JavaScript

STUDIA PODYPLOMOWE
POLITECHNIKA BIAŁOSTOCKA


HOMEWORK

IIFE

- Using IIFE and closure create a stopwatch function, that will console log time passed since the IIFE invocation

example.js

```
1  const logPassedTime = (function logPassedTimeIIFE() {
2    const timestamp = Date.now();
3
4    return function logger() {
5      console.log(Date.now() - timestamp);
6    };
7  })();
8
9  for (let i = 0; i < 10_000_000; i++) {}
10
11 logPassedTime();
12
```

Module Pattern

- Create a math module
- Use Module Pattern with IIFE
- The module should have 4 functions
  - add, subtract, divide, multiply

example.js

```
'use strict'

const mathModule = (function () {
    return {
        add(a, b) {
            return a + b;
        },
        subtract(a, b) {
            return a - b;
        },
        divide(a, b) {
            return a / b;
        },
        multiply(a, b) {
            return a * b;
        },
    };
})();

```

Module Pattern

- Add another method - repeat
- This function should repeat the last operation on the result of that operation
- ex. last operation: 2+2 = 4
  - repeat should do 4+2 = 6
- ex. last operation: 4*5 = 20
  - repeat should do 20*5 = 100

example.js

```
'use strict'

const mathModule = (function () {
    let lastOperation;
    let lastB;
    let lastResult;

    function cacheLastOperation(operation, b, result) {
        lastOperation = operation;
        lastB = b;
        lastResult = result;
    }

    function add(a, b) {
        return a + b;
    }

    function subtract(a, b) {
        return a - b;
    }

    function divide(a, b) {
        return a / b;
    }

    function multiply(a, b) {
        return a * b;
    }

    return {
        add(a, b) {
            const result = add(a, b);
            console.log(result);
            cacheLastOperation(add, b, result);
            return result;
        },
        subtract(a, b) {
            const result = subtract(a, b);
            cacheLastOperation(subtract, b, result);
            return result;
        },
        divide(a, b) {
            const result = divide(a, b);
            cacheLastOperation(divide, b, result);
            return result;
        },
        multiply(a, b) {
            const result = multiply(a, b);
            cacheLastOperation(multiply, b, result);
            return result;
        },
        repeat() {
            return lastOperation(lastResult, lastB);
        },
    };
})();

console.log(mathModule.add(2, 3));
console.log(mathModule.repeat());
```

Revealing Module Pattern

- Use revealing module pattern
- Make multiple calls to repeat work
- Try using the wrapper function pattern

example.js

```js
'use strict';

const mathModule = (function () {
    let lastOperation;
    let lastB;
    let lastResult;

    function withCache(operation) {
        return function (a, b) {
            const result = operation(a, b);
            lastOperation = withCache(operation);
            lastB = b;
            lastResult = result;
            return result;
        };
    }

    function add(a, b) {
        return a + b;
    }

    function subtract(a, b) {
        return a - b;
    }

    function divide(a, b) {
        return a / b;
    }

    function multiply(a, b) {
        return a * b;
    }

    function repeat() {
        return lastOperation(lastResult, lastB);
    }

    return {
        add: withCache(add),
        subtract: withCache(subtract),
        divide: withCache(divide),
        multiply: withCache(multiply),
        repeat,
    };
})();

console.log(mathModule.add(2, 3)); // 5
console.log(mathModule.repeat()); // 8
console.log(mathModule.repeat()); // 11
```

ES Modules

- Refactor with ESM

example.js

```
let lastOperation;
let lastB;
let lastResult;

function withCache(operation) {
  return function (a, b) {
    const result = operation(a, b);
    lastOperation = withCache(operation);
    lastB = b;
    lastResult = result;
    return result;
  };
}

const add = withCache((a, b) => a + b);
const subtract = withCache((a, b) => a - b);
const multiply = withCache((a, b) => a * b);
const divide = withCache((a, b) => a / b);

function repeat() {
  return lastOperation(lastResult, lastB);
}

export { add, subtract, multiply, divide, repeat };

console.log(add(2, 2)); // 4
console.log(repeat()); // 6
console.log(repeat()); // 8
```

Singleton

- Create a function that connects the app to a database and returns the connection object
- There should be 1 connection to the database in our app
- Utilise mongoose.createConnection

example.js

```
import mongoose from 'mongoose';

let connection;

async function getdbConnection() {
  if (!connection) {
    connection = await mongoose.createConnection('mongodb://127.0.0.1:27017/myapp');
  }

  return connection;
}

export default getdbConnection;
```

Wrapper

- Create a wrapper for an async function, so when the function fails it would log the current time, date, function's name and error message

example.js

```
function withAsyncDebug(callback) {
  return async function withAsyncDebugWrapper(...args) {
    try {
      await callback(...args);
    } catch (e) {
      console.log(`${Date.now()} Error with ${callback.name}, message: ${e.message}`);
    }
  };
}

const fetchWithDebug = withAsyncDebug(fetch);

fetchWithDebug('example.nonexist.com');
// 1712590673198 Error with fetch, message: Failed to parse URL from example.nonexist.com
```

Wrapper 2

- Create a wrapper that memoize the result of a function call with a given set of arguments
- when the function is called multiple times with the same set of arguments, the functions should return result from cache instead of running again

example.js

```
function withMemo(callback) {
  const cache = {};

  return function withMemoWrapper(...args) {
    const stringArgs = JSON.stringify(args);

    if (!cache[stringArgs]) {
      cache[stringArgs] = callback(...args);
    }

    return cache[stringArgs];
  };
}

function add2(num) {
  return num + 2;
}

const add2Memo = withMemo(add2);

console.log(add2Memo(2)); // 4
console.log(add2Memo(2)); // 4
console.log(add2Memo(3)); // 5
```

Curry

- Create a logger function that takes 3 arguments - title, console method, message
- Available console methods: log, warn, error
- It should output to the console with a selected method, title and message

example.js

```
const METHOD = {
  log: 'log',
  warn: 'warn',
  error: 'error',
};

function log(title, method, message) {
  try {
    console[method](`${title}: ${message}`);
  } catch {
    console.error('Unsupported log method');
  }
}

log('Debug', METHOD.warn, 'This is a warning');
```

Curry

- Let’s curry it


example.js
const METHOD = {
log: 'log',
warn: 'warn',
error: 'error',
};
function log(title) {
return function logWithTitle(method) {
return function logWithTitleAndMethod(message) {
try {
console[method](`${title}: ${message}`);
} catch {
console.error('Unsupported log method');
}
};
};
}
log('Debug')(METHOD.warn)('This is a warning');

Curry

- Create specialized functions:
  - “debug” logger with all available methods
  - “production”. error logger

example.js

```
const METHOD = {
  log: 'log',
  warn: 'warn',
  error: 'error',
};

function log(title) {
  return function logWithTitle(method) {
    return function logWithTitleAndMethod(message) {
      try {
        console[method](`${title}: ${message}`);
      } catch {
        console.error('Unsupported log method');
      }
    };
  };
}

const logDebug = log('Debug');

logDebug(METHOD.log)('Sample debug log.'); // Debug: Sample debug log.

const logProduction = log('Production');
const logProductionError = logProduction(METHOD.error);

logProductionError('Sample production error!'); // Production: Sample production error!
```

Composition

- Create three functions:
  - filterEvenNumbers
  - mapToSquare
  - sumNumbers
- Compose the functions using compose

composition.js

```
function compose(...functions) {
  return function (input) {
    return functions.reduceRight(function (acc, fn) {
      return fn(acc);
    }, input);
  };
}

const filterEvenNumbers = (numbers) => numbers.filter((num) => num % 2 !== 0);

const mapToSquare = (numbers) => numbers.map((num) => num * num);

const sumNumbers = (numbers) => numbers.reduce((acc, num) => acc + num, 0);

const processNumbers = compose(sumNumbers, mapToSquare, filterEvenNumbers);

const numberArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
console.log(processNumbers(numberArray)); // Output: 165
```

Decorator

- Create a class decorator that adds a static create method to any class it decorates
- The create method should return a new instance of the decorated class with some default values

example.js

```
class User {
  constructor(name, id) {
    this.name = name;
    this.id = id;
  }
}

function withSample(InputClass) {
  Object.defineProperty(InputClass, 'create', {
    value: () => new InputClass('Bob', 2),
  });
}

withSample(User);

const sampleUser = User.create();

console.log(sampleUser); // User { name: 'Bob', id: 2 }
```

Mixin

- Create a mixin that adds a loadData method to an object, which fetches data from a URL and sets it on a data property on the object

example.js

```
const userObject = {
  name: 'Bob',
  id: 123,
};

const loadDataMixin = {
  loadData: async function fetchAndSetData(url) {
    const res = await fetch(url);
    const data = await res.json();
    this.data = data;
  },
};

Object.assign(userObject, loadDataMixin);

userObject.loadData('https://api.github.com/users/dyrpit').then(() => console.log(userObject));
```

HOMEWORK

- Treasure hunt

EXAM

- Bouncing ball
- Game of life