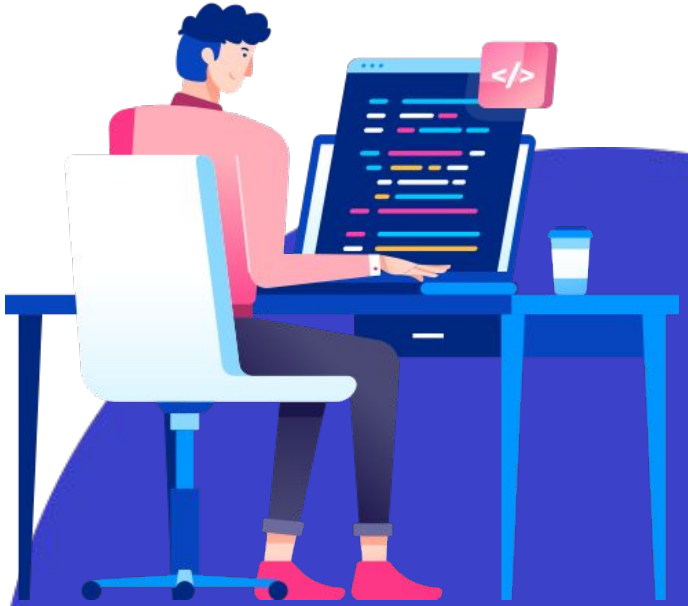
PROGRAMMING FUNDAMENTALS

JS Fundamentals 1

DIPLOMA IN FULL-STACK DEVELOPMENT

Certificate in **Computing Fundamentals**



Loops

DEFINITION

Repetition aka. Loop control structure.
Loops check a condition. If it returns true, a code block will run. Then the condition will be checked again and if it still remains true, the code block will run again. It repeats until the condition returns false

Loops Repeats tasks while condition(s) is/are met

DEFINITION

Each time we perform a loop action, it is known as an iteration.

Loops Repeats tasks while condition(s) is/are met

REMEMBER

Loop, provide a way to repeat the same set of actions over and over again.

#Simplify #RepetitiveTasks

For Loops

*Example: Display the word "Practice: x" using **for loop***

KEYWORD

INITIALIZATION

CONDITION

INCREMENT/ UPDATE

OPENING CURLY BRACE *(START OF LOOP)*

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

CODE TO EXECUTE DURING LOOP *(INSIDE THE LOOP)*

CLOSING CURLY BRACE *(END OF LOOP)*

Output

Practice: 0
Practice: 1
Practice: 2
Practice: 3
Practice: 4
Practice: 5
Practice: 6
Practice: 7
Practice: 8
Practice: 9

Loop stops because i < 10

Q. How do we loop 10 times and get number 10?

INITIALISATION (Begin)

Create a variable and set it to 0
Acts as a counter

```
let i = 0;
```

Variable is only created the **first time** the loop is run.

CONDITION

Loop continues to run until the counter reaches a specific number

```
i < 10;
```

The value of i was initially set to 0, so in this case the loop will run 10 times before stopping.

INCREMENT / UPDATE (Step)

Every time the loop has run the statements in the curly braces, it adds **one to the counter**

```
i++
```

One is added to the counter using the increment (++) operator.

It is also possible for loops to count downwards using the decrement operator (--)
Runs **AFTER** the block of code is executed

CODE SAMPLE

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

Example: Suppose we want to print a statement 5 times

Inline variable declaration. Variable i is only visible within the loop

```
for (let i = 1; i < 5; i++){
    console.log("Thanks!:" + i);
}
```

Explanation

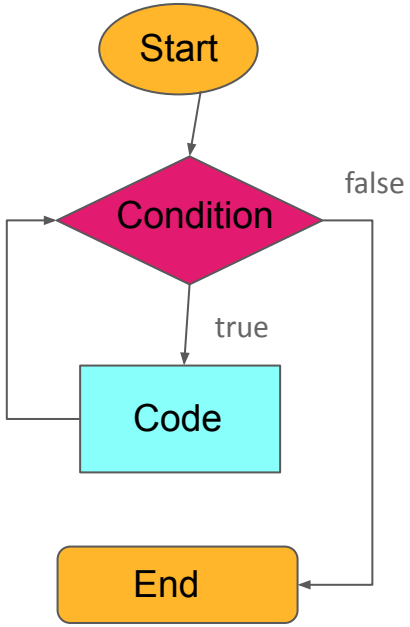
Step 1: a variable is initialized with value = 1. This is done once.
Step 2: Condition is checked, if it is true, statement will be displayed
Step 3: i = variable is incremented to display it again
Step 4: Again condition will be checked and step2 and step3 would be followed until the condition returns false

While Loops

While Loops

While Loop

While loop runs a block of code as long as any specific condition is true



Example: Display the word "Practice: x" using ***while loop***

INITIALIZATION

```
let i = 0;
```

OPENING CURLY BRACE *(START OF LOOP)*

KEYWORD

CONDITION

```
while (i < 10){
    console.log("Practice:" + i);
    i++;
}
```

CODE TO EXECUTE DURING LOOP *(INSIDE THE LOOP)*

INCREMENT

CLOSING CURLY BRACE *(END OF LOOP)*

https://repl.it/@malcolmyam/wk06-basic-js

Output

Practice: 0
Practice: 1
Practice: 2
Practice: 3
Practice: 4
Practice: 5
Practice: 6
Practice: 7
Practice: 8
Practice: 9

Loop stops because i < 10

Q. What if we take out the increment?

OR *i = 10*

WHILE LOOPS

Basically a ***while loop*** is the same as a ***for loop***; however, sometimes, you may not know how many times it will iterate (repeat).

The ***while loop*** **repeats** as long as the condition is true.

One disadvantage of using while loop is that it's easier to end up with an **infinite loop**, which may cause your computer to hang because the loop will run forever until the browser is out of memory. *#dontbelievetryit*

FOR LOOP

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

V

S

WHILE LOOP

```
let i = 0;
while (i < 10){
    console.log("Practice:" + i);
    i++;
}
```

Both are equivalent and produce the same results

Example: Display the word "Practice: x" using ***do.. while loop***

INITIALIZATION

```
let i = 0;
```

OPENING CURLY BRACE *(START OF LOOP)*

KEYWORD

CONDITION

CODE TO EXECUTE DURING LOOP *(INSIDE THE LOOP aka loop body)*

```
do{
    console.log("Practice:" + i);
    i++;
}while (i < 10)
```

INCREMENT

CLOSING CURLY BRACE *(END OF LOOP)*

This form of syntax should only be used when you want the body of the loop to execute at least once regardless of the condition being truthy.

Usually, the other form is preferred: while(…) {…}.

Output

Practice: 0
Practice: 1
Practice: 2
Practice: 3
Practice: 4
Practice: 5
Practice: 6
Practice: 7
Practice: 8
Practice: 9

Loop stops because i < 10

Q. What if we take out the increment?

OR *i = 10*

https://replit.com/@immalcolm/do-while

Loop Control

break

Loop Control

Those *loop control* statements can be used in both **WHILE** and **FOR** loops:

break - exit out of the loop completely

continue - skip the rest of the loop, and start a new iteration

Those are usually use together with a **IF** statement.

Break

This will just break out of the loop. We can rewrite the *while* loop that keep asking the user to key in a positive number till they enter one like this:

V1: Non breaking

```
let x = parseInt(prompt("Enter a number: "));
while (x < 0) {
    x = prompt("Enter a number");
console.log("You have entered a positive number");
}
```

Using Break

```
let x = parseInt(prompt("Enter a number: "));
while (true) {
    x = prompt("Enter a number");
    if (x > 0) {
        break;
    }
}
console.log("You have entered a positive number");
```

The `break` directive is activated if the x > 0. It stops the loop immediately, passing control to the first line after the loop. Namely, `alert`.

Continue

The ***continue*** statement will skip the rest of the loop and start a new iteration. (**For a WHILE loop, the sentinel condition will be checked again).**

The following code asks the user to key in two positive numbers:

The loop repeats if the first number is not positive

```
let x = 0;
let y = 0;
while (x <= 0 || y <= 0) {
    x = prompt("Enter the first number: ");
    if (x < 0) {
        continue;
    }
    y = prompt("Enter the second number: ");
}
console.log("You have entered two positive numbers");
```

What we covered

3 Types of Loops

`while` – The condition is checked before each iteration.
`do..while` – The condition is checked after each iteration.
`for (;;)` – The condition is checked before each iteration, additional settings available.

To create a loop that runs indefinitely, the while(true) construct is commonly employed. This type of loop, like any other, can be terminated using the break directive.
When there's nothing to execute in the current iteration and the intention is to move directly to the next one, the continue directive can be used.

Arrays

Simple ones..

Arrays Importance

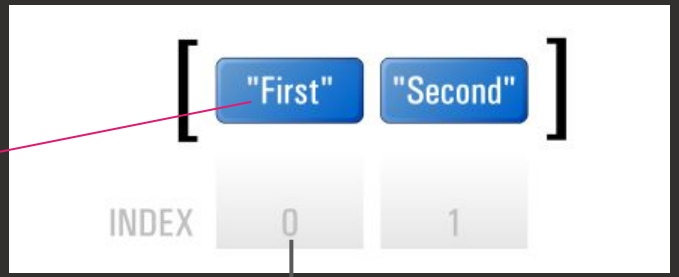
Arrays help us to avoid having many variables of different names

Instead of variables "name1", "name2", "name3", just use an array called names[].

Arrays helps us to….

Organise many variables into a container
Make lists/groups/collections (Names, Numbers, etc)

Array Terminology



element

index or *subscript or key*

console.log(list**[0]**)

accessor

ARRAY INDEXING

```
                 0          1          2
colors= ['pink','yellow','green'];
```

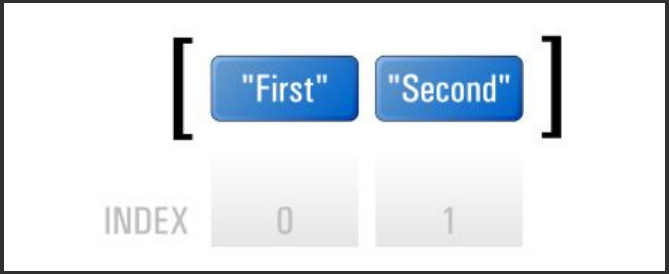
Elements

Think of the index as an address of this particular element in the memory.
'Yellow' is store at **colors[1]** that means on the **second location** in the array

Defining Arrays

Declare an array called list

```
let list=[];
list[0] = "First";
list[1] = "Second";
```



We can insert a **variable** into an array by *specifying which **index** it goes into*

Defining Arrays (Shortcut)

```
let list = ["First", "Second", "Third", "Fourth"];
```

0	1	2	3

```
console.log(list[0]);//"First"
```

Javascript Arrays are Special.

Unlike other languages, arrays in JavaScript can store values of **different data types**

```
let rojak = [1, 3.14, true, "Hello World!", function() {
   console.log("foobar!");
}];
```

Rojak is an array that stores an integer, a float, a string and a closure.

You can store number value, character values, etc all in one variable. **What matters is the order** of them.

Creating Arrays

```
//[Creating literal]
let arr = [23, "foodBank", "is great", 388];
```

Array literal

```
//[Array Constructor]
let arr = new Array(23, "foodBank", "is Great", 388);
```

Array Constructor

```
//[Creating instance with new]

let arr = new Array();
arr[0] = 23;
arr[1] = "foodBank";
arr[2] = "is Great!";
arr[3] = 388;
```

Creating instance with “new”

COMMON METHODS & PROPERTIES

METHOD	DESCRIPTION
pop()	Removes the last element in an array
push()	Adds a new element to array
shift()	Removes the first element in array
unshift()	Adds a new element to array
splice(a,b,<elements>)	a defines position where new elements are added b defines how many elements to be removed <elements> define new elements to be added
sort()	Sorts array alphabetically
indexOf()	Search the array for an element and returns its position

PROPERTY	DESCRIPTION
length	Number of keys

http://www.w3schools.com/jsref/jsref_obj_array.asp

Working with Arrays

Maximum length of an array
Use ***length*** to determine how many elements there are in an array

```
let list = [3, 10, 11];
console.log(list.length);
```

http://www.w3schools.com/jsref/jsref_obj_array.asp

Working with Arrays

Add to the **end** of an array

```
list.push(3);
```

http://www.w3schools.com/jsref/jsref_obj_array.asp

Working with Arrays

Remove and get the last element in array

```
list.pop();
```

http://www.w3schools.com/jsref/jsref_obj_array.asp

Working with Arrays

Return a selected portion of an array

```
let smaller_list = list.slice(2,5);
```

http://www.w3schools.com/jsref/jsref_obj_array.asp

Working with Arrays

You can **skip indexes** in an array

```
let list = [];
list[0] = "a";
list[26] = "z";
```

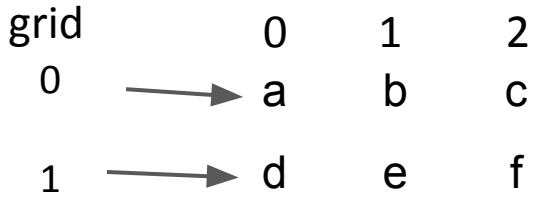
All the values you skip are filled with ***undefined***

2D Arrays

Array in Array

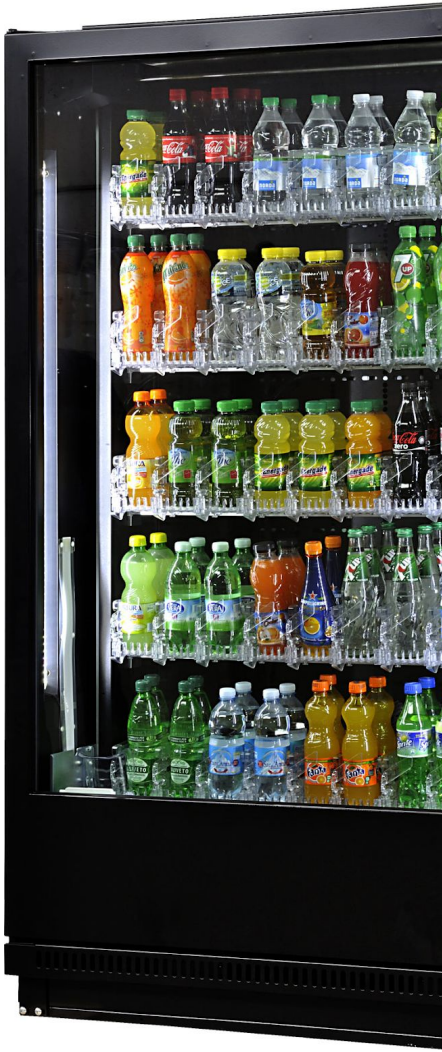
```
let grid = [
  ["a", "b", "c"],
  ["d", "e", "f"],
]
// hint: start from 0!
grid[0][0] -- "a"
grid[1][2] -- "f"

// set index 0 of the array in index 1
// of grid value to g
grid[1][0] = "g";
```



Example - Vending Machine

	0	1	2
0	Oreo	Potato Chips	Twisties
1	Kit Kat	Hello Panda (Chocolate)	Hello Panda (Strawberry)
2	OK Pocky	Snickers	Cup Noodle



Associative Arrays

Associative Arrays

Normal arrays use ***integers*** as the index or subscript.

Associative arrays use ***strings*** as the index
In an associative array, the **index or subscript** is often known as the ***key*.**

Further Reading: *Simply JavaScript,* Page 34 - 35

How to use Associative Array

Simply use a string as the index instead of a number:

```
let country_codes = [];
country_codes["sg"] = "Singapore";
country_codes["uk"] = "United Kingdoms";
```

Functions

Functions

If I ask you to build a system that does the entry each time a student logs in, would it be helpful writing a code for adding this entry for each student individually and also everytime he/she logins in?

Functions

Writing code for the same task again and again can cause readability issue, prone to error, hard to maintain and debug.

We got **functions** as our savior 🥰

Functions are a block of code to perform a specific task.

So now you don't have to write the code for the same task over and over again but create a function and call it wherever the task is needed to be performed.

KEYWORD to depict function

FUNCTION name

```
function sayHello() {
  console.log('Hello');
}
```

CODE BLOCK (IN CURLY BRACES)

```
sayHello();
```

FUNCTION NAME

To use the function we declared, we need to **call it** at the place it is needed to run.

CALLING A FUNCTION THAT NEEDS INFORMATION

PARAMETER PARAMETER

```
function getArea(width, height) {
  return width * height;
}
```

Having a return statement is optional however it's a necessary good. It depends on your code utility.

THE PARAMETERS ARE USED LIKE VARIABLES WITHIN THE FUNCTION

```
getArea(3, 5);
```

ARGUMENTS

The () Operator invokes the function. Accessing a function without () will return the function object instead of the function result

OPTIONAL DEFAULTS

You can mix normal arguments (aka. Positional arguments) with optional arguments when you call the function, but the former must **come first.**

PARAMETER PARAMETER OPTIONAL PARAMETER

```
function getArea(width, height, modifier = 10)
{
  return width * height * modifier;
}
```

```
getArea(3, 5, 100);
getArea(3, 5);
```

The () Operator invokes the function. Accessing a function without () will return the function object instead of the function result

Immediately Invoked Function Expression IIFE

Pronounced as ‘iffy’

IIFE

Before diving into IIFE, let us understand what function expression is.

Function expression is a function defined as an expression.

What does that mean?

```
let result = function(a, b){
    return a + b;
}
```

https://developer.mozilla.org/en-US/docs/Glossary/IIFE

IIFE

This is same as a function declaration but the difference lies in the fact that function expression are declared with the name of function and they won't throw an error.

It is allowed using a function with no name(anonymous function) in function expression

```
let result = function(a, b){
    return a + b;
}
```

IIFE

Immediately invoked function(IIFE) is a design pattern that allows a function to be declared as an **expression** and executed immediately after creation.

```
//IIFE
let finalResult = (function(a, b){
    return a + b;
})(80, 10);

console.log(finalResult);
```

Regular Function vs IIFE

Whenever a regular function or variable is declared, JS engine adds it to the global object that results in their access before they are defined or in simpler words, regular function and variables are hoisted.

This results in the inefficient use of memory because memory won't be relocated until the global object releases it.

To solve this issue we have IIFE, that are called immediately after creation. They are not hoisted.

Variable on which we assign this function will not be invoked afterwards

```
//regular function
function finalResult(a, b){
    return a + b;
}

console.log(finalResult(80, 10));
```

```
//IIFE
let finalResult = (function(a, b){
    return a + b;
})(80, 10);

console.log(finalResult);
```

When to use IIFE?

A real world application of **IIFE** is when you want list of functions to run as soon as the page loads, using them would be clearer way of writing them. Something to use once, but won't use again.
You would want to avoid name collisions by using IIFE

```
//IIFE syntax
(function() {
    /* */
  })()
```

```
//Named IIFE syntax
(function doSomething() {
    /* */
  })()
```

Arrow Function

Arrow functions

Arrow functions allow us to write shorter and cleaner syntax as compared to regular functions.
They are more likely to be used when you want to create an anonymous function.

```
//[function expression]
let x = function(x, y){
    return x * y;
}
```

```
//[arrow function]
let x = (x, y) => x * y;
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Function Expression vs Arrow Function

```
//[function expression]
let x = function(x, y){
    return x * y;
}
```

```
//[arrow function]
let x = (x, y) => x * y;
```

Noticed? Our variable remains the same
The `**function**` keyword is omitted with function symbol and the code remains the same for whatever task you are creating it for.

The arrow function is clearer and shorter

To call the arrow function, use `x();`

Ways to declare Arrow Function

```
//[Arrow function with no arguments]
let hi = () => console.log('hi'); //code;
```

Function with no arguments

When function doesn’t accept arguments, then you can use empty parentheses.

```
//[Arrow function with arguments]
let showText = (text) => console.log(text);
```

Function with arguments

Here you can use arguments inside parentheses. Works same as regular function but in shorter way.

Ways to declare Arrow Function

```
//[Arrow function as expression]
let num = 8;
let message = (num < 10 ) ?
    () => console.log('Correct!') :
    () => console.log('Try Again!');

message();
```

Function as expression

You can easily use arrow function as expression. We are using with ternary operators.

```
//[Multiline Arrow function]
let result = (a, b) => {
    let sum = a + b;
    return sum;
}
```

Multiline function

We simply use curly braces to wrap the statements of a function

Drawbacks of arrow functions

1. When you want to refer to the function at some point in the code (maybe in the form of recursion or event handler that you want to use somewhere else as well.)

2. You won’t be able to easily debug the code because it has no name.

Variable Scope

Think of Scope as a "Context" for Variables

- We need "scope" or else we'll run out of variable names
- We may also accidentally reassign to important variables

What is a scope?

A set of curly braces form a scope:

```
let x = parseInt(prompt("Enter a number: "));
if (x > 0) {
  let y = parseInt(prompt("Enter a second number: "));
}
```

This is a scope

Variable in a Scope

A variable is only available in a scope that it is defined in.

If a variable is not inside a set of curly braces, then it is in the **global scope.**

Variable *x* is in the global scope

```
let x = parseInt(prompt("Enter a number: "));
if (x > 0) {
  let y = parseInt(prompt("Enter a second number: "));
}
```

Variable Access (part 1)

A line of code **cannot** access a variable **not in** its scope.

```
let x = parseInt(prompt("Enter a number: "));
if (x > 0) {
  let y = parseInt(prompt("Enter a second number: "));
}
console.log(y);
```

Error! Out of scope!

Variable Access (part 2)

A line of code can access a variable that **exist** in the scope it is enclosed in

```
let x = parseInt(prompt("Enter a number: "));
if (x > 0) {
  let y = parseInt(prompt("Enter a second number: "));
  console.log(x * y);
}
```

Variable Access (part 3)

A line of code can access a variable that **exist** in the scope it is enclosed in

```
let x = parseInt(prompt("Enter a number: "));
if (x > 0)
{
  let y = parseInt(prompt("Enter a second number: "));
  if (y > 0)
  {
    let z = parseInt(prompt("Enter a third number: "));
    console.log(x * y * z);
  }
}
```

Define the *y* variable

The green scope is **nested** (or <u>enclosed in</u>) the yellow scope, so it can access variable y.

Variable Access (part 4)

A line of code can access a variable that **exist** in the scope it is enclosed in

```
let x = parseInt(prompt("Enter a number: "));
if (x > 0)
{
  let y = parseInt(prompt("Enter a second number: "));
  if (y > 0)
  {
    let z = parseInt(prompt("Enter a third number: "));
    console.log(x * y * z);
  }
}
```

The yellow scope enclosed in the global scope, so it can access the global variable ***x***

Overwriting Variable Names

When a line of code looks for a variable, it will check its own scope, then the scope above it and so on.

```
let myNumber = 3;
{
  let myNumber = 4;
  console.log(myNumber);
}
```

Because there is a definition of *myNumber* in this scope, the *console.log(myNumber*) will use that *local* variable instead of the global one.

Important!

As a programmer, you must always know the purpose and origin of every variable you define in a program.