This/Hoisting/Closure/Use Strict:

- **This:**
- **This** depends on the context. Hence, we need to specify the context.
- If we are using **"use strict"** and do not specify the context, then we get an error. However, if we specify the context, but use an attribute that has not been initialized yet, then we will get **"undefined"**.
  I.e. Under **"use strict"**, if **"this"** is undefined, then we get an error, but if the attribute is undefined, then we get **"undefined"**.
  E.g. Consider the code and output below:

```
"use strict"

// Example of undefined.
const person = {
    "name": "Rick",
    "test": function(){
        console.log(this.Name)
    }
}

person.test()

// Example of an error.
const person2 = {
    "name": "Rick",
    "test": function(){
        console.log(this.name)
    }
}

const x = person2.test
console.log(x)
x()
```

```
undefined
ƒ (){
        console.log(this.name)
    }
Uncaught TypeError: Cannot read property 'name' of undefined
    at test (test.js:17)
    at test.js:23
```

In the first example, **"this"** is defined. It's referring to the person object. However, there

is no property called Name, so **console.log(this.Name)** prints **"undefined"**. In the second example, **"this"** is undefined, so we get an error.

- However, if you do not use **"use strict"**, we get a different result.
  E.g. Consider the code and output below:

```
const person = {
    "name": "Rick",
    "test": function(){
        console.log(this)
    }
}

person.test()

// Example of windows object.
const person2 = {
    "name": "Rick",
    "test": function(){
        console.log(this)
    }
}

const x = person2.test
x()
```

```
▶ {name: "Rick", test: f}
▶ Window {parent: Window, opener: null, top: Window, length: 0, frames: Window, …}
```

When **"use strict"** is not used, **"this"** is bound to the window object of the browser. Hence, in the second example, you see that **console.log(this)** prints Window, as I did not define **"this"**. The first example still shows that **"this"** refers to the person object.

- **Hoisting:**
- E.g. Consider the code and output below:

```
"use strict"

var a = 7
function bar(){
    console.log("a is ", a)
    var a = 3
}

bar()
```

```
a is  undefined
```

Recall that when you use **var**, the declaration and definition gets separated and that the declaration gets hoisted to the top of the function or body. In the above example, what the code is actually doing is this:

```
"use strict"

var a = 7
function bar(){
    var a
    console.log("a is ", a)
    a = 3
}

bar()
```

Hence, when we do **console.log("a is ", a)**, it prints undefined, as **"a"** has not been defined yet.
Now, consider the code and output below again.

```
"use strict"

var a = 7
function bar(){
    console.log("a is ", a)
    a = 3
}

bar()
```

```
a is  7
```

Here, since there's no **var** before **a = 3**, **a = 3** doesn't get hoisted, and thus, at the time of **console.log("a is ", a)**, **"a"** = 7.

- E.g. Consider the code and output below:

```
"use strict"

var b = 7
function f(){
    b = 10
    function b(){

    }
    return b
}

var a = f()
console.log(b)
console.log(a)
```

```
7
10
```

Here, the function b() is hoisted to the top of function f(). Hence, when we do return b, it returns the variable **"b"** and not the function, and hence, it outputs 10.

- **Closure:**
- **Closure** is when a function has access to the parent scope, even after the parent function has closed.
- E.g. Consider the code and output below:

```
"use strict"

var b = 7
function func1(){
    var b = 10
    function func2(){
        console.log(b)
    }
    return func2
}

var a = func1()
console.log(a)
a()
```

```
f func2(){
        console.log(b)
    }
10
```

Here, after func1() has ended, func2() still has access to the variable b.

- E.g. Consider the code and output below:

```
"use strict"

var b = 7
function func1(){
    var b = 10
    function func2(){
        console.log(b)
    }
    b = 3
    b = 4
    b = 5
    b = 6
    return func2
    b = 7
}

var a = func1()
console.log(a)
a()
```

```
ƒ func2(){
        console.log(b)
    }
6
```

Note: Anything after the return statement is ignored.

- E.g. Consider the code and output below:

```
"use strict"

function add(){
    var counter = 0
    return function(){
        counter += 1
        console.log(counter)
    }
}

var x = add
var y = x()
y() // Prints 1
y() // Prints 2
y() // Prints 3
y() // Prints 4
```

```
1
2
3
4
```

The inner function still has access to “counter” even after add() has been exited.

Difference between function and function():

- If you access a function without the parentheses, (), you will get the function definition back. If you access a function with parentheses, you get its return value back.
- E.g. Consider the code and output below:

```
"use strict"

function add(){
    var counter = 0
    return function(){
        counter += 1
        return counter
    }
}

var x = add
console.log(x)
var y = x()
console.log(y)
console.log(y())
```

```
f add(){
    var counter = 0
    return function(){
        counter += 1
        return counter
    }
}
f (){
        counter += 1
        return counter
    }
1
```

Here, when we did var x = add, because we didn't use (), x got the function definition of the add function back. Hence, when we do console.log(x), it prints out the function definition of the add function. Next, when we did var y = x(), y gets the return value of the add function, which is the inner function in this case. Hence, when we did console.log(y), it printed the function definition of the inner function, but when we did console.log(y()), it printed the return value of the inner function, which is the value of "counter".

- E.g. Consider the code and output below:

```
"use strict"

function add(){
    var counter = 0
    return function(){
        counter += 1
        return counter
    }
}

var x = add()
console.log(x)
var y = x()
console.log(y)
```

```
f (){
        counter += 1
        return counter
    }
1
```

Here, when we did var x = add(), x got the return value of add(), which is the inner function. When we did var y = x(), y got the return value of the inner function, which is the value of “counter”.

Prototypes:

- JS works on a delegation framework. If a property can’t be found in an object, JS looks for that property in a delegate object. Delegate objects can be chained.
- **Prototypes** are objects that are used by other objects to add delegate properties.
- Prototypes are not superclasses. No new instances are created for each object.
- An object will just have a reference to its delegate prototype.
- When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property which holds a link to another object called its **prototype**. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. By definition, null has no prototype, and acts as the final link in this **prototype chain**.
- JavaScript objects are dynamic bags of properties. JavaScript objects have a link to a prototype object. When trying to access a property of an object, the property will not only be sought on the object but on the prototype of the object, the prototype of the prototype, and so on until either a property with a matching name is found or the end of the prototype chain is reached.
- E.g. Consider the code and output below:

```
"use strict"

function sayName(){
    console.log("My name is " + this.FirstName + " " + this.LastName + ".")
}

const person = {
    "sayName": sayName
}

const student = {
    "FirstName": "Rick",
    "LastName": "Lan"
}

// student.sayName() Will give an error.
Object.setPrototypeOf(student, person)
student.sayName()
```

```
My name is Rick Lan.
```

Here, “student” doesn’t have a way to access the sayName function, but, “person” does. Hence, when we do **Object.setPrototypeOf(student, person)**, we’re making “person” a prototype of “student”. Now, when we do **student.sayName()**, it will first check if student can access sayName(), and if it can’t, it will check if “student’s” prototype, in this case “person”, can access sayName(). “person” can access sayName() and so **student.sayName()** is able to run.

- E.g. Consider the code and output below:

```
"use strict"

function sayName(){
    console.log("My name is " + this.FirstName + " " + this.LastName + ".")
}

const chain_link_1 = {
    "sayName": sayName
}

const chain_link_2 = {
    "studentNumber": "12222344444"
}

const chain_link_3 = {
    "utorid": "abcdefg"
}

const chain_link_4 = {
    "FirstName": "Rick",
    "LastName": "Lan"
}

Object.setPrototypeOf(chain_link_4, chain_link_3)
Object.setPrototypeOf(chain_link_3, chain_link_2)
Object.setPrototypeOf(chain_link_2, chain_link_1)
chain_link_4.sayName()
```

```
My name is Rick Lan.
```

Here, notice that only chain_link_1 has access to sayName(). However, chain_link_4 is able to access it, because chain_link_1 is a prototype of chain_link_2, which is a prototype of chain_link_3, which is a prototype of chain_link_4. When trying to access a property of an object, the property will not only be sought on the object but on the prototype of the object, the prototype of the prototype, and so on until either a property with a matching name is found or the end of the prototype chain is reached.

- Multiple objects can have the same prototype object reference.
- E.g. Consider the code and output below:

```
"use strict"

function sayName(){
    console.log("My name is " + this.FirstName + " " + this.LastName + ".")
}

const person = {
    "sayName": sayName
}

const student1 = {
    "FirstName": "Rick",
    "LastName": "Lan"
}

const student2 = {
    "FirstName": "A",
    "LastName": "B"
}

const student3 = {
    "FirstName": "C",
    "LastName": "D"
}

// For student1
Object.setPrototypeOf(student1, person)
student1.sayName()

// For student2
Object.setPrototypeOf(student2, person)
student2.sayName()

// For student3
Object.setPrototypeOf(student3, person)
student3.sayName()
```

```
My name is Rick Lan.
My name is A B.
My name is C D.
```

Here, we see that “person” is the prototype for “student1”, “student2”, and “student3”.

- The main purpose of a prototype is for fast object creation.

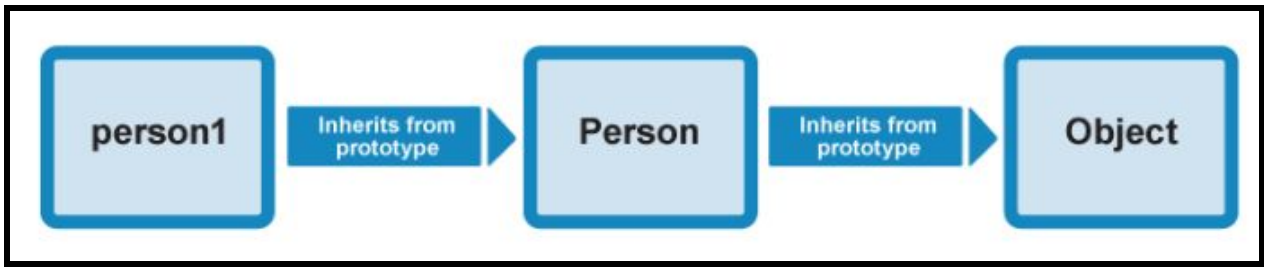
New Keyword:

- The **new** operator lets developers create an instance of a user-defined object type or of one of the built-in object types that has a constructor function.
- 4 things that **new** does (in order):
    1. Creates a new empty object.
    2. Sets the new object's delegate prototype (the **__proto__**) to the constructor's prototype property value.
    3. Calls the constructor function with this set to the new object.
    4. Returns the new object.

__proto__ vs prototype:

- **__proto__** is the property of an object that points to the object's delegate prototype.
- **prototype** is the property of a function that points to the object which, when that function is called as a constructor, will be assigned to a new object's __proto__.
- I.e. **__proto__** is the actual object that is used in the lookup chain to resolve methods, whereas **prototype** is the object that is used to build **__proto__** when you create an object with **new**.
- E.g. Consider the code and output below:

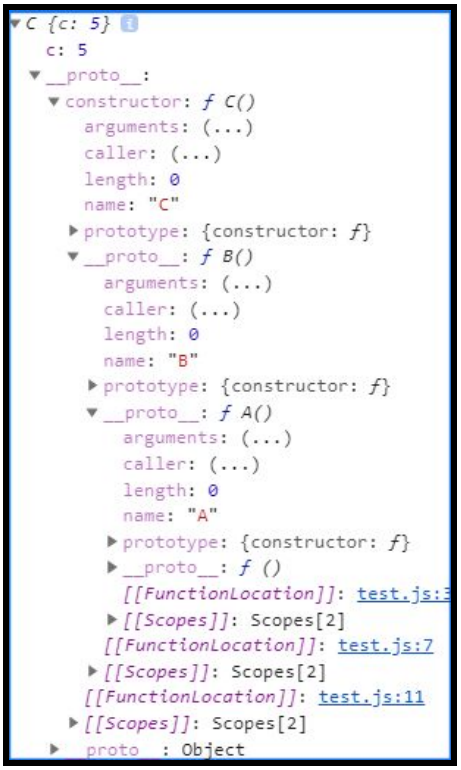
```
"use strict"

function Person(firstname, lastname, age, gender){
    this.name = {
        'first': firstname,
        'last': lastname
    }

    this.age = age
    this.gender = gender
}

const person1 = new Person("Rick", "Lan", "20", "Male");
console.log(person1)
```



See here that person1's __proto__ points to Person(), and Person()'s __proto__ points to Object.

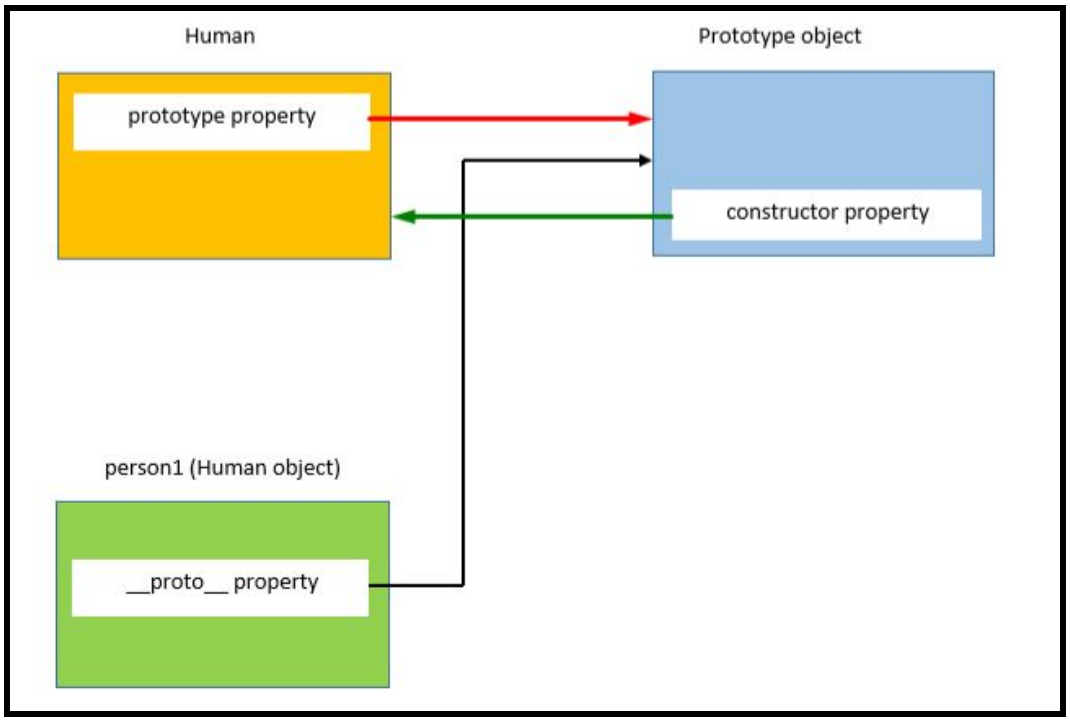
- E.g. Consider the code and output below:

```
"use strict"

function A(){
    this.a = 3
}

function B(){
    this.b = 4
}

function C(){
    this.c = 5
}

Object.setPrototypeOf(C, B)
Object.setPrototypeOf(B, A)
const x = new C()
console.log(x)
```



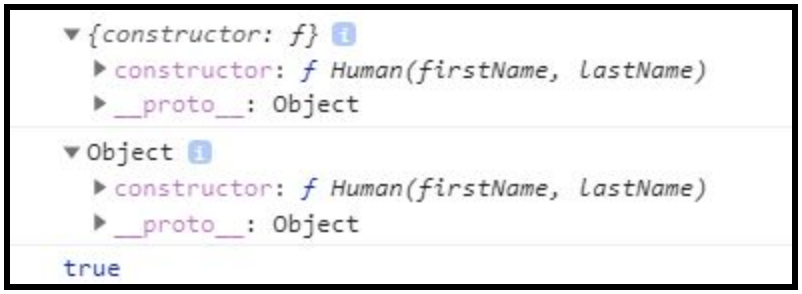
Here, x's __proto__ points to C(), while C()'s __proto__ points to B(), while B()'s __proto__ points to A() and A()'s __proto__ points to Object.

- When a function is created in JavaScript, the JavaScript engine adds a prototype property to the function. This prototype property is an object that has a constructor property by default. The constructor property points back to the function on which prototype object is a property. We can access the function's prototype property using **functionName.prototype**. Furthermore, when an object is created in JavaScript, the JavaScript engine adds a __proto__ property to the newly created object which is called __proto__ points to the prototype object of the constructor function.



E.g. Consider the code and output below:

```
"use strict"

function Human(firstName, lastName) {
    this.firstName = firstName,
    this.lastName = lastName
}

const person1 = new Human("Rick", "Lan");
console.log(person1.__proto__)
console.dir(Human.prototype)
console.log(person1.__proto__ === Human.prototype)
```



Notice that person1.__proto__ and Human.prototype are pointing to the same constructor property.

- E.g. Consider the code and output below:

```
"use strict"

function Human(firstName, lastName) {
    this.firstName = firstName,
    this.lastName = lastName
}

const person1 = new Human("Rick", "Lan");
const person2 = new Human("Rick", "Lan");
const person3 = new Human("Rick", "Lan");

Human.prototype.age = 5
console.log(person1.age)
console.log(person2.age)
console.log(person3.age)
```

```
5
5
5
```

Since Human() is the prototype for person1, person2, and person3, when I do **Human.prototype.age = 5**, it sets person1.age, person2.age and person3.age to 5.

Object.create():

- Another way to create objects using prototypes is by using **Object.create(o)**. It creates an object with “o” as the prototype.
- It can create multiple objects with the same delegate prototype (__proto__), but remember that all of their __proto__ properties will point to the same reference. No new instances or copies of the delegate are created.

- E.g. Consider the code and output below:

```
"use strict"

const person = {
    intro: function(){
        console.log("Hi. My name is " + this.name)
    }
}

const me = Object.create(person)
me.name = "Rick Lan"
me.intro()
```

```
Hi. My name is Rick Lan
```

- E.g. Consider the code and output below:

```
"use strict"

function Human(firstName, lastName) {
    this.firstName = firstName,
    this.lastName = lastName
}

const person1 = new Human("Rick", "Lan");

Human.prototype.sayLastName = function(){
    console.log("My last name is " + this.lastName)
}
person1.sayLastName()

const person2 = Object.create(Human.prototype)
person2.sayLastName()
```

```
My last name is Lan
My last name is undefined
```

Here, person2's lastName value is undefined, so when we do person2.sayLastName(), it will print "My last name is undefined"

Classes:

- A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class, and the properties are assigned inside a constructor() method.
- **Note:** The constructor method is called automatically when the object is initialized.
- Mostly, it's just a neat way to repackage prototypes and object creation in a way that's more digestible for object-oriented programmers. There are no private variables.
- Syntax:

```
class MyClass {
  // class methods
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  method3() { ... }
  ...
}
```

- E.g. Consider the code and output below:

```
"use strict"

class Human {
    constructor (firstname, lastname){
        this.firstname = firstname,
        this.lastname = lastname
    }
}

console.dir(Human)
```

```
▼class Human
    arguments: (...)
    caller: (...)
    length: 2
    name: "Human"
  ▶prototype: {constructor: ƒ}
  ▶__proto__: ƒ ()
    [[FunctionLocation]]: test.js:4
  ▶[[Scopes]]: Scopes[2]
```

Notice that the class Human has prototype and __proto__ properties. Recall that only functions have prototype and __proto__ properties, hence, Human is a function.

- E.g. Consider the code and output below:

```
"use strict"

class Human {
    constructor (firstname, lastname){
        this.firstname = firstname,
        this.lastname = lastname
    }

    getFirstName(){
        return this.firstname
    }

    getLastName(){
        return this.lastname
    }

    getFullName(){
        return this.firstname + " " + this.lastname
    }
}

const person1 = new Human("Rick", "Lan")
console.log(person1.getFirstName())
console.log(person1.getLastName())
console.log(person1.getFullName())
```

```
Rick
Lan
Rick Lan
```

- E.g. Notice how flimsy classes can be in the example below:

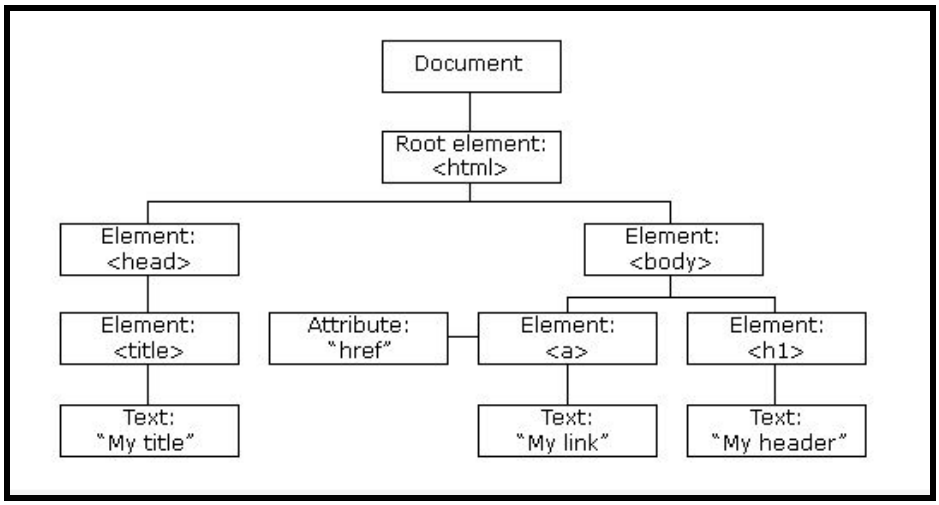
```
"use strict"

class Human {
    constructor (firstname, lastname){
        this.firstname = firstname,
        this.lastname = lastname
    }

    getFirstName(){
        return this.firstname
    }

    getLastName(){
        return this.lastname
    }

    getFullName(){
        return this.firstname + " " + this.lastname
    }
}

const person1 = new Human("Rick", "Lan")
person1.firstname = "ABC" // Changed person1's first name to ABC
console.log(person1.getFirstName()) // Prints ABC
Human.prototype.getFullName = 5
console.log(person1.getFullName) // Prints 5
```

```
ABC
5
```

Notice that we're able to modify values very easily.

DOM:

- Stands for **Document Object Model**.
- When a web page is loaded, the browser creates a Document Object Model of the page.
- The HTML DOM model is constructed as a tree of Objects.



- The entire document is a document node.
- Every HTML element is an element node.
- The text inside HTML elements are text nodes.

- Every HTML attribute is an attribute node.
- All comments are comment nodes.
- 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.
- **HTML DOM methods** are actions you can perform on HTML Elements.
- **HTML DOM properties** are values of HTML Elements that you can set or change.
- In the DOM, all HTML elements are defined as objects.
- A **property** is a value that you can get or set (like changing the content of an HTML element).
- A **method** is an action you can do (like add or deleting an HTML element).
- **Finding HTML Elements:**

Method	Description
document.getElementById(id)	Find an element by element id
document.getElementsByTagName(name)	Find elements by tag name
document.getElementsByClassName(name)	Find elements by class name

- **Changing HTML Elements:**

Property	Description
element.innerHTML = new html content	Change the inner HTML of an element
element.attribute = new value	Change the attribute value of an HTML element
element.style.property = new style	Change the style of an HTML element
Method	**Description**
element.setAttribute(attribute, value)	Change the attribute value of an HTML element

- **Adding and Deleting Elements:**

Method	Description
document.createElement(element)	Create an HTML element
document.removeChild(element)	Remove an HTML element
document.appendChild(element)	Add an HTML element

document.replaceChild(new, old)	Replace an HTML element
document.write(text)	Write into the HTML output stream

- **Finding HTML Element by Id:**
- To get the HTML element from its id, you can use the **document.getElementById() method**.
- If the element is found, the method will return the element as an object (in myElement).
- If the element is not found, myElement will contain null.
- E.g.

```
<!DOCTYPE html>
<html>
    <body>
        <p id="test"> Sample Text </p>
        <script type="text/javascript" src="test.js"></script>
    </body>
</html>
```

```
"use strict"

const text = document.getElementById("test")
console.log(text)
```

```
<p id="test"> Sample Text </p>
```

- E.g.

```
<!DOCTYPE html>
<html>
    <body>
        <p id="test"> Sample Text </p>
        <script type="text/javascript" src="test.js"></script>
    </body>
</html>
```

```
"use strict"

const text = document.getElementById("test").innerHTML
console.log(text)
```

```
Sample Text
```

- **Finding HTML Elements by Tag Name:**
- To find HTML elements by their tag name, use **document.getElementsByTagName()**.
- **Note: document.getElementsByTagName()** returns an HTML Collection.

- E.g.

```html
<!DOCTYPE html>
<html>
    <body>
        <p id="test"> Sample Text </p>
        <script type="text/javascript" src="test.js"></script>
    </body>
</html>
```

```javascript
"use strict"

const text = document.getElementsByTagName("p")
console.log(text[0])
console.log(text[0].innerHTML)
```

```
<p id="test"> Sample Text </p>
Sample Text
```

- E.g.

```
"use strict"

const text = document.getElementsByTagName("p")

for (let i = 0; i < text.length; i++){
    console.log(text[i])
    console.log(text[i].innerHTML)
}
```

```
 <p> Sample Text 1 </p>
Sample Text 1
 <p> Sample Text 2 </p>
Sample Text 2
 <p> Sample Text 3 </p>
Sample Text 3
 <p> Sample Text 4 </p>
Sample Text 4
 <p> Sample Text 5 </p>
Sample Text 5
 <p> Sample Text 6 </p>
Sample Text 6
```

- **Finding HTML Elements by Class Name:**
- If you want to find all HTML elements with the same class name, use **getElementsByClassName()**.
- **Note: getElementsByClassName()** returns an HTML Collection.
- E.g.

```
<!DOCTYPE html>
<html>
    <body>
        <p class="text"> Sample Text 1 </p>
        <p class="text"> Sample Text 2 </p>
        <p class="text"> Sample Text 3 </p>
        <p class="text"> Sample Text 4 </p>
        <p class="text"> Sample Text 5 </p>
        <p class="text"> Sample Text 6 </p>
        <script type="text/javascript" src="test.js"></script>
    </body>
</html>
```

```
"use strict"

const text = document.getElementsByClassName("text")

for (let i = 0; i < text.length; i++){
    console.log(text[i])
    console.log(text[i].innerHTML)
}
```

```
 <p class="text"> Sample Text 1 </p>
Sample Text 1
 <p class="text"> Sample Text 2 </p>
Sample Text 2
 <p class="text"> Sample Text 3 </p>
Sample Text 3
 <p class="text"> Sample Text 4 </p>
Sample Text 4
 <p class="text"> Sample Text 5 </p>
Sample Text 5
 <p class="text"> Sample Text 6 </p>
Sample Text 6
```

- **Changing HTML Content:**
- The easiest way to modify the content of an HTML element is by using the innerHTML property.
- To change the content of an HTML element, use this syntax: **document.getElementById(id).innerHTML = new HTML**
- E.g.

```
<!DOCTYPE html>
<html>
    <body>
        <p id="text"> Sample Text 1 </p>
        <script type="text/javascript" src="test.js"></script>
    </body>
</html>
```

```
"use strict"

const text = document.getElementById("text")
text.innerHTML = "New Text"
```

New Text

- **Changing the Value of an Attribute:**
- To change the value of an HTML attribute, use this syntax:
  document.getElementById(id).attribute = new value
- E.g.

```
<!DOCTYPE html>
<html>
    <body>
        <a id="link" href="https://www.google.com/"> Google Link</a>
        <script type="text/javascript" src="test.js"></script>
    </body>
</html>
```

```
"use strict"

document.getElementById("link").href = "https://www.facebook.com/" // Links to facebook instead of Google.
```

- **Query Selector:**
- Returns a nodelist instead of an HTML Collection.
- The difference between node list and HTML Collection is that an HTMLCollection is a collection of HTML elements while a nodelist is a collection of document nodes.
- Taken from JQuery.
- To access a class, do .classname.
- To access an id, do #idname.
- E.g.

```
<!DOCTYPE html>
<html>
    <body>

        <div id="Text1">
            <p id="SampleText1"> Sample Text 1</p>
            <p class="test"> Sample Text 2 </p>
        </div>

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

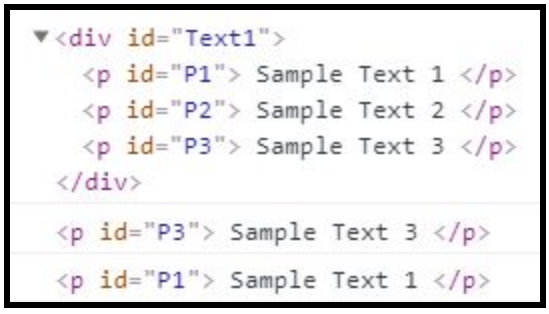
```
"use strict"

const test = document.querySelector("#SampleText1")
console.log(test)

const test2 = document.querySelector(".test")
console.log(test2)
```

```
<p id="SampleText1"> Sample Text 1</p>
<p class="test"> Sample Text 2 </p>
```

- **DOM Relationships:**
- The nodes in the node tree have a hierarchical relationship to each other.
- The terms **parent**, **child**, and **sibling** are used to describe the relationships.
- In a node tree, the top node is called the **root** or **root node**.
- Every node has exactly one **parent**, except the root which has no parent.
- A node can have a number of children.
- **Siblings** are nodes with the same parent.
- You can use the following methods to navigate between nodes with JavaScript:
    - parentElement
    - children
    - firstElementChild
    - lastElementChild
    - nextElementSibling
    - previousElementSibling
- E.g.

```
<!DOCTYPE html>
<html>
    <body>

        <div id="Text1">
            <p id="P1"> Sample Text 1 </p>
            <p id="P2"> Sample Text 2 </p>
            <p id="P3"> Sample Text 3 </p>
        </div>

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

```
"use strict"

const p1_family = document.querySelector("#P2")
console.log(p1_family.parentElement) // Gets the parent element.
console.log(p1_family.nextElementSibling) // Gets the next sibling element.
console.log(p1_family.previousElementSibling) // Gets the previous sibling element.
```



- E.g.

```html
<!DOCTYPE html>
<html>
    <body>

        <div id="Text1">
            <p id="P1"> Sample Text 1 </p>
            <p id="P2"> Sample Text 2 </p>
            <p id="P3"> Sample Text 3 </p>
        </div>

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

```javascript
"use strict"

const parent_element = document.querySelector("#Text1")
console.log(parent_element.parentElement) // Gets the parent element.
console.log(parent_element.firstElementChild) // Gets the first child element.
console.log(parent_element.lastElementChild) // Gets the last child element.
console.log(parent_element.children) // Gets all child element(s).
```

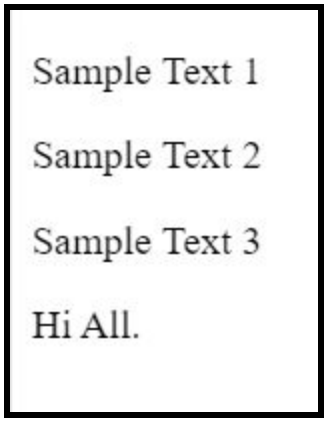
```
▶ <body>…</body>
  <p id="P1"> Sample Text 1 </p>
  <p id="P3"> Sample Text 3 </p>
▶ HTMLCollection(3) [p#P1, p#P2, p#P3, P1: p#P1, P2: p#P2, P3: p#P3]
```

- You can also create new HTML elements (nodes). To add a new element to the HTML DOM, you must create the element first, and then append it to an existing element. You can use the following methods to create new HTML elements:
    - createElement
    - createTextNode
    - appendChild
    - insertBefore
- **Note:** appendChild appends the new element as the last child of the parent.
  Syntax: **parent_node.append(child_node)**
- **Note:** The insertBefore() method inserts a node as a child, right before an existing child, which you specify.
  Syntax: **parent_node.insertBefore(new_node, existing_node)**

- E.g. of append()

```
<!DOCTYPE html>
<html>
    <body>

        <div id="Text1">
            <p id="P1"> Sample Text 1 </p>
            <p id="P2"> Sample Text 2 </p>
            <p id="P3"> Sample Text 3 </p>
        </div>

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

```
"use strict"

const parent_element = document.querySelector("#Text1")
const paragraph_tag = document.createElement('p')
const paragraph_text = document.createTextNode("Hi All.")
paragraph_tag.append(paragraph_text)
parent_element.append(paragraph_tag)
```



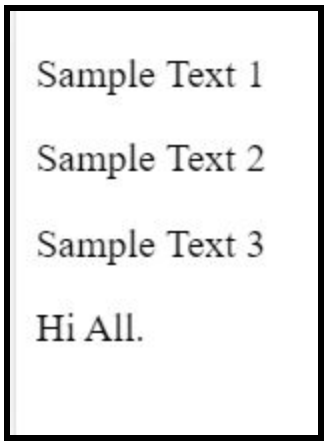
Notice that “Hi All.” appears at the end.

- **Note:** The order in which you append nodes matter. In the example above, I had to append textNode to createElement, and then append createElement to the tag.

- **Note:** If you append() on the same node twice with the same parent element, nothing new will happen.
  E.g.

```
<!DOCTYPE html>
<html>
    <body>

        <div id="Text1">
            <p id="P1"> Sample Text 1 </p>
            <p id="P2"> Sample Text 2 </p>
            <p id="P3"> Sample Text 3 </p>
        </div>

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

```
"use strict"

const parent_element = document.querySelector("#Text1")
const first_child = document.querySelector("#P1")
const paragraph_tag = document.createElement('p')
const paragraph_text = document.createTextNode("Hi All.")
paragraph_tag.append(paragraph_text)
parent_element.append(paragraph_tag)
parent_element.append(paragraph_tag)
parent_element.append(paragraph_tag)
parent_element.append(paragraph_tag)
```

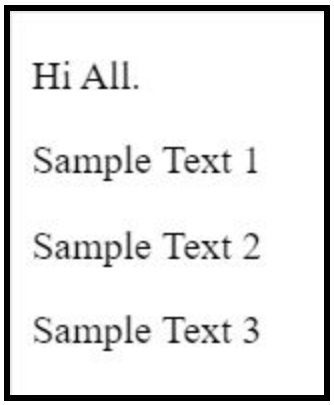


Notice that there's still only 1 "Hi All."

- E.g. of insertBefore()

```
<!DOCTYPE html>
<html>
    <body>

        <div id="Text1">
            <p id="P1"> Sample Text 1 </p>
            <p id="P2"> Sample Text 2 </p>
            <p id="P3"> Sample Text 3 </p>
        </div>

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

```
"use strict"

const parent_element = document.querySelector("#Text1")
const first_child = document.querySelector("#P1")
const paragraph_tag = document.createElement('p')
const paragraph_text = document.createTextNode("Hi All.")
paragraph_tag.append(paragraph_text)
parent_element.insertBefore(paragraph_tag, first_child)
```

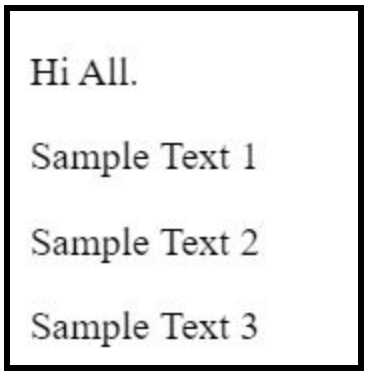


Notice here that “Hi All.” is at the beginning, instead of the end.

- **Note:** If you do multiple insertBefore() on the same parent element, same node and same child element, nothing new happens, just like how if you have multiple append()s on the same node and parent element, nothing happens.

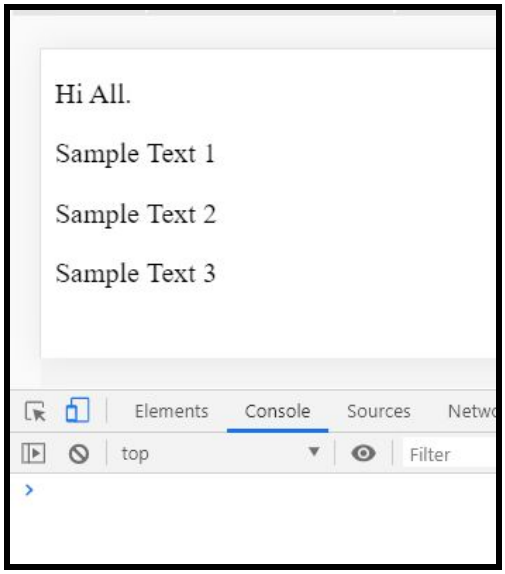
- E.g.

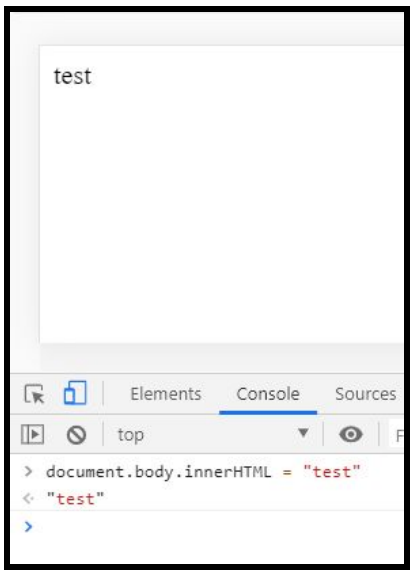
```
"use strict"

const parent_element = document.querySelector("#Text1")
const first_child = document.querySelector("#P1")
const paragraph_tag = document.createElement('p')
const paragraph_text = document.createTextNode("Hi All.")
paragraph_tag.append(paragraph_text)
parent_element.insertBefore(paragraph_tag, first_child)
parent_element.insertBefore(paragraph_tag, first_child)
parent_element.insertBefore(paragraph_tag, first_child)
parent_element.insertBefore(paragraph_tag, first_child)
```



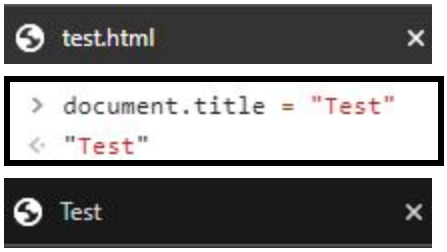
Notice that there's only one "Hi All."

- **Miscellaneous Information:**
- You can change the HTML using things like document.body or document.title.
- E.g.





- E.g.



Here, after I changed document.title to Test, the title of the page changed to Test.

- document.all gets all the elements.
  E.g.