.

```
 9 <script type="text/javascript">
10      var b = "Taye";
11      console.log(7 + 5 + b);
12 </script>
```

Fig 7.35

Answer: 12Taye

Explanation: JavaScript reads codes from left to right. On line 11, it will first add 7 and 5 together, then concatenate the result with the string “Taye”.

Problem 7.7

What will be the result of the code below?

```
 9 <script type="text/javascript">
10      var b = "3";
11      console.log(7 + 5 + b);
12 </script>
```

Fig 7.36

Answer: 123

Explanation: JavaScript reads codes from left to right. On line 11, it will first add 7 and 5 together, then concatenate the result with the string “3”.

Problem 7.8

What will be the result of the codes below?

```
<script type="text/javascript">
    var b = "Taye";
    console.log(7 + 5 + b + 1 + 2);
</script>
```

Fig 7.37

Answer: 12Taye12

Explanation: JavaScript reads codes from left to right. On line 11, it will first add 7 and 5 together, then concatenate the result with the string “Taye”. When you concatenate 12 with Taye, you get 12Taye.

The next question is: should we add or concatenate 12Taye with 1?

Although 1 is a number but 12Taye is not. So, JavaScript will concatenate 12Taye with 1 and you get 12Taye1.

Remember, the plus sign only adds when both operands are of type number.

Again, should we concatenate 12Taye1 with 2? Although 2 is a number, 12Taye1 is not, so we get 12Taye12.

Important to note

It is important to note that whenever JavaScript concatenates a string with anything, it treats the result as a string.

This is the reason JavaScript will concatenate 12Taye with 1 instead of adding them because it sees 12Taye as a string and not number. Immediately we concatenate 12 with Taye, the result (12Taye) is a string. This is also the reason JavaScript will concatenate 12Taye1 with 2 because it sees 12Taye1 as a string.

Again, whenever JavaScript concatenates a string with anything, it treats the result as a string.

Problem 7.9

What will be the output of the codes below?

```
<script type="text/javascript">
    var b = "3";
    console.log(7 + 5 + b + 1 + 2);
</script>
```

Fig 7.38

Answer: 12312

Explanation: Variable b is of type string. Apply the logic in the last question to this.

CHAPTER EIGHT

JAVASCRIPT EVENTS AND FUNCTIONS

This is the part I love most in programming. To me, it is the most the interesting part of programming. Read, enjoy and practice.

Events

Events are occurrences, the happenings at a particular time. With events, you know the activities of your users at a particular time. There are two broad classes of events. They are “mouse events” and “key events”.

Mouse events are those activities you perform using the mouse. The activities we perform with mouse include: click, double click, right click and many more which will be considered later. Key events are those activities you perform using the keyboard.

Mouse Events

Please code along and practice on your PC. It is very important.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title></title>
5  </head>
6  <body>
7      <button>display 7</button>
8  </body>
9  </html>
```

Fig 8.1

In the code above, we placed a text “display 7” in a button. At the moment, nothing happens when we click on the button. The next thing we want to do is to add an **event** to the button. We want it to alert 7 whenever users click on the button. Events can be added same way we add attributes to an HTML tag, as shown below.

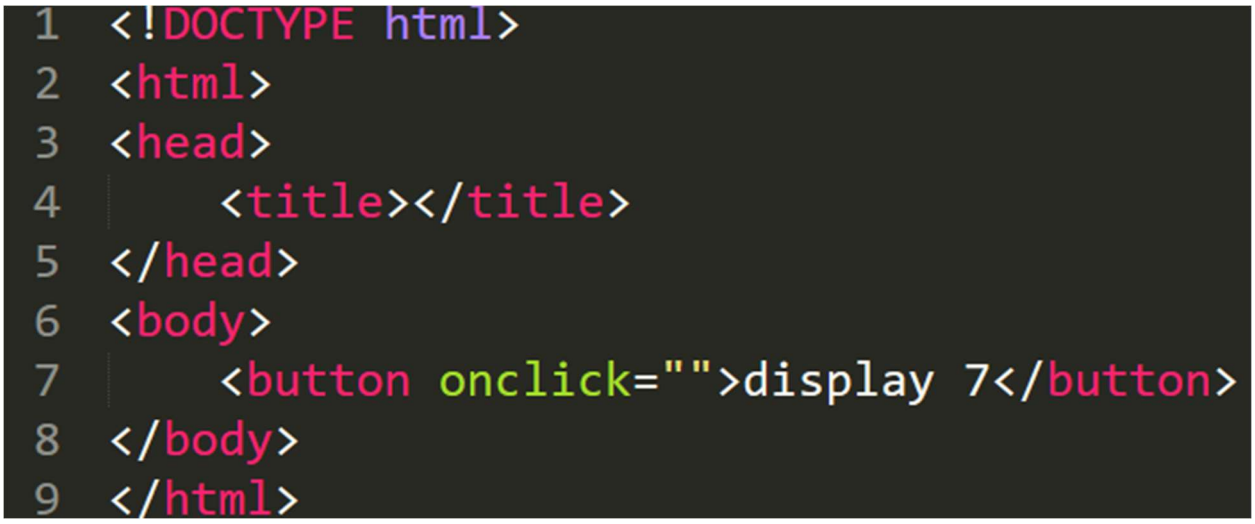

Fig 8.2

When events are added like HTML attributes, they start with “on”. The “onclick” above is an event. Whatever you want to do whenever users click on the button, is placed as the value of the event, as shown below.

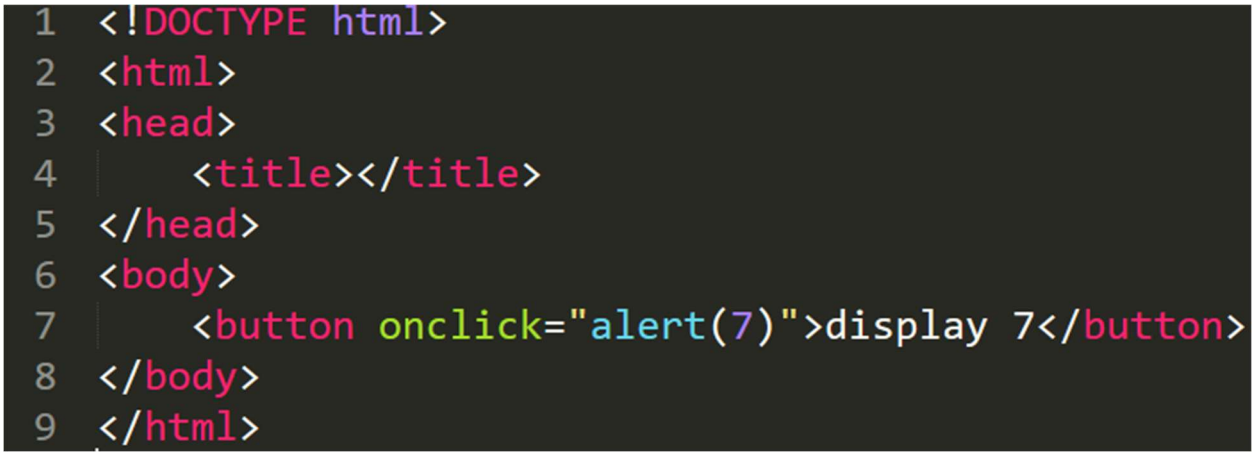

Fig 8.3

Run the codes on the browser. It will alert 7 each time you click on the button.

Another mouse event is doubleclick.

Fig 8.4

Again, when events are added like HTML attributes, they start with “on”. The dblclick above stands for double-click. Whenever you double click on the button above, “hi” will be logged to the browser’s console.

Key events: These are events you perform using your keyboards

Fig 8.5

In the code above, we placed a text (display hello) as the value of an input. The next thing we want to do is to add an **event** to the input. We will log “hello” to the console as users type in the input.

Fig 8.6

Run the code on the browser, open your browser’s console, move your cursor to the input so you can edit the value of the input. Start typing. You will observe that “hello” is logged to your browser’s console as you type.

The “keydown” event is triggered as you type within the input. Whenever you press any key on your keyboard, the key goes down, therefore the keydown event is triggered and “hello” is logged to the console.

The “keyup” works almost the same way as the “keydown” event but there is a little difference. Consider the codes below:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <input onkeyup="console.log('hello')" value="display hello">
</body>
</html>
```

Fig 8.7

Run it on the browser, open your browser's console and edit the input like you did earlier. You will get the same result just like the "keydown" event. What then is the difference between the "keyup" and the "keydown" events?

The keydown event is triggered when any key on your keyboard is pressed while the keyup event is triggered when the pressed key is released. In few seconds time, you will understand this better.

Go back to fig 8.7, run it on the browser, open your console and start typing within the input just like before. The next thing you will do is to hold any key on your keyboard for 5 seconds (Press without releasing it), you will observe that "hello" is not logged to the console throughout the period you held the key. "hello" is only logged to the console when the key is released. This is because the "keyup" event is not triggered when the key is down but only triggered when a pressed key goes back up. Now change the "onkeyup" in the code above to "onkeydown", run it on the browser, start typing some text within the inputs and hold a key for 5 seconds, like you did earlier. Do you notice that "hello" is logged to the console throughout the period the key was held? As long as a key is held down, the "keydown" event is triggered.

Here is a link to other JavaScript events. Check them out and practice them at your leisure.

https://www.tutorialspoint.com/JavaScript/JavaScript_events.htm

INLINE, INTERNAL AND EXTERNAL JAVASCRIPT

There are three ways to write JavaScript codes

They are: inline JavaScript, internal JavaScript, external JavaScript.

I will explain inline and internal JavaScript now while external JavaScript will be discussed at the end of this chapter.

INLINE JAVASCRIPT

Inline JavaScript: This is when JavaScript codes are written directly within HTML codes.

Example:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <button onclick="alert(4)">click me</button>
</body>
</html>
```

Fig 8.8

The codes above alerts 4 whenever the button is clicked.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <input onkeyup="console.log(7)">
</body>
</html>
```

Fig 8.9

The codes above logs 7 to the console as users type within the input.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <button ondblclick="alert('Hello')">click twice</button>
</body>
</html>
```

Fig 8.10

The codes above alerts “Hello” when users double click on the button.

The problem with inline JavaScript is that: there is no reusability of codes. This brings us to internal JavaScript.

INTERNAL JAVASCRIPT

Consider the code before internal JavaScript is introduced.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <button onclick="alert(6 * 3)">First</button>
    <button onclick="alert(6 * 3)">Second</button>
    <button onclick="alert(6 * 3)">Third</button>
</body>
</html>
```

Fig 8.11

18 will be displayed on the browser onclick of any of the three buttons. If I change my mind to display 6 (4+2) instead of 18 (6*3) onclick of any the three buttons, I need to change it at three different places as shown below.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <button onclick="alert(4 + 2)">First</button>
    <button onclick="alert(4 + 2)">Second</button>
    <button onclick="alert(4 + 2)">Third</button>
</body>
</html>
```

Fig 8.12

If I forgot to change one of the three buttons, two of the buttons will display 6 while the third will display 18. Here is the code below:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <button onclick="alert(4 + 2)">First</button>
    <button onclick="alert(4 + 2)">Second</button>
    <button onclick="alert(6 * 3)">Third</button>
</body>
</html>
```

Fig 8.13

This problem can be solved using internal JavaScript and functions. This allows reusability of codes.

Internal JavaScript: This is when JavaScript codes are written within the script tag of a HTML file.

To use internal JavaScript, open a script tag within the HTML file as shown below

Fig 8.14

Place all JavaScript codes within the script tag. Do you observe we have been doing this? We have been using internal JavaScript all along. I will add functions to the code above.

This brings us to the question “What is a function?”.

A function is a set of codes that perform a task. The beauty of functions is that they can be reused.

Let’s rewrite the codes in fig 8.13 using internal JavaScript and function.

Fig 8.15

At the moment, we aren't doing anything when users click on any of the three buttons. We opened a script tag. Our function will be created within the script tag. Let's create the function together.

Procedure to create a function.

1. Write the function keyword as shown below. Immediately the browser sees the function keyword, it knows you are about creating a function.

Fig 8.16

2. The next thing is to give your function a name. The rules for naming a function are also the same as that of variable declaration (Chapter two). I will name my function "myFunc".

Fig 8.17

3. The next thing is to add a parenthesis after your function name as shown below.

Fig 8.18

4. You place a curly braces after the parenthesis as shown below

Fig 8.19

5. Lastly, you put the task you want to execute within the curly braces. This is known as the body of the function.

In my own case, I want the function to alert 18 (6*3).

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <button onclick="">First</button>
    <button onclick="">Second</button>
    <button onclick="">Third</button>
    <script type="text/javascript">
        function myFunc(){
            alert(6 * 3);
        }
    </script>
</body>
</html>
```

Fig 8.20

I have successfully created a function that alerts 18 (6*3). Next thing is to run the codes above on the browser, then click on any of the three buttons. Well, you will see nothing. It will not alert 18. This is because I am yet to connect the function with the click event of the buttons.

To connect them together, I will call on the function when any of the buttons is clicked.

When you call a function, the codes within the curly braces of the function will be executed.

To call a function is very simple, just call the name of the function and place a parenthesis right after it. Since our function's name is myFunc, this is how we will call on the function: myFunc().

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <button onclick="myFunc()">First</button>
    <button onclick="myFunc()">Second</button>
    <button onclick="myFunc()">Third</button>
    <script type="text/javascript">
        function myFunc(){
            alert(6 * 3);
        }
    </script>
</body>
</html>
```

Fig 8.21

Run the codes above on the browser and click on any of the three buttons, you should see 18 on the browser.

Let's assume we want to display 6, that is, the result of 4+2 instead of 18 (the result of 6 * 3) onclick of any of the three buttons, we only need to change it within the body of the function as shown below.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <button onclick="myFunc()">First</button>
    <button onclick="myFunc()">Second</button>
    <button onclick="myFunc()">Third</button>
    <script type="text/javascript">
        function myFunc(){
            alert(4 + 2);
        }
    </script>
</body>
</html>
```

Fig 8.22

We don't need to change it at three places as in the case of inline JavaScript. Henceforth, we will use internal JavaScript and functions and not inline JavaScript that we used initially.

Note: "To trigger a function" is synonymous to "To call a function" and they will be used interchangeably in this book.

A quick one: What will the user see on the browser when he/she clicks on the button below?

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <button onclick="">call me</button>
    <script type="text/javascript">
        function sayHi(){
            alert("Hi");
        }
    </script>
</body>
</html>
```

Fig 8.23

a. “Hi” will be displayed on the browser
b. Nothing will be displayed on the browser.

Answer: Nothing will be displayed on the browser.

Explanation: Although I added a click event to the button, I didn’t connect the event with the function. Since the codes in the body of a function is not executed until the function is triggered, nothing is displayed on the browser.

Problem 8.1: Connect the function with the event such that when a user clicks on the button, “Hi” will be displayed on the browser.

Solution:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <button onclick="sayHi()">call me</button>
    <script type="text/javascript">
        function sayHi(){
            alert("Hi");
        }
    </script>
</body>
</html>
```

Fig 8.24

Explanation: To trigger a function, write the name of the function and place a parenthesis after the name. With the code above, when a user clicks on the button, the block of codes within function sayHi will be executed.

Problem 8.2

Create a button in your HTML codes, create a function called “displayName”. When a user clicks on the button, trigger the function such that the body of the function will log “Taye Abidakun” to the console.

Solution:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <button onclick="displayName()">See my name</button>
    <script type="text/javascript">
        function displayName(){
            console.log("Taye Abidakun");
        }
    </script>
</body>
</html>
```

Fig 8.25

Another task to do

Problem 8.3

Create a HTML input of type number, give it an id. Create a button. Create a function such that whenever a user clicks on the button, multiply the number typed in the input by 2 and log the result to the console.

Solution:

```
<body>
    <input type="number" id="inp">
    <button onclick="multiplyByTwo()">multiply</button>
    <script type="text/javascript">
        function multiplyByTwo(){
            var num = inp.value * 2;
            console.log(num);
        }
    </script>
</body>
```

Fig 8.26

Explanation: We created an input of type number on line 7 and gave it an id of “inp”. On line 8, a button was created with a click event that triggers the function “multiplyByTwo” when clicked.

Line 11 of function “multiplyByTwo” will get the value of the inp, multiply it by two and set the result as the value of variable num. Line 12 will log the result to the browser’s console.

Problem 8.4:

Create two inputs. As users type a number in the first input, the half of the number will be displayed in the second input.

Solution:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title></title>
5  </head>
6  <body>
7      <input type="number" id="first" onkeyup="divideByTwo()">
8      <input type="number" id="second">
9      <script type="text/javascript">
10         function divideByTwo(){
11             var myNum = first.value / 2;
12             second.value = myNum;
13         }
14     </script>
15 </body>
16 </html>
```

Fig 8.27

Explanation: The keyup event was added to the first input so that the function will be called as the user types in the first input. Therefore, function divideByTwo is triggered as the user types in the first input.

Now to the body of function.

On line 11, we got the value of the first input and divide it by two, the result is saved in variable myNum.

On line 12, myNum is set as the value of second so that the result of line 11 is displayed in the second input.

Quickly answer this question to see how much you understand what I explained before we move to the next thing.

What will I see on the browser if I run the code in fig 8.28?

a. It will alert “hello”

b. Nothing will be displayed on the browser.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <script type="text/javascript">
        function myFunc(){
            alert('hello');
        }

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

Fig 8.28

Answer: Nothing will be displayed on the browser.

Explanation: While the function below is meant to alert 'hello', nothing is calling/triggering the function. You need to trigger/call on a function before the codes in the body of the function is executed.

Take home: To execute the codes in the body of a function, you need to trigger the function.

Functions like function divideByTwo in fig 8.27 are known as event handlers since you trigger them when users perform an event.

Not all functions are event handlers because functions can be called without using events. Let's look at this together.

Calling function outside events:

So far, we have triggered functions only with events. We have used events such as keyup, click, dblclick etc. We need to be aware that functions can also be triggered without events.

Recall that to trigger a function using an event, you follow the steps below:

1. You write the event like an attribute to the HTML tag. Example: <button onclick></button>
2. You write the function's name as the value of the event. Example: <button onclick='myFunc'></button>
3. Finally, you place a parenthesis after the function's name. Example: <button onclick='myFunc()'></button>

To trigger a function without using an event, just write the function's name and place a parenthesis after it.

Consider the block of codes below

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title></title>
5  </head>
6  <body>
7      <script type="text/javascript">
8          function myFunc(){
9              alert('hello');
10         }
11         myFunc();
12
13     </script>
14 </body>
15 </html>
```

Fig 8.29

The codes above will alert 'hello' on the browser. This is because we triggered function myFunc on line 11.

The only difference between the block of codes above and fig 8.28 is line 11. While the codes in fig 8.28 displayed nothing, the code above displayed hello.

To call on a function without an event is simple, just write the function's name and place a parenthesis after the function. Instantly, the codes in the body of the function will be executed.

Problem 8.5: What will be the output of the codes below when the button is clicked?

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title></title>
5  </head>
6  <body>
7      <button onclick="secondFunc()">click me</button>
8      <script type="text/javascript">
9          function myFunc(){
10             console.log('hello');
11         }
12         function secondFunc() {
13             console.log('hi');
14             myFunc();
15         }
16     </script>
17 </body>
18 </html>
```

Fig 8.30

a. It will log hello to the console, then log hi to the console
b. It will log hi to the console, then log hello
c. Nothing will be logged to the console
d. Only hi will be logged to the console
e. Only hello will be logged to the console

Answer: b. It will log hi to the console, then log hello

Explanation: When the button is clicked, the secondFunc function will be triggered, therefore the codes in the body of secondFunc function will be executed.

In the body of secondFunc function, line 13 will be executed, therefore it will log hi to the console. Then it will move to line 14. myFunc function is triggered on line 14, therefore the codes in the body of function myFunc will be executed and it will log hello to the console.

If you are confused, don't disturb your brain. Let's do justice to this part with one more example.
Consider the codes below.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title></title>
5  </head>
6  <body>
7      <button onclick="secondFunc()">click me</button>
8      <script type="text/javascript">
9          function myFunc(){
10             console.log('hello');
11         }
12         function secondFunc() {
13             console.log('hi');
14             myFunc();
15             console.log('good');
16         }
17     </script>
18 </body>
19 </html>
```

Fig 8.31

Let's explain what happens when the button is clicked. You will agree with me that the secondFunc is triggered when the button on line 7 is clicked. Then the codes in the body of secondFunc is executed. Note that function secondFunc starts from line 12 and ends on line 16, meaning that everything between the two lines will be executed.

Let's take a look at the body of secondFunc.

On line 13, "hi" will be logged to the console. When it is done with line 13, it moves to line 14.

On line 14, myFunc is triggered meaning that the codes in the body of myFunc will be executed. Note that we aren't done with function secondFunc.

Understand what is going on: An event triggered secondFunc function, while executing it, it realized that another function (myFunc) was triggered. What should it do? Should it continue with the execution of secondFunc function or should it move to myFunc function?

A short story

You are rushing home to bake a cake, on your way home, your son called that he has used your flour to make chin chin so you need to get flour from the market. Will you continue to rush home? Of course not. You branch to the market, get the flour, then continue with your initial journey (to head home). This is what happens here.

An event triggered secondFunc. While executing it, it realized the need to execute another function(myFunc). It stops executing secondFunc at that instant and move to myFunc. When it is done with myFunc, it goes back to secondFunc to complete it.

With this explanation, myFunc will be executed, therefore "hello" will be logged to the console. Since that is all we have within myFunc function, it will go back to secondFunc function and continue with it. It will then log "good" to the console.

In summary, in the browser's console, "hi" will first be logged to the browser's console, then it will log "hello" and finally logs "good".

Take home: If function B is called within the body of function A, then, when function A is triggered (through whatever means) ,it executes the codes in the body of function A till it gets to the point where function B is called, moves to function B, executes the codes within function B. When it is done, it goes back to complete function A.

Local and global variable

Take home from this section: A local variable belongs to a particular function while a global variable does not belong to any function.

Variables can either be local variables or global variables. I want you to see local variables as private properties while global variables are public/ government properties. This phrase by President Buhari "I belong to everybody, I belong to nobody" is very true of global variables.

To know if a variable is a local variable or a global variable, just ask yourself "Where was the variable created?"

If it is created within a function, it is a local variable. If it is created outside a function, it is a global variable. Remember that we create variables using the 'var' keyword.

Local variables:

- Local variables are created within functions
- Local variables cannot be shared with other functions

```
<script type="text/javascript">
    function myFunc(){
        var a = 3;
    }
    function secondFunc() {

    }
</script>
```

Fig 8.32

In the code above, variable “a” was created within myFunc, therefore it is a local variable of myFunc. The value of variable “a” cannot be used within secondFunc function because variable “a” is a local variable of myFunc.

```
<body>
    <button onclick="firstFunc()">first</button>
    <button onclick="secondFunc()">second</button>
    <script type="text/javascript">
        function firstFunc(){
            var a = 3;
            alert(a);
        }
        function secondFunc() {
            alert(a);
        }
    </script>
</body>
```

Fig 8.33

3 will be displayed on the browser when the first button is clicked. An error will be thrown in the browser’s console when the second button is clicked.

Explanation: When the first button is clicked, firstFunc is triggered. Variable “a” is a local variable to firstFunc, therefore it can be used within firstFunc. When the second button is clicked, secondFunc is triggered. An error will be thrown because secondFunc does not have access to variable “a” and it isn’t allowed to use “a” because “a” is a local variable to another function. The error thrown in the browser’s console-a is not defined- explains what we did wrong. It shows that secondFunc doesn’t recognize “a”.

Global variables

Global variables are not created within functions. They are created outside functions.

Example

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <script type="text/javascript">
        var num = 4;
        function myFunc(){

        }
        function secondFunc() {

        }

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

Fig 8.34

In the codes above, variable num is a global variable because it wasn't created within any function. Global variables can be used by all functions.

```
<body>
    <button onclick="firstFunc()">first</button>
    <button onclick="secondFunc()">second</button>
    <script type="text/javascript">
        var num = 4;
        function firstFunc(){
            alert(num * 2);

        }
        function secondFunc() {
            alert(num);
        }

    </script>
</body>
```

Fig 8.35

When the first button is clicked, 8 is displayed on the browser. When the second button is clicked, 4 is displayed on the browser.
Explanation: When the first button is clicked, firstFunc is triggered. Within the body of firstFunc, we multiplied the value of num by 2 and alert the result, therefore it will alert 8.

When the second button is clicked, secondFunc is triggered and it alerts the value of variable num which is 4.

Note that both functions (firstFunc and secondFunc) have access to variable num because it is a global variable. It wasn't created within any function.

How browsers read functions and variables

Browsers read codes from up to bottom, then it reads each line from left to right. When browsers see functions, browsers only store the names of the functions in memory but they ignore the body of the functions. The body of the functions is only read when the function is triggered.

Based on the explanation above, what will be the output of the codes below when the button is clicked?

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title></title>
5  </head>
6  <body>
7      <button onclick="displaySchool()">school</button>
8      <script type="text/javascript">
9          function displaySchool(){
10             alert(mySchool);
11         }
12         var mySchool = "SQI";
13     </script>
14 </body>
15 </html>
```

Fig 8.36

a. SQI will be displayed on the browser
b. Nothing will be displayed
c. An error will be thrown in the console

Answer: SQI will be displayed on the browser.

Explanation: When we load the codes above on the browser for the first time (before the user performs any action), this happens:

The browser will read the codes above line by line. It will read line 1, then 2 till it gets to 7. On line 7, it will read the button tag. The displaySchool function will **not** be triggered because the user hasn't performed any action (Remember that events are occurrences). Then, it will read line 8, then line 9. It will realize that we have a function on line 9 therefore it will only store the function's name -displaySchool- in memory, then it will skip the body of the function(since the function is yet to be triggered). Then it will jump to line 12 and store variable mySchool and its value in memory. After this, it will proceed to read the other parts of the code (till the end on line 15).

Again, what we have above is what happens immediately we load the codes on the browser (before the user performs any action).
When the user clicks on the button, function displaySchool will be triggered and the codes in the body of the function will be executed. It will alert mySchool (remember that variable mySchool and its value has been stored in memory when we run the codes on the browser the very first time).

According to what we have above, placing variable mySchool before the function (similar to fig 8.35) or placing it after the function (as it is in fig 8.36) produced the same result,

but I will advice that you place your global variables before your functions and not after. This makes you write neater codes.

Don't skip this part

A key difference between local and global variable

This part is a bit tricky but you will enjoy it. Put your mind in it as it is very important.

What will be the output of the codes below when the button is clicked?

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title></title>
5  </head>
6  <body>
7      <button onclick="showNum()">show number</button>
8      <script type="text/javascript">
9          function showNum(){
10             var num = 5;
11             num += 3;
12             console.log(num);
13         }
14     </script>
15 </body>
16 </html>
```

Fig 8.37

a. 8
b. 5
c. 3

Answer: 8. showNum is triggered when the button is clicked. It means the block of codes within the body of the function is executed. It will execute them line by line.

On line 10, we created variable num and inserted 5 in it (note that variable num is a local variable).

On line 11, 3 is added to the value of num, therefore the value of num changes from 5 to 8 (Check chapter six if you don't get this).

We finally log the value of num to the console on line 12.

Note this: If we click on the button again (the second time), then the process is repeated and we see 8 again. Same thing happens when the button is clicked the third time. The process is repeated even if the button is clicked a million times. “This is obvious”, you may say. Well, not really. You will understand my reasons soon.

What I’m about to do maybe boring to you. Please don’t skip it.

The codes on line 10 (fig 8.37) is my main concern here.

Permit me to explain again what happens each time you click on that button and state the reason you have the same result (8).
When you click on the button (the first time), showNum is triggered and the codes within it is executed. On line 10, it creates a variable num (remember that when the browser sees the var keyword, a new container is created) and sets 5 as its value.

Line 11 adds 3 to it and line 12 logs it to the console.

When you click on the button again (the second time), showNum is triggered again and the codes within it is executed. Again, it sees the codes on line 10 and creates another variable num (recall that variable num was created when you clicked on the button the first time). It therefore means that we have two variables bearing the same name (num). The new one overrides the first.

On line 11, it adds 3 to the value of variable num and line 12 logs it to the console.
It is important to note that the variable num that was created when the button was clicked the first time isn’t the same variable num that was used the second time. They are two different containers.

If you click on the button 10 times, it means 10 variable num will be created.

This is because, each time you click on the button, showNum is triggered and the block of codes within it is executed. It sees line 10 each time. Line 10 reads “var num = 5”.

Due to the presence of the var keyword, it creates a new container (num) each time it reads line 10.

Global variable doesn’t behave this way. Let’s look at it together. Consider the codes below.

Fig 8.38

We have converted variable num from local variable to a global variable. When you click on the button the first time, 8 will be logged to the console. When you click on the button the second time, 11 will be logged to the console. When you click on it the third time, 14 will be logged to the console. 3 is added to the value of num each time the button is clicked.

Explanation: When we load the codes on the browser the first time, the browser reads the codes from top to bottom. It reads from line 1 to 7. On line 7, it sees the button, but it won't trigger the showNum function because the click event hasn't been called upon as at the time. It moves to line 8, then to line 9. On line 9, it creates a new container called num and sets 5 as its value. Then it proceeds to line 10. On line 10, it stores the function's name and ignores the body of the function.

When the user clicks on the button the first time, it triggers showNum and executes the codes in the body of the function. It sees line 11.

Note that there is no var keyword on line 11. Therefore, no new variable is created. Instead, it makes reference to the already created variable num and adds 3 to its value. Line 12 logs the value to the console (8).

The value of variable num is no longer 5 but 8. When the user clicks on the button the second time, showNum is triggered again and the codes in the body of the function is executed again.

It sees line 11 again. Note, there is no var keyword on line 11, therefore, no new container is created, instead, JavaScript uses the value in the already create container (num). It adds 3 to the value of variable num (8) and assigns it back to "num". Therefore, the value of variable num changes to 11. This happens each time you click on the button.

Variable num is only created once (when you load the codes on the browser the very first time). Each click of the button changes the value of variable num (but doesn't create a new container unlike the case of local variables).

Take note: Global variable are only recreated whenever you refresh the browser while local variables are recreated each time you trigger its function.

Problem 8.6:

```
 6  <body>
 7      <button onclick="firstFunc()">First</button>
 8      <button onclick="secondFunc()">Second</button>
 9      <script type="text/javascript">
10          function firstFunc(){
11              var num = 1;
12              num++;
13              console.log(num);
14          }
15          function secondFunc() {
16              console.log(num);
17          }
18      </script>
19  </body>
```

Fig 8.39

Q1. What will I see on the browser's console if the first button is clicked the first time?

a. 1
b. 2
c. 3
d. Error

Answer: 2

Q2. What will I see on the browser's console if the second button is clicked the first time?

a. 1
b. 2
c. 3

d. Error

Answer: Error

Q3. What will I see on the browser's console if the first button is clicked the second time?

a. 1
b. 2
c. 3
d. Error

Answer: 2

Q4. What will I see on the browser's console if the second button is clicked the second time?

a. 1
b. 2
c. 3
d. Error

Answer: Error

Explanation: The second button will always throw an error each time it is clicked because variable num is a local variable to the first function therefore, it cannot be accessed by the second function. The first button will always display 2 each time it is clicked because variable num is a local variable and will be recreated each time the button is clicked.

Let's dive into politics a bit.

Story time.

A state has 100 billion naira in the state's treasury. When governor Tunde was elected, he spent 40 billion naira on infrastructure, remaining 60 billion naira. Governor Tunde lost his reelection and governor James was elected. Governor James met 60 billion naira and embezzled 30 billion naira, remaining 30 billion naira. The people complained and wished governor Tunde was brought back. Governor James lost his reelection and governor Tunde was reelected. Governor Tunde met 30 billion naira. He improved the state's IGR and generated 40 billion naira into the state's account. The state now has 70 billion naira. He finished his term and governor Aina was elected. Governor Aina met 70 billion naira in the state's treasury.

This is the end of my story.

State's fund = global variable.

Each governor = A function.
When a governor is elected = When a function is triggered.

When a function changes the value of a global variable, the next function to use the global variable will see the new value not the previous value.

Take a look at the codes below

```
<body>
    <button onclick="firstFunc()">First</button>
    <button onclick="secondFunc()">Second</button>
    <script type="text/javascript">
        var num = 1;
        function firstFunc(){
            num++;
            console.log(num);
        }
        function secondFunc() {
            console.log(num);
        }
    </script>
</body>
```

Fig 8.40

In the code above, variable num is a global variable meaning that the two functions above can access the value of the variable.

Example: If I run the code on the browser and then click on the **second button**, 1 will be logged to the console.

Explanation: The first function is the one increasing the value of variable num by 1, since firstFunc isn't triggered (The first button triggers firstFunc), the value of variable num will remain 1. When I click on the second button, secondFunc will be triggered and the codes in the body of the function will be executed. secondFunc will log the value of variable num to the console.

Example: If I click on the second button, then click on the first button, then click on the second button, the console will look like this:

```
1                                    index.html:16
2                                    index.html:13
2                                    index.html:16
```

Fig 8.41

Explanation:

Step 1: When I clicked on the second button, the value of variable num (1) will be logged to the console.

Step 2: When I clicked on the first button, firstFunc will be triggered. firstFunc adds 1 to the value of variable num. Therefore, the value of variable num will change to 2. The value of variable num (2) is logged to the console.

Step 3: When I click on the second button again. It logs the value of num (2) to the console. We added 1 to the value of num in step 2. Do you remember our politics story?

When a governor changes the value of funds in the state's treasury, the next elected governor sees the new value, not the old value.

When a function changes the value of a global variable, the next function to use the global variable sees the new value, not the old value.

Play with the codes above. Click on the second button, then first button, then second button, then first button as much as you like. Check the result in the browser's console. Enjoy.

Last thing before we leave this leave local and global variable.

What happens when local variables and global variables bear the same name?

```
<body>
    <button onclick="firstFunc()">First</button>
    <button onclick="secondFunc()">Second</button>
    <script type="text/javascript">
        var num = 1;
        function firstFunc(){
            var num = 7;
            num++;
            console.log(num);
        }
        function secondFunc() {
            console.log(num);
        }
    </script>
</body>
```

Fig 8.42

Example: If I click on the second button, then click on first button, then click on second button, then finally click on the first button. The console will look like this

Fig 8.43

Explanation:

Step 1: When I clicked on the second button, the value of variable num will be sent to the console. We have two variable num here. One is a local variable to firstFunc function while the other is a global variable. Since the second button triggers secondFunc, and secondFunc doesn't have access to the local variable in firstFunc, it is evident that the 1 logged to the console is the global variable.

Step 2: When I clicked on the first button, firstFunc is triggered. firstFunc adds 1 to the value of variable num. We have two variable num. One is a global variable while the other is a local variable to firstFunc. Within firstFunc function, the local variable overrides the global variable since they bear the same name.

Whenever we call on variable num within function firstFunc, we are calling on the local variable and not the global variable. Then, we added one to the value of variable num (the local variable) and logged it to the console. Therefore, 8 is sent to the console.

Step 3: When I click on the second button again. It logs the value of variable num (1) to the console. "Oops, but you added one to the value of variable num in step 2". Well, the 1 was added to the local variable in firstFunc and not to the global variable. Therefore, the value of the global variable will still remain 1.

Step 4: When I click in the first button again. It adds 1 to the value of variable num and log the result (8) to the console. "But we added one to variable num in step 2, why are we seeing 8 and not 9?". Remember that local variables are recreated each time the function is triggered.

Take home: Only use global variable when multiple functions need to access the value of the variable. If the variable is used only by one function, make it a local variable.

Parameterized function

I love calling parameterized function "enhanced function" or "super function". Here is the reason.

Consider the codes below:

```
<body>
    <button onclick="firstFunc()">First</button>
    <button onclick="secondFunc()">Second</button>
    <button onclick="thirdFunc()">Third</button>
    <script type="text/javascript">
        function firstFunc(){
            var school = "SQI";
            console.log("My name is Taye, I am a graduate of "+school);
        }
        function secondFunc() {
            var school = "LAUTECH";
            console.log("My name is Taye, I am a graduate of "+school);
        }
        function thirdFunc() {
            var school = "UNILORIN";
            console.log("My name is Taye, I am a graduate of "+school);
        }
    </script>
</body>
```

Fig 8.44

When users click on the first button, it logs “My name is Taye, I am a graduate of SQI” to the console.

When users click on the second button, it logs “My name is Taye, I am a graduate of LAUTECH” to the console.

When users click on the third button, it logs “My name is Taye, I am a graduate of UNILORIN” to the console.

Although the three functions are displaying different things, a closer look at them will reveal to you that they are actually doing the same thing. The only difference in the three functions is the value of variable “school”.

Since they are doing almost the same thing, we can empower one function to do the three tasks such that when we click on the first button, the value of variable school is SQI. When we click on the second button, the value of variable school is LAUTECH. When we click on the third button, the value of variable school is UNILORIN. We will empower a single function to do the work of the three functions.

My dear student, I believe that you would have inferred from the explanation above that one of the uses of parameterized functions is that: When multiple functions perform similar things, we can use parameterized function to reduce the codes.

It doesn’t only reduce the codes, it also makes it easier to maintain your codes as you will see soon. Let’s reduce the codes with parameterized function.

Enough of the speed, slow down here.

Understand what we want to do: We want to upgrade a single function to perform the work of three functions such that when we click on three different buttons, they will trigger the same function but will do different things. It sounds ridiculous.

If you are yet to understand what we want to do, consider the codes below

```
<body>
    <button onclick="firstFunc()">First</button>
    <button onclick="firstFunc()">Second</button>
    <button onclick="firstFunc()">Third</button>
    <script type="text/javascript">
        function firstFunc(){
            var school = "SQI";
            console.log("My name is Taye, I am a graduate of "+school);
        }
    </script>
</body>
```

Fig 8.45

When we click on any of three buttons above, it will log “My name is Taye, I am a graduate of SQI” to the console. Although we want to get that result when we click on the first button, we want a different result when we click on the second and third buttons, yet we want only one function to do that for us.

I hope you understand the problem at hand.

Since we want the function to behave in three different ways when it is triggered at three different locations, then we have to add sensitivity to the function so it knows where it is triggered and reacts accordingly.

If this sensitivity is not added, the function will react the same way to the three buttons.

Adding the sensitivity is simple. Whenever we trigger the function, a message will be sent to the function. Each of the three buttons will send different messages to the function. The function can use the message sent to know the button that triggered it and react accordingly.

To send a message to the function, just place a value within the parenthesis at the point of triggering the function.

As shown below, I will send the name of the school as a message to the function whenever the function is triggered.

```
<body>
    <button onclick="firstFunc('SQI')">First</button>
    <button onclick="firstFunc('LAUTECH')">Second</button>
    <button onclick="firstFunc('UNILORIN')">Third</button>
    <script type="text/javascript">
        function firstFunc(){
            console.log("My name is Taye, I am a graduate of "+school);
        }
    </script>
</body>
```

Fig 8.46

With the code above, a message is added to the function's call. When users click on the first button, 'SQI' is sent to the function. When users click on the second button, 'LAUTECH' is sent to the function. When users click on the third button, 'UNILORIN' is sent to the function.

Schools names are in quotes because they are strings, other datatypes can also be sent as we will see later in this chapter.

If you run the codes above on the browser, an error will be logged to the console saying "school is not defined". If you've been following us, you should know the reason for the error. Variable "school" has been removed therefore line 12 doesn't understand "school" (Check chapter two). For the meantime, I will remove the "school" on line 12 so there won't be any error.

```
<body>
    <button onclick="firstFunc('SQI')">First</button>
    <button onclick="firstFunc('LAUTECH')">Second</button>
    <button onclick="firstFunc('UNILORIN')">Third</button>
    <script type="text/javascript">
        function firstFunc(){
            console.log("My name is Taye, I am a graduate of ");
        }
    </script>
</body>
```

Fig 8.47

When users click on any of the buttons, "My name is Taye, I am a graduate of " will be logged to the console. "But we sent the names of the schools" as messages to the function, why are they not displayed?

Although we are sending messages to the function, the function is not equipped to receive the messages sent to it. To equip the function, place any variable name of your choice between the parenthesis of the function's definition as shown below.

```
<body>
    <button onclick="firstFunc('SQI')">First</button>
    <button onclick="firstFunc('LAUTECH')">Second</button>
    <button onclick="firstFunc('UNILORIN')">Third</button>
    <script type="text/javascript">
        function firstFunc(mySchool){
            console.log(mySchool);
            // console.log("My name is Taye, I am a graduate of ");
        }
    </script>
</body>
```

Fig 8.48

As seen above, I named the variable within the parenthesis of the function's definition mySchool. You can give it any name of your choice but it must comply with the rules of variable declaration. When the function is triggered, the message within the parenthesis of the function's call is sent as the value of variable mySchool.

If users click on the first button, 'SQI' is sent as the value of variable mySchool. If users click on the second button, 'LAUTECH' is sent as the value of variable mySchool. If users click on the third button, 'UNILORIN' is sent as the value of variable mySchool.

Run the codes above on the browser and see how it works.

Finally, we have

```
<body>
    <button onclick="firstFunc('SQI')">First</button>
    <button onclick="firstFunc('LAUTECH')">Second</button>
    <button onclick="firstFunc('UNILORIN')">Third</button>
    <script type="text/javascript">
        function firstFunc(mySchool){
            console.log("My name is Taye, I am a graduate of "+mySchool);

        }
    </script>
</body>
```

Fig 8.49

The codes above will work exactly like before. When the button is clicked, firstFunc is triggered and the message in the function's call is sent as the value of variable mySchool.

The message in the function's call is known as **argument**. The variable in the parenthesis of the function's definition ('mySchool' in the code we have above) is known as **parameter**.

Parameters behave like local variables meaning they cannot be used outside the function.

Remember I said that parameterized function also helps you in code maintenance. Let's look at that together. If I need to add two more schools to the initial code, then I would have to create two more functions as shown below

```
<button onclick="firstFunc()">First</button>
<button onclick="secondFunc()">Second</button>
<button onclick="thirdFunc()">Third</button>
<button onclick="fourthFunc()">Fourth</button>
<button onclick="fifthFunc()">Fifth</button>
<script type="text/javascript">
    function firstFunc(){
        var school = "SQI";
        console.log("My name is Taye, I am a graduate of "+school);
    }
    function secondFunc() {
        var school = "LAUTECH";
        console.log("My name is Taye, I am a graduate of "+school);
    }
    function thirdFunc() {
        var school = "UNILORIN";
        console.log("My name is Taye, I am a graduate of "+school);
    }
    function fourthFunc() {
        var school = "OAU";
        console.log("My name is Taye, I am a graduate of "+school);
    }
    function fifthFunc() {
        var school = "UI";
        console.log("My name is Taye, I am a graduate of "+school);
    }
</script>
```

Fig 8.50

Compare it with parameterized function

```
 6  <body>
 7      <button onclick="firstFunc('SQI')">First</button>
 8      <button onclick="firstFunc('LAUTECH')">second</button>
 9      <button onclick="firstFunc('UNILORIN')">Third</button>
10      <button onclick="firstFunc('OAU')">Fourth</button>
11      <button onclick="firstFunc('UI')">Fifth</button>
12      <script type="text/javascript">
13          function firstFunc(mySchool){
14              console.log("My name is Taye, I am a graduate of "+mySchool);
15          }
16
17      </script>
18  </body>
```

Fig 8.51

This is more efficient and easier to maintain.

Let’s look at other examples:

```
<body>
    <button onclick="myFunc(3)">First</button>
    <button onclick="myFunc(4)">second</button>
    <button onclick="myFunc(5)">Third</button>
    <script type="text/javascript">
        function myFunc(val){
            alert(2 * val);
        }
    </script>
</body>
```

Fig 8.52

When users click on the first button, it will alert 6 because 3 is sent as the value of val. When they click on the second button, it will alert 8 because 4 is sent as the value of val. When they click on the third button, it will alert 10 because 5 is sent as the value of val.

Example:

```
<body>
    <button onclick="myFunc(1, 2)">First</button>
    <button onclick="myFunc(3, 4)">second</button>
    <button onclick="myFunc(5, 6)">Third</button>
    <script type="text/javascript">
        function myFunc(val){
            alert(val);
        }
    </script>
</body>
```

Fig 8.53

The code above alerts 1, 3, and 5 when the first, second and third buttons are clicked respectively. What’s going on? The other values-2, 4, 6- are ignored.

Explanation: When a user clicks on the first button, myFunc is triggered and the arguments (1 and 2) are sent as messages to function myFunc. Unfortunately, the function cannot receive the two values because we only have one parameter (val). Instead, it assigns the first argument (1) to the only parameter present (val).

If we want the function to receive the second argument, then we need to equip the function with two parameters as shown below.

```
<body>
    <button onclick="myFunc(1, 2)">First</button>
    <button onclick="myFunc(3, 4)">second</button>
    <button onclick="myFunc(5, 6)">Third</button>
    <script type="text/javascript">
        function myFunc(val, num){
            alert(val * num);
        }
    </script>
</body>
```

Fig 8.54

When users click on the first button, the two arguments (1, 2) are sent to the two parameters in the order they were written. 1 is assigned to val, 2 is assigned to num. It will alert 2 (1 * 2).

When users click on the second button, the two arguments (3, 4) are sent to the two parameters in the order they were written. 3 is assigned to val, 4 is assigned to num. It will alert 12 (3 * 4).

When users click on the third button, the two arguments (5, 6) are sent to the two parameters in the order they were written. 5 is assigned to val, 6 is assigned to num. It will alert 30 (5 * 6).

This is known as positional argument because the arguments are sent to the parameters in the order they were written.

Note that you can send as many arguments as you want.

Summary: Parameterized functions allow arguments to be sent to parameters so the function can react and give the output based on the values of the arguments sent.

Side note: Do you figure out that the inbuilt method, console.log is a parameterized function? When the function is triggered it logs the argument we pass to it to the console.

Default parameter

Consider the codes below.

```
<body>
    <button onclick="myFunc(1, 2)">First</button>
    <button onclick="myFunc(3)">second</button>
    <button onclick="myFunc()">Third</button>
    <script type="text/javascript">
        function myFunc(val, num){
            alert(val * num);
        }
    </script>
</body>
```

Fig 8.55

We have established that when users click on the first button, 1 and 2 are sent as the value of val and num respectively.

What happens when users click on the second button? 3 will be sent as the value of parameter val but nothing is sent as the value of parameter num since the second argument was not provided. The default value of a parameter is undefined. For this reason, the value of num will be undefined. It alerts NaN which means “Not a Number” because it tries to execute 3 * undefined.

When users click on the third button, myFunc is triggered but no argument was provided therefore the value of both parameters (val and num) will be undefined. The function alerts NaN because it tries to execute undefined * undefined.

Well I’m not pleased with the NaN output. This can be solved using default parameter.

Check it below.

```
<body>
    <button onclick="myFunc(1, 2)">First</button>
    <button onclick="myFunc(3)">second</button>
    <button onclick="myFunc()">Third</button>
    <script type="text/javascript">
        function myFunc(val=0, num=1){
            alert(val * num);
        }
    </script>
</body>
```

Fig 8.56

You will observe that I have set the value of parameter val to 0 and the value of parameter num to 1. These are known as default parameters.

Default parameters are set when you set a value for your parameters right within the parenthesis of the function's definition.

How does this affect our code?

When a parameterized function is triggered, it will check if an argument is provided for the parameter. If it is provided, it uses the argument as the value of the parameter. If it is not provided, it sets the value you set within the parenthesis of the function's definition as the value of the parameter.

In the codes above, when a user clicks on the first button, 1 and 2 are sent as the value of val and num respectively. The 1 and 2 overrides the values (0 and 1) we set as default parameters. It therefore alerts 2 (1 * 2).

When a user clicks on the second button, 3 is sent as the value of the first parameter (val) but nothing is sent as the value of the second parameter, it therefore uses the default value (1) as the value of num. It alerts 3 (3 * 1).

When a user clicks on the third button, nothing is sent as the values of the two parameters, therefore, the function uses the default values as the value of both parameters. It sets the value of val to 0 and the value of num to 1. It alerts 0 (0 * 1).

Event Keyword

Look at this magic.

Consider these two sets of codes

First one:

```
 6  <body>
 7      <button onclick="myFunc(a)">First</button>
 8      <script type="text/javascript">
 9          function myFunc(b){
10              console.log(b);
11          }
12      </script>
13  </body>
```

Fig 8.57

Run the codes on the browser.

When a user clicks on the button, myFunc is triggered, argument “a” is sent as the value of parameter b and it is logged to the console. An error is displayed in the browser’s console because there is no variable “a” in our code (You should have gotten used to this ReferenceError by now and should be able to fix it). This error can be removed by declaring variable “a” or by placing “a” in a quote (making it a string).

Second one:

```
 6  <body>
 7      <button onclick="myFunc(event)">First</button>
 8      <script type="text/javascript">
 9          function myFunc(b){
10              console.log(b);
11          }
12      </script>
13  </body>
```

Fig 8.58

When a user clicks on the button, argument “event” is sent as the value of parameter b and logged to the console. Surprisingly, an error is not sent to the console. Instead, you see something like this:

```
                                                          index.html:10
▶ MouseEvent {isTrusted: true, screenX: 218, screenY: 147, clientX: 29, clien
  tY: 18, …}
```

Fig 8.59

“We didn’t create any variable called event, how come it is not throwing an error?” you may ask. The text ‘event’ is a keyword in JavaScript meaning that the browser immediately understands it when it sees it.

When users perform an event (click, keyup, mouseenter etc), JavaScript immediately creates an object (I will talk about objects in chapter ten) that gives you a lot of information about the specific event calling on the event handler (function). If you need this information, you add the event keyword as an argument to your function. We’ve not been adding the event keyword to our function because we haven’t needed it.

Let’s look at some of the information made available by the event keyword:

Path

```
<body>
    <button onclick="myFunc(event)">First</button>
    <div>
        <button onclick="myFunc(event)">Second</button>
    </div>
    <script type="text/javascript">
        function myFunc(b){
            console.log(b.path);
        }
    </script>
</body>
```

Fig 8.60

When a user clicks on the first button, this is logged to the console

Fig 8.61

When a user clicks on the second button, this is logged to the console

Fig 8.62

According to w3schools.com, path displays the elements involved in the event flow, in the correct execution order. Can you spot the difference between the two. Fig 8.62 includes a div.

For the first button: button lies in body, body lies in html, html lies in document, document lies in Window.

For the second button: button lies in div, div lies in body, body lies in html, html lies in document, document lies in Window.

Notice that both buttons called the same function. The event keyword was added to both buttons but they produced different result because the event keyword gives information about the particular event calling on the event handler.

Target

This is arguably the most used of all the properties of event. According to w3schools.com, the target event property returns the element that triggered the event.

```
 6  <body>
 7      <button onclick="myFunc(event)">First</button>
 8      <button onclick="myFunc(event)">Second</button>
 9      <script type="text/javascript">
10          function myFunc(b){
11              console.log(b.target);
12          }
13      </script>
14  </body>
```

Fig 8.63

Run the codes above on the browser.

If a user clicks on the first button, this is sent to the console

Fig 8.64

When a user clicks on the second button, this is sent to the console

Fig 8.65

You can see the button element in the console. We can access the innerText of the button tag since a button is an uneditable container. Here it is:

```
 6  <body>
 7      <button onclick="myFunc(event)">First</button>
 8      <button onclick="myFunc(event)">Second</button>
 9      <script type="text/javascript">
10          function myFunc(b){
11              console.log(b.target.innerText);
12          }
13      </script>
14  </body>
```

Fig 8.66

When a user clicks on the first button “First” is logged to the console. When a user clicks on the second button, “Second” is logged to the console. Try it out.

Finally, we can use this information to manipulate the page as shown below:

```
<body>
    <button onclick="myFunc(event)">First</button>
    <button onclick="myFunc(event)">Second</button>
    <script type="text/javascript">
        function myFunc(b){
            b.target.innerText = "Hello";
        }
    </script>
</body>
```

Fig 8.67

Run the codes below and click on the two buttons one after the other.

When you click on the first button, its innerText changes to “Hello”. When you click on the second button, its innerText changes to “Hello”.

Following conventions

According to Wikipedia “coding conventions are a set of guidelines for a specific programming language that recommend programming style, practices and methods for each aspect of a program written in that language”.

The code above is working well but we aren’t following convention. Your codes will work well even if you don’t follow conventions but I strongly recommend that you follow them. Following conventions make it easy for you to collaborate/work with other developers.

Conventions are like customs that we follow in a particular programming language.

It is a convention that you name the parameter of an event argument ‘e’ or ‘event’. For this reason, I will rename parameter b to e.

```
<body>
    <button onclick="myFunc(event)">First</button>
    <button onclick="myFunc(event)">Second</button>
    <script type="text/javascript">
        function myFunc(e){
            e.target.innerText = "Hello";
        }
    </script>
</body>
```

Fig 8.68

The advantage of this is that: Immediately you see the function's definition, you instantly know that it is expecting an event. I only discussed two properties-path and target- of the event object in this section. It has many other properties and methods but I believe the target property is enough to get started with the event object. You can read more on the event keyword here.

Return Keyword:

A short story

Isaac runs a logistic company. He often sends dispatch riders on errands. When a dispatch rider completes a delivery, the rider must return to Isaac to give him feedback except in cases of accidents.

Isaac = You.

Dispatch riders = functions.

Accident= Syntax error.

When a function is triggered, the function will always return a feedback to the **exact location** where it is triggered upon completion of the task. The only situation where feedbacks aren't returned to the exact location is when there is a syntax error.

Emphasis on exact location.

In this section, we want to assume everything goes as planned and there is no accident (syntax error). Error will be discussed in chapter twenty-two.

Since we aren't considering error in this section, it therefore means that when you trigger a function, it will always return a feedback to that **exact location** where it is triggered.

Although we have been calling on functions, we haven't been using the feedback from the function's call. In fact, it is possible to think our functions don't return feedbacks when they are called because the feedbacks aren't obvious. Let me call your attention to it. Walk with me. Consider the codes below:

```
 6  <body>
 7      <script type="text/javascript">
 8          function myFunc() {
 9              alert('hi');
10          }
11      </script>
12  </body>
```

Fig 8.69

Run the codes on the browser. Nothing will be seen on the browser since we are yet to trigger the function. Let's do that together.

We aren't going to trigger it in our code editor, instead, we will trigger it using the browser's console. Open the browser's console (Make sure you are doing this on the same tab you used to run the codes above), trigger the function in the browser's console as shown below.

```
> myFunc()
```

Fig 8.70

(To trigger a function, type the function's name and put a parenthesis right after it). Press ENTER.

It will alert "hi" on the browser, press "Ok" to close the alert dialog box. Check your console again. What do you notice? Do you notice the "undefined" as shown below? How did it get there?

```
> myFunc()
< undefined
```

Fig 8.71

Functions return undefined as feedbacks by default. When we triggered the function, the codes in the body of the function were executed which made it alert hi. It returned a feedback (undefined) when it completed the execution of the block of codes within the function. Notice that the feedback was returned to the exact place where the function was triggered. We can choose to return any value other than undefined but whenever you don't return any value within your function, it automatically returns undefined when it completes the execution of codes within the function.

Let's return a value of our choice.

```
 6  <body>
 7      <script type="text/javascript">
 8          function myFunc() {
 9              alert('hi');
10              return 5;
11          }
12      </script>
13  </body>
```

Fig 8.72

Run the codes on the browser and trigger the function in the browser's console. It will return 5 and not undefined.

Things to know about return

- The return keyword automatically terminates the function immediately it is read. Example:

```
6  <body>
7      <script type="text/javascript">
8          function myFunc() {
9              console.log('hi');
10             return 5;
11             console.log('hello');
12         }
13     </script>
14 </body>
```

Fig 8.73

Run the codes on the browser and call the function in the console. What do you notice?
Do you observe that 'hello' isn't shown in the console?
Here is the reason: When you trigger the function, the codes in the function's body is executed. It logs 'hi' to the console on line 9. It returns 5 on line 10 and immediately terminates the function. For this reason, every other codes in the body of the function is ignored.

- Since the return keyword returns the value to the exact location where the function is triggered, we can make use of the returned value.

```
6  <body>
7      <script type="text/javascript">
8          function myFunc() {
9              console.log('hi');
10             return 5;
11         }
12     </script>
13 </body>
```

Fig 8.74

Do this in your browser's console:

```
> myFunc() * 2
```

Fig 8.75

Trigger function myFunc and multiply it by 2 as shown in fig 8.75. You will have this result

Fig 8.76

Explanation: When you trigger the function, the codes in function myFunc is executed. It logs 'hi' to the console on line 9. It returns 5 on line 10. 5 is returned to the exact place we triggered the function (as a feedback). It then multiplies the feedback (5) by 2. So, we have 10 in the console.

Example:

Fig 8.77

What will I see if I run the codes below in the browser's console?

Fig 8.78

Answer: "Taye Abidakun"

When the function is triggered, it executes the code within the function and returns "Taye" to the exact spot where the function is triggered. It then concatenates this feedback ("Taye") with "Abidakun".

Enough of using the console to trigger a function. Let's go back to our code editor

Consider the codes below:

Fig 8.79

We triggered myFunc on line 11. It executes the codes within the function and returns "Taye" back to the location where the function was triggered. It then logs the feedback to the console.

Example:

```
6  <body>
7      <script type="text/javascript">
8          function myFunc() {
9              return "SQI";
10         }
11         console.log("My school is " +myFunc());
12     </script>
13 </body>
```

Fig 8.80

It will log "My school is SQI" to the console.

Example:

```
6  <body>
7      <script type="text/javascript">
8          function myFunc(age) {
9              return 2 * age;
10         }
11         console.log("I am "+myFunc(4)+ " years old");
12     </script>
13 </body>
```

Fig 8.81

It will log "I am 8 years old' to the console.

Example:

```
6  <body>
7      <script type="text/javascript">
8          function myFunc(age) {
9              return 2 * age;
10         }
11         var newAge = myFunc(10);
12         alert(newAge);
13     </script>
14 </body>
```

Fig 8.82

myFunc is a parameterized function that multiplies the value of age by 2 and returns the result.
On line 11, I triggered myFunc and passed 10 as an argument to parameter age. The feedback (20)(the returned value) is set as the value of variable newAge.
On line 12, I alert newAge, therefore 20 is displayed on the browser.

EXTERNAL JAVASCRIPT

External JavaScript: This is when JavaScript codes are written in a JavaScript file. To use external JavaScript, create a file with an extension of ".js". Place your JavaScript codes in the file. Finally, you link the JavaScript file with your HTML files.

Example:

Fig 8.83

The codes in fig 8.83 alerts "Hello". It is in a JavaScript file (Note the extension of the file). In order for the codes to alert "Hello" on the browser, it needs to be linked to a HTML file.

Let's do that together. I will create a HTML file.

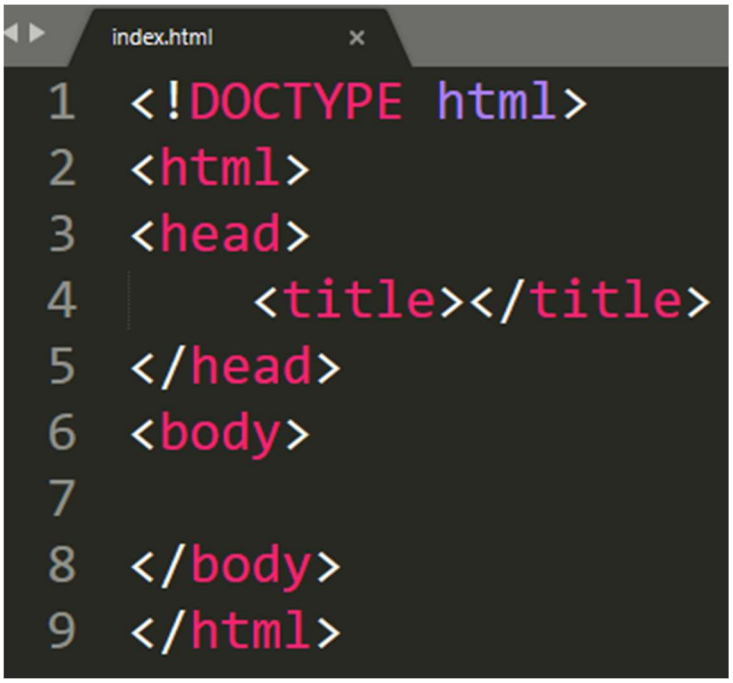

Fig 8.84

To link the HTML file in fig 8.84 with the JavaScript file in fig 8.83, open a script tag in the HTML file and provide the URL of the JavaScript file as the src attribute of the script tag as shown below.

index.html

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>

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

Fig 8.85

We have successfully linked the HTML file with the JavaScript file.

Run the HTML file on the browser and it will alert “Hello”.

When building applications, it is advisable to use external JavaScript over internal JavaScript. This is because external JavaScript allows different HTML files to share the same codes.

In summary: Internal JavaScript allows reusability of codes within a single file, thus it is preferred to inline JavaScript. External JavaScript allows reusability across different HTML files, thus it is preferred to internal JavaScript.

Nonetheless, I used internal JavaScript in the remaining part of this book, this is to capture the JavaScript and HTML codes in a single image.

CHAPTER NINE

JAVASCRIPT OPERATORS PART 3 AND IF STATEMENT

In this chapter, we will be discussing the comparison operators, logical operators and ternary operator.

Comparison operators

These operators allow us to execute commands if some conditions are met. This brings us to **IF statement.**

The if statement has three parts:

The first part is the "if" keyword.

The second part is the braces after the if keyword.

The third part is the curly braces after the second part.

Let us combine these three together. The if statement looks like this:

if(){}

Let us dive deeper here.

First part (the if keyword):

The if statement starts with the "if keyword". Just type the "if keyword" as shown below:

If

The second part (The braces after the if keyword): Immediately after typing the "if keyword", you put a braces just after it. You put the condition you want to check for, within the braces.

Let's assume you want to sleep by 10pm. It means you have to check the time to verify whether it is 10pm or not. If it is 10pm, you will sleep. If it isn't 10pm yet, you won't sleep. It means you need to keep checking the time to know whether it is 10pm or not.

The condition you are checking for, is the time. If the time is 10pm, then sleep. You put the condition you are checking for within the braces. As shown below:

If(time is 10pm)

The third part (The curly braces): You put whatever you want to do if the condition in the second part is met, right inside the curly braces in the third part. As in the example above, you want to sleep when it is 10pm.

So, we write it like this

if(time is 10pm){

sleep;

}

The code in the third part will not be executed if the condition in the second part is not met.

The block of code in the third part will only be executed if the condition in the second part is met.

This is the concept of "if statement".

Let us treat an example together.

Example: We want to create a variable "age" and set the value to any number of our choice. We will then display "I am a child" if the value of "age" is less than 18.

Let us do this together.

```
 7  <body>
 8  <div id="display"></div>
 9  <script type="text/javascript">
10  var age = 8;
11  if (age < 18) {
12      display.innerText = "I am a child";
13  }
14  </script>
15  </body>
```

Fig 9.1

Explanation: I created an empty div on line 8 and gave it an id of my choice (display). On line 10, I created a variable "age" and inserted 8 into it.

On line 11, I checked if the value of age is lesser than 18. Since 8 is lesser than 18, then it proceeds to read the codes within the curly braces. It places "I am a child" right inside the div we created earlier.

Run it on the browser, and see it yourself.

If we change the value of age to a number like 21. We won't see "I am a child" on the screen. This is because the condition will not be meant as 21 is not lesser than 18.

The less than sign (<) is a comparison operator and I just explained its usage.

Here is a list of other comparison operators and their names:

- <= Less than or equal to
- > Greater than
- >= Greater than or equal to
- == Equal to
- === Exactly equal to

The less than or equal to (<=):

Example:

Step 1: I will create a div in my HTML and label it “result”

Step 2: I will create a variable and label it num

Step 3: If the value of num is less than or equal to 10, I will display “This is small” in the div I created.

Here is the code:

```
 7  <body>
 8  <div id="result"></div>
 9  <script type="text/javascript">
10  var num = 3;
11  if (num <= 10) {
12      result.innerText = "This is small";
13  }
14  </script>
15  </body>
```

Fig 9.2

If we change the value of num to any number greater than 10, we won’t see anything on the screen.

Let’s have a small play around this.

In the code above, change the less than or equal to (<=) to less than (<), so that you have this:

```
 7  <body>
 8  <div id="result"></div>
 9  <script type="text/javascript">
10  var num = 3;
11  if (num < 10) {
12      result.innerText = "This is small";
13  }
14  </script>
15  </body>
```

Fig 9.3

Run it on the browser. You will see the same result. What then is the difference between <= and < ?

To see the difference, set the value of num to 10 as shown below:

```
 7 <body>
 8     <div id="result"></div>
 9     <script type="text/javascript">
10         var num = 10;
11         if (num < 10) {
12             result.innerText = "This is small";
13         }
14     </script>
15 </body>
```

Fig 9.4

If we use the less than sign, the condition on line 11 will not be true, therefore, nothing will be displayed in the div, but if we use the less than or equal to sign as shown below

```
 7  <body>
 8      <div id="result"></div>
 9      <script type="text/javascript">
10          var num = 10;
11          if (num <= 10) {
12              result.innerText = "This is small";
13          }
14      </script>
15  </body>
```

Fig 9.5

Then we will see “This is small” in the div. This is because the condition on line 11 will be true if the value of num is either less than or it is equal to 10.

The only difference between “<” and “<=” is that “<” returns true when the value at the left hand side of the sign is lesser than the value at the right hand side of the sign while “<=” returns true either when the value at the left hand side is lesser than the value at the right hand side or when the two values are the same.

The rule above is also applicable to “>” and “>=”.

Let’s take a look at the last two (== and ===).

Double equal sign (==)

Here is a question for you:

```
 9      <script type="text/javascript">
10          var num = 4;
11          if (num = 10) {
12              alert("hi");
13          }
14      </script>
```

Fig 9.6

What will I see on the browser if I run the code above?

a. hi b. Nothing will be displayed c. undefined d. null

Answer: The answer is option a. “hi” will be displayed.

Are you surprised? Oh! If you picked option b (Nothing will be displayed on the screen), although you are wrong but it shows you understand what I have ben explaining so far in this chapter.

Let's discuss the reason we got option a and not option b.

Explanation:

On line 10, we created variable num and inserted 4 in it. On line 11, we checked if the value of num is 10.

STOP THERE! We aren't checking on line 11, we are setting the value of num to 10. Remember "=" is an assignment operator, so the equal sign "=" will change the value of num to 10.

This is the reason "hi" will be displayed as the result.

Let us fix this. On line 11, our intention is not to set the value of num to 10, rather, it is to check if the value of num is 10. If it is true, we want it to alert the "hi" on line 12. If it is not true, we aren't doing anything. We only want to compare on line 11, not to assign a value to num.

We have to use another syntax since the equal sign (=) is an assignment operator.

JavaScript has given us another syntax to do that which is "==". The double equal sign (==) will only compare. It will not perform any assignment operation.

Let us rewrite our previous codes.

```
 9    <script type="text/javascript">
10        var num = 4;
11        if (num == 10) {
12            alert("hi");
13        }
14    </script>
```

Fig 9.7

Let us analyze the code above together:

Variable num was created on line 10 and 4 was set as the value of the variable.

We checked if the value of num is 10 on line 11. Since the value of num is not 10, then the block of codes in the third part of the "if statement" will be ignored.

The triple equal sign (===):

Let us take two tasks together.

Problem 9.1

What will I get if I run the block of codes below?

```
<script type="text/javascript">
    var val = 7;
    if (val == 7) {
        alert("success");
    }
</script>
```

Fig 9.8

a. success b. (Nothing will be displayed) c. undefined d. null

Answer: success

Problem 9.2

What will I get if I run the codes below?

```
<script type="text/javascript">
    var val = "7";
    if (val == 7) {
        alert("success");
    }
</script>
```

Fig 9.9

a. success b. (Nothing will be displayed) c. undefined d. null

Answer: success

Explanation: in problem 9.2, you can argue that nothing should be displayed since the values we compared are of different datatypes. One is a string while the other is a number. Your

argument is valid, but this is what you need to know. The double equal sign (==) only compares values but not datatypes.

As in the case above, it checks if 7 is 7. It doesn't care about the datatypes.

Sometimes, you don't just want to compare values alone, you also want to compare datatypes.

In that case, you use the triple equal sign.

The triple equal sign is known as strict equality. It only returns true if both datatypes and values that are compared are the same.

Problem 9.3: What will I see if I run the codes below on the browser?

```
9   <script type="text/javascript">
10      var val = "7";
11      if (val === 7) {
12          alert("success");
13      }
14  </script>
```

Fig 9.10

a. success b. (Nothing will be displayed) c. undefined d. null

Answer: b (Nothing will be displayed)

Nothing will be displayed because the condition is not met. The triple equal checks both the values and datatypes. In the codes in fig 9.10, the values are the same, but their datatypes are different, therefore, it returns false. The condition is not met.

Problem 9.4: What will I get if I run the code below on the browser?

```
<script type="text/javascript">
    var val = "7";
    if (val === "8") {
        alert("success");
    }
</script>
```

Fig 9.11

a. success b. (Nothing will be displayed) c. undefined d. null

Answer: b (Nothing will be displayed)

Explanation: Nothing will be displayed because the triple equal sign compares both the values and datatypes. Although the datatypes are the same (string), the values are different (7, 8).

Problem 9.5: What will I get if I run the codes below on the browser?

```
<script type="text/javascript">
    var val = "3";
    if (val === "3") {
        alert("success");
    }
</script>
```

Fig 9.12

a. success b. (Nothing will be displayed) c. undefined d. null

Answer: a. success

Explanation: The values and datatypes of what we are comparing are the same, therefore it will alert success.

Dealing with Boolean

QUESTION: What will I get if I run the code below on the browser?

```
<body>
    <div id="display"></div>
    <script type="text/javascript">
        var checked = true;
        if (checked==true) {
            display.innerText = "present";
        }
    </script>
</body>
```

Fig 9.13

Answer: “present” will be displayed in the div with an id of “display” since the condition is met. The value of “checked” is true.

Problem 9.6: What will I get if I run the code below on the browser?

```
<body>
    <div id="display"></div>
    <script type="text/javascript">
        var checked = false;
        if (checked==true) {
            display.innerText = "present";
        }
    </script>
</body>
```

Fig 9.14

Answer: Nothing will be displayed in the div since the condition is not met. The value of “checked” is not true.

BEHIND THE SCENE

This part is very important. Do not skip.

Problem 9.7: What will I get if I run the code below on the browser?

```
<script type="text/javascript">
    var checked = false;
    if (checked) {
        alert("hello");
    }
</script>
```

Fig 9.15

a. I will see “hello”
b. I won’t see anything
c. Undefined
d. It will throw an error.

Answer: b. I won’t see anything.

Explanation:

Here is how “if statement” works in JavaScript.

```
<script type="text/javascript">
    if (true) {
        alert("hello");
    }
</script>
```

Fig 9.16

The codes in fig 9.16 will alert “hello” if I run it on the browser.

```
<script type="text/javascript">
    if (false) {
        alert("hello");
    }
</script>
```

Fig 9.17

The codes in fig 9.17 will not alert “hello” If I run it on the browser. The reason is explained below.

Remember, I said “if statement” has three parts. The first part is the “if keyword”. The second part is the braces after the “if keyword”. The third part is the curly braces after the second part.

Our concern here is the second part. The braces after the “if keyword”. This is where we place the condition to be met. When the braces in the second part hold a condition that is true, then it will run the codes in the third part of the “if statement”. If the second part holds a condition that is false, then the codes in the third part will not be executed.

Before now we have been writing something like this in the second part

if(a < 4)

JavaScript will check the value of ‘a’ and check if it is less than 4 or not. If it is less than 4, it will return true and change it to this behind the scene: if(true)

If the value of a is not less than 4, it will return false, and change it to this behind the scene: if(false)

Whatever condition you type within the curly braces will be changed to either true or false behind the scene.

Let us play with this together.

Problem 9.8.

Behind the scene, what will JavaScript convert these blocks of codes to?

```
var  num = 5;
if(num==5){
        alert(“Taye”);
}
```

Options

a. var num = 5;

```
if(false){
    alert("Taye");
}
```

b. var num = 5;

```
if(true){
        alert("Taye");
}
```

Answer: b. Since the value of variable num is 5, then the condition is true, therefore, it will return true.

Problem 9.9.

Behind the scene, what will JavaScript convert these blocks of codes to?

```
var name = "Taye";
if(name == "Kehinde"){
    alert("good");
}
```

Options

a. var name = "Taye";

```
if(false){
    alert("good");
}
```

b. var name = "Taye"

```
if(true){
        alert("good");
}
```

Answer: a. Since the value of variable num is not Kehinde, then the condition is false. Therefore, it will return false.

Now that you know what happens behind the scene, let's establish some facts together. If JavaScript puts "true" within the curly braces in the second part of the "if statement", then it will execute the codes in the third part but if JavaScript puts "false" within the curly braces, it won't execute the codes in the third part.

It means we can help JavaScript by writing the “true” and “false” ourselves.

If we have a block of codes that looks like this:

If(true){

    alert(‘afternoon’);

}

Since we have “true” within the braces in the second part of the “if statement”, then it will alert “afternoon”.

If we have a block of codes that looks like this:

If(false){

    alert(‘morning’);

}

Since we have “false” within the braces in the second part of the “if statement”, then it will not alert “morning”.

I hope it is clear. You need to understand what I just explained before moving forward.

So, if we have

```
9     <script type="text/javascript">
10        var myBoolean = true;
11        if (myBoolean) {
12            alert("hello");
13        }
14    </script>
```

Fig 9.18

Since the value of variable myBoolean is true, line 11 is the same as

If(true){

Since we know that it will execute the third part of the “if statement” if we have true in the braces, then the codes above will alert “hello”.

Nothing will be seen on the browser if we run the codes below.

```
 9 <script type="text/javascript">
10      var proceed = false;
11      if (proceed) {
12          alert("move forward");
13      }
14 </script>
```

Fig 9.19

The reason is because the value of variable “proceed” is false. Line 11 can be written as: if(false){

As discussed earlier, when we have false in the second part of the “if statement”, then the third part is ignored.

Truthy and falsy:

```
 9 <script type="text/javascript">
10      var num = 5;
11      if (num) {
12          alert("hello world");
13      }
14 </script>
```

Fig 9.20

In the code above, we aren’t doing any comparison in the second part of the “if statement”. I said earlier that if JavaScript sees true in the second part of the “if statement”, it will run the codes in the third part. If it sees false in the second part of the “if statement”, it will not run the code in the third part. Now, we have a number in the second part, not a Boolean, what should JavaScript do?

Answer: JavaScript will convert whatever we have there to a Boolean (true or false). If it converts the value to true, then it will run the codes in the third part. If it converts it to false, it will ignore the codes in the third part of the if statement.

Here is a general rule: JavaScript only treats 6 things as false, every other thing is treated as true.

The 6 things here: false, null, undefined, 0, “ ”, NaN. These 6 things are considered falsy.

Anything aside the 6 things listed above is treated as true and they are considered truthy.

Back to fig 9.20, since 5 is not among the things JavaScript treats as false, then it will run the block of codes in the “if block”. Therefore, it will alert “hello world”.

Problem 9.10

What will be the output of the block of the codes below?

```
<script type="text/javascript">
    var val = undefined;
    if (val) {
        alert('good');
    }
```

Fig 9.21

a. It will alert “good”
b. Nothing will be shown on the browser.

Answer: Nothing will be shown on the browser.

Explanation: Remember I said only 6 things are treated as false, undefined is part of the 6 things that are treated as false. Since the value of ‘val’ is treated as false, then, it will not run the block of codes in the “if block”. Nothing will be shown on the browser.

Problem 9.11:

What will be the output of the codes below?

```
<script type="text/javascript">
    var val = "";
    if (val) {
        alert('Hi');
    }
```

Fig 9.22

a. “Hi” will be displayed on the browser.
b. Nothing will be displayed on the browser.

Answer: Nothing will be displayed on the browser since empty string (“”) is treated as false.

Problem 9.12:

What will be the output of the codes below:

```
<script type="text/javascript">
    var val = " ";
    if (val) {
        alert('Taye');
    }
</script>
```

Fig 9.23

a. “Taye” will be displayed on the browser.
b. Nothing will be displayed on the browser.

Answer: “Taye” will be displayed on the browser.

Explanation: The value of val in the code above is not an empty string. The value of val is “ “, which is different from “” (empty string). The whitespace between the opening and closing quotation marks is treated as true, therefore it runs the codes in the “if block”.

“ ” is not the same as “”. “ ” is treated as true while “” is treated as false.

LOGICAL OPERATORS:

There are three logical operators that we will be considered. They are: Not, Or, And.

Let’s pick them one after the other.

1. Not.
   This operator is used to negate something. The JavaScript syntax for “Not” is “!”.
   Let’s compare the English language with JavaScript.
   English Language: 5 is not equal to 7.
   JavaScript Language: 5 != 7
   Example

Image source

Fig 9.24

Imagine Someone says: if the image above is not of a goat, jump up.
Will you jump up or not?

a. You will jump up
b. You will not jump up.

Answer: You will jump up because the image above is that of a dog and not of a goat. Infact, you will jump up if the image is that of any other thing aside that of a goat.

If the person had said: If the image above is not of a dog, jump up.

Will you jump up or not?

a. You will jump up.
b. You will not jump up.

Answer: You will not jump up because the person asked you to jump if the image is not of a dog, since it is the image of a dog, you will not jump up.
If you don't understand the example above, go back to it and read it again till you understand it.
If you understand it, please proceed.

Problem 9.13:
What will be the output of the code below?

```
<script type="text/javascript">
    var num = 6;
    if (num != 14 ) {
        alert('success');
    }
</script>
```

Fig 9.25

a. It will alert “success”
b. It will display nothing.

Answer: It will alert “success”.

Explanation: In the second part of the “if statement”, the condition checks if the value of num is not equal to 14.

It will alert “success” if the value of num is any other thing aside 14.

Since 6 (the value of num) is not equal to 14, then the condition is true, therefore it alerts “success”.

Problem 9.14:

What will be the output of the code below?

```
<script type="text/javascript">
    var num = 3;
    if (num != 3 ) {
        alert('yes');
    }
</script>
```

Fig 9.26

a. It will alert “yes”
b. Nothing will be shown on the browser.

Answer: Nothing will be shown on the browser.

Explanation: The value of num is 3, the condition checks if the value of num is not equal to 3. Since the value of num is 3, then the condition is false, therefore, it will not run the codes in the third part of the “if statement”.

Strict inequality (!==)

Do you remember the strict equality syntax (===) that checks both the values and datatypes? The strict inequality syntax negates it.

What will be the output of the codes below?

```
 9      <script type="text/javascript">
10          var num = "5";
11          if (num != 5) {
12              alert("Hello");
13          }
14      </script>
```

Fig 9.27

a. It will alert “Hello”
b. Nothing will be seen on the browser.

Answer: Nothing will be seen on the browser.

Explanation: On line 10, we created a variable and inserted a string with a value of 5 into the variable.

On line 11, we checked if the value of num is not the same as number 5. JavaScript assume that string 5 is the same as number 5 (that is “5” == 5), so it will not alert “Hello”.

If you understand the explanation above, jump the next paragraph. If you don’t, move to the next paragraph.

This paragraph centers on line 11. Based on my earlier explanation, the code above will alert “Hello” if the value of num is any other thing aside 5, but since the JavaScript sees “5” and 5 as the same thing, it will not alert “Hello”.

If we want to compare the datatype also, then we need to change the code to

```
 9  <script type="text/javascript">
10      var num = "5";
11      if (num !== 5 ) {
12          alert('Hello');
13      }
14  </script>
```

Fig 9.28

Note the double equal sign. This will alert “Hello” and it will check if “5” is not equal to 5 in both value and datatypes.

The code above won’t see 5 and “5” as the same thing.

Problem 9.15:

What will be the output of the codes below?

```
 9 <script type="text/javascript">
10      var verify = false;
11      if (!verify) {
12          alert("login");
13      }
14  </script>
```

Fig 9.29

a. It will alert “login”
b. Nothing will be displayed on the browser.

Answer: It will alert “login”.

Under the “dealing with Boolean” section, I explained that whenever you have “true” in the second part of the “if statement”, it will run the codes in the third part. Whenever you have “false” in the second part of the “if statement”, it will not run the codes in the third part.

In the second part of the “if statement” in fig 9.29, we have “!verify”, which is “not false” because the value of “verify” is false. “not false” is “true”, So JavaScript converts line 11 to: if(true){

Since we have true in the second part of the “if statement”, then it will run the codes in the third part and alert “login”.

Conversely, the code below will not display anything on the browser.

```
 9 <script type="text/javascript">
10      var verify = true;
11      if (!verify) {
12          alert("login");
13      }
14  </script>
```

Fig 9.30

This is because “Not true” on line 11 is evaluated as false. So, JavaScript converts line 11 to: if(false){

2. **Or**
   Here is a story that explains how “or” works in JavaScript.

   A short story
   Your neighbor has a wild scary dog. Your children are scared of the dog and they avoid the dog as much as possible. Your neighbor has two spots where he keeps his dog. He has a cage at the left side of the house where he locks his dog. He also has a chain at the right side of the house where he locks his dog.
   Whenever the dog isn’t at any of these two spots, it means it isn’t locked and it can attack visitors.
   Your children love playing football and would always call you whenever they play the ball into your neighbor’s compound. You got tired of helping them pick the ball, so you gave them an instruction: Always look at both sides of the house to see if the dog is locked.

   Question 1: One day, your children played ball into your neighbor’s compound and need to pick the ball. They peeped through the gate and saw the dog locked in its cage. What should they do?
   a. To leave the ball because of the dog.
   b. To enter and pick the ball.

   Answer: To enter and pick the ball since the dog is locked in its cage.

   Question 2: One day, your children played ball into your neighbor’s compound and need to pick the ball. They peeped through the gate, looked at the left side and saw that the cage was empty, then they looked at the right side and saw the dog locked in chains. What should they do?
   a. To leave the ball because of the dog.
   b. To enter and pick the ball.

   Answer: To enter and pick the ball since the dog is locked with chain.

   Question 3: One day, your children played ball into your neighbor’s compound and need to pick the ball. They peeped through the gate, looked at the left side and saw that the cage was empty, they looked at the right side, the dog wasn’t there. What should they do?

   a. To leave the ball because of the dog.
   b. To enter and pick the ball.

   Answer: To leave the ball because of the dog.

If the dog is in its cage or in chains, then they can enter. If none of these is true, they shouldn't enter.

Note: Once they see the dog locked in its gate they can enter, there is no need to check if it is locked with chains. It is very important to note this.

If it is not in its cage, they check if it is locked with chains.
If it is in chains, they can enter.

If it is not in its cage and it is not locked in its chains, then they stay out.
This is how "or" works in JavaScript.
The syntax for "or" in JavaScript is "||".
Example:

```
 9 <script type="text/javascript">
10     var myNumber = 4;
11     if (myNumber== 4 || myNumber == 12) {
12         alert('successful');
13     }
14 </script>
```

Fig 9.31

The code above will alert "successful".

Explanation

On line 10: We created variable "myNumber" and inserted a value of 4 into it.

Let's take line 11 step by step.

If(myNumber==4 || myNumber == 12){

We have the "if keyword", then we have two conditions within the parenthesis. The two conditions within the parenthesis is the area of concern here (that is "myNumber == 4 || myNumber == 12").

Let's discuss only these two conditions leaving every other thing out. The first condition is "myNumber == 4", the second condition is "myNumber == 12". The two conditions are joined together with "or"("||").

This is how JavaScript processes it. It will check the first condition and see if the condition is true or false. The condition is true (because the value of myNumber == 4), it notes that and move to the next thing. The next thing is the "or syntax(||)"

joining the two conditions (“||”). Immediately it sees the “or syntax”, it won’t bother checking the second condition since the first condition was met. It jumps straight to the third part of the “if statement”, that is, the part in the curly braces of the “if statement”. Therefore, it alerts “successful”.

Can you remember your children and your neighbor’s dog? Once your children see the dog locked in its cage, they don’t need to check the right side of the house (whether it is locked in chains or not). They only check if it is locked in chains provided the first condition is false (that is, the dog is not in its cage).

Example:

```
<script type="text/javascript">
    var myNumber = 12;
    if (myNumber== 4 || myNumber == 12) {
        alert('successful');
    }
</script>
```

Fig 9.32

The code above will alert “successful”.

Explanation

On line 10: We created variable “myNumber” and inserted a value of 12 into it.

Let’s take line 11 step by step.

If(myNumber==4 || myNumber == 12){

We have the “if” keyword, then we have two conditions within the parenthesis. The two conditions within the parenthesis is the area of concern here (that is “myNumber == 4 || myNumber == 12”).

Let’s discuss only these two conditions leaving every other thing out. The first condition is “myNumber == 4”, the second condition is “myNumber == 12”. The two conditions are joined together with “or”(“||”).

This is how JavaScript processes it. It will check the first condition and see if the condition is true or false. The condition is false (because the value of myNumber is not 4), it notes that and move to the next thing. The next thing is the “or syntax(||)” joining the two conditions (“||”).

It checks the second condition because the first condition wasn't met. Fortunately for us, the second condition is met, so it moves to the third part of the "if statement", that is the part in the curly braces of "if statement". Therefore, it alerts "successful".

Back to the analogy of your children and your neighbor's dog, your children checked the cage and didn't see the dog, the next thing is to check the right side of the house and see if it is in chains. Congratulations to the children, the dog is locked in chains, so they can enter to pick the ball.

Example:

```
 9 <script type="text/javascript">
10      var myNumber = 7;
11      if (myNumber== 4 || myNumber == 12) {
12          alert('successful');
13      }
14 </script>
```

Fig 9.33

The code above will alert nothing.

Explanation:

On line 10: We created variable "myNumber" and inserted a value of 7 into it.

Let's take line 11 step by step.

If(myNumber==4 || myNumber == 12){

We have the "if keyword", then we have two conditions within the parenthesis. The two conditions within the parenthesis is the area of concern here (that is "myNumber == 4 || myNumber == 12").

Let's discuss only these two conditions leaving every other thing out. The first condition is "myNumber == 4", the second condition is "myNumber == 12". The two conditions are joined together with "or"("||").

This is how JavaScript processes it. It will check the first condition and see if the condition is true or false. The condition is false (because the value of myNumber is not 4), it notes that and move to the next thing. The next thing is the "or syntax(||)" joining the two conditions ("||").

It checks the second condition because the first condition wasn't met. Sadly, the second condition isn't true either. Since both conditions are false, it won't proceed to the part of the "if statement", so it does nothing.

Back to the analogy of your children and your neighbor's dog, your children checked the cage and didn't see the dog, then they checked the right side of the house to see if it was in chains.

The dog wasn't there either, so they wouldn't enter because none of the conditions is true.

Problem 9.16

What will be the output of the codes below?

```
<script type="text/javascript">
    var myNumber = 7;
    var secondNumber = 3;
    if (myNumber== 4 || myNumber == 12 || secondNumber > 2) {
        alert('successful');
    }
</script>
```

Fig 9.34

a. It will alert "successful"
b. It will alert nothing

Answer: It will alert "successful", since one of the conditions is true. The value of variable secondNumber is greater 2 since 3 is greater than 2.

Summary: When dealing with the or operator (||), the command in the third part of the "if statement" will be executed when any of the conditions in the second part is true.

3. **AND**

Analogy: A couple is made up of husband **AND** wife. One of them doesn't make a couple. We don't have a couple when only the man is present, neither do we have a couple when only the woman is present. We only have a couple when **BOTH** of them are present. This is how the **AND** operator works, that is, both conditions must be true.

The take home of this part is that: When you have two conditions joined together with the "and operator" at the second part of an "if statement" (within the parenthesis), the third part of the "if statement" will only be executed if and only if both conditions are met.

The JavaScript syntax for the "and operator" is &&.

Example:

```
 9 <script type="text/javascript">
10      var myNumber = 7;
11      var secondNumber = 3;
12      if (myNumber== 7 && secondNumber > 5) {
13          alert('successful');
14      }
15  </script>
```

Fig 9.35

The code in fig 9.35 will not display nothing.

Explanation: On line 10, we created a variable myNumber and inserted 7 in it. On line 11, we created a variable secondNumber and inserted 3 in it.

Line 12 is the area of interest. On line 12, we have the "if keyword", followed by a parenthesis. We have two conditions within the parenthesis. It checks the first condition (myNumber==7), which is true and proceed to the next thing. The next thing is the "and operator (&&)" joining the two conditions. Remember that in our analogy for the "and operator", I said both conditions must be true, therefore there is a need to check the other condition (Unlike the "Or operator" where the action can be executed when only one condition is true). JavaScript proceeds to the second condition to check if it is true or false. Well, it is false. Since the second condition is not true, the third part of the "If statement" is not executed, therefore, nothing will be seen on the browser.

Problem 9.17

What will be the output of the codes below?

```
 9 <script type="text/javascript">
10      var myNumber = 7;
11      var secondNumber = 3;
12      if (myNumber== 2 && secondNumber > 1) {
13          alert('successful');
14      }
15  </script>
```

Fig 9.36

a. It will alert "successful"

b. Nothing will be displayed on the browser.

Answer: Nothing will be displayed.

Explanation: On line 10, we created variable myNumber and inserted 7 in it. On line 11, we created a variable secondNumber and inserted 3 in it.

Line 12 is the main area of concentration. On line 12, we have the "if keyword", followed by a parenthesis. We have two conditions within the parenthesis. It checks the first condition (myNumber==2), which is false and proceed to the next thing. The next thing is the "and operator (&&)" joining the two conditions. Since the first condition was false and it sees an "and operator" after that, it won't disturb itself checking the second condition. It will do nothing.

If you don't understand the explanation, let's look at this together and it will be clearer.

Remember we said a couple is made up of a man and a woman. Imagine a pastor is called to join a couple together. The pastor will only join them together if and only if we have a man and a woman. On getting to the venue, the pastor realized that one of those he is to join together, is a child. He doesn't need to check the other person. He would be annoyed and angrily walk out of the venue.

That is how the "and operator" works. Once one of the conditions is not met (false), it doesn't need to check the second condition, it will angrily terminate the "if statement" without running the code in the third part of the "if statement".

For "or" operator, if one of them is true, it won't bother checking the other, immediately, it will execute the codes in the third part of the "if statement".

For the "and operator" if one of them is false, it won't bother checking the other, immediately, it will ignore the codes in the third part of the "if statement".

Problem 9.18

What will be the output of the codes below?

a. It will alert "successful"
b. It will alert nothing

```
 9 <script type="text/javascript">
10     var myNumber = 7;
11     var secondNumber = 3;
12     if (myNumber== 7 && secondNumber > 1) {
13         alert('successful');
14     }
15 </script>
```

Fig 9.37

Answer: It will alert “successful”.

Explanation: On line 10, we created a variable myNumber and inserted 7 in it. On line 11, we created a variable secondNumber and inserted 3 in it.

Line 12 is the area of interest. On line 12, we have the “if keyword”, followed by a parenthesis. We have two conditions within the parenthesis. It checks the first condition (myNumber==7), which is true and proceed to the next thing. The next thing is the “and operator (&&)” joining the two conditions. Remember that in our analogy for the “and operator”, we said both conditions must be true, therefore there is need to check the other condition (Unlike the “Or operator” where the action can be executed when only one is true). JavaScript proceeds to the second condition to check if it is true or false. The second condition is also true because the value of secondNum is greater than 1. Since the second condition is true, it proceeds to the third part of the “If statement”. Therefore, it will alert “successful”.

Problem 9.19

What will be the output of the following block of codes?

```
 9 <script type="text/javascript">
10     var myNumber = 7;
11     var secondNumber = 3;
12     if (myNumber== 7 && secondNumber > 1 && myNumber < 2) {
13         alert('successful');
14     }
15 </script>
```

Fig 9.38

a. It will alert “successful”
b. Nothing will be displayed on the browser.

Answer: Nothing will be displayed on the browser.

Explanation: There are three conditions all joined together with the “and operator”. Two of these conditions are true while the third is false. It will not run the third part of the “if statement” because one of the conditions is false.

Summary: When dealing with the “and operator (&&)”, the command in the third part of the “if statement” will NOT be executed when any of the conditions in the second part is false.

The truth table

Do you have problem understanding the “or” and the “and” operators? The truth table is here to help you.

Or table

True	Or	True	=	true
True	Or	False	=	true
False	Or	True	=	true
False	Or	False	=	false

And table

True	and	true	=	true
True	and	false	=	false
False	and	true	=	false
False	and	false	=	false

Let’s use this truth table for some “if statements” problems.

Example:

```
<script type="text/javascript">
    var firstValue = 4;
    var secondValue = 12;
    if (firstValue == 4 || secondValue == 6) {
        alert("Taye Abidakun");
    }
</script>
```

Fig 9.39

The code in fig 9.39 will alert “Taye Abidakun”.

Explanation: We have two conditions on line 12. The first condition is “firstValue== 4” which is true, the second condition is “seconndValue == 6” which is false.

Therefore line 12 can be rewritten as:

If (true || false){

From our truth table, true or false is true, therefore, the line above can be rewritten as

If(true){

Under the “dealing with Boolean” section, I explained that when you have “true” in the second part of an “if statement”, JavaScript will proceed to the third part but when we have “false” in the second part of the “if statement”, JavaScript will ignore the in the third part of the “if statement”.

Since we have “true” in the second part, then it will proceed to alert “Taye Abidakun”.

Whenever you need to combine the logical operators, the order of precedence goes thus

Not > And > Or

Example:

```
<script type="text/javascript">
    var firstValue = 4;
    var secondValue = 12;
    if (firstValue == 4 || secondValue == 5 && secondValue > 20 ) {
        alert("Good");
    }
</script>
```

Fig 9.40

The code above will alert “Good”.

Explanation: There are three conditions in the codes above.

The first is true, the second is false, so also is the third.

So line 12 looks like this:

If (true || false && false){

We only have two logical operators in the code above, they are “or operator (||)” and “and operator (&&)”. Recall that the “and operator” takes precedence over the “or operator”. It means JavaScript will evaluate the “false && false” section before the “true or false”.

From the truth table, we know that “false && false” will give “false”, therefore, the code above can be rewritten as:

If (true || false){

From the truth table,, we know that “true || false” will give “true”, therefore it can be rewritten as:

If (true){

It will alert “Good” because we have “true” in the second part of the “if statement”.

Parenthesis in “if statement”.

Parenthesis precedes other logical operators.

```
 9 <script type="text/javascript">
10     var firstValue = 4;
11     var secondValue = 12;
12     if ((firstValue == 4 || secondValue == 5) && secondValue > 20 ) {
13         alert("Good");
14     }
15 </script>
```

Fig 9.41

The code above will display nothing.

Explanation: There are three conditions in the codes above. The first is true, the second is false, so also is the third.

So, line 12 looks like this:

If ((true || false) && false){

Parenthesis precedes the other logical operators. The values within the parenthesis is first executed. From our truth table, “true || false” gives “true”. So the code above can be rewritten as:

if (true && false){

From our truth table, “true && false” will give “false”. It is executed as:

if (false){

Since we have “false” in the second part of the “if statement”, the third part of the “if statement” is ignored.

IF ELSE

In the “if statement” we have considered so far, if the condition in the second part of the “if statement” is true, it will proceed to the third part of the “if statement” but if the condition in the second part of the “if statement” is not met, it will ignore the codes in the third part.

Sometimes in programming, you want to do something when a certain condition is met and you want to do something else if the condition is not met (This is in contrast to what we have discussed so far where nothing is done if the condition is not met).

This is where the “if else” comes in.

Summary of what we will discuss in this section is this:

If the condition in the second part is met, it will run the codes in the “if block”, if the condition is not met, it will run the codes in the “else block”.

Example:

```
<script type="text/javascript">
    var num = 3;
    if (num > 7 ) {
        alert('high');
    }else{
        alert("low");
    }
</script>
```

Fig 9.42

In the block of codes above, the condition is false (that is, the value of num is not greater than 7), therefore, it won’t execute the codes in the “If block” but the codes in the “else block”. It will alert “low”.

```
<script type="text/javascript">
    var name = "Taye"
    if (name == "Taye" ) {
        alert('First');
    } else {
        alert("Second");
    }
</script>
```

Fig 9.43

What will be the output of the codes above?

a. It will alert “First”
b. It will alert “Second”
c. It will alert nothing

Answer: It will alert “First”. This is because the condition is true. The value of “name” is “Taye”.

IF ELSE IF:

For the “if statement”, a block of code will be executed when the condition is met while nothing is done when the condition is not met. For the “if else”, a block of code is executed when the condition is met while another block of codes is executed when the condition is not met. For the “if else if” we are about to consider now, we want to execute different blocks of codes based on different conditions.

Example:

Fig 9.44

On line 10, we created a variable “first” and inserted 4 in it. On line 11, we created a variable “second” and inserted 8 in it. Then, we created an “if block”.

In the “if block”, only the first condition is true while every other condition is false. The code will alert “Good” on the browser.

Example:

```
<script type="text/javascript">
    var first = 4;
    var second = 8;
    if (first == 3) {
        alert("Good");
    }else if(second== 8){
        alert("hi");
    } else if (first == 12) {
        alert("Hello");
    } else if (second > 10) {
        alert("success");
    }
</script>
```

Fig 9.45

Out of all the four conditions, only the second condition is true, therefore, the code will alert “hi” on the browser.

Example:

Fig 9.46

The code above will alert “Hello” on the browser, since only the third condition is true.

```
 9  <script type="text/javascript">
10      var first = 4;
11      var second = 8;
12      if (first == 3) {
13          alert("Good");
14      }else if(second== 2){
15          alert("hi");
16      } else if (first == 12) {
17          alert("Hello");
18      } else if (second > 5) {
19          alert("success");
20      }
21  </script>
```

Fig 9.47

The code above will alert “success” on the browser since only the fourth condition is true.

It is important to note that the “else if” part of the code can be as long as you want it to be.

A deeper look into the “if else if”.

Consider the codes below:

```
 9  <script type="text/javascript">
10      var first = 4;
11      var second = 8;
12      if (first == 4) {
13          alert("Good");
14      }else if(second== 2){
15          alert("hi");
16      } else if (first == 12) {
17          alert("Hello");
18      } else if (second > 5) {
19          alert("success");
20      }
21  </script>
```

Fig 9.48

Browser reads codes from the top to the bottom. We have one “else if” on line 14, another “else if” on line 16, another “else if” on line 18. It is important to note that the “else if” on line 14 will only be read if and only if the condition on line 12 is not met. The “else if” on 16 will only be read if and only if the condition on line 14 is not met. The “else if” on line 18 will only be read, if and only if the condition on line 16 is not met.

Note that the condition in an “else if” will only be read if the condition in the previous “if” is not met.

Based on the explanation above, what will be the output of the codes in fig 9.48

a. It will alert “Good”, then alert “success”
b. It will only alert “Good”
c. It will only alert “success”
d. It will alert nothing.

Answer: It will only alert “Good”.

The “else if” on line 14 will only be read if and only if the condition in the previous “if” is not met. The previous “if” in this case is on line 12. Since the condition is true, therefore the “else if” on line 14 and every other “else if” in the “if block” is ignored.

Example:

What will be the output of the codes below?

```
 9  <script type="text/javascript"
10      var first = 4;
11      var second = 8;
12      if (first == 3) {
13          alert("Good");
14      }else if(second== 8){
15          alert("hi");
16      } else if (first == 4) {
17          alert("Hello");
18      } else if (second > 10) {
19          alert("success");
20      }
21  </script>
```

Fig 9.49

a. It will only alert “Good”
b. It will only alert “hi”
c. It will only alert “Hello”
d. It will alert “hi” and “Hello”

Answer: It will only alert “hi”.

The first condition (on line 12) is false because the value of “first” is not equal to 3, therefore, it will proceed to line 14. The condition on line 14 is true, so it will alert “hi”. The condition on line 16 will not be read since the previous “if” condition is met.

If elseif else

Consider the codes below

```
<script type="text/javascript">
    var first = 4;
    var second = 8;
    if (first == 2) {
        alert("Good");
    }else if(second== 2){
        alert("hi");
    } else if (first == 12) {
        alert("Hello");
    } else if (second > 10) {
        alert("success");
    }
</script>
```

Fig 9.50

Nothing will be displayed on the browser since none of the conditions is true. In some circumstances, you want to execute some commands if none of the conditions is true. This is where the “if elseif else” comes in. A final else is added to the block of codes so the codes in it can be executed if none of the conditions is met.

Example

```
<script type="text/javascript">
    var first = 4;
    var second = 8;
    if (first == 2) {
        alert("Good");
    }else if(second== 2){
        alert("hi");
    } else if (first == 12) {
        alert("Hello");
    } else if (second > 10) {
        alert("success");
    }else{
        alert("Go away");
    }
</script>
```

Fig 9.51

The code above will alert “Go away” since none of the conditions is true. Note that the final else does not have any condition.

Problem 9.20

What will be the output of the following codes?

```
<script type="text/javascript">
    var getTime = 15;
    var myNum = 3;
    if (getTime >= 0 && getTime < 12) {
        alert("Morning");
    } else if (getTime >= 12 && getTime < 16) {
        alert("Afternoon");
    } else if (getTime>=16 && getTime< 19) {
        alert("Evening");
    } else if (getTime >= 20 && getTime < 24) {
        alert("Night");
    }
    if (myNum== 3) {
        alert("Hello");
    }
</script>
```

Fig 9.52

a. It will only alert “Afternoon”
b. It will only alert “Hello”
c. It will alert both “Afternoon” and “Hello”

Answer: It will alert “Afternoon” and “Hello”

Explanation: We have two **independent** “If blocks” in the codes above. An “if block” starts with “if”. In the first “if block”, only a block of codes will be executed, in the second “if block”, only a block of codes will be executed. Note that line 21 is not an “else if”, it is a new “if block” which is independent of the first “if block”.

Ternary operator

This is a cleaner and shorter form of writing “if else” statement.

The syntax for ternary operation is a question mark and a colon (?:).

I will divide the ternary operation into three parts.

1. The first part is the part before the question mark
2. The second part is the part after the question mark
3. The third part is the part after the colon.

A condition is placed in the first part (the part before the question mark). If the condition in the first part is met, JavaScript returns the codes in the second part (the part after the question mark) and ignores the codes in the third part (the part after the colon).

If the condition in the first part is not met, JavaScript ignores the codes in the second part and returns the codes in the third part.

Example:

```
7       <script type="text/javascript">
8           5 > 3 ? "Great" : "Small";
9       </script>
```

Fig 9.53

Since the condition in the first part is met, the codes above returns “Great”. You may realize that nothing is shown on the browser when you run the codes. The reason is that we are yet to use an output command to display the returned value. Let’s do that together. I will use console.log

Example:

```
7       <script type="text/javascript">
8           console.log(5 > 3 ? "Great" : "Small");
9       </script>
```

Fig 9.54

The codes above logs “Great” to the console.

You can also save the returned value in a variable as shown below.

```
7       <script type="text/javascript">
8           var status = 5 > 3 ? "Great" : "Small";
9           console.log(status);
10      </script>
```

Fig 9.55

Problem 9.21

What will be the output of the codes below?

```
7    <script type="text/javascript">
8        var num = 2;
9        var val = num == 7 ? "Perfect" : "Imperfect";
10       console.log(val);
11   </script>
```

Fig 9.56

a. Perfect b. Imperfect.

Answer: Imperfect.

On line 9, I checked if the value of num is equal to 7, since this isn't true, it returns the codes in the third part of the ternary operator ("Imperfect) and saves it as the value of val.

CHAPTER TEN

OBJECTS

What is an object?

This chapter discusses object while the next discusses array. Both of them have something in common. They are both used to store multiple values.

All the variables we have considered right to this point only store one value at a time but there are times you want to store multiple values in your container (variable). Objects and arrays come to the rescue.

An object is defined using curly braces. Here is an empty object.

```
7       <script type="text/javascript">
8           var myObject = {};
9       </script>
```

Fig 10.1

Let's store some values in an object.

A Short Story

I used to be a regular visitor at the national library, Akure. There was a gatekeeper at the entrance to the library that used to keep our bags/items in his office since bags were not allowed within the library. When you drop your bag/item with him, the gatekeeper would tie a tag to your bag and give you the duplicate of the tag. When it's time to collect your bag, all you have to do is to show him the tag and he will fetch your bag from the store.

Store = Object

Tag = key

Bags = values

To store an item/value (bag) in an object (store), you need to add a tag (key) to the item. The tag (key) is used to retrieve the items (value). The key names follow the same rules as that of variable declaration. Here is an example below

```
6  <body>
7      <script type="text/javascript">
8          var person = {name: "Taye", school: "SQI", dept: "Software"};
9      </script>
10 </body>
```

Fig 10.2

I created a container and named it “person”. This is the store. Variable “person” is an object. I stored some values in the store. The values I stored are “Taye”, “SQI” and “Software”. From our story, you can’t store a value without adding a key. The keys are: name, school, dept. These keys are used to retrieve the values. If I want to retrieve “Taye”, I need to use its key (name). If I want to retrieve “SQI”, then I need to use its key “school”. Key “dept” will be used to retrieve “Software”.

Note that each key-value pair is separated from the next with a comma (,).

While naming your key, ensure it explains the value it holds, the same way a variable name should explain the value it holds.

From the example above, the three keys (name, school, dept), and their respective values (“Taye”, “SQI”, “Software”) all belong to variable person.

We have stored values in our object (person). The next is to retrieve the values.

To retrieve a value, you use the object’s name (Since you can have many objects in your codes) and the key of the value you want to retrieve.

Example:

If I want to retrieve “Taye” from the object in fig 10.2, I will use the object’s name (person) and the key (name) to retrieve it. Here is the code below

```
7     <script type="text/javascript">
8         var person = {name: "Taye", school: "SQI", dept: "Software"};
9         console.log(person.name);
10    </script>
```

Fig 10.3

The codes above will log “Taye” to the browser’s console.

Problem 10.1

Write codes to retrieve the value of school.

Solution

```
7     <script type="text/javascript">
8         var person = {name: "Taye", school: "SQI", dept: "Software"};
9         console.log(person.school);
10    </script>
```

Fig 10.4

When dealing with objects, the order doesn’t matter since values are retrieved using keys. This means that the codes on line 8 and line 9 in fig 10.5 will work the same way.

```
7     <script type="text/javascript">
8         var person = {name: "Taye", school: "SQI", dept: "Software"};
9         var person = {dept: "Software", name: "Taye", school: "SQI"};
10    </script>
```

Fig 10.5

To retrieve the value of dept on line 8 codes, we will use person.dept. This is also the same code we need to retrieve the value of dept in line 9 codes.

Properties and methods

An object has two things, they are: Properties and methods. Properties are attributes of an object while methods are those things you can do with the object or those things you can do on the object.

Example: If you see me as an object, my properties are my attributes and those things that can be used to describe me, which includes my name, height, complexion etc. My methods are those things I can do: I can sleep, I can eat, I can walk etc, therefore, sleep, eat, walk are parts of my methods.

Properties

Let's create the object above. I will start with the properties

```
7     <script type="text/javascript">
8         var mySelf = {name: "Taye Abidakun", height: "1.76m", complexion: "dark"};
9     </script>
```

Fig 10.6

Each key-value pair in the image above is a property.

You can change the value of a key.

```
7     <script type="text/javascript">
8         var mySelf = {name: "Taye Abidakun", height: "1.76m", complexion: "dark"};
9         mySelf.height = "1.78m";
10        console.log(mySelf);
11    </script>
```

Fig 10.7

On line 8, I created an object called "myself" and gave it three properties. On line 9, I changed the value height from 1.76m to 1.78m. Here is the result when logged to the console on line 10.

```
▶ {name: "Taye Abidakun", height: "1.78m", complexion: "dark"} index.html:10
```

Fig 10.8

Problem 10.2: In the codes in fig 10.7, change the value of property name to “Samuel Bello”.

Solution:

```
7     <script type="text/javascript">
8         var mySelf = {name: "Taye Abidakun", height: "1.76m", complexion: "dark"};
9         mySelf.height = "1.78m";
10        mySelf.name = "Samuel Bello";
11        console.log(mySelf);
12    </script>
```

Fig 10.9

To add a new property to an already created object, you assign a value to the key of your choice.

I want to add property “age” to the object above, here is the code below

```
7     <script type="text/javascript">
8         var mySelf = {
9                         name: "Taye Abidakun",
10                        height: "1.76m",
11                        complexion: "dark",
12                    };
13
14            mySelf.age = 18;
15            console.log(mySelf);
16
17    </script>
```

Fig 10.10

Object within an object

```
 7      <script type="text/javascript">
 8          var mySelf=  {
 9                              name: "Taye Abidakun",
10                              height: "1.76m",
11                              complexion: "dark",
12                              school: {
13                                  name: "LAUTECH",
14                                  vc: "Prof. Gbadegeshin"
15                              }
16                          }
17              console.log(mySelf.school.name);
18      </script>
```

Fig 10.11

There are times you need to place an object within another object. In the codes above, Object "mySelf" has key 'school'. The value of the key is another object. On line 17, I accessed property "name" present in property "school" of object "mySelf". This is to show you that the value of an object's property can be another object.

Line 17 will log "LAUTECH" to the console.

Methods

A method is what an object can do or what you can do on an object. Methods are written like functions

```
 7      <script type="text/javascript">
 8          var mySelf = {
 9                              name: "Taye Abidakun",
10                              height: "1.76m",
11                              complexion: "dark",
12                              walk: function () {
13                                  console.log("Take a step");
14                              },
15                              sleep: function(){
16                                  alert("close eyes");
17                              }
18                          };
19
20      </script>
```

Fig 10.12

In the codes above, walk and sleep are methods.

Method = A function in an object.

You trigger the methods same way you trigger your functions except that you include the object's name.

Problem 10.3: Try to trigger method "walk" in fig 10.12 and see if you will get it right.

Solution:

```
7     <script type="text/javascript">
8         var mySelf = {
9                         name: "Taye Abidakun",
10                        height: "1.76m",
11                        complexion: "dark",
12                        walk: function () {
13                            console.log("Take a step");
14                        },
15                        sleep: function(){
16                            alert("close eyes");
17                        }
18                    };
19
20                    mySelf.walk();
21
22    </script>
```

Fig 10.13

A shorter way of creating methods is to remove the function keyword as shown below.

```
<script type="text/javascript">
    var mySelf = {
                        name: "Taye Abidakun",
                        height: "1.76m",
                        complexion: "dark",
                        walk() {
                            console.log("Take a step");
                        },
                        sleep(){
                            alert("close eyes");
                        }
                    };

                    mySelf.walk();

</script>
```

Fig 10.14

Note the removal of the function keyword. Our code is shorter. Fig 10.13 and fig 10.14 work the same way. Kindly use the one you are more comfortable with.

Parameters can be added to methods same way we add parameters to functions.

```
<script type="text/javascript">
    var mySelf = {
                        name: "Taye Abidakun",
                        height: "1.76m",
                        complexion: "dark",
                        walk(num) {
                            console.log("Take "+num+" steps");
                        },
                        sleep(){
                            alert("close eyes");
                        }
                    };

                    mySelf.walk(3);
</script>
```

Fig 10.15

When myself.walk is triggered, 3 is sent as the value of num of ‘walk’ method, therefore, “Take 3 steps” is logged to the console.

Accessing variables and properties within methods, introducing "this" keyword.

i. Accessing global and local variables

Methods have direct access to global and local variables.

Example:

```
<script type="text/javascript">
    var dept = "software";
    var mySelf = {
                    name: "Taye Abidakun",
                    height: "1.76m",
                    complexion: "dark",
                    walk(num) {
                        console.log("Take "+num+" steps to "+dept+ "department");
                    },
                    sleep(){
                        alert("close eyes");
                    }
                };
                mySelf.walk(3);
</script>
```

Fig 10.16

The codes above will log, "Take 3 steps to software department". In the code above, "dept" is a global variable. As you can see, it is easy to access global variables within methods.

```
<script type="text/javascript">
    var mySelf = {
                    name: "Taye Abidakun",
                    height: "1.76m",
                    complexion: "dark",
                    walk(num) {
                        var dept = "software";
                        console.log("Take "+num+" steps to "+dept+ "department");
                    },
                    sleep(){
                        alert("close eyes");
                    }
                };
                mySelf.walk(3);
</script>
```

Fig 10.17

The codes above (fig 10.17) is the same as the codes in fig 10.16 except that "dept" is now a local variable. The codes produced the same result. Accessing local variables within a method is not a problem.

ii. Accessing properties within a method:

```
<script type="text/javascript">
    var mySelf = {
                    name: "Taye Abidakun",
                    height: "1.76m",
                    complexion: "dark",
                    walk(num) {
                        var dept = "software";
                        console.log("Take "+num+" steps to "+dept+ "department");
                    },
                    sleep(){
                        alert("close eyes");
                    },
                    displayHeight(){
                        console.log("I am "+height+" tall");
                    }
                };
                mySelf.displayHeight();
</script>
```

Fig 10.18

Fig 10.18 is expected to log “I am 1.76m tall” to the console, for some unknown reasons, it isn’t working. If you run it on the browser, it will throw an error in the console as shown below.

```
▸Uncaught ReferenceError: height is not defined                index.html:20
    at Object.displayHeight (index.html:20)
    at index.html:23
```

Fig 10.19

Method displayHeight cannot access property “height”. Methods cannot access properties directly even if they belong to the same object (as in the case above, “height” property and displayHeight method both belong to object “mySelf”). How do we go about it?

Since methods can access global variables directly and object “mySelf” is a global variable, method “displayHeight” can take advantage of that and access any property it wants within the object. Here is the code below:

```
<script type="text/javascript">
    var mySelf = {
                    name: "Taye Abidakun",
                    height: "1.76m",
                    complexion: "dark",
                    walk(num) {
                        var dept = "software";
                        console.log("Take "+num+" steps to "+dept+ "department");
                    },
                    sleep(){
                        alert("close eyes");
                    },
                    displayHeight(){
                        console.log("I am "+mySelf.height+" tall");
                    }
                };
                mySelf.displayHeight();
</script>
```

Fig 10.20

This will log “I am 1.76m tall” to the console.

When the property you are trying to access and the method accessing the property both belong to the same object/parent (as we have in the code above where property “height” and method “displayHeight” both belong to object/parent “mySelf”), you can replace the parent’s name with “this” keyword”.

Here is the code below:

```
<script type="text/javascript">
    var mySelf = {
                    name: "Taye Abidakun",
                    height: "1.76m",
                    complexion: "dark",
                    walk(num) {
                        var dept = "software";
                        console.log("Take "+num+" steps to "+dept+ "department");
                    },
                    sleep(){
                        alert("close eyes");
                    },
                    displayHeight(){
                        console.log("I am "+this.height+" tall");
                    }
                };
                mySelf.displayHeight();
</script>
```

Fig 10.21

The codes above will give the same result as the on in fig 10.20.

Note: The “this” keyword within a method refers to the parent of the method calling on the “this” keyword.

If that sounds confusing, here is a short story:

Mr Ayinde owns a car and a house, both objects (car and house) have chairs. One day, Mr Ayinde was driving his wife to work and said “honey, these chairs are old. I need to change them”.

Question: Which chair was he referring to?

a. The chairs in the car
b. The chairs in the house

Without a doubt, Mr. Ayinde was referring to the chairs in the car. If he made the same statement while he was in his living room, then we would conclude that he was referring to the chairs in his house.

It means the value of ‘this’ is relative to the location of Mr. Ayinde. If Mr. Ayinde was in the car, ‘this’ refers to the car. If he was in his house, ‘this’ refers to the house.

‘this’ also works the same way in JavaScript. When you call on ‘this’ within a method, ‘this’ refers to the parent of the method calling on ‘this’.

```
<script type="text/javascript">
    var mySelf = {
                    name: "Taye Abidakun",
                    height: "1.76m",
                    complexion: "dark",
                    walk(num) {
                        var dept = "software";
                        console.log("Take "+num+" steps to "+dept+ "department");
                    },
                    sleep(){
                        alert("close eyes");
                    },
                    displayHeight(){
                        console.log(this);
                    }
                };
                mySelf.displayHeight();
</script>
```

Fig 10.22

The codes in fig 10.22 gives this result (fig 10.23).

```
                                                                    index.html:20
▼{name: "Taye Abidakun", height: "1.76m", complexion: "dark", walk: ƒ, slee
p: ƒ, …} i
    complexion: "dark"
  ▶ displayHeight: ƒ displayHeight()
    height: "1.76m"
    name: "Taye Abidakun"
  ▶ sleep: ƒ sleep()
  ▶ walk: ƒ walk(num)
  ▶ __proto__: Object
```

Fig 10.23

In the code in fig 10.22, 'this' refers to object 'mySelf'.

Problem 10.4: What will be the output of the codes below?

```
<script type="text/javascript">
    var person = {
                    name: "Taye Abidakun",
                    height: "1.76m",
                    complexion: "dark",
                    displaySchool(){
                        alert(this.school);
                    }

                };
                person.displaySchool();
</script>
```

Fig 10.24

Answer: It will alert undefined because "person" object doesn't have "school" as one of its properties. The codes can be fixed by adding "school" as a property of the object as shown below.

```
<script type="text/javascript">
    var person = {
                    name: "Taye Abidakun",
                    height: "1.76m",
                    complexion: "dark",
                    school: "SQI",
                    displaySchool(){
                        alert(this.school);
                    }

                };
                person.displaySchool();
</script>
```

Fig 10.25

The code above will alert "SQI".

Problem 10.5: What will be the output of the codes below?

```
7     <script type="text/javascript">
8         var person = {
9                             name: "Taye Abidakun",
10                            height: "1.76m",
11                            complexion: "dark",
12                            setSchool(){
13                                school = "SQI";
14                            },
15                            displaySchool(){
16                                alert(this.school);
17                            }
18
19                        };
20            person.displaySchool();
21    </script>
```

Fig 10.26

a. It will alert undefined
b. It will alert “SQI”

Answer: a. It will alert undefined.

Explanation: The parent of method displaySchool doesn’t have any property named school.

Problem 10.6: What will be the output of the codes below?

```
<script type="text/javascript">
    var person = {
                    name: "Taye Abidakun",
                    height: "1.76m",
                    complexion: "dark",
                    setSchool(){
                        this.school = "SQI";
                    },
                    displaySchool(){
                        alert(this.school);
                    }

                };
            person.displaySchool();
</script>
```

Fig 10.27

a. It will alert undefined
b. It will alert “SQI”

Answer: It will alert undefined.

Explanation: Remember that the codes in a function are ignored until the function is triggered. Same way, the codes in a method are ignored until the method is triggered. Method “setSchool” is meant to set “school” property on “person” object but the method (setSchool) was not triggered at any point in our codes.

Let’s fix it.

Fig 10.28

Method setSchool is triggered on line 20, this sets “school” property on “person” object. Method displaySchool is triggered on line 21, this alerts the value of property ‘school’, therefore, it alerts “SQI”.

There is more to ‘this’ keyword in JavaScript, it will be discussed in chapter twenty-one.

Dot and square bracket notation

There are two ways to set and retrieve the values in your object using the keys. They are

- Dot notation
- Square braces notation

We have been using the dot notation.

```
 7      <script type="text/javascript">
 8          var person = {
 9                              name: "Taye Abidakun",
10                              height: "1.76m",
11                              complexion: "dark",
12                              setSchool(){
13                                  this.school = "SQI";
14                              },
15                              displaySchool(){
16                                  alert(this.school);
17                              }
18
19                          };
20              console.log(person.name);
21      </script>
```

Fig 10.29

On line 20, there is a dot between “person” and “name”. This is the dot notation I am referring to.

- Square braces notation

  Instead of using dot, you can use square braces. The difference is that the square braces notation takes a string. Let’s rewrite the codes in fig 10.29 using square braces notation.

Fig 10.30

Do you notice that the “name” on line 20 is in quotes? Fig 10.29 will give the same result as fig 10.30. It is important to note that JavaScript allows us to write key names in quotes (variable names cannot be written in quotes but keys can be written in quotes) as shown below.

Fig 10.31

The quotes do not affect how you access the keys. You can access the keys using either dot notation or square braces notation as shown below:

```
7   <script type="text/javascript">
8       var person = {
9                       "name": "Taye Abidakun",
10                      "height": "1.76m",
11                      complexion: "dark",
12                      setSchool(){
13                          this.school = "SQI";
14                      },
15                      displaySchool(){
16                          alert(this.school);
17                      }
18                  };
19          console.log(person.name);
20          console.log(person['height']);
21          console.log(person['complexion']);
22
23  </script>
```

Fig 10.32

The code above will work fine with no error.

On line 19, I accessed “name” key using dot notation. Even though the key is in quotes (line 9), I didn’t include quotes while accessing it on line 19.

On line 20, I accessed “height” key using square braces notation. I placed a quote around “height” while using the square braces notation.

On line 21, I accessed “complexion” key using square braces notation. I placed a quote around “complexion” while accessing it on line 21 even though the key name isn’t in quotes (line 11).

Point to note: While the square notation begs for quotes, the dot notation hates it. An error will be thrown if I remove the quotes in the square braces notation or if I add a quote to the dot notation syntax.

You can use any of the two. Just make sure you’re consistent. I prefer the dot notation to the square braces notation because the dot notation is shorter and looks neater.

Advantages of dot notation over square braces notation

- It is shorter and looks neater.
- You can't use square braces notation to trigger methods.

The code below will do nothing

```
7       <script type="text/javascript">
8           var person = {
9                               name: "Taye Abidakun",
10                              height: "1.76m",
11                              complexion: "dark",
12                              setSchool(){
13                                  this.school = "SQI";
14                              },
15                              displaySchool(){
16                                  alert(this.school);
17                              }
18
19                          };
20              person['displaySchool()'];
21      </script>
```

Fig 10.33

Advantages of square braces notation over dot notation

1. Key names can contain whitespace.

```
7       <script type="text/javascript">
8           var person = {
9                               name of student: "Taye Abidakun",
10                              height: "1.76m",
11                              complexion: "dark",
12                              setSchool(){
13                                  this.school = "SQI";
14                              },
15                              displaySchool(){
16                                  alert(this.school);
17                              }
18
19                          };
20      </script>
```

Fig 10.34

The codes in fig 10.34 will throw an error.

Recall that I said earlier that key names can be written in quotes, we can take advantage of this to include white spaces in our key names.

```
7     <script type="text/javascript">
8         var person = {
9                         "name of student": "Taye Abidakun",
10                        height: "1.76m",
11                        complexion: "dark",
12                        setSchool(){
13                            this.school = "SQI";
14                        },
15                        displaySchool(){
16                            alert(this.school);
17                        }
18
19                    };
20    </script>
```

Fig 10.35

This will no longer throw an error. How then do we access the key name? You can't use the dot notation to access key names that contain white spaces, you can only the square braces notation.

The codes below will throw an error

```
7     <script type="text/javascript">
8         var person = {
9                         "name of student": "Taye Abidakun",
10                        height: "1.76m",
11                        complexion: "dark",
12                        setSchool(){
13                            this.school = "SQI";
14                        },
15                        displaySchool(){
16                            alert(this.school);
17                        }
18
19                    };
20            console.log(person.name of student);
21    </script>
```

Fig 10.36

While the code below will work fine

```
7     <script type="text/javascript">
8         var person = {
9                         "name of student": "Taye Abidakun",
10                        height: "1.76m",
11                        complexion: "dark",
12                        setSchool(){
13                            this.school = "SQI";
14                        },
15                        displaySchool(){
16                            alert(this.school);
17                        }
18
19                    };
20            console.log(person["name of student"]);
21    </script>
```

Fig 10.37

Notice the square braces on line 20.

Personally, I avoid adding whitespaces to my key names as much as possible.

2. Dynamic keys

The is the biggest advantage the square braces notation has over the dot notation.
Dynamic keys cannot be set using dot notation. You can only use square braces notation.
Before we move to dynamic keys, I will firstly discuss dynamic value.

Consider the codes below

```
6  <body>
7      <input type="text" id="myInp">
8      <button onclick="myFunc()">click me</button>
9      <script type="text/javascript">
10         function myFunc() {
11             var myObj = {};
12             myObj.dept = myInp.value;
13             console.log(myObj);
14         }
15     </script>
16 </body>
```

Fig 10.38

Whenever user clicks on the button, ‘myFunc’ is triggered and the codes therein gets executed. An empty object is created in line 11. On line 12, property with a key name of ‘dept’ is created and its value will be whatever the user typed in the input.

If the user types “computer science” in the input, then clicks on the button, the value of key “dept” will be “computer science”. If the user changed it to “software”, then clicks on the button, the value of key “dept” will be “software”. This means that users are able to set the value of key “dept”. The value is set at runtime. The value is dynamic. This is called dynamic value.

Now to dynamic key.

In the codes above (fig 10.38), although the value is dynamic, the key is static. The key name will always be “dept” irrespective of the users input. Let’s make the key dynamic, same way the value is dynamic so users can set the key name.

```
 6  <body>
 7      <input type="text" id="myKey">
 8      <input type="text" id="myValue">
 9      <button onclick="myFunc()">click me</button>
10      <script type="text/javascript">
11          function myFunc() {
12              var myObj = {};
13          }
14      </script>
15  </body>
```

Fig 10.39

Your task is to make add a dynamic key and a dynamic value to “myObj” such that the key name will be the value of the input with an id of myKey and the value of the key will be the value of the input with an id of myValue.

Start working on it.

Did you try something like this?

```
 6  <body>
 7      <input type="text" id="myKey">
 8      <input type="text" id="myValue">
 9      <button onclick="myFunc()">click me</button>
10      <script type="text/javascript">
11          function myFunc() {
12              var myObj = {};
13              myObj.myKey.value = myValue.value;
14              console.log(myObj);
15          }
16      </script>
17  </body>
```

Fig 10.40

Even though the code above doesn't work, it shows you've been following me.

You will get this error when the button is clicked.

```
▸Uncaught TypeError: Cannot set property 'value' of undefined index.html:13
    at myFunc (index.html:13)
    at HTMLButtonElement.onclick (index.html:9)
```

Fig 10.41

The reason is this: When a user clicks on the button, the codes in the body of function myFunc is executed. An empty object is created on line 12. On line 13, JavaScript assumes there is a property with a key name of 'myKey' in object 'myObj' and tries to set the 'value' property of key 'myKey' present in 'myObj' (that is, it expects to see: var myObj = {myKey: {}}). Since there is no key with a name of 'myKey' in 'myObj' (meaning it is undefined), it cannot set a 'value' property on a key that doesn't exist. Hence the error.

Give it another try.

Did you try this?

```
<body>
    <input type="text" id="myKey">
    <input type="text" id="myValue">
    <button onclick="myFunc()">click me</button>
    <script type="text/javascript">
        function myFunc() {
            var myObj = {};
            var dept = myKey.value;
            myObj.dept = myValue.value;
            console.log(myObj);
        }
    </script>
</body>
```

Fig 10.42

It's a good try even though it doesn't give the desired result. The key name will always be 'dept' irrespective of the users input. This is not what we want.

You cannot use the dot notation to set dynamic keys. You need to switch to square braces notation.

Here is the solution below:

```
<body>
    <input type="text" id="myKey">
    <input type="text" id="myValue">
    <button onclick="myFunc()">click me</button>
    <script type="text/javascript">
        function myFunc() {
            var myObj = {};
            var dept = myKey.value;
            myObj[dept] = myValue.value;
            console.log(myObj);
        }
    </script>
</body>
```

Fig 10.43

This will give us our desired result.

Let’s explain the codes line by line. When the button is clicked, function ‘myFunc’ is triggered.

Line 12: An empty object is created.

Line 13: variable ‘dept’ is created and its value is the value of the input with an id of myValue.

Recall that JavaScript will always treat any value gotten using these three output commands as string: value, innerText, innerHTML.

Therefore, the value of variable dept is a string.

For instance, if the user types ‘school’ as the value of input ‘myKey’ and ‘SQI’ as the value of input ‘myValue’, then click on the button. Line 13 will look like this: var dept = ‘school’;

Line 14: The value of variable dept is put in square braces. Since the value of ‘dept’ is a string, it makes it a perfect syntax for square braces notation. It means line 14 will look like this: myObj[‘school’] = ‘SQI’;

Line 15: The object is logged to the console.

You now know how to set dynamic keys.

Here is a small task for you:

In the codes above (fig 10.43), when a user sets the first key-value pair, it works fine, but there is a glitch when a user tries to set the second key-value pair. The first key-value pair suddenly disappears and it’s replaced by the second key-value pair. The third pair overrides the second, and so on, such that we only see one key-value pair in the console irrespective of the times the button is clicked.

Your task is to stop this: We want to retain the previous pair.

Try it out.

Solution:

```
<body>
    <input type="text" id="myKey">
    <input type="text" id="myValue">
    <button onclick="myFunc()">click me</button>
    <script type="text/javascript">
        var myObj = {};
        function myFunc() {
            var dept = myKey.value;
            myObj[dept] = myValue.value;
            console.log(myObj);
        }
    </script>
</body>
```

Fig 10.44

All you need to do is to convert variable ‘myObj’ from local variable to global variable. “Just like that?” Yes, it’s that simple.

The reason is this: If ‘myObj’ is a local variable, it implies that the object is recreated each time the button is clicked. By making it a global variable, the object is only created when the browser loads (Refer to chapter eight for details).

CHAPTER ELEVEN

ARRAY AND LOOP

Array and loop

Under this section, we will look at array separately, loop separately, then we combine them together.

Array

Like an object, an array is used to store multiple values. We use curly braces to create an object but we use square braces to create an array.

Here is an empty array:

```
7       <script type="text/javascript">
8           var countries = [];
9       </script>
```

Fig 11.1

Unlike objects, arrays don't have keys. All you need to do is to place your values directly in the array. Each value is separated from the next using a comma (,) as shown below.

```
7       <script type="text/javascript">
8           var countries = ["Nigeria", "Japan", "USA", "Ghana"];
9       </script>
```

Fig 11.2

How then do we retrieve values since we use keys to retrieve values in objects? In arrays, values are retrieved using the position of the value in the array. To see "Nigeria", I will retrieve the value in the first position of the array. To see "Japan", I will retrieve the value in the second position of the array. These positions are known as indexes. It means that: To see "Nigeria", I will retrieve the value in the first index of the array. To see "Japan", I will retrieve the value in the second index of the array. One more thing, the indexes of an array starts from 0, not 1 as one may assume. It means that, to see Nigeria, I will retrieve the value at index 0. To see Japan, I will retrieve the value at index 1. To see USA, I will retrieve the value at index 2. To see Ghana, I will retrieve the value at index 3.

Here is the code to retrieve "Nigeria" from the array:

```
7       <script type="text/javascript">
8           var countries = ["Nigeria", "Japan", "USA", "Ghana"];
9           console.log(countries[0]);
10      </script>
```

Fig 11.3

To retrieve a value from array, you follow these steps

1. You write the name of the array
2. You place square braces after the name
3. You place the index of the value of choice right between the square braces.

Problem 11.1: Log "USA" to the console from the countries array in fig 11.3

Solution:

```
7     <script type="text/javascript">
8         var countries = ["Nigeria", "Japan", "USA", "Ghana"];
9         console.log(countries[2]);
10    </script>
```

Fig 11.4

JavaScript arrays have some inbuilt properties and methods

Some properties and methods of arrays

- Length: This is a property that tells us the number of items in an array

Example:

```
7     <script type="text/javascript">
8         var countries = ["Nigeria", "Japan", "USA", "Ghana"];
9         console.log(countries.length);
10    </script>
```

Fig 11.5

This logs 4 to the browser's console. There are 4 items in the array in fig 11.5.

- Push: This method is used to add item(s) to an array.

Example:

```
7     <script type="text/javascript">
8         var countries = ["Nigeria", "Japan", "USA", "Ghana"];
9         countries.push("Italy", "India");
10        console.log(countries);
11    </script>
```

Fig 11.6

Two countries were pushed into the countries array on line 9. Here is the result on the browser.

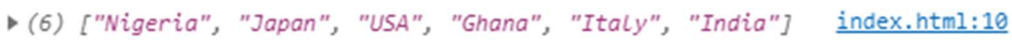

Fig 11.7

- Pop: This is used to remove the last item in an array.

Example:

```
7     <script type="text/javascript">
8         var countries = ["Nigeria", "Japan", "USA", "Ghana"];
9         countries.pop();
10        console.log(countries);
11    </script>
```

Fig 11.8

Here is the result. The last item in the array -Ghana- is removed by pop.

Fig 11.9

Here is a link to other array methods.

LOOP

Loops are important whenever you need to do something repeatedly. It also allows us to do some manipulations while repeating it.

Example: If I want to display "I am a boy" 10 times on my page. I can type it 10 times (Using a loop would have made it easier though), but what happens if I need to display it a thousand times? Well, I can copy and paste it numerous times (Again, using a loop would have made it easier). Oh, what happens if I want number to add serial number to it, for instance "1. I am a boy", "2. I am a boy", "3. I am a boy" etc . Copy and paste will not help. We need to use a loop. There are three loops in JavaScript: For loop, While loop, Do while loop.

The "for loop" and "while loop" work the same way, while the "Do while" loop is a bit different. In this book, I will only discuss the "for loop".

For loop

The "for loop" has three parts. The first part is the "for" keyword. The second part is the braces after the "for" keyword. The third part is the curly braces after the braces.

Let us combine these three together. The "for loop" looks like this: **for(){}**

Let's dive deeper here.

First part (the "for" keyword):

The “for loop” starts with the “for” keyword. Just type the “for keyword” as shown below:

for

The second part (The braces after the for keyword): A braces is placed right after the “for” keyword. There are at least two things within the braces.

- First, there is an initialization of a variable. I will call this part 2A.
- Second, you place a condition immediately after you initialize a variable. I will call this part 2B.

If the condition in part 2B is true, JavaScript proceeds to run the codes in the third part. If the condition is false, JavaScript won’t.

The third part (The curly braces): This part contains the task you want to execute if the condition in part 2B is met.

You will agree with me that the three steps are similar to the ones in “if statement”. This makes it easier for you to understand the “for loop”.

Summary of what I am about to explain: Part 2A is only executed once while part 2B and part three of the “for loop” are executed repeatedly.

Let’s say we want to log “I am a boy” to the console 3 times. Here is the code to do that.

```
7     <script type="text/javascript">
8         for(var i = 1; i < 4;){
9             console.log("I am a boy");
10            i++;
11        }
12    </script>
```

Fig 11.10

Let’s explain the codes above.

First part

On line 8, we have the first part of the “for loop” which is the “for keyword”.

Second part

On the same line 8, we have the second part of the “for loop”, which is the parenthesis after the “for” keyword. Within this parenthesis, I did two things

- First, I created a variable “i” and gave it a value of 1. (Part 2A).

- Second, I placed a condition after initializing variable “i”. The condition is to check if the value of “i” is lesser than 4 (Part 2B). Since the value of “i” (1) is lesser than 4, then JavaScript proceeds to run the codes in the third part.

Third part

Line 9: “I am a boy” is logged to the console.

Line 10: There is a post-increment of “i”. Therefore, the value of “i” increases from 1 to 2.

Line 11: We have the closing part of the curly braces.

Do you understand what I just explained? If not, kindly go over it again before you proceed.

We aren’t done. The explanation above will only log “I am a boy” to the console once. Our plan is to log it to the console three times. This is where a loop is different from “if statement”. After getting to line 11, JavaScript goes back to line 8. This time around, it skips part 2A and proceeds to check the condition in part 2B. It checks if the value of “i” is lesser than 4. Note that the value of “i” is no longer 1 but 2 (the post-increment on line 10 has increased the value of “i”). Since 2 is lesser than 4, JavaScript proceeds to run the codes in the third part of the loop. It executes the codes on line 9.

Line 9: “I am a boy” is logged to the console.

Line 10: There is a post-increment of “i”. Therefore, the value of increases from 2 to 3.

Line 11: We have the closing part of the curly braces.

Again, JavaScript goes back to line part 2B on line 8 to check if the value of “i” is lesser than 4. Since 3 is lesser than 4, it proceeds to the third part of the loop. It executes codes on line 9.

Line 9: “I am a boy” is logged to the console.

Line 10: There is a post-increment of “i”. Therefore, the value of increases from 3 to 4.

Line 11: We have the closing part of the curly braces.

Again, it goes back to line part 2B on line 8 to check if the value of “i” is lesser than 4. The condition is not true. 4 is not lesser than 4, therefore it terminates the loop.

Problem 11.2: Write a code to log “Good Morning” to the console 7 times

Solution:

```
7       <script type="text/javascript">
8           for(var i = 1; i < 8;){
9               console.log("Good Morning");
10              i++;
11          }
12      </script>
```

Fig 11.11

Don’t’ cram codes, there isn’t one solution to it. Any of the codes below will give you the same result.

```
7       <script type="text/javascript">
8           for(var i = 3; i < 10;){
9               console.log("Good Morning");
10              i++;
11          }
12      </script>
```

Fig 11.12

```
7       <script type="text/javascript">
8           for(var i = 0; i < 7;){
9               console.log("Good Morning");
10              i++;
11          }
12      </script>
```

Fig 11.13

```
7       <script type="text/javascript">
8           for(var i = 0; i <= 6;){
9               console.log("Good Morning");
10              i++;
11          }
12      </script>
```

Fig 11.14

I wrote three variations of the codes, your task is to write three other variations of the codes so you have the result when you run it on the browser. Ensure you do it.

Let's go deeper.

I will use the loop to display number 0-3, that is 0, 1, 2, 3.

Here is the solution.

```
7       <script type="text/javascript">
8           for(var i = 0; i < 4;){
9               console.log(i);
10              i++;
11          }
12      </script>
```

Fig 11.15

Explanation: I initialized variable "i" to 0 in part 2A of the "for loop".

I checked if the value of "i" is lesser than 4 in part 2B. Since the condition is true, it moves to the third part of the loop.

I logged the value of "i" -0-to the console on line 9.

Line 10 increases the value of "i" by 1, so "i" becomes 1.

When it gets to the line 11, it returns to part 2B on line 8 to check if the value of "i" is lesser than 4, since 1 is lesser than 4, it proceeds to the third part of the loop.

The value of i-1- is logged to the console on line 9.

Line 10 increases the value of "i" by 1, so "i" becomes 2.

When it gets to the line 11, it returns to part 2B on line 8 to check if the value of "i" is lesser than 4, since 2 is lesser than 4, it proceeds to the third part of the loop.

The value of i-2- is logged to the console on line 9.

Line 10 increases the value of "i" by 1, so "i" becomes 3.

When it gets to line 11, it returns to part 2B on line 8 to check if the value of "i" is lesser than 4, since 3 is lesser than 4, it proceeds to the third part of the loop.

The value of i-3- is logged to the console on line 9.

Line 10 increases the value of "i" by 1, so "i" becomes 4.

When it gets to the line 11, it returns to part 2B on line 8 to check if the value of "i" is lesser than 4. The condition is false since 4 is not lesser than 4, therefore it terminates the loop.

I hope you've been enjoying the explanation of the "for loop"

Every part of the codes is important. I will make two adjustments to the codes in fig 11.15 and we will consider the result together.

What will be the result of the codes below?

```
7   <script type="text/javascript">
8       for(var i = 5; i < 4;){
9           console.log(i);
10          i++;
11      }
12  </script>
```

Fig 11.16

a. It will result in infinite loop
b. It will log 0-3 to the console
c. Nothing will be logged to the console

Answer: c. Nothing will be logged to the console

Explanation: On line 8, we initialized variable "i" to 5 and check if the value is lesser 4. Since 5 isn't lesser than 4, then it won't proceed to the third part of the loop and the loop is terminated.

What will be the output of the codes below?

```
7   <script type="text/javascript">
8       for(var i = 0; i < 4;){
9           console.log(i);
10      }
11  </script>
```

Fig 11.17

a. It will result in infinite loop
b. It will log 0-3 to the console
c. Nothing will be logged to the console

Answer: a. It will result in an infinite loop.

Explanation: On line 8, we initialized variable “i” to 0 and check if the value is lesser 4. The condition is true and it proceeds to the third part of the loop.

The value of “i” -0- is logged to the console on line 9. We have the closing part of the curly braces on line 10.

When it gets to the line 10, it returns to part 2B on line 8 to check if the value of “i” is lesser than 4. The value of “i” is still 0 (Since the i++ has been removed). The condition is true since “i” -0- is lesser than 4, then it proceeds to the third part of the loop, logs 0 to the console on line 9, jumps to the part 2B of the loop again. The cycle continues till eternity because the value of “i” -0- will always be lesser than 4.

Problem 11.3 Write a code to log number 1-10 (1,2, 3, 4, 5, 6, 7 8, 9, 10) to the console

Solution:

```
7     <script type="text/javascript">
8         for(var i = 1; i < 11;){
9             console.log(i);
10            i++;
11        }
12    </script>
```

Fig 11.18

Problem 11.4: Write a code to log number 10-1 (10, 9, 8, 7, 6, 5, 4, 3, 2, 1) to the console

Solution:

```
7     <script type="text/javascript">
8         for(var i = 10; i >= 0;){
9             console.log(i);
10            i--;
11        }
12    </script>
```

Fig 11.19

Problem 11.5: Write a code to display even numbers between 1-10 (2, 4, 6, 8).

Solution:

```
7       <script type="text/javascript">
8           for(var i = 2; i < 10;){
9               console.log(i);
10              i+=2;
11          }
12      </script>
```

Fig 11.20

Using loop with innerHTML/innerText

You won't use console to communicate with your users. It is a common practice to make your loop communicate directly with a HTML element.

Consider this

```
7       <script type="text/javascript">
8           for (var i = 1; i < 4;) {
9               console.log(i);
10              i++;
11          }
12      </script>
```

Fig 11.21

The codes in fig 11.21 will log 1,2,3 to the console. Your task is to display the same result in a div instead of logging it to the console.

Are you done? Now, let's do it together.

Probably you tried this solution

```
6  <body>
7      <div id="result"></div>
8      <script type="text/javascript">
9          for (var i = 1; i < 4;) {
10             result.innerHTML = i;
11             i++;
12         }
13     </script>
14 </body>
```

Fig 11.22

And realized only 3 is displayed in the div. Where is 1 and 2? What's going? It worked perfectly in the console. Why isn't it working now?

To understand what is going on, consider the codes in fig 11.23

```
6  <body>
7      <div id="display"></div>
8      <script type="text/javascript">
9          display.innerHTML = 7;
10         display.innerHTML = 12;
11     </script>
12 </body>
```

Fig 11.23

Let's analyze the codes above.

When the browser gets to line 9, it sets the innerHTML of "display" to 7. When it gets to line 10, it sets the innerText of "display" to 12. This means that 12 overrides 7 as the innerHTML of "display". The fact that the browser reads our codes very fast means we only see 12 as the innerHTML of "display" when we run it on the browser.

This is the exact problem we have with the codes in fig 11.22

Let's analyze fig 11.22 together

On line 9, we initialized variable "i" to 1 and check if the value is lesser 4, which is true. It proceeds to the third part of the loop. The value of "i"-1- is set as the innerHTML of "result". The value of "i" is increased by 1 on line 11 which makes "i" 2.

It returns to part 2B of line 9 to check if the value of “i” is lesser than 4. Since 2 is lesser than 4, it proceeds to the third part of the loop. It sets the innerHTML of “result” to 2. This makes 2 overrides 1 as the innerHTML of “result”. The value of “i” is increased by 1 on line 11 which makes “i” 3.

It returns to part 2B of line 9 to check if the value of “i” is lesser than 4. Since 3 is lesser than 4, it proceeds to the third part of the loop. It sets the innerHTML of “result” to 3. This makes 3 overrides 2 as the innerHTML of “result”. The value of “i” is increased by 1 in line 11 which makes “i” 4.

It returns to part 2B of line 9 to check if the value of “i” is lesser than 4. Since 4 is not lesser than 4, it terminates the loop.

This is the reason we have 3 when we run the codes in fig 11.22 on the browser.

To fix it, we need to find a way of keeping the innerHTML of “result” intact so that new values won’t override old values.

Here is the solution:

```
 6  <body>
 7      <div id="result"></div>
 8      <script type="text/javascript">
 9          for (var i = 1; i < 4;) {
10              result.innerHTML = result.innerHTML + i;
11              i++;
12          }
13      </script>
14  </body>
```

Fig 11.24

Explanation:

On line 9, we initialized variable “i” to 1 and check if the value is lesser 4, which is true. It proceeds to the third part of the loop.

On line 10, it concatenates result.innerHTML -recall that I stated in chapter seven that JavaScript always treats any value gotten using any of these three output commands as string: value, innerText, innerHTML- with “i" and assign it to result.innerHTML. Therefore it concatenates an empty string with 1 and set it in result.innerHTML.

The value of “i” is increased by 1 on line 11 which makes “i” 2.

It returns to part 2B of line 9 to check if the value of “i” is lesser than 4. Since 2 is lesser than 4, it proceeds to the third part of the loop.

On line 10, it concatenates result.innerHTML -empty string + 1(recall that in chapter seven, I stated that whenever JavaScript concatenates anything with a string, it treats the result as a string)-with 2 and assign it to result.innerHTML.

The value of "i" is increased by 1 in line 11 which makes "i" 3.

It returns to part 2B of line 9 to check if the value of "i" is lesser than 4. Since 3 is lesser than 4, it proceeds to the third part of the loop.

On line 10, it concatenates result.innerHTML -the result of empty string + 1 + 2- with 3 and assign it to result.innerHTML.

The value of "i" is increased by 1 in line 11 which makes "i" 4.

It returns to part 2B of line to check if the value of "i" is lesser than 4. Since 4 is not lesser than 4, it terminates the loop.

We can shorten the code using this shorthand assignment operator (refer to chapter six):

```
 6  <body>
 7      <div id="result"></div>
 8      <script type="text/javascript">
 9          alert(typeof(result.innerHTML));
10          for (var i = 1; i < 4;) {
11              result.innerHTML += i;
12              i++;
13          }
14      </script>
15  </body>
```

Fig 11.25

Although it works fine. I want each of the numbers to be on a line, so I will add the break tag.

```
 6  <body>
 7      <div id="result"></div>
 8      <script type="text/javascript">
 9          alert(typeof(result.innerHTML));
10          for (var i = 1; i < 4;) {
11              result.innerHTML += i+ "<br>";
12              i++;
13          }
14      </script>
15  </body>
```

Fig 11.26

“break” and “continue”

These two key words can be said to be opposite of each other.

A short story

There are 100 boxes in a bag, 50 are labelled blue and 50 are labelled red. You were told there is a gold in only one of the boxes.

First scenario: You opened the first box, no gold. You opened the second one and found a gold in it. Ideally, you should stop the search since there is only one box containing gold. This is how “break” keyword works.

The explanation of the “break” keyword is: You have seen what you are looking for, stop searching, stop disturbing yourself.

Second scenario: Someone gave you a hint that the gold is in one of the boxes labelled red. It means once you pick a box, the first thing you do is to check the label on the box to see if it is red or blue. What do you do when it’s blue? You throw it away of course and pick the next box. You won’t bother yourself opening it. This is how the “continue” keyword works.

The explanation of the continue keyword is: This isn’t close to what you are looking for, skip it and jump to the next.

I would have named it “skip” instead of “continue” if I were in the position to name it. The “continue” keyword works like skip and it may be helpful to see it as such.

The “break” keyword

The codes below will only log 0 to the console.

```
7     <script type="text/javascript">
8         for(var i = 0; i < 4;){
9             console.log(i);
10            break;
11            i++;
12        }
13    </script>
```

Fig 11.27

Explanation: On line 8, we initialized variable “i” to 0 and check if the value is lesser 4, which is true. It proceeds to the third part of the loop. The value of “i”-0- is logged to the console on line 9. The “break” keyword is on line 10. Immediately it sees the break keyword, it terminates the loop.

Problem 11.6: What will be the output of the codes below?

```
7     <script type="text/javascript">
8         for (var i = 1; i < 4;) {
9             console.log(i);
10            if (i == 2) {
11                break;
12            }
13            i++;
14        }
15    </script>
```

Fig 11.28

a. It will log 1 to the console
b. It will log 1 and 2 to the console
c. It will log 1, 2, 3 to the console
d. Nothing will be logged to the console
e. It will result in infinite loop

Answer: b. It will log 1 and 2 to the console.

Explanation:

Line 8: “i” is initialized and its value set to 1. JavaScript checks if “i” is lesser than 4.

Line 9: “i" (1) is logged to the console

Line 10: JavaScript checks if the value of “i" is equal to 2. Since the value of “i" isn’t 2, The body of the “if statement” is ignored. JavaScript jumps to line 13.

Line 13: “i” is incremented by 1 and it becomes 2. It returns to part 2B on line 8.

Line 8: JavaScript checks if “i” is lesser than 4.

Line 9: “i" (2) is logged to the console.

Line 10: JavaScript checks if the value of “i" is equal to 2, which is true. JavaScript reads the codes in the body of the “if statement”. It sees the “break” keyword and immediately terminates the loop.

Continue

At this point, it is important to make a little adjustment to our codes before explaining “continue”. Kindly note that the codes in the two images below work the same way and will produce the same result.

```
 7      <script type="text/javascript">
 8          for (var i = 1; i < 4;) {
 9              console.log(i);
10              i++;
11          }
12      </script>
```

Fig 11.29

```
 7      <script type="text/javascript">
 8          for (var i = 1; i < 4; i++) {
 9              console.log(i);
10          }
11      </script>
```

Fig 11.30

Do you notice that the i++ on line 10 in fig 11.29 has been moved up to line 8 in fig 11.30 . I will name the i++ in fig 11.30, part 2C. The i++ is only up syntactically. This doesn’t affect the behavior of the program. Although it is up, I want you to picture it the way it was written before (fig 11.29). The i++ in part 2C is read after the blocks of codes in the third part of the

loop is read and executed. You need to get used to the changes made as that's what will be used in the remaining part of this textbook. You will see the syntax in fig 11.30 in almost all programs and applications. I only started with the syntax in fig 11.29 because it is easier to understand.

Later in this chapter, I will explain the reason it is important to make this shift in syntax at this point.

Let's proceed to explain "continue".

Whenever JavaScript sees "continue", it breaks the current iteration and moves to the next iteration.

Example:

```
7   <script type="text/javascript">
8       for (var i = 1; i < 4; i++) {
9           console.log("hello");
10          continue;
11          console.log("hi");
12      }
13  </script>
```

Fig 11.31

On line 8, "i" is initialized and its value set to 1. JavaScript checks if "i" is lesser than 4.

On line 9, "hello" is logged to the console.

On line 10, JavaScript sees the "continue" keyword. It immediately skips all the codes between that spot and the end of the loop to start the next iteration. Therefore, it skips the codes between line 10 and line 12 and jumps back up to line 8. It moves to part 2C of line 8 to increment "i" by 1, "i" becomes 2. It then moves to part 2B to check if the value of "i" is lesser than 4. Since 2 is lesser than 4, it proceeds to the third part of the loop.

On line 9, "hello" is logged to the console again. On Line 10, JavaScript sees the "continue" keyword. It skips the codes between line 10 and 12, moves to part 2C to make "i" 3, proceeds to part 2B to check if "i" is lesser than 4. Since 3 is lesser than 4, it proceeds to the third part of the loop.

On line 9, "hello" is logged to the console again. On line 10, JavaScript sees the "continue" keyword. It skips the codes between line 10 and 12, moves to part 2C to make "i" 4, proceeds to part 2B to check if "i" is lesser than 4. Since 4 is not is lesser than 4, it terminates the loop.

In the console, we will see three “hello”. “hi” is not seen in the console because of the continue keyword.

Two tasks for you before we leave “continue”.

Problem 11.7: What will be the output of the codes below?

```
7     <script type="text/javascript">
8         for (var i = 1; i < 4;) {
9             console.log("hello");
10            continue;
11            console.log("hi");
12            i++;
13        }
14    </script>
```

Fig 11.32

a. It will result in infinite loop
b. “hello” and “hi” will be logged to the console three times each
c. “hello” is logged to the console three times.
d. Nothing will be logged to the console

Answer: a. It will result in infinite loop.

Explanation:

On line 8, “i” is initialized and its value set to 1. JavaScript checks if “i” is lesser than 4. This is true since 1 is lesser than 4. It moves to the third part of the loop.

On line 9, “hello” is logged to the console. JavaScript sees the “continue” keyword on line 10, so it skips every code from that spot (line 10) till the end of the loop (line 13). It jumps back up. Since there is no part 2C, it moves to part 2B to check if “i" is lesser than 4. 1 is lesser than 4, so it moves to the third part of the loop, logs “hello” to the console, skips codes between line 10 to 13, jumps back up to start another cycle till infinity because the value of “i" (1) will always be lesser than 4.

Problem 11.8: What will be the output of the codes below?

```
7    <script type="text/javascript">
8        for (var i = 1; i < 4; i++) {
9            if (i==2) {
10               continue;
11           }
12           console.log(i);
13       }
14   </script>
```

Fig 11.33

a. 1 will be logged to the console
b. 1, 2 will be logged to the console
c. 1, 3 will be logged to the console
d. 1, 2, 3 will be logged to the console

Answer: c. 1, 3 will be logged to the console.

Explanation:

Line 8: “i” is initialized and its value set to 1. JavaScript checks if “i” is lesser than 4. Since 1 is lesser than 4, it moves to the third part of the loop.

Line 9: Checks if “i" is 2. Since this isn’t true, the body of the “if block” is ignored. Proceeds to line 12.

Line 12: 1 is logged to the console. It moves to part 2C on line 8.

Line 8 part 2C: increments “i" by 1 to make it 2.

Line 8 part 2B: Checks if “i" is lesser than 4. Since 2 is lesser than 4, it moves to the third part of the loop.

Line 9: Checks if “i" is 2. Since this is true, it sees the “continue” keyword in the “if block” and skips the codes between line 10 and the end of the loop (line 13). Moves to part 2C line 8.

Line 8 part 2C: increments “i" by 1 to make it 3.

Line 8 part 2B: Checks if “i" is lesser than 4. Since 3 is lesser than 4, it moves to the third part of the loop.

Line 9: Checks if “i" is 2. Since this isn’t true, the body of the “if block” is ignored. Proceeds to line 12.

Line 12: 3 is logged to the console. It moves to part 2C line 8.

Line 8 part 2C: increments “i" by 1 to make it 4.

Line 8 part 2B: Checks if “i" is lesser than 4. Since 4 is not lesser than 4, it terminates the loop.

Array and loop

We have separately considered arrays, we have also considered loop. Let’s combine them together.

What will the output of the codes below?

```
7    <script type="text/javascript">
8        var students = ["Busola", "Winning", "Taye", "Kehinde"];
9        for (var i = 1; i < 3; i++) {
10           console.log(students[1]);
11       }
12   </script>
```

Fig 11.34

a. It will log “Winning” to the console twice.
b. It will log “Winning” to the console three times.
c. It will log “Busola” to the console three times.
d. It will log “Busola” and “Winning” to the console.

Answer: a. It will log “Winning” to the console twice.

Explanation: On line 8, I created an array and inserted 4 values in it.

Line 9: “i” is initialized and its value set to 1. JavaScript checks if “i” is lesser than 3. Since 1 is lesser than 3, it moves to the third part of the loop.

Line 10: It will log students[1] (“Winning”) to the console.

It moves to part 2C to increase the value of “i” by 1 to make “i" 2. It moves to part 2B to check if “i" is lesser than 3. Then it moves to the third part of the loop.

Line 10: It will log students[1] (“Winning”) to the console the second time.

It moves to part 2C to increase the value of “i” by 1 to make “i" 3. It moves to part 2B to check if “i" is lesser than 3. The loop is terminated.

Problem 11.9:

What will be the output of this code?

```
7     <script type="text/javascript">
8         var students = ["Busola", "Winning", "Taye", "Kehinde"];
9         for (var i = 1; i < 3; i++) {
10            console.log(students[i]);
11        }
12    </script>
```

Fig 11.35

a. It will log “Winning” to the console twice.
b. It will log “Winning” to the console three times.
c. It will log “Busola” to the console three times.
d. It will log “Winning” and “Taye” to the console.

Answer: d. It will log “Winning” and “Taye” to the console.

Explanation: On line 8, I created an array and inserted 4 values in it.

Line 9: “i” is initialized and its value set to 1. JavaScript checks if “i” is lesser than 3. Since 1 is lesser than 3, it moves to the third part of the loop.

Line 10: It will log students[i] to the console. Since the value of “i" is 1, students[i] becomes students[1], so it logs “Winning” to the console.

It moves to part 2C to increase the value of “i” by 1 to make “i" 2. It moves to part 2B to check if “i" is lesser than 3. Then it moves to the third part of the loop.

Line 10: Since the value of “i" is 2, students[i] becomes students[2], so it logs “Taye” to the console.

It moves to part 2C to increase the value of “i” by 1 to make “i" 3. It moves to part 2B to check if “i" is lesser than 3. The loop is terminated.

Problem 11.10: Adjust the codes in fig 11.35 to display every name in the array.

Solution:

```
7     <script type="text/javascript">
8         var students = ["Busola", "Winning", "Taye", "Kehinde"];
9         for (var i = 0; i < 4; i++) {
10            console.log(students[i]);
11        }
12    </script>
```

Fig 11.36

I was able to do this by changing the 3 on line 9 to 4, but there is a problem with this approach, if I increase the numbers of items in the students array, then I need to increase the number at part 2B of the loop in order to display every name in the array.

```
<script type="text/javascript">
    var students = ["Busola", "Winning", "Taye", "Kehinde",
                    "Peter", "James", "John"];
    for (var i = 0; i < 7; i++) {
        console.log(students[i]);
    }
</script>
```

Fig 11.37

Most times, you will be working with dynamic arrays. The number of items in your array will change at runtime. Users will add or remove items from your array. You need to write your program to cope with these changes.

The code above (fig 11.37) can be rewritten as

```
<script type="text/javascript">
    var students = ["Busola", "Winning", "Taye", "Kehinde",
                    "Peter", "James", "John"];
    for (var i = 0; i < students.length; i++) {
        console.log(students[i]);
    }
</script>
```

Fig 11.38

The length of the students array is 7. Therefore fig 11.37 and fig 11.38 will produce the same result. The advantage of using the loop in fig 11.38 over fig 11.37 is that, if more items are added to array in fig 11.37, we need to increase the number at part 2B of the loop to display the new items while we don't need to touch the codes in fig 11.38 when more items are added to the array or if part or all the current items in the array are removed.

Problem 11.11: Adjust the loop in fig 11.38 to log "Busola", "Taye", "Peter" and "John" to the console.

Answer:

```
<script type="text/javascript">
    var students = ["Busola", "Winning", "Taye", "Kehinde",
                    "Peter", "James", "John"];
    for (var i = 0; i < students.length; i+=2) {
        console.log(students[i]);
    }
</script>
```

Fig 11.39

Problem 11.12: Adjust the codes in fig 11.38 to display every name in the array except the name at index 2

Solution:

```
<script type="text/javascript">
    var students = ["Busola", "Winning", "Taye", "Kehinde",
                    "Peter", "James", "John"];
    for (var i = 0; i < students.length; i++) {
        if (i==2) {
            continue;
        }
        console.log(students[i]);
    }
</script>
```

Fig 11.40

Problem 11.13

Consider the codes below

```
<script type="text/javascript">
    var students = ["Busola", "Winning", "Taye", "Kehinde",
                    "Peter", "James", "John", "Winning", "Tunde",
                    "Winning"];
</script>
```

Fig 11.41

Use for loop to log every name in the array to the console except Winning.

Probably you came up with this

```
<script type="text/javascript">
    var students = ["Busola", "Winning", "Taye", "Kehinde",
                    "Peter", "James", "John", "Winning", "Tunde",
                    "Winning"];
        for (var i = 0; i < students.length; i++) {
            if (i == 1 || i == 7 || i ==9) {
                continue;
            }
            console.log(students[i]);
        }
</script>
```

Fig 11.42

While the codes will produce the desired result. It has one problem. If another “Winning” is added to the array, I need to adjust the body of the loop so the new “Winning” is not logged to the console. I also need to adjust the codes in the loop if the index of “Winning” is changed. The code isn’t flexible.

Imagine a new name is added to the beginning of the array such that we now have this:

[“Jide”, "Busola", "Winning", "Taye", "Kehinde", "Peter", "James", "John", "Winning", "Tunde", "Winning"];

In this case the indexes of the “Winning” in the array are no longer 1, 7, 9 but now 2, 8, 10. So we have to adjust the codes in the body of the loop so that “Winning” is not logged to the console.

You need to write your loop to cope with the dynamics of your array. Here is a better solution

```
7   <script type="text/javascript">
8       var students = ["Busola", "Winning", "Taye", "Kehinde",
9                       "Peter", "James", "John", "Winning", "Tunde",
10                      "Winning"];
11          for (var i = 0; i < students.length; i++) {
12              if (students[i]=="Winning") {
13                  continue;
14              }
15              console.log(students[i]);
16          }
17  </script>
```

Fig 11.43

The codes above will ignore all the “Winning” in the array irrespective of the index.

SECTION TWO

ES6

ES6 is an updated version of JavaScript. It was released in 2015. In this section, we will take a look at some of the things added. In some situations, you aren't going to learn new logic, just another syntax to do the same thing you have been doing while in other situations, you will learn entirely new things. For instance, the concept of classes and constructor may look entirely new to you.

CHAPTER TWELVE

VARIABLE DECLARATION IN ES6

In chapter two, you learnt how to declare variables using the "var" keyword. Two other keywords were introduced in es6 for declaring variables, they are: let and const.

To start with, we will look at the differences between the "var" keyword and the "let" keyword.

Differences between "var" and "let"

1. Redeclaration: While you can redeclare variables created using the "var" keyword, you cannot redeclare variables created using the "let" keyword.
   Consider this:

```
 7    <script type="text/javascript">
 8        var a = 5;
 9        var a = 7;
10        alert(a);
11    </script>
```

Fig 12.1

The browser will read this code from top to bottom. On line 8, a container is created, this container is labelled "a" and 5 is inserted into the container. On line 9, another container is created, this container is also labelled "a", 7 is inserted into the new container. This means we have two containers with the same name. On line 10, we alert the value of variable "a". Which of the two containers should JavaScript pick? It becomes confused and eventually picks the most recent container. That means the first container (line 8) become redundant after line 9. The "let" keyword was created to remove this confusion and redundancy.

```
 7    <script type="text/javascript">
 8        let a = 5;
 9        let a = 7;
10        alert(a);
11    </script>
```

Fig 12.2

The code above will throw an error.

On line 8, a container is created, this container is labelled "a" and 5 is inserted into the container. When JavaScript sees that you are creating another container with the same label on line 9, it throws an error.

2. Scope: ‘var’ is function scoped while ‘let’ is block scoped. What function scoped means is that when a variable is created with ‘var’ within a function, it cannot be used outside the function (local variable). What block scoped means is that, when a variable is created with ‘let’ within a block, it cannot be used outside its block.
What is a block? A block in this sense can be said to be a set of codes embedded within curly braces (this definition is oversimplification but it is true in most circumstances).
Example:

```
7   <script type="text/javascript">
8       function myFunc() {
9           var a = 5;
10          if (a == 5) {
11              var b = 7;
12          }
13          alert(a * b);
14      }
15      myFunc();
```

Fig 12.3

The code above will alert 35.
On the other hand, a variable created with ‘let’ cannot be used outside its block
The block of variable “a” in fig 12.3 is the function (the curly braces that holds the initialization of variable “a” starts on line 8 and ends on line 14).
The block of variable “b” in fig 12.3 is the “if block” (the curly braces that holds the initialization of variable “b” starts on line 10 and ends on line 12).
The code below will throw an error

```
7   <script type="text/javascript">
8       function myFunc() {
9           let a = 5;
10          if (a == 5) {
11              let b = 7;
12          }
13          alert(a * b);
14      }
15      myFunc();
16  </script>
```

Fig 12.4

This is because variable “b” was created with the “let” keyword and it cannot be accessed outside its block. The block of variable “b” ends on line 12 and we tried accessing it on line 13.

Difference between let and const

“const” stands for constant. “const” behave like “let” in terms of declaration and scoping but with one added feature. The value of a constant cannot be changed.

The code below will alert 7

```
7    <script type="text/javascript">
8        let a = 5;
9        a = 7;
10       alert(a);
11   </script>
```

Fig 12.5

While the code below will throw an error

```
7    <script type="text/javascript">
8        const a = 5;
9        a = 7;
10       alert(a);
11   </script>
```

Fig 12.6

I set the value of constant “a” to 5 on line 8. On line 9, I tried changing the value of constant “a”. This throws an error.

By convention, you write constants in uppercase.

```
7    <script type="text/javascript">
8        const NAME = "Taye";
9    </script>
```

Fig 12.7

CHAPTER THIRTEEN

TEMPLATE LITERAL

A template literal is created using backtick (``). It is mainly used with strings and can be used in place of concatenation.

Uses of template literals

1. When your string contains a quote, or apostrophe

```
 7      <script type="text/javascript">
 8          let myString = 'I'm coming';
 9
10      </script>
```

Fig 13.1

The codes above will throw an error. The reason for the error is this: There are three apostrophes on line 8. The first is used to open the string, the second is used to close the string. Everything after the second apostrophe, including the third apostrophe is seen as bugs.

The bug can be removed by adding a backslash before the second apostrophe in order to escape it. Another solution is to change the quotes to a double quote so it isn't confused with the apostrophe in the text. **The third solution is to use a template literal**. Here are the three solutions.

```
 7      <script type="text/javascript">
 8          let myString = 'I\'m coming';//first
 9          let str = "I'm coming";//second
10          let myStr = `I'm coming`;//third
11      </script>
```

Fig 13.2

2. To create multiline string: Before the introduction of ES6, you need to use the new line character to create strings that span through multiple lines. With ES6, the template literal can easily help with that.

```
 7      <script type="text/javascript">
 8          let myString = 'Welcome \n here'; //before ES6
 9          let secondString = `Welcome
10          here`; // after ES6
11      </script>
```

Fig 13.3

3. String substitution: This produces a result similar to concatenation. The dollar sign and a curly braces was introduced for string substitution.

```
7    <script type="text/javascript">
8        let name = "Taye";
9        console.log("My name is "+name); //before es6
10       console.log(`My name is ${name}`); //after es6
11   </script>
```

Fig 13.4

Notice the backticks in line 10.

Anything written within the dollar and curly braces syntax is seen as a valid JavaScript code, therefore, when JavaScript sees ${name} on line 10, it inserts the value of variable name.

CHAPTER FOURTEEN

DESTRUCTURING

We can destructure an array or an object. Destructuring allows us to easily assign one or more values of an array or object to variables.

Array destructuring

You use square braces to destructure an array and curly braces to destructure an object. Imagine we have a big array and we want to assign one or more values of the array to variables.

```
7   <script type="text/javascript">
8       // Before destructuring was introduced
9       let names = ["Taye", "Kenny", "Idowu"];
10      let firstPerson = names[0];
11      let secondPerson = names[1];
12      console.log(firstPerson); //"Taye"
13      console.log(secondPerson) //"Kenny"
14
15      // After destructuring was introduced
16      let [firstName, secondName] = names;
17      console.log(firstName); //"Taye"
18      console.log(secondName); //"Kenny"
19  </script>
```

Fig 14.1

The “names” array was destructured on line 16. The first value in the names array is assigned to the first item in the square braces (firstName). The second value in the names array is assigned to the second item in the square braces (secondName). **The order matters.**

There may be situations where you want to assign a given value in an array to a variable without assigning the preceding values to variables. For instance, I wish to assign the first and third values in the names array to variables without assigning the second value to a variable.

```
<script type="text/javascript">
    let names = ["Taye", "Kenny", "Idowu"];
    let [firstName, , thirdName] = names;
    console.log(firstName); //"Taye"
    console.log(thirdName); //"Idowu"
</script>
```

Fig 14.2

There are two commas in the codes in fig 14.2. The second comma means we are aware a value exists here but we are not interested in the value.

Object destructuring

Object destructuring allows us to assign one or more values of an object to variables. Unlike array destruturing where values are assigned to variables based on their position in an array, it isn't the same for objects. In objects, values are assigned using their keys. The key name is automatically used as the variable name.

```
<script type="text/javascript">
    let person = {name: "Taye", dept: "Software", school: "SQI"};
    let {name, school} = person;
    console.log(name); //"Taye"
    console.log(school); //"SQI"
</script>
```

Fig 14.3

If I tried destructuring a key that does not exist in the object, it returns undefined. Here is an example below.

```
<script type="text/javascript">
    let person = {name: "Taye", dept: "Software", school: "SQI"};
    let {age} = person;
    console.log(age); // undefined
</script>
```

Fig 14.4

There are situations where you want your variable name to bear a name other than the key name of the object. To do this, the key name is first used to extract the value from the object, which is then assigned to a new name using the colon symbol (:).

```
7    <script type="text/javascript">
8        let person = {name: "Taye", dept: "Software", school: "SQI"};
9        let {name:myNewName} = person;
10       console.log(myNewName); //"Taye"
11   </script>
```

Fig 14.5

Line 9: “name” is destrutured from “person”. The colon sign is used to assign the value of “name” to “myNewName”.

CHAPTER FIFTEEN

SPREAD OPERATOR

The spread operator spreads out the items in an iterable. The syntax for the spread operator is three dots (…).

Spreading an array

A short story

My mother has some plates in a basket. One day, she asked my twin sister to count the number of plates in the basket. My twin sister placed one of the plates on the floor, then placed the second on it till she stacked all the plates on one another.

Basket of plates = Array of values

My twin sister = Spread operator

This is how spread operator works but with an exception. In the story above, the basket will be empty when she is done counting the plates in the basket, but in the case of JavaScript, the array of values will still be intact even after the action of the spread operator.

```
7     <script type="text/javascript">
8         let myArray = [4, 6, 8];
9         console.log(...myArray);// 4 6 8
10        console.log(myArray);// [4, 6, 8]
11    </script>
```

Fig 15.1

On line 9, do you notice that the result of the spread operator does not contain square braces? The result is 4 6 8, not [4, 6, 8]. This is because the spread operator only takes the values in the array and not the whole container.

On line 10, I logged myArray to the console so you can see that myArray is still intact after the action of the spread operator.

It is a common practice to place the result of the spread operator in another container.

Example:

```
<script type="text/javascript">
    let myArray = [4, 6, 8];
    let newArray = [1, 2, 3, ...myArray, 10, 12];
    console.log(newArray); //[1, 2, 3, 4, 6, 8, 10, 12]
</script>
```

Fig 15.2

Spreading an object

You spread an object same way you spread an array.

```
<script type="text/javascript">
    let myObject = {name: "Taye", dept: "software"};
    let newObject = {school: "SQI", ...myObject};
    console.log(newObject); // {school: "SQI", name: "Taye", dept: "software"}
</script>
```

Fig 15.3

We want to consider two scenarios.

The take home from the two scenarios is that: The new value will always replace the old one whenever they have the same key.

Scenario 1:

```
<script type="text/javascript">
    let obj = {name: "Taye", dept: "software"};
    let newObj = {name: "Kenny", school: "SQI", ...obj};
    console.log(newObj); // {name: "Taye", school: "SQI", dept: "software"}
</script>
```

Fig 15.4

The value of the “name” property in newObj is “Taye”.

Scenario 2:

```
<script type="text/javascript">
    let obj = {name: "Taye", dept: "software"};
    let newObj = {...obj, name: "Kenny", school: "SQI"};
    console.log(newObj); // {name: "Kenny", school: "SQI", dept: "software"}
</script>
```

Fig 15.5

The value of the “name” property in newObj is “Kenny”.

CHAPTER SIXTEEN

REST OPERATOR

Although the syntax for rest operator is the same as spread operator, they work differently.

'Rest' means 'the rest', in other words, remainder. Before there can be a remainder, firstly, there must be a whole item. Secondly, part of the whole item must be taken. Whatever remains after part of a whole is taken, is known as remainder. This is what the rest operator does.

A good understanding of destructuring will make it easier to understand rest operator.

Rest operator with an array

```
7   <script type="text/javascript">
8       let myArray = [3, 4, 5, 6, 7, 8, 9, 10];
9       let [a, b, ...c] = myArray;
10      console.log(c); // [5, 6, 7, 8, 9, 10]
11  </script>
```

Fig 16.1

On line 9, I destructured 'myArray' to assign the first value to variable "a" and the second value to variable "b". I assigned the rest to variable c using the rest operator (…).

On line 9 of fig 16.1, I took part of myArray (I took the first and second value), the rest is assigned to variable c.

Problem 16.1: What will be the output of the codes below?

```
7   <script type="text/javascript">
8       let names = ["Peter", "James", "John"];
9       let [name, ...myNames] = names;
10      console.log(myNames);
11  </script>
```

Fig 16.2

a. "James"
b. ["James", "John"]
c. ["Peter", "James", "John"]
d. "John"

Answer: b. ["James", "John"]

Explanation: I only took the first value, other values in the array were assigned to "myNames".

Rest operator with object

To use rest operator with an object, you destructure the key you want, while the rest is assigned to a variable.

Example:

```
7     <script type="text/javascript">
8         let person= {name: "Taye", dept: "software", school: "SQI"};
9         let {dept, ...newPerson} = person;
10        console.log(newPerson); //{name: "Taye", school: "SQI"}
11    </script>
```

Fig 16.3

CHAPTER SEVENTEEN

ARROW FUNCTIONS

A shorter way of creating functions is to use arrow functions. To do this, you save functions as variables (in JavaScript, functions are variables).

The function keyword is removed while using arrow functions.

```
<script type="text/javascript">
    // Without arrow function
    function myFunc() {
        console.log("Hello world");
    }

    // with arrow function
    let myFunc = () =>{
        console.log(`Hello world`);
    }
```

Fig 17.1

It is a common practice to create functions using ‘const’ instead of ‘let’ so you don’t accidentally change the value of the function.

```
<script type="text/javascript">
    const myFunc = () =>{
        console.log(`Hello world`);
    }
</script>
```

Fig 17.2

Parameters can be added to arrow functions, the parameters are placed within parenthesis.

```
<script type="text/javascript">
    const myFunc = (name, course)=>{
        console.log(`My name is ${name} studying ${course}`);
    }
</script>
```

Fig 17.3

You are only allowed to remove the parenthesis if and only if your function receives only one parameter.

```
7       <script type="text/javascript">
8           const myFunc = name =>{
9               console.log(`My name is ${name}`);
10          }
11      </script>
```

Fig 17.4

The parenthesis should not be removed when your function receives multiple parameters or when your function receives no parameter.

```
7       <script type="text/javascript">
8           const myFunc = () =>{
9               console.log(`Hello world`);
10          }
11          const displayInfo= (a, b)=>{
12              console.log(a * b)
13          }
14      </script>
```

Fig 17.5

Return in arrow functions

You can use the return keyword in arrow functions the same way you use it in regular functions (that is, ES5 functions).

```
7       <script type="text/javascript">
8           const myFunc = num =>{
9               return num * 3;
10          }
11          console.log(myFunc(5)); //15
12
13      </script>
```

Fig 17.6

ES6 provides us with a shorter syntax to return a value from a function without writing the return keyword. To do this, you need to remove the function's curly braces.

Example

```
7    <script type="text/javascript">
8        const myFunc = num => num * 3;
9        console.log(myFunc(5)); //15
10   </script>
```

Fig 17.7

Example:

```
7    <script type="text/javascript">
8        const myFunc = name => `My name is ${name}`;
9        console.log(myFunc("Taye")); //My name is Taye
10
11   </script>
```

Fig 17.8

CHAPTER EIGHTEEN

CLASSES AND CONSTRUCTORS

A short story

Mrs Bimbo is a building contractor that has many teams. She got a contract to build 100 look-alike houses. Immediately, she made a building plan. Whenever it's time to construct a new house, she would send the building plan to the team lead of any team of her choice. The team lead would assemble the necessary parameters and construct a house from the building plan.

Building plan = Class

House = Object

Team lead = Constructor

Mrs Bimbo = Programmer

As a building makes it easy to create many similar houses, so also, a JavaScript class makes it easy to create many similar objects. The team lead creates a house from the building plan, so also, a constructor creates an object from a class. When it's time to create a new object, you (programmer) trigger the constructor to create a new object from the class.

Class

A class is like a building plan. A building plan has all the rooms in the house, albeit on paper. Same way, a class contains all the properties and methods you want to give to your objects but it isn't the object.

To create a class, you write the class keyword, followed by the name of your class. By convention, you write a class name using Pascal case. You add properties and methods to the body of the class.

Methods are added to class same way you add them to object but adding properties is a bit different. You use the equal sign (=) instead of colon.

Example:

```
7     <script type="text/javascript">
8         class Student{
9             name= "Taye";
10            dept= "software";
11            displayHello(){
12                console.log("Hello world");
13            }
14        }
15    </script>
```

Fig 18.1

In the image above, the class name is Student. The class has two properties (name and dept) and one method (displayHello).

Object of a class

Recall that we created a class because we want to make object(s) from the class. Let's create an object from the class in fig 18.1. To create an object from a class, you trigger the constructor of the class. The constructor of the class creates a new object from the class. There are two questions here. First, where is the constructor? Second, how do we trigger the constructor?

Where is the constructor?

Every class has an inbuilt constructor but you can create yours. Later in this chapter, we will take a look into how to create a constructor in a class. I didn't place any constructor in class 'Student' in fig 18.1 but it has an inbuilt constructor. Let's trigger the inbuilt constructor.

How do we trigger the constructor?

The constructor of a class is triggered with the 'new' keyword.

Example:

```
7     <script type="text/javascript">
8         class Student{
9             name= "Taye";
10            dept= "software";
11            displayHello(){
12                console.log("Hello world");
13            }
14        }
15        let myObject = new Student;
16        console.log(myObject.name); // Taye
17        myObject.displayHello(); //Hello world
18    </script>
```

Fig 18.2

The 'new' keyword on line 15 triggers the constructor which creates an object from class 'Student' and save it in variable 'myObject'. Same way a house constructed from a building plan takes the shape of the building plan, an object constructed from a class takes the shape of the class. For this reason, variable 'myObject' is an object that contains all the properties and methods in class 'Student'.

By typing the 'new' keyword, I created a new instance of the class which returns an object. This object is saved as variable "myObject".

You can create as many objects as you want from your class.

The value of a property can be changed after an object is created from a class but this doesn't affect other objects created from the same class. It only affects the object in question.

Example:

```
7   <script type="text/javascript">
8       class Student{
9           name= "Taye";
10          dept= "software";
11          displayHello(){
12              console.log("Hello world");
13          }
14      }
15      let firstObject = new Student;
16      let secondObject = new Student;
17      firstObject.name = "Kenny";
18      console.log(firstObject.name); // Kenny
19      console.log(secondObject.name); // Taye
20  </script>
```

Fig 18.3

On line 15, object firstObject was created from class 'Student'. On line 16, object secondObject was created from class 'Student'. On line 17, the value of 'name' of firstObject was changed to Kenny. This change only affects firstObject, not other objects (secondObject) created from the class.

Setting the properties of a class with methods

A method of a class can set properties on the class.

```
7   <script type="text/javascript">
8       class Student{
9           name= "Taye";
10          dept= "software";
11          setSchool(){
12              this.school = "SQI";
13          };
14          displaySchool(){
15              console.log(`My school is ${this.school}`);
16          }
17      }
18      let person = new Student;
19      person.setSchool();
20      person.displaySchool(); // My school is SQI
21  </script>
```

Fig 18.4

Parameters can be added to methods of a class.

Example:

```
<script type="text/javascript">
    class Student{
        name= "Taye";
        dept= "software";
        setSchool(sch){
            this.school = sch;
        };
        displaySchool(){
            console.log(`My school is ${this.school}`);
        }
    }
    let firstPerson = new Student;
    let secondPerson = new Student;
    firstperson.setSchool("SQI");
    secondperson.setSchool("LAUTECH");
    firstperson.displaySchool(); // My school is SQI
    secondperson.displaySchool(); // My school is LAUTECH
</script>
```

Fig 18.5

User Defined Constructor

You can add your constructor to the class. A constructor is a method created with the constructor keyword. A constructor method is triggered like every other method with two exceptions.

1. You don't call on the constructor method directly
2. A constructor is triggered immediately you create a new instance of the class

```
7     <script type="text/javascript">
8         class Student{
9             name= "Taye";
10            dept= "software";
11            constructor(){
12                console.log("I am the constructor");
13            };
14            setSchool(sch){
15                this.school = sch;
16            };
17            displaySchool(){
18                console.log(`My school is ${this.school}`);
19            }
20        }
21        let person = new Student; //I am the constructor
22        person.constructor(); // Throws an error
23    </script>
```

Fig 18.6

Line 22 will throw an error because I triggered the constructor directly.

Line 21 logs “I am the constructor” to the console because the constructor is triggered immediately you create a new instance of the class.

We can take advantage of these attributes of the constructor.

```
7   <script type="text/javascript">
8       class Student{
9
10          constructor(){
11              this.name= "Taye";
12              this.dept= "software";
13          };
14          setSchool(sch){
15              this.school = sch;
16          };
17          displaySchool(){
18              console.log(`My school is ${this.school}`);
19          };
20          displayDept(){
21              console.log(`My department is ${this.dept}`);
22          }
23      }
24      let person = new Student; //I am the constructor
25      person.displayDept(); //My department is software
26
27  </script>
```

Fig 18.7

I created a new instance of "Student" class on line 24 which triggers the constructor. The constructor sets two properties (name and dept) on the "Student" class. On line 25, I triggered displayDept method. Therefore, "My department is software" is logged to the console.

Summary: The constructor is triggered when you create a new instance of a class.

Constructors with parameters.

Since a constructor is a method, parameters can be added to it to make the class dynamic and more powerful.

```
constructor(n, d){
    this.name= n;
    this.dept= d;
};
```

Fig 18.8

If constructors aren't triggered directly, how do we pass arguments to the parameters in the constructor?

We do this by passing the arguments to the class name as shown below.

```
let person = new Student("James", "Hardware");
```

Fig 18.9

Example:

```
<script type="text/javascript">
    class Student{

        constructor(n, d){
            this.name= n;
            this.dept= d;
        };
        displayInfo(){
            console.log(`I am ${this.name} studying ${this.dept}`);
        }
    }

    let firstPerson = new Student("James", "Hardware");
    let secondPerson = new Student("Titi", "English");
    firstPerson.displayInfo(); //I am James studying Hardware
    secondPerson.displayInfo(); //I am Titi studying English

</script>
```

Fig 18.10

Class inheritance

Sometimes, you want a class to have all the properties and methods of another class and even more.

I will explain this with Biology

A short story

All animals (be it goat, hen, Gorilla, Shark etc) are multicellular organisms, contains eukaryotic cells, can move, can reproduce. Therefore, if we create a class called animal, it will have two properties (multicellular and eukaryotic) and two methods (move and reproduce). If we have another class called mammal, it will also have two properties (multicellular and eukaryotic) and two methods (move and reproduce) since a mammal is a type of animal. In order words, we can say mammal is an extension of animal. Mammals have some things that differentiate them from other animals, mammals have hair and can breastfeed. This implies that the mammal class will have one more property (hairy) and method (breastfeed).

To create these two classes in JavaScript, we have two options.

Option 1:

- Create a class called ‘Animal’, give it two properties (multicellular and eukaryotic) and two methods (move and reproduce).
- Create another class called ‘Mammal’, give it three properties (multicellular, eukaryotic and hairy) and three methods (move, reproduce and breastfeed).

STOP! Don’t Repeat Yourself (DRY). Don’t use option 1.

Since a mammal is an extension of animal, why not write our codes in a way that depicts this? This leads us to option 2.

Option 2:

- Create a class called ‘Animal’, give it two properties (multicellular and eukaryotic) and two methods (move and reproduce).
- Create another class called ‘Mammal’ and make it an extension of the ‘Animal’ class. This makes it automatically have all the properties and methods in the ‘Animal’ class. Add one property (hairy) and method (breastfeed) to the ‘Mammal’ class.

In total, the mammal class has three properties (multicellular, eukaryotic and hairy) and three methods (move, reproduce and breastfeed). This makes the codes shorter and easier to maintain.

Please note that the Mammal is an extension of the animal class, not the other way. This makes the Mammal class inherits all the properties and methods of the Animal class while the Animal class does not inherit anything from the Mammal class.

In this case, the Animal class is known as the parent’s class while the Mammal class is known as the child’s class. A parent’s class can have as many children as possible while a child’s class can only have one parent class.

Let’s write the explanation in codes.

```
 7      <script type="text/javascript">
 8          class Animal{
 9              multicellular = true;
10              eukaryotic = true;
11              move(){
12                  console.log("Move forward");
13              };
14              reproduce(){
15                  console.log("Give birth to offspring");
16              }
17          }
18
19          class Mammal extends Animal{
20              hairy= true;
21              breastfeed(){
22                  console.log("Give milk to offspring");
23              }
24          }
25      </script>
```

Fig 18.11

Do you notice the 'extends' keyword on line 19? This shows the Mammal class is an extension of the 'Animal' class. Objects created from the Animal class will have two properties and methods while objects created from the Mammal class will have three properties and methods.

Example:

```
8   class Animal{
9       multicellular = true;
10      eukaryotic = true;
11      move(){
12          console.log("Move forward");
13      };
14      reproduce(){
15          console.log("Give birth to offspring");
16      }
17  }
18
19  class Mammal extends Animal{
20      hairy= true;
21      breastfeed(){
22          console.log("Give milk to offspring");
23      }
24  }
25  let first = new Animal; // two properties and methods
26  let second = new Mammal; // three properties and methods
```

Fig 18.12

Let's consider another example.

```
7   <script type="text/javascript">
8       class First{
9           constructor(name, dept){
10              this.name = name;
11              this.dept = dept;
12          }
13      }
14      class Second extends First{
15          school = "SQI";
16      }
17      let student = new Second("Taye", "software");
18      console.log(student.name); //Taye
19      console.log(student.dept); // software
20      console.log(student.school); // SQI
21  </script>
```

Fig 18.13

Explanation: The parent's class (First) constructor contains two arguments (line 9). Since the 'Second' class extends the 'First' class, the 'Second' class has all the properties and methods (including the constructor) of the 'First' class. I created a new instance of the child's class (Second) on line 17 and passed two parameters to the constructor. These

parameters are passed to the constructor of the ‘Second’ class (The ‘Second’ class has a constructor because it has inherited it from the ‘First’ class).

Method overriding

There are times when both the parent’s class and the child’s class have the same property name or method. In this case, objects created from the parent’s class contain the implementation present in the parent’s class while objects created from the child’s class contain the implementation present in the child’s class.

Example:

```
 8        class First{
 9            constructor(name, dept){
10                this.name = name;
11                this.dept = dept;
12            }
13            sayName(){
14                console.log(`My name is ${this.name}`);
15            }
16        }
17        class Second extends First{
18            school = "SQI";
19            sayName(){
20                console.log(`Hey, ${this.name}`);
21            }
22        }
23
24        let firstStudent = new First("Taye", "software");
25        let secondStudent = new Second("Kenny", "hardware");
26        firstStudent.sayName(); //My name is Taye
27        secondStudent.sayName(); //Hey, Kenny
```

Fig 18.14

On line 26, the method sayName of firstStudent is triggered. Since object firstStudent is an instance of ‘First’ class, it executes the codes on line 13.

On line 27, the method sayName of secondStudent is triggered. Since object secondStudent is an instance of ‘Second’ class, it executes the codes on line 19.

The peculiarity of the constructor

JavaScript throws and error if a constructor is introduced to a child’s class.

```
8         class First{
9             constructor(name, dept){
10                this.name = name;
11                this.dept = dept;
12            }
13            sayName(){
14                console.log(`My name is ${this.name}`);
15            }
16        }
17        class Second extends First{
18            constructor(school){
19                this.school = school;
20            }
21            sayName(){
22                console.log(`Hey, ${this.name}`);
23            }
24        }
25        let myStudent = new Second("Kenny", "hardware");
```

Fig 18.15

Here is the error

```
▸Uncaught ReferenceError: Must call super constructor in derived class before accessing 'this' or returning from derived constructor    index.html:19
    at new Second (index.html:19)
    at index.html:25
```

Fig 18.16

Note: A child's class cannot have its own constructor excepts it honors its parent.

How does a child's class honor its parent? By introducing 'super' keyword to the child's constructor. The 'super' keyword references the parent's class.

```
8         class First{
9             constructor(name, dept){
10                this.name = name;
11                this.dept = dept;
12            }
13            sayName(){
14                console.log(`My name is ${this.name}`);
15            }
16        }
17        class Second extends First{
18            constructor(school){
19                super();
20                this.school = school;
21            }
22            sayName(){
23                console.log(`Hey, ${this.name} from ${this.school}`);
24            }
25        }
26        let myStudent = new Second("Kenny", "hardware");
27        myStudent.sayName(); // Hey, undefined from Kenny
```

Fig 18.17

Explanation: Method overriding played a part in the codes above. I created an instance of the child's class (Second) on line 26. Two parameters are sent to the child's constructor, not the parent's constructor because the child now has its own constructor. Since the child's constructor has only one parameter, the first argument is sent as the value of the only parameter in the child's constructor while the second argument is discarded. For this reason, "Kenny" is sent as the value of "school" in the child's constructor.

This isn't the result I want.

The child's constructor can send values to the parent's constructor through the super keyword.

```
 8        class First{
 9            constructor(name, dept){
10                this.name = name;
11                this.dept = dept;
12            }
13            sayName(){
14                console.log(`My name is ${this.name}`);
15            }
16        }
17        class Second extends First{
18            constructor(name, dept, school){
19                super(name, dept);
20                this.school = school;
21            }
22            sayName(){
23                console.log(`Hey, ${this.name} from ${this.school}`);
24            }
25        }
26        let myStudent = new Second("Kenny", "hardware", "SQI");
27        myStudent.sayName(); // Hey, Kenny from SQI
```

Fig 18.18

The three arguments on line 26 are sent as the values of the three parameters in the child's constructor (line 18) which in turn forward two arguments to the parent's constructor through the super keyword (line 19).

Getters and Setters

A getter is used to get a value while a setter is used to set a value. Getters and setters are methods but you use them as though they are properties. You must not place parenthesis after their names while triggering them. Setters and getters are opposites of each other. While a setter **must** have just one argument, a getter **must** have no argument. A getter **should** return a value while a setter **shouldn't** return any value.

Getters

Getters are used to get values. To make a method a getter, all you need to do is to place a 'get' keyword before the method.

A getter is accessed like a property, not like a method.

Example:

```
 8          class Person{
 9              constructor(name){
10                  this.name = name;
11              }
12              get firstname(){
13                  return this.name;
14              }
15          }
16          let myPerson = new Person("Taiwo");
17          console.log(myPerson.firstname); //Taiwo
```

Fig 18.19

The method on line 12 (firstname) is a getter. Kindly observe I used it on line 17. It is used as though it is a property, that is, no parenthesis is placed after the name while calling the getter.

Setter

A setter is used to set a value. Add a 'set' keyword before a method to make it a setter. Since a setter must have a parameter and we aren't allowed to place a parenthesis after a setter, how do we pass an argument to the parameter in the setter?

An argument is passed to the parameter in a setter by using the assignment operator (check line 17 in fig 18.20).

Example:

```
 8          class Person{
 9              constructor(name){
10                  this.name = name;
11              }
12              set myName(name){
13                  this.name = name;
14              }
15          }
16          let myPerson = new Person("Taiwo");
17          myPerson.myName = "Kenny";
18          console.log(myPerson.name); //Kenny
```

Fig 18.20

Let’s use both of them together.

```
 8        class Person{
 9            constructor(name){
10                this.name = name;
11            }
12            get firstname(){
13                return this.name;
14            }
15            set firstname(name){
16                this.name = name;
17            }
18        }
19        let myPerson = new Person("Taiwo");
20        myPerson.firstname = "Kenny";
21        console.log(myPerson.firstname); //Kenny
```

Fig 18.21

The setter was called on line 20 while the getter was called on line 21.

SECTION THREE

OTHER IMPORTANT THINGS

This section explains other things you need to know as a frontend JavaScript developer.

CHAPTER NINETEEN

LOCAL STORAGE

Global variables are reset on browser's refresh or on browser's restart. Sometimes, you want to persist values beyond browser's refresh or restart, you want a value to be available even after you close the browser. Local storage allows you store some values on the browser. Any page can access the stored values on the browser. Items are stored in local storage in key-value pairs. JavaScript provides us with an object to interact with the browser's local storage. The JavaScript object is localStorage.

Example:

```
7       <script type="text/javascript">
8           localStorage.fullname = "Taye Abidakun";
9       </script>
```

Fig 19.1

The code above sets a key-value pair in the browser's local storage. The key is 'fullname' while the value is "Taye Abidakun". The property is set when the page is loaded on the browser.

How do we locate the stored value on the browser?

Open the browser's console, click on 'application'- If you can't find 'application', click on the double arrow (>>) to see it or check the browser's dev tools- click on "Local Storage", finally, click on "file://" to see the items stored in the browser's local storage.

Restart your browser, can you still find the item(s) stored in your local storage? I expect you to see the item(s). This means that items stored in local storage persist beyond browser's restart or refresh.

To retrieve the items stored in local storage through JavaScript is simple.

```
7       <script type="text/javascript">
8           console.log(localStorage.fullname) // "Taye Abidakun";
9       </script>
```

Fig 19.2

JavaScript also provides us with two methods to store and retrieve values from localStorage. The methods are 'setItem' and 'getItem'.

setItem: This method receives two arguments. The first argument is the key while the second argument is the value.

Example:

```
7       <script type="text/javascript">
8           localStorage.setItem("dept", "software");
9       </script>
```

Fig 19.3

The codes above set a key-value pair on the browser where the key is “dept" and the value is “software”.

getItem: This method receives one argument. The argument is used as the key while fetching item from local storage.

```
7       <script type="text/javascript">
8           console.log(localStorage.getItem("dept"));
9       </script>
```

Fig 19.4

This logs “software” to the console.

Every item stored in local storage is converted to a string irrespective of the initial datatype of the value to be stored. To understand the implication of the statement above, let’s store an array in the local storage.

```
7       <script type="text/javascript">
8           let nums = [4, 5, 6, 7];
9           console.log(nums[0], nums[1]); // 4 5
10          localStorage.setItem("myNums", nums);
11          let arr = localStorage.getItem("myNums");
12          console.log(arr[0], arr[1]); // 4 ,
13      </script>
```

Fig 19.5

Explanation:

Line 8: Variable “nums” is initialized and array is stored in it.

Line 9: The indexes 0 and 1 of “nums” is logged to the console. Since it is an array, the index 0 is 4 while the index 1 is 5 (before it is stored in local storage).

Line 10: “nums” is stored in local storage with a key of “myNums”. Local storage converts it to string before storing it. It removes the square braces around the array in the process

Key	Value
myNums	4,5,6,7

Fig 19.6

Line 11: The value of myNums is retrieved from local storage and stored in variable "arr".

Note that the retrieved value is a string because local storage has converted the array to string while storing it.

Line 12: The indexes 0 and 1 of the retrieved value is logged to the console. Since it is a string, the index 0 is 4 while the index 1 is comma (,).

This isn't the result I want. I want to retrieve an array, not a string. How do I go about it?

An African proverb says: The best way to keep an item safe from a known thief is to keep it with the thief. A thief can't steal what is under his/her care.

In the same vein, the best way to stop local storage from converting a value to string is to convert it to string before storing it in local storage. Since it is a string, local storage will no longer convert it to string. Finally, we will use the same tool (used in converting a value to string) to convert it back from string.

```
7    <script type="text/javascript">
8        let nums = [4, 5, 6, 7];
9        console.log(nums[0], nums[1]); // 4 5
10       localStorage.setItem("myNums", JSON.stringify(nums));
11       let arr = JSON.parse(localStorage.getItem("myNums"));
12       console.log(arr[0], arr[1]); // 4 5
13   </script>
```

Fig 19.7

Line 8: Variable "nums" is initialized and array is stored in it.

Line 9: The indexes 0 and 1 of "nums" is logged to the console. Since it is an array, the index 0 is 4 while the index 1 is 5.

Line 10: The stringify method of JSON (I will talk more on JSON in this chapter) is used to convert the array to string before storing it in local storage. This preserves the square braces of the array.

myNums	[4,5,6,7]

Fig 19.8

Line 11: The parse method of JSON is used to convert the value retrieved from local storage back to a JavaScript array.

Line 12: The indexes 0 and 1 of the retrieved value is logged to the console. Since it is a string, the index 0 is 4 while the index 1 is 5.

Arrays and objects are treated like this when storing them in local storage.

Lapses of local storage

- The information in local storage is only available on the browser that sets it. It is not a substitute for database.
- Do not store sensitive information in local storage as it can be accessed by any page that runs on the browser.
- For most browsers, the maximum volume of data you can store in local storage is 5MB.

JSON

JSON stands for JavaScript Object Notation. JSON is a lightweight, human-readable format for storing and transporting data. JSON stores its data in key-value pair just like JavaScript object. Most programming languages can read (data stored in JSON) and convert data to JSON. This means a programming language can store data in JSON and transfer it to another programming language if both languages can read data stored in JSON.

An object is available in JavaScript to read and convert data to JSON format. The object name is also named JSON. The object has two methods. They are: stringify and parse.

Stringify: According to developer.mozilla.org, JSON.stringify method converts a JavaScript object or value to a JSON string.

Parse: According to developer.mozilla.org, JSON.parse method parses a JSON string, constructing the JavaScript value or object described by the string.

Summary: JSON.stringify converts JavaScript data to a JSON string while JSON.parse converts a JSON string to its JavaScript equivalent. The two methods are opposite of each other.

CHAPTER TWENTY

HIGHER ORDER FUNCTIONS (HOF) AND CALLBACKS

Higher order function (HOF)

HOF is a function that take other functions as arguments or a function that returns another function. HOF can also do both, that is, takes a function as argument and also returns a function.

A callback function is a function passed to another function as an argument which is then triggered inside the outer function.

Example:

```
7     <script type="text/javascript">
8         const firstFunc = ()=>{
9             console.log("hey first");
10        }
11        const secondFunc = fn=>{
12            console.log("hi second");
13            fn();
14        }
15    </script>
```

Fig 20.1

There are two functions in the codes above. firstFunc starts on line 8 and logs “hey first” to the console when it is triggered. secondFunc starts online 11 and logs “hi second” to the console when it is triggered, but there is more to secondFunc. secondFunc is a parameterized function. It is expected that an argument will be sent as the value of “fn” parameter. The argument to be sent is expected to be a function such that when “fn” is triggered on line 13, the block of codes within the body of the sent function will be executed.

At the moment, nothing is logged to the console because none of the two functions is triggered. “but fn is triggered on line 13” you may argue. You’ve got a point, take a look at the codes again and you will see that line 13 is within secondFunc. Every code within secondFunc is ignored until secondFunc is triggered.

Let’s trigger the functions together.

Fig 20.2

Explanation:

- In JavaScript, functions are variables. I saved a function as the value of firstFunc on line 8. A parameterized function is saved as the value of secondFunc on line 11.
- secondFunc is triggered as a parameterized function while firstFunc is sent as an argument to secondFunc on line 15.
- firstFunc is not triggered on line 15. Recall that we need to place a parenthesis after a function in order to trigger the function. No parenthesis is placed after firstFunc on line 15. Since functions are variables, it therefore means I passed the value of firstFunc as an argument to secondFunc on line 15.

The interpretation of this is that: On line 15, I passed a function as an argument to secondFunc. This argument is sent as the value of "fn" parameter in secondFunc. Since secondFunc is triggered, the codes within the function will be executed.

Line 12 logs "hi second" to the console. Line 13 triggers "fn". Since the value of "fn" is the firstFunc function, the codes within firstFunc is executed and "hey first" is logged to the console.

In the codes above, secondFunc is a HOF while firstFunc is a callback.

secondFunc is a HOF because it takes another function (firstFunc) as an argument.

firstFunc is a callback function because it is a function passed to another function (secondFunc) and it is triggered (line 13 in fig 20.2 above) within the outer function (secondFunc).

The codes in fig 20.2 can be rewritten as

Fig 20.3

Instead of saving the callback function as a variable as I did in fig 20.2, I passed the function directly as an argument to secondFunc in fig 20.3. It still works the same way as the codes in fig 20.2. You will see codes like this as you learn more about HOF and callbacks, I introduced it so you can get familiar with it.

Inbuilt HOF

Some HOFs have been built into JavaScript. Let's take a look at some of them.

SetTimeout HOF

This HOF calls a function after some milliseconds.

Example:

The codes below triggers "displayName" after 2000ms.

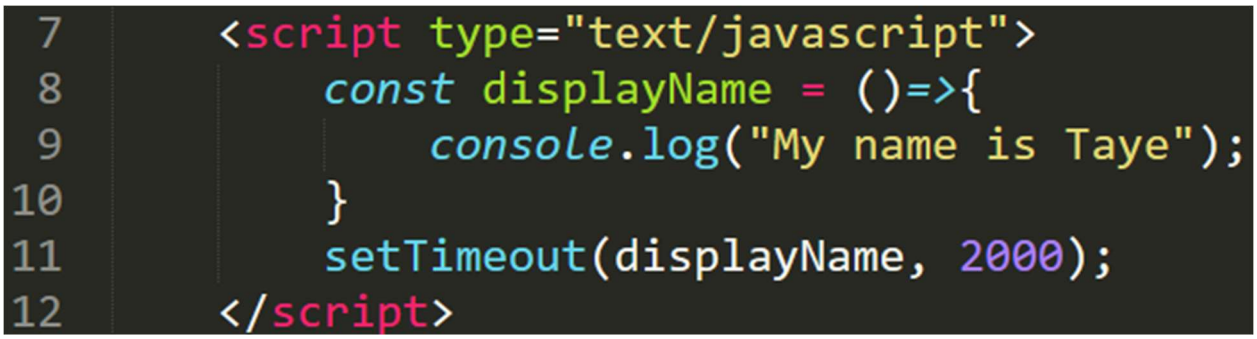

Fig 20.4

The first argument passed of setTimeout is a callback function (displayName in fig 20.4), the second argument is the delay time (in milliseconds) before the setTimeout is triggered.

The codes above triggers displayName after 2000ms.

Example:

```
7       <script type="text/javascript">
8           setTimeout(()=>{console.log("good")}, 500);
9       </script>
```

Fig 20.5

The codes above logs “good” to the console after 500ms. Again, get used to this syntax.

ForEach HOF

This HOF is used on arrays. It takes up to two arguments. The first argument is a callback, the second argument is the “this” argument. I will only talk about the first parameter (callback) in this section, the second parameter will be discussed in chapter twenty-one where I explain “this” extensively.

The callback is triggered for each of the items in the forEach array. If the array has three items, the callback is triggered three times.

Example:

```
7       <script type="text/javascript">
8           let arr = [5, 7, 8, 2];
9           const myFunc = ()=>{
10              console.log("hi");
11          }
12          arr.forEach(myFunc);
13      </script>
```

Fig 20.6

ForEach is used on line 12. This triggers myFunc four times because there are 4 elements in the “arr” array.

The callback within a forEach method can take up to three parameters.

- The first parameter represents each value in the array.

Example:

```
 7      <script type="text/javascript">
 8          let arr = [5, 7, 8, 2];
 9          const myFunc = v=>{
10              console.log(v * 2);
11          }
12          arr.forEach(myFunc);// 10 14 16 4
13      </script>
```

Fig 20.7

Here is the result

Fig 20.8

Let's explain what happens on line 12

"arr" has 4 items in it. On line 12,

1. JavaScript picks the first item in the arr array (5) and give it to the forEach HOF.
2. ForEach in turn gives the value to its callback (myFunc).
3. "myFunc" receives it and set it as the value of its parameter (v).
4. The codes in "myFunc" is executed (v is multiplied by 2 and logged to the console).

When it is done, JavaScript picks the second value in the array (7) and go through the same process: JavaScript hands it over to the forEach HOF. ForEach again gives it to its callback. myFunc receives it and set it as the value of "v". myFunc multiples "v" by 2 and log the result to the console.

The cycle continues till JavaScript gives all the values in the array to the forEach HOF.

- The second parameter holds the index of the picked value.

Example:

```
7     <script type="text/javascript">
8         let arr = ["Peter", "James", "John"];
9         const myFunc = (v, i)=>{
10            console.log(`${v} stays at ${i}`);
11        }
12        arr.forEach(myFunc);
13    </script>
```

Fig 20.9

Here is the result on the browser

```
Peter stays at 0                    index.html:10
James stays at 1                    index.html:10
John stays at 2                     index.html:10
```

Fig 20.10

- The third parameter represents the whole array

Example:

```
7     <script type="text/javascript">
8         let arr = ["Peter", "James", "John"];
9         const myFunc = (v, i, myArr)=>{
10            console.log(`${v} stays at ${i} taken from ${myArr}`);
11        }
12        arr.forEach(myFunc);
13    </script>
```

Fig 20.11

Here is the result on the browser

```
Peter stays at 0 taken from Peter,James,John        index.html:10
James stays at 1 taken from Peter,James,John        index.html:10
John stays at 2 taken from Peter,James,John         index.html:10
```

Fig 20.12

Map HOF

This is also used with arrays and it is similar to forEach with few differences. Like forEach, it also takes two arguments. The first argument is a callback while the second argument is the “this” argument (discussed in chapter twenty-one).

The callback also takes up to 3 parameters.

Again, like ForEach, the first parameter of the callback represents each value in the array, the second parameter represents the index of the picked value in the array while the third parameter represents the whole array.

Example:

```
7    <script type="text/javascript">
8        let arr = ["Peter", "James", "John"];
9        const myFunc = (v, i, myArr)=>{
10           console.log(`${v} stays at ${i} taken from ${myArr}`);
11       }
12       arr.map(myFunc);
13   </script>
```

Fig 20.13

Here is the result on the browser

```
Peter stays at 0 taken from Peter,James,John        index.html:10
James stays at 1 taken from Peter,James,John        index.html:10
John stays at 2 taken from Peter,James,John         index.html:10
```

Fig 20.14

"Based on what you have explained and the result generated, 'map' works the same was as 'forEach'", you are thinking. You are right. Let's move to consider the differences between the two.

Differences between forEach and Map

The difference between them is that the callback of "map" will return an array while the callback of "forEach" will always return undefined.

Let's return a value from forEach callback.

Example:

```
7    <script type="text/javascript">
8        let arr = [5, 7, 8, 2];
9        const myFunc = v=> v * 2;
10       let a = arr.forEach(myFunc);
11       console.log(a);// undefined
12   </script>
```

Fig 20.15

Explanation:

On line 10,

1. JavaScript gives the first value of the arr array (5) to forEach.
2. forEach gives it to "myFunc".
3. myFunc assigns it as the value of "v".
4. myFunc multiplies the value of "v" by 2.

5. myFunc returns the result (10).
   All the steps above (from 1 to 4) is executed except the fifth step. Instead of returning the result, myFunc returns undefined. forEach will always return undefined.
   Therefore, the value of "a" is undefined.
   JavaScript picks the second value in the array (7) and goes through the same process. The process is repeated for all the values in the array.

Returning from map

```
 7      <script type="text/javascript">
 8          let arr = [5, 7, 8, 2];
 9          const myFunc = v=> v * 2;
10          let a = arr.map(myFunc);
11          console.log(a);// [10, 14, 16, 4]
12      </script>
```

Fig 20.16

Explanation:

On line 10,

1. JavaScript gives the first value of the array (5) to map.
2. map gives it to myFunc.
3. myFunc assigns it as the value of "v".
4. myFunc multiplies the value of "v" by 2.
5. myFunc returns the result (10).

Map goes through the 5 steps listed above. Map doesn't stop there, it creates a new array and push the returned value into the new array. This means 10 is kept in the new array.

JavaScript picks the second value in the array (7), goes through the same process. 14 is pushed into the new array. The process is repeated for the third value (8), 16 is pushed into the new array. The process is repeated for the last value (2), 4 is pushed into the new array.

Finally, map assigns the new array as the value of variable "a".

On line 11, the value of variable "a" is logged to the console.

Filter HOF

Filter is like a sieve. The size of the sieve is set such that some items will be able to pass through it while others won't. Same way, we write our filter codes to allow some values pass through it while others won't. Any value that passes through the filter will be returned while values that don't, will be discarded.

Like forEach and Map, filter also takes two arguments. The first argument is a callback while the second is the "this" argument (discussed in chapter twenty-one).

Example:

```
7    <script type="text/javascript">
8        let arr = [5, 7, 8, 2];
9        const myFunc = v=> v > 6;
10       let a = arr.filter(myFunc);
11       console.log(a);// [7, 8]
12   </script>
```

Fig 20.17

Again, like map and forEach, filter callback also takes three parameters. The first represents the value, the second represents the index while the third represents the whole array.

Explanation:

On line 10,

1. JavaScript gives the first value of the arr array (5) to filter.
2. filter gives it to myFunc.
3. myFunc assigns it as the value of "v".
4. myFunc checks if the condition is met. 5 is not greater than 6, therefore it is discarded.

JavaScript moves to the next value in the array.

1. JavaScript gives the second value of the array (7) to filter.
2. filter gives it to myFunc.
3. myFunc assigns it as the value of "v".
4. myFunc checks if the condition is met. 7 is greater than 6, therefore myFunc returns the value.

Filter doesn't stop there, it creates a new array and push the returned value into the new array. This means 7 is kept in the new array.

JavaScript moves to the next value in the array.

1. JavaScript gives the third value of the array (8) to filter.
2. filter gives it to myFunc.
3. myFunc assigns it as the value of "v".
4. myFunc checks if the condition is met. 8 is greater than 6, therefore myFunc returns the value.

Filter pushes the returned value into the new array created.

JavaScript moves to the next value in the array.

1. JavaScript gives the fourth value of the array (2) to “filter”.
2. “filter” gives it to myFunc.
3. myFunc assigns it as the value of “v”.
4. myFunc checks if the condition is met. 2 is not greater than 6, therefore it is discarded.

The new array is assigned as the value of variable “a”.

The value of variable “a” is logged to the console on line 11.

Example:

```
7     <script type="text/javascript">
8         let myArray = [1, 2, 3, 4];
9         const myFunc = v=> v > 6;
10        let newArray = myArray.filter(myFunc);
11        console.log(newArray);// []
12    </script>
```

Fig 20.18

The code above gives an empty array because none of the values in the array meets the condition in the callback.

Like forEach and map, the second and third parameters of filter callback holds the index of each value and the whole array respectively.

Find

“Find” is very similar to filter with two exceptions: “find” stops iteration immediately a value meets its condition, **“find” doesn’t create a new array.**

“filter” is like “select all” while “find” is like “select”. “Filter” selects all the values that meets its condition while “find” selects only the first value that meets its condition.

Example:

```
 7      <script type="text/javascript">
 8          let arr = [5, 7, 8, 2];
 9          const myFunc = v=> v > 6;
10          let a = arr.find(myFunc);
11          console.log(a);// 7
12      </script>
```

Fig 20.19

Explanation:

On line 10,

1. JavaScript gives the first value of the arr array (5) to “find”.
2. “find” gives it to myFunc.
3. myFunc assigns it as the value of “v”.
4. myFunc checks if the condition is met. 5 is not greater than 6, therefore it is discarded.

JavaScript moves to the next value in the array.

1. JavaScript gives the second value of the array (7) to “find”.
2. “find” gives it to myFunc.
3. myFunc assigns it as the value of “v”.
4. myFunc checks if the condition is met. 7 is greater than 6, therefore myFunc returns the value.

“Find” doesn’t create a new array, instead, it assigns the returned value to variable “a” and stops the iteration.

The value of “a” is logged to the console on line 11.

Example:

```
 7      <script type="text/javascript">
 8          let arr = [1, 2, 3, 4];
 9          const myFunc = v=> v > 6;
10          let a = arr.find(myFunc);
11          console.log(a);// undefined
12      </script>
```

Fig 20.20

The code above gives undefined because none of the values in the array meets the condition in the callback.

“Filter” gives an empty array if none of values in the array meets the condition while “find” gives undefined if none of the values meets the condition. This is because “find” doesn’t create a new array.

Like forEach, map, and filter the second and third parameters of find callback holds the index of each value and the whole array respectively.

Note: If you’re just starting out, I don’t advice the use of map, filter and find. I will advice you use loop (for or while loop) to achieve the result. You can use these HOFs after you’ve mastered the use of loops.

Read on loops in chapter eleven.

CHAPTER TWENTY-ONE

"THIS" KEYWORD

Introduction

"this" is arguably the most confusing keyword in JavaScript. I will start this explanation from what I have discussed in the previous chapter of this textbook. I have used 'this' in two scenarios in this book. First, I have used "this" in a method of an object. Second, I have used "this" in a method of a class. "this" behaves the same way in the two scenarios considered above, that is, **when you call on 'this' within a method, 'this' refers to the parent of the method calling on 'this'.** We will consider 'this' in other situations.

- **'this' in a regular function**

'this' refers to the window object when used within a regular function.

Regular function refers to function we considered in chapter eight (ES5 function).

```
 8        function myFunc() {
 9            console.log(this); // Window
10        }
11        myFunc();
```

Fig 21.1

Example:

In the codes below, we have an object. The object has two properties (name, friends) and a method (displayFriends).

```
 8        let person = {
 9            name: "Taye",
10            friends: ["Peter", "James", "John"],
11            displayFriends(){
12                console.log(`My name is ${this.name},
13                    I have ${this.friends.length} friends`);
14
15                this.friends.forEach(function () {
16                    console.log(`My name is ${this.name},
17                        I have ${this.friends.length} friends`);
18                });
19
20            }
21
22        }
23        person.displayFriends();
```

Fig 21.2

person.displayFriends is triggered on line 23, this executes the block of codes between line 11 and line 20 (that is, the body of the method).

Line 12 displays “My name is Taye, I have three friends”, while line 16 throws an error (cannot read property length of undefined).

I will reduce the codes above so we can go to the root of the error.

```
 8        let person = {
 9            name: "Taye",
10            friends: ["Peter", "James", "John"],
11            displayFriends(){
12                console.log(`My name is ${this.name}`);
13
14                this.friends.forEach(function () {
15                    console.log(`My name is ${this.name}`);
16                });
17
18            }
19
20        }
21        person.displayFriends();
```

Fig 21.3

Here is the result on the browser:

```
My name is Taye                                    index.html:12
3 My name is                                       index.html:15
```

Fig 21.4

"My name is Taye" is displayed on line 12.

"My name is " is displayed three times on line 15.
Why is "Taye" omitted on line 15 but displayed on line 12?

Explanation:

Although both line 12 and 15 are inside the displayFriends method, there is a difference between the two. The codes on line 15 is within a forEach. ForEach has a regular function as callback within it (refer to chapter twenty). Recall that 'this' refers to the window object whenever it is called within a regular function, therefore, 'this' on line 15 refers to the window object.

How do we make the 'this' on line 15 point to 'person' and not window object? There are different solutions to this.

Solution 1:

The solution I am about to discuss is an old solution and I won't recommend it, nonetheless, I am including it because it explains the scope of 'this' and summarizes everything I have discussed in this chapter than any other solution.

```
 8        let person = {
 9            name: "Taye",
10            friends: ["Peter", "James", "John"],
11            displayFriends(){
12                console.log(`My name is ${this.name}`);
13                let that = this;
14                this.friends.forEach(function () {
15                    console.log(`My name is ${that.name}`);
16                });
17
18            }
19
20        }
21        person.displayFriends();
```

Fig 21.5

Here is the result on the browser:

```
My name is Taye                                    index.html:12
3 My name is Taye                                  index.html:15
```

Fig 21.6

Explanation: The regular function is a callback within forEach, therefore I saved the value of "this" into another variable (that) on line 13 ("this" refers to "person" on line 13), so I can make use of "that" within the regular function (where "this" points to the window object).

Solution 2: So far, we have passed only one argument to the forEach function but it can take more. It can take a second argument. Whatever you pass as the second argument will be used as the value of 'this' within the forEach callback.

Example:

```
8           let arr = [3, 4, 5];
9           arr.forEach(function(){
10              console.log(this);
11          }, {country: "Nigeria", state: "Ondo"});
```

Fig 21.7

Here is the result on the browser:

```
▸ {country: "Nigeria", state: "Ondo"}                    index.html:10
▸ {country: "Nigeria", state: "Ondo"}                    index.html:10
▸ {country: "Nigeria", state: "Ondo"}                    index.html:10
```

Fig 21.8

Explanation: I passed an object -{country: "Nigeria", state: "Ondo"}- as the second argument of forEach. This object is used as the value of 'this' within the forEach callback.

Let's apply this same logic to the codes in fig 21.5.

```
8           let person = {
9               name: "Taye",
10              friends: ["Peter", "James", "John"],
11              displayFriends(){
12                  console.log(`My name is ${this.name}`);
13
14                  this.friends.forEach(
15                      function () {
16                          console.log(`My name is ${this.name}`);
17                      },
18                      this);
19
20              }
21
22          }
23          person.displayFriends();
```

Fig 21.9

On line 18, I passed ‘this’ as the second argument of forEach. The ‘this’ on line 18 refers to ‘person’ and not the window object (this refers to the window object whenever you call it within a regular function. The regular function starts on line 15 and ends on line 17.). The value passed as the second argument of forEach will be used as the value of ‘this’ in the first argument of forEach (within the callback).

Solution 3:

Bind, call, apply:

Not all functions behave like forEach where the value of the second argument is used as ‘this’ in the first argument. This is the reason we have: bind, apply and call.

The three functions (bind, apply, call) are attached to another function while using them.

Let’s explain them one after the other.

a. Bind: This function receives at least one argument and returns a new function. The value passed as the first argument of ‘bind’ will be used as the value of ‘this’ in the new function returned.
Example:

```
 8        function myFunc() {
 9            console.log(`My name is ${this.name}`);
10        }
11        let newFunc = myFunc.bind({name: "Taye", dept: "software"});
12        newFunc(); // My name is Taye
```

Fig 21.10

On line 11, ‘bind’ is attached to ‘myFunc’ and an object passed as the only argument of ‘bind’. This returns a new function where the only parameter passed to ‘bind’ is used as the value of ‘this’ in the returned function. This returned function is saved as newFunc.

newFunc is triggered on line 12 and we have “My name is Taye” as the result.

If it looks confusing to you, look at it from this perspective:

When ‘bind’ is attached to a function (myFunc), it copies all the codes in the function and save it in a new function (newFunc) but with one addition. It uses the first argument of ‘bind’ as the value of ‘this’ within the new function (newFunc).

I hope this is simple and clear enough.

“What happens when the function involved is a parameterized function?”, you may ask.

The first argument of ‘bind’ is used as the value of ‘this’, the second argument of ‘bind’ is sent as the value of the first parameter present in the function. The third argument of ‘bind’ is sent as the value of the second parameter in the function and so on.

Example:

```
function myFunc(school) {
    console.log(`My name is ${this.name},
        my school is ${school}`);
}
let newFunc = myFunc.bind({name: "Taye", dept: "software"}, "SQI");
newFunc(); //My name us Taye, my school is SQI
```

Fig 21.11

In this case, the first argument of 'bind' is sent as the value of 'this' while the second argument of 'bind' (SQI) is sent as the value of the first parameter (school) in the function.

Let's rewrite the codes in fig 21.5 using bind.

```
let person = {
    name: "Taye",
    friends: ["Peter", "James", "John"],
    displayFriends(){
        console.log(`My name is ${this.name}`);

        this.friends.forEach(
            function () {
                console.log(`My name is ${this.name}`);
            }.bind(this));

    }

}
person.displayFriends();
```

Fig 21.12

Call: This method works like 'bind' but with a little difference, 'call' is triggered immediately. You do not need to save it in a new variable.

Let's write the codes in fig 21.11 using 'call'.

```
function myFunc(school) {
    console.log(`My name is ${this.name},
        my school is ${school}`);
}
myFunc.call({name: "Taye", dept: "software"}, "SQI");
```

Fig 21.13

Apply: 'apply' works like 'call' but with a little difference, it only receives two arguments. The first argument is used as the value of 'this' (just like 'bind' and 'call'). The second argument must be an array. The first value in the array is sent as the value of the function's

first parameter. The second value in the array is sent as the value of the function's second parameter and so on.

It is triggered immediately, just like call.

Let's write the codes in fig 21.11 using 'apply'.

```
function myFunc(school) {
    console.log(`My name is ${this.name},
        my school is ${school}`);
}
myFunc.apply({name: "Taye", dept: "software"}, ["SQI"]);
```

Fig 21.14

Solution 4:

In JavaScript, an arrow function inherits 'this' from its parent. By converting the regular function to an arrow function, the problem is solved.

```
let person = {
    name: "Taye",
    friends: ["Peter", "James", "John"],
    displayFriends(){
        console.log(`My name is ${this.name}`);

        this.friends.forEach(
            () =>{
                console.log(`My name is ${this.name}`);
            });

    }

}
person.displayFriends();
```

Fig 21.15

Here is the result:

```
My name is Taye                                    index.html:12
3 My name is Taye                                  index.html:16
```

Fig 21.16

Perfect. Can you see how simple it is?

A simple hack to know the value of 'this' when dealing with arrow function is to ask this question**: What is the value of 'this' when it is called within the direct parent of the arrow function?** Whatever your answer is, is the value of 'this' within the arrow function.

In the codes above (fig 21.15), the direct parent of the arrow function is the displayFriends method. Whatever the value of 'this' is, within displayFriends will be used as the value of 'this' in the arrow function. For this reason, 'this' in the arrow function will refer to the 'person' object.

To understand this better, I will split things up, just like I did while explaining forEach in chapter twenty.

```
 8          const myFunc = () =>{
 9                              console.log(`My name is ${this.name}`);
10                          };
11          let person = {
12              name: "Taye",
13              friends: ["Peter", "James", "John"],
14              displayFriends(){
15                  console.log(`My name is ${this.name}`);
16
17                  this.friends.forEach(myFunc);
18
19              }
20
21          }
22          person.displayFriends();
```

Fig 21.17

Here is the result:

```
My name is Taye                                    index.html:15
3 My name is                                       index.html:9
```

Fig 21.18

The arrow function has been taken out of displayFriends method. The arrow function is now on line 8. displayFriends is no longer the direct parent of the arrow function even though the arrow function is used within displayMethod on line 17. The direct parent of the arrow function on line 8 is the window object. For this reason, 'this' refers to the window object.

Let's consider another example

```
8       let person = {
9           name: "Taye",
10          displayName:()=>{
11              console.log(`My name is ${this.name}`);
12          },
13          showName(){
14              console.log(`My name is ${this.name}`);
15          }
16
17      }
18      person.displayName(); // My name is
19      person.showName(); // My name is Taye
```

Fig 21.19

I converted displayName method to an arrow function and somehow, it isn't giving the desired result.

Explanation: A simple question to understand the problem at hand is to ask the question "What is the value of 'this' when it is called within the parent of the arrow function?". The parent of the arrow function is the 'person' object. 'this' within the 'person' object points to window object. Therefore 'this' within the arrow function points to the window object.

Sounds confusing? I will explain better. Before I proceed, take a 10 secs break. Relax.

Are you ready?

Let's forget about the arrow function and let's take a look at the 'showName' method in fig 21.19.

As I discussed earlier, whenever 'this' is used within a method of an object (I repeat, forget about arrow function for now), 'this' refers to the parent of the method (the 'person' object in the codes above).

When 'this' is called within showName, 'this' points to the parent of 'showName', which is 'person'. Following the same logic, when 'this' is called within 'person', 'this' points to the parent of 'person', which is the window object (this is only used for the sake of explanation, you will only call "this" within the method of an object, not directly within the object).

Back to arrow function.

The value of 'this' within the direct parent of an arrow function is used as the value of 'this' within the arrow function. The direct parent of the arrow function in fig 21.19 is the 'person' object. The value of 'this' within the 'person' object will be used as the value of 'this' within the arrow function. Since 'this' points to the window object when called within 'person', 'this' will also point to window object when called within the arrow function.

For this reason, you don’t use an arrow function as a method of an object.

CHAPTER TWENTY-TWO

ERROR HANDLING

Encountering errors is part of the journey. The good news is that, debugging and fixing errors make you better and make less errors in the future. The bad news is that, you will never outgrow errors. Therefore, get used to seeing, debugging and handling errors.

The best way to avoid errors is to anticipate and handle them appropriately.

Before we proceed, I want to explain two important terms: Compile time and runtime.

JavaScript is a high-level programming language. When you write codes in JavaScript, the codes are converted to machine language. The period when your codes are interpreted to machine language is known as compile time.

Runtime is when the commands in your codes are getting executed, in order words, this is the time your codes are running.

Errors can be generated both at compile time and at runtime.

To handle errors, you create two blocks. The first block is a “try block” while the second block is a “catch block”. You place the codes that **might** generate errors in the “try block”. If an error is encountered in the “try block”, JavaScript terminates the remaining codes in the “try block” and jumps to execute the codes in the “catch block”. If there is no error, JavaScript will successfully execute every code in the “try block” and ignore the codes in the “catch block”.

Example:

```
 7      <script type="text/javascript">
 8          try{
 9              console.log("hello");
10              alert(a);
11              console.log("hi");
12          }
13          catch{
14              console.log("OOPS! Something went wrong");
15          }
16      </script>
```

Fig 22.1

Here is the result on the browser

```
hello                                   index.html:9
OOPS! Something went wrong              index.html:14
```

Fig 22.2

In fig 22.1, the “try block” starts on line 8 and ends on line 12. The “catch block” starts on line 13 and ends on line 15.

When the codes are executed on the browser, JavaScript tries executing the codes in the “try block”. It executes the codes on line 9 successfully because it doesn’t contain any error. It moves to line 10. It tries to alert “a” and encounters an error because variable “a” is not defined. For this reason, it terminates the “try block” and run the codes in the “catch block”.

Exceptions

Knowing there is an error is not enough, you want to have information about the error so you can handle it appropriately.

```
7   <script type="text/javascript">
8       try{
9           console.log("hello");
10          alert(a);
11          console.log("hi");
12      }
13      catch(e){
14          console.log(e);
15      }
16  </script>
```

Fig 22.3

Here is the result on the browser

```
hello                                                    index.html:9
ReferenceError: a is not defined                         index.html:14
    at index.html:10
```

Fig 22.4

Do you notice the “e” on line 13, just after the “catch” keyword? It holds the information about the error. The “e” stands for exception. You can give it any name of your choice. I see you asking “what are exceptions?”. Exceptions are runtime errors. These exceptions (“e” on line 13) hold errors generated at runtime. Oh! You are asking “What about the errors generated at compile time?”. **Sorry, in JavaScript, you can only catch runtime errors, not compile time errors.**

“Finally” keyword

Sometimes you want to execute some codes whether an exception is thrown or not. The “finally block” got you covered. The codes within the “finally block” will be executed whether an error is thrown or not.

Example:

```
<script type="text/javascript">
    try{
        console.log("hello");
        alert(a);
        console.log("hi");
    }
    catch(e){
        console.log(e);
    }
    finally{
        console.log("show anyway");
    }
</script>
```

Fig 22.5

Here is the result on the browser.

```
hello                                                    index.html:9
ReferenceError: a is not defined                         index.html:14
    at index.html:10
show anyway                                              index.html:17
```

Fig 22.6

“Throw” keyword

We are also allowed to throw exceptions. This raises a custom error. When JavaScript sees the “throw keyword”, it moves straight to executes the codes in the catch block. Whatever message you add to the throw keyword will be sent to the exception in the catch block. This message can be of any datatype.

Example:

```
7     <script type="text/javascript">
8         let val = 10;
9         try{
10            console.log("hello");
11            if (val > 5){
12                throw "Number is too large";
13            }
14            console.log("hi");
15        }
16        catch(e){
17            console.log(`Here is the error message: ${e}`);
18        }
19    </script>
```

Fig 22.7

Here is the result on the browser

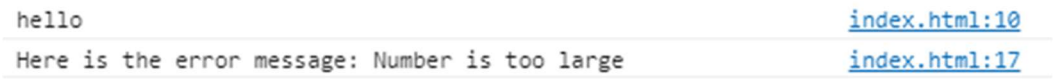

Fig 22.8

In the codes in fig 22.7, the condition on line 11 is true since val is greater than 5. For this reason, JavaScript enters the “if block” and sees the “throw” keyword, therefore, it raises an exception. It moves straight to execute the codes in the catch block, sending “Number is too large” as the value of “e” on line 16.

What if the condition on line 11 is not met?

```
7     <script type="text/javascript">
8         let val = 2;
9         try{
10            console.log("hello");
11            if (val > 5){
12                throw "Number is too large";
13            }
14            console.log("hi");
15        }
16        catch(e){
17            console.log(`Here is the error message: ${e}`);
18        }
19    </script>
```

Fig 22.9

Here is the result on the browser

Fig 22.10

JavaScript ignores the “if block’ on line 11 since the condition is not met. For this reason, it doesn’t execute the “throw” command.

Error handler

A common practice is to create a function to be triggered when there is an error. This function reacts to different errors based on the error message. Such functions are known as error handlers.

Example:

```
8         let val = 1;
9         const myFunc = err=>{
10            if(err.type=="lowNum"){
11                console.log(`${err.message} is too low`);
12            } else if(err.type=="highNum") {
13                console.log(`${err.message} is too high`);
14            }
15        }
16
17        try{
18            if (val < 2){
19                throw {type: "lowNum", message: val}
20            } else if (val > 10 ){
21                throw {type: "highNum", message: val}
22            }
23            console.log("hello");
24        }
25        catch(e){
26            myFunc(e)
27        }
```

Fig 22.11

In fig 22.11, myFunc is an error handler. It handles the error thrown. If an exception is thrown within the “try block” an object is sent as the value of “e” on line 25 in the catch block. The catch block triggers myFunc and passes the received object as an argument to myFunc. myFunc handles the error appropriately.

CHAPTER TWENTY-THREE

ASYNC AND PROMISES

Async stands for asynchronous. Asynchronous is the opposite of synchronous. To understand asynchronous codes, let's first take a look at synchronous codes.

Synchronous codes

Synchronous codes run in sequence. The second line runs after the first. The third line runs after the second, and so on.

Example:

```
7    <script type="text/javascript">
8        console.log("first");
9        console.log("second");
10       console.log("third");
11   </script>
```

Fig 23.1

The codes above logs "first" to the console, followed by "second", followed by "third". So, it is said to run synchronously. This is good in some circumstances and bad in other circumstances.

Look at this scenario.

Imagine we have three lines of codes just like we have above, but the codes on the first line performs a time-consuming intensive task. It means that the codes on the second and third lines have to wait till the first line is done. This isn't good for performance and user experience.

Why not instruct JavaScript to execute the codes on the second and third lines before going back to complete the time-consuming intensive task on the first line? This is what asynchronous codes means. The codes aren't executed in the order they were written.

Example:

```
 8          console.log("first");
 9          setTimeout(function () {
10              console.log("second");
11          }, 3000);
12          console.log("third");
13          console.log("fourth");
```

Fig 23.2

The setTimeout above is introduced to explain what an asynchronous code is. I delayed the codes within the setTimeout for 3000 milliseconds (equal to 3 seconds). The codes above logs “first” to the console, followed by “third”, followed by “fourth”, finally “second”. The codes in fig 23.2 runs asynchronously.

Asynchronous codes lead us to another problem: In the image above (fig 23.2), what if the codes on line 12 depends on the result of line 10? This means we need to find a way to delay the codes on line 12 till the setTimeout is done. “Why not add another setTimeout on line 12 and set the delay time to 4000 milliseconds?” you may think. Well, that will work for the example above, because I manually delayed the codes on line 10 for 3000ms, but in real life scenario, you may not be able to predict the execution time. This is where “promise” comes in.

The way “promise” works is this: The codes on line 10 will say this to the codes on line 12 “Please don’t execute, wait for me, I promise to send you a message when I’m done”.

With promises, we will have this result:

“first” will be logged to the console, followed by “fourth” (Since line 10 promised line 12 and not line 13), followed by “second” and finally “third”.

Introduction to “promise”

I will use a story to illustrate this. This story will be broken into bits. Read and enjoy.

A short story

One day, Tunde wanted to do two things, first, make mango juice. Second, cook noodles. He had a blender and every other thing needed for the juice except mangoes. Tunde said to his brother, Ade “Do you remember the mango tree we saw on our way home yesterday? Go there and get some mangoes”.

Ade replied “Okay, off I go. I will ensure I get some mangoes”. He left afterwards. Tunde prepared the noodles before the arrival of Ade. (To be continued).

Tunde = Programmer/you

Ade = Request

Mango tree = Endpoint

Mangoes = data

You fetch data from an endpoint via a request.

Things to note:

- The cooking of noodles is independent of the availability of mangoes. For this reason, Tunde doesn’t need the **promise** from Ade to prepare the noodles.
- Ade left his house to fetch the mangoes from another location. In JavaScript, “promises” are usually used when you need to fetch data from another source or server (called an endpoint).
- How long will it take Ade to get the mangoes? No one knows. Most time, you don’t know how long it will take to fetch data. The more reason you need to use “promises”.
- Will Ade get some mangoes? Will he be allowed to pluck the mangoes? Are there mangoes on the tree? This is how JavaScript “promises” works. Not all promises get resolved, some get rejected.

HTTP status Codes

While trying to fetch data, you may encounter some errors. These errors come with messages and status codes to enable you know what went wrong. Below are some of the errors that can be caused by the client side of your application.

(To make this section interesting, instructors can ask students to read about status codes before the class. While in class, instructor reads out the instructor’s part asking students to give the corresponding status codes and messages. In cases where there is no instructor, students can form a group and choose a leader).

In continuation of our story.

- When Ade got to the mango tree, the owner quickly recognized Ade as the bully that beat her child some days back. She sent him away. Ade returned to Tunde to give him the feedback.

Instructor/Group leader: Which status code fits this description?

Answer: 400 **(**Bad request). This happens when there is a problem with the request itself. This is what just happened, Ade was the problem.

- Tunde gave Ade a new description “go get it now”, Tunde yelled.

“Ko ko ko”, Ade knocked at the gate. “What do you want?”, the gatekeeper responded.

“I want to pluck some mangoes”.

“You need a signed letter from the owner of this compound before you are allowed to pick anything. Do you have a letter?”

“No I don’t”

“You are not allowed to enter”.

Instructor: Which status code fits this description?

Answer: 401(Unauthorized). This happens when the user trying to access the data is not authorized to do so. The problem here is that: Ade didn’t receive the necessary permission before visiting the compound.

- Ade returned to Tunde to give him the response. Tunde replied “Go to the next street, count three poles to your right, you will see a mango tree”.

“Okay brother”, Ade replied.

On getting there, he saw a red clothe and a message tied to the tree. The message reads “Someone died after taking a mango from this tree”. Ade ran for his dear life.

Instructor: Which status code fits this description?

Answer: 403(Forbidden). This happens when the user is not allowed to visit an endpoint. Although it is similar to unauthorized, it sends a stronger message.

- Ade got home panting and explained what happened to Tunde. Tunde replied “sorry for the stress, there is a mango tree next to Tobi’s house”. “Off I go”, Ade replied.

Ade searched without a success. No mango tree found.

Instructor: Which status code fits this description?

Answer: 404 (Not Found). This occurs when the requested resource cannot be found at the location you provided.

- “I’m so tired”, Ade told Tunde. Tunde replied “wait a minute while I think”. “Oh, there is a mango tree beside the garage” Tunde shouted as if he just won a jackpot. “Yes, yes”, Ade replied enthusiastically and left.

Ade filled his bags with mangoes and returned home happily.

Instructor: Which status code fits this description?

Answer: 200 (OK). This means the request was successful.

Here is a link to read more about HTTP status codes.

At the client side, most times, you will be like Tunde who sends out Ade (request) to fetch mangoes (data) from a mango tree (endpoint) and expects a feedback (response) to make your mango juice (your application).

Sometimes, you will get the desired response (mangoes). Some other times, you will get an error (complaints).

For this reason, this book will only cover the process of sending a request, receiving a response and error handling from "promises". This book won't cover the server side.

While there is an inbuilt fetch API in JavaScript for this, I will use Axios - a lightweight JavaScript library-for three reasons.

- To introduce you to the use of libraries in JavaScript
- Axios is simpler to use than fetch API
- Error handling in Axios is better

What is a JavaScript library?

According to Wikipedia, a JavaScript library contains pre-written JavaScript codes for easier development of JavaScript-based application. Axios is an example of such. To use a library, you can either download it into your project or use URL to link the library with your project (this is called CDN option). I will use the CDN option to link axios with my codes. Ensure you read the documentation of a library before using it. This makes life easy for you. Here is axios documentation.

This is what I want to do: I will use axios to fetch data from an endpoint. This is the link to the endpoint. This fetches a JSON data, the goodnews is that axios will convert this JSON data to JavaScript object on its own. The object fetched contains information about a user but I am only interested in the username of the user.

```
7     <button onclick="displayUsername()">fetch data</button>
8     <script type="text/javascript">
9         const displayUsername = ()=>{
10            console.log(`I am ${username}`);
11            console.log(`Good morning`);
12
13        }
14    </script>
```

Fig 23.3

There is a button on line 7, when the button is clicked, displayUsername is triggered. It logs "I am (username)" the value of username isn't known yet. The username will be fetched from a different server. Line 11 logs "Good morning" to the console.

Note two things

- The codes on line 10 depends of the username
- The codes on line 11 doesn't

Since we are fetching the username from a different server, it will take some time. For the sake of performance, I want to delay the codes on line 10 from running until the username is fetched but not the codes on line 11 since line 11 is independent of the username.

```
7     <button onclick="displayUsername()">fetch data</button>
8     <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
9     <script type="text/javascript">
10        const displayUsername = ()=>{
11            axios.get('https://jsonplaceholder.typicode.com/users/1')
12            .then(
13                function (res) {
14                    let username = res.data.username;
15                    console.log(`I am ${username}`);
16                }
17            );
18            console.log(`Good morning`);
19        }
20    </script>
```

Fig 23.4

"What is this you put up here?" you just said to yourself. I will explain every bit of it in a second.

Explanation

There is a button on line 7 that triggers displayUsername when clicked.

Line 8: I want to use axios in my codes. The codes on line 8 links axios codes with mine so I can use some axios functions within my codes.

Line 10- 19: Function displayUsername starts on line 10 and ends on line 19, it means all the codes within the function will be executed when the button is clicked.

Line 11: On line 8, I brought axios codes into my application, now is the time to use it. Back to our mango story, Tunde wanted to fetch mangoes from a mango tree, so he sent Ade. I want to fetch data, so I will send axios. From axios documentation , axios has many methods. One of the methods is "get". This method gets/fetches data from a source (you can read on HTTP methods to learn more).

Tunde gave the location of the mango tree to Ade, I also need to give axios the location of the data. This location is given as URL. The URL points to the endpoint where the data is located. It is a string supplied as the first argument (the only argument in the codes in fig 23.4) of axios get method. The codes on line 11 ask axios to fetch data from https://jsonplaceholder.typicode.com/users/1. Fetching data from another server takes time,

so axios goes to the URL to fetch the data promising JavaScript that it will return with the data (It may or may not return with the data).

Line 12: A “then” method is chained to the “get” method. The “then” method starts on line 12 and ends on 17. **The codes in the “then” block will be executed if and only if axios visits the endpoint successfully.** You place whatever you want to delay till the promise is fulfilled within the “then” method.

Let’s talk about the codes within the “then” block.

There is a parameterized function within the “then” block (a callback). The parameterized function starts on line 13 and ends on line 16 as shown in fig 23.4. I named the parameter within the parameterized function “res”, you can give it any name of your choice.

Axios saves whatever data it fetches from the endpoint into “res” (at this point, axios fulfils the promise by fetching the data successfully).

The fetched data is converted from JSON to JavaScript object before axios saves it into “res”.

On line 14, I extracted the username from the fetched data and saved it into variable username.

I logged the value of “username” to the console on line 15.

Line 18: The codes on line 18 is not within the “then” block, this means JavaScript doesn’t need to wait for the promise before running it. This is the cooked noodles in our mango story.

For this reason, the codes on line 18 is executed before the codes on line 15.

What if the promise fails for one reason or the other?

Promise may get rejected/fail for different reasons. The reason includes but not limited to: incorrect URL, unauthorized to make the request and other reasons stated under the HTTP status codes.

How do we handle that? You can chain a “catch” method to handle this. The codes within the catch method is executed if the promise is rejected.

```
    <button onclick="displayUsername()">fetch data</button>
    <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
    <script type="text/javascript">
        const displayUsername = ()=>{
            axios.get('https://jsonplaceholder.typicode.com/users/1')
            .then(
                function (res) {
                    let username = res.data.username;
                    console.log(`I am ${username}`);
                }
            ).catch(
                function (err) {
                    console.log(`Something went wrong ${err}`);
                }
            );
            console.log(`Good morning`);
        }
    </script>
```

Fig 23.5

A parameterized function is also placed with the "catch block". I named the parameter within the parameterized function "err" (You can give it any name of your choice). The error encountered by axios is saved into "err".

Summary: JavaScript will execute either the "then block" or the "catch block" but not the two. The "then block" is executed if the promise is fulfilled. The "catch block" is executed if the promise is rejected.

The codes within both blocks wait for the arrival of the promise.

The functions within both blocks are known as callbacks. You can read more on callbacks in chapter twenty.

Async await

A more elegant way to write promises is to use the "async" keyword. It is shorter and neater. According to JavaScript documentation, async makes our codes look like old-school synchronous codes (I will explain this later).

A function returns a promise just by adding the "async" keyword to the function. Another keyword that is mostly used with the "async" keyword is the "await" keyword. The "await" keyword only works within an async function. "Await" pauses the codes within the async function till the promise is either fulfilled or rejected.

In simpler terms: Promises aren't fulfilled immediately, they are fulfilled or rejected sometime in the future. Adding the "async" keyword to a function tells JavaScript that the function will return a promise. Using "await" within the async function tells JavaScript to stop the execution of codes on that line (that contains the await keyword) till the promise is resolved (fulfilled or rejected).

Example:

```
7     <button onclick="displayUsername()">fetch data</button>
8     <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
9     <script type="text/javascript">
10        const displayUsername = async ()=>{
11            console.log("hi");
12            let res = await axios.get('https://jsonplaceholder.typicode.com/users/1');
13            let username = res.data.username;
14            console.log(`I am ${username}`);
15            console.log("welcome");
16        }
17    </script>
```

Fig 23.6

When the button is clicked, displayUsername is triggered. Do you notice the “async” keyword on line 10? displayUsername is an asynchronous function.

Line 11: “hi” is logged to the console.

Do you notice the “await” keyword on line 12? The presence of the “await” keyword stops the execution of codes on the line till the promise is resolved.

Probably you are thinking JavaScript will proceed to the line 12, instead of waiting for the data to be fetched on line 11. Well, it won’t. Recall that I wrote that async makes our codes behave like old-school **synchronous** codes. It makes our codes get executed in sequence. Once JavaScript sees the “await” keyword in an async function, it delays every other codes within the async function till the promise is resolved. This means the codes on line 12 to line 15 is delayed till the data on line 12 is fetched. The only codes that don’t wait for the “promise” are the codes before the await keyword (line 11).

Error handling in an async function

Please note that the execution of the codes on and after the await keyword is delayed until the promise is resolved (either fulfilled or rejected). It therefore means the codes is executed even if the promise fails (rejected). This can cause some problems to our application. It is important we handle the error appropriately. A try and catch block is added to an async function to handle the errors.

Example:

```
<button onclick="displayUsername()">fetch data</button>
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
<script type="text/javascript">
    const displayUsername = async ()=>{
        try{
         console.log("hi");
         let res= await axios.get('https://jsonplaceholder.typicode.com/users/1');
         let username = res.data.username;
         console.log(`I am ${username}`);
         console.log("welcome");
        } catch(err){
            console.log(`Something went wrong ${err}`);
        }
    }
</script>
```

Fig 23.7

If an error is thrown at any point within the “try block”, it is caught by the “catch block”. Read more about try and catch in chapter twenty-two.

THANK YOU FOR TAKING YOUR TIME TO READ THROUGH MY TEXTBOOK.

I have a gift for you. Checkout the workbook added to this textbook and go through the exercises therein.

To contact me

Email: tayeabidakun@gmail.com

LinkeIn Profile Link https://linkedin.com/in/taye-abidakun-64649117b

Twitter username: @TayeAbidakun

Medillery Username: @TayeAbidakun

JAVASCRIPT SIMPLIFIED

About The Author

Abidakun Taye is the CEO of medillery.com , a company that offers branding to SMEs at affordable costs. He is committed to creating values and impacting lives positively using technology.

He had his first degree in Biochemisty at Ladoke Akintola University of Technology. He had a post graduate Diploma degree in Computer science at university of Ilorin

He owes his programming skills and knowledge to SQI college of ICT where he got a professional Certificate in software engineering. He was retained after his training and went on to become the Head of software engineering department at the same institution.
He trained hundreds of developers in JavaScript, React JS, Vue JS, Node JS, Angular , PHP, Laravel, Flutter and other technologies during his period of service at the institution.

JAVASCRIPT WORKBOOK

While writing the JavaScript Simplified Textbook, I realized the need for a workbook. I have made the workbook a short and precise one, in fact, not every chapter in the textbook is present in this workbook. I felt the textbook alone won't do enough justice to some chapters, only such chapters are included in this workbook.

The best way to use the workbook is this: After studying a chapter in the textbook, open the workbook and solve the exercises under the chapter.

The most important thing in this textbook is not for your solution to be the same as mine but for your solution to solve the given task. The beauty of programming is that, we can provide different solutions to solve a given problem.

Have a great time.

Taye Abidakun

tayaeabidakun@gmail.com

CHAPTER ONE

WRITING YOUR FIRST JAVASCRIPT CODES

Task 1.1

Create a div in HTML, place 40 in the div using JavaScript.

Solution

```
 6  <body>
 7      <div id="display"></div>
 8      <script type="text/javascript">
 9          display.innerText = 40;
10      </script>
11  </body>
```

Fig 1.1

Task 1.2

Create a textarea in HTML, using JavaScript, place the result of 2* 8 (16) in the textarea.

Solution

```
 6 ▾ <body>
 7      <textarea id="result"></textarea>
 8      <script type="text/javascript">
 9          result.value = 2 * 8;
10      </script>
11  </body>
```

Fig 1.2

CHAPTER TWO

VARIABLES

Task 2.1

Create a variable, store 15 in the variable. Display the value of the variable in a HTML div.

Solution

```
<body>
    <div id="display"></div>
    <script type="text/javascript">
        var val = 15;
        display.innerText = val;
    </script>
</body>
```

Fig 2.1

Task 2.2

Create a variable, store 3 in the variable. Change the value of the variable to 7. Display the result in a HTML button.

Solution

```
<body>
    <button id="result"></button>
    <script type="text/javascript">
        var num = 3;
        num = 7;
        result.innerText = num;
    </script>
</body>
```

Fig 2.2

Task 2.3

Create a first variable with a value of 12. Create a second variable such its value is three times the value of the first variable. Display the second variable in a HTML h2 tag.

Solution

```
<body>
    <h2 id="display"></h2>
    <script type="text/javascript">
        var first = 12;
        var second = first * 3;
        display.innerText = second;
    </script>
</body>
```

Fig 2.3

CHAPTER THREE

NAMING CONVENTIONS

Task 3.1

Using Pascal casing, create a variable with a label of “my school”, insert the name of your school in the variable. Use an alert to display the variable created.

Solution

```
 8     <script type="text/javascript">
 9         var MySchool = "SQI";
10         alert(MySchool);
11     </script>
```

Fig 3.1

Task 3.2

Using camel casing, create a variable with a name of “students in class”. Let the variable holds a value of 30. Display the value of the variable in a HTML div.

Solution:

```
 6 <body>
 7     <div id="display"></div>
 8     <script type="text/javascript">
 9         var studentsInClass = 30;
10         display.innerText = studentsInClass;
11     </script>
12 </body>
```

Fig 3.2

CHAPTER FOUR

DATATYPES

Task 4.1

What is the inbuilt method used to check the datatype of a value in JavaScript?

Answer: typeof

Task 4.2

Which is of these is not a string?

a. “4”
b. true
c. “Taye”

Answer: b. true

Task 4.3

What will be the output of the codes below?

```
7    <script type="text/javascript">
8        var a = 1.1;
9        alert(typeof(a));
10   </script>
```

Fig 4.1

a. number b. string c. boolean d. object

Answer: a. number

Task 4.4

What will be the output of the codes below?

```
7    <script type="text/javascript">
8        var a = null;
9        alert(typeof(a));
10   </script>
```

Fig 4.2

a. undefined b. null c. boolean d. object

Answer: d. object

CHAPTER SIX

JAVASCRIPT OPERATORS PART 1.

Task 6.1

Create a variable with a name of myNum, set the value to 3. Reduce the value of myNum by 2 so it becomes 1. Display the result in a HTML input tag.

Solution

```
6  <body>
7      <input id="display">
8      <script type="text/javascript">
9          var myNum = 3;
10         myNum-=2;
11         display.value = myNum;
12     </script>
13 </body>
```

Fig 6.1

Task 6.2

Create a first variable, set the value to 13. Divide the variable by 4 and save the remainder in a second variable. Display the result in a HTML div.

Solution

```
6  <body>
7      <div id="result"></div>
8      <script type="text/javascript">
9          var first = 13;
10         var second = first%4;
11         result.innerText = second;
12     </script>
13 </body>
```

Fig 6.2

CHAPTER SEVEN

JAVASCRIPT OPERATORS PART 2

Task 7.1

Create a div in HTML. Create a JavaScript variable to hold your name. Concatenate the name with “prays for world peace”. For instance, I am Taye, so I will have “Taye prays for world peace”.

Solution

```
<body>
    <div id="result"></div>
    <script type="text/javascript">
        var name = "Taye";
        result.innerText = name + " prays for world peace";
    </script>
</body>
```

Fig 7.1

CHAPTER EIGHT

JAVASCRIPT EVENTS AND FUNCTIONS

Task 8.1

Create a button in HTML. Write JavaScript codes such that when a user clicks on the button, the browser will alert “Hello world”.

Solution

```
<body>
    <button onclick="myFunc()">click</button>
    <script type="text/javascript">
        function myFunc() {
            alert("Hello world");
        }
    </script>
</body>
```

Fig 8.1

Task 8.2

Create two inputs (the type attribute of both inputs should be number), a button and a div in HTML. When a user clicks on the button, find the product of the two input values and place the result in the div.

Solution

```
<body>
    <input type="number" id="first">
    <input type="number" id="second">
    <div id="result"></div>
    <button onclick="multiplyNums()">multiply</button>

    <script type="text/javascript">
        function multiplyNums() {
            var firstInp = first.value;
            var secondInp = second.value;
            result.innerText = firstInp * secondInp;
        }
    </script>
</body>
```

Fig 8.2

Task 8.3

Create a function that receives a number as argument and returns half the received number

Solution

```
7    <script type="text/javascript">
8        function reduceByHalf(num) {
9            return num/2;
10       }
11       console.log(reduceByHalf(10));
12   </script>
```

Fig 8.3

CHAPTER NINE

JAVASCRIPT OPERATORS PART 3 AND IF STATEMENT

Task 9.1

Create an input and a button in HTML. When a user clicks on the button, get the value in the input. If the value is greater or equal to 18, alert “You can vote”. If it isn’t, alert “Underage voting isn’t allowed”.

Solution

Step 1:

I will start by creating the HTML part

```
6  <body>
7      <input type="number">
8      <button>check eligibility</button>
9  </body>
```

Fig 9.1

Step 2: Add a function and an event that will trigger the function when the button is clicked

```
6  <body>
7      <input type="number">
8      <button onclick="checkEligibility()">check eligibility</button>
9      <script type="text/javascript">
10         function checkEligibility() {
11
12         }
13     </script>
14 </body>
```

Fig 9.2

Step 3: Retrieve the value of the input when the button is clicked. To do this, I will add an id to the input

```
6  <body>
7      <input type="number" id="val">
8      <button onclick="checkEligibility()">check eligibility</button>
9      <script type="text/javascript">
10         function checkEligibility() {
11             var age = val.value;
12         }
13     </script>
14 </body>
```

Fig 9.3

Step 4: Write an “if statement” to check the value of age

```
<body>
    <input type="number" id="val">
    <button onclick="checkEligibility()">check eligibility</button>
    <script type="text/javascript">
        function checkEligibility() {
            var age = val.value;
            if ( age>=18 ) {
                alert("You can vote");
            } else{
                alert("Underage voting isn't allowed");
            }
        }
    </script>
</body>
```

Fig 9.4

Task 9.2: Create a div, a button and an input in HTML. Write a script that gets the value of the input when the button is clicked. If the number of characters of the input’s value is between 0-7, display “too short” in the div. If the number of characters of the input’s value is more than 20, display “too long” in the div. If the number of characters of the input’s value is between 8-20, display “perfect” in the div.

Solution

Step 1

Create the HTML part

```
    <input type="text">
    <button>check length</button>
    <div></div>
```

Fig 9.5

Step 2: Add a function and an event that will trigger the function when the button is clicked

```
<body>
    <input type="text">
    <button onclick="checkLength()">check length</button>
    <div></div>
    <script type="text/javascript">
        function checkLength() {

        }
    </script>
</body>
```

Fig 9.6

Step 3: Retrieve the value of the input when the button is clicked. To do this, I will add an id to the input

```
<body>
    <input type="text" id="val">
    <button onclick="checkLength()">check length</button>
    <div></div>
    <script type="text/javascript">
        function checkLength() {
            var myInp = val.value;
        }
    </script>
</body>
```

Fig 9.7

Step 4: Write an “if statement” to display the result

```
<body>
    <input type="text" id="val">
    <button onclick="checkLength()">check length</button>
    <div id="display"></div>
    <script type="text/javascript">
        function checkLength() {
            var myInp = val.value;
            var inpLength = myInp.length;
            if (inpLength < 8) {
                display.innerText = "too short";
            } else if (inpLength < 20) {
                display.innerText = "perfect";
            } else{
                display.innerText = "too long";
            }
        }
    </script>
</body>
```

Fig 9.8

Task 9.3

Build a school grading system. If the entered score is above 100 or below 0, display “Out of range”. If the score is between 70 to 100, display “Excellent”. If the score is between 60 to 70, display “Good”. If the score is between 50 to 60, display “Average”. If the score is between 45 to 50, display “Fair”. If the score is between 40 to 45, display “Poor”. If the score is between 0 to 40, display “Fail”.

Solution

There should be an input where teacher will type student’s score, therefore, I need a HTML input. I will add a button that triggers an event when clicked. Also, I will add a div that displays the result.

Step 1: Setup the HTML part

```
    <input type="number">
    <button>grade student</button>
    <div></div>
```

Fig 9.9

Step 2: Add a function and an event that will trigger the function when the button is clicked

```
<body>
    <input type="number" id="grade">
    <button onclick="gradeStudent()">grade student</button>
    <div id="result"></div>
    <script type="text/javascript">
        function gradeStudent() {

        }
    </script>
</body>
```

Fig 9.10

Step 3: Retrieve value from input and display the right result when the button is clicked.

```
    <script type="text/javascript">
        function gradeStudent() {
            var getGrade = grade.value;
            if (getGrade > 100) {
                result.innerText = "Out of range";
            } else if (getGrade >= 70 && getGrade <= 100) {
                result.innerText = "Excellent";
            } else if (getGrade >= 60 && getGrade < 70) {
                result.innerText = "Good";
            } else if (getGrade >= 50 && getGrade < 60) {
                result.innerText = "Average";
            } else if (getGrade >= 45 && getGrade < 50) {
                result.innerText = "Fair";
            } else if (getGrade >= 40 && getGrade < 45) {
                result.innerText = "Poor";
            } else if (getGrade >= 0 && getGrade < 40) {
                result.innerText = "Fail";
            } else if (getGrade < 0 ) {
                result.innerText = "Out of range";
            }
        }
    </script>
```

Fig 9.11

The codes above can be written in a shorter way. Here it is

```
10    <script type="text/javascript">
11        function gradeStudent() {
12            var getGrade = grade.value;
13            if (getGrade > 100 || getGrade < 0) {
14                result.innerText = "Out of range";
15            } else if (getGrade >= 70) {
16                result.innerText = "Excellent";
17            } else if (getGrade >= 60) {
18                result.innerText = "Good";
19            } else if (getGrade >= 50) {
20                result.innerText = "Average";
21            } else if (getGrade >= 45) {
22                result.innerText = "Fair";
23            } else if (getGrade >= 40) {
24                result.innerText = "Poor";
25            } else if (getGrade >= 0) {
26                result.innerText = "Fail";
27            }
28        }
29    </script>
```

Fig 9.12

Why will this work?

Explanation: If the value of getGrade is, let's say, 120. It meets all the conditions in fig 9.12 but remember that the condition in an "else if" will only be read if the condition in the previous "if" is not met. Therefore, it will execute the codes on line 14 and ignore other conditions.

If the value of getGrade, let's say 76, it meets all the conditions in fig 9.12 except the condition on line 13. For this reason, it will execute the codes on line 16 and ignore others.

CHAPTER TEN

OBJECTS

Task 10.1

Create an object, give it two properties (state and country), set the value of the properties to your state and country respectively.

Solution

```
10      <script type="text/javascript">
11          var myDetails= {state: "Ondo", country: "Nigeria"};
12      </script>
```

Fig 10.1

Task 10.2

Add a method that receives an argument to the object in fig 10.1, such that when the method is triggered, it sets the argument received as the value of the "name" property of the object (myDetails).

Solution

```
10      <script type="text/javascript">
11          var myDetails= {
12                              state: "Ondo",
13                              country: "Nigeria",
14                              setName(name){
15                                  this.name = name;
16                              }
17                          };
18      </script>
```

Fig 10.2

Task 10.3

Create two inputs and a button that adds a new key-value pair to the object (myDetails) in fig 10.2 when the button is clicked.

Solution

```
<body>
    <input id="keyName">
    <input id="vals">
    <button onclick="addProps()">Add new property</button>
    <div id="result"></div>
    <script type="text/javascript">
        var myDetails= {
                                state: "Ondo",
                                country: "Nigeria",
                                setName(name){
                                    this.name = name;
                                }
                            };
        function addProps() {
            var key = keyName.value;
            var val = vals.value;
            myDetails[key] = val;
            console.log(myDetails);
        }
    </script>
</body>
```

Fig 10.3

Task 10.4:

Edit the codes in fig 10.3 such that when a key already exists, it will alert “Key already exists”.

Solution

```
<script type="text/javascript">
    var myDetails= {
                            state: "Ondo",
                            country: "Nigeria",
                            setName(name){
                                this.name = name;
                            }
                        };
    function addProps() {
        var key = keyName.value;
        var val = vals.value;
        if (key in myDetails) {
            alert("Key already exists");
        } else{
            myDetails[key] = val;
            console.log(myDetails);
        }
    }
</script>
```

Fig 10.4

CHAPTER ELEVEN

ARRAY AND LOOP

Read on JavaScript array slice method and how it works. You will recreate one. Let's take it step by step.

Task 11.1

Create a parameterized function named "mySlice". Make the function receives two arguments, the first argument is an array while the second argument is a number. Finally, let the function return an array. For instance, if the first function receives [3, 4, 7, 9, 11] as the first argument and 2 as the second argument, it will return a new array starting from index 2. The returned array will be [7, 8, 9].

Example 1: [8, 1, 5, 12, 3]- first parameter. 3- second parameter. Returned array will be [12, 3].

Example 2: ["Taye", "Kenny", "Idowu", "Alaba"] – first parameter. 1- second parameter. Returned array will be ["Kenny", "Idowu", "Alaba"].

You must not use JavaScript inbuilt slice method in your solution.

Solution

```
11      <script type="text/javascript">
12          function mySlice(arr, num){
13              var newArray = [];
14              for (var i = num; i < arr.length; i++) {
15                  newArray.push(arr[i]);
16              }
17              return newArray;
18          }
19      </script>
```

Fig 11.1

Task 11.2

Add one more parameter to your function (mySlice) so that it receives three arguments. The first argument will be an array. The second and third arguments will be numbers. The second will be the starting point (inclusive) while the third will be the ending point (not inclusive) just like JavaScript inbuilt slice method.

Example 1: [8, 1, 5, 12, 3]- first parameter. 1- second parameter. 3- third parameter. Returned array will be [1, 5].

Example 2: [“Taye”, “Kenny”, “Idowu”, “Alaba”] – first parameter. 1- second parameter. 2- third parameter. Returned array will be [“Kenny”].

Again, don’t use JavaScript inbuilt slice method.

Solution

```
11    <script type="text/javascript">
12        function mySlice(arr, start, end){
13            var newArray = [];
14            for (var i = start; i < end; i++) {
15                newArray.push(arr[i]);
16            }
17            return newArray;
18        }
19    </script>
```

Fig 11.2

Task 11.3

There is a problem with the solution in fig 11.2 . If the value of the third parameter (end) is higher than the length of the first parameter (arr), “undefined” will be included as part of the values in the returned array.

Example: [3, 4, 5]- first parameter. 1- second parameter. 5- third parameter. Returned array will be [4, 5, undefined, undefined].

Your next task is to ensure that only values within the initial array are included in the returned array and not undefined.

Solution

```
12        function mySlice(arr, start, end){
13            var newArray = [];
14            for (var i = start; i < end && i < arr.length; i++) {
15                newArray.push(arr[i]);
16            }
17            return newArray;
18        }
```

Fig 11.3

Task 11.4

```
<body>
    <button onclick="displayStudent()">display students</button>
    <ol id="display"></ol>
    <script type="text/javascript">
        var students = [
                                {
                                    name: "Kunle",
                                    dept: "software",
                                    school: "SQI"
                                },
                                {
                                    name: "Jide",
                                    dept: "CSE",
                                    school: "LAUTECH"
                                }
                            ];
    </script>
</body>
```

Fig 11.4

Fig 11.4 contains array of objects, a button and an ordered list (ol). Your task is this: Create a function called displayStudent such that when users click on the button, the values in the array are displayed in the ordered list as shown below.

1. My name is Kunle studying software at SQI
2. My name is Jide studying CSE at LAUTECH

Solution

```
        function displayStudent() {
            for (var i = 0; i < students.length; i++) {
                display.innerHTML += "<li>My name is "+ students[i].name+
                                    " studying "+students[i].dept + " at "+
                                    students[i].school+"</li>";
            }
        }
```

Fig 11.5

Task 11.5

With the solution in fig 11.5, when the button is clicked multiple times, you have an unwanted result. For instance, if the button is clicked three times, you have this result

1. My name is Kunle studying software at SQI
2. My name is Jide studying CSE at LAUTECH
3. My name is Kunle studying software at SQI
4. My name is Jide studying CSE at LAUTECH
5. My name is Kunle studying software at SQI
6. My name is Jide studying CSE at LAUTECH

Your next task is to remove this bug so there won't be repetition when the button is clicked multiple times.

Solution

```
22          function displayStudent() {
23              display.innerHTML = "";
24              for (var i = 0; i < students.length; i++) {
25                  display.innerHTML += "<li>My name is "+ students[i].name+
26                                       " studying "+students[i].dept + " at "+
27                                       students[i].school+"</li>";
28              }
29          }
```

Fig 11.6

Line 23 helps with that. The innerHTML of "display" is cleared each time the function is triggered.

Task 11.6

Add three inputs and a button to the codes in fig 11.6 such that when users click on the newly added button, a new object is added to the students array. The value of the first, second and third inputs are respectively set as the values of "name", "dept" and "school" of the newly created object.

Solution

Here is the HTML part

```
 7      <input id="setName">
 8      <input id="setDept">
 9      <input id="setSchool">
10      <button onclick="addStudent()">Add new Student</button>
11      <button onclick="displayStudent()">display students</button>
12      <ol id="display"></ol>
```

Fig 11.7

Here is the function definition of addStudent

```
34          function addStudent() {
35              var newStudent = {
36                  name: setName.value,
37                  dept: setDept.value,
38                  school: setSchool.value
39              };
40              students.push(newStudent);
41          }
```

Fig 11.8

CHAPTER FOURTEEN

DESTRUCTURING

Task 14.1

Here is an array: [5, 10, 20]. Your task is to save only the third value in the array (20) in a variable labelled myNum using destructuring.

Solution

```
 9      <script type="text/javascript">
10          let [ , , myNum] = [5, 10, 20];
11      </script>
```

Fig 14.1

Task 14.2

Here is an object {country: “Nigeria”, state: “Ondo”}. Your task is to save the value of state (“Ondo”) in a variable labelled “province” using destructuring.

Solution

```
 9      <script type="text/javascript">
10          let {state: province} = {country: "Nigeria", state: "Ondo"};
11      </script>
```

Fig 14.2

CHAPTER FIFTEEN

SPREAD OPERATOR

Task 15.1

```
9    <script type="text/javascript">
10       let firstSet = ["Peter", "James", "John"];
11       let secondSet = ["Foluke", "Sade", "Kemi"];
12   </script>
```

Fig 15.1

There are two arrays in fig 15.1. Your task is to create a new array called allStudents. allStudents must contain all the values in both array

Solution

```
9    <script type="text/javascript">
10       let firstSet = ["Peter", "James", "John"];
11       let secondSet = ["Foluke", "Sade", "Kemi"];
12       let allStudents = [...firstSet, ...secondSet];
13   </script>
```

Fig 15.2

Task 15.2

```
9    <script type="text/javascript">
10       let person = {name: "Buhari", age: 79};
11       let state = {name: "Ondo", country: "Nigeria"};
12   </script>
```

Fig 15.3

There are two objects in fig 15.3, your task is to create a new object from the two such that the new object will be {name: “Buhari”, country: “Nigeria”, age: 79}

Solution

```
9    <script type="text/javascript">
10       let person = {name: "Buhari", age: 79};
11       let state = {name: "Ondo", country: "Nigeria"};
12       let president = {...state, ...person};
13   </script>
```

Fig 15.4

Note that I spread “state” in the new object before spreading “person”. This is to ensure that the “name” property in “person” overrides the “name” property in “state”.

CHAPTER SIXTEEN

REST OPERATOR

Task 16.1

What will be the value of val in fig 16.1

```
9     <script type="text/javascript">
10        function myFunc([a, ...val]) {
11            console.log(val);
12        }
13        myFunc([5, 6, 7, 8]);
14    </script>
```

Fig 16.1

Solution

[6, 7, 8]

Task 16.2

What will be the value of val in fig 16.2

```
9     <script type="text/javascript">
10        function myFunc({name, country, ...val}) {
11            console.log(val);
12        }
13        myFunc({name: "Taye", state: "Ondo", country: "Nigeria"});
14    </script>
```

Fig 16.2

Solution

{state: “Ondo”}

CHAPTER EIGHTEEN

CLASSES AND CONSTRUCTORS

Task 18.1

Create a class called “Info”. Let the class has two properties (state and country) and a method (displayState). Insert any values of your choice in the properties. Log the value of the “state” property to the console within the method (displayState).

Solution

```
<script type="text/javascript">
    class Info{
        state = "Ondo";
        country = "Nigeria";
        displayState(){
            console.log(this.state);
        }
    }
</script>
```

Fig 18.1

Task 18.2

Create a new instance of the class and trigger “displayState”.

Solution

```
<script type="text/javascript">
    class Info{
        state = "Ondo";
        country = "Nigeria";
        displayState(){
            console.log(this.state);
        }
    }
    let myInfo = new Info;
    myInfo.displayState();
</script>
```

Fig 18.2

Task 18.3

Add a constructor to the class in fig 18.2 so that the value of “state” property is dynamic.

Solution

```
<script type="text/javascript">
    class Info{
        constructor(state){
            this.state = state;
            this.country = "Nigeria";
        }
        displayState(){
            console.log(this.state);
        }
    }
</script>
```

Fig 18.3

Task 18.4

Create a child’s class from the “Info”. The child’s class will have a property called “localGovt”. Add a constructor to the child’s class such that the constructor will make the value of “localGovt” dynamic.

Solution

```
 9      <script type="text/javascript">
10          class Info{
11              constructor(state){
12                  this.state = state;
13                  this.country = "Nigeria";
14              }
15              displayState(){
16                  console.log(this.state);
17              }
18          }
19          class Details extends Info{
20              constructor(state, lg){
21                  super(state);
22                  this.localGovt = lg;
23              }
24          }
25      </script>
```

Fig 18.4

CHAPTER TWENTY

HIGHER ORDER FUNCTIONS (HOF) AND CALLBACKS

Task 20.1

Here is an array [3, 14, 5, 1, 2, 9, 12, 7, 8, 12]. Your task is to create a new array that contains all numbers greater than 6 in the given array. That is, [14, 9, 12, 7, 8, 12]

Solution

```
 9      <script type="text/javascript">
10          let arr = [3, 14, 5, 1, 2, 9, 12, 7, 8, 12];
11          let newArray = arr.filter(val=>val>6);
12          console.log(newArray);
13      </script>
```

Fig 20.1

Task 20.2

Using the arr provided in fig 20.1, create a new array that contains all the values where index is greater than or equal to 3.

Solution

```
 9      <script type="text/javascript">
10          let arr = [3, 14, 5, 1, 2, 9, 12, 7, 8, 12];
11          let newArray = arr.filter((_, i)=> i>=3 );
12          console.log(newArray);
13      </script>
```

Fig 20.2

The underscore in fig 20.1 is a way to ignore the first parameter.

Task 20.3

```
<script type="text/javascript">
    let arr = [
                {
                    id: 1,
                    name: "Taye",
                    dept: "software",
                    school: "SQI"
                },
                {
                    id: 2,
                    name: "Kenny",
                    dept: "CSE",
                    school: "LAUTECH"
                },
                {
                    id: 3,
                    name: "Idowu",
                    dept: "Multimedia",
                    school: "SQI"
                }
            ];
</script>
```

Fig 20.3

The fig 20.3 above contains an array of objects with a unique id. Your task is to save the object with an id of 2 in a new variable.

Solution

```
let person = arr.find(val=>val.id ==2);
```

Fig 20.4

Task 20.4

From the array of objects provided in fig 20.3, your task is to save objects with id of 3 or more in a new variable.

Solution

Fig 20.5

"Why not use 'find' instead of 'filter'", you may ask. From the textbook, "find" will only return one value while "filter" returns an array. With the codes we have in fig 20.5, only one value meets the condition and you may be tempted to use "find", but don't. The reason is that you need to include some flexibility to your codes. What if more objects are added to this array? "find" may stop providing solution to the task at hand. This is the reason I used "filter". The question asked us to save objects with id of 3 or more, meaning that, it isn't expected to be a single object.

CHAPTER TWENTY-THREE

ASYNC AND PROMISES

Task 23.1

Create a div and a button in HTML. When a user clicks on the button, fetch a cat fact from this URL (https://catfact.ninja/fact) and display it in the div.

Solution

```
<body>
    <button onclick="getFact()">get cat fact</button>
    <div id="display"></div>
    <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
    <script type="text/javascript">
        const getFact = ()=>{
            axios.get('https://catfact.ninja/fact')
            .then(res=>{
                display.innerText = res.data.fact;
            })
            .catch(err=>{
                console.log(err);
                display.innerText = "Something went wrong";
            });
        }
    </script>

</body>
```

Fig 23.1

Task 23.2

Write the codes in fig 23.1 using async await.

Solution

```
<body>
    <button onclick="getFact()">get cat fact</button>
    <div id="display"></div>
    <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
    <script type="text/javascript">
        const getFact = async ()=>{
            try{
                let res = await axios.get('https://catfact.ninja/fact');
                display.innerText = res.data.fact;
            } catch(err){
                console.log(err);
                display.innerText = "Something went wrong";
            }
        }
    </script>
</body>
```

Fig 23.2

Thank you for reading through the workbook.

I wish you success in your programming journey.

You can reach me on

Email: tayeabidakun@gmail.com

LinkeIn Profile Link https://linkedin.com/in/taye-abidakun-64649117b