Lesson 1: Introduction to Javascript

If we imagine a web page as a house, the HTML makes up the materials (wood, brick, and concrete), CSS is how we put all of those materials together, and Javascript is the electricity, plumbing and gas. It's what makes the web page "run". Javascript was originally designed to be used purely on the front end as a way for web developers to add functionality to their web pages, and in its early days it did just that. Recently, the introduction of the "V8 engine" by Google has improved the speed and functionality of JS. That led to the development and release of exciting new front-end Javascript frameworks and eventually Node.js, a way to run Javascript on a server (back-end). This new development has led to a resurgence of Javascript. Javascript is one of the world's most widely used programming languages. We now find Javascript used in front-end, back-end, mobile development, IoT, and really anywhere a traditional programming language would be used. Keep in mind, **Javascript != Java**. Although they share similar names (this was, unfortunately, considered a feature by Javascript's early pioneers) that is where the similarities end.

You can use the free online IDE repl.it to run your Javascript code online.No need to sign up or create account unless you want to save your work for a later reference.

Variables

At the heart of Javascript are variables. A variable is a way to store the value of something to use later. (A note for those with previous programming knowledge: Javascript is a loosely typed language, a variable can be set (and reset) to any type, we do not need to declare its type when initiating the variable.)

The anatomy of a variable is first the keyword, a space, the name we are giving the variable, an equal sign, the value we are assigning the variable and then the semi-colon.

There are three ways to declare a variable.

```
    var firstName = 'Anil';
    let lastName = 'Kumar';
    const favoriteFood = 'Ice Cream';
```

var
`var` is the ES5 way of declaring a variable. This is a generic variable keyword. I suggest not to use this notation.

let
`let` is a new ES6 variable keyword, this will assign a variable much like `var`, but with a little bit different behavior. Most notably, it differs by creating "block level scope".

const
`const` is also new in ES6. A `const` is a variable that will not be able to be changed. This is short for "constant".

console.log
Another concept we will talk about right away is

```
console.log();
```

This basic method will allow us to print to the console anything we put between the parentheses.

Comments

Have you ever written a script or a program in the past only to look at it six months later with no idea what's going on in the code? You probably forgot to do what all programmers tend to forget to do: write comments!

When writing code you may have some complex logic that is confusing, this is a perfect opportunity to include some comments in the code that will explain what is going on. Not only will this help you remember it later, but if you or someone else views your code, they will also be able to understand the code (hopefully)!

Another great thing about comments is the ability for comments to remove bits of code from execution when you are debugging your scripts. This lesson will teach you how to create two types of comments in JavaScript: single line comments and multi-line comments.

Creating single line comments

To create a single line comment in JavaScript, you place two slashes "//" in front of the code or text you wish to have the JavaScript interpreter ignore. When you place these two slashes, all text to the right of them will be ignored, until the next line.

These types of comments are great for commenting out single lines of code and writing small notes.

```
<script type="text/javascript">
<!--
// This is a single line JavaScript comment

console.log("I have comments in my JavaScript code!");
//console.log("You can't see this!");
//-->
</script>
```

Each line of code that is colored red is commented out and will not be interpreted by the JavaScript engine.

Creating multi-line comments

Although a single line comment is quite useful, it can sometimes be burdensome to use when disabling long segments of code or inserting long-winded comments. For this large comments you can use JavaScript's multi-line comment that begins with /* and ends with */.

```
<script type="text/javascript">
<!--
console.log("I have multi-line comments!");
/*console.log("You can't see this!");
console.log("You can't see this!");
console.log("You can't see this!");
console.log("You can't see this!");
console.log("You can't see this!");
console.log("You can't see this!");
console.log("You can't see this!");*/
```

```
//-->
</script>
```

Quite often text editors have the ability to comment out many lines of code with a simple key stroke or option in the menu. If you are using a specialized text editor for programming, be sure that you check and see if it has an option to easily comment out many lines of code!

Strings, Numbers, and Booleans

These are the most basic data types in Javascript.

Strings

Strings are blocks of text, they will always be defined with quotation marks around them, either single or double. Any text with quotes around it is a string.

```
    const programName = '10000 Coders Bootcamp';
```

Numbers

Numbers are just that, numbers. Numbers do NOT have quotes around them. They can be negative as well. Javascript does have a limitation on the size of a number (+/-9007199254740991), but only very rarely will that limitation come up.

```
    const answer = 42;
    const negative = -13;
```

Boolean

Booleans come from low level computer science. It is a concept that powers binary code and the very core of computers. You may have seen binary code in the past (0001 0110...), this is boolean logic. It essentially means you have two choices, on or off, 0 or 1, true of false. In

Javascript we use Booleans to mean true or false. This may seem simple at first but can get complicated later on.

```
const isBootcampHelpful = true;
```

Introduction to Arrays

In the previous section we discussed the 3 basic data types (strings, numbers, and booleans), and how to assign those data types to variables. We discussed how a variable can only point to a single string, number, or boolean. In many cases though we want to be able to point to a collection of data types. For example what if we wanted to keep track of every student's name in this class using a single variable, studentsNames. We can do that using Arrays. We can think of arrays as storage containers for collections of data. Building an array is simple, declare a variable and set it to []. We can then add however many strings, numbers, or booleans as we want to the container (comma separated), and access those items whenever we want.

```
const studentsNames = ['Dan', 'Maria', 'Sara', 'Raj'];
```

.length

Just like the String data type has a built in .length method, so does the array. In fact the array has a lot of useful built in methods (we will be discussing those in later lessons). Just like the string .length counts the characters, array .length will return the number of items in an array:

```
const studentsNames = ['Dan', 'Maria', 'Sara', 'Raj'];

console.log(studentNames.length);  // 4
```

Accessing Items in an Array

We can access an item at anytime in an array, we just need to call the item by its position in the array. Items are given a numerical position (index) according to where it is in the array, in order. An array's numerical order ALWAYS starts at 0, so the first item is in the 0 index, the second in the 1 index, the third in the 2, and so on (this can be tricky at first, but just remember arrays always start at 0).

```
    const studentsNames = ['Dan', 'Maria', 'Sara', 'Raj'];
                            0        1        2       3
```

In order to access the item, we will type the name or the array variable, followed by brackets containing the numerical assignment.

```
    const studentsNames = ['Dan', 'Maria', 'Sara', 'Raj'];

    console.log(studentNames[1]);  // 'Maria'
```

To dynamically access the last item in the array, we will use the `.length` method. In our `studentsNames` array, the length is 4. We know the first item is always going to be 0, and every item after is shifted over one number. So in our example the last item has an index of 3. Using our length property we will show how it is done when we don't know the number of items in an array:

```
    const studentsNames = ['Dan', 'Maria', 'Sara', ...
,'Raj'];

    console.log(studentNames[studentNames.length - 1]);  //
'Raj'
```

Assignment
We can assign and reassign any index in the array using the bracket/index and an =.

```
    const studentsNames = ['Dan', 'Maria', 'Sara', 'Raj'];

    studentNames[0] = 'Ryan';

    console.log(studentNames);  // ['Ryan', 'Maria', 'Sara',
'Raj']
```

.push & .pop
Two more very useful built in array methods are `.push` and `.pop`. These methods refer to the adding and removing of items from the array after it's initial declaration.
`.push` adds an item to the end of the array, incrementing it's length by 1. (`.push` returns the new length)

```
    const studentsNames = ['Dan', 'Maria', 'Sara', 'Raj'];

    studentNames.push('Ryan');

    console.log(studentNames);  // ['Dan', 'Maria', 'Sara',
 'Raj', 'Ryan']
```

`.pop` removes the last item in the array, decrementing the length by 1. (`.pop` returns the "popped" item)

```
    const studentsNames = ['Dan', 'Maria', 'Sara', 'Raj'];

    studentNames.pop();

    console.log(studentNames);  // ['Dan', 'Maria', 'Sara']
```

.unshift & .shift

`.unshift` and `.shift` are exactly like `.push` and `.pop`, except they operate on the first item in the array. `.unshift(item)` will put a new item in the first position of the array, and `.shift()` will remove the first item in the array.

```
    const studentsNames = ['Dan', 'Maria', 'Sara', 'Raj'];

    studentNames.unshift('Ryan');

    console.log(studentNames);  // ['Ryan', 'Dan', 'Maria',
 'Sara', 'Raj']

    studentNames.shift();

    console.log(studentNames);  // ['Dan', 'Maria', 'Sara',
 'Raj']
```

Notes on Arrays

Because Javascript is not a strongly typed language, arrays do not need to be typed either. Arrays in Javascript can contains multiple different data types in the same array.

Introduction to Objects

In the last lesson we introduced Arrays. Arrays are containers that hold collections of data. In this lesson we will introduce another data container, the Object. Objects and arrays are similar in some ways and very different in others. Where arrays hold multiple items related to each other, Objects will hold a lot of information about one thing. Objects are instantiated by using braces (`{}`).

```
const newObj = {};
```

Key:Value pairs

Unlike arrays that have index valued items, objects use a concept called key:value pairs. The key is the identifier and the value is the value we want to save to that key. The syntax is "key: value". Objects can hold many key:value pairs, they must be separated by a comma (no semi-colons inside of an object!). Keys are unique in an object, there can be only one key of that name. Although, multiple keys can have the same value. Values can be any Javascript type, string, number, boolean, array, function or even another object. In this demo we will create a user object.

```
const user = {
    username: 'anil.kumar',
    password: '10000Coders',
    lovesJavascript: true,
    favoriteNumber: 42,
};
```

Accessing Values

Once we have key:value pairs we can access those values by calling the object name and the key. There are two different ways to do this, dot notation and bracket notation. With dot notation we can just call the object name, a dot, and the key name. Just as we call the `.length` property on an array (hint: the length property is a key:value pair):

```
user.lovesJavascript; // true
user.username;        // anil.kumar
```

Bracket notation is just like calling an item on an array, although with brackets we MUST use a string or number, or variable pointing to a string or number. Each key can be called by wrapping it with quotes:

```
const passString = 'password';
user['lovesJavascript']; // true
```

```
user['username'];        // anil.kumar
user[passString];        // 10000Coders
```

In the wild you will see brackets almost always being used with variables.

Assigning Values

Assigning values works just like accessing them. We can assign them, when we create the object, with dot notation, or with bracket notation:

```
const newUser = {
    isNew: true,
}

const loveJSString = 'lovesJavascript';

newUser.username = 'new.username';
newUser['password'] = '12345';
newUser[loveJSString] = true;
```

Removing Properties

If we want to remove a property we can do it using the `delete` keyword:

```
const newObject = {
    removeThisProperty: true,
};

delete newObject.removeThisProperty;
```

It is rare that we will see the use of the `delete` keyword, many find best practice to be setting the value of a keyword to `undefined`. It will be up to you when the time comes.

Undefined and Null

There are a couple of Javascript objects that don't really fit into any type. Those are the values `undefined` and `null`. You will get `undefined` when you are looking for something that does not exist like a variable that does not have a value yet. `undefined` simply means what you are asking for does not exist.

```
console.log(unkownVar); // undefined
```

`null` is an object that we, the developers, set when we want to tell other developers that the item they are looking for exists, but there is no value associated with it. While `undefined` is set by the Javascript language, `null` is set by a developer. If you ever receive `null`, know that another developer set that value to `null`

```
let phoneNumber = '123-456-7890';
phoneNumber = null;

phoneNumer; // null
```

One last thing to note, neither `undefined` nor `null` are strings, they are written just as they are with no quotes around them, like a boolean.

Arithmetic Operators

Math operators work in javascript just as they would on your calculator.

+ - * / =

```
1 + 1 = 2
2 * 2 = 4
2 - 2 = 0
2 / 2 = 1
```

%

Something you may not have seen before is the Modulo (`%`), this math operator will divide the two numbers and return the remainder.

```
21 % 5 = 1;
21 % 6 = 3;
21 % 7 = 0;
```

Comparison Operators

Comparison Operators evaluate two items and return either `true` or `false`. These operators are: `<` , `<=`, `>`, `>=`, `===`, `!==`. These operators work just as they would in a math class, greater than, less than, etc. We use these operators to evaluate two expressions.

As the computer runs the code the operator will return either a true (if the statement is true) or a false.

```
1 > 2;      // alse
2 < 3;      // true
10 >= 10;   // true
100 <= 1;   // false
```

The "triple equals" (===) must not be confused with a single equal sign (which indicates assigning a value to a variable). The triple equal will compare everything about the two items, including type, and return if they are exactly equal or not: (Something to note: there is a "double equals" (==) which will compare two items, but it will NOT take into account their types (1 == '1' // true). Due to this, it is considered bad practice to use the double equal. We would like to see you always using the triple, and you will always see us using it.)

```
1 === 1;            // true
1 === '1';          // false
'cat' === 'cat';    // true
'cat' === 'Cat';    // false
```

The last comparison operator we would like to introduce you to has two parts to it. First is the "NOT" (!) when you see this it will mean that we are asking the opposite of the expression (we will revisit the NOT operator later in this lesson).
With that in mind, we can introduce the "not equals" (!==). This will return true if the items are NOT equal to each other, in any way. This, like the triple equal, takes type into account.

```
1 !== 1;            // false
1 !== '1';          // true
'cat' !== 'cat';    // false
'cat' !== 'Cat';    // true
```

Logical Operators

We can also combine two equality expressions and ask if either of the are true, both of them are true, or neither of them are true. To do this we will use Logical Operators.

&&

The first logical operator we will look at is the "AND" operator. It is written with two ampersands (`&&`). This will evaluate both expressions and will return true if BOTH expressions are true. If one (or both) of them is false this operator will return false:

```
    if (100 > 10 && 10 === 10) {
        console.log('Both statements are true, so this code
will be run');
    }

    if (10 === 9 && 10 > 9) {
        console.log('One of the statements is false, so the &&
will return false, this code will not be run');
    }
```

||

The next is the "OR" operator. It is written with two vertical bars (`||`). It will determine if one of the expressions is `true`. It will return `true` if one (or both) of the expressions is `true`. It will return `false` if BOTH expressions are `false`:

```
    if (100 > 10 || 10 === 10) {
        console.log('Both statements are true, so this code
will be run');
    }

    if (10 === 9 || 10 > 9) {
        console.log('One of the statements is true so the ||
will return true, this code will be run');
    }

    if (10 === 9 || 1 > 9) {
        console.log('Both of the statements are false, so the
|| will return false. This code will not be run.');
    }
```

!

The last logical operator is the "NOT" operator. It is written as a single exclamation mark (`!`). We saw this operator earlier when determining equality (`!==`). As before, the NOT operator will return the opposite boolean value of what is passed to it:

```
    if (!false) {
        console.log('The ! will return true, because it is the
opposite of false. This code will be run');
    }

    if (!(1 === 1)) {
        console.log('1 does equal 1, so that expression
returns true. The ! operator will then return the opposite of
that. This code will NOT run.');
    }
```

Notes About Logical Operators
A couple things to note about logical operators.

- The expressions are evaluated in order, and the computer will skip any redundant expressions. In an `&&` statement, if the first expression is `false`, the second expression will not be evaluated because BOTH expressions need to be `true`. Same for the `||` statement. If the first expression is `true`, the second will not be evaluated because there only needs to be one `true` statement to fulfill the requirements of the operator.
- Use parentheses. As we saw in the second `!` operator example, we used parentheses to evaluate what was inside of the parentheses FIRST, then applied the `!` operator. We can wrap ANY expression in parentheses and it will be evaluated before evaluating the expression as a whole.

The ++ and – operators

The ++ operator is Javascript shorthand for "Set the value of the variable to it's current value plus one". Similary -- decreses the current value by 1.

```
let myNum = 15;
console.log(myNum); //15

myNum++;
console.log(myNum); //16

myNum--;
console.log(myNum); //14
```

Control Flow

We can use `if` to check and see if an expression is `true`, if it is, run some code. If it is not, skip the code and keep running the program.

```
    if (1 + 1 === 2) {
        console.log('The expression is true!');
    }
```

To add on to `if`, we can also use the `else if` and `else` statements. These statements must be used with `if` and must come after it. These statements will be evaluated if the initial `if` returns `false`. We can think of the `else if` as another `if`statement that has been chained (we can have as many else if statements we want). Only one `if` or `else if` statement code block will be run. If at any time a statement returns `true`, that code will be run and the rest will be skipped:

```
    if (false) {
        console.log('This will be skipped!');
    } else if (true) {
        console.log('This code will be run');
    } else if (true) {
        console.log('This code will NOT be run');
    }
```

The `else` statement will always come at the end of an if-else `if` chain, and will act as a default. If none of the expressions returned `true`, the `else` code block will be run no matter what. If any of the previous `if` or `else if` expressions are `true`, the `else` statement code block will not be run.

```
    if (false) {
        console.log('This will be skipped!');
    } else if (false) {
        console.log('This code will NOT be run');
    } else {
        console.log('This code will be run');
    }
```

You can use multiple if...else…if statements, to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable. You can use a switch statement which handles exactly this situation, and it does so more efficiently than repeated if...else if statements.

The objective of a switch statement is to give an expression to evaluate and several different statements to execute based on the value of the expression. The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

```
switch (expression) {
   case condition 1: statement(s)
   break;
   
   case condition 2: statement(s)
   break;
   ...
   
   case condition n: statement(s)
   break;
   
   default: statement(s)
}
```

The break statements indicate the end of a particular case. If they were omitted, the interpreter would continue executing each statement in each of the following cases. We will explain break statement in Loop Control chapter.

Example

Try the following example to implement switch-case statement.

```
<html>
   <body>
      <script type = "text/javascript">
         <!--
            var grade = 'A';
            document.write("Entering switch block<br />");
            switch (grade) {
               case 'A': document.write("Good job<br />");
               break;
```

```
                 case 'B': document.write("Pretty good<br />");
                 break;

                 case 'C': document.write("Passed<br />");
                 break;

                 case 'D': document.write("Not so good<br />");
                 break;

                 case 'F': document.write("Failed<br />");
                 break;

                 default:  document.write("Unknown grade<br />")
              }
              document.write("Exiting switch block");
           //-->
        </script>
        <p>Set the variable to different value and then
try...</p>
     </body>
</html>
```

Output

```
Entering switch block
Good job
Exiting switch block
Set the variable to different value and then try...
```

Break statements play a major role in switch-case statements. Try the following code that uses switch-case statement without any break statement.

Live Demo

```
<html>
   <body>
      <script type = "text/javascript">
         <!--
            var grade = 'A';
            document.write("Entering switch block<br />");
```

```
              switch (grade) {
                 case 'A': document.write("Good job<br />");
                 case 'B': document.write("Pretty good<br />");
                 case 'C': document.write("Passed<br />");
                 case 'D': document.write("Not so good<br />");
                 case 'F': document.write("Failed<br />");
                 default: document.write("Unknown grade<br />")
              }
              document.write("Exiting switch block");
           //-->
        </script>
        <p>Set the variable to different value and then
try...</p>
     </body>
</html>
```

Output

```
Entering switch block
Good job
Pretty good
Passed
Not so good
Failed
Unknown grade
Exiting switch block
Set the variable to different value and then try...
```

for Loops

Most software runs on loops, evaluating expressions over and over again until it either returns what we are looking for, or stops after a certain time. Javascript has two looping expressions built in to it and today we will look at the first one, the "for" loop.
for loops have a unique syntax, similar to the if statement, but slightly more complex.

First we have the for keyword, followed by parentheses and then open and close braces. Within the parentheses we will need three things. First, we must declare a variable, this is what the loop will be looping over. Then we will have a conditional expression, the loop will continue happening until this statement is false. Third, we will increment our variable. All three of these statements are separated by a semi-colon.

```
for (let i = 0      ; i < 10                  ; i++           ) {
{ //| declare a var | conditional expression | increment var
console.log(i);
}
```

In this example we see that we initially set our counter variable to 0, the loop will run and each time it gets to the end, it will increase the counter by one. The for loop will then evaluate the conditional expression. If it is true, it will run again, if it is false it will stop running.

Introduction to Functions

Now that we have variables set we need functions to compute them, change them, do something with them. There are three ways we can build a function.

```
    function myFunc() {}
    const anotherFunc = function () {};
    const yetAnother = () => {};
```

We will be using the first way in this lesson, and talk about the other ways in future lessons.

Anatomy of a Function

```
    function myFunc() {}
```

A function will start with the function keyword, this tells whatever is running your program that what follows is a function and to treat it as such. After that comes the name of the function, we like to give functions names that describe what they do. Then comes an open and a close parentheses. And finally, open and close brackets. In between these brackets is where all of our function code will go.

```
    function logsHello() {
        console.log('hello');
    }
```

```
logsHello();
```

In this example we declare a function logsHello and we set it up to console.log 'hello'. We can then see in order to run this function, we need to write the name and parentheses. This is the syntax to run a function. A function always needs parentheses to run.

Arguments
Now that we can run a basic function, we are going to start passing it arguments.

```
function logsHello(name) {
    console.log('Hello from ' + name);
}

logsHello('10000 Coders');
```

If we add a variable to the parentheses when we declare the function we can use this variable within our function. We initiate the value of this variable by passing it into the function when we call it. So in this case name = '10000 Coders'. We can pass other variables into this as well:

```
function logsHello(name) {
    console.log( `Hello from ${name}`);
}

const myName = '10000 Coders';
logsHello(myName);
```

We can add multiple arguments by placing a comma in between them:

```
function addsTwoNumbers(a, b) {
  const sum = a + b;
  return sum;
}

addsTwoNumbers(1, 5); // 6
```

Return statement and Scope

In the last example we introduced the return statement. We will not console.log everything that comes out of a function. Most likely we will want to return something. In this case it is the sum of the two numbers. Think of the return statement as the only way for data to escape a function. Nothing other than what is returned can be accessed outside of the function. Also note, that when a function hits a return statement, the function immediately stops what it is doing and returns.

```
function dividesTwoNumbers(a, b) {
  const product = a / b;
  return product;
}

dividesTwoNumbers(6, 3); // 2
console.log(product); // undefined
```

If we tried to console.log something that we declared inside of the function it would return undefined because we do not have access to it outside of the function. This is called scope. The only way to access something inside of the function is to return it. We can also set variables to equal what a function returns.

```
function subtractsTwoNumbers(a, b) {
  const difference = a - b;
  return difference;
}

const differenceValue = subtractsTwoNumbers(10, 9);
console.log(differenceValue); // 1
console.log(difference); // undefined
```

We can see that difference is set inside of the function. The variable inside the function only belongs inside the function.

Lesson 2: Javascript and HTML DOM

The DOM (as you will repeatedly hear it called) refers to 'Document Object Model'. When a browser loads a webpage, it takes all of the HTML and creates a model from it. Using Javascript we can access and manipulate that model. Adding and removing elements, changing attributes of elements, and changing styling of elements.

The script Element

We can inject our Javascript code into an HTML page by using the script element. We can do this two ways.
First is to insert opening and closing script tags in the head element, the same way we would use the title or style elements. We then insert our Javascript code directly on the HTML page inline.

```
    <html>
        <head>
            <script>
                // Here is our javascript code.
                alert('In our Javascript code');
            </script>
        </head>
    </html>
```

The second way is to use the script tag to retrieve our external Javascript file and inject that into our HTML page. Note, the attributes (flags) we use in script are type which should be set to "text/javascript" and src which will be set to the location of your file. We also want to include the keyword async and the end of our script tag to tell the browser to load the script asynchronously from the HTML. NOTE: script is not a self closing tag, you must include a closing tag.

```
    <html>
        <head>
            <script type="text/javascript" src="./index.js"
async></script>
        </head>
    </html>
```

document

The first thing to note about Javascript running on a webpage is it's access to a global object called document. Remember that DOM stands for Document Object Model, the document object contains our DOM and prototype methods that allow us to access elements on the DOM and manipulate them.

document Selectors

document contains dozens of methods on it's prototype. But most useful are it's selectors. We will take a look at the five most common.

document.getElementsByClassName

getElementsByClassName will find elements based on their class names. It will return an array-like object that we can use to iterate through. The class name supplied will be a string with the class name.

```
const divs = document.getElementsByClassName('divClass');
```

document.getElementById

getElementById will find a single element based on it's id. It will return the element itself. The id supplied must be a string of the id name.

```
const div = document.getElementById('divId');
```

document.querySelector

querySelector (and querySelectorAll) is a new method that takes a CSS style selector as it's argument. Remember that we can ask for classes in CSS using the ., ids using the #, and elements by using the element name (eg: 'body'). These selectors will use the same format. It is best to only use ids with querySelector because it will only return the first item matching that selector.

```
const div = document.querySelector('#divId');
```

document.querySelectorAll

querySelectorAll works just like querySelector except it returns an array like object containing all elements that match the selector. Because of this, you can use ids OR class names with this method.

```
const divs = document.querySelectorAll('#divId');
```

document.createElement

If we want to create an element to be added to our DOM, we can use document.createElement. This method takes one argument, the element type and returns an empty element of that type.

```
const newDiv = document.createElement('div');
```

Element Methods and Properties

Once we have our elements selected we can use a wide range of methods and properties to affect everything on the element, including: changing the CSS styles, changing the attributes on the element, adding or removing the children of the element, adding or removing event listeners(clicks, etc). There are countless things we can do to affect the element. We will go over some basic methods and properties here.

.innerHTML

When we have an element, we can set it's innerHTML. This is essentially setting the data that is stored between the opening and closing tags of the element.

```
    const p = document.querySelector('#pId');
    console.log(p.innerHtml) // This is the text between the
<p></p> tags

    p.innerHTML = 'This is new text to display between the
tags';

    console.log(p.innerHTML); // This is new text to display
between the tags
```

.[attribute] and .setAttribute

We can call .setAttribute on an element to either add an attribute to the element or reassign one that is already on that element. calling .[name of attribute] = [new value] is a shorthand way of doing this.

```
const img = document.querySelector('#imgId');

img.setAttribute('src',
'http://10000coders.com/assets/10000coders-logo.png');

img.src = 'http://10000coders.com/assets/10000coders-
logo.png';
```

.style

Calling the .style property on an element gives us access to the styles associated with the element. Note, this does not give us access to the CSS styles, only the inline styles written in HTML. We chain the style we want to read, or affect, on to the end of the .style .We can use this to set certain styles on the element.

```
const div = document.querySelector('#divId');

div.style.height = '300px';
div.style.background = 'red';
```

.className and .id

Using the .className and .id properties we can read and reassign class names and ids. This is most useful when we have two different styles associated and we want to switch the element to another style.

```
const div = document.querySelector('#divId');

console.log(div.id); // divId
div.className = 'newClassName';

div.id = 'newId';
```

.appendChild

We have the ability to create a new element set its style, class, id, attributes, and innerHTML, and add it to the DOM directly. To do this we use .appendChild on a parent node:

```
const body = document.querySelector('body');
const newDiv = document.createElement('div');

body.appendChild(newDiv);
```

Event Listeners

An event listener is a function that fires when an event occurs. Events can be anything from a click, to a mouse entering the content area, to an image download finishing. We will explore a few different events, but there are dozens we can choose from.

Click

The most common event listener to assign to an element is the 'click handler' in fact, it is the only one with it's own property, .onclick. To use the onclick property, we set it equal to a function that we want fired each time the element is clicked.

```
const div = document.querySelector('#divId');
div.onclick = function() {
    console.log('clicked!');
};
```

addEventListener and Other Events.

.onclick works if we want to add a click listener, but what happens if we want to fire a function when a user enters text in a form input, or the screen is scrolled. There are dozens of built in event listeners, but we must use .addEventListener. .addEventListener is a method that takes two arguments, the first is the type of event it is listening for, and the second is a callback function that is called when that event happens. Note: it is best to use addEventListener for all events, even clicks.

```
const div = document.querySelector('#divId');
div.addEventListener('mouseenter', function() {
    console.log('mouse has entered!');
});
```

You can find a list of all events here: MDN: Events. Using these document methods coupled with the knowledge we have of Javascript, HTML, and CSS, we now how the tools to build a fully functional front end web application.

Conclusion

With the object model, JavaScript gets all the power it needs to create dynamic HTML:

- JavaScript can change all the HTML elements in the page
- JavaScript can change all the HTML attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page

Lesson 3: Building profile to impress employers

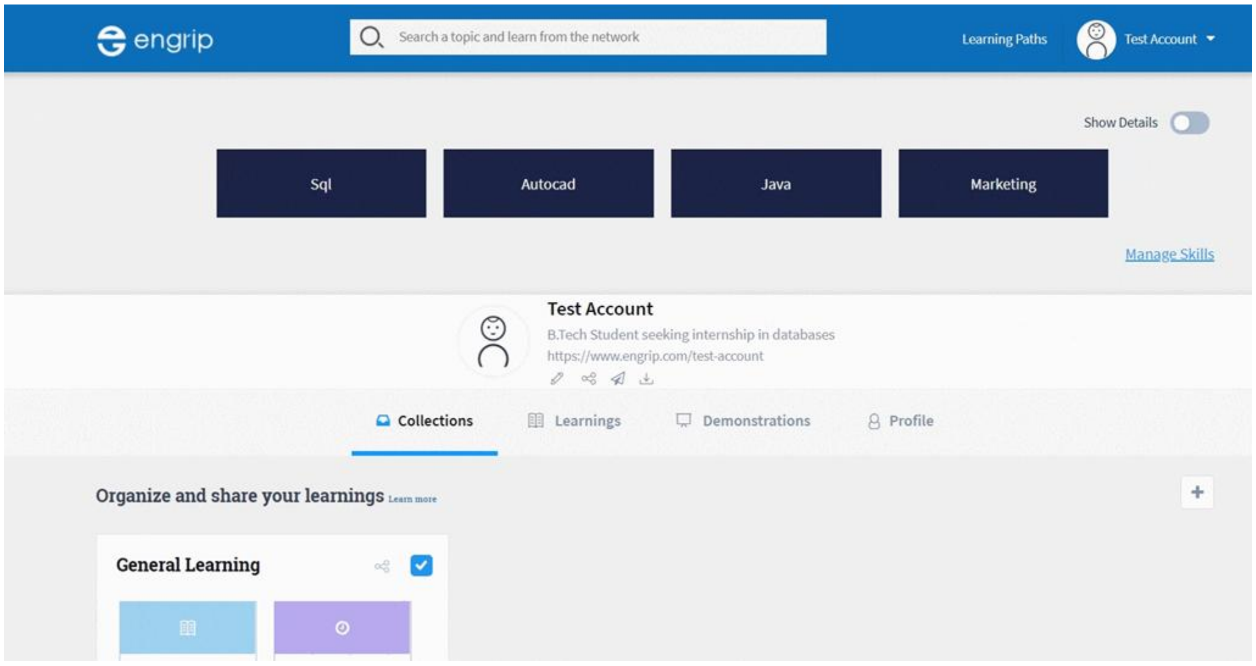
A successful job search begins with a great job profile. As they say "First Impression is the Best Impression" . So, you must present yourself better to an employer. This email explains you about how to use EnGrip to create a perfect job profile. And this is completely shareable as well, so if you spend time in building this one profile, you could use it with any number of employers. This is a ong email but worth reading. Please make sure that you carefully follow the instructions and fill the sections mentioned in this email as soon as possible .

Note that you can edit the knowledge profile anytime later. To complete the following actions, you must first login to EnGrip.

Skills

Companies are most interested in your skills than your degree or percentage. So make sure your skills on EnGrip are up to date and accurate.

Note Please enter proper skill name Ex. C is a valid skill name where as Introduction to C, Beginner in C, Basics of C are not valid skill titles. Please reach out to your career development cell or email us if you have any questions regarding skill titles.

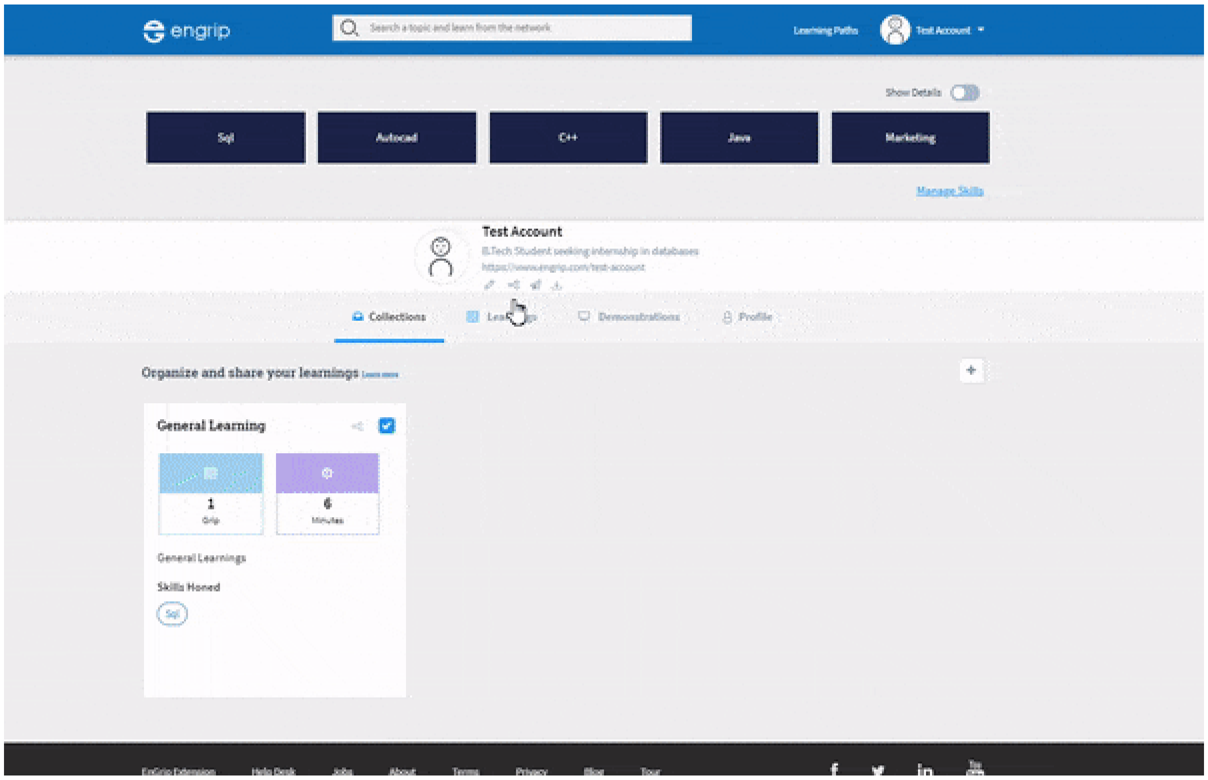

engrip
Search a topic and learn from the network
Learning Paths
Test Account
Show Details
Sql
Autocad
Java
Marketing
Manage Skills
Test Account
B.Tech Student seeking internship in databases
https://www.engrip.com/test-account
Collections
Learnings
Demonstrations
Profile
Organize and share your learnings Learn more
General Learning

Demonstrations

Add demonstrations as proof of your skills and increase your chances of hiring. EnGrip supports five types of demonstrations.

1. **Add Project** Add your min/major project or any freelancing work that you have done here.
2. **Add Accomplishment** Add your paper presentation, hackathons, programming or product building challenges here.
3. **Add clubs, affiliations** Add your volunteer activities inside or outside your college here. These activities demonstrate your leadership or organizational skills.
4. **Add Courses** Add any courses that you have done (online/offline) outside your curriculum here.
5. **Add Certifications** Add any certificates your have received here

Note It is advised to mention the skills demonstrated along with each and every demonstration. This will help employers find candidates with relevant knowledge quickly. Also you can attach image or pdf documents with each demonstration to visually demonstrate your application of skills.



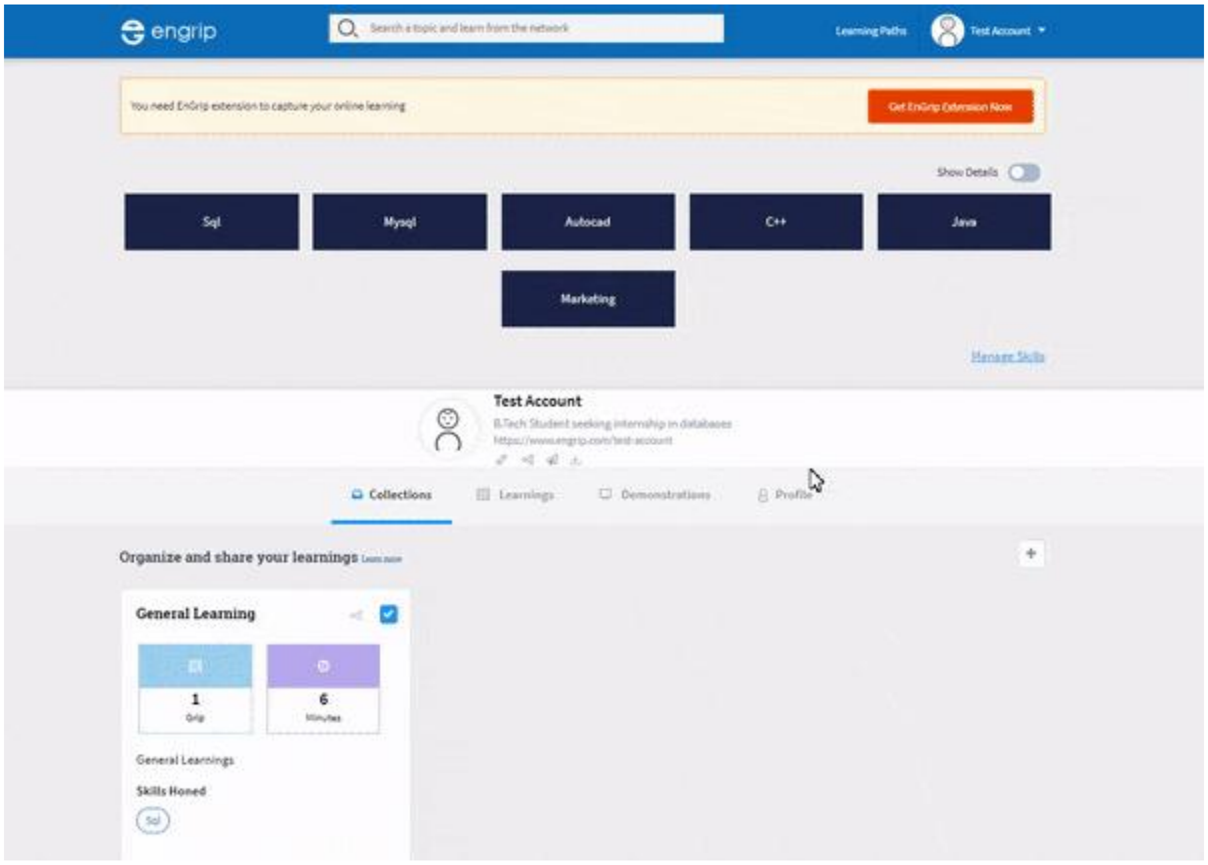
Test Account
B.Tech Student seeking internship in databases
https://www.engrip.com/test-account

Organize and share your learnings

General Learning
1 Grip
6 Minutes
General Learnings
Skills Honed
Sql

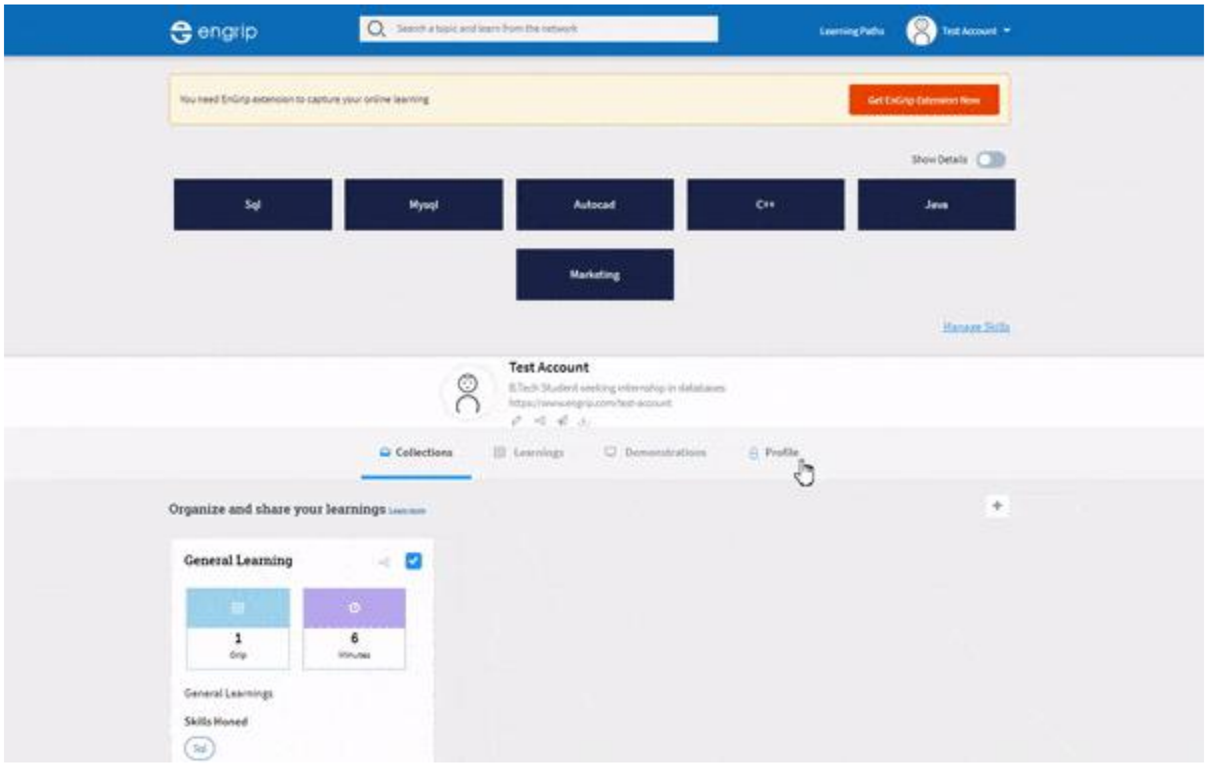
Profile

This is your general resume section and easy peasy to fill. Focus on the following details while filling this section.
Career Objective It acts as the pitch of your resume. It mentions the goal and objective of your career .
Education Mention your diploma/graduation/post graduation details here. Don't mention your X or XII details here.
General Information Mention your contact number, languages known and hobbies here.
Professional Summary For those of you who have done your internship please mention your internship details here. Leave this blank if you have not done any internship.



Job Profile

Mention your percentages and education details here. For percentages, just mention numbers upto two decimal places. Don't add any text like upto 3-2, CGPA etc. Software will automatically identify if you are entering CGPA.



You can refer to "EnGrip - Getting Started" user guide explaining how to get started on EnGrip and how to use it effectively for better results. A good profile will impress your employers and increase your chances of hiring dramatically!

Also make sure your social profiles are clean and have your presence on Linkedin.