Lab 08: ES6 Features

Objective(s):

1. Learn ES6 Features

Lab Task(s):

Exercises

1. Write a function called **raceResults** which accepts a single array argument. It should return an object with the keys first, second, third, and rest.
   - first: the first element in the array
   - second: the second element in the array
   - third: the third element in the array
   - rest: all other elements in the array

Write a one line function to make this work using

- An arrow function
- Destructuring
- ‘Enhanced’ object assignment (same key/value shortcut)

```
raceResults(['Tom', 'Margaret', 'Allison', 'David', 'Pierre'])
/*
 {
  first: "Tom",
  second: "Margaret",
  third: "Allison",
  rest: ["David", "Pierre"]
 }
*/
```

2. Write a function which generates an animal object. The function should accepts 3 arguments:
   - species: the species of animal (‘cat’, ‘dog’)
   - verb: a string used to name a function (‘bark’, ‘bleet’)
   - noise: a string to be printed when above function is called (‘woof’, ‘baaa’)

   Use one or more of the object enhancements we’ve covered.

```
const d = createAnimal("dog", "bark", "Wooooof!")
// {species: "dog", bark: ƒ}
d.bark()  //"Wooooof!"


const s = createAnimal("sheep", "bleet", "BAAAAaaaa")
// {species: "sheep", bleet: ƒ}
s.bleet() //"BAAAAaaaa"
```

3. Write the following functions using rest, spread and refactor these functions to be arrow functions!
   Make sure that you are always returning a new array or object and not modifying the existing inputs.

```
/** Return a new array with every item in array1 and array2. */
function extend(array1, array2) {
}


/** Return a new object with all the keys and values
from obj and a new key/value pair */
function addKeyVal(obj, key, val) {
}


/** Return a new object with a key removed. */
function removeKey(obj, key) {
}
```

```
/** Combine two objects and return a new object. */
function combine(obj1, obj2) {
}

/** Return a new object with a modified key and value. */
function update(obj, key, val) {
}
```

4. The built-in function setTimeout uses callbacks. Create a promise-based alternative.
   The function delay(ms) should return a promise. That promise should resolve after ms milliseconds, so that we can add .then to it, like this:

```
function delay(ms) {
  // your code
}

delay(3000).then(() => alert('runs after 3 seconds'));
```

5. Rewrite this example code from the chapter Promises chaining using async/await instead of .then/catch:

```
function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new Error(response.status);
      }
    });
}

loadJson('no-such-user.json')
  .catch(alert); // Error: 404
```

6. Below you can find the "rethrow" example. Rewrite it using async/await instead of .then/catch.

   And get rid of the recursion in favour of a loop in demoGithubUser: with async/await that becomes easy to do.

```
class HttpError extends Error {
  constructor(response) {
    super(`${response.status} for ${response.url}`);
    this.name = 'HttpError';
    this.response = response;
  }
}

function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new HttpError(response);
      }
    });
}

// Ask for a user name until github returns a valid user
function demoGithubUser() {
  let name = prompt("Enter a name?", "iliakan");

  return loadJson(`https://api.github.com/users/${name}`)
    .then(user => {
      alert(`Full name: ${user.name}.`);
      return user;
    })
    .catch(err => {
      if (err instanceof HttpError && err.response.status == 404) {
        alert("No such user, please reenter.");
        return demoGithubUser();
      } else {
        throw err;
      }
    });
}

demoGithubUser();
```

7. We have a “regular” function called `f`. How can you call the async function wait() and use its result inside of `f`?

```
async function wait() {

  await new Promise(resolve => setTimeout(resolve, 1000));

  return 10;

}
```

```
function f() {

  // ...what should you write here?

  // we need to call async wait() and wait to get 10

  // remember, we can't use "await"

}
```

8. Write a function called inOrder that accepts two callbacks and invokes them in order. Implement inOrder using the callback pattern.

```
var logOne = setTimeout(function() {
  console.log("one!");
}, Math.random() * 1000);

var logTwo = setTimeout(function() {
  console.log("two!");
}, Math.random() * 1000);

inOrder(logOne, logTwo);

// one
// two

// it should always log those two in order regardless of their timing
```

- Refactor inOrder to use promises.

- Implement a simple version of **Promise.all**. This function should accept an array of promises and return an array of resolved values. If any of the promises are rejected, the function should catch them.