ES6

Naveen Pete

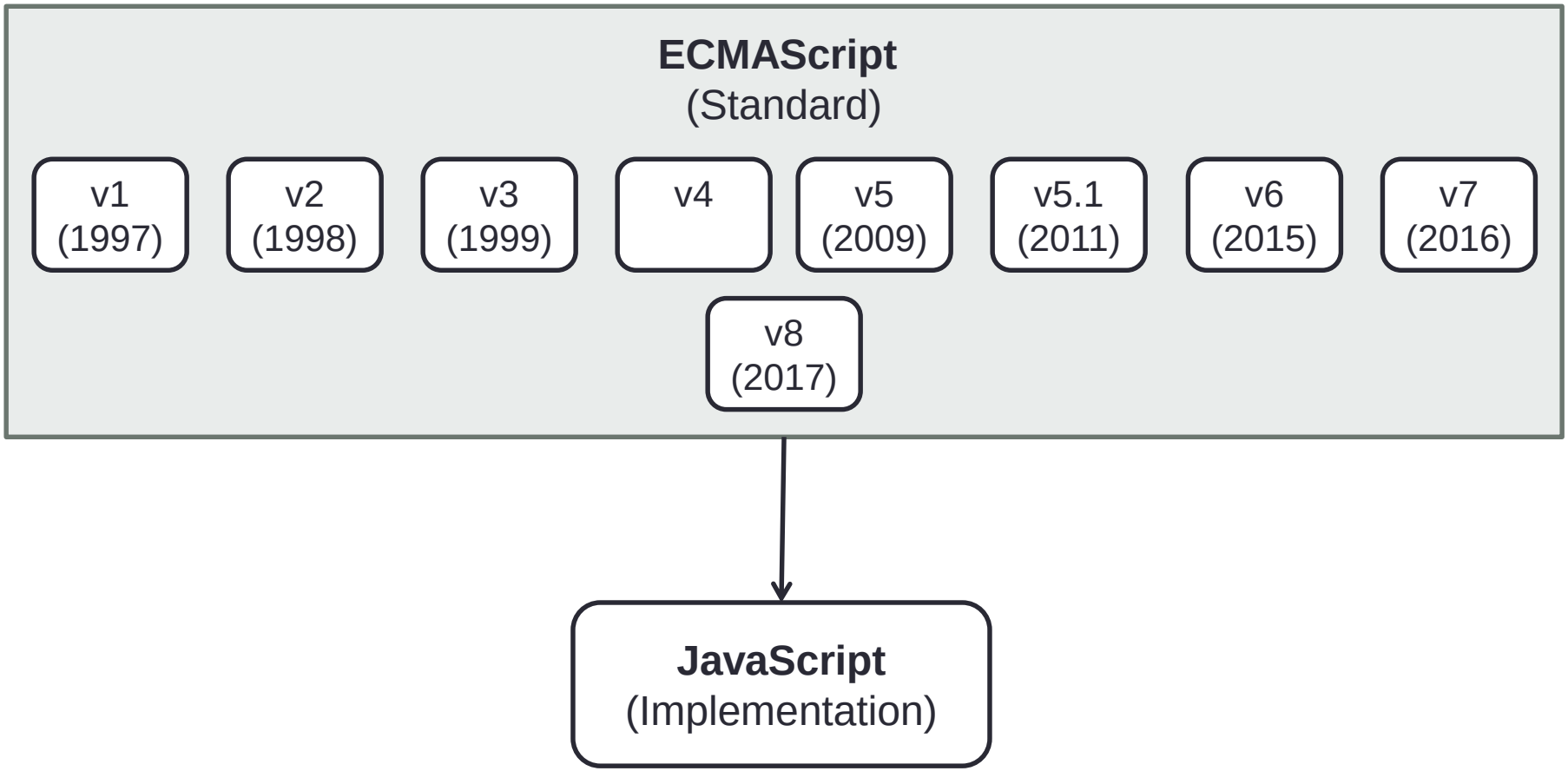
Agenda

- Introduction to ES6
- Array Helper Methods
- const & let
- Template Literals
- Arrow Functions
- Enhanced Object Literals
- Default Function Arguments
- Rest & Spread
- Destructuring
- Classes
- Generators
- Promises
- Fetch
- Q & A

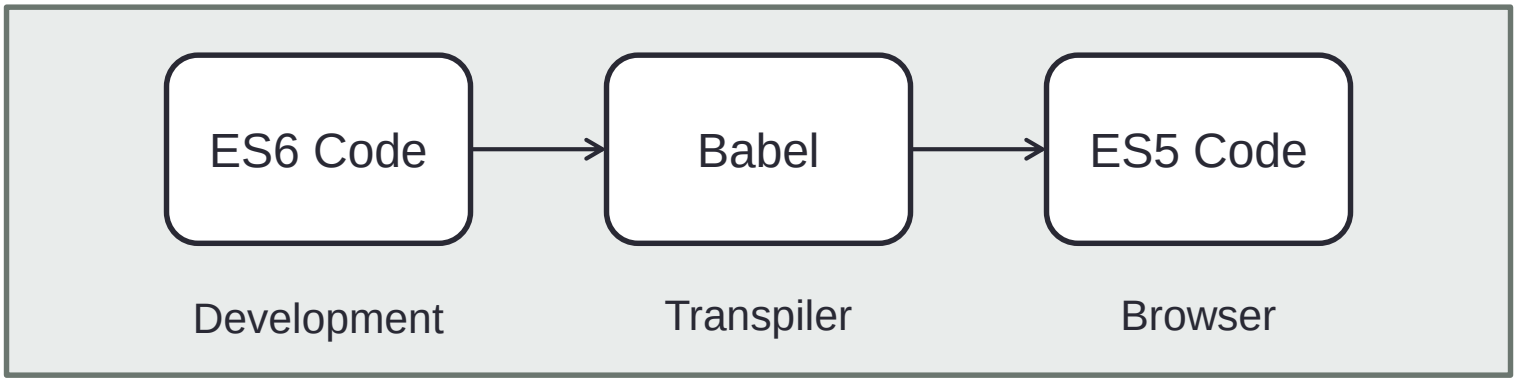
Introduction to ES6

- ECMAScript
  - Scripting language specification
  - First edition – published in 1997
  - ES5 – published in 2009
  - ES5.1 – 2011
  - ES6 – published in 2015, also called ES6 Harmony
    - Added significant new syntax for writing complex apps, including classes and modules
  - ES8 – finalized in 2017
- JavaScript
  - Implementation of ECMAScript
- Write faster, cleaner, expressive and more efficient code

Introduction to ES6



Introduction to ES6



ES6 Compatibility Table
https://kangax.github.io/compat-table/es6/

Array Helper Methods

- Work with collections of data
- Work in much the same way
- Themed around avoiding writing classic ‘for’ loop
  - forEach()
  - map()
  - filter()
  - find()
  - every()
  - some()
  - reduce()

forEach()

- Executes a provided function once for each array element
  - array.forEach(iterator-function)
- Less code compared to ‘for’ loop

```
var names = [‘Hari’, ‘Krishna’, ‘Shiva’];

names.forEach(function(name) {
   console.log(name);
});
```

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

var sum = 0;
numbers.forEach(function(number) {
   sum += number;
});

console.log(sum);
```

map()

- Creates a new array with the results of calling a provided function on every element in the calling array

```
var numbers = [10, 20, 30];

var doubledNumbers = numbers.map(function(number) {
   return number * 2;
});

console.log(doubledNumbers);
```

```
var images = [
  { height: '34px', width: '39px' },
  { height: '54px', width: '19px' },
  { height: '83px', width: '75px' }
];

var heights = images.map(function(image) {
    return image.height;
});

console.log(heights);
```

filter()

- Creates a new array with all elements that pass the test implemented by the provided function

```
var products = [
   { name: 'Nikon Coolpix A10', type: 'Camera'},
   { name: 'Sony DSC W830', type: 'Camera'},
   { name: 'Apple MacBook Air', type: 'Laptop'},
   { name: 'Dell Inspiron', type: 'Laptop'}
];

var filteredProducts = products.filter(function(product) {
   return product.type === 'Laptop';
});

console.log(filteredProducts);
```

find()

- Returns the value of the first element in the array that satisfies the provided testing function. Otherwise undefined is returned

```
var accounts = [
  { id: 1, balance: -10 },
  { id: 2, balance: 12 },
  { id: 3, balance: 0 }
];

var account = accounts.find(function(account) {
    return account.id === 2;
});

console.log(account);
```

every()

- Tests whether all elements in the array pass the test implemented by the provided function
- Condense an array into a single value, for e.g., a boolean or a number

```
var computers = [
  { name: 'Apple', ram: 32 },
  { name: 'Dell', ram: 16 },
  { name: 'Acer', ram: 4 },
  { name: 'HP', ram: 24 }
];

var everyComputerIsEligible = computers.every(function(computer) {
    return computer.ram > 16;
});

console.log(everyComputerIsEligible);
```

some()

- Tests whether at least one element in the array passes the test implemented by the provided function

```
var requests = [
  { url: '/photos', status: 'complete' },
  { url: '/albums', status: 'pending' },
  { url: '/users', status: 'failed' }
];

var inProgress = requests.some(function(request) {
    return request.status === 'pending';
});

console.log(inProgress);
```

reduce()

- Applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value

```
var trips = [
   { distance: 34 },
   { distance: 12 },
   { distance: 1 }
];

var totalDistance = trips.reduce(function(acc, trip) {
    return acc += trip.distance;
}, 0);

console.log(totalDistance);
```

const & let

- let
  - declares a block scope local variable

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

```
const name = 'Hari';
let age = 25;
const dateOfBirth = '10/10/2017';
```

Template Literals

- String literals allowing embedded expressions
- Enclosed by the back-tick (```)
- Can contain place holders (${expression})

Example #1

```
function doubleMessage(number) {
  return `Your number doubled is ${number * 2}`;
}
```

Example #2

```
function fullName(firstName, lastName) {
  return `${firstName} ${lastName}`;
}
```

Arrow Functions

- Very similar to regular functions in behavior, but are quite different syntactically
- Shorter syntax
- Does not have its own ‘this’; the this value of the enclosing execution context is used

```
var materials = [
  'Hydrogen',
  'Helium',
  'Lithium',
  'Beryllium'
];

materials.map(function(material) {
  return material.length;
}); // [8, 6, 7, 9]

materials.map((material) => {
  return material.length;
}); // [8, 6, 7, 9]

materials.map(material => material.length); // [8, 6, 7, 9]
```

Enhanced Object Literals

```
function CreateProductStore(inventory) {
   return {
      inventory,
      inventoryValue() {
         return this.inventory.reduce(
            (total, product) => total + product.price, 0);
      },
      getPrice(productId) {
         return this.inventory.find(
            product => product.id === productId).price;
      }
   };
}

const inventory = [
  { id: 1, name: 'Bahubali DVD', price: 399 },
  { id: 2, name: 'Timex watch', price: 1249 }
];

const myProductStore = CreateProductStore(inventory);

myProductStore.inventoryValue();
myProductStore.getPrice(2);
```

Default Function Arguments

```
function calculateBill(total, tax, tip) {
   if(tax === undefined) {
     tax = 0.13;
   }

   if(tip === undefined) {
     tip = 0.15;
   }

   return total + (total * tax) + (total * tip);
}

const totalBill = calculateBill(100);
console.log(totalBill);
```

```
function calculateBill(total, tax = 0.13, tip = 0.15) {
   return total + (total * tax) + (total * tip);
}

const totalBill = calculateBill(100);
console.log(totalBill);
```

Rest & Spread

- The rest parameter syntax allows us to
  - represent an indefinite number of arguments as an array
  - gather variables together
- Rest collects multiple elements and 'condenses' them into a single element

```
function addNumbers(...numbers) {
   return numbers.reduce((total, number) => {
      return total + number;
   }, 0);
}

console.log(addNumbers(1, 2, 3));
console.log(addNumbers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
```

Rest & Spread

- Spread syntax allows array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected
- Spread 'expands' an array into its elements

```
var dateFields = [1970, 0, 1];  // 1 Jan 1970
var d = new Date(...dateFields);
console.log(d);
```

```
var parts = ['shoulders', 'knees'];
var lyrics = ['head', ...parts, 'and', 'toes'];
// ["head", "shoulders", "knees", "and", "toes"]
```

```
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
arr1 = [...arr1, ...arr2];
```

Destructuring

- A JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables

```
// Example 1
const companies = [ 'Google', 'Facebook', 'Infosys', 'TCS' ];

let company1, company2, rest;

[company1, company2] = companies;
console.log(company1);
console.log(company2);

[company1, company2, ...rest] = companies;
console.log(company1);
console.log(company2);
console.log(rest);
```

Destructuring

```
// Example 2
let customer = {
   name: 'Hari',
   email: 'hari@xyz.com',
   phone: '+91-90000-80000'
};
let { name, email } = customer;
console.log(name);
console.log(email);
```

```
// Example 3
let fileInfo = {
   name: 'my-profile',
   extension: 'jpg',
   size: 25012
};

function fileSummary({ name, extension, size }) {
   return `The file '${name}.${extension}' is of size ${size} bytes`;
}

console.log(fileSummary(fileInfo));
```

Classes

- JavaScript is not a class-based language
- In JavaScript
  - we use functions to create objects
  - to inherit data and functionality, we use prototypal inheritance
- ES6 introduces new keywords
  - class
  - super
  - extends
- JavaScript still uses functions and prototypal inheritance under the hood
- JavaScript classes are just a thin mirage over regular functions and prototypal inheritance

Classes

```
// ES5
// constructor function
function Plane(numEngines) {
  this.numEngines = numEngines;
  this.enginesActive = false;
}

// methods "inherited" by all instances of Plane
Plane.prototype.startEngines = function () {
  console.log('starting engines...');
  this.enginesActive = true;
};

const civilPlane = new Plane(2);
civilPlane.startEngines();

const fighterPlane = new Plane(4);
fighterPlane.startEngines();
```

Classes

```
// ES6
class Plane {
  constructor(numEngines) {
    this.numEngines = numEngines;
    this.enginesActive = false;
  }

  startEngines() {
    console.log('starting engines…');
    this.enginesActive = true;
  }
}

const civilPlane = new Plane(2);
civilPlane.startEngines();

const fighterPlane = new Plane(4);
fighterPlane.startEngines();
```

Classes

- constructor()
  - A new, special method in a class
  - Used to initialize new objects
- Benefits
  - Less code
  - Cleaner code
  - Clearly defined constructor function
  - All code that's needed for the class is contained in the class declaration
- Points to be noted
  - Class is just a function
  - Under the hood, a class just uses prototypal inheritance
  - Using classes requires the use of new keyword

Classes

- Static methods
  - The keyword static is placed in front of the method name

```
// Static methods
class Plane {
  constructor(numEngines) {
    this.numEngines = numEngines;
    this.enginesActive = false;
  }

  startEngines() {
    console.log('starting engines…');
    this.enginesActive = true;
  }

 static badWeather(planes) {
    for (plane of planes) {
      plane.enginesActive = false;
    }
  }
}
```

```
// Calling a static method
Plane.badWeather([
   plane1,
   plane2,
   plane3
]);
```

Classes

- Sub classing with extends

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(this.name +
        ' makes a noise.');
  }
}
```

```
class Dog extends Animal {
  constructor(name, color = 'white') {
    super(name);
    this.color = color;
  }

  speak() {
    console.log(this.name + ' barks.');
  }
}
```

```
let a = new Animal('Snowy');
a.speak();

let d = new Dog('Tommy');
d.speak();

let l = new Lion('Leo');
l.speak();
```

```
class Lion extends Animal {
  constructor(name, color = 'ochre') {
    super(name);
    this.color = color;
  }

  speak() {
    console.log(this.name + ' roars.');
  }
}
```

Classes

- Working with subclasses
  - Less setup code
  - Cleaner syntax
  - In a subclass constructor function, before this can be used, a call to the super class must be made

Generators

- A function that can be exited and later re-entered
- Their context will be saved across re-entrances
- Defined using *function** declaration

How does it work?

- Calling a generator function does not execute its body immediately
- Instead, an iterator object is returned
- The iterator's ***next()*** method is called, the generator function's body is executed until the first ***yield*** expression
- ***yield*** expression can optionally specify the value to be returned from the iterator
- The ***next()*** method returns an object with
  - a ***value*** property containing the yielded value
  - a ***done*** boolean property which indicates whether the generator has yielded its last value
- Calling the ***next()*** method with an argument will resume the generator function execution, replacing the ***yield*** expression where execution was paused with the argument from next()
- A ***return*** statement in a generator, when executed, will make the generator ***done***

Generators

- We can use generators to iterate through any data structure that we want
- The yield keyword is used to pause a generator and used to send data outside of the generator
- The .next() method is used to pass data into the generator

Promises

- The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value
- Will let you start some work that will be done asynchronously and let you get back to your regular work
- Natively implemented in ES6
- Promise states
  - pending – waiting for long running task to get over
  - resolved – task finished and it all went ok
  - rejected – task finished but something went wrong
- resolve(value)
  - should be called when the request completes successfully
- reject(reason)
  - should be used when the request could not be completed

Promises

- then(onFulfilled, onRejected)
  - Appends fulfillment and rejection handlers to the promise
- catch(onRejected)
  - Appends a rejection handler callback to the promise

```
function myAsyncFunction(url) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest();
    xhr.open("GET", url);
    xhr.onload = () => resolve(xhr.responseText);
    xhr.onerror = () => reject(xhr.statusText);
    xhr.send();
  });
}

const promise = myAsyncFunction('https://xyz.com')
promise
   .then((data) => console.log(data))
   .catch((reason) => console.log(reason));
```

Fetch

- Provides an easy, logical way to fetch resources asynchronously across the network
- Promise based

Q & A

- Thank you!