ES6+

Javascript as language is evolving rapidly, most significant release was ECMA 2015 which is also called as ES6.

LANGUAGE UPDATES

- ES6 (ECMA 2015)
- ES7 (ECMA 2016)
- ES8 (ECMA 2017)
- ...
- ES11 (ECMA 2020)
- ESNext

NEW FEATURES

- Variables
- Block Scope
- Arrow functions
- Default Parameters

- Spread and Rest operators
- Template strings
- Object Destructuring
- Modules
- Classes

- Iterators and Generators
- Collections
- New methods for built in classes
- Promises

VARIABLES

The language has added new variable type declarations such as 'let' and 'const' these are block scoped variables

WHAT ARE LET AND COST?

let Holds re-assignable value in block scope

const Holds read only values. here read only means that the value of variable can not be re-assigned

```
let counter = 10;

const maxCount = 100;

const calculate = function() {
  // function body
}
```

WHAT ARE LET AND COST?

let Holds re-assignable value in block scope

const Holds read only values. here read only means that the value of variable can not be re-assigned

```
let counter = 10;

const maxCount = 100;

const calculate = function() {
  // function body
}
```

WHAT ARE LET AND COST?

let Holds re-assignable value in block scope

const Holds read only values. here read only means that the value of variable can not be re-assigned

```
let counter = 10;

const maxCount = 100;

const calculate = function() {
  // function body
}
```

BLOCK SCOPE

ES6 provides new way of declaring variables using **let** keyword unlike **var** which is hoisted at top, '**let**' is in block level scope

scope can be denoted by curly braces { }

```
let x = 10;
if (x == 10) {
    let x = 20;
    console.log(x); // 20:  reference x inside the block
}
```

BLOCK SCOPE

ES6 provides new way of declaring variables using **let** keyword unlike **var** which is hoisted at top, '**let**' is in block level scope

scope can be denoted by curly braces { }

```
let x = 10;
if (x == 10) {
    let x = 20;
    console.log(x); // 20:  reference x inside the block
}
```

BLOCK SCOPE

ES6 provides new way of declaring variables using **let** keyword unlike **var** which is hoisted at top, '**let**' is in block level scope

scope can be denoted by curly braces { }

```
let x = 10;
if (x == 10) {
    let x = 20;
    console.log(x); // 20:  reference x inside the block
}
```

ARROW FUNCTIONS

Writing less with running scope of parent

```
1  const getAccountById = accountId => {
2
3  }
4
5  const getAccounts = () => {
6
7  }
8
9  const getGreeting = (firstName, lastName) = {
10
11 }
```

ARROW FUNCTIONS

Writing less with running scope of parent

```
1  const getAccountById = accountId => {
2
3  }
4
5  const getAccounts = () => {
6
7  }
8
9  const getGreeting = (firstName, lastName) = {
10
11 }
```

ARROW FUNCTIONS

Writing less with running scope of parent

```
1  const getAccountById = accountId => {
2
3  }
4
5  const getAccounts = () => {
6
7  }
8
9  const getGreeting = (firstName, lastName) = {
10
11 }
```

Prior to ES6 (that = this)

```
var that = this;
const getAccountById = function(accountId) {
  var params = {
    success: function() {},
    error: function() {}
  }
  that.service.getAccountById(accountId, params);
};
```

DEFAULT PARAMETERS

You can have default values for your function arguments

```
1 const setLanguage = (language = 'en') => {
2   console.log("Language?", language);
3 }
4
5 setLanguage() // prints en;
6 setLanguage("es") // prints es
```

DEFAULT PARAMETERS

You can have default values for your function arguments

```
1 const setLanguage = (language = 'en') => {
2   console.log("Language?", language);
3 }
4
5 setLanguage() // prints en;
6 setLanguage("es") // prints es
```

DEFAULT PARAMETERS

You can have default values for your function arguments

```
1 const setLanguage = (language = 'en') => {
2   console.log("Language?", language);
3 }
4
5 setLanguage() // prints en;
6 setLanguage("es") // prints es
```

SPREAD AND REST OPERATORS

ES6 provides new operator `...` which is used with spread and rest parameters

Spread This is useful in copying and creating new objects

```
const savingAccounts = [sa1, sa2, sa3, sa4];
const loanAccounts = [la1, la2, la3, la4];
const accounts = [...savingAccounts, ...loanAccounts];

const user = {
  firstName: '',
  lastName: ''
};
const address = {
  addressLine1: "",
  city: "",
  state: "",
  zip: "",
  country: ""
};
```

Spread This is useful in copying and creating new objects

```
const savingAccounts = [sa1, sa2, sa3, sa4];
const loanAccounts = [la1, la2, la3, la4];
const accounts = [...savingAccounts, ...loanAccounts];

const user = {
  firstName: '',
  lastName: ''
};
const address = {
  addressLine1: "",
  city: "",
  state: "",
  zip: "",
  country: ""
};
```

Spread This is useful in copying and creating new objects

```
const savingAccounts = [sa1, sa2, sa3, sa4];
const loanAccounts = [la1, la2, la3, la4];
const accounts = [...savingAccounts, ...loanAccounts];

const user = {
  firstName: '',
  lastName: ''
};
const address = {
  addressLine1: "",
  city: "",
  state: "",
  zip: "",
  country: ""
};
```

Spread This is useful in copying and creating new objects

```
const savingAccounts = [sa1, sa2, sa3, sa4];
const loanAccounts = [la1, la2, la3, la4];
const accounts = [...savingAccounts, ...loanAccounts];

const user = {
  firstName: '',
  lastName: ''
};
const address = {
  addressLine1: "",
  city: "",
  state: "",
  zip: "",
  country: ""
};
```

Spread This is useful in copying and creating new objects

```
3  const accounts = [...savingAccounts, ...loanAccounts];
4
5  const user = {
6    firstName: '',
7    lastName: ''
8  };
9  const address = {
10   addressLine1: "",
11   city: "",
12   state: "",
13   zip: "",
14   country: ""
15 };
16
17 const userDetails = {...user, ...address};
```

Spread This is useful in copying and creating new objects

```
3  const accounts = [...savingAccounts, ...loanAccounts];
4
5  const user = {
6    firstName: '',
7    lastName: ''
8  };
9  const address = {
10   addressLine1: "",
11   city: "",
12   state: "",
13   zip: "",
14   country: ""
15 };
16
17 const userDetails = {...user, ...address};
```

Rest Helps in aggregating parameters in single argument

```
const sum = (number1, number, ...numbers) => {
  //body
}
sum(5, 10);
sum(5, 10, 15);

const marks = [65, 78, 98, 75];
sum(5, 10, ...marks);
```

Rest Helps in aggregating parameters in single argument

```
const sum = (number1, number, ...numbers) => {
  //body
}
sum(5, 10);
sum(5, 10, 15);

const marks = [65, 78, 98, 75];
sum(5, 10, ...marks);
```

Rest Helps in aggregating parameters in single argument

```
const sum = (number1, number, ...numbers) => {
  //body
}
sum(5, 10);
sum(5, 10, 15);

const marks = [65, 78, 98, 75];
sum(5, 10, ...marks);
```

TEMPLATE STRING

Template string allows to manipulate string in javascript much cleaner way, string can be substituted, can be scaped and also can be written much readable way

```
const greetingComponent = (firstName, lastName) => {
  return `
    <div>
      <h2>Hello ${firstName},  ${lastName}</h2><h2>
      <span>${new Date().toString()}
    </span></h2></div>
  `;
}
```

OBJECT DESTRUCTURING

Helps to define and assign values from object with matching keys automatically.

Example prior to ES6

```
1 function displayAccountDetails (account) {
2   var accountId = account.accountId;
3   var balance = account.balance;
4   var currency = account.currency;
5
6   console.log("Account Details", accountId, balance, currenc
7 }
```

How can we write this in better way using ES6?

```
1 function displayAccountDetails (account) {
2   const {accountId, balance, currency} = account;
3   console.log("Account Details", accountId, balance, currenc
4 }
```

MODULES

Module allows us to write better maintainable code in separate files there are module patterns like CommonJs, AMD etc

ES6 allows us define our own modules, it can specify its dependencies and also export objects to its bindings

> *"UI5 uses AMD pattern"*

COMMONJS

```
var accountsService = require("./AccountsService.js");
var forexService = require("./ForexService.js");
var util = require("./Util.js");

// Module definition
...
```

AMD (ASYNCHRONOUS MODULE DEFINITION)

```
define("TransferController",
  ["AccountsService", "ForexService", "Util"],
  function (AccountsService, ForexService, Util) {
    // Module Definition
  ...

  });
```

ES6 MODULES

```
1  // Example of Transfer Service Module
2  import accountService from "./accounts/service/AccountServi
3  import forexService from "./forex/service/ForexService";
4  import { formDate, formatCurrency } from "./common/util/Uti
5
6  const TransferService = function () {
7    // Module definition
8  };
9
10 export default TransferService;
```

ES6 MODULES

```
1  // Example of Transfer Service Module
2  import accountService from "./accounts/service/AccountServi
3  import forexService from "./forex/service/ForexService";
4  import { formDate, formatCurrency } from "./common/util/Uti
5
6  const TransferService = function () {
7    // Module definition
8  };
9
10 export default TransferService;
```

ES6 MODULES

```
1  // Example of Transfer Service Module
2  import accountService from "./accounts/service/AccountServi
3  import forexService from "./forex/service/ForexService";
4  import { formDate, formatCurrency } from "./common/util/Uti
5
6  const TransferService = function () {
7    // Module definition
8  };
9
10 export default TransferService;
```

HOW DO WE BUNDLE?

Webpack

CLASSES

ES6 allows you to define classes and also supports inheritance.

Class Definition

```
class Product {
  constructor(id, name, title){
      this.id = id;
      this.name = name;
      this.title = title;
  }
  get productInfo(){
      return this.id + " - "+ this.name + " - " + this.titl
  }
}
let s1 = new Product(101,'CreditCard','Diners Club');
console.log(s1);
console.log(s1.productInfo);
```

Class Definition

```
1  class Product {
2    constructor(id, name, title){
3        this.id = id;
4        this.name = name;
5        this.title = title;
6    }
7    get productInfo(){
8        return this.id + " - "+ this.name + " - " + this.titl
9    }
10 }
11 let s1 = new Product(101,'CreditCard','Diners Club');
12 console.log(s1);
13 console.log(s1.productInfo);
```

Class Definition

```
1  class Product {
2    constructor(id, name, title){
3        this.id = id;
4        this.name = name;
5        this.title = title;
6    }
7    get productInfo(){
8        return this.id + " - "+ this.name + " - " + this.titl
9    }
10 }
11 let s1 = new Product(101,'CreditCard','Diners Club');
12 console.log(s1);
13 console.log(s1.productInfo);
```

Class Definition

```
1  class Product {
2    constructor(id, name, title){
3        this.id = id;
4        this.name = name;
5        this.title = title;
6    }
7    get productInfo(){
8        return this.id + " - "+ this.name + " - " + this.titl
9    }
10 }
11 let s1 = new Product(101,'CreditCard','Diners Club');
12 console.log(s1);
13 console.log(s1.productInfo);
```

Inheritance

```
class CreditCard extends Product {
  constructor(id, name, title, balance){
      super(arguments);
      this.balance = balance;
  }
  get balance(){
      return this.balance;
  }
}
let s1 = new CreditCard(101,'CreditCard','Diners Club', 120
console.log(s1.productInfo);
```

Inheritance

```
class CreditCard extends Product {
  constructor(id, name, title, balance){
      super(arguments);
      this.balance = balance;
  }
  get balance(){
      return this.balance;
  }
}
let s1 = new CreditCard(101,'CreditCard','Diners Club', 120
console.log(s1.productInfo);
```

Inheritance

```
1  class CreditCard extends Product {
2    constructor(id, name, title, balance){
3        super(arguments);
4        this.balance = balance;
5    }
6    get balance(){
7        return this.balance;
8    }
9  }
10 let s1 = new CreditCard(101,'CreditCard','Diners Club', 120
11 console.log(s1.productInfo);
```

ITERATORS

Iterators allow processing sequential data efficiently in javascript with ES6 we can make any collection iterable

```
class ProductList {
    constructor( products = [] ) {
        this.products = products;
        this.size = products.length;
    }
    [Symbol.iterator]() {
        let counter = 0;
        return  {
            next: () => {
                if ( counter <= this.size ) {
                    let result = {
                      value: this.products[counter],
                      done: false
                    }
                    counter++;
```

```
 6      [Symbol.iterator]() {
 7          let counter = 0;
 8          return  {
 9              next: () => {
10                  if ( counter <= this.size ) {
11                      let result = {
12                        value: this.products[counter],
13                        done: false
14                      }
15                      counter++;
16                      return result;
17                  }
18                  return { value: counter, done: true };
19              }
20          }
```

```
7           let counter = 0;
8           return  {
9               next: () => {
10                  if ( counter <= this.size ) {
11                      let result = {
12                        value: this.products[counter],
13                        done: false
14                      }
15                      counter++;
16                      return result;
17                  }
18                  return { value: counter, done: true };
19              }
20          }
```

```
 8          return  {
 9              next: () => {
10                  if ( counter <= this.size ) {
11                      let result = {
12                        value: this.products[counter],
13                        done: false
14                      }
15                      counter++;
16                      return result;
17                  }
18                  return { value: counter, done: true };
19              }
20          }

21      }
```

... finally we have iterable object!

```
1 // Instantiate the Iterable Object
2 const productList = new ProductList(["CreditCard", "FixedDep
3
4 // Iterate over Product List
5 for (product of productList) {
6   console.log(product);
7 }
```

... finally we have iterable object!

```
1 // Instantiate the Iterable Object
2 const productList = new ProductList(["CreditCard", "FixedDep
3
4 // Iterate over Product List
5 for (product of productList) {
6   console.log(product);
7 }
```

GENERATORS

"Pausable functions in javascript"

Generator functions returns Iterators. these functions stops when it encounters **yield** keyword and returns back to caller.

Defining random number generator

```
function* randomNumbers() {
  while (true) {
    let time = new Date().getTime();
    let nextNumber =
    Math.floor(Math.random() * Math.floor(time));

    yield nextNumber;
  };
}
```

Generator functions returns Iterators. these functions stops when it encounters **yield** keyword and returns back to caller.

Defining random number generator

```
function* randomNumbers() {
  while (true) {
    let time = new Date().getTime();
    let nextNumber =
    Math.floor(Math.random() * Math.floor(time));

    yield nextNumber;
  };
}
```

Generator functions returns Iterators. these functions stops when it encounters **yield** keyword and returns back to caller.

Defining random number generator

```
function* randomNumbers() {
  while (true) {
    let time = new Date().getTime();
    let nextNumber =
    Math.floor(Math.random() * Math.floor(time));

    yield nextNumber;
  };
}
```

Processing workflow which is asynchronous, in sequential manner!

```
1 // Call generator
2 const uniqueNumberGenerator = randomNumbers();
3
4 // Process stream of numbers
5 uniqueNumberGenerator.next().value;
6 uniqueNumberGenerator.next().value;
7 uniqueNumberGenerator.next().value;
```

Processing workflow which is asynchronous, in sequential manner!

```
1 // Call generator
2 const uniqueNumberGenerator = randomNumbers();
3
4 // Process stream of numbers
5 uniqueNumberGenerator.next().value;
6 uniqueNumberGenerator.next().value;
7 uniqueNumberGenerator.next().value;
```

Processing workflow which is asynchronous, in sequential manner!

```
1 // Call generator
2 const uniqueNumberGenerator = randomNumbers();
3
4 // Process stream of numbers
5 uniqueNumberGenerator.next().value;
6 uniqueNumberGenerator.next().value;
7 uniqueNumberGenerator.next().value;
```

Processing workflow which is asynchronous, in sequential manner!

```
1 // Call generator
2 const uniqueNumberGenerator = randomNumbers();
3
4 // Process stream of numbers
5 uniqueNumberGenerator.next().value;
6 uniqueNumberGenerator.next().value;
7 uniqueNumberGenerator.next().value;
```

COLLECTIONS

ES6 provides new collections like Set, WeakSet and Map, WeakMap

```
const status = new Set(["SUCCESS", "ERROR", "IN PROCESS"]);

let views = new Map();
   views.set('transfer', 'transfer.view.xml');
   views.set('payment', 'payment.view.xml');
   views.set('alerts', 'alerts.view.xml');
```

NEW BUILT-IN METHODS AND USEFUL API'S

```
Object.assign(dest, src1, src2);

[ 1, 3, 4, 2 ].find(x => x > 3) // 4

[ 1, 3, 4, 2 ].findIndex(x => x > 3) // 2

[ 1, 3, 4, 2 ].forEach(x => console.log(x));

[{id: 1000, bal: 1200}, {id: 1000, bal: 1200}]
  .map({id, bal} => `<li>${id}-${bal}</li>`);

[1, 60, 34, 30, 20, 5].filter(x => x < 20);

[1, 60, 34, 30, 20, 5].map(x => x * 2);
```

PROMISES

"Promises represents eventual completion or failure of task which is asynchronous"

Promises are very useful in handling asynchronous executions efficiently

Example prior to ES6

```
function validateTouchId(successFunction, errorFunction) {
  var success = function (result) {
    successFunction.apply(results);
  };

  var error = function (result) {
    errorFunction.apply(results);
  }

  // Touch Id Plugin
  TouchId.validate(success, error);
}

// Invocation of async function  and handling
```

Things do complicate even more when you call multiple async functions and nest

```
validateTouchId(function (){
  acceptTerms(function() {
    registerUser(function (){
      registerForPush(function() {
        // process next ?
        // The lost flow....
      }, function() {
        // handle Error
      })
    }, function () {
      // handle Error
    })
  }, function(){
    // handle Error
  })
```

HOW DO WE SIMPLIFY CALLBACK HELL?

"Use promises"

```
1 function validateTouchId() {
2   return Promise((resolve, reject) => {
3     TouchId.validate()
4       .then(result => resolve(result))
5       .catch(error => reject(error));
6   });
7 }
```

*"Wrap every async function in to **Promises** for better processing"*

HOW DO WE SIMPLIFY CALLBACK HELL?

"Use promises"

```
function validateTouchId() {
  return Promise((resolve, reject) => {
    TouchId.validate()
      .then(result => resolve(result))
      .catch(error => reject(error));
  });
}
```

*"Wrap every async function in to **Promises** for better processing"*

HOW DO WE SIMPLIFY CALLBACK HELL?

"Use promises"

```
1 function validateTouchId() {
2   return Promise((resolve, reject) => {
3     TouchId.validate()
4       .then(result => resolve(result))
5       .catch(error => reject(error));
6   });
7 }
```

*"Wrap every async function in to **Promises** for better processing"*

HOW DO WE SIMPLIFY CALLBACK HELL?

"Use promises"

```
1 function validateTouchId() {
2   return Promise((resolve, reject) => {
3     TouchId.validate()
4       .then(result => resolve(result))
5       .catch(error => reject(error));
6   });
7 }
```

*"Wrap every async function in to **Promises** for better processing"*

PROMISE CHAINING

```
acceptTerms()
  .then(function (result){
    return validateTouchId();
  })
  .then(function (result){
    return registerUser();
  })
  .then(function (result){
    return registerForPush();
  })
  .catch(function (error){
    // handle Error
  })
```

PROMISE CHAINING

```
acceptTerms()
  .then(function (result){
    return validateTouchId();
  })
  .then(function (result){
    return registerUser();
  })
  .then(function (result){
    return registerForPush();
  })
  .catch(function (error){
    // handle Error
  })
```

PROMISE CHAINING

```
1   acceptTerms()
2     .then(function (result){
3       return validateTouchId();
4     })
5     .then(function (result){
6       return registerUser();
7     })
8     .then(function (result){
9       return registerForPush();
10    })
11    .catch(function (error){
12      // handle Error
13    })
```

PROMISE CHAINING

```
1   acceptTerms()
2     .then(function (result){
3       return validateTouchId();
4     })
5     .then(function (result){
6       return registerUser();
7     })
8     .then(function (result){
9       return registerForPush();
10    })
11    .catch(function (error){
12      // handle Error
13    })
```

PROMISE CHAINING

```
1   acceptTerms()
2     .then(function (result){
3       return validateTouchId();
4     })
5     .then(function (result){
6       return registerUser();
7     })
8     .then(function (result){
9       return registerForPush();
10    })
11    .catch(function (error){
12      // handle Error
13    })
```

PROMISE CHAINING

```
1   acceptTerms()
2     .then(function (result){
3       return validateTouchId();
4     })
5     .then(function (result){
6       return registerUser();
7     })
8     .then(function (result){
9       return registerForPush();
10    })
11    .catch(function (error){
12      // handle Error
13    })
```

PROMISE CHAINING

```
1   acceptTerms()
2     .then(function (result){
3       return validateTouchId();
4     })
5     .then(function (result){
6       return registerUser();
7     })
8     .then(function (result){
9       return registerForPush();
10    })
11    .catch(function (error){
12      // handle Error
13    })
```

PROMISE CHAINING

```
1   acceptTerms()
2     .then(function (result){
3       return validateTouchId();
4     })
5     .then(function (result){
6       return registerUser();
7     })
8     .then(function (result){
9       return registerForPush();
10    })
11    .catch(function (error){
12      // handle Error
13    })
```

PROMISE CHAINING

```
1   acceptTerms()
2     .then(function (result){
3       return validateTouchId();
4     })
5     .then(function (result){
6       return registerUser();
7     })
8     .then(function (result){
9       return registerForPush();
10    })
11    .catch(function (error){
12      // handle Error
13    })
```

CONCURRENT EXECUTIONS

```
1   Promise.all([
2     getFromAccounts(),
3     getToAccounts(),
4     getPayees()
5   ])
6   .then((results) => {
7     // process results
8   })
9   .catch((error) => {
10    // handle error
11  });
```

CONCURRENT EXECUTIONS

```
1   Promise.all([
2     getFromAccounts(),
3     getToAccounts(),
4     getPayees()
5   ])
6   .then((results) => {
7     // process results
8   })
9   .catch((error) => {
10    // handle error
11  });
```

CONCURRENT EXECUTIONS

```
Promise.all([
  getFromAccounts(),
  getToAccounts(),
  getPayees()
])
.then((results) => {
  // process results
})
.catch((error) => {
  // handle error
});
```

ASYNC/AWAIT

"Handle your asynchronous execution in better, readable way using async/await"

HOW TO MAKE FUNCTION ASYNC?

```
1 async function validateTouchId() {
2     TouchId.validate()
3       .then(result => return(result))
4       .catch(error => throw Error(error));
5 }
```

HOW TO MAKE FUNCTION ASYNC?

```
1 async function validateTouchId() {
2     TouchId.validate()
3       .then(result => return(result))
4       .catch(error => throw Error(error));
5 }
```

HOW TO MAKE FUNCTION ASYNC?

```
1 async function validateTouchId() {
2     TouchId.validate()
3       .then(result => return(result))
4       .catch(error => throw Error(error));
5 }
```

HOW DO WE IMPROVE ON PROMISE CHAINING?

```
function onBoardUser() {
  try {
    await acceptTerms();
    await validateTouchId();
    await registerUser();
    await registerForPush();
  } catch(error) {
    // One handler for errors
  }
}
```

HOW DO WE IMPROVE ON PROMISE CHAINING?

```
function onBoardUser() {
  try {
    await acceptTerms();
    await validateTouchId();
    await registerUser();
    await registerForPush();
  } catch(error) {
    // One handler for errors
  }
}
```

HOW DO WE IMPROVE ON PROMISE CHAINING?

```
function onBoardUser() {
  try {
    await acceptTerms();
    await validateTouchId();
    await registerUser();
    await registerForPush();
  } catch(error) {
    // One handler for errors
  }
}
```

HOW DO WE IMPROVE ON PROMISE CHAINING?

```
1  function onBoardUser() {
2    try {
3      await acceptTerms();
4      await validateTouchId();
5      await registerUser();
6      await registerForPush();
7    } catch(error) {
8      // One handler for errors
9    }
10 }
```

HOW DO WE IMPROVE ON PROMISE CHAINING?

```
1  function onBoardUser() {
2    try {
3      await acceptTerms();
4      await validateTouchId();
5      await registerUser();
6      await registerForPush();
7    } catch(error) {
8      // One handler for errors
9    }
10 }
```

HOW DO WE IMPROVE ON PROMISE CHAINING?

```
function onBoardUser() {
  try {
    await acceptTerms();
    await validateTouchId();
    await registerUser();
    await registerForPush();
  } catch(error) {
    // One handler for errors
  }
}
```

"Async/Await are syntactic sugar around Promises, makes asynchronous code more readable and sequential in nature"

DEMO

Async/Await

Generators

RESOURCES

- The modern JavaScript
- Compatibility
- Babel
- ESLint