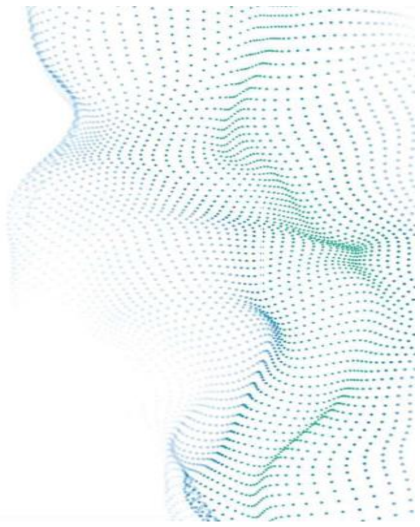
Module 1

JavaScript



Introduction to JavaScript

JavaScript (js) is a light-weight object-oriented programming language which is used by several websites for scripting the webpages. It is an interpreted, full-fledged programming language that enables dynamic interactivity on websites when applied to an HTML document. It was introduced in the year 1995 for adding programs to the webpages in the Netscape Navigator browser. Since then, it has been adopted by all other graphical web browsers. With JavaScript, users can build modern web applications to interact directly without reloading the page every time. The traditional website uses js to provide several forms of interactivity and simplicity.

Features of JavaScript

There are following features of JavaScript:

All popular web browsers support JavaScript as they provide built-in execution environments.
JavaScript follows the syntax and structure of the C programming language. Thus, it is a structured programming language.
JavaScript is a weakly typed language, where certain types are implicitly cast (depending on the operation).
JavaScript is an object-oriented programming language that uses prototypes rather than using classes for inheritance.
It is a light-weighted and interpreted language.

It is a case-sensitive language.

JavaScript is supportable in several operating systems including, Windows, macOS, etc.

It provides good control to the users over the web browsers.

History of JavaScript

In 1993, Mosaic, the first popular web browser, came into existence. In the year 1994, Netscape was founded by Marc Andreessen. He realized that the web needed to become more dynamic. Thus, a 'glue language' was believed to be provided to HTML to make web designing easy for designers and part-time programmers. Consequently, in 1995, the company recruited Brendan Eich intending to implement and embed Scheme programming language to the browser.

But, before Brendan could start, the company merged with Sun Microsystems for adding Java into its Navigator so that it could compete with Microsoft over the web technologies and platforms. Now, two languages were there: Java and the scripting language. Further, Netscape decided to give a similar name to the scripting language as Java's. It led to 'Javascript'. Finally, in May 1995, Marc Andreessen coined the first code of Javascript named 'Mocha'. Later, the marketing team replaced the name with 'LiveScript'. But, due to trademark reasons and certain other reasons, in December 1995, the language was finally renamed to 'JavaScript'. From then, JavaScript came into existence.

Application of JavaScript

JavaScript is used to create interactive websites. It is mainly used for:

Client-side validation,

Dynamic drop-down menus,

Displaying date and time,

Displaying pop-up windows and dialog boxes (like an alert dialog box, confirm dialog box and prompt dialog box),

Displaying clocks etc.

JavaScript Example

```
<script>
document.write("Hello JavaScript by JavaScript");
</script>
```

Control Structures and Functions

Most programs don't operate by carrying out a straightforward sequence of statements. A code is written to allow making choices and several pathways through the program to be followed depending on shifts in variable values.

All programming languages contain a pre-included set of control structures that enable these control flows to execute, which makes it conceivable.

This tutorial will examine how to add loops and branches, i.e., conditions to our Python programs.

Types of Control Structures

Control flow refers to the sequence a program will follow during its execution.

Conditions, loops, and calling functions significantly influence how a Python program is controlled.

There are three types of control structures in Python:

Sequential - The default working of a program

Selection - This structure is used for making decisions by checking conditions and branching

Repetition - This structure is used for looping, i.e., repeatedly executing a certain piece of a code block.

Sequential

Sequential statements are a set of statements whose execution process happens in a sequence. The problem with sequential statements is that if the logic has broken in any one of the lines, then the complete source code execution will break.

```
a = 20
b = 10
c = a - b
d = a + b
e = a * b
print("The result of the subtraction is: ", c)
print("The result of the addition is: ", d)
print("The result of the multiplication is: ", e)
```

Output:

```
The result of the subtraction is:  10
The result of the addition is :
30
The result of the multiplication is:  200
```

Selection/Decision Control Statements

The statements used in selection control structures are also referred to as branching statements or, as their fundamental role is to make decisions, decision control statements.
A program can test many conditions using these selection statements, and depending on whether the given condition is true or not, it can execute different code blocks.
There can be many forms of decision control structures. Here are some most commonly used control structures:

Only if
if-else
The nested if
The complete if-elif-else

Simple if

If statements in Python are called control flow statements. The selection statements assist us in running a certain piece of code, but only in certain circumstances. There is only one condition to test in a basic if statement.

The if statement's fundamental structure is as follows:

Syntax

```
if <conditional expression> :
    The code block to be executed if the condition is True
```

These statements will always be executed. They are part of the main code.
All the statements written indented after the if statement will run if the condition giver after the if the keyword is True. Only the code statement that will always be executed regardless of the if the condition is the statement written aligned to the main code. Python uses these types of indentations to identify a code block of a particular control flow statement. The specified control structure will alter the flow of only those indented statements.

```
v = 5
t = 4
print("The initial value of v is", v, "and that of t is ",t)

# Creating a selection control structure
if v > t :
    print(v, "is bigger than ", t)
    v -= 2

print("The new value of v is", v, "and the t is ",t)

# Creating the second control structure
if v < t :
    print(v , "is smaller than ", t)
    v += 1

print("the new value of v is ", v)
```

```
if v < t :
    print(v , "is smaller than ", t)
    v += 1

print("the new value of v is ", v)

# Creating the third control structure
if v == t:
    print("The value of v, ", v, " and t,", t, ", are equal")
```

```
The initial value of v is 5 and that of t is  4
5 is bigger than  4
The new value of v is 3 and the t is  4
3 is smaller than  4
the new value of v is
4
The value of v,
4  and t, 4, are equal
```

if-else

If the condition given in if is False, the if-else block will perform the code t=given in the else block.

```
# Initializing two variables
v = 4
t = 5
print("The value of v is ", v, "and that of t is ", t)

# Checking the condition
if v > t :
    print("v is greater than t")
# Giving the instructions to perform if the if condition is not true
else :
    print("v is less than t")
```

Output:

```
The value of v is  4 and that of t is  5
v is less than t
```

Repetition
To repeat a certain set of statements, we use the repetition structure.
There are generally two loop statements to implement the repetition structure:

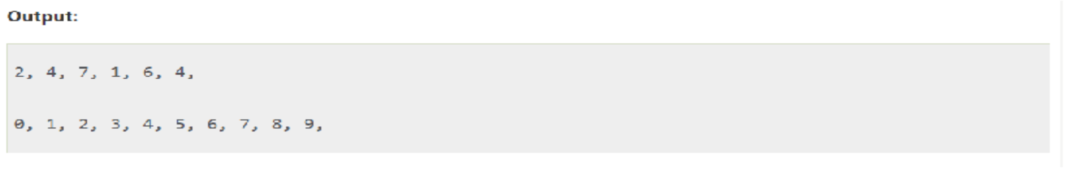
The for loop
The while loop

For Loop
We use a for loop to iterate over an iterable Python sequence. Examples of these data structures are lists, strings, tuples, dictionaries, etc. Under the for loop code block, we write the commands we want to execute repeatedly for each sequence item.

```
# Python program to show how to execute a for loop

# Creating a sequence. In this case, a list
l = [2, 4, 7, 1, 6, 4]

# Executing the for loops
for i in range(len(l)):
     print(l[i], end = ", ")
print("\n")
for j in range(0,10):
    print(j, end = ", ")
```



Output:

```
2, 4, 7, 1, 6, 4,

0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

While loops are also used to execute a certain code block repeatedly, the difference is that loops continue to work until a given precondition is satisfied. The expression is checked before each execution. Once the condition results in Boolean False, the loop stops the iteration.

```
b = 9
a = 2
# Starting the while loop
# The condition a < b will be checked before each iteration
while a < b:
    print(a, end = " ")
    a = a + 1
print("While loop is completed")
```

Output:

```
2 3 4 5 6 7 8 While loop is completed
```

JavaScript Functions

JavaScript functions are used to perform operations. We can call JavaScript function many times to reuse the code.

Advantage of JavaScript function

There are mainly two advantages of JavaScript functions.

Code reusability: We can call a function several times so it save coding.

Less coding: It makes our program compact. We don’t need to write many lines of code each time to perform a common task.

JavaScript Function Syntax

The syntax of declaring function is given below.

```
function functionName([arg1, arg2, ...argN]){
 //code to be executed
}
```

JavaScript Functions can have 0 or more arguments.

JavaScript Function Example

Let’s see the simple example of function in JavaScript that does not has arguments.

```
<script>
function msg(){
alert("hello! this is message");
}
</script>
<input type="button" onclick="msg()" value="call function"/>
```

JavaScript Function Arguments

We can call function by passing arguments. Let's see the example of function that has one argument.

```
<script>
function getcube(number){
alert(number*number*number);
}
</script>
<form>
<input type="button" value="click" onclick="getcube(4)"/>
</form>
```

Function with Return Value

We can call function that returns a value and use it in our program. Let's see the example of function that returns value.

```
<script>
function getInfo(){
return "hello javatpoint! How r u?";
}
</script>
<script>
document.write(getInfo());
</script>
```

JavaScript Function Object

In JavaScript, the purpose of Function constructor is to create a new Function object. It executes the code globally. However, if we call the constructor directly, a function is created dynamically but in an unsecured way.

Syntax

```
new Function ([arg1[, arg2[, ....argn]],] functionBody)
```

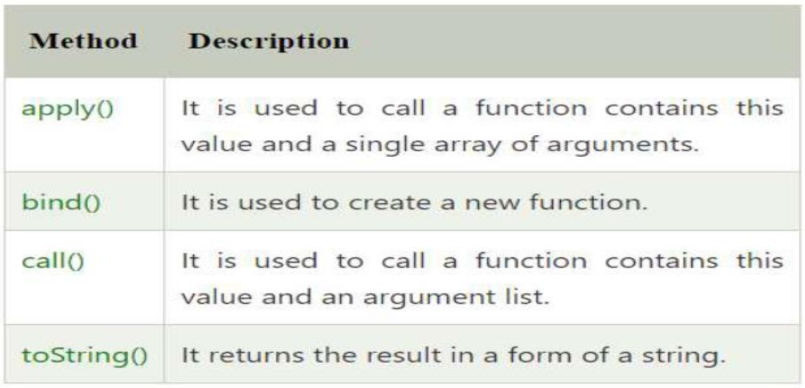
Parameter

arg1, arg2, ... , argn - It represents the argument used by function.

functionBody - It represents the function definition.

JavaScript Function Methods

Let's see function methods with description.



Method	Description
apply()	It is used to call a function contains this value and a single array of arguments.
bind()	It is used to create a new function.
call()	It is used to call a function contains this value and an argument list.
toString()	It returns the result in a form of a string.

JavaScript Function Object Examples

Example 1

Let's see an example to display the sum of given numbers.

```
<script>
var add=new Function("num1","num2","return num1+num2");
document.writeln(add(2,5));
</script>
```

Output:

```
7
```

JavaScript Array

JavaScript array is an object that represents a collection of similar type of elements.

There are 3 ways to construct array in JavaScript
By creating instance of Array directly (using new keyword)
By using an Array constructor (using new keyword)

1) JavaScript array literal
The syntax of creating array using array literal is given below:

```
var arrayname=[value1,value2.....valueN];
```

Let's see the simple example of creating and using array in JavaScript.

```
<script>
var emp=["Sonoo","Vimal","Ratan"];
for (i=0;i<emp.length;i++){
document.write(emp[i] + "<br/>");
}
</script>
```

The .length property returns the length of an array.

Output of the above example

```
Sonoo
Vimal
Ratan
```

JavaScript Array directly (new keyword)
The syntax of creating array directly is given below:

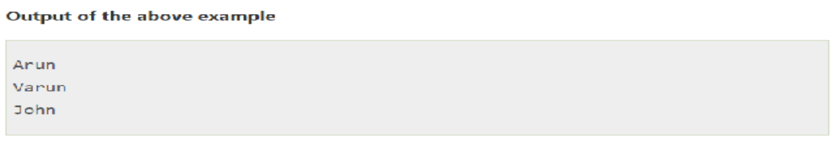
```
var arrayname=new Array();
```

Here, new keyword is used to create instance of array.

Let's see the example of creating array directly.

```
<script>
var i;
var emp = new Array();
emp[0] = "Arun";
emp[1] = "Varun";
emp[2] = "John";

for (i=0;i<emp.length;i++){
document.write(emp[i] + "<br>");
}
</script>
```



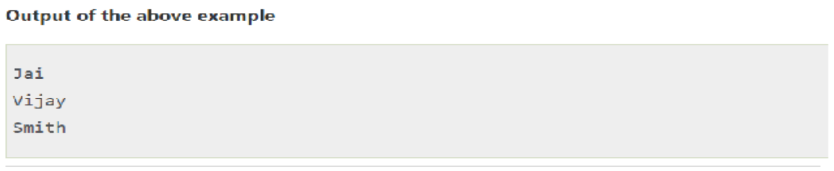
Output of the above example

```
Arun
Varun
John
```

JavaScript array constructor (new keyword)
Here, you need to create instance of array by passing arguments in constructor so that we don't have to provide value explicitly.

The example of creating object by array constructor is given below.

```
<script>
var emp=new Array("Jai","Vijay","Smith");
for (i=0;i<emp.length;i++){
document.write(emp[i] + "<br>");
}
</script>
```



Output of the above example

```
Jai
Vijay
Smith
```

Array of Class Objects in Java
The Classes and the objects are the foundation of the Java programming language because it is an object-oriented language. We have used a variables of type Object whenever we need to store just one object in our programme. However, using an Array of items is better when dealing with a large number of objects.

A collection of objects, the term alone implies that this stores a variety of the items. An array of objects contains the objects, which means that the objects were held as elements of an array, as opposed to the typical array which holds elements such String, integer, Boolean, etc. Keep in mind that when we refer to a "Array of Objects," what is actually stored inside the array is the reference to the object, not the actual object.

An array is a data structure that stores a fixed-size sequence of the elements of the same type. In Java, arrays are the objects, and they can be used to store the objects of any class type. It means that an array of class objects can be created and used to store instances of a particular class.

Array of Objects

After the object array is created, it needs to be initialised with values. Since the array is not the same as the array of primitive types, we are unable to initialise it in the same manner as we do with primitive types. Each element of the array, or each object or object reference, must be initialised when using an array of objects.

Creating an array of class objects in Java is similar to creating an array of any other type. The syntax for creating an array of class objects is as follows:

```
ClassName[] arrayName = new ClassName[size];
```

For example, to create an array of objects of the class "Person", we would use the following code:

```
Person[] people = new Person[10];
```

This creates an array named "people" that can hold up to 10 objects of the class "Person". However, it should be noted that the array is initially filled with null values and the objects needs to be initialized before they can be used.

```
people[0] = new Person("John Smith");
people[1] = new Person("Jane Doe");
```

```
people[0] = new Person("John Smith");
people[1] = new Person("Jane Doe");
```

Once an array of class objects is created, we can use the standard array access notation to access and manipulate the elements of the array. For example, to access the first element of the "people" array, we would use the following code:

```
Person firstPerson = people[0];
```

We can also use loops to iterate through the elements of an array of class objects. For example, the following code will print the names of all the people in the "people" array:

```
for (int i = 0; i < people.length; i++) {
   System.out.println(people[i].getName());
}
```

Arrays of class objects can also be used to store objects of different classes that are related through the inheritance. For example, if we have a class "Student" that extends the "Person" class, we could create an array of "Student" objects and store them in an array of "Person" objects:

```
Person[] students = new Student[5];
students[0] = new Student("John Smith", "Computer Science");
students[1] = new Student("Jane Doe", "Physics");
```

It is possible because a "Student" object is also a "Person" object, and the "Student" class inherits all the properties and methods of the "Person" class.

Arrays of class objects also allow us to use the functionality of the class they are holding. For example, if we have a class "Car" which has the method "drive()", we can use this method on all the elements of the array of cars.

```
Car[] cars = new Car[5];
cars[0] = new Car("Honda Civic");
cars[1] = new Car("Toyota Camry");
for (int i = 0; i < cars.length; i++) {
   cars[i].drive();
}
```

Document Object Model

Document Object

Properties of document object

Methods of document object

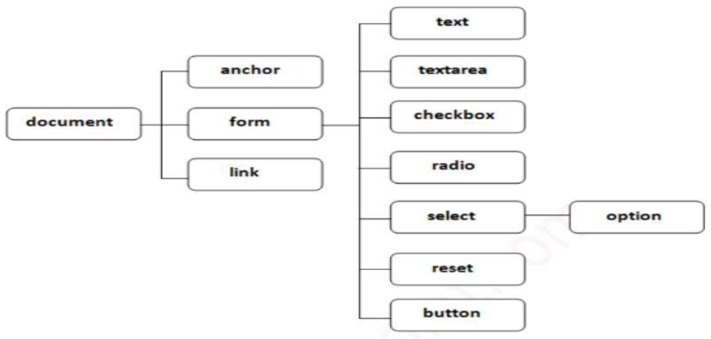
Example of document object

The document object represents the whole html document.

When html document is loaded in the browser, it becomes a document object. It is the root element that represents the html document. It has properties and methods. By the help of document object, we can add dynamic content to our web page.

As mentioned earlier, it is the object of window. So

Let's see the properties of document object that can be accessed and modified by the document object.



Methods of document object

We can access and change the contents of document by its methods.

The important methods of document object are as follows:

Method	Description
write("string")	writes the given string on the doucment.
writeln("string")	writes the given string on the doucment with newline character at the end.
getElementById()	returns the element having the given id value.
getElementsByName()	returns all the elements having the given name value.
getElementsByTagName()	returns all the elements having the given tag name.
getElementsByClassName()	returns all the elements having the given class name.

Accessing field value by document object

In this example, we are going to get the value of input text by user. Here, we are using document.form1.name.value to get the value of name field.

Here, document is the root element that represents the html document.

form1 is the name of the form.

name is the attribute name of the input text.

value is the property, that returns the value of the input text.

```
<script type="text/javascript">
function printvalue(){
var name=document.form1.name.value;
alert("Welcome: "+name);
}
</script>

<form name="form1">
Enter Name:<input type="text" name="name"/>
<input type="button" onclick="printvalue()" value="print name"/>
</form>
```

Asynchronous JavaScript

JavaScript is always synchronous and single-threaded that provides the event loops. The event loops enable us to queue up an activity. This activity will not happen until the loops become available after the program that queued the action has completed the execution. However, our program contains a large number of functionalities, which causes our code to be asynchronous. The Async/Await functionality is one of them. Async/Await is an extension of promises that we get as language support.

In this article, we are going to discuss the JavaScript Async/Await with some examples.

JavaScript Async

An async function is a function that is declared with the async keyword and allows the await keyword inside it. The async and await keywords allow asynchronous, promise-based behavior to be written more easily and avoid configured promise chains. The async keyword may be used with any of the methods for creating a function.

Syntax:

The syntax of JavaScript may be defined as:

```
Async function myfirstfunction() {
return "Hello World"
}
```

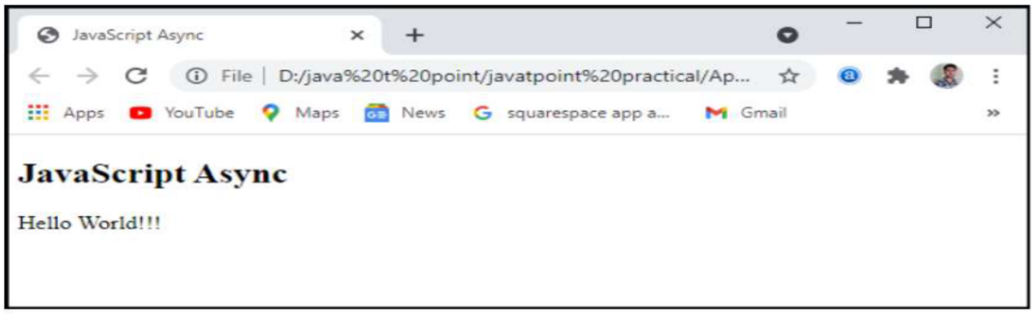
It is the same as:

```
async function myfirstfunction() {
return Promise.resolve("Hello World");
}
```

Example:
Let's take an example to understand how we can use a JavaScript Async in the program.

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8">
<title>JavaScript Async</title>
</head>
<body>
   <h2>JavaScript Async</h2>
   <p id="main"></p>
<script>
function myDisplayer(some) {
  document.getElementById("main").innerHTML = some;
}
```

```
async function myfirstFunction() {
    return "Hello World!!!";
    }
myfirstFunction().then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
</script>
</body>
</html>
```



JavaScript Async

Hello World!!!

JavaScript Await

JavaScript Await function is used to wait for the promise. It could only be used inside the async block. It instructs the code to wait until the promise returns a response. It only delays the async block. Await is a simple command that instructs JavaScript to wait for an asynchronous action to complete before continuing with the feature. It's similar to a "pause until done" keyword. The await keyword is used to retrieve a value from a function where we will usually be used the then() function. Instead of calling after the asynchronous function, we'd use await to allocate a variable to the result and then use the result in the code as we will in the synchronous code.

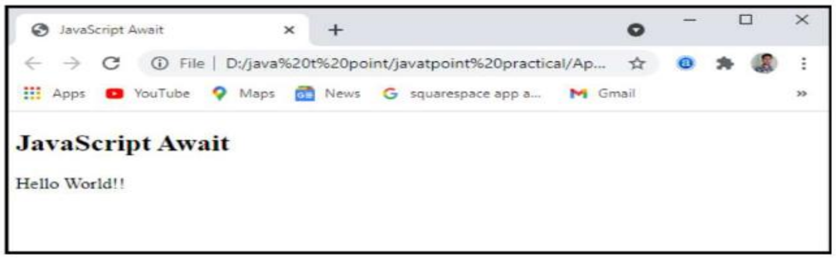
Syntax:

The syntax of JavaScript Await function may be defined as:

```
// Await function works only inside the async function
let value = await promise;
```

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8">
<title>JavaScript Await</title>
</head>
<body>
   <h2>JavaScript Await</h2>
   <p id="main"></p>
<script>
async function myDisplay() {
 let myPromise = new Promise(function(myResolve, myReject) {
   myResolve("Hello World!!");
 });
  document.getElementById("main").innerHTML = await myPromise;
}
myDisplay();
</script>
</body>
</html>
```

Output: After executing this code, we will get the output as shown below in the screenshot:



JavaScript Await

Hello World!!

Error Handling

It is very easy to handle errors in async functions. Promises have a catch() method for dealing with rejected promises, and because the async functions only return a promise, we may call the function and add a method to the end. We should use the promise's capture in the same way as we would any other catch. And all are easy to grasp. Remember that a then callback will fail. It can generate an error (with an explicit throw or by trying to access a property of a null variable). These crashes would also be caught by the grab process. Remind yourself that the promise's capture approach is similar to a standard catch.

Syntax:

The syntax of error handling may be defined as:

```
asyncFunc().catch(err =>
{
Console.error(err)
// catch error and do something
});
```

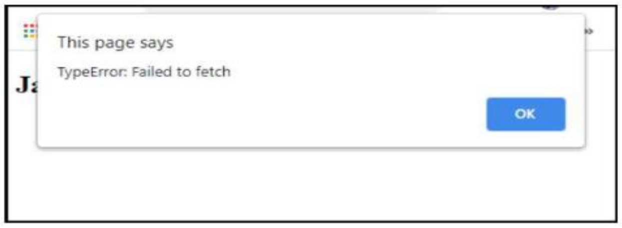
But there is another option: the all-powerful try/catch block. If we want to handle errors directly inside the async function, we may use try/catch in the same way we would in synchronous code.

Let's take an example to understand the error handling in the JavaScript Async and Await function.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
<title>JavaScript Await</title>
</head>
<body>
  <h2>JavaScript Await</h2>
<script>
async function f() {
```

```
try {
  let response = await fetch('http://no-url');
} catch(err) {
  alert(err); // TypeError: failed to fetch
}
}
f();
</script>
</body>
</html>
```

Output: After executing this code, we will get the output as shown below in the screenshot.



```
This page says
TypeError: Failed to fetch
```

JavaScript Fetch API

To get information from a server, utilize JavaScript's get() method. Any sort of API that returns data in JSON or XML can be requested. The URL to request is the only parameter needed for the Fetch () method, which also returns a promise.

Send a Request using Fetch

The javascript fetch api sends a request for the url. If the request sends, then it displays the response. If the request does not send, then it shows an error.

Syntax

The following syntax shows the fetch api method using JavaScript for sending the url request.

```
fetch('url')
.then(response  =>  { //handle request});
.then(error  = >  { //handle error});
```

We pass a configuration object and the endpoint's URL to the fetch method.
We submitted the given information in our HTML or Javascript components shown as an output.

Reading response using Fetch API
The javascript fetch api sends a request for the url and gets a response from the API. If the request sends successfully, then API gets a response and shows data.

Syntax
The following syntax shows the fetch api method using JavaScript for the reading response of the url.

```
fetch('url')
.then(response  =>  response.json())
.then(data  = >  console.log(data));
```

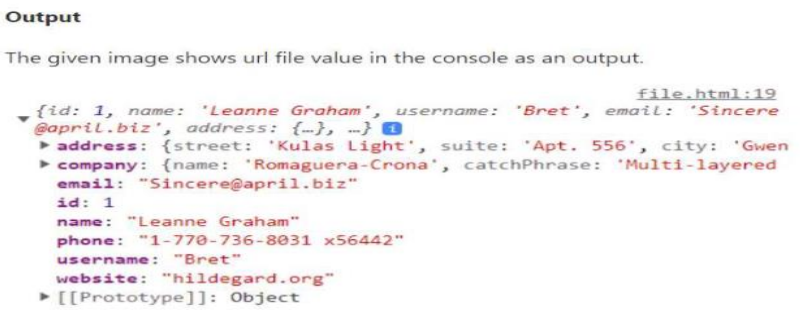
Parameters:
This technique accepts two arguments, but one parameter is essential for the method.
URL: This is the URL that the request should be sent to.
Options: The set of properties is an array. It is a choice-based parameter.

Examples
The following examples show the fetch api method in javascript.
Example 1
The example shows the basic fetch api method using a javascript function.

```
<!DOCTYPE html>
<html>
<head>
<title> Javascript Fetch API </title>
</head>
<body>
<p> Javascript Fetch API  </p>
<script>
```

```
//get the URL of the data
let url = 'https://jsonplaceholder.typicode.com/users/1';
console.log(url);
//Fetch API method for getting requests
let fetchfetch_Res = fetch(
url);
// fetch variable is the promise to resolve response using.then() method
// display data as an output
fetch_Res.then(resp =>
resp.json()).then(datas => {
console.log(datas)
})
```

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



Output

The given image shows url file value in the console as an output.

```
                                                    file.html:19
▼{id: 1, name: 'Leanne Graham', username: 'Bret', email: 'Sincere
 @april.biz', address: {…}, …}
  ▶ address: {street: 'Kulas Light', suite: 'Apt. 556', city: 'Gwen
  ▶ company: {name: 'Romaguera-Crona', catchPhrase: 'Multi-layered
    email: "Sincere@april.biz"
    id: 1
    name: "Leanne Graham"
    phone: "1-770-736-8031 x56442"
    username: "Bret"
    website: "hildegard.org"
  ▶ [[Prototype]]: Object
```

The example shows the basic fetch api method using the javascript function. The url files data displays in the console using the javascript function. We can get json data on the web page.

```
<!DOCTYPE html>
<html>
<head>
<title> Javascript Fetch API </title>
</head>
<body>
<p> Javascript Fetch API  </p>
<script>
//get the URL of the data
let url = 'https://jsonplaceholder.typicode.com/users/2';
console.log(url);
```

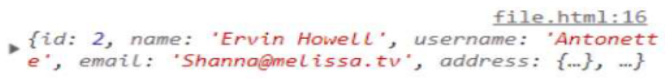
The example shows the basic fetch api method using the javascript function. The url files data displays in the console using the javascript function. We can get json data on the web page.

```
<!DOCTYPE html>
<html>
<head>
<title> Javascript Fetch API </title>
</head>
<body>
<p> Javascript Fetch API  </p>
<script>
//get the URL of the data
let url = 'https://jsonplaceholder.typicode.com/users/2';
console.log(url);
```

```
//Fetch API method for getting requests
// fetch variable is the promise to resolve response using.then() method
fetch(url)
.then(response => response.json())  // convert to json data
.then(json => console.log(json))   // display data as an output
</script>
</body>
</body>
</html>
```

Output

The given image shows url file information in the console as an output.



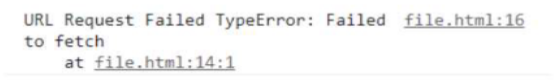
```
                                       file.html:16
▶ {id: 2, name: 'Ervin Howell', username: 'Antonett
  e', email: 'Shanna@melissa.tv', address: {…}, …}
```

The example shows the basic fetch api method for sending requests using a javascript function. The fetch method sends the wrong url then javascript sends an error message on the console.

```
<!DOCTYPE html>
<html>
<head>
<title> Javascript Fetch API </title>
</head>
<body>
<p> Javascript Fetch API  </p>
<script>
//get the URL of the data
let url = 'https://jsonpl1aceholder.typicode.com/users/12';
console.log(url);
```

```
//Fetch API method for getting requests
// fetch variable is the promise to resolve response using.then() method
fetch(url).then(response => response.json())  // convert to json data
.then(json => console.log(json))   // display data as an output
.catch(error => console.log('URL Request Failed', error)); // Display Catch
  errors
</script>
</body>
</body>
</html>
```

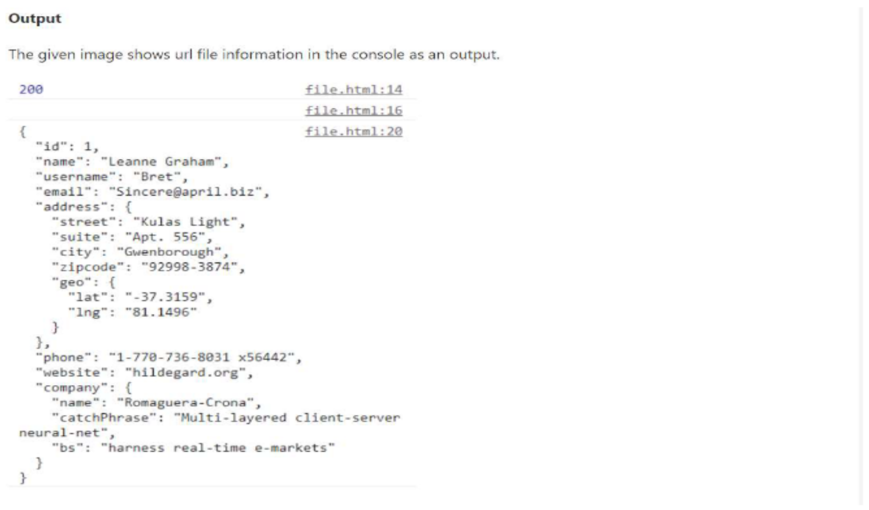
The given image shows error information in the console as an output.



```
URL Request Failed TypeError: Failed  file.html:16
to fetch
    at file.html:14:1
```

```
<!DOCTYPE html>
<html>
<head>
<title> Javascript Fetch API </title>
</head>
<body>
<p> Javascript Fetch API  </p>
<script>
//get the URL of the data
let url = 'https://jsonplaceholder.typicode.com/users/1';
async function fetchData() {
   let response_data = await fetch(url);
   // 200: display the response status of the URL
   console.log(response_data.status);
```

```
   // ok
      console.log(response_data.statusText);
      if (response_data.status === 200) {
         let data = await response_data.text();
         //display data of the url as output
         console.log(data); // 200
         }
   }
   fetchData();
   </script>
   </body>
   </html>
```



Output

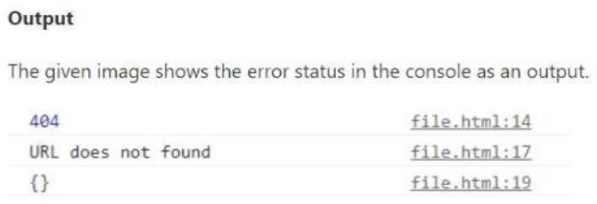
The given image shows url file information in the console as an output.

```
200                                          file.html:14
                                             file.html:16
{                                            file.html:20
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "Sincere@april.biz",
  "address": {
    "street": "Kulas Light",
    "suite": "Apt. 556",
    "city": "Gwenborough",
    "zipcode": "92998-3874",
    "geo": {
      "lat": "-37.3159",
      "lng": "81.1496"
    }
  },
  "phone": "1-770-736-8031 x56442",
  "website": "hildegard.org",
  "company": {
    "name": "Romaguera-Crona",
    "catchPhrase": "Multi-layered client-server
neural-net",
    "bs": "harness real-time e-markets"
  }
}
```

The example shows the basic fetch api method for URL requests using a javascript function. The fetch method sends the wrong url and status data to the console.

```
<!DOCTYPE html>
<html>
<head>
<title> Javascript Fetch API </title>
</head>
<body>
<p> Javascript Fetch API  </p>
<script>
//get the URL of the data
let url = 'https://jsonplaceholder.typicode.com/users1/1';
async function fetchData() {
  let response_data = await fetch(url);
  // 200: display the response status of the URL
  console.log(response_data.status);
  if (response_data.status == 404) {
```

```
 let data = await response_data.text();
      console.log("URL does not found");
      //display data of the url as output
      console.log(data); // 200
      }
}
fetchData();
</script>
</body>
</html>
```



Output

The given image shows the error status in the console as an output.

```
404                                 file.html:14
URL does not found                  file.html:17
{}                                  file.html:19
```

Introduction to ES6

ES6 or ECMAScript 6 is a scripting language specification which is standardized by ECMAScript International. This specification governs some languages such as JavaScript, ActionScript, and Jscript. ECMAScript is generally used for client-side scripting, and it is also used for writing server applications and services by using Node.js.

ES6 allows you to write the code in such a way that makes your code more modern and readable. By using ES6 features, we write less and do more, so the term 'Write less, do more' suits ES6.

This tutorial introduces you to the implementation of ES6 in JavaScript.

What is ES6?

ES6 is an acronym of ECMAScript 6 and also known as ECMAScript 2015.

ES6 or ECMAScript6 is a scripting language specification which is standardized by ECMAScript International. It is used by the applications to enable client-side scripting. This specification is affected by programming languages like Self, Perl, Python, Java, etc. This specification governs some languages such as JavaScript, ActionScript, and Jscript. ECMAScript is generally used for client-side scripting, and it is also used for writing server applications and services by using Node.js.

ES6 allows you to make the code more modern and readable. By using ES6 features, we write less and do more, so the term 'Write less, do more' suits ES6. ES6 introduces you many great features such as scope variable, arrow functions, template strings, class destructions, modules, etc.

ES6 was created to standardize JavaScript to help several independent implementations. Since the standard was first published, JavaScript has remained the well-known implementation of ECMAScript, comparison to other most famous implementations such as Jscript and ActionScript.

Exception Handling in JavaScript

An exception signifies the presence of an abnormal condition which requires special operable techniques. In programming terms, an exception is the anomalous code that breaks the normal flow of the code. Such exceptions require specialized programming constructs for its execution.

What is Exception Handling

In programming, exception handling is a process or method used for handling the abnormal statements in the code and executing them. It also enables to handle the flow control of the code/program. For handling the code, various handlers are used that process the exception and execute the code. For example, the Division of a non-zero value with zero will result into infinity always, and it is an exception. Thus, with the help of exception handling, it can be executed and handled.

In exception handling:

A throw statement is used to raise an exception. It means when an abnormal condition occurs, an exception is thrown using throw.

The thrown exception is handled by wrapping the code into the try…catch block. If an error is present, the catch block will execute, else only the try block statements will get executed.

Thus, in a programming language, there can be different types of errors which may disturb the proper execution of the program.

Types of Errors

While coding, there can be three types of errors in the code:

Syntax Error: When a user makes a mistake in the pre-defined syntax of a programming language, a syntax error may appear.

Runtime Error: When an error occurs during the execution of the program, such an error is known as Runtime error. The codes which create runtime errors are known as Exceptions. Thus, exception handlers are used for handling runtime errors.

Logical Error: An error which occurs when there is any logical mistake in the program that may not produce the desired output, and may terminate abnormally. Such an error is known as Logical error.

Error Object

When a runtime error occurs, it creates and throws an Error object. Such an object can be used as a base for the user-defined exceptions too. An error object has two properties:

name: This is an object property that sets or returns an error name.

message: This property returns an error message in the string form.

Although Error is a generic constructor, there are following standard built-in error types or error constructors beside it:

EvalError: It creates an instance for the error that occurred in the eval(), which is a global function used for evaluating the js string code.

InternalError: It creates an instance when the js engine throws an internal error.

RangeError: It creates an instance for the error that occurs when a numeric variable or parameter is out of its valid range.

ReferenceError: It creates an instance for the error that occurs when an invalid reference is de-referenced.

SyntaxError: An instance is created for the syntax error that may occur while parsing the eval().

TypeError: When a variable is not a valid type, an instance is created for such an error.

URIError: An instance is created for the error that occurs when invalid parameters are passed in encodeURI() or decodeURI().

Exception Handling Statements

There are following statements that handle if any exception occurs:

throw statements

try…catch statements

try…catch…finally statements.