Introduction to ES6

- Part 2

Objective

- Have a basic understanding of the major features of ES6
- Understand how to use default, rest, spread.
- Learn the difference between let, const vs var.
- Get started with iterators, generators and for .. of loop.

Content

- ES features
- Part 2:
  - Default + rest + spread
  - Let + const
  - Iterators + for .. of
  - Generators

ES6 features

- Part 1:
  - Arrows
  - Classes
  - Enhanced object literals
  - Template strings
  - Destructuring

ES6 features

- Part 2:
  - Default + rest + spread
  - Let + const
  - Iterators + for .. of
  - Generators

ES6 features

- Part 3:
  - Modules (NodeJS)
  - Map + Set + WeakMap + WeakSet
  - Promises
  - Math + Number + String + Array + Object APIs

Part 2

Part 2: Default + rest + spread

- **Default**

```
function f(x, y=12) {
        // y is 12 if not passed (or passed as undefined)
        return x + y;
}

f(3) == 15
```

Part 2: Default + rest + spread

- **Rest**

```
function f(x, ...y) {
        // y is an Array
        return x * y.length;
}

f(3, "hello", true) == 6
```

Part 2: Default + rest + spread

- **Spread**

```
function f(x, y, z) {
        return x + y + z;
}

// Pass each element of array as argument
f(...[1,2,3]) == 6
```

Part 2: Let + const

- **let**

let allows you to declare variables that are limited in scope to the block, statement, or expression on which it is used.

This is unlike the **var** keyword, which defines a variable globally, or locally to an entire function regardless of block scope

Part 2: Let + const

- **let**

```
function varTest() {
  var x = 1;
  if (true) {
    var x = 2;  // same variable!
    console.log(x);  // 2
  }
  console.log(x);  // 2
}
```

```
function letTest() {
  let x = 1;
  if (true) {
    let x = 2;  // different variable
    console.log(x);  // 2
  }
  console.log(x);  // 1
}
```

Part 2: Let + const

- **const**

Constants are **block-scoped**, much like variables defined using the **let** statement. The value of a constant cannot change through re-assignment, and it can't be redeclared.

The **const** declaration creates a **read-only** reference to a value. It does not mean the value it holds is immutable, just that the variable identifier cannot be reassigned. For instance, in the case where the content is an object, this means the object's contents (e.g. its parameters) can be altered.

Part 2: Iterators + for .. of

- **Iterators**

An object is an **iterator** when it knows how to access items from a collection one at a time, while keeping track of its current position within that sequence.

In JavaScript an **iterator** is an object that provides a next() method which returns the next item in the sequence. This method returns an object with two properties: done and value.

Part 2: Iterators + for .. of

- **Iterators**

Property	Value
next	A *zero* arguments function that returns an object with two properties: • **done** (boolean) • - Has the value true if the iterator is past the end of the iterated sequence. In this case value optionally specifies the return value of the iterator. The return values are explained here. • - Has the value false if the iterator was able to produce the next value in the sequence. This is equivalent of not specifying the done property altogether. • **value** - any JavaScript value returned by the iterator. Can be omitted when done is true. The **next** method always has to return an object with appropriate properties including done and value. If a non-object value gets returned (such as false or undefined), a TypeError ("iterator.next() returned a non-object value") will be thrown

Part 2: Iterators + for .. of

- **Iterators**

Example:

```
const sequence = () => {
  let cur = 0;
  return {
                next: () => {
  return {
done: false, value: cur++,
  };
                },
  };
};
```

Part 2: Iterators + for .. of

- **for .. of**

The **for .. of** statement creates a loop iterating over iterable objects (including Array, Map, Set, String, TypedArray, arguments object and so on), invoking a custom iteration hook with statements to be executed for the value of each distinct property

Part 2: Iterators + for .. of

- **Iterable protocol**

The **iterable** protocol allows JavaScript objects to define or customize their iteration behavior, such as what values are looped over in a for..of construct. Some built-in types are built-in iterables with a default iteration behavior, such as Array or Map, while other types (such as Object) are not.

In order to be **iterable**, an object must implement the **@@iterator method**, meaning that the object (or one of the objects up its prototype chain) must have a property with a **@@iterator key** which is available via constant *Symbol.iterator*.

Part 2: Iterators + for .. of

- **Iterable & for .. of**

Example:

```
const sequence = {
  [Symbol.iterator]() {
          let cur = 0;
          return {
  next() {
          return { done: false, value: cur++ };
  },
          };
  },
};

for (const n of sequence) {
  if (n > 20)
          break;
  console.log(n);
}
```

Part 2: Generators

- **Definitions**

Generators are functions which can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances.

Calling a generator function does not **execute its body immediately**; an iterator object for the function is returned instead. When the iterator's **next()** method is called, the generator function's body is executed until the first **yield** expression.

Part 2: Generators

- **Syntax**

```
function* gen() {
  yield 1;
  yield 2;
  yield 3;
}

const g = gen(); // "Generator { }"
```

Part 2: Generators

- **Simple example**

```
        function* foo() {
yield 1;
yield 2;
yield 3;
yield 4;
yield 5;
        }

        var it = foo();
        ...
```

Part 2: Generators

- **Simple example**

```
...
console.log( it.next() ); // { value:1, done:false }
console.log( it.next() ); // { value:2, done:false }
console.log( it.next() ); // { value:3, done:false }
console.log( it.next() ); // { value:4, done:false }
console.log( it.next() ); // { value:5, done:false }
console.log( it.next() ); // { value:undefined, done:true }
```

Thank you