Bridge Frontend Development

ES6 & Modern JavaScript Features

bridge

Modern JavaScript

ES5 vs ES6

ECMAScript

- Also known as **ES** for short
- Scripting language specification to standardize JavaScript
- All new language feature proposals are submitted to the TC39 committee
- Proposals pass through several stages before becoming official
- All Browsers support the **ES5** spec but not all fully support **ES6** and beyond
- Build tools like Babel are used to convert modern JS code down to **ES5**

Why do we use **ES6** if it’s not fully supported?

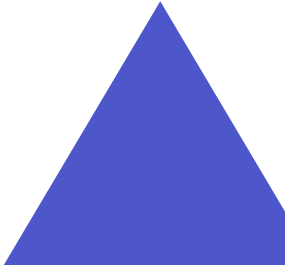
ES6 is full of great features that make development far easier

ES6 Features

Template Literals

What Are Template Literals?

- Special strings that allow you to add variables and run JavaScript inline
- Denoted with backticks ``



The following ES5 code...

```
const name = "Purvi";
const job = "CTO";
const company = "Bridge";

const sentence = "Hi, I'm " + name + "! I work for " + company
+ " as the " + job;

console.log(sentence); // "Hi, I'm Purvi! I work for Bridge as
the CTO"
```

...Becomes this in ES6

```
const name = "Purvi";
const job = "CTO";
const company = "Bridge";

const sentence = `Hi, I'm ${name}! I work for ${company} as the
${job}`;

console.log(sentence); // "Hi, I'm Purvi! I work for Bridge as
the CTO"
```

What the following code console log?

```
const quantity = 10;
const thing = 'apple';

console.log(
`I would like to buy ${quantity} ${thing}${quantity
!== 1 ? 's' : ''}`
);
```

a) "I would like to buy ${quantity} ${thing}${quantity !== 1 ? 's' : ''}"

b) "I would like to buy 10 apples"

c) "I would like to buy 10 apple"

d) An Error will be thrown

ES6 Features

Arrow Functions

What Are Arrow Functions?

- Called arrow function because of the “fat arrow” **=>**
- Shorthand syntax for declaring a function without *function* keyword
- Can return a value without using *return* keyword, known as **implicit** return

The following ES5 code...

```
function add(x, y) {
    return x + y;
}
console.log(add(5,7)) // 12
```

...Becomes this in ES6

```
const add = (x, y) => x + y;

console.log(add(5,7)) // 12
```

How would we write an arrow function that takes only one value and returns a string? (Remember to use implicit return!)

```
function (oneParam) {
    return oneParam
}

// your arrow function here
```

Fix this arrow function so it can return an object

```
const fn = () => { key: 'value' } // this is an error
```

ES6 Features

Destructuring

What Is Destructuring?

- Shorthand for extracting values from an object or array
- Can be used for variables or function parameters

Destructuring for variables

The following ES5 code...

```
// Objects
const cat = {name: 'Carrot', weight: 20};
const name = cat.name;
const weight = cat.weight;
console.log(name, weight); // Carrot 20

// Arrays
const numbers = [1,2,3,4];
const firstNumber = numbers[0];
const thirdNumber = numbers[2];
console.log(firstNumber, thirdNumber); // 1 3
```

...Becomes this in ES6

```
// Objects
const cat = {name: 'Carrot', weight: 20};
const { name, weight } = cat;
console.log(name, weight); // Carrot 20

// Arrays
const numbers = [1,2,3,4];
const [firstNumber,,thirdNumber] = numbers;
console.log(firstNumber, thirdNumber); // 1 3
```

Use array destructuring to swap the two values

```
let [fruit1, fruit2] = ['orange', 'apple']

//expected values
console.log(fruit1, fruit2) // apple orange
```

How can we get the last fruit in this array without defining other variables?

```
let fruits = ['orange', 'apple', 'pomegranate']


const lastFruit = fruits


// expected value
console.log(lastFruit) // pomegranate
```

Fix this snippet to get the expected result. Use array destructuring to extract values from nested arrays

```
let fruits = ['orange',['granny smith', 'ambrosia'], 'pomegranate']

const [orange, apple1, apple2, pomegranate] = fruits

console.log(orange, apple1, apple2, pomegranate)
// orange, granny smith, ambrosia, pomegranate
```

Destructuring works on nested objects too. Use it to assign the variable `ellasAge` to Ella the cat's age in the object

```
const pets = {
    'Ella': {
        age: 13,
        type: 'cat',
    },
    'Carrot': {
        age: 6,
        type: 'cat',
    }
}

const ellasAge;

//expected result
console.log(ellasAge) // 13
```

Destructuring for Function parameters

The following ES5 code...

```
function formatFullName(nameObject) {
    const first = nameObject.first;
    const middleInitial = nameObject.middleInitial;
    const last = nameObject.last;
    return first + " " + middleInitial + " " + last;
}
const fullName = formatFullName({
    first: 'Jenna',
    middleInitial: 'T',
    last: 'Davis'
})
console.log(fullName) // Jenna T Davis
```

...Becomes this in ES6

```
const formatFullName = ({first, middleInitial, last}) =>
`${first} ${middleInitial} ${last}`;

const fullName = formatFullName({
first: 'Jenna',
middleInitial: 'T',
last: 'Davis'
})
console.log(fullName) // Jenna T Davis
```

ES6 Features

The Rest/Spread Operator

What Is the rest/spread operator?

- Denoted by an ellipsis “...” followed by a variable name
- Works with Arrays and Objects
- Has two functionalities— **Rest** and **Spread**

Rest

Collect the **rest** of the values

Get the **rest** of the values when destructuring

```
// Objects
const cat = { name: 'Moose', color: 'gray', age: 10 };
const { age, ...restOfCat } = cat;
console.log(age) // 10
console.log(restOfCat) // { name: 'Moose', color: 'gray' }

// Arrays
const numbers = [1, 2, 3, 4, 5];
const [first, second, ...otherNumbers] = numbers;
console.log(first) // 1
console.log(second) // 2
console.log(otherNumbers) // [3, 4, 5];
```

Get the **rest** of the parameters as an array

```
const makeArray = (...numbers) => numbers;

const numbersArray = makeArray(1, 2, 3, 4, 5);

console.log(numbersArray); // [1, 2, 3, 4, 5]
```

Use the rest operator in an array to get the expected value

```
const fruits = ['orange', 'apple', 'pineapple']
const fruitsWeWant;

console.log(fruitsWeWant) // ['apple', 'pineapple']
```

Spread

Spread these values out

Spread values into new object/array

```
// Objects
const someDogData = { name: 'Optimus', age: 8 };
const moreDogData = {
...someDogData, // adds name and age to allDogData
breed: 'labrador'
};
console.log(moreDogData); // { name: 'Optimus', age: 8, breed:
'labrador' }

// arrays
const numbers = [1, 2, 3];
const moreNumbers = [...numbers, 4, 5];
console.log(moreNumbers) // [1, 2, 3, 4, 5];
```

Exercise: use the **spread** operator to add to create the newNumbers array

```
const numbers = [2,3,4]

const newNumbers;


console.log(newNumbers) // [1,2,3,4,5]
```

Spread array as function arguments

```
const add = (x, y, z) => x + y + z;

const numbers = [2, 4, 6];

const sum = add(...numbers);

console.log(sum) // 12
```

Exercise: use the **spread** operator to supply arguments to the function `fnThatUsesNumbers` when it is invoked

```
const numbers = [3,5,6]


const fnThatUsesNumbers = (a, b, c) => { return a + b + c }


console.log(fnThatUsesNumbers()) // 14
```