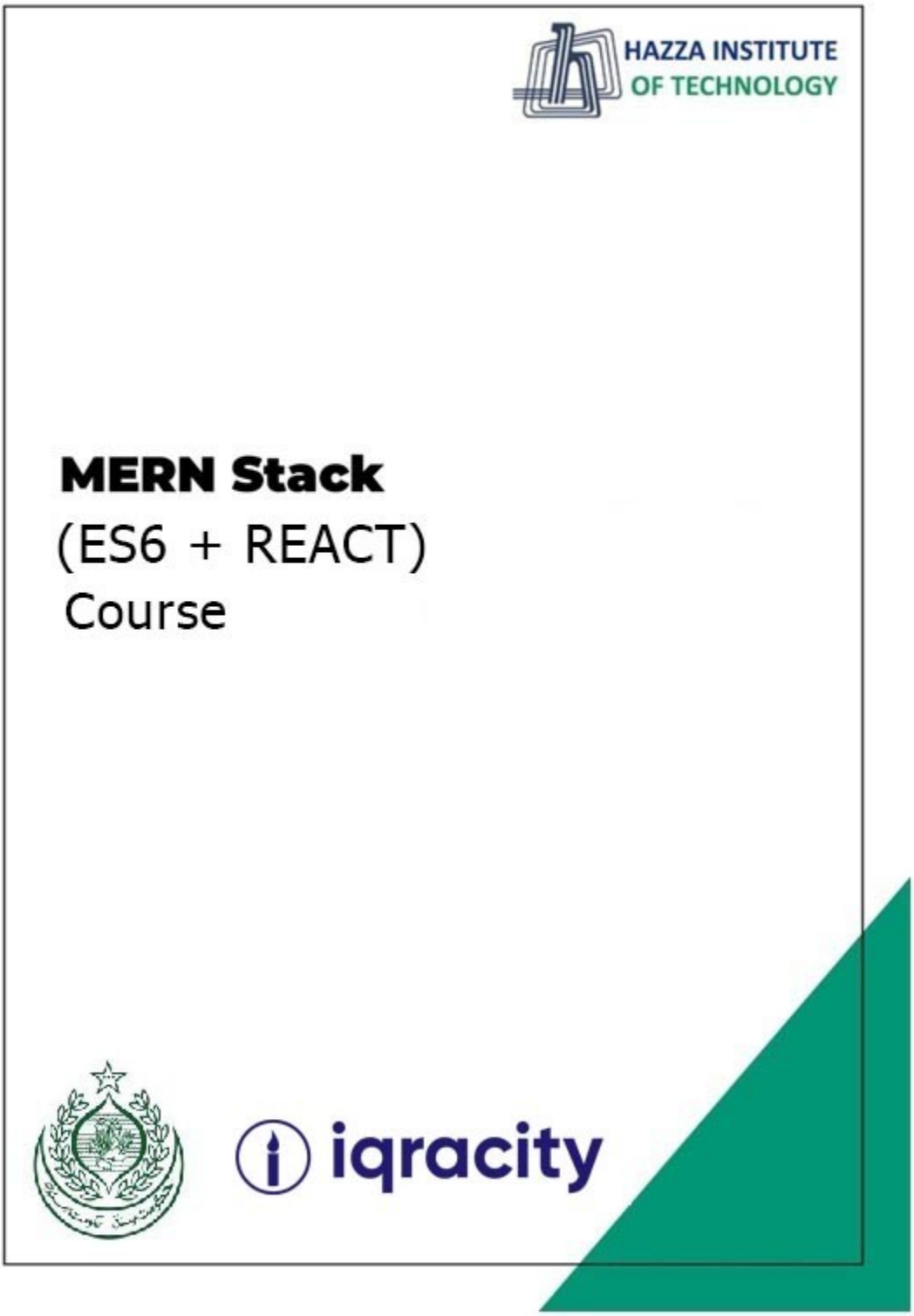
HAZZA INSTITUTE OF TECHNOLOGY

iqracity



Lab 9

Total Time:
3 hours

Pre-Lab Activities:

- Have a good knowledge of JavaScript up to ES5.

Learning Outcomes:

- Have a good understanding of Javascript and ES6

Lab Tasks:

- New ES6 Syntax
- Destructuring
- ES6 Modules
- ES6 Classes
- Arrow Functions
- Symbol
- Iterators & Generators

Student Activities:

- Explore New ES6 Syntax
- Explore Destructuring
- Explore ES6 Modules
- Explore ES6 Classes
- Explore Arrow Functions
- Explore Symbol
- Explore Iterators & Generators

Lab Solution

Section 1. New ES6 syntax

Introduction to the JavaScript let keyword

In ES5, when you declare a variable using the var keyword, the scope of the variable is either global or local. If you declare a variable outside of a function, the scope of the variable is global. When you declare a variable inside a function, the scope of the variable is local.

ES6 provides a new way of declaring a variable by using the let keyword. The let keyword is similar to the var keyword, except that these variables are blocked-scope. For example:

```
let variable_name;
```
Code language: JavaScript (javascript)

In JavaScript, blocks are denoted by curly braces {} , for example, the if else, for, do while, while, try catch and so on:

```
if(condition) {
  // inside a block
}
```
Code language: JavaScript (javascript)

See the following example:

```
let x = 10;
if (x == 10) {
   let x = 20;
   console.log(x); // 20:  reference x inside the block
}
console.log(x); // 10: reference at the begining of the script
```
Code language: JavaScript (javascript)

How the script works:

- First, declare a variable x and initialize its value to 10.
- Second, declare a new variable with the same name x inside the if block but with an initial value of 20.
- Third, output the value of the variable x inside and after the if block.

Because the let keyword declares a block-scoped variable, the x variable inside the if block is a **new variable** and it shadows the x variable declared at the top of the script. Therefore, the value of x in the console is 20.

When the JavaScript engine completes executing the if block, the x variable inside the if block is out of scope. Therefore, the value of the x variable that following the if block is 10.

JavaScript let and global object

When you declare a global variable using the var keyword, you add that variable to the property list of the global object. In the case of the web browser, the global object is the window. For example:

```
var a = 10;
console.log(window.a); // 10
```
Code language: JavaScript (javascript)

However, when you use the let keyword to declare a variable, that variable is **not** attached to the global object as a property. For example:

```
let b = 20;
console.log(window.b); // undefined
```
Code language: JavaScript (javascript)

JavaScript let and callback function in a for loop

See the following example.

```
for (var i = 0; i < 5; i++) {
   setTimeout(function () {
      console.log(i);
   }, 1000);
}
```
Code language: JavaScript (javascript)

The intention of the code is to output numbers from 0 to 4 to the console every second. However, it outputs the number 5 five times:

```
5
5
5
5
5
```

In this example, the variable i is a global variable. After the loop, its value is 5. When the callback functions are passed to the setTimeout() function executes, they reference the same variable i with the value 5.

In ES5, you can fix this issue by creating another scope so that each callback function references a new variable. And to create a new scope, you need to create a function. Typically, you use the IIFE pattern as follows:

```
for (var i = 0; i < 5; i++) {
   (function (j) {
      setTimeout(function () {
         console.log(j);
      }, 1000);
   })(i);
}
```
Code language: JavaScript (javascript)

Output:

```
0
1
2
3
4
```

In ES6, the let keyword declares a new variable in each loop iteration. Therefore, you just need to replace the var keyword with the let keyword to fix the issue:

```
for (let i = 0; i < 5; i++) {
   setTimeout(function () {
      console.log(i);
   }, 1000);
}
```
Code language: JavaScript (javascript)

To make the code completely ES6 style, you can use an arrow function as follows:

```
for (let i = 0; i < 5; i++) {
   setTimeout(() => console.log(i), 1000);
}
```
Code language: JavaScript (javascript)

Note that you'll learn more about the arrow functions in the later tutorial.

Redeclaration

The var keyword allows you to redeclare a variable without any issue:

```
var counter = 0;
var counter;
console.log(counter); // 0
```
Code language: JavaScript (javascript)

However, redeclaring a variable using the let keyword will result in an error:

```
let counter = 0;
let counter;
console.log(counter);
```
Code language: JavaScript (javascript)

Here's the error message:

```
Uncaught SyntaxError: Identifier 'counter' has already been declared
```
Code language: JavaScript (javascript)

JavaScript let variables and hoisting

Let's examine the following example:

```
{
  console.log(counter); //
  let counter = 10;
}
```
Code language: JavaScript (javascript)

This code causes an error:

```
Uncaught ReferenceError: Cannot access 'counter' before initialization
```
Code language: JavaScript (javascript)

In this example, accessing the counter variable before declaring it causes a ReferenceError. You may think that a variable declaration using the let keyword does not **hoist,** but it does**.**

In fact, the JavaScript engine will hoist a variable declared by the let keyword to the top of the block. However, the JavaScript engine does not initialize the variable. Therefore, when you reference an uninitialized variable, you'll get a ReferenceError.

Temporal death zone (TDZ)

A variable declared by the let keyword has a so-called temporal dead zone (TDZ). The TDZ is the time from the start of the block until the variable declaration is processed.

The following example illustrates that the temporal dead zone is time-based, not location-based.

```
{ // enter new scope, TDZ starts
  let log = function () {
    console.log(message); // messagedeclared later
  };

  // This is the TDZ and accessing log
```

```
    // would cause a ReferenceError

    let message= 'Hello'; // TDZ ends
    log(); // called outside TDZ
}
```

Code language: JavaScript (javascript)

In this example:

First, the curly brace starts a new block scope, therefore, the TDZ starts.

Second, the log() function expression accesses the message variable. However, the log() function has not been executed yet.

Third, declare the message variable and initialize its value to 10. The time from the start of the block scope to the time that the message variable is accessed is called a *temporal death zone*. When the JavaScript engine processes the declaration, the TDZ ends.

Finally, call the log() function that accesses the message variable outside of the TDZ.

Note that if you access a variable declared by the let keyword in the TDZ, you'll get a ReferenceError as illustrated in the following example.

```
{ // TDZ starts
    console.log(typeof myVar); // undefined
    console.log(typeof message); // ReferenceError
    let message; // TDZ ends
}
```

Code language: JavaScript (javascript)

Notice that myVar variable is a non-existing variable, therefore, its type is undefined.

The temporal death zone prevents you from accidentally referencing a variable before its declaration.

Summary

- Variables are declared using the let keyword are block-scoped, are not initialized to any value, and are not attached to the global object.
- Redeclaring a variable using the let keyword will cause an error.
- A temporal dead zone of a variable declared using the let keyword starts from the block until the initialization is evaluated.

Introduction to the JavaScript const keyword

ES6 provides a new way of declaring a constant by using the const keyword. The const keyword creates a read-only reference to a value.

```
const CONSTANT_NAME = value;
```
Code language: JavaScript (javascript)

By convention, the constant identifiers are in uppercase.

Like the let keyword, the const keyword declares blocked-scope variables. However, the block-scoped variables declared by the const keyword can't be **reassigned**.

The variables declared by the let keyword are mutable. It means that you can change their values anytime you want as shown in the following example:

```
let a = 10;
a = 20;
a = a + 5;
console.log(a); // 25
```
Code language: JavaScript (javascript)

However, variables created by the const keyword are "immutable". In other words, you can't reassign them to different values.

If you attempt to reassign a variable declared by the const keyword, you'll get a TypeError like this:

```
const RATE = 0.1;
RATE = 0.2; // TypeError
```
Code language: JavaScript (javascript)

Unlike the let keyword, you need to initialize the value to the variable declared by the const keyword.

The following example causes a SyntaxError due to missing the initializer in the const variable declaration:

```
const RED; // SyntaxError
```
Code language: JavaScript (javascript)

JavaScript const and Objects

The const keyword ensures that the variable it creates is read-only. However, it doesn't mean that the actual value to which the const variable reference is immutable. For example:

```
const person = { age: 20 };
person.age = 30; // OK
console.log(person.age); // 30
```
Code language: JavaScript (javascript)

Even though the person variable is a constant, you can change the value of its property.

However, you cannot reassign a different value to the person constant like this:

```
person = { age: 40 }; // TypeError
```
Code language: JavaScript (javascript)

If you want the value of the person object to be immutable, you have to freeze it by using the Object.freeze() method:

```
const person = Object.freeze({age: 20});
person.age = 30; // TypeError
```
Code language: JavaScript (javascript)

Note that Object.freeze() is shallow, meaning that it can freeze the properties of the object, not the objects referenced by the properties.

For example, the company object is constant and frozen.

```
const company = Object.freeze({
   name: 'ABC corp',
   address: {
      street: 'North 1st street',
      city: 'San Jose',
      state: 'CA',
      zipcode: 95134
   }
});
```
Code language: JavaScript (javascript)

But the company.address object is not immutable, you can add a new property to the company.address object as follows:

```
company.address.country = 'USA'; // OK
```
Code language: JavaScript (javascript)

JavaScript const and Arrays

Consider the following example:

```
const colors = ['red'];
colors.push('green');
console.log(colors); // ["red", "green"]
```

```
colors.pop();
colors.pop();
console.log(colors); // []

colors = []; // TypeError
```
Code language: JavaScript (javascript)

In this example, we declare an array colors that has one element using the const keyword. Then, we can change the array's elements by adding the green color. However, we cannot reassign the array colors to another array.

JavaScript const in a for loop

ES6 provides a new construct called for...of that allows you to create a loop iterating over iterable objects such as arrays, maps, and sets.

```
let scores = [75, 80, 95];

for (let score of scores) {
        console.log(score);
}
```
Code language: JavaScript (javascript)

If you don't intend to modify the score variable inside the loop, you can use the const keyword instead:

```
let scores = [75, 80, 95];
for (const score of scores) {
  console.log(score);
}
```
Code language: JavaScript (javascript)

In this example, the for...of creates a new binding for the const keyword in each loop iteration. In other words, a new score constant is created in each iteration.

Notice that the const will not work in an imperative for loop. Trying to use the const keyword to declare a variable in the imperative for loop will result in a TypeError:

```
for (const i = 0; i < scores.length; i++) { // TypeError
  console.log(scores[i]);
}
```
Code language: JavaScript (javascript)

The reason is that the declaration is only evaluated once before the loop body starts.

Summary

- The const keyword creates a read-only reference to a value. The readonly reference cannot be reassigned but the value can be change.
- The variables declared by the const keyword are blocked-scope and cannot be redeclared.

Arguments vs. Parameters

Sometimes, you can use the terms argument and parameter interchangeably. However, by definition, parameters are what you specify in the function declaration whereas the arguments are what you pass into the function.

Consider the following add() function:

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

add(100,200);
```

Code language: JavaScript (javascript)

In this example, the x and y are the parameters of the add() function, and the values passed to the add() function 100 and 200 are the arguments.

Setting JavaScript default parameters for a function

In JavaScript, a parameter has a default value of undefined. It means that if you don't pass the arguments into the function, its parameters will have the default values of undefined.

See the following example:

```
function say(message) {
  console.log(message);
}

say(); // undefined
```

Code language: JavaScript (javascript)

The say() function takes the message parameter. Because we didn't pass any argument into the say() function, the value of the message parameter is undefined.

Suppose that you want to give the message parameter a default value 10.

A typical way for achieving this is to test parameter value and assign a default value if it is undefined using a ternary operator:

```
function say(message) {
   message = typeof message !== 'undefined' ? message : 'Hi';
   console.log(message);
}
say(); // 'Hi'
```

Code language: JavaScript (javascript)

In this example, we didn't pass any value into the say() function. Therefore, the default value of the message argument is undefined. Inside the function, we reassigned the message variable the Hi string.

ES6 provides you with an easier way to set the default values for the function parameters like this:

```
function fn(param1=default1, param2=default2,..) {
}
```

Code language: JavaScript (javascript)

In the syntax above, you use the assignment operator (=) and the default value after the parameter name to set a default value for that parameter. For example:

```
function say(message='Hi') {
   console.log(message);
}

say(); // 'Hi'
say(undefined); // 'Hi'
say('Hello'); // 'Hello'
```

Code language: JavaScript (javascript)

How it works.

- In the first function call, we didn't pass any argument into the say() function, therefore message parameter took the default value 'Hi'.
- In the second function call, we passed the undefined into the say() function, hence the message parameter also took the default value 'Hi'.
- In the third function call, we passed the 'Hello' string into the say() function, therefore message parameter took the string 'Hello' as the default value.

More JavaScript default parameter examples

Let's look at some more examples to learn some available options for setting default values of the function parameters.

1) Passing undefined arguments

The following createDiv() function creates a new element in the document with a specific height, width, and border style:

```
function createDiv(height = '100px', width = '100px', border = 'solid 1px red') {
   let div = document.createElement('div');
   div.style.height = height;
   div.style.width = width;
   div.style.border = border;
   document.body.appendChild(div);
   return div;
}
```

Code language: JavaScript (javascript)

The following doesn't pass any arguments to the function so the createDiv() function uses the default values for the parameters.

```
createDiv();
```

Suppose you want to use the default values for the height and width parameters and specific border style. In this case, you need to pass undefined values to the first two parameters as follows:

```
createDiv(undefined,undefined,'solid 5px blue');
```

Code language: JavaScript (javascript)

2) Evaluating default parameters

JavaScript engine evaluates the default arguments at the time you call the function. See the following example:

```
function put(toy, toyBox = []) {
   toyBox.push(toy);
   return toyBox;
}

console.log(put('Toy Car'));
// -> ['Toy Car']
console.log(put('Teddy Bear'));
// -> ['Teddy Bear'], not ['Toy Car','Teddy Bear']
```

Code language: JavaScript (javascript)

The parameter can take a default value which is a result of a function.

Consider the following example:

```
function date(d = today()) {
   console.log(d);
}
function today() {
   return (new Date()).toLocaleDateString("en-US");
}
date();
```

Code language: JavaScript (javascript)

The date() function takes one parameter whose default value is the returned value of the today() function. The today() function returns today's date in a specified string format.

When we declared the date() function, the today() function has not yet evaluated until we called the date() function.

We can use this feature to make arguments are mandatory. If the caller doesn't pass any argument, we throw an error as follows:

```
function requiredArg() {
  throw new Error('The argument is required');
}
function add(x = requiredArg(), y = requiredArg()){
  return x + y;
}

add(10); // error
add(10,20); // OK
```

Code language: JavaScript (javascript)

3) Using other parameters in default values

You can assign a parameter a default value that references to other default parameters as shown in the following example:

```
function add(x = 1, y = x, z = x + y) {
   return x + y + z;
}

console.log(add()); // 4
```

Code language: JavaScript (javascript)

In the add() function:

- The default value of the y is set to x parameter.
- The default value of the z is the sum of x and y
- The add() function returns the sum of x, y, and z.

The parameter list seems to have its own scope. If you reference the parameter that has not been initialized yet, you will get an error. For example:

```
function subtract( x = y, y = 1 ) {
   return x - y;
}
subtract(10);
```
Code language: JavaScript (javascript)

Error message:

```
Uncaught ReferenceError: Cannot access 'y' before initialization
```
Code language: JavaScript (javascript)

Using functions

You can use a return value of a function as a default value for a parameter. For example:

```
let taxRate = () => 0.1;
let getPrice = function( price, tax = price * taxRate() ) {
   return price + tax;
}

let fullPrice = getPrice(100);
console.log(fullPrice); // 110
```
Code language: JavaScript (javascript)

In the getPrice() function, we called the taxRate() function to get the tax rate and use this tax rate to calculate the tax amount from the price.

The arguments object

The value of the arguments object inside the function is the number of actual arguments that you pass to the function. For example:

```
function add(x, y = 1, z = 2) {
   console.log( arguments.length );
   return x + y + z;
}

add(10); // 1
add(10, 20); // 2
add(10, 20, 30); // 3
```
Code language: JavaScript (javascript)

Now, you should understand the JavaScript default function parameters and how to use them effectively.

Section 2. Destructuring

Introduction to JavaScript Array destructuring

Assuming that you have a function that returns an array of numbers as follows:

```
function getScores() {
  return [70, 80, 90];
}
```
Code language: JavaScript (javascript)

The following invokes the getScores() function and assigns the returned value to a variable:

```
let scores = getScores();
```
Code language: JavaScript (javascript)

To get the individual score, you need to do like this:

```
let x = scores[0],
   y = scores[1],
   z = scores[2];
```
Code language: JavaScript (javascript)

Prior to ES6, there was no direct way to assign the elements of the returned array to multiple variables such as x, y and z.

Fortunately, starting from ES6, you can use the destructing assignment as follows:

```
let [x, y, z] = getScores();

console.log(x); // 70
console.log(y); // 80
console.log(z); // 90
```
Code language: JavaScript (javascript)

The variables x, y and z will take the values of the first, second, and third elements of the returned array.

Note that the square brackets [] look like the array syntax but they are not.

If the getScores() function returns an array of two elements, the third variable will be undefined, like this:

```
function getScores() {
  return [70, 80];
}
```

```
let [x, y, z] = getScores();

console.log(x); // 70
console.log(y); // 80
console.log(z); // undefined
```

Code language: JavaScript (javascript)

In case the getScores() function returns an array that has more than three elements, the remaining elements are discarded. For example:

```
function getScores() {
  return [70, 80, 90, 100];
}

let [x, y, z] = getScores();

console.log(x); // 70
console.log(y); // 80
console.log(z); // 90
```

Code language: JavaScript (javascript)

Array Destructuring Assignment and Rest syntax

It's possible to take all remaining elements of an array and put them in a new array by using the rest syntax (...):

```
let [x, y ,...args] = getScores();
console.log(x); // 70
console.log(y); // 80
console.log(args); // [90, 100]
```

Code language: JavaScript (javascript)

The variables x and y receive values of the first two elements of the returned array. And the args variable receives all the remaining arguments, which are the last two elements of the returned array.

Note that it's possible to destructure an array in the assignment that separates from the variable's declaration. For example:

```
let a, b;
[a, b] = [10, 20];
console.log(a); // 10
console.log(b); // 20
```

Code language: JavaScript (javascript)

Setting default values

See the following example:

```
function getItems() {
   return [10, 20];
}

let items = getItems();
let thirdItem = items[2] != undefined ? items[2] : 0;

console.log(thirdItem); // 0
```
Code language: JavaScript (javascript)

How it works:

- First, declare the getItems() function that returns an array of two numbers.
- Then, assign the items variable to the returned array of the getItems() function.
- Finally, check if the third element exists in the array. If not, assign the value 0 to the thirdItem variable.

It'll be simpler with the destructuring assignment with a default value:

```
let [, , thirdItem = 0] = getItems();

console.log(thirdItem); // 0
```
Code language: JavaScript (javascript)

If the value taken from the array is undefined, you can assign the variable a default value, like this:

```
let a, b;
[a = 1, b = 2] = [10];
console.log(a); // 10
console.log(b); // 2
```
Code language: JavaScript (javascript)

If the getItems() function doesn't return an array and you expect an array, the destructing assignment will result in an error. For example:

```
function getItems() {
   return null;
}

let [x = 1, y = 2] = getItems();
```
Code language: JavaScript (javascript)

Error:

```
Uncaught TypeError: getItems is not a function or its return value is not iterable
```
Code language: JavaScript (javascript)

A typical way to solve this is to fallback the returned value of the getItems() function to an empty array like this:

```
function getItems() {
   return null;
}

let [a = 10, b = 20] = getItems() || [];

console.log(a); // 10
console.log(b); // 20
```

Code language: JavaScript (javascript)

Nested array destructuring

The following function returns an array that contains an element which is another array, or nested array:

```
function getProfile() {
   return [
      'John',
      'Doe',
      ['Red', 'Green', 'Blue']
   ];
}
```

Code language: JavaScript (javascript)

Since the third element of the returned array is another array, you need to use the nested array destructuring syntax to destructure it, like this:

```
let [
   firstName,
   lastName,
   [
      color1,
      color2,
      color3
   ]
] = getProfile();

console.log(color1, color2, color3); // Red Green Blue
```

Code language: JavaScript (javascript)

Array Destructuring Assignment Applications

Let's see some practical examples of using the array destructuring assignment syntax.

1) Swapping variables

The array destructuring makes it easy to swap values of variables without using a temporary variable:

```
let a = 10,
    b = 20;

[a, b] = [b, a];

console.log(a); // 20
console.log(b); // 10
```

Code language: JavaScript (javascript)

2) Functions that return multiple values

In JavaScript, a function can return a value. However, you can return an array that contains multiple values, for example:

```
function stat(a, b) {
    return [
        a + b,
        (a + b) / 2,
        a - b
    ]
}
```

Code language: JavaScript (javascript)

And then you use the array destructuring assignment syntax to destructure the elements of the return array into variables:

```
let [sum, average, difference] = stat(20, 10);
console.log(sum, average, difference); // 30, 15, 10
```

Code language: JavaScript (javascript)

In this tutorial, you have learned how to use the ES6 destructuring assignment to destructure elements in an array into individual variables.

Introduction to the JavaScript object destructuring assignment

Suppose you have a person object with two properties: firstName and lastName.

```
let person = {
   firstName: 'John',
   lastName: 'Doe'
};
```
Code language: JavaScript (javascript)

Prior to ES6, when you want to assign properties of the person object to variables, you typically do it like this:

```
let firstName = person.firstName;
let lastName = person.lastName;
```
Code language: JavaScript (javascript)

ES6 introduces the object destructuring syntax that provides an alternative way to assign properties of an object to variables:

```
let { firstName: fname, lastName: lname } = person;
```
Code language: JavaScript (javascript)

In this example, the firstName and lastName properties are assigned to the fName and lName variables respectively.

In this syntax:

```
let { property1: variable1, property2: variable2 } = object;
```
Code language: JavaScript (javascript)

The identifier before the colon (:) is the property of the object and the identifier after the colon is the variable.

> Notice that the property name is always on the left whether it's an object literal or object destructuring syntax.

If the variables have the same names as the properties of the object, you can make the code more concise as follows:

```
let { firstName, lastName } = person;

console.log(firstName); // 'John'
console.log(lastName); // 'Doe'
```
Code language: JavaScript (javascript)

In this example, we declared two variables firstName and lastName, and assigned the properties of the person object to the variables in the same statement.

It's possible to separate the declaration and assignment. However, you must surround the variables in parentheses:

```
({firstName, lastName} = person);
```

If you don't use the parentheses, the JavaScript engine will interpret the left-hand side as a block and throw a syntax error.

When you assign a property that does not exist to a variable using the object destructuring, the variable is set to undefined. For example:

```
let { firstName, lastName, middleName } = person;
console.log(middleName); // undefined
```

Code language: JavaScript (javascript)

In this example, the middleName property doesn't exist in the person object, therefore, the middleName variable is undefined.

Setting default values

You can assign a default value to the variable when the property of an object doesn't exist. For example:

```
let person = {
   firstName: 'John',
   lastName: 'Doe',
   currentAge: 28
};

let { firstName, lastName, middleName = '', currentAge: age = 18 } = person;

console.log(middleName); // ''
console.log(age); // 28
```

Code language: JavaScript (javascript)

In this example, we assign an empty string to the middleName variable when the person object doesn't have the middleName property.

Also, we assign the currentAge property to the age variable with the default value of 18.

However, when the person object does have the middleName property, the assignment works as usual:

```
let person = {
```

```
  firstName: 'John',
  lastName: 'Doe',
  middleName: 'C.',
  currentAge: 28
};

let { firstName,  lastName, middleName = '', currentAge: age = 18 } = person;

console.log(middleName); // 'C.'
console.log(age); // 28
```

Code language: JavaScript (javascript)

Destructuring a null object

A function may return an object or null in some situations. For example:

```
function getPerson() {
  return null;
}
```

Code language: JavaScript (javascript)

And you use the object destructuring assignment:

```
let { firstName, lastName } = getPerson();

console.log(firstName, lastName);
```

Code language: JavaScript (javascript)

The code will throw a TypeError:

```
TypeError: Cannot destructure property 'firstName' of 'getPerson(...)' as it is null.
```

Code language: JavaScript (javascript)

To avoid this, you can use the OR operator (||) to fallback the null object to an empty object:

```
let { firstName, lastName } = getPerson() || {};
```

Code language: JavaScript (javascript)

Now, no error will occur. And the firstName and lastName will be undefined.

Nested object destructuring

Assuming that you have an employee object which has a name object as the property:

```
let employee = {
  id: 1001,
  name: {
    firstName: 'John',
    lastName: 'Doe'
```

```
    }
};
```

Code language: JavaScript (javascript)

The following statement destructures the properties of the nested name object into individual variables:

```
let {
    name: {
        firstName,
        lastName
    }
} = employee;

console.log(firstName); // John
console.log(lastName); // Doe
```

Code language: JavaScript (javascript)

It's possible to do multiple assignement of a property to multiple variables:

```
let employee = {
    id: 1001,
    name: {
        firstName: 'John',
        lastName: 'Doe'
    }
};

let {
    name: {
        firstName,
        lastName
    },
    name
} = employee;

console.log(firstName); // John
console.log(lastName); // Doe
console.log(name); // { firstName: 'John', lastName: 'Doe' }
```

Code language: JavaScript (javascript)

Destructuring function arguments

Suppose you have a function that displays the person object:

```
let display = (person) => console.log(`${person.firstName} ${person.lastName}`);

let person = {
    firstName: 'John',
    lastName: 'Doe'
};
```

```
display(person);
```
Code language: JavaScript (javascript)

It's possible to destructure the object argument passed into the function like this:

```
let display = ({firstName, lastName}) => console.log(`${firstName} ${lastName}`);

let person = {
   firstName: 'John',
   lastName: 'Doe'
};

display(person);
```
Code language: JavaScript (javascript)

It looks less verbose especially when you use many properties of the argument object. This technique is often used in React.

Summary

- Object destructuring assigns the properties of an object to variables with the same names by default.

Section 3. ES6 Modules

Executing modules on web browsers

First, create a new file called message.js and add the following code:

```
export let message = 'ES6 Modules';
```
Code language: JavaScript (javascript)

The message.js is a module in ES6 that contains the message variable. The export statement exposes the message variable to other modules.

Second, create another new file named app.js that uses the message.js module.
The app.js module creates a new heading 1 (h1) element and attaches it to an HTML page.
The import statement imports the message variable from the message.js module.

```
import { message } from './message.js'

const h1 = document.createElement('h1');
h1.textContent = message

document.body.appendChild(h1)
```
Code language: JavaScript (javascript)

Third, create a new HTML page that uses the app.js module:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>ES6 Modules</title>
</head>
<body>
<script type="module" src="./app.js"></script>
</body>
</html>
```

Code language: HTML, XML (xml)

Note that we used the type="module" in the script tag to load the app.js module. If you view the page on a web browser, you will see the following page:

Let's examine the export and import statements in more detail.

Exporting

To export a variable, a function, or a class, you place the export keyword in front of it as follows:

```
// log.js
export let message = 'Hi';

export function getMessage() {
  return message;
}

export function setMessage(msg) {
  message = msg;
}

export class Logger {
}
```

Code language: JavaScript (javascript)

In this example, we have the log.js module with a variable, two functions, and one class. We used the export keyword to exports all identifiers in the module.

Note that the export keyword requires the function or class to have a name to be exported. You can't export an anonymous function or class using this syntax.

JavaScript allows you to define a variable, a function, or a class first then export it later as follows:

```
// foo.js
function foo() {
  console.log('foo');
}

function bar() {
 console.log('bar');
}
export foo;
```
Code language: JavaScript (javascript)

In this example, we defined the foo() function first and then exported it. Since we didn't export the bar() function, we couldn't access it in other modules. The bar() function is inaccessible outside the module or we say it is private.

Importing

Once you define a module with exports, you can access the exported variables, functions, and classes in another module by using the import keyword. The following illustrates the syntax:

```
import { what, ever } from './other_module.js';
```
Code language: JavaScript (javascript)

In this syntax:

- First, specify what to import inside the curly braces, which are called bindings.
- Then, specify the module from which you import the given bindings.

> Note that when you import a binding from a module, the binding behaves like it was defined using const. It means you can't have another identifier with the same name or change the value of the binding.

See the following example:

```
// greeting.js
export let message = 'Hi';

export function setMessage(msg) {
 message = msg;
}
```

Code language: JavaScript (javascript)

When you import the message variable and setMessage() function, you can use the setMessage() function to change the value of the message variable as shown below:

```
// app.js
import {message, setMessage } from './greeting.js';
console.log(message); // 'Hi'

setMessage('Hello');
console.log(message); // 'Hello'
```
Code language: JavaScript (javascript)

However, you can't change the value of the message variable directly. The following expression causes an error:

```
message = 'Hallo'; // error
```
Code language: JavaScript (javascript)

Behind the scenes, when you called the setMessage() function. JavaScript went back to the greeting.js module and executed code in there and changed the message variable. The change was then automatically reflected on the imported message binding.

The message binding in the app.js is the local name for exported message identifier. So basically the message variables in the app.js and greeting.js modules aren't the same.

Import a single binding

Suppose you have a module with the foo variable as follows:

```
// foo.js
export foo = 10;
```
Code language: JavaScript (javascript)

Then, in another module, you can reuse the foo variable:

```
// app.js
import { foo } from './foo.js';
console.log(foo); // 10;
```
Code language: JavaScript (javascript)

However, you can't change the value of foo. If you attempt to do so, you will get an error:

```
foo = 20; // throws an error
```
Code language: JavaScript (javascript)

Import multiple bindings

Suppose you have the cal.js module as follows:

```
// cal.js
export let a = 10,
        b = 20,
        result = 0;

export function sum() {
  result = a + b;
  return result;
}

export function multiply() {
  result = a * b;
  return result;
}
```

Code language: JavaScript (javascript)

And you want to import these bindings from the cal.js, you can explicitly list them as follows:

```
import {a, b, result, sum, multiply } from './cal.js';
sum();
console.log(result); // 30

multiply();
console.log(result); // 200
```

Code language: JavaScript (javascript)

Import an entire module as an object

To import everything from a module as a single object, you use the asterisk (*) pattern as follows:

```
import * as cal from './cal.js';
```

Code language: JavaScript (javascript)

In this example, we imported all bindings from the cal.js module as the cal object. In this case, all the bindings become properties of the cal object, so you can access them as shown below:

```
cal.a;
cal.b;
cal.sum();
```

Code language: CSS (css)

This import is called *namespace import*.

It's important to keep in mind that the imported module executes *only once* even import it multiple times. Consider this example:

```
import { a } from './cal.js';
import { b } from './cal.js';
import {result} from './cal.js';
```
Code language: JavaScript (javascript)

After the first import statement, the cal.js module is executed and loaded into the memory, and it is reused whenever it is referenced by the subsequent import statement.

Limitation of import and export statements

Note that you must use the import or export statement *outside* other statements and functions. The following example causes a SyntaxError:

```
if( requiredSum ) {
  export sum;
}
```
Code language: JavaScript (javascript)

Because we used the export statement inside the if statement. Similarly, the following import statement also causes a SyntaxError:

```
function importSum() {
  import {sum} from './cal.js';
}
```
Code language: JavaScript (javascript)

Because we used the import statement inside a function.

The reason for the error is that JavaScript must *statically* determine what will be exported and imported.

> Note that ES2020 introduced the function-like object import() that allows you to dynamically import a module.

Aliasing

JavaScript allows you to create aliases for variables, functions, or classes when you export and import. See the following math.js module:

```
// math.js
function add( a, b ) {
  return a + b;
}
```

```
export { add as sum };
```
Code language: JavaScript (javascript)

In this example, instead of exporting the add() function, we used the as keyword to assign the sum() function an alias.

So when you import the add() function from the math.js module, you must use sum instead:

```
import { sum } from './math.js';
```
Code language: JavaScript (javascript)

If you want to use a different name when you import, you can use the as keyword as follows:

```
import {sum as total} from './math.js';
```
Code language: JavaScript (javascript)

Re-exporting a binding

It's possible to export bindings that you have imported. This is called re-exporting. For example:

```
import { sum } from './math.js';
export { sum };
```
Code language: JavaScript (javascript)

In this example, we imported sum from the math.js module and re-export it. The following statement is equivalent to the statements above:

```
export {sum} from './math.js';
```
Code language: JavaScript (javascript)

In case you want to rename the bindings before re-exporting, you use the as keyword. The following example imports sum from the math.js module and re-export it as add.

```
export { sum as add } from './math.js';
```
Code language: JavaScript (javascript)

If you want to export all the bindings from another module, you can use the asterisk (*):

```
export * from './cal.js';
```
Code language: JavaScript (javascript)

Importing without bindings

Sometimes, you want to develop a module that doesn't export anything, for example, you may want to add a new method to a built-in object such as the Array.

```
// array.js
```

```
if (!Array.prototype.contain) {
  Array.prototype.contain = function(e) {
    // contain implementation
    // ...
  }
}
```
Code language: JavaScript (javascript)

Now, you can import the module without any binding and use the contain() method defined in the array.js module as follows:

```
import './array.js';
[1,2,3].contain(2); // true
```
Code language: JavaScript (javascript)

Default exports

A module can have one and only one default export. The default export is easier to import. The default for a module can be a variable, a function, or a class.

The following is the sort.js module with a default export.

```
// sort.js
export default function(arr) {
  // sorting here
}
```
Code language: JavaScript (javascript)

Note that you don't need to specify the name for the function because the module represents the function name.

```
import sort from sort.js;
sort([2,1,3]);
```
Code language: JavaScript (javascript)

As you see, the sort identifier represents the default function of the sort.js module. Notice that we didn't use the curly brace {} surrounding the sort identifier.

Let's change the sort.js module to include the default export as well as the non-default one:

```
// sort.js
export default function(arr) {
  // sorting here
}
export function heapSort(arr) {
  // heapsort
}
```
Code language: JavaScript (javascript)

To import both default and non-default bindings, you use the specify a list of bindings after the import keyword with the following rules:

- The default binding must come first.
- The non-default binding must be surrounded by curly braces.

See the following example:

```
import sort, {heapSort} from './sort.js';
sort([2,1,3]);
heapSort([3,1,2]);
```
Code language: JavaScript (javascript)

To rename the default export, you also use the as keyword as follows:

```
import { default as quicksort, heapSort} from './sort.js';
```
Code language: JavaScript (javascript)

In this tutorial, you have learned about ES6 modules and how to export bindings from a module and import them into another module.

Section 4. ES6 Classes

- Class – introduce you to the ES6 class syntax and how to declare a class.
- Getters and Setters – define the getters and setters for a class using the get and set keywords.
- Class Expression – learn an alternative way to define a new class using a class expression.
- Static methods – guide you on how to define methods associated with a class, not instances of that class.
- Static Properties – show you how to define static properties shared by all instances of a class.
- Computed property – explain the computed property and its practical application.
- Inheritance – show you how to extend a class using the extends and super keywords.
- new.target – introduce you to the new.target metaproperty.

Section 5. Arrow Functions

- Arrow functions – introduce you to the arrow functions (=>)

- Arrow functions: when you should not use – learn when not to use the arrow functions.

Section 6. Symbol

- Symbol – introduce you to a new primitive type called symbol in ES6

Section 7. Iterators & Generators

- Iterators – introduce you to the iteration and iterator protocols.
- Generators – develop functions that can pause midway and then continue from where they paused.
- yield – dive into how to use the yield keyword in generators.