Let’s roll by 09:05 !!!

Javascript 1

Gopi Krishnan R

Topics to be Discussed

- What’s Javascript?
- How to use JS?
- Variable
- Identifiers
- Data Types
- Operators
- Function

Javascript

Javascript is a high level programming language. Together with HTML & CSS, JS is one of the three core technologies of the world wide web that is the internet.

Initially JS was implemented only on web browsers. Today it runs on the server, mobile, IOT devices, inside pdf readers etc

It is a very popular language now. Since most of the applications we use on a day to day basis run on the browsers now, Js is one language you cannot avoid in your journey to become a web developer.

Although there are similarities between JavaScript and Java, including language name, syntax, and respective standard libraries, the two languages are distinct and differ greatly in design.

JS Inclusion

- Internal JS
- External JS

Variable and Scope

- Scope
  - Global scope
  - Block scope or local scope
- Variable
  - var
  - let
  - const

Identifiers

An identifier is simply a name. In JavaScript, identifiers are used to name variables and functions. A JavaScript identifier must begin with a letter, an underscore (_), or a dollar sign ($).

Data Types

The latest ECMAScript standard defines overall nine types:

- Six **Data Types** that are primitives, checked by typeof operator:
  - Number : typeof instance === "number"
  - String : typeof instance === "string"
  - Boolean : typeof instance === "boolean"
  - BigInt : typeof instance === "bigint"
  - Symbol : typeof instance === "symbol"
  - undefined : typeof instance === "undefined"
- Structural Types:
  - Object : typeof instance === "object". Special non-data but Structural type for any constructed object instance also used as data structures: new Object, new Array, new Map, new Set, new WeakMap, new WeakSet, new Date and almost everything made with new keyword;
  - Function : a non-data structure, though it also answers for typeof operator: typeof instance === "function". This is merely a special shorthand for Functions, though every Function constructor is derived from Object constructor.
- Structural Root Primitive:
  - null : typeof instance === "object". Special primitive type having additional usage for its value: if object is not inherited, then null is shown;

Operators

Arithmetic Operators

Javascript supports all the basic arithmetic operations - * (multiplication), / (division), % (modulo: remainder after division), + (addition), and - (subtraction).

Unary Arithmetic Operators

Unary operators modify the value of a single operand to produce a new value.

Increment (++) - The ++ operator increments (i.e., adds 1 to) its single operand, which must be an lvalue (a variable, an element of an array, or a property of an object). The operator converts its operand to a number, adds 1 to that number, and assigns the incremented value back into the variable, element, or property.

Decrement (--) - The -- operator expects an lvalue operand. It converts the value of the operand to a number, subtracts 1, and assigns the decremented value back to the operand.

Relational Operations

Relational expressions always evaluate to a boolean value, and that value is often used to control the flow of program execution in if , while , and for statements.

The == and === operators check whether two values are the same, using two different definitions of sameness.

Comparison Operations

The comparison operators test the relative order (numerical or alphabetics) of their two operands:

Less than (<) - The < operator evaluates to true if its first operand is less than its second operand; otherwise it evaluates to false.

Greater than (>) - The > operator evaluates to true if its first operand is greater than its second operand; otherwise it evaluates to false.

Less than or equal (<=) - The <= operator evaluates to true if its first operand is less than or equal to its second operand; otherwise it evaluates to false.

Greater than or equal (>=) - The >= operator evaluates to true if its first operand is greater than or equal to its second operand; otherwise it evaluates to false.

Logical Operations

The logical operators && , || , and ! perform Boolean algebra and are often used in conjunction with the relational operators to combine two relational expressions into one more complex expression.

instanceof Operator

The instanceof operator expects a left-side operand that is an object and a right-side operand that identifies a class of objects. The operator evaluates to true if the left-side object is an instance of the right-side class and evaluates to false otherwise.

typeof Operator

The typeof operator returns the type of the argument. It’s useful when we want to process values of different types differently or just want to do a quick check.

It supports two forms of syntax:

1. As an operator: typeof x.
2. As a function: typeof(x).

Function

A JavaScript function is defined with the function keyword, followed by a name, followed by parentheses ().

Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas:
(*parameter1, parameter2, ...*)

The code to be executed, by the function, is placed inside curly brackets: {}

- Function **parameters** are listed inside the parentheses () in the function definition.
- Function **arguments** are the **values** received by the function when it is invoked.
- Inside the function, the arguments (the parameters) behave as local variables.

Let’s roll by 09:05 !!!

Javascript 2

Gopi Krishnan R

Topics to be Discussed

- JS Conditional Statement
- Loops
- Javascript Jump Statement
- Function

JS Conditional Statement

- if else
- switch

JS Loops

- for loop
- while loop
- do/while loop
- for/in loop

JS Jump Statement

- break
- continue
- return

Function

A JavaScript function is defined with the function keyword, followed by a name, followed by parentheses ().

Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas:
(*parameter1, parameter2, ...*)

The code to be executed, by the function, is placed inside curly brackets: {}

- Function **parameters** are listed inside the parentheses () in the function definition.
- Function **arguments** are the **values** received by the function when it is invoked.
- Inside the function, the arguments (the parameters) behave as local variables.

Let’s roll by 09:05 !!!

Javascript 3

Gopi Krishnan R

Topics to be Discussed

- Regular Function
- Callback Function
- Anonymous Function

Function

A JavaScript function is defined with the function keyword, followed by a name, followed by parentheses ().

Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas:
(*parameter1, parameter2, ...*)

The code to be executed, by the function, is placed inside curly brackets: {}

- Function **parameters** are listed inside the parentheses () in the function definition.
- Function **arguments** are the **values** received by the function when it is invoked.
- Inside the function, the arguments (the parameters) behave as local variables.

Callback Function

A callback function is a function that is passed as an argument to another function, to be “called back” at a later time. A function that accepts other functions as arguments is called a higher-order function, which contains the logic for when the callback function gets executed. It’s the combination of these two that allow us to extend our functionality.

- Synchronous
- Asynchronous

```
function createQuote(quote, callback){
  var myQuote = "Like I always say, " + quote;
  callback(myQuote); // 2
}

function logQuote(quote){
  console.log(quote);
}

createQuote("eat your vegetables!", logQuote); // 1

// Result in console:
// Like I always say, eat your vegetables!
```

Synchronous

Asynchronous

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => response.json())
  .then(json => console.log(json))
```

Anonymous Function

The meaning of the word 'anonymous' defines something that is unknown or has no identity. In JavaScript, an anonymous function is that type of function that has no name or we can say which is without any name. When we create an anonymous function, it is declared without any identifier. It is the difference between a normal function and an anonymous function. Not particularly in JavaScript but also in other various programming languages also. The role of an anonymous function is the same.

```
let x = function () {
    console.log('It is an anonymous function');
};
x();
```

Another benefit of anonymous functions are the are **not preserved in the memory like named functions**. As soon as the use/calling of anonymous functions is done it is destroyed from the memory. However, this is not the case with named functions. **If a named function is declared, it remains in the memory even if it is not being used**.

Let’s roll by 09:05 !!!

Javascript 4

Gopi Krishnan R

Topics to be Discussed

- Arrays
- Ways to Declare Array in JS
- JS Array inbuilt function

Array

An array is an ordered collection of values. Each value is called an element, and each element has a numeric position in the array, known as its index. JavaScript arrays are untyped: an array element may be of any type, and different elements of the same array may be of different types.

Create and Use Array

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]; //most used method
var arr = new Array(); //another less used method
```

```
var fruits = ["Apple", "Orange", "Plum"];
alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum
```

We can replace an element:

```
fruits[2] = 'Pear'; // now ["Apple", "Orange", "Pear"]
```

...Or add a new one to the array:

```
fruits[3] = 'Lemon'; // now ["Apple", "Orange", "Pear", "Lemon"]
```

An array can store elements of any type.

```
// mix of values
var arr = [ 'Apple', { name: 'John' }, true, function() { alert('hello'); } ];

// get the object at index 1 and then show its name
alert( arr[1].name ); // John

// get the function at index 3 and run it
arr[3](); // hello
```

Methods in Javascript

- push() and pop()
  - The push() method appends one or more new elements to the end of an array and returns the new length of the array. The pop() method does the reverse: it deletes the last element of an array, decrements the array length, and returns the value that it removed.
- unshift() and shift()
  - The unshift() and shift() methods behave much like push() and pop() , except that they insert and remove elements from the beginning of an array rather than from the end.
- toString()
  - An array, like any JavaScript object, has a toString() method. For an array, this method converts each of its elements to a string (calling the toString() methods of its elements, if necessary) and outputs a comma-separated list of those strings.

.join()

The Array.join() method converts all the elements of an array to strings and concatenates them, returning the resulting string.

```
var a = [1, 2, 3];

a.join();

a.join(" ");
```

.reverse()

Reverses an array. It also returns the array arr after the reversal. The reversal is done in-place.

```
let arr = [1, 2, 3, 4, 5];

arr.reverse();

alert( arr ); // 5,4,3,2,1
```

.concat()

The Array.concat() method creates and returns a new array that contains the elements of the original array on which concat() was invoked, followed by each of the arguments to concat().

```
var a = [1,2,3];

a.concat(4, 5)

a.concat([4,5]);
```

.slice()

The Array.slice() method returns a slice, or subarray, of the specified array. Its two arguments specify the start and end of the slice to be returned.

```
var a = [1,2,3,4,5];

a.slice(0,3); // Returns [1,2,3]
```

.splice()

The Array.splice() method is a general-purpose method for inserting or removing elements from an array. Unlike slice() and concat() , splice() modifies the array on which it is invoked. Note that splice() and slice() have very similar names but per- form substantially different operations.

splice() can delete elements from an array, insert new elements into an array, or perform both operations at the same time.

```
var a = [1,2,3,4,5,6,7,8];

a.splice(4);                // Returns [5,6,7,8]; a is [1,2,3,4]

a.splice(1,2); // Returns [2,3]; a is [1,4]

a.splice(1,1); // Returns [4]; a is [1]
```

.forEach()

The arr.forEach method allows to run a function for every element of the array.

Syntax:

```
arr.forEach(function(item, index, array) {

  // ... do something with item

});
```

.map()

The arr.map method is one of the most useful and often used. It calls the function for each element of the array and returns the array of results.

Syntax:

```
let result = arr.map(function(item, index, array) {

  // returns the new value instead of item

});
```

.sort()

Array.sort() sorts the elements of an array in place and returns the sorted array. When sort() is called with no arguments, it sorts the array elements in alphabetical order.

If an array contains undefined elements, they are sorted to the end of the array. To sort an array into some order other than alphabetical, you must pass a comparison function as an argument to sort() . This function decides which of its two arguments should appear first in the sorted array. If the first argument should appear before the second, the comparison function should return a number less than zero. If the first argument should appear after the second in the sorted array, the function should return a number greater than zero. And if the two values are equivalent (i.e., if their order is irrelevant), the comparison function should return 0.

```
var numbers = [2, 5, 1, 3, 8, 9, 4, 2, 1];

numbers.sort( function(a, b) { return a - b; } )
```

filter()

The filter() method returns an array containing a subset of the elements of the array on which it is invoked. The function you pass to it should be predicate: a function that returns true or false . The predicate is invoked just as for forEach() and map() . If the return value is true , or a value that converts to true , then the element passed to the predicate is a member of the subset and is added to the array that will become the return value. Examples:

```
a = [5, 4, 3, 2, 1];

smallvalues = a.filter(function(x) { return x < 3 });          // [2, 1]

everyother = a.filter(function(x,i) { return i%2==0 });      // [5, 3, 1]
```

every() and some()

The every() and some() methods are array predicates: they apply a predicate function you specify to the elements of the array, and then return true or false.

The every() method is like the mathematical "for all" quantifier ∀: it returns true if and only if your predicate function returns true for all elements in the array:

```
a = [1,2,3,4,5];

a.every(function(x) { return x < 10; });          // => true: all values < 10.

a.every(function(x) { return x % 2 === 0; });     // => false: not all values even.
```

The some() method is like the mathematical "there exists" quantifier ∃: it returns true if there exists at least one element in the array for which the predicate returns true, and returns false if and only if the predicate returns false for all elements of the array:

```
a = [1,2,3,4,5];

a.some(function(x) { return x%2===0; });    // => true a has some even numbers.
```

reduce(), reduceRight()

The reduce() and reduceRight() methods combine the elements of an array, using the function you specify, to produce a single value. This is a common operation in functional programming and also goes by the names "inject" and "fold".

```
a = [1,2,3,4,5]''

sum = a.reduce(function(x,y) { return x+y }, 0);          // Sum of values

product = a.reduce(function(x,y) { return x*y }, 1);      // Product of values

max = a.reduce(function(x,y) { return (x>y)?x:y; });      // Largest value
```

indexOf() and lastIndexOf()

indexOf() and lastIndexOf() search an array for an element with a specified value, and return the index of the first such element found, or –1 if none is found. indexOf() searches the array from beginning to end, and lastIndexOf() searches from end to beginning.

```
a = [0,1,2,1,0];

a.indexOf(1);                    // => 1: a[1] is 1

a.lastIndexOf(1);          // => 3: a[3] is 1

a.indexOf(3);                    // => -1: no element has value 3
```

Let’s roll by 09:05 !!!

Javascript 5

Gopi Krishnan R

Topics to be Discussed

- JS Array inbuilt function
- Objects

.forEach()

The arr.forEach method allows to run a function for every element of the array.

Syntax:

```
arr.forEach(function(item, index, array) {

  // ... do something with item

});
```

.map()

The arr.map method is one of the most useful and often used. It calls the function for each element of the array and returns the array of results.

Syntax:

```
let result = arr.map(function(item, index, array) {

  // returns the new value instead of item

});
```

.sort()

Array.sort() sorts the elements of an array in place and returns the sorted array. When sort() is called with no arguments, it sorts the array elements in alphabetical order.

If an array contains undefined elements, they are sorted to the end of the array. To sort an array into some order other than alphabetical, you must pass a comparison function as an argument to sort() . This function decides which of its two arguments should appear first in the sorted array. If the first argument should appear before the second, the comparison function should return a number less than zero. If the first argument should appear after the second in the sorted array, the function should return a number greater than zero. And if the two values are equivalent (i.e., if their order is irrelevant), the comparison function should return 0.

```
var numbers = [2, 5, 1, 3, 8, 9, 4, 2, 1];

numbers.sort( function(a, b) { return a - b; } )
```

filter()

The filter() method returns an array containing a subset of the elements of the array on which it is invoked. The function you pass to it should be predicate: a function that returns true or false . The predicate is invoked just as for forEach() and map() . If the return value is true , or a value that converts to true , then the element passed to the predicate is a member of the subset and is added to the array that will become the return value. Examples:

```
a = [5, 4, 3, 2, 1];

smallvalues = a.filter(function(x) { return x < 3 });          // [2, 1]

everyother = a.filter(function(x,i) { return i%2==0 });      // [5, 3, 1]
```

every() and some()

The every() and some() methods are array predicates: they apply a predicate function you specify to the elements of the array, and then return true or false.

The every() method is like the mathematical "for all" quantifier ∀: it returns true if and only if your predicate function returns true for all elements in the array:

```
a = [1,2,3,4,5];

a.every(function(x) { return x < 10; });          // => true: all values < 10.

a.every(function(x) { return x % 2 === 0; });     // => false: not all values even.
```

The some() method is like the mathematical "there exists" quantifier ∃: it returns true if there exists at least one element in the array for which the predicate returns true, and returns false if and only if the predicate returns false for all elements of the array:

```
a = [1,2,3,4,5];

a.some(function(x) { return x%2===0; });    // => true a has some even numbers.
```

reduce(), reduceRight()

The reduce() and reduceRight() methods combine the elements of an array, using the function you specify, to produce a single value. This is a common operation in functional programming and also goes by the names "inject" and "fold".

```
a = [1,2,3,4,5]''

sum = a.reduce(function(x,y) { return x+y }, 0);          // Sum of values

product = a.reduce(function(x,y) { return x*y }, 1);      // Product of values

max = a.reduce(function(x,y) { return (x>y)?x:y; });      // Largest value
```

indexOf() and lastIndexOf()

indexOf() and lastIndexOf() search an array for an element with a specified value, and return the index of the first such element found, or –1 if none is found. indexOf() searches the array from beginning to end, and lastIndexOf() searches from end to beginning.

```
a = [0,1,2,1,0];

a.indexOf(1);                    // => 1: a[1] is 1

a.lastIndexOf(1);          // => 3: a[3] is 1

a.indexOf(3);                    // => -1: no element has value 3
```

Objects

JavaScript's fundamental datatype is the object. An object is a composite value: it aggregates multiple values (primitive values or other objects) and allows you to store and retrieve those values by name.

An object is an unordered collection of properties, each of which has a name and a value. Property names are strings, so we can say that objects map strings to values.

Object Creation

Object Literals

```
var person = {
        name: "Roger",
        location: "Switzerland"
}
```

Creating Objects with new

The new operator creates and initializes a new object. The new keyword must be followed by a function invocation. A function used in this way is called a constructor and serves to initialize a newly created object.

```
var o = new Object();
var a = new Array();
var d = new Date();
```

Object.create()

ECMAScript 5 defines a method, Object.create(), that creates a new object, using its first argument as the prototype of that object. Object.create() also takes an optional second argument that describes the properties of the new object.

Setting and Getting Properties

To obtain the value of a property, use the dot (.) or square bracket ([]) operators.

Multi words properties

```
let user = {};


// set
user["likes birds"] = true;
// get
alert(user["likes birds"]); // true
```

Deleting Properties

The delete operator removes a property from an object.

```
// delete
delete user["likes birds"];
```

Serializing Objects

Object serialization is the process of converting an object's state to a string from which it can later be restored. ECMAScript 5 provides native functions JSON.stringify() and JSON.parse() to serialize and restore JavaScript objects.

Constructor function

Constructor functions technically are regular functions. There are two conventions though:

1. They are named with capital letter first.
2. They should be executed only with "new" operator.

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}

let user = new User("Jack");

alert(user.name); // Jack
alert(user.isAdmin); // false
```

Methods in constructor

Using constructor functions to create objects gives a great deal of flexibility. The constructor function may have parameters that define how to construct the object, and what to put in it.

Of course, we can add to this not only properties, but methods as well.

```
function User(name) {
  this.name = name;

  this.sayHi = function() {
    alert( "My name is: " + this.name );
  };
}

let john = new User("John");
```

Let’s roll by 09:05 !!!

Javascript 6

Gopi Krishnan R

Topics to be Discussed

- JS Array inbuilt function
- DOM
- DOM Manipulation
- Events
- Event Bubbling

every() and some()

The every() and some() methods are array predicates: they apply a predicate function you specify to the elements of the array, and then return true or false.

The every() method is like the mathematical "for all" quantifier ∀: it returns true if and only if your predicate function returns true for all elements in the array:

```
a = [1,2,3,4,5];

a.every(function(x) { return x < 10; });          // => true: all values < 10.

a.every(function(x) { return x % 2 === 0; });     // => false: not all values even.
```

The some() method is like the mathematical "there exists" quantifier ∃: it returns true if there exists at least one element in the array for which the predicate returns true, and returns false if and only if the predicate returns false for all elements of the array:

```
a = [1,2,3,4,5];

a.some(function(x) { return x%2===0; });    // => true a has some even numbers.
```

reduce(), reduceRight()

The reduce() and reduceRight() methods combine the elements of an array, using the function you specify, to produce a single value. This is a common operation in functional programming and also goes by the names "inject" and "fold".

```
a = [1,2,3,4,5]''

sum = a.reduce(function(x,y) { return x+y }, 0);          // Sum of values

product = a.reduce(function(x,y) { return x*y }, 1);      // Product of values

max = a.reduce(function(x,y) { return (x>y)?x:y; });      // Largest value
```

DOM

Document Object Model, or DOM for short, represents all page content as objects that can be modified.

The document object is the main “entry point” to the page. We can change or create anything on the page using it.

- getElementById()
- getElementByClassName()
- getElementByTagName()
- getElementByName()

Events

- Mouse Events
  - click – when the mouse clicks on an element (touchscreen devices generate it on a tap).
  - contextmenu – when the mouse right-clicks on an element.
  - mouseover / mouseout – when the mouse cursor comes over / leaves an element.
  - mousedown / mouseup – when the mouse button is pressed / released over an element.
  - mousemove – when the mouse is moved.
- Keyboard events:
  - keydown and keyup – when a keyboard key is pressed and released.
- Form element events:
  - submit – when the visitor submits a <form>. focus – when the visitor focuses on an element, e.g. on an <input>.
- Document events:
  - DOMContentLoaded – when the HTML is loaded and processed, DOM is fully built.

addeventListener

The fundamental problem of the aforementioned ways to assign handlers – we can’t assign multiple handlers to one event.

Let’s say, one part of our code wants to highlight a button on click, and another one wants to show a message on the same click.

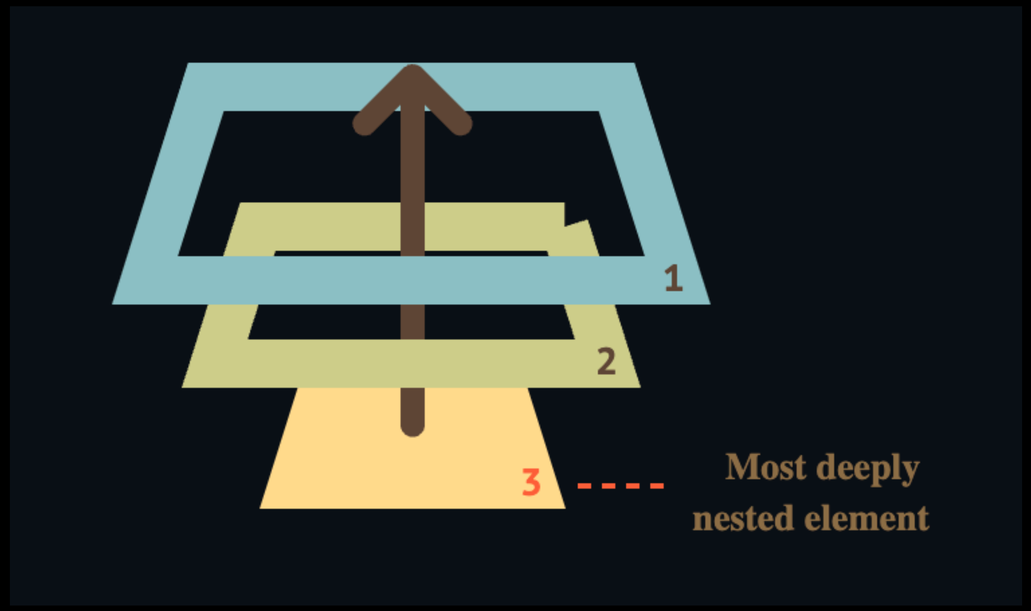
We’d like to assign two event handlers for that. But a new DOM property will overwrite the existing one.

```
addEventListener(type, listener);
addEventListener(type, listener, options);
addEventListener(type, listener, useCapture);
```

Event Bubbling

The bubbling principle is simple.

When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.



Stopping bubbling

A bubbling event goes from the target element straight up. Normally it goes upwards till <html>, and then to document object, and some events even reach window, calling all handlers on the path.

But any handler may decide that the event has been fully processed and stop the bubbling.

The method for it is event.stopPropagation().

For instance, here body.onclick doesn’t work if you click on <button>:

```
<body onclick="alert(`the bubbling doesn't reach here`)">

  <button onclick="event.stopPropagation()">Click me</button>

</body>
```

event.stopImmediatePropagation()

If an element has multiple event handlers on a single event, then even if one of them stops the bubbling, the other ones still execute.

In other words, event.stopPropagation() stops the move upwards, but on the current element all other handlers will run.

To stop the bubbling and prevent handlers on the current element from running, there’s a method event.stopImmediatePropagation(). After it no other handlers execute.

Browser Default Action

Many events automatically lead to certain actions performed by the browser.

For instance:

- A click on a link – initiates navigation to its URL.
- A click on a form submit button – initiates its submission to the server.
- Pressing a mouse button over a text and moving it – selects the text.

If we handle an event in JavaScript, we may not want the corresponding browser action to happen, and want to implement another behavior instead.

Preventing browser actions

There are two ways to tell the browser we don’t want it to act:

- The main way is to use the event object. There’s a method event.preventDefault().
- If the handler is assigned using on<event> (not by addEventListener), then returning false also works the same.

Let’s roll by 09:05 !!!

Javascript 7

Gopi Krishnan R

Topics to be Discussed

- Preventing Browser Default Action
- DOM Manipulation (Cont.)
- TODO App
- HTML5 APIs

Browser Default Action

Many events automatically lead to certain actions performed by the browser.

For instance:

- A click on a link – initiates navigation to its URL.
- A click on a form submit button – initiates its submission to the server.
- Pressing a mouse button over a text and moving it – selects the text.

If we handle an event in JavaScript, we may not want the corresponding browser action to happen, and want to implement another behavior instead.

Preventing browser actions

There are two ways to tell the browser we don’t want it to act:

- The main way is to use the event object. There’s a method event.preventDefault().
- If the handler is assigned using on<event> (not by addEventListener), then returning false also works the same.

DOM Manipulation (Contd)

Creating an Element

To create DOM nodes, there are two methods:

document.createElement(tag)

Creates a new element node with the given tag:

var div = document.createElement('div');

document.createTextNode(text) Creates a new text node with the given text:

var textNode = document.createTextNode('Here I am');

Creating the message

```
// 1. Create <div> element

let div = document.createElement('div');


// 2. Set its class to "alert"

div.className = "alert";


// 3. Fill it with the content

div.innerHTML = "<strong>Hi there!</strong> You've read an important message.";
```

We’ve created the element. But as of now it’s only in a variable named div, not in the page yet. So we can’t see it.

Insertion methods

To make the div show up, we need to insert it somewhere into document. For instance, into <body> element, referenced by document.body.

There’s a special method append for that: document.body.append(div).

```
<script>
  let div = document.createElement('div');
  div.className = "alert";
  div.innerHTML = "<strong>Hi there!</strong> You've read an important message.";

  document.body.append(div);
</script>
```

- `node.append(...nodes or strings)` – append nodes or strings at the end of node,
- `node.prepend(...nodes or strings)` – insert nodes or strings at the beginning of node,
- `node.before(...nodes or strings)` –- insert nodes or strings before node,
- `node.after(...nodes or strings)` –- insert nodes or strings after node,
- `node.replaceWith(...nodes or strings)` –- replaces node with the given nodes or strings.

HTML5 APIs

API	What It Does
Ambient Light API	Provides information about the ambient light levels, as detected by a device's light sensor.
Battery Status API	Provides information about the battery status of the device.
Canvas 2D Context	Allows drawing and manipulation of graphics in a browser.
Clipboard API	Provides access to the operating system's copy, cut, and paste functionality.
Contacts	Allows access to a user's contacts repository in the web browser.
Drag and Drop	Supports dragging and dropping items within and between browser windows.
File API	Provides programs with secure access to the device's file system.
Forms	Gives programs access to the new data types defined in HTML5.
Fullscreen API	Controls the use of the user's full screen for web pages, without the browser user interface.
Gamepad API	Supports input from USB gamepad controllers.
Geolocation	Provides web applications with access to geographical location data about the user's device.
getUserMedia/Stream API	Provides access to external device data (such as webcam video).

History API	Allows programs to manipulate the browser history.
HTML Microdata	Provides a way to annotate content with computer-readable labels.
Indexed database	Creates a simple client-side database system in the web browser.
Internationalization API	Provides access to locale-sensitive formatting and string comparison.
Offline apps	Allows programmers to make web apps available in offline mode.
Proximity API	Provides information about the distance between a device and an object.
Screen Orientation	Reads the screen orientation state (portrait or landscape) and gives programmers the ability to know when it change and to lock it in place.
Selection	Supports selecting elements in JavaScript using CSS-style selectors.
Server-sent events	Allows the server to push data to the browser without the browser needing to request it.
User Timing API	Gives programmers access to high-precision timestamps to measure the performance of applications.
Vibration API	Allows access to the vibration functionality of the device.
Web Audio API	API for processing and synthesizing audio.
Web Messaging	Allows browser windows to communicate with each other across different origins.
Web Speech API	Provides speech input and text-to-speech output features.
Web storage	Allows the storage of key-value pairs in the browser.
Web sockets	Opens an interactive communication session between the browser and server.
Web Workers	Allows JavaScript to execute scripts in the background.
XMLHttpRequest2	Improves XMLHttpRequest to eliminate the need to work around the same-origin policy errors and to make XMLHttpRequest work with new features of HTML5.

Let’s roll by 09:05 !!!

Javascript Intermediate 1

Gopi Krishnan R

Topics to be Discussed

- Basics of CallStack / Execution Context
- Javascript CallStack / Execution Context
- Hoisting
- Let & Const Temporal Dead Zone
- Closure
- Function borrowing
- Call, Apply and Bind.

Hoisting

Hoisting in JavaScript is a behavior in which a function or a variable can be used before declaration. For example,

```
// using test before declaring

console.log(test);   // undefined

var test;
```

The above program works and the output will be undefined. The above program behaves as

```
// using test before declaring

var test;

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

Since the variable test is only declared and has no value, undefined value is assigned to it.

- Variable Hoisting
- Function Hoisting

Closure

JavaScript uses lexical scoping. This means that functions are executed using the variable scope that was in effect when they were defined, not the variable scope that is in effect when they are invoked.

The combination of a function object and a scope (a set of variable bindings) in which the function’s variables are resolved is called a closure.

```
function makeCounter() {
        var counter = 0;
        function counterFunction() {
                return ++counter;
        }
        return counterFunction;
}

var myCounter = makeCounter();
console.log(myCounter()); // 1
console.log(myCounter()); // 2
```

Call, Apply and Bind

The difference between Call, Apply and Bind can be explained with below examples,

Call: The call() method invokes a function with a given this value and arguments provided one by one

```
var employee1 = {firstName: 'John', lastName: 'Rodson'};
var employee2 = {firstName: 'Jimmy', lastName: 'Baily'};

function invite(greeting1, greeting2) {
    console.log(greeting1 + ' ' + this.firstName + ' ' + this.lastName+ ', '+ greeting2);
}

invite.call(employee1, 'Hello', 'How are you?'); // Hello John Rodson, How are you?
invite.call(employee2, 'Hello', 'How are you?'); // Hello Jimmy Baily, How are you?
```

Apply: Invokes the function with a given this value and allows you to pass in arguments as an array

```
var employee1 = {firstName: 'John', lastName: 'Rodson'};
var employee2 = {firstName: 'Jimmy', lastName: 'Baily'};

function invite(greeting1, greeting2) {
    console.log(greeting1 + ' ' + this.firstName + ' ' + this.lastName+ ', '+ greeting2);
}

invite.apply(employee1, ['Hello', 'How are you?']); // Hello John Rodson, How are you?
invite.apply(employee2, ['Hello', 'How are you?']); // Hello Jimmy Baily, How are you?
```

bind: returns a new function, allowing you to pass any number of arguments

```
var employee1 = {firstName: 'John', lastName: 'Rodson'};
var employee2 = {firstName: 'Jimmy', lastName: 'Baily'};

function invite(greeting1, greeting2) {
    console.log(greeting1 + ' ' + this.firstName + ' ' + this.lastName+ ', '+
greeting2);
}

var inviteEmployee1 = invite.bind(employee1);
var inviteEmployee2 = invite.bind(employee2);
inviteEmployee1('Hello', 'How are you?'); // Hello John Rodson, How are you?
inviteEmployee2('Hello', 'How are you?'); // Hello Jimmy Baily, How are you?
```

Let’s roll by 09:05 !!!

Javascript Intermediate 2

Gopi Krishnan R

Topics to be Discussed

- Function borrowing
- Call, Apply and Bind
- IIFE
- Date and Time
- Async operations
  - setTimeout()
  - setInterval()

Call, Apply and Bind

The difference between Call, Apply and Bind can be explained with below examples,

Call: The call() method invokes a function with a given this value and arguments provided one by one

```
var employee1 = {firstName: 'John', lastName: 'Rodson'};
var employee2 = {firstName: 'Jimmy', lastName: 'Baily'};

function invite(greeting1, greeting2) {
    console.log(greeting1 + ' ' + this.firstName + ' ' + this.lastName+ ', '+ greeting2);
}

invite.call(employee1, 'Hello', 'How are you?'); // Hello John Rodson, How are you?
invite.call(employee2, 'Hello', 'How are you?'); // Hello Jimmy Baily, How are you?
```

Apply: Invokes the function with a given this value and allows you to pass in arguments as an array

```
var employee1 = {firstName: 'John', lastName: 'Rodson'};
var employee2 = {firstName: 'Jimmy', lastName: 'Baily'};

function invite(greeting1, greeting2) {
    console.log(greeting1 + ' ' + this.firstName + ' ' + this.lastName+ ', '+ greeting2);
}

invite.apply(employee1, ['Hello', 'How are you?']); // Hello John Rodson, How are you?
invite.apply(employee2, ['Hello', 'How are you?']); // Hello Jimmy Baily, How are you?
```

bind: returns a new function, allowing you to pass any number of arguments

```
var employee1 = {firstName: 'John', lastName: 'Rodson'};
var employee2 = {firstName: 'Jimmy', lastName: 'Baily'};

function invite(greeting1, greeting2) {
    console.log(greeting1 + ' ' + this.firstName + ' ' + this.lastName+ ', '+
greeting2);
}

var inviteEmployee1 = invite.bind(employee1);
var inviteEmployee2 = invite.bind(employee2);
inviteEmployee1('Hello', 'How are you?'); // Hello John Rodson, How are you?
inviteEmployee2('Hello', 'How are you?'); // Hello Jimmy Baily, How are you?
```

IIFE

An IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined.

```
(function () {

    statements

})();
```

It is a design pattern which is also known as a Self-Executing Anonymous Function and contains two major parts:

1. The first is the anonymous function with lexical scope enclosed within the Grouping Operator (). This prevents accessing variables within the IIFE idiom as well as polluting the global scope.

2. The second part creates the immediately invoked function expression () through which the JavaScript engine will directly interpret the function.

The function becomes a function expression which is immediately executed. The variable within the expression can not be accessed from outside it.

```
(function () {
    var aName = "Barry";
})();
// Variable aName is not accessible from the outside scope
aName // throws "Uncaught ReferenceError: aName is not defined"
```

Assigning the IIFE to a variable stores the function's return value, not the function definition itself.

```
var result = (function () {
    var name = "Barry";
    return name;
})();
// Immediately creates the output:
result; // “Barry"
```

By placing functions and variables inside an immediately invoked function expression, you can avoid polluting them to the global object:

```
(function() {
    var counter = 0;

    function add(a, b) {
        return a + b;
    }

    console.log(add(10,20)); // 30
})();
```

Date and Time

Date and Time is represented in JavaScript using a built in Date object. There are a lot of functions on the date object, we will try to cover only the basics.

Creation

To create a new Date object call new Date() with one of the following arguments:

new Date()

```
var now = new Date();
alert( now ); // shows current date/time
```

Methods in Date object

- Date.now()
- Date.UTC()
- Date.getDate()
- Date.getDay()
- Date.getFullYear()
- Date.getHours()
- Date.getMilliseconds()
- Date.getMinutes()
- Date.getMonth()
- Date.getSeconds()

setTimeout()

setTimeout allows us to run a function once after the interval of time.

```
var timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)
```

Parameters:

func|code Function or a string of code to execute. Usually, that’s a function. For historical reasons, a string of code can be passed, but that’s not recommended.

delay The delay before run, in milliseconds (1000 ms = 1 second), by default 0.

arg1, arg2… Arguments for the function (not supported in IE9-).

Canceling with clearTimeout

A call to setTimeout returns a “timer identifier” timerId that we can use to cancel the execution.

The syntax to cancel:

```
var timerId = setTimeout(...);

clearTimeout(timerId);
```

In the code below, we schedule the function and then cancel it (changed our mind). As a result, nothing happens:

```
var timerId = setTimeout(function () { alert("never happens") }, 1000);

alert(timerId); // timer identifier: a number in browser, object in Node.js



clearTimeout(timerId);

alert(timerId); // same identifier (doesn't become null after canceling)
```

setInterval()

The setInterval method has the same syntax as setTimeout,
however setInterval runs the function not only once, but regularly after the given interval of time.

To stop further calls, we should call clearInterval(timerId).

```
// repeat with the interval of 2 seconds

let timerId = setInterval(() => alert('tick'), 2000);



// after 5 seconds stop

setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Let’s roll by 09:05 !!!

Javascript Intermediate 3

Gopi Krishnan R

Topics to be Discussed

- ES6
- Let and Const
- Template literals
- New string methods

ES6

he JavaScript programming language is standardized by ECMA (a standards body like W3C) under the name ECMAScript. Browsers and Javascript environments like NodeJS take this ECMAScript core and add features on top of it. ES5 came out in 2015 and after 6 years, in 2015, ES6 got finalised.

The official name is ECMAScript 2015, but ES6 is the name that everyone knows and uses.

ECMAScript 6 is a superset of ECMAScript 5. Therefore, all of your ES5 code is automatically ES6 code.

ES6 is an important upgrade for Javascript, bringing many good features that are popular in other programming languages.

Also, ES6 runs in strict mode by default. You don't need to explicitly set the strict mode.

ES6

This is pretty much the starting point for all confusion. All you need to remember is this → ES6 is the same as ES2015!!

After it was initially released in June 2015, it was known as ES6, but then later the committee wanted to keep the release in par with the year it was releasing and hence it was renamed to ES2015. Subsequent releases were also named according to the year of release such as ES2016, ES2017, etc.

ES6 or ES2015 was one of the most important releases due to a number of features released to bring JavaScript in par with other modern languages.

Notable Features

- Class declarations (class Person() { ... })
- Introduction to Modules - import * as moduleName from '.filename'; export const Person
- Iterators for...of loops
- Function Expressions (function() ⇒ { ...}())
- Collections such as Maps, Sets

ES7 (ES2016)

ES7 or officially known as ES2016 was released on June 2016.

Notable Features

- Async/Await for asynchronous programming.
- Block Scoping of variables & functions.
- Destructuring patterns of variables.

ES8 (ES2017)

ES8 or officially known as ES2017 was released on June 2017.

Notable Features

- Async/Await Constructors.
- Features for concurrency and atomics.

ES9 (ES2018)

S9 or officially known as ES2018 was released on June 2018.

Notable Features

- Rest/Spread operators for variables (three dots ... identifier)
- Asynchronous Iteration
- Promise.prototype.finally()

Template Literals

ES6 introduces a new kind of string literal syntax called template strings. They look like ordinary strings, except using the backtick character ` rather than the usual quote marks ' or ".

Template strings bring simple string interpolation to JavaScript. That is, they can be used to plug values into Javascript strings. Example:

console.log(`User ${user.name} is not authorized to do ${action}.`);

${} is called template substitution.

The code in a template substitution can be any JavaScript expression, so function calls, arithmetic, and so on are allowed.

If you need to write a backtick inside a template string, you must escape it with a backslash: `\` is the same as "`".

Template strings can span multiple lines.

Template strings are not a replacement of template engines like Handlebars as they do not handle loops, ifs etc

String New methods

string.startsWith
The startsWith() method determines whether a string begins with the characters of a specified string, returning true or false as appropriate.

```
const str1 = 'Saturday night plans';

console.log(str1.startsWith('Sat'));
// expected output: true

console.log(str1.startsWith('Sat', 3));
// expected output: false
```

string.endsWith
The endsWith() method determines whether a string ends with the characters of a specified string, returning true or false as appropriate.

```
const str1 = 'Cats are the best!';

console.log(str1.endsWith('best', 17));
// expected output: true

const str2 = 'Is this a question';

console.log(str2.endsWith('?'));
// expected output: false
```

string.includes

The includes() method determines whether one string may be found within another string, returning true or false as appropriate.

```
const sentence = 'The quick brown fox jumps over the lazy dog.';

const word = 'fox';

console.log(`The word "${word}" ${sentence.includes(word) ? 'is' : 'is not'} in the sentence`);
// expected output: "The word "fox" is in the sentence"
```

string.repeat

The repeat() method constructs and returns a new string which contains the specified number of copies of the string on which it was called, concatenated together.

```
const chorus = 'Because I\'m happy. ';



console.log(`Chorus lyrics for "Happy": ${chorus.repeat(3)}`);



// expected output: "Chorus lyrics for "Happy": Because I'm happy. Because I'm happy. Because I'm happy. "
```

string.raw

The static String.raw() method is a tag function of template literals. This is similar to the r prefix in Python, or the @ prefix in C# for string literals. It's used to get the raw string form of template strings, that is, substitutions (e.g. ${foo}) are processed, but escapes (e.g. \n) are not.

```
// Create a variable that uses a Windows

// path without escaping the backslashes:

const filePath = String.raw`C:\Development\profile\aboutme.html`;



console.log(`The file was uploaded from: ${filePath}`);

// expected output: "The file was uploaded from: C:\Development\profile\aboutme.html"
```

Let’s roll by 09:05 !!!

Javascript Intermediate 4

Gopi Krishnan R

Topics to be Discussed

- Arrow functions
- Destructuring
- Rest and Spread

Arrow function

An arrow function expression is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.

Differences & Limitations:

- Does not have its own bindings to this or super, and should not be used as methods.
- Does not have arguments, or new.target keywords.
- Not suitable for call, apply and bind methods, which generally rely on establishing a scope.
- Can not be used as constructors.
- Can not use yield, within its body.

```
// Traditional Function
function (a){
  return a + 100;
}

// Arrow Function Break Down

// 1. Remove the word "function" and place arrow between the argument and opening body bracket
(a) => {
  return a + 100;
}

// 2. Remove the body brackets and word "return" -- the return is implied.
(a) => a + 100;

// 3. Remove the argument parentheses
a => a + 100;
```

Default Parameters

Often, a function doesn’t need to have all its possible parameters passed by callers, and there are sensible defaults that could be used for parameters that are not passed.

ES6 introduces the default parameter concept to Javascript, the way it works in other languages. Example:

```
function animalSentence(animals2="tigers", animals3="bears") {

    return `Lions and ${animals2} and ${animals3}! Oh my!`;

}
```

Destructuring

The two most used data structures in JavaScript are Object and Array.

- Objects allow us to create a single entity that stores data items by key.
- Arrays allow us to gather data items into an ordered list.

Although, when we pass those to a function, it may need not an object/array as a whole. It may need individual pieces.

Destructuring assignment is a special syntax that allows us to “unpack” arrays or objects into a bunch of variables, as sometimes that’s more convenient.

Destructuring also works great with complex functions that have a lot of parameters, default values, and so on. Soon we’ll see that.

- Array Destructuring
- Object Destructuring

Rest Parameter

ES6 supports functions that can accept any number of parameters. We do not need to use the arguments anymore. Instead we can use the rest operator.

A rest parameter is indicated by three dots (...) preceding a named parameter. That named parameter becomes an Array containing the rest of the parameters passed to the function, which is where the name “rest” parameters originates.

```
function logArguments(...args) {
    for (let arg of args) {
        console.log(arg);
    }
}
```

Spread Operator

We’ve just seen how to get an array from the list of parameters.

But sometimes we need to do exactly the reverse.

```
let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5 (spread turns array into a list of arguments)
```

Let’s roll by 09:05 !!!

Javascript Intermediate 5

Gopi Krishnan R

Topics to be Discussed

- Prototypal inheritance in ES5
- ES6 Classes
- ES6 Getter and Setter

Prototypal Inheritance

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.

Nearly all objects in JavaScript are instances of Object which sits on the top of a prototype chain.

```
// Let's create an object o from function f with its own properties a and b:
let f = function () {
   this.a = 1;
   this.b = 2;
}
let o = new f(); // {a: 1, b: 2}

// add properties in f function's prototype
f.prototype.b = 3;
f.prototype.c = 4;

// do not set the prototype f.prototype = {b:3,c:4}; this will break the prototype chain
// o.[[Prototype]] has properties b and c.
// o.[[Prototype]].[[Prototype]] is Object.prototype.
// Finally, o.[[Prototype]].[[Prototype]].[[Prototype]] is null.
// This is the end of the prototype chain, as null,
// by definition, has no [[Prototype]].
// Thus, the full prototype chain looks like:
// {a: 1, b: 2} ---> {b: 3, c: 4} ---> Object.prototype ---> null

console.log(o.a); // 1
// Is there an 'a' own property on o? Yes, and its value is 1.

console.log(o.b); // 2
// Is there a 'b' own property on o? Yes, and its value is 2.
// The prototype also has a 'b' property, but it's not visited.
// This is called Property Shadowing
```

```
var o = {
  a: 2,
  m: function() {
    return this.a + 1;
  }
};

console.log(o.m()); // 3
// When calling o.m in this case, 'this' refers to o

var p = Object.create(o);
// p is an object that inherits from o

p.a = 4; // creates a property 'a' on p
console.log(p.m()); // 5
// when p.m is called, 'this' refers to p.
// So when p inherits the function m of o,
// 'this.a' means p.a, the property 'a' of p
```

```
function Superhero(superName, realName, powers){
  this.superName = superName,
  this.realName = realName,
  this.powers = powers
}
const wonderWoman = new Superhero('Wonder Woman', 'Diana Prince', 'Strength and flight');
console.log(wonderWoman); /* Superhero {
  superName: 'Wonder Woman',
  realName: 'Diana Prince',
  powers: 'Strength and flight' } */
Superhero.prototype.equipment = 'Lasso of truth';
console.log(wonderWoman.realName); // Diana Prince
console.log(wonderWoman.equipment); // Lasso of truth
console.log(wonderWoman.catchPhrase); // undefined
console.log(wonderWoman.hasOwnProperty('equipment')); // false
console.log(Superhero.hasOwnProperty('equipment')); // false
console.log(Superhero.prototype.hasOwnProperty('equipment')); // true
```

Classes

ES6 supports creating a class like the way it is created in other object oriented language like Java.

This feature is actually a syntatic sugar coating for the ES5 constructor based design.

Class declarations begin with the class keyword followed by the name of the class.

```
class Person {
    constructor(name, age, gender) {
        this.name   = name;
        this.age    = age;
        this.gender = gender;
    }

    incrementAge() {
      this.age += 1;
    }
}
```

Subclass

```
class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }

    getArea() {
        return this.length * this.width;
    }
}

class Square extends Rectangle {
    constructor(length) {

        // same as Rectangle.call(this, length, length)
        super(length, length);
    }
}

var square = new Square(3);

console.log(square.getArea());              // 9
console.log(square instanceof Square);      // true
console.log(square instanceof Rectangle);   // true
```

Getter and Setter

Getter

The get syntax binds an object property to a function that will be called when that property is looked up.

Setter

The set syntax binds an object property to a function to be called when there is an attempt to set that property.

In our class above we have a getter and setter for our name property. We use _ convention to create a backing field to store our name property. Without this every time get or set is called it would cause a stack overflow. The get would be called and which would cause the get to be called again over and over creating an infinite loop.

Something to note is that our backing field this._name is not private. Someone could still access bob._name and retrieve the property. To achieve private state on objects, you would use ES6 symbol and module to create true encapsulation and private state. Private methods can be created using module or traditional closures using an IIFE.

```
// ES6 get and set
class Person {
  constructor(name) {
    this._name = name;
  }

  get name() {
    return this._name.toUpperCase();
  }

  set name(newName) {
    this._name = newName; // validation could be checked here such as only allowing non numerical values
  }

  walk() {
    console.log(this._name + ' is walking.');
  }
}

let bob = new Person('Bob');
console.log(bob.name); // Outputs 'BOB'
```

Let’s roll by 09:05 !!!

Javascript Intermediate 6

Gopi Krishnan R

Topics to be Discussed

- ES6 Getter and Setter
- HTTP and HTTP Methods
- AJAX

Getter and Setter

Getter

The get syntax binds an object property to a function that will be called when that property is looked up.

Setter

The set syntax binds an object property to a function to be called when there is an attempt to set that property.

In our class above we have a getter and setter for our name property. We use _ convention to create a backing field to store our name property. Without this every time get or set is called it would cause a stack overflow. The get would be called and which would cause the get to be called again over and over creating an infinite loop.

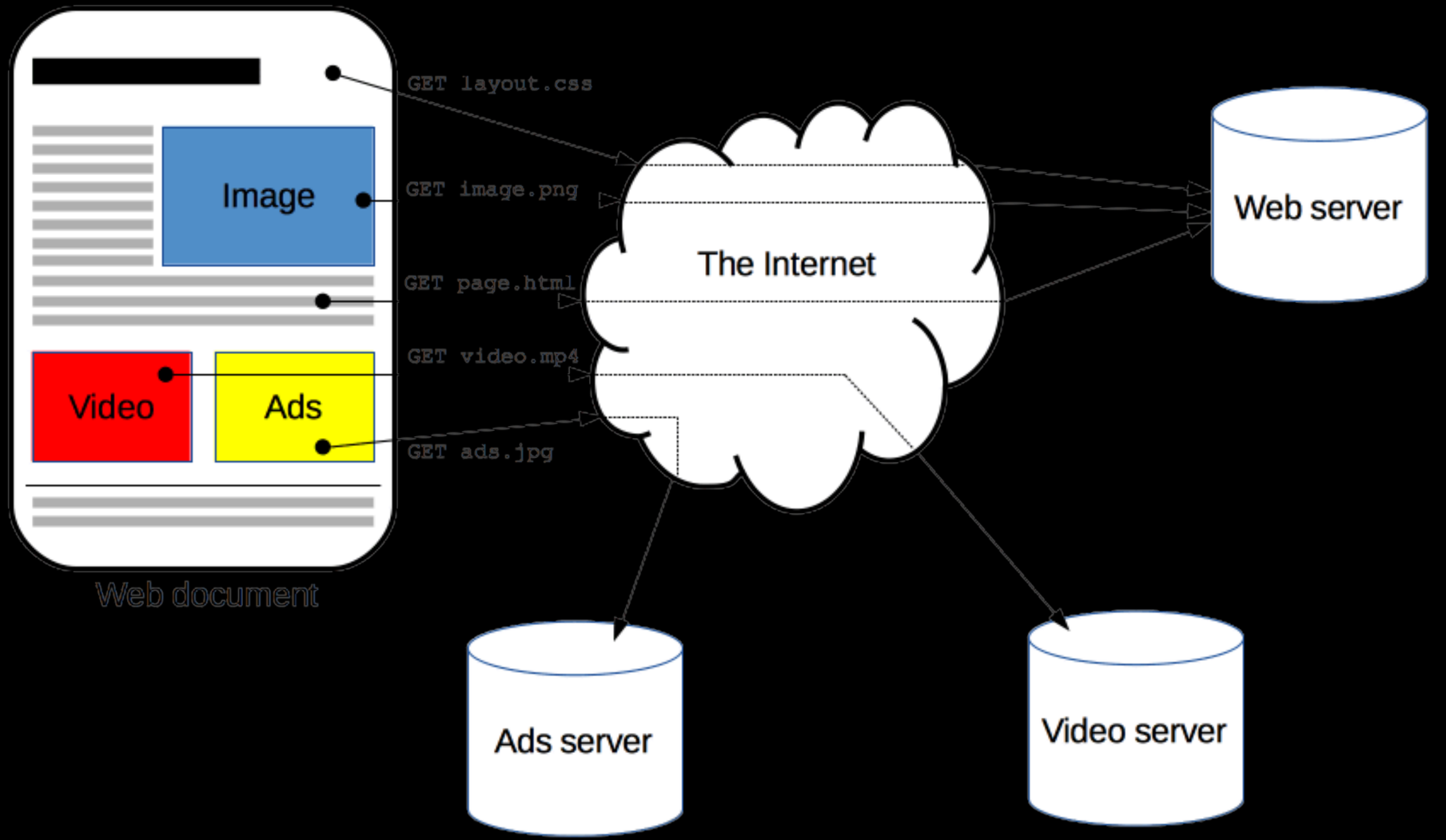
Something to note is that our backing field this._name is not private. Someone could still access bob._name and retrieve the property. To achieve private state on objects, you would use ES6 symbol and module to create true encapsulation and private state. Private methods can be created using module or traditional closures using an IIFE.

```
// ES6 get and set
class Person {
  constructor(name) {
    this._name = name;
  }

  get name() {
    return this._name.toUpperCase();
  }

  set name(newName) {
    this._name = newName; // validation could be checked here such as only allowing non numerical values
  }

  walk() {
    console.log(this._name + ' is walking.');
  }
}

let bob = new Person('Bob');
console.log(bob.name); // Outputs 'BOB'
```

HTTP

HTTP is a protocol which allows the fetching of resources, such as HTML documents. It is the foundation of any data exchange on the Web and it is a client-server protocol, which means requests are initiated by the recipient, usually the Web browser. A complete document is reconstructed from the different sub-documents fetched, for instance text, layout description, images, videos, scripts, and more.

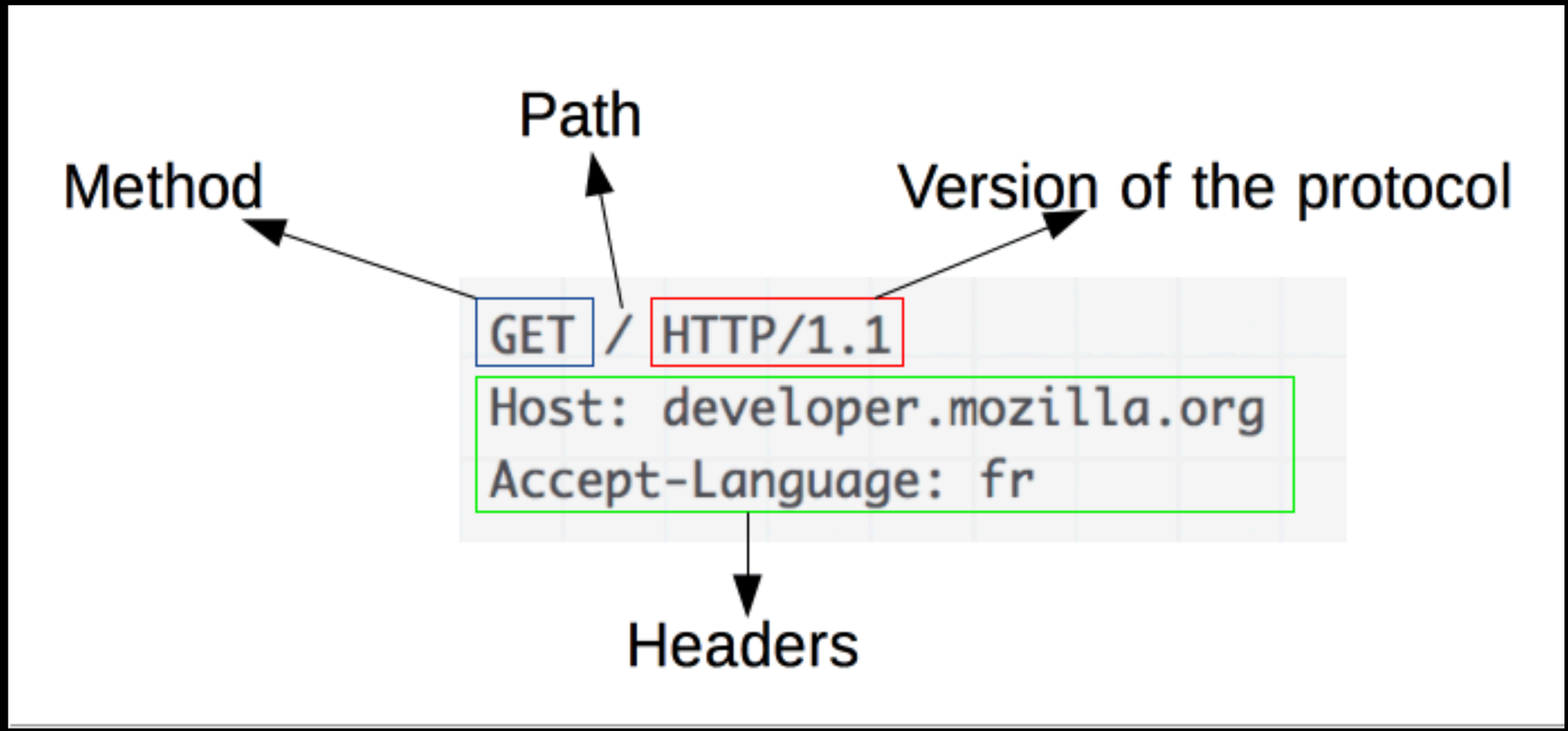
Clients and servers communicate by exchanging individual messages (as opposed to a stream of data). The messages sent by the client, usually a Web browser, are called requests and the messages sent by the server as an answer are called responses.


GET layout.css
Image
GET image.png
GET page.html
Video
Ads
GET video.mp4
GET ads.jpg
Web document
The Internet
Web server
Ads server
Video server

Request

Requests consists of the following elements:

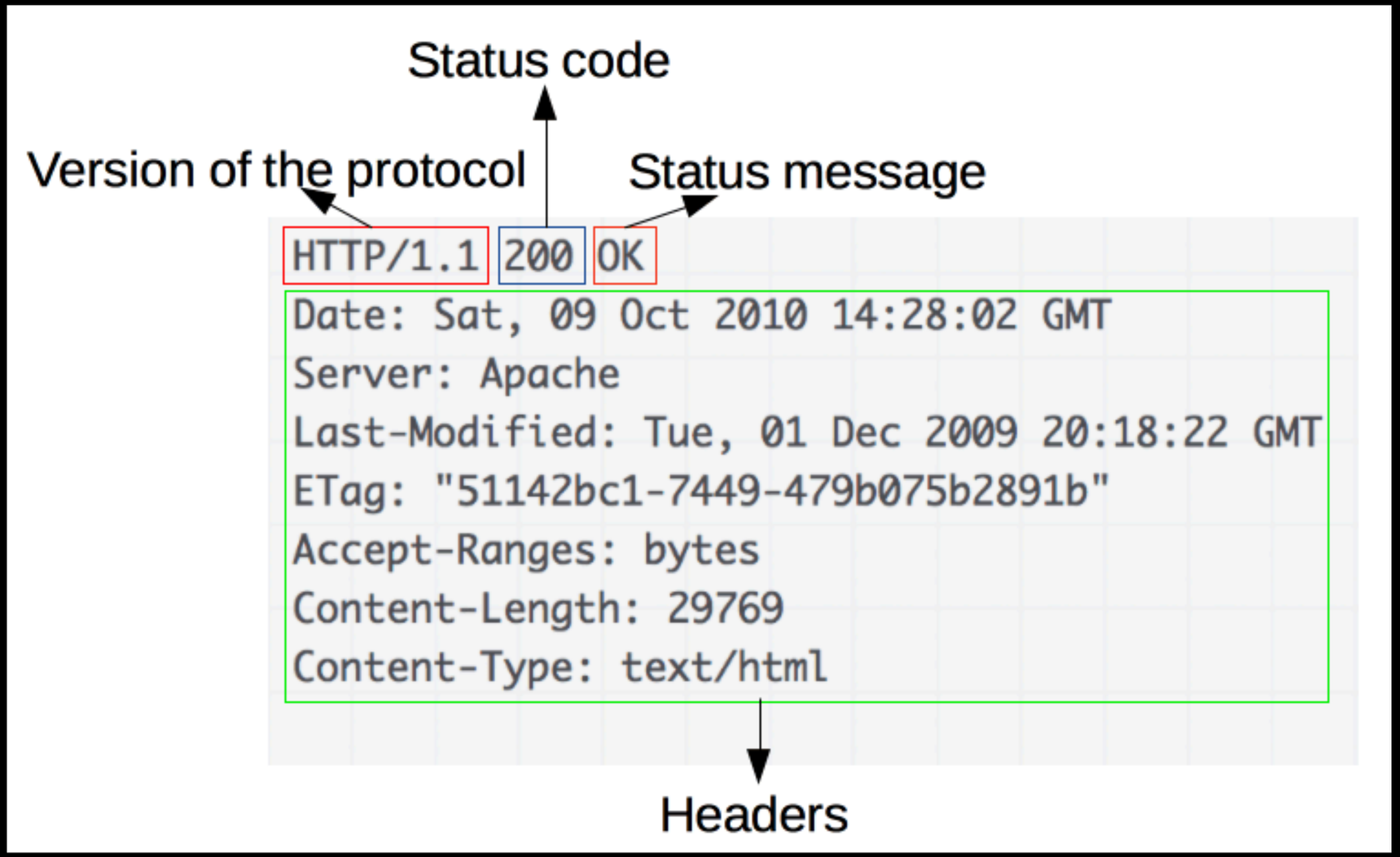
- An HTTP method, usually a verb like GET, POST or a noun like OPTIONS or HEAD that defines the operation the client wants to perform. Typically, a client wants to fetch a resource (using GET) or post the value of an HTML form (using POST), though more operations may be needed in other cases.
- The path of the resource to fetch; the URL of the resource stripped from elements that are obvious from the context, for example without the protocol (http://), the domain (here, developer.mozilla.org), or the TCP port (here, 80).
- The version of the HTTP protocol.
- Optional headers that convey additional information for the servers.
- Or a body, for some methods like POST, similar to those in responses, which contain the resource sent.


Path
Method
Version of the protocol
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
Headers

Response

Responses consist of the following elements:

- The version of the HTTP protocol they follow.
- A status code, indicating if the request was successful, or not, and why.
- A status message, a non-authoritative short description of the status code.
- HTTP headers, like those for requests.
- Optionally, a body containing the fetched resource.


Status code
Version of the protocol
Status message
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html
Headers

APIs based on HTTP

The most commonly used API based on HTTP is the XMLHttpRequest API, which can be used to exchange data between a user agent and a server. The modern Fetch API provides the same features with a more powerful and flexible feature set.

Request Types, Methods or Verbs

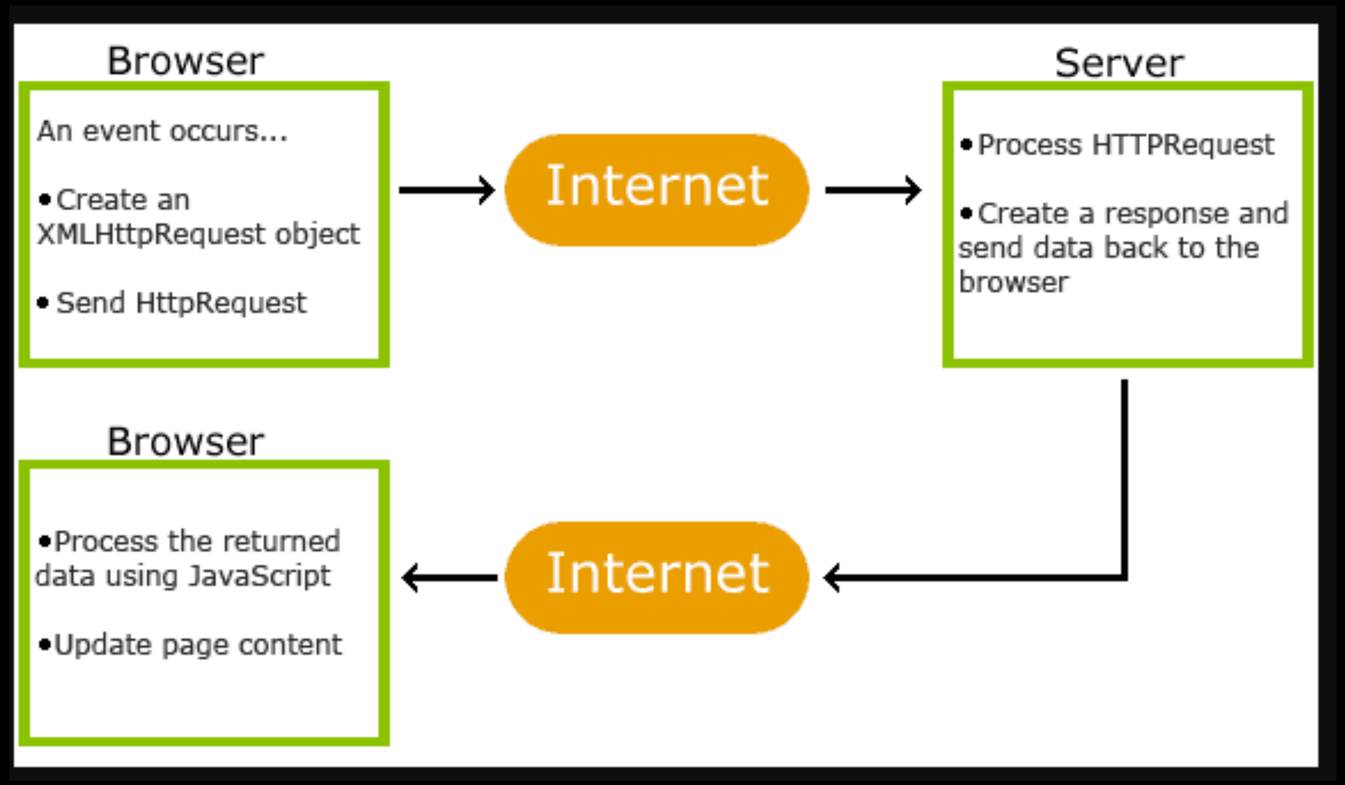
The HTTP protocol now support 8 request types, also called methods or verbs in the documentation,they are:

- GET – Requesting resource from server
- POST – submitting a resource to a server (e.g. file uploads)
- PUT -As POST but replaces a resource
- DELETE-Delete a resource from a server
- HEAD – As GET but only return headers and not content
- OPTIONS -Get the options for the resource
- PATCH -Apply modifications to a resource
- TRACE -Performs message loop-back

On the Internet today the GET (getting web pages) and POST (submitting web forms)methods are the ones most commonly used.

AJAX

Ajax is an acronym for Asynchronous Javascript and XML. It is used to communicate with the server without refreshing the web page and thus increasing the user experience and better performance.



```
// 1. Create a new XMLHttpRequest object
let xhr = new XMLHttpRequest();

// 2. Configure it: GET-request for the URL /article/.../load
xhr.open('GET', '/article/xmlhttprequest/example/load');

// 3. Send the request over the network
xhr.send();

// 4. This will be called after the response is received
xhr.onload = function() {
  if (xhr.status != 200) { // analyze HTTP status of the response
    alert(`Error ${xhr.status}: ${xhr.statusText}`); // e.g. 404: Not Found
  } else { // show the result
    alert(`Done, got ${xhr.response.length} bytes`); // response is the server response
  }
};

xhr.onprogress = function(event) {
  if (event.lengthComputable) {
    alert(`Received ${event.loaded} of ${event.total} bytes`);
  } else {
    alert(`Received ${event.loaded} bytes`); // no Content-Length
  }

};

xhr.onerror = function() {
  alert("Request failed");
};
```

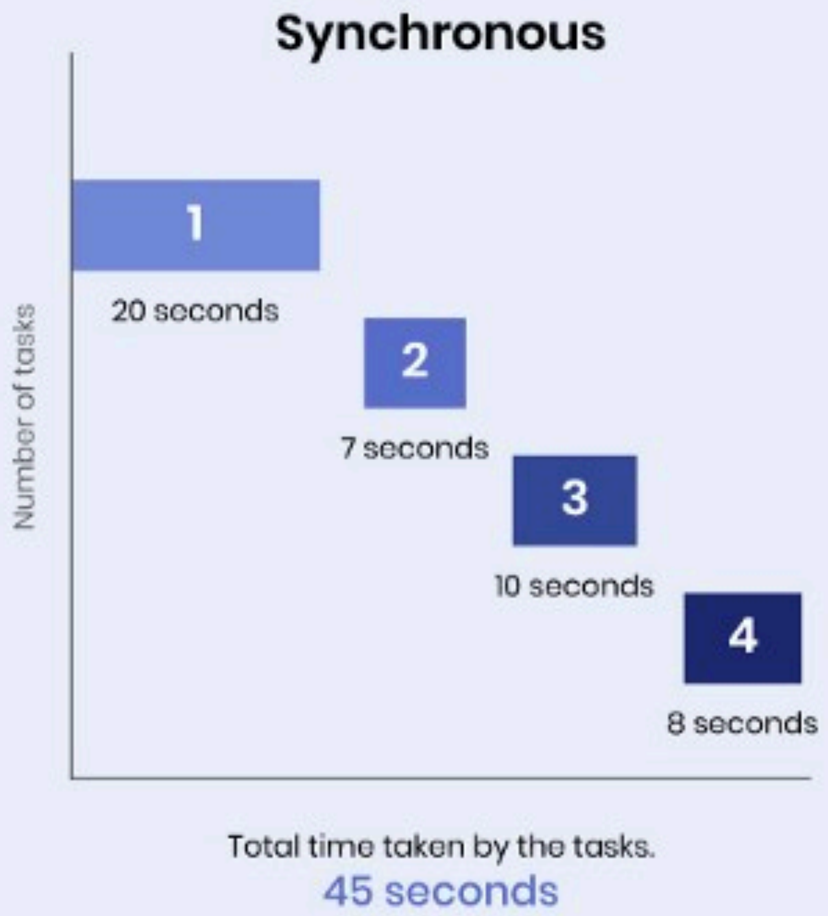
Let’s roll by 09:05 !!!

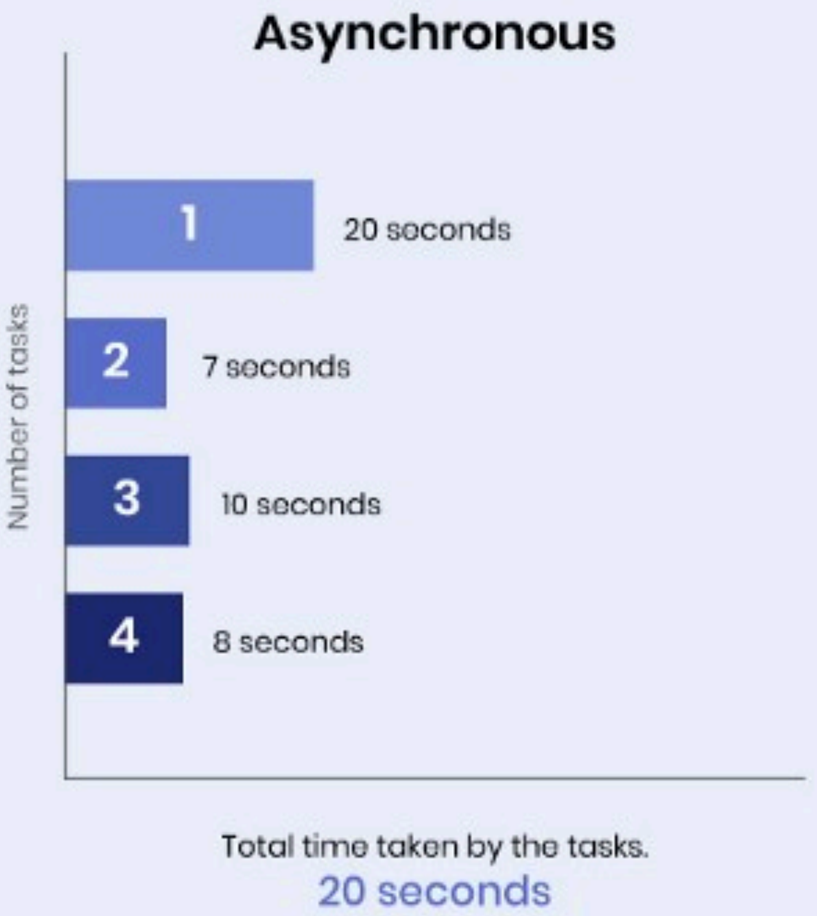
Javascript Intermediate 7

Gopi Krishnan R

Topics to be Discussed

- Promise
- AJAX using Fetch
- AJAX using async and await


Synchronous
Number of tasks
1
20 seconds
2
7 seconds
3
10 seconds
4
8 seconds
Total time taken by the tasks.
45 seconds


Asynchronous
Number of tasks
1
20 seconds
2
7 seconds
3
10 seconds
4
8 seconds
Total time taken by the tasks.
20 seconds

Promise

A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

When a program is executed, it is combined with a series of tasks.

- When a task is executed **synchronously**, it means that each task will have to be fully completed before moving on to the next task.
- When it is executed **asynchronously**, it means that the program can move onto another task before the other one is finished. The execution of the asynchronous task will not affect the execution of the overall program.

These are the three stages of a promise.

1. Pending: you don’t know whether you’ll get the video game at the end of the week.
2. Resolved or Fulfilled: your parents buy you the video game.
3. Rejected: you don’t get the video game because your parents are not happy with your grades.

.then() method

The object promise of type Promise has the method then(). This method takes two functions as its parameters. One is a function that will be passed the resolved value of the Promise once it is fulfilled. The other is a function to be called if the Promise is rejected.

.catch() method

When the Promise has both a success and an error handler, what happens if the success handler throws an error? There’s nothing there to catch it. As a result, the error gets swallowed. This is where the .catch() method comes in.

Fetch API

The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a global fetch() method that provides an easy, logical way to fetch resources asynchronously across the network.

This kind of functionality was previously achieved using XMLHttpRequest. Fetch provides a better alternative that can be easily used by other technologies such as Service Workers. Fetch also provides a single logical place to define other HTTP-related concepts such as CORS and extensions to HTTP.

```
fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(data => console.log(data));
```

The fetch() method can optionally accept a second parameter, an init object that allows you to control a number of different settings:

```
const url = 'https://randomuser.me/api';

let data = {
  name: 'Sara'
}

var request = new Request(url, {
    method: 'POST',
    body: data,
    headers: new Headers()
});

fetch(request)
.then(function() {
    // Handle response we get from the API
})
```

Async

Let’s start with the async keyword. It can be placed before a function, like this:

```
async function f() {
  return 1;
}
```

The word “async” before a function means one simple thing: a function always returns a promise. Other values are wrapped in a resolved promise automatically.

For instance, this function returns a resolved promise with the result of 1; let’s test it:

```
async function f() {
  return 1;
}

f().then(alert); // 1
```

We could explicitly return a promise, which would be the sam

```
async function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

So, async ensures that the function returns a promise, and wraps non-promises in it. Simple enough, right? But not only that. There’s another keyword, await, that works only inside async functions, and it’s pretty cool.

Await

The keyword await makes JavaScript wait until that promise settles and returns its result.

Here’s an example with a promise that resolves in 1 second:

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise; // wait until the promise resolves (*)

  alert(result); // "done!"
}

f();
```

The function execution “pauses” at the line (*) and resumes when the promise settles, with result becoming its result. So the code above shows “done!” in one second.

Let’s emphasize: await literally suspends the function execution until the promise settles, and then resumes it with the promise result. That doesn’t cost any CPU resources, because the JavaScript engine can do other jobs in the meantime: execute other scripts, handle events, etc.

Error handling

If a promise resolves normally, then await promise returns the result. But in the case of a rejection, it throws the error, just as if there were a throw statement at that line.

This code:

```
async function f() {
  await Promise.reject(new Error("Whoops!"));
}
```

In real situations, the promise may take some time before it rejects. In that case there will be a delay before await throws an error.

We can catch that error using try..catch, the same way as a regular throw:

```
async function f() {

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

Let’s roll by 09:05 !!!

Javascript Intermediate 8

Gopi Krishnan R

Topics to be Discussed

- Promise
- Iterators

Promise Contd.

- Promise Chaining
- Promise.all
- Promise.any
- Promise.resolve
- Promise.reject

Iterators

In JavaScript an iterator is an object which defines a sequence and potentially a return value upon its termination.

Specifically, an iterator is any object which implements the Iterator protocol by having a next() method that returns an object with two properties:

value

- The next value in the iteration sequence.

done

- This is true if the last value in the sequence has already been consumed. If value is present alongside done, it is the iterator's return value.

Once created, an iterator object can be iterated explicitly by repeatedly calling next(). Iterating over an iterator is said to consume the iterator, because it is generally only possible to do once. After a terminating value has been yielded additional calls to next() should continue to return {done: true}.

The most common iterator in JavaScript is the Array iterator, which returns each value in the associated array in sequence.

While it is easy to imagine that all iterators could be expressed as arrays, this is not true. Arrays must be allocated in their entirety, but iterators are consumed only as necessary. Because of this, iterators can express sequences of unlimited size, such as the range of integers between 0 and Infinity.

Here is an example which can do just that. It allows creation of a simple range iterator which defines a sequence of integers from start (inclusive) to end (exclusive) spaced step apart. Its final return value is the size of the sequence it created, tracked by the variable iterationCount.

```
function makeRangeIterator(start = 0, end = Infinity, step = 1) {
    let nextIndex = start;
    let iterationCount = 0;

    const rangeIterator = {
       next: function() {
           let result;
           if (nextIndex < end) {
               result = { value: nextIndex, done: false }
               nextIndex += step;
               iterationCount++;
               return result;
           }
           return { value: iterationCount, done: true }
       }
    };
    return rangeIterator;
}
```

Let’s roll by 09:05 !!!

Javascript Intermediate 9

Gopi Krishnan R

Topics to be Discussed

- Generators
- Modules in ES6

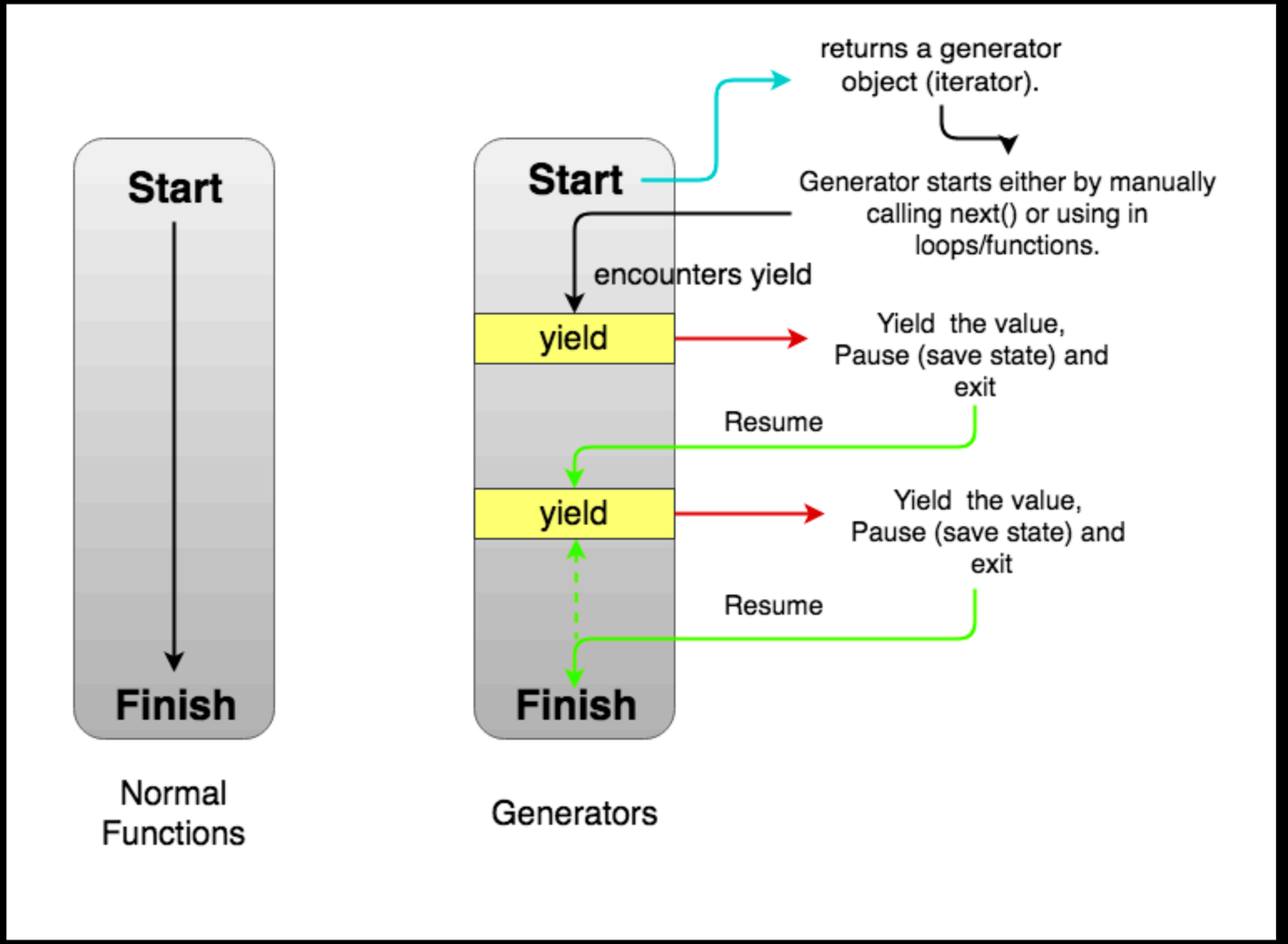
Generators

While custom iterators are a useful tool, their creation requires careful programming due to the need to explicitly maintain their internal state. Generator functions provide a powerful alternative: they allow you to define an iterative algorithm by writing a single function whose execution is not continuous. Generator functions are written using the `function*` syntax.

When called, generator functions do not initially execute their code. Instead, they return a special type of iterator, called a Generator. When a value is consumed by calling the generator's next method, the Generator function executes until it encounters the yield keyword.

The function can be called as many times as desired, and returns a new Generator each time. Each Generator may only be iterated once.

We can now adapt the example from above. The behavior of this code is identical, but the implementation is much easier to write and read.


Start
Finish
Normal Functions
Start
returns a generator object (iterator).
Generator starts either by manually calling next() or using in loops/functions.
encounters yield
yield
Yield the value, Pause (save state) and exit
Resume
yield
Yield the value, Pause (save state) and exit
Resume
Finish
Generators

```
function* makeRangeIterator(start = 0, end = 100, step = 1) {
    let iterationCount = 0;
    for (let i = start; i < end; i += step) {
        iterationCount++;
        yield i;
    }
    return iterationCount;
}
```

Modules

Prior to ES6, developers had to use third party libraries to use modules, like require in NodeJS.

Now ES6 supports native module systems.

An ES6 module is a file containing JS code. Everything declared inside a module is local to the module, by default. If you want something declared in a module to be public, so that other modules can use it, you must export that feature.

Exporting in ES6
The following are the various flavors of exports in ES6.

Named Exports

```
export let name = 'David';
export let age  = 25;
```

Export Lists
Rather than tagging each exported feature, you can write out a single list of all the names you want to export, wrapped in curly braces.

```
function sumTwo(a, b) {
    return a + b;
}

function sumThree(a, b, c) {
    return a + b + c;
}

export { sumTwo, sumThree };
```

Export defaults

```
let api = {
    sumTwo,
    sumThree
};

export default api;
```

Importing Everything form a Module

To import everything from a file, we can use

```
import 'underscore';
```

Named Imports

```
import { sumTwo, sumThree } from 'math/addition';
```

ES6 lets you rename things when you import them too.

```
import {
    sumTwo as addTwoNumbers,
    sumThree as sumThreeNumbers
} from 'math/addition';
```