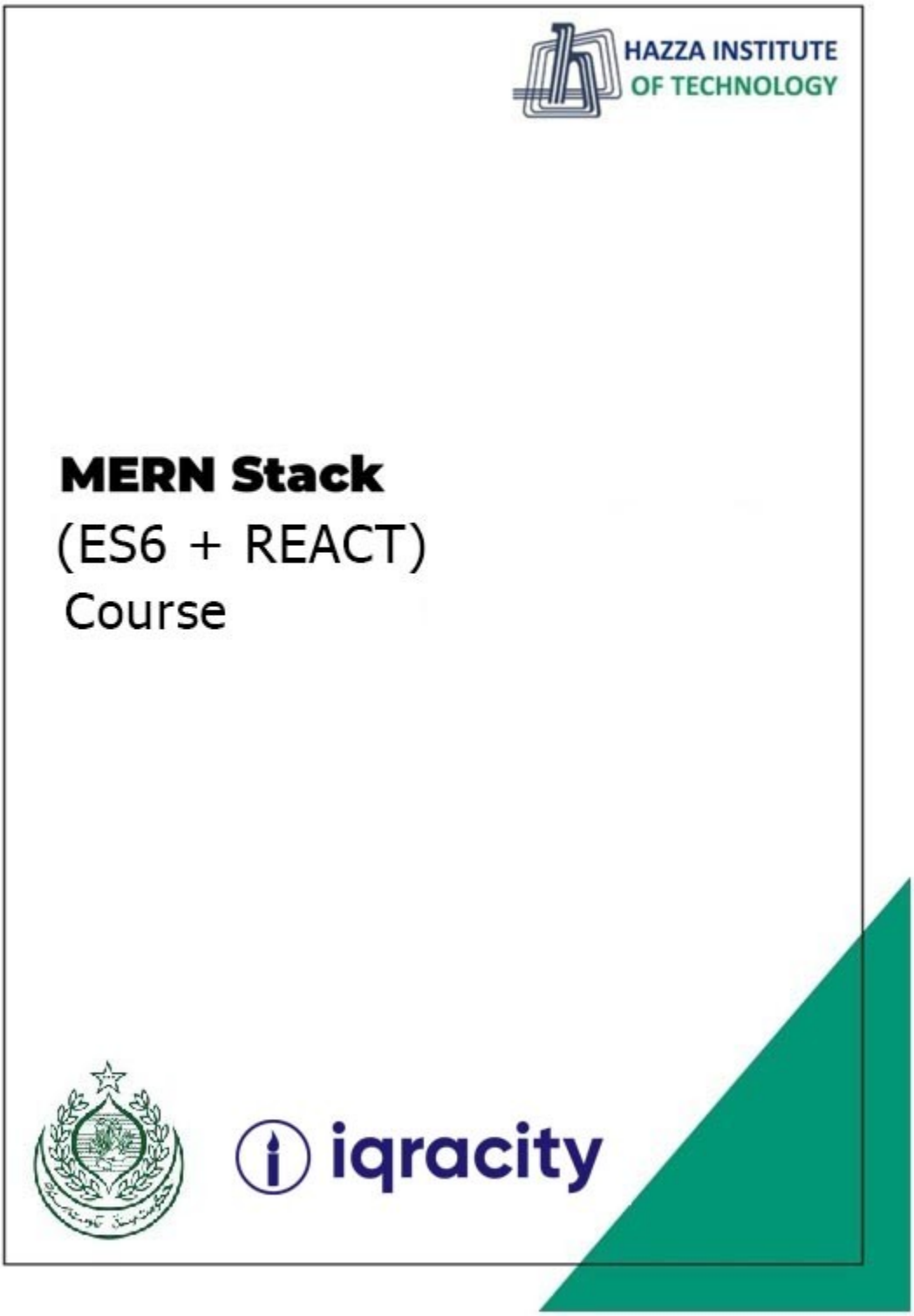
HAZZA INSTITUTE OF TECHNOLOGY

iqracity



Lab 10

Total Time:
3 hours

Pre-Lab Activities:

- Have a good knowledge of JavaScript up to ES5

Learning Outcomes:

- Have an good understanding of Javascript and ES6

Lab Tasks:

- Promises
- ES6 Collections
- Array Extensions
- Object Extensions
- String Extensions
- Proxy & Reflection

Student Activities:

- To explore Promises
- To explore ES6 Collections
- To explore Array Extensions
- To explore Object Extensions
- To explore String Extensions
- To explore Proxy & Reflection

Lab Solution

Section 8. Promises

- Promises – learn about Javascript Promises, what they are, and how to use them effectively.

Why JavaScript promises

The following example defines a function getUsers() that returns a list of user objects:

```
function getUsers() {
  return [
    { username: 'john', email: 'john@test.com' },
    { username: 'jane', email: 'jane@test.com' },
  ];
}
```
Code language: JavaScript (javascript)

Each user object has two properties username and email.

To find a user by username from the user list returned by the getUsers() function, you can use the findUser() function as follows:

```
function findUser(username) {
  const users = getUsers();
  const user = users.find((user) => user.username === username);
  return user;
}
```
Code language: JavaScript (javascript)

In the findUser() function:

- First, get a user array by calling the getUsers() function
- Second, find the user with a specific username by using the find() method of the Array object.
- Third, return the matched user.

The following shows the complete code for finding a user with the username 'john':

```
function getUsers() {
  return [
    { username: 'john', email: 'john@test.com' },
    { username: 'jane', email: 'jane@test.com' },
  ];
}

function findUser(username) {
```

```
  const users = getUsers();
  const user = users.find((user) => user.username === username);
  return user;
}

console.log(findUser('john'));
```
Code language: JavaScript (javascript)

Output:

```
{ username: 'john', email: 'john@test.com' }
```
Code language: CSS (css)

The code in the findUser() function is synchronous and blocking. The findUser() function executes the getUsers() function to get a user array, calls the find() method on the users array to search for a user with a specific username, and returns the matched user.

In practice, the getUsers() function may access a database or call an API to get the user list. Therefore, the getUsers() function will have a delay.

To simulate the delay, you can use the setTimeout() function. For example:

```
function getUsers() {
  let users = [];

  // delay 1 second (1000ms)
  setTimeout(() => {
    users = [
      { username: 'john', email: 'john@test.com' },
      { username: 'jane', email: 'jane@test.com' },
    ];
  }, 1000);

  return users;
}
```
Code language: JavaScript (javascript)

How it works.

- First, define an array users and initialize its value with an empty array.
- Second, assign an array of the users to the users variable inside the callback of the setTimeout() function.
- Third, return the users array

The getUsers() won't work properly and always returns an empty array. Therefore, the findUser() function won't work as expected:

```
function getUsers() {
  let users = [];
  setTimeout(() => {
    users = [
      { username: 'john', email: 'john@test.com' },
      { username: 'jane', email: 'jane@test.com' },
    ];
  }, 1000);
  return users;
}

function findUser(username) {
  const users = getUsers(); // A
  const user = users.find((user) => user.username === username); // B
  return user;
}

console.log(findUser('john'));
```

Code language: JavaScript (javascript)

Output:

```
undefined
```

Code language: JavaScript (javascript)

Because the getUsers() returns an empty array, the users array is empty (line A). When calling the find() method on the users array, the method returns undefined (line B)

The challenge is how to access the users returned from the getUsers() function after one second. One classical approach is to use the callback.

Using callbacks to deal with an asynchronous operation

The following example adds a callback argument to the getUsers() and findUser() functions:

```
function getUsers(callback) {
  setTimeout(() => {
    callback([
      { username: 'john', email: 'john@test.com' },
      { username: 'jane', email: 'jane@test.com' },
    ]);
  }, 1000);
}

function findUser(username, callback) {
  getUsers((users) => {
    const user = users.find((user) => user.username === username);
```

```
  callback(user);
 });
}

findUser('john', console.log);
```
Code language: JavaScript (javascript)

Output:

```
{ username: 'john', email: 'john@test.com' }
```
Code language: CSS (css)

In this example, the getUsers() function accepts a callback function as an argument and invokes it with the users array inside the setTimeout() function. Also, the findUser() function accepts a callback function that processes the matched user.

The callback approach works very well. However, it makes the code more difficult to follow. Also, it adds complexity to the functions with callback arguments.

If the number of functions grows, you may end up with the callback hell problem. To resolve this, JavaScript comes up with the concept of promises.

Understanding JavaScript Promises

By definition, a promise is an **object** that encapsulates the result of an **asynchronous operation**.

A promise object has a state that can be one of the following:

- Pending
- Fulfilled with a **value**
- Rejected for a **reason**

In the beginning, the state of a promise is pending, indicating that the asynchronous operation is in progress. Depending on the result of the asynchronous operation, the state changes to either fulfilled or rejected.

The fulfilled state indicates that the asynchronous operation was completed successfully:

The rejected state indicates that the asynchronous operation failed.

Creating a promise

To create a promise object, you use the Promise() constructor:

```
const promise = new Promise((resolve, reject) => {
  // contain an operation
  // ...

  // return the state
  if (success) {
    resolve(value);
  } else {
    reject(error);
  }
});
```

Code language: JavaScript (javascript)

The promise constructor accepts a callback function that typically performs an asynchronous operation. This function is often referred to as an executor.

In turn, the executor accepts two callback functions with the name resolve and reject.

> Note that the callback functions passed into the executor are resolve and reject by convention only.

If the asynchronous operation completes successfully, the executor will call the resolve() function to change the state of the promise from pending to fulfilled with a value.

In case of an error, the executor will call the reject() function to change the state of the promise from pending to rejected with the error reason.

Once a promise reaches either fulfilled or rejected state, it stays in that state and can't go to another state.

In other words, a promise cannot go from the fulfilled state to the rejected state and vice versa. Also, it cannot go back from the fulfilled or rejected state to the pending state.

Once a new Promise object is created, its state is pending. If a promise reaches fulfilled or rejected state, it is *resolved*.

> Note that you will rarely create promise objects in practice. Instead, you will consume promises provided by libraries.

Consuming a Promise: then, catch, finally

1) The then() method

To get the value of a promise when it's fulfilled, you call the then() method of the promise object. The following shows the syntax of the then() method:

```
promise.then(onFulfilled,onRejected);
```
Code language: CSS (css)

The then() method accepts two callback functions: onFulfilled and onRejected.

The then() method calls the onFulfilled() with a value, if the promise is fulfilled or the onRejected() with an error if the promise is rejected.

> Note that both onFulfilled and onRejected arguments are optional.

The following example shows how to use then() method of the Promise object returned by the getUsers() function:

```
function getUsers() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve([
        { username: 'john', email: 'john@test.com' },
        { username: 'jane', email: 'jane@test.com' },
      ]);
    }, 1000);
  });
}

function onFulfilled(users) {
  console.log(users);
}

const promise = getUsers();
promise.then(onFulfilled);
```
Code language: JavaScript (javascript)

Output:

```
[
  { username: 'john', email: 'john@test.com' },
  { username: 'jane', email: 'jane@test.com' }
]
```
Code language: JavaScript (javascript)

In this example:

- First, define the onFulfilled() function to be called when the promise is fulfilled.
- Second, call the getUsers() function to get a promise object.
- Third, call the then() method of the promise object and output the user list to the console.

To make the code more concise, you can use an arrow function as the argument of the then() method like this:

```
function getUsers() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve([
        { username: 'john', email: 'john@test.com' },
        { username: 'jane', email: 'jane@test.com' },
      ]);
    }, 1000);
  });
}

const promise = getUsers();

promise.then((users) => {
  console.log(users);
});
```

Code language: JavaScript (javascript)

Because the getUsers() function returns a promise object, you can chain the function call with the then() method like this:

```
// getUsers() function
//...

getUsers().then((users) => {
  console.log(users);
});
```

Code language: JavaScript (javascript)

In this example, the getUsers() function always succeeds. To simulate the error, we can use a success flag like the following:

```
let success = true;

function getUsers() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (success) {
```

```
      resolve([
        { username: 'john', email: 'john@test.com' },
        { username: 'jane', email: 'jane@test.com' },
      ]);
    } else {
      reject('Failed to the user list');
    }
  }, 1000);
 });
}

function onFulfilled(users) {
 console.log(users);
}
function onRejected(error) {
 console.log(error);
}

const promise = getUsers();
promise.then(onFulfilled, onRejected);
```

Code language: JavaScript (javascript)

How it works.

First, define the success variable and initialize its value to true.

If the success is true, the promise in the getUsers() function is fulfilled with a user list. Otherwise, it is rejected with an error message.

Second, define the onFulfilled and onRejected functions.

Third, get the promise from the getUsers() function and call the then() method with the onFulfilled and onRejected functions.

The following shows how to use the arrow functions as the arguments of the then() method:

```
// getUsers() function
// ...

const promise = getUsers();
promise.then(
 (users) => console.log,
 (error) => console.log
);
```

Code language: JavaScript (javascript)

2) The catch() method

If you want to get the error only when the state of the promise is rejected, you can use the catch() method of the Promise object:

```
promise.catch(onRejected);
```
Code language: CSS (css)

Internally, the catch() method invokes the then(undefined, onRejected) method.

The following example changes the success flag to false to simulate the error scenario:

```
let success = false;

function getUsers() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (success) {
        resolve([
          { username: 'john', email: 'john@test.com' },
          { username: 'jane', email: 'jane@test.com' },
        ]);
      } else {
        reject('Failed to the user list');
      }
    }, 1000);
  });
}

const promise = getUsers();

promise.catch((error) => {
  console.log(error);
});
```
Code language: JavaScript (javascript)

3) The finally() method

Sometimes, you want to execute the same piece of code whether the promise is fulfilled or rejected. For example:

```
const render = () => {
  //...
};

getUsers()
```

```
  .then((users) => {
    console.log(users);
    render();
  })
  .catch((error) => {
    console.log(error);
    render();
  });
```
Code language: JavaScript (javascript)

As you can see, the render() function call is duplicated in both then() and catch() methods.

To remove this duplicate and execute the render() whether the promise is fulfilled or rejected, you use the finally() method, like this:

```
const render = () => {
  //...
};

getUsers()
  .then((users) => {
    console.log(users);
  })
  .catch((error) => {
    console.log(error);
  })
  .finally(() => {
    render();
  });
```
Code language: JavaScript (javascript)

A practical JavaScript Promise example

The following example shows how to load a JSON file from the server and display its contents on a webpage.

Suppose you have the following JSON file:

https://www.javascripttutorial.net/sample/promise/api.json
Code language: JavaScript (javascript)

with the following contents:

```
{
    "message": "JavaScript Promise Demo"
}
```
Code language: JSON / JSON with Comments (json)

The following shows the HTML page that contains a button. When you click the button, the page loads data from the JSON file and shows the message:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>JavaScript Promise Demo</title>
<link href="css/style.css" rel="stylesheet">
</head>
<body>
<div id="container">
<div id="message"></div>
<button id="btnGet">Get Message</button>
</div>
<script src="js/promise-demo.js">
</script>
</body>
</html>
```

Code language: HTML, XML (xml)

The following shows the promise-demo.js file:

```
function load(url) {
  return new Promise(function (resolve, reject) {
    const request = new XMLHttpRequest();
    request.onreadystatechange = function () {
      if (this.readyState === 4 && this.status == 200) {
        resolve(this.response);
      } else {
        reject(this.status);
      }
    };
    request.open('GET', url, true);
    request.send();
  });
}

const url = 'https://www.javascripttutorial.net/sample/promise/api.json';
const btn = document.querySelector('#btnGet');
const msg = document.querySelector('#message');

btn.addEventListener('click', () => {
  load(URL)
    .then((response) => {
      const result = JSON.parse(response);
      msg.innerHTML = result.message;
    })
```

```
  .catch((error) => {
    msg.innerHTML = `Error getting the message, HTTP status: ${error}`;
  });
});
```

Code language: JavaScript (javascript)

How it works.

First, define the load() function that uses the XMLHttpRequest object to load the JSON file from the server:

```
function load(url) {
  return new Promise(function (resolve, reject) {
    const request = new XMLHttpRequest();
    request.onreadystatechange = function () {
      if (this.readyState === 4 && this.status == 200) {
        resolve(this.response);
      } else {
        reject(this.status);
      }
    };
    request.open('GET', url, true);
    request.send();
  });
}
```

Code language: JavaScript (javascript)

In the executor, we call resolve() function with the Response if the HTTP status code is 200. Otherwise, we invoke the reject() function with the HTTP status code.

Second, register the button click event listener and call the then() method of the promise object. If the load is successful, then we show the message returned from the server. Otherwise, we show the error message with the HTTP status code.

```
const url = 'https://www.javascripttutorial.net/sample/promise/api.json';
const btn = document.querySelector('#btnGet');
const msg = document.querySelector('#message');

btn.addEventListener('click', () => {
  load(URL)
    .then((response) => {
      const result = JSON.parse(response);
      msg.innerHTML = result.message;
    })
    .catch((error) => {
      msg.innerHTML = `Error getting the message, HTTP status: ${error}`;
```

```
    });
});
```

Code language: JavaScript (javascript)

Summary

- A promise is an object that encapsulates the result of an asynchronous operation.
- A promise starts in the pending state and ends in either fulfilled state or rejected state.
- Use then() method to schedule a callback to be executed when the promise is fulfilled, and catch() method to schedule a callback to be invoked when the promise is rejected.
- Place the code that you want to execute in the finally() method whether the promise is fulfilled or rejected.

- Promise chaining – show you how to execute multiple asynchronous operations in sequence.
- Promise composition: Promise.all() & Promise.race() – learn how to compose a new promise out of several promises.
- Promise error handling – guide you on how to handle errors in promises.

Section 9. ES6 collections

- Map – introduce you to the Map type that holds a collection of key-value pairs.

Introduction to JavaScript Map object

Before ES6, we often used an object to emulate a map by mapping a key to a value of any type. But using an object as a map has some side effects:

1. An object always has a default key like the prototype.
2. A key of an object must be a string or a symbol, you cannot use an object as a key.
3. An object does not have a property that represents the size of the map.

ES6 provides a new collection type called Map that addresses these deficiencies.

By definition, a Map object holds key-value pairs where values of any type can be used as either keys or values. In addition, a Map object remembers the original insertion order of the keys.

To create a new Map, you use the following syntax:

```
let map = new Map([iterable]);
```

Code language: JavaScript (javascript)

The Map() accepts an optional iterable object whose elements are key-value pairs.

Useful JavaScript Map methods

- clear() – removes all elements from the map object.
- delete(key) – removes an element specified by the key. It returns if the element is in the map, or false if it does not.
- entries() – returns a new Iterator object that contains an array of [key, value] for each element in the map object. The order of objects in the map is the same as the insertion order.
- forEach(callback[, thisArg]) – invokes a callback for each key-value pair in the map in the insertion order. The optional thisArg parameter sets the this value for each callback.
- get(key) – returns the value associated with the key. If the key does not exist, it returns undefined.
- has(key) – returns true if a value associated with the key exists, otherwise, return false.
- keys() – returns a new Iterator that contains the keys for elements in insertion order.
- set(key, value) – sets the value for the key in the map object. It returns the map object itself therefore you can chain this method with other methods.
- values() returns a new iterator object that contains values for each element in insertion order.

JavaScript Map examples

Create a new Map object

Suppose you have a list of user objects as follows:

```
let john = {name: 'John Doe'},
    lily = {name: 'Lily Bush'},
    peter = {name: 'Peter Drucker'};
```
Code language: JavaScript (javascript)

Assuming that you have to create a map of users and roles. In this case, you use the following code:

```
let userRoles = new Map();
```
Code language: JavaScript (javascript)

The userRoles is an instance of the Map object and its type is an object as illustrated in the following example:

```
console.log(typeof(userRoles)); // object
console.log(userRoles instanceof Map); // true
```
Code language: JavaScript (javascript)

Add elements to a Map

To assign a role to a user, you use the set() method:

```
userRoles.set(john, 'admin');
```
Code language: JavaScript (javascript)

The set() method maps user john with the admin role. Since the set() method is chainable, you can save some typings as shown in this example:

```
userRoles.set(lily, 'editor')
        .set(peter, 'subscriber');
```
Code language: JavaScript (javascript)

Initialize a map with an iterable object

As mentioned earlier, you can pass an iterable object to the Map() constructor:

```
let userRoles = new Map([
   [john, 'admin'],
   [lily, 'editor'],
   [peter, 'subscriber']
]);
```
Code language: JavaScript (javascript)

Get an element from a map by key

If you want to see the roles of John , you use the get() method:

```
userRoles.get(john); // admin
```
Code language: JavaScript (javascript)

If you pass a key that does not exist, the get() method will return undefined.

```
let foo = {name: 'Foo'};
userRoles.get(foo); //undefined
```
Code language: JavaScript (javascript)

Check the existence of an element by key

To check if a key exists in the map, you use the has() method.

```
userRoles.has(foo); // false
userRoles.has(lily); // true
```
Code language: JavaScript (javascript)

Get the number of elements in the map

The size property returns the number of entries of the map.

```
console.log(userRoles.size); // 3
```
Code language: JavaScript (javascript)

Iterate over map keys

To get the keys of a Map object, you use the keys() method. The keys() returns a new iterator object that contains the keys of elements in the map.

The following example displays the username of the users in the userRoles map object.

```
let john = { name: 'John Doe' },
  lily = { name: 'Lily Bush' },
  peter = { name: 'Peter Drucker' };

let userRoles = new Map([
  [john, 'admin'],
  [lily, 'editor'],
  [peter, 'subscriber'],
]);

for (const user of userRoles.keys()) {
  console.log(user.name);
}
```
Code language: JavaScript (javascript)

Output:

```
John Doe
Lily Bush
Peter Drucker
```

Iterate over map values

Similarly, you can use the values() method to get an iterator object that contains values for all the elements in the map:

```
let john = { name: 'John Doe' },
  lily = { name: 'Lily Bush' },
  peter = { name: 'Peter Drucker' };

let userRoles = new Map([
  [john, 'admin'],
  [lily, 'editor'],
  [peter, 'subscriber'],
]);

for (let role of userRoles.values()) {
  console.log(role);
}
```
Code language: JavaScript (javascript)

Output:

```
admin
editor
subscriber
```

Iterate over map elements

Also, the entries() method returns an iterator object that contains an array of [key,value] of each element in the Map object:

```
let john = { name: 'John Doe' },
  lily = { name: 'Lily Bush' },
  peter = { name: 'Peter Drucker' };

let userRoles = new Map([
  [john, 'admin'],
  [lily, 'editor'],
  [peter, 'subscriber'],
]);

for (const role of userRoles.entries()) {
  console.log(`${role[0].name}: ${role[1]}`);
}
```

Code language: JavaScript (javascript)

To make the iteration more natural, you can use destructuring as follows:

```
let john = { name: 'John Doe' },
  lily = { name: 'Lily Bush' },
  peter = { name: 'Peter Drucker' };

let userRoles = new Map([
  [john, 'admin'],
  [lily, 'editor'],
  [peter, 'subscriber'],
]);

for (let [user, role] of userRoles.entries()) {
  console.log(`${user.name}: ${role}`);
}
```

Code language: JavaScript (javascript)

In addition to for...of loop, you can use the forEach() method of the map object:

```
let john = { name: 'John Doe' },
  lily = { name: 'Lily Bush' },
```

```
  peter = { name: 'Peter Drucker' };

let userRoles = new Map([
  [john, 'admin'],
  [lily, 'editor'],
  [peter, 'subscriber'],
]);

userRoles.forEach((role, user) => console.log(`${user.name}: ${role}`));
```
Code language: JavaScript (javascript)

Convert map keys or values to a array

Sometimes, you want to work with an array instead of an iterable object, in this case, you can use the spread operator.

The following example converts keys for each element into an array of keys:

```
var keys = [...userRoles.keys()];
console.log(keys);
```
Code language: JavaScript (javascript)

Output:

```
[ { name: 'John Doe' },
  { name: 'Lily Bush' },
  { name: 'Peter Drucker' } ]
```
Code language: JavaScript (javascript)

And the following converts the values of elements to an array:

```
let roles = [...userRoles.values()];
console.log(roles);
```
Code language: JavaScript (javascript)

Output

```
[ 'admin', 'editor', 'subscriber' ]
```
Code language: JSON / JSON with Comments (json)

Delete an element by key

To delete an entry in the map, you use the delete() method.

```
userRoles.delete(john);
```
Code language: CSS (css)

Delete all elements in the map

To delete all entries in the Map object, you use the clear() method:

```
userRoles.clear();
```
Code language: CSS (css)

Hence, the size of the map now is zero.

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

WeakMap

A WeakMap is similar to a Map except the keys of a WeakMap must be objects. It means that when a reference to a key (an object) is out of scope, the corresponding value is automatically released from the memory.

A WeakMap only has subset methods of a Map object:

- get(key)
- set(key, value)
- has(key)
- delete(key)

Here are the main difference between a Map and a WeekMap:

- Elements of a WeakMap cannot be iterated.
- Cannot clear all elements at once.
- Cannot check the size of a WeakMap.

In this tutorial, you have learned how to work with the JavaScript Map object and its useful methods to manipulate entries in the map.

- Set – learn how to use the Set type that holds a collection of unique values.

Section 10. Array extensions

- Array.of() – improve array creation.

Introduction to the JavaScript Array.of() method

In ES5, when you pass a number to the Array constructor, JavaScript creates an array whose length equals the number. For example:

```
let numbers = new Array(2);
console.log(numbers.length); // 2
```

```
console.log(numbers[0]); // undefined
```
Code language: JavaScript (javascript)

However, when you pass to the Array constructor a value that is not a number, JavaScript creates an array that contains one element with that value. For example:

```
numbers = new Array("2");
console.log(numbers.length); // 1
console.log(numbers[0]); // "2"
```
Code language: JavaScript (javascript)

This behavior is sometimes confusing and error-prone because you may not know the type of data that you pass to the Array constructor.

ES6 introduces the Array.of() method to solve this problem.

The Array.of() method is similar to the Array constructor except the Array.of() method does not treat a single numeric value special.

In other words, the Array.of() method always creates an array that contains the values that you pass to it regardless of the types or the number of arguments.

The following shows the syntax of the Array.of() method:

```
Array.of(element0[, element1[, ...[, elementN]]])
```
Code language: CSS (css)

JavaScript Array.of() examples

See the following example:

```
let numbers = Array.of(3);
console.log(numbers.length); // 1
console.log(numbers[0]); // 3
```
Code language: JavaScript (javascript)

In this example, we passed the number 3 to the Array.of() method. The Array.of() method creates an array of one number.

Consider the following example:

```
let chars = Array.of('A', 'B', 'C');
console.log(chars.length); // 3
console.log(chars); // ['A','B','C']
```
Code language: JavaScript (javascript)

In this example, we created an array of three strings by passing 'A', 'B', and 'C' to the Array.of() method. The size of the array is 3.

JavaScript Array.of() polyfill

If you execute the JavaScript in the environment that doesn't support the Array.of() method, you can use the following polyfill:

```
if (!Array.of) {
   Array.of = function() {
      return Array.prototype.slice.call(arguments);
   };
}
```
Code language: JavaScript (javascript)

In this tutorial, you have learned how to improve array construction using the JavaScript Array.of() method in ES6.

- Array.from() – create arrays from array-like or iterable objects.
- Array find() – find an element in an array
- Array findIndex() – find the index of an element in an array.

Section 11. Object extensions

- Object.assign() – copy an object or merge objects.

Using JavaScript Object.assign() to clone an object

The following example uses the Object.assign() method to clone an object.

```
let widget = {
   color: 'red'
};

let clonedWidget = Object.assign({}, widget);

console.log(clonedWidget);
```
Code language: JavaScript (javascript)

Output

```
{ color: 'red' }
```
Code language: CSS (css)

Note that the Object.assign() only carries a shallow clone, not a deep clone.

Using JavaScript Object.assign() to merge objects

The Object.assign() can merge source objects into a target object which has properties consisting of all the properties of the source objects. For example:

```
let box = {
   height: 10,
   width: 20
};

let style = {
   color: 'Red',
   borderStyle: 'solid'
};

let styleBox = Object.assign({}, box, style);

console.log(styleBox);
```
Code language: JavaScript (javascript)

Output:

```
{
   height: 10,
   width: 20,
   color: 'Red',
   borderStyle: 'solid'
}
```
Code language: CSS (css)

If the source objects have the same property, the property of the later object overwrites the earlier one:

```
let box = {
   height: 10,
   width: 20,
   color: 'Red'
};

let style = {
   color: 'Blue',
   borderStyle: 'solid'
};

let styleBox = Object.assign({}, box, style);

console.log(styleBox);
```
Code language: JavaScript (javascript)

Output:

```
{
   height: 10,
   width: 20,
   color: 'Blue',
   borderStyle: 'solid'
}
```

Code language: CSS (css)

Summary

- Object.assign() assigns enumerable and own properties from a source object to a target object.
- Object.assign() can be used to clone an object or merge objects.

- Object.is() – check if two values are the same value.

Section 12. String extensions

- String startsWith() – check if a string starts with another string.

  The startsWith() returns true if a string begins with the characters of a specified string; otherwise false.

  The following shows the syntax of the startsWith() method:

  ```
  String.startsWith(searchString [,position])
  ```

  Code language: CSS (css)

  Arguments

  - searchString is the characters to be searched for at the start of this string.
  - position is an optional parameter that determines the start position to search for the searchString. It defaults to 0.

  JavaScript String startsWith() examples

  Suppose that you have a string called title as follows:

  ```
  const title = 'Jack and Jill Went Up the Hill';
  ```

  Code language: JavaScript (javascript)

  The following example uses the startsWith() method to check if the title starts with the string 'Jack':

  ```
  console.log(title.startsWith('Jack'));
  ```

  Code language: JavaScript (javascript)

  Output:

true
Code language: JavaScript (javascript)

The startsWith() method matches characters case-sensitively, so the following statement returns false:

title.startsWith('Jack');
Code language: JavaScript (javascript)

This example uses the startsWith method with the second parameter that determines the begin position to start searching:

console.log(title.startsWith('Jill', 9));
Code language: JavaScript (javascript)

Output:

true
Code language: JavaScript (javascript)

Put it all together:

const title = 'Jack and Jill Went Up the Hill';

console.log(title.startsWith('Jack'));
console.log(title.startsWith('jack'));
console.log(title.startsWith('Jill', 9));
Code language: JavaScript (javascript)

Output:

true
false
true
Code language: JavaScript (javascript)

- String endsWith() – determine if a string ends with another string.
- String includes() – check if a string contains another string.

Section 13. Proxy & Reflection

- Proxy – learn how to use the proxy object that wraps another object (target) and intercepts the fundamental operations of the target object.

What is a JavaScript Proxy object

A JavaScript Proxy is an object that wraps another object (target) and intercepts the fundamental operations of the target object.

The fundamental operations can be the property lookup, assignment, enumeration, and function invocations, etc.

Creating a proxy object

To create a new proxy object, you use the following syntax:

```
let proxy = new Proxy(target, handler);
```
Code language: JavaScript (javascript)

In this syntax:

- target – is an object to wrap.
- handler – is an object that contains methods to control the behaviors of the target. The methods inside the handler object are called traps.

A simple proxy example

First, define an object called user:

```
const user = {
   firstName: 'John',
   lastName: 'Doe',
   email: 'john.doe@example.com',
}
```
Code language: JavaScript (javascript)

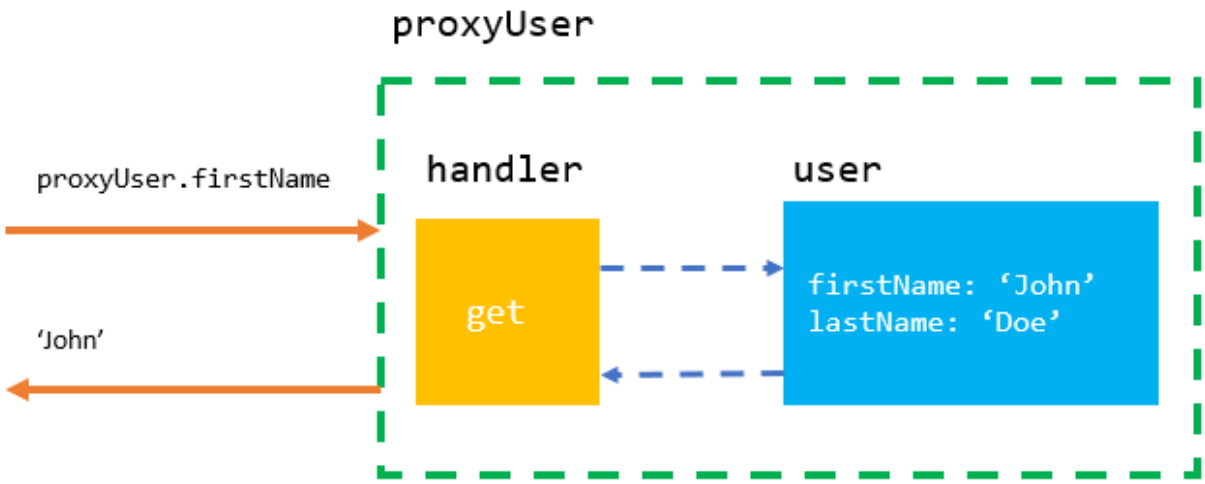
Second, define a handler object:

```
const handler = {
   get(target, property) {
      console.log(`Property ${property} has been read.`);
      return target[property];
   }
}
```
Code language: JavaScript (javascript)

Third, create a proxy object:

```
const proxyUser = new Proxy(user, handler);
```
Code language: JavaScript (javascript)

The proxyUser object uses the user object to store data. The proxyUser can access all properties of the user object.



Fourth, access the firstName and lastName properties of the user object via the proxyUser object:

```
console.log(proxyUser.firstName);
console.log(proxyUser.lastName);
```
Code language: CSS (css)

Output:

```
Property firstName has been read.
John
Property lastName has been read.
Doe
```

When you access a property of the user object via the proxyUser object, the get() method in the handler object is called.

Fifth, if you modify the original object user, the change is reflected in the proxyUser:

```
user.firstName = 'Jane';
console.log(proxyUser.firstName);
```
Code language: JavaScript (javascript)

Output:

```
Property firstName has been read.
Jane
```

Similarly, a change in the proxyUser object will be reflected in the original object (user):

```
proxyUser.lastName = 'William';
console.log(user.lastName);
```
Code language: JavaScript (javascript)

Output:

```
William
```

Proxy Traps

The **get()** trap

The get() trap is fired when a property of the target object is accessed via the proxy object.

In the previous example, a message is printed out when a property of the user object is accessed by the proxyUser object.

Generally, you can develop a custom logic in the get() trap when a property is accessed.

For example, you can use the get() trap to define computed properties for the target object. The computed properties are properties whose values are calculated based on values of existing properties.

The user object does not have a property fullName, you can use the get() trap to create the fullName property based on the firstName and lastName properties:

```
const user = {
   firstName: 'John',
   lastName: 'Doe'
}

const handler = {
   get(target, property) {
      return property === 'fullName' ?
         `${target.firstName} ${target.lastName}` :
         target[property];
   }
};

const proxyUser = new Proxy(user, handler);

console.log(proxyUser.fullName);
```
Code language: JavaScript (javascript)

Output:

```
John Doe
```

The **set()** trap

The set() trap controls behavior when a property of the target object is set.

Suppose that the age of user must be greater than 18. To enforce this constraint, you develop a set() trap as follows:

```
const user = {
   firstName: 'John',
   lastName: 'Doe',
   age: 20
}

const handler = {
   set(target, property, value) {
      if (property === 'age') {
         if (typeof value !== 'number') {
            throw new Error('Age must be a number.');
         }
         if (value < 18) {
            throw new Error('The user must be 18 or older.')
         }
      }
      target[property] = value;
   }
};

const proxyUser = new Proxy(user, handler);
```
Code language: JavaScript (javascript)

First, set the age of user to a string:

```
proxyUser.age = 'foo';
```
Code language: JavaScript (javascript)

Output:

```
Error: Age must be a number.
```
Code language: JavaScript (javascript)

Second, set the age of the user to 16:

```
proxyUser.age = '16';
```
Code language: JavaScript (javascript)

Output:

The user must be 18 or older.

Third, set the age of the user to 21:

```
proxyUser.age = 21;
```

No error occurred.

The apply() trap

The handler.apply() method is a trap for a function call. Here is the syntax:

```
let proxy = new Proxy(target, {
   apply: function(target, thisArg, args) {
      //...
   }
});
```
Code language: JavaScript (javascript)

See the following example:

```
const user = {
   firstName: 'John',
   lastName: 'Doe'
}

const getFullName = function (user) {
   return `${user.firstName} ${user.lastName}`;
}


const getFullNameProxy = new Proxy(getFullName, {
   apply(target, thisArg, args) {
      return target(...args).toUpperCase();
   }
});

console.log(getFullNameProxy(user)); //
```
Code language: JavaScript (javascript)

Output

```
JOHN DOE
```
More traps

The following are more traps:

- construct – traps usage of the new operator
- getPrototypeOf – traps an internal call to [[GetPrototypeOf]]
- setPrototypeOf – traps a call to Object.setPrototypeOf
- isExtensible – traps a call to Object.isExtensible
- preventExtensions – traps a call to Object.preventExtensions
- getOwnPropertyDescriptor – traps a call to Object.getOwnPropertyDescriptor

In this tutorial, you have learned about the JavaScript Proxy object used to wrap another object to change the fundamental behaviors of that object.

- Reflection – show you how to use ES6 Reflection API to manipulate variables, properties, and methods of objects at runtime.