JavaScript Introduction

Javascript is a dynamic computer programming language. It is lightweight and most commonly used as a part of web pages, whose implementations allow client-side script to interact with the user and make dynamic pages.

The ECMA-262 Specification defined a standard version of the core JavaScript language.

- **JavaScript is a lightweight, interpreted programming language.**
- **Designed for creating network-centric applications.**
- **Complementary to and integrated with Java.**
- **Complementary to and integrated with HTML.**
- **Open and cross-platform.**

Objects in javascript

Objects are composed of attributes. If an attribute contains a function, it is considered to be a method of the object, otherwise the attribute is considered a property.

The syntax for adding a property to an object is:

```
objectName.objectProperty = propertyValue;
```

For example: The following code gets the document title using the "title" property of the document object.

```
var str = document.title;
```

Methods are the functions that let the object do something or let something be done to it. There is a small difference between a function and a method – at a function is a standalone unit of statements and a method is attached to an object and can be referenced by the this keyword.

For example: Following is a simple example to show how to use the write() method of document object to write any content on the document.

```
document.write ("This is test");
```

Template Strings

Template strings provide us with an alternative to string concatenation. They also allow us to insert variables into a string. You'll hear these referred to as template strings, template literals, or string templates interchangeably.

Traditional string concatenation uses plus signs to compose a string using variable values and strings:

```
console.log(lastName + ", " + firstName + " " + middleName);
```

With a template, we can create one string and insert the variable values by surrounding them with ${ }:

```
console.log(`${lastName}, ${firstName} ${middleName}`);
```

Any JavaScript that returns a value can be added to a template string between the ${ } in a template string.

Creating Functions

Function Declarations

```
function   logCompliment() {
console.log("You're doing great!");
}
```

Function Expressions

```
const   logCompliment = function() {
console.log("You're doing great!");
};
logCompliment();
```

Arrow Functions

Arrow functions are a useful new feature of ES6. With arrow functions, you can create functions without using the function keyword. You also often do not have to use the return keyword.

```
// Typical function

Const  lordify = function(firstName, land) {

return `${firstName} of ${land}`;

};

// Arrow Function

const  lordify = (firstName, land) => `${firstName} of ${land}`;

console.log(lordify("Don", "Piscataway")); // Don of Piscataway

console.log(lordify("Todd", "Schenectady")); // Todd of Schenectady
```

Returning objects with arrow function

```
const person = (firstName, lastName) =>

{

first: firstName,

last: lastName

}

console.log(person("Brad", "Janson"));
```

As soon as you run this, you’ll see the error: Uncaught SyntaxError: Unexpected token :. To fix this, just wrap the object you’re returning with parentheses:

```
const person = (firstName, lastName) => ({

first: firstName,

last: lastName

});

console.log(person("Flad", "Hanson"));
```

These missing parentheses are the source of countless bugs in JavaScript and React apps, so it’s important to remember this one!

Compiling JavaScript Babel

As an example, let’s look at an arrow function with some default arguments:

```
const add = (x = 5, y = 10) => console.log(x + y);
```

If we run Babel on this code, it will generate the following:

```
"use strict";

var add = function add() {

var x =

arguments.length <= 0 || arguments[0] === undefined ? 5 : arguments[0];

var y =

arguments.length <= 1 || arguments[1] === undefined ? 10 : arguments[1];

return console.log(x + y);

};
```

Babel added a “use strict” declaration to run in strict mode. The variables x and y are defaulted using the arguments array, a technique you may be familiar with. The resulting JavaScript is more widely supported.

Destructuring objects

Destructuring assignment allows you to locally scope fields within an object and to declare which values will be used. Consider the sandwich object. It has four keys, bu we only want to use the values of two. We can scope bread and meat to be used locally

```
const sandwich = {

bread: "dutch crunch",

meat: "tuna",

cheese: "swiss",

toppings: ["lettuce", "tomato", "mustard"]

};

const { bread, meat } = sandwich;

console.log(bread, meat); // dutch crunch tuna
```

Javascript as functional programming

JavaScript supports functional programming which means that functions can do the same things that variables can do.

In JavaScript, functions can represent data in your application. You may have noticed that you can declare functions with the var, let, or const keywords the same way you can declare strings, numbers, or any other variables:

```
var log = function(message) {  console.log(message); };
```

Since functions are variables, we can add them to objects:

```
const obj = {   message: "They can be added to objects like variables",

log(message) {console.log(message); } };

obj.log(obj.message);
```

We can also add functions to arrays in JavaScript:

```
const messages = [ "They can be inserted into arrays", message => console.log(message),

"like variables",   message => console.log(message) ];
```

Functions can be sent to other functions as arguments, just like other variables:

```
const insideFn = logger => {  logger("They can be sent to other functions as arguments"); };

insideFn(message => console.log(message));
```

They can also be returned from other functions, just like variables:

```
const createScream = function(logger) {

return function(message) { logger(message.toUpperCase() + "!!!");  };

};

const scream = createScream(message => console.log(message));

scream("functions can be returned from other functions");

scream("createScream returns a function");

scream("scream invokes that returned function");
```

Imperative Versus Declarative

Functional programming is a part of a larger programming paradigm: declarative programming.

Declarative programming is a style of programming where applications are structured in a way that prioritizes describing what should happen over defining how it should happen.

Let’s consider a common task:

Making a string URL-friendly which can be accomplished by replacing all of the spaces in a string with hyphens, since spaces are not URL-friendly.

Imperative Approach:

```
const string = "Restaurants in Bhubaneswar";

const urlFriendly = "";

for (var i = 0; i < string.length; i++) {

if (string[i] === " ") {

urlFriendly += "-";

} else {

urlFriendly += string[i];

}

}

console.log(urlFriendly); // "Restaurants-in-Bhubaneswar"
```

In this example, we loop through every character in the string, replacing spaces as they occur. The structure of this program is only concerned with how such a task can be achieved. We use a for loop and an if statement and set values with an equality operator. Just looking at the code alone does not tell us much. Imperative programs require lots of comments in order to understand what's going on.

Declarative approach to the same problem:

```
const string = "Restaurants in Bhubaneswar";

const urlFriendly = string.replace(/ /g, "-");

console.log(urlFriendly);
```

Here we are using string.replace along with a regular expression to replace all instances of spaces with hyphens. Using string.replace is a way of describing what's supposed to happen: spaces in the string should be replaced. The details of how spaces are dealt with are abstracted away inside the replace function. In a declarative program, the syntax itself describes what should happen, and the details of how things happen are abstracted away.

Building a DOM(Imperative approach)

```
const target = document.getElementById("target");

const wrapper = document.createElement("div");

const headline = document.createElement("h1");

wrapper.id = "welcome";

headline.innerText = "Hello World";

wrapper.appendChild(headline);

target.appendChild(wrapper);
```

This code is concerned with creating elements,setting elements, and adding them to the document. It would be very hard to make changes, add features, or scale 10,000 lines of code where the DOM is constructed imperatively.

Building a DOM(Declarative approach)

```
const { render } = ReactDOM;

const Welcome = () => (

<div id="welcome">

<h1>Hello World</h1>

</div>

);

render(<Welcome />, document.getElementById("target"));
```

React is declarative. Here, the Welcome component describes the DOM that should be rendered. The render function uses the instructions declared in the component to build the DOM, abstracting away the details of how the DOM is to be rendered. We can clearly see that we want to render our Welcome component into the element with the ID of target.

Concepts of functional programming approach

- Immutability
- Purity
- Data transformation
- Higher-order functions
- Recursion

Immutability

To mutate is to change, so to be immutable is to be unchangeable. In a functional program,data is immutable. It never changes.Instead of changing the original data structures, we build changed copies of those data structures and use them instead.

```
let color_lawn = {

title: "lawn",

color: "#00FF00",

rating: 0

};
```

```
function rateColor(color, rating) {
color.rating = rating;
return color;
}
console.log(rateColor(color_lawn, 5).rating); // 5
console.log(color_lawn.rating); // 5

const rateColor = function(color, rating) {
return Object.assign({}, color, { rating: rating });
};
console.log(rateColor(color_lawn, 5).rating); // 5
console.log(color_lawn.rating); // 0
```

```
const rateColor = (color, rating) => ({
...color,
rating
});
```

PURITY

A pure function is a function that returns a value that's computed based on its arguments.Pure functions take at least one argument and always return a value or another function. They do not cause side effects, set global variables, or change anything about application state. They treat their arguments as immutable data.

Pure functions are another core concept of functional programming. They will make your life much easier because they will not affect your application's state. When writing functions, try to follow these three rules:

1. The function should take in at least one argument.

2. The function should return a value or another function.

3. The function should not change or mutate any of its arguments.

Data transformation

Array.map and Array.reduce.

const schools = ["Yorktown", "Washington & Liberty", "Wakefield"];

Array.map method takes a function as its argument. This function will be invoked once for every item in the array, and whatever it returns will be added to the new array:

const highSchools = schools.map(school => `${school} High School`);

console.log(highSchools.join("\n"));

// Yorktown High School// Washington & Liberty High School // Wakefield High School

console.log(schools.join("\n"));

// Yorktown // Washington & Liberty // Wakefield

In this case, the map function was used to append “High School” to each school name. The schools array is still intact.

The map function can produce an array of objects, values, arrays, other functions—any JavaScript type.

The reduce and reduceRight functions can be used to transform an array into any value, including a number, string, boolean, object, or even a function. Let’s say we need to find the maximum number in an array of numbers. We need to transform an array into a number; therefore, we can use reduce:

```
const ages = [21, 18, 42, 40, 64, 63, 34];

const maxAge = ages.reduce((max, age) => {

console.log(`${age} > ${max} = ${age > max}`);

if (age > max) {          // 21 > 0 = true
                          // 18 > 21 = false
return age;               // 42 > 21 = true
                          // 40 > 42 = false
} else {                  // 64 > 42 = true
                          // 63 > 64 = false
return max;               // 34 > 64 = false
                          // maxAge 64
}

}, 0);

console.log("maxAge", maxAge);
```

The ages array has been reduced into a single value: the maximum age, 64. Reduce takes two arguments: a callback function and an original value. In this case, the original value is 0, which sets the initial maximum value to 0. The callback is invoked once for every item in the array.

Higher order functions

The use of higher-order functions is also essential to functional programming.Higher-order functions are functions that can manipulate other functions. They can take functions in as arguments or return functions or both.

The first category of higher-order functions are functions that expect other functions as arguments. Array.map, Array.filter, and Array.reduce all take functions as arguments. They are higher-order functions.

```
const invokeIf = (condition, fnTrue, fnFalse) =>
condition ? fnTrue() : fnFalse();
const showWelcome = () => console.log("Welcome!!!");
const showUnauthorized = () => console.log("Unauthorized!!!");
invokeIf(true, showWelcome, showUnauthorized); // "Welcome!!!"
invokeIf(false, showWelcome, showUnauthorized); // "Unauthorized!!!"
```

RECURSION

Recursion is a technique that involves creating functions that recall themselves.Often, when faced with a challenge that involves a loop, a recursive function can be used instead.

```
const countdown = (value, fn) => {
fn(value);
return value > 0 ? countdown(value - 1, fn) : value;
};
countdown(10, value => console.log(value));
```

```
// 10
// 9
// 8
// 7
// 6
// 5
// 4
// 3
// 2
// 1
// 0
```