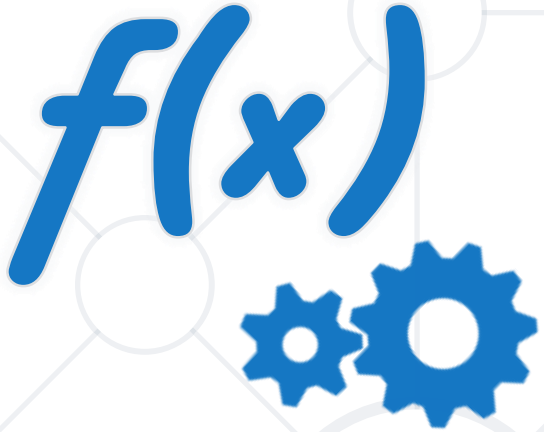
Advanced Functions

First-Class Functions, Function Expressions, IIFE, this, call, apply



SoftUni Team

Technical Trainers

Software University

http://softuni.bg

Table of Content

SoftUni Foundation

Have a Question?

SoftUni Foundation

sli.do

#JS-CORE

First Class Functions

Functions behaving like variables

First-Class Functions

A function can be passed as an **argument** to other function, can be **returned** by another function and can be **assigned** as a value to a variable.

- Assign a function to a **variable**

```
const write = function () {
  console.log("Hello, world!");
}

write();          // Hello, world!
```

First-Class Functions

- Pass a function as an **Argument**

```
function sayHello() {
  return "Hello, ";
}
function greeting (helloMessage, name) {
  console.log(helloMessage() + name);
}
greeting (sayHello, "JavaScript!");
//Hello, JavaScript!
```

Pass 'sayHello' as an argument to 'greeting' function

First-Class Functions

- Return a function

```
function sayHello() {
    return function() {
        console.log('Hello!');
    }
}
```

Higher-Order Functions

Take other **functions as argument** or **return a function** as result

```
const sayHello = function() {
  return function() {
     console.log("Hello!");
  }
}

const myFunc = sayHello();
myFunc();
//Hello!
```

Higher-Order Functions

The **filter()** method creates a new array with all elements that pass the test implemented by the provided function.

```
const words = ['JavaScript', 'programming', 'development', 'code'];

const result = words.filter(word => word.length > 6);

console.log(result);

// Expected output: Array["JavaScript", "programming", "development
```

Higher-Order Functions

The **map()** method creates a new array with the results of calling a provided function on every element in the calling array.

```
let arr = [4, 2, 1, 5];

const map1 = arr.map(x => x * 2);

console.log(map1);

//Expected output: Array [8, 4, 2, 10]
```

Higher-Order Functions

The **reduce()** method executes a **reducer** function on each member of the array resulting in a single output value.

```
const array1 = [1, 2, 3, 4];

const reducer = (acc, cur) => acc + cur;
// 1 + 2 + 3 + 4

console.log(array1.reduce(reducer));
// Expected output: 10

console.log([].reduce(reducer, 5));
// Expected output: 5
```

Problem: Aggregates

Write a JS program that uses a **reducer** function to **display** information about an **input array**.

```
2
3
10
5
```

```
Sum = 20
Min = 2
Max = 10
Product = 300
Join = 23105
```

```
5
-3
20
7
0.5
```

```
Sum = 29.5
Min = -3
Max = 20
Product = -1050
Join = 5-32070.5
```

Solution: Aggregates

```
function solution(arr) {
    console.log('Sum = ' + arr.reduce((a,b) => a + b));
    console.log('Min = ' + arr.reduce((a,b) => Math.min(a,b)));
    console.log('Max = ' + arr.reduce((a,b) => Math.max(a,b)));
    console.log('Product = ' + arr.reduce((a,b) => a * b));
    console.log('Join = ' + arr.reduce((a,b) => '' + a + b));
}
```

Check your solution here: https://judge.softuni.bg/Contests/1582

Currying

SoftUni Foundation

Currying is the process of **breaking down** a function into a series of functions - each takes a **single** argument.

Takes three numbers as input

```
function sum3(a, b, c) {
    return a + b + c;
}
console.log(sum3(5,6,8);
// Expected output: 19
```

Takes one parameter and returns a function

```
function sum3(a) {
    return (b) => {
        return (c) => {
            return a + b + c;
        }
    }
}
console.log(sum3(5)(6)(8)); //19
```

Currying Usage

- **Function Composition** - Building new function from old function by passing arguments.
- **Memoization** - Functions that are called repeatedly with the same set of inputs but whose result is relatively expensive to produce.
- **Handle Errors** - Throwing functions and exiting immediately after an error.

Partial Application

- Converting a function with a **given number** of arguments into a function with **smaller number** of arguments
- Pass the **remaining parameters** when a final **result** is needed
  - The partially applied function can be **used multiple times**
- This helps write **reusable code** with **fewer bugs**

Set first parameter to 1

Same as increment operator (++)

```
f(x, y) = x + y
```

```
g(x) = f(1, x)
```

Currying vs Partial Application

- **Currying** always produces nested unary (1-ary) functions.
- **Partial** application produces functions of arbitrary number of arguments.
- Currying is **not** partial application. It can be implemented using partial application.

Problem: Currency Format

Your program will receive a function that **takes 4 parameters** and **returns** a formatted string.

Your task is to **return** another function that only takes **1 parameter** and returns the same formatted string.

```
let formatter = getDollarFormatter(formatCurrency);
formatter(5345); // $ 5345,00
```

Solution: Currency Format

- We take the initial **function as parameter**
- We **return a function** that takes only one parameter

```
function getDollarFormatter(formatter) {
  function dollarFormatter(value) {
      return formatter(',', '$', true, value);
  };
  return dollarFormatter;
}
```

Fix parameters

Return result of original function

Check your solution here: https://judge.softuni.bg/Contests/1582

Immediately-Invoked Function Expressions (IIFE)

What is IIFE?

Immediately-Invoked Function Expressions (IIFE)

- Define anonymous function expression
- Invoke it immediately after declaration

```
(function() { console.log("invoked!"); }());
```

```
(function() { console.log("invoked!"); })();
```

```
let iife = function() { console.log("invoked!"); }();
```

Functions Returning Functions

A **state** is preserved in the outer function, a.k.a. **closure**

```
let f = (function() {
  let counter = 0;
  return function() {
    console.log(++counter);
  }
})();
```

```
f(); // 1
f(); // 2
f(); // 3
f(); // 4
f(); // 5
f(); // 6
f(); // 7
```

Problem: Command Processor

SoftUni Foundation

Write a JS program that keeps a string **inside it's context** and can execute different **commands** that modify or output the string on the console.

- **append(str)** - add **str** to the end of the internal string
- **removeStart(n)** - **remove** the **first n** characters from the string, **n** is an integer
- **removeEnd(n)** - **remove** the **last n** characters from the string, **n** is an integer
- **print** - **output** the stored string to the **console**

Solution: Command Processor

```
function solve(arr) {
    let closure = (function () {
        let str = '';
        return {
            append: (s) => str += s,
            removeStart: (n) => str = str.substring(n),
            removeEnd: (n) => str = str.substring(0, str.length - n),
            print: () => console.log(str)
        }
})();                                   // Continue on the next slide…
```

Solution: Command Processor

- We loop through **all** elements in the array.
- **Execute** the closure with the current values.

```
for (let st of arr) {
        let [comm, value] = st.split(' ');
        closure[comm](value);
  }
}
```

Check your solution here: https://judge.softuni.bg/Contests/1582

Function "this" Context

this, call, apply, bind

What is Function Context?

The **function context** is the object that "**owns**" the currently executed code

- Function context == "**this**" object
- Depends on how the function is invoked
  - Global invoke: **func()**
  - **object.function()**
  - **domElement.event()**
  - Using **call()** / **apply()** / **bind()**

The Function Context

```
function f() {
  console.log(this);
}
f(); // Window ("this" is the global context)
```

```
function f() {
  'use strict';
  console.log(this);
}
f(); // undefined ("this" is missing)
```

The Function Context with Object

```
function func() {
  console.log(this);
}

let obj = {
  name: 'Peter',
  f: func
};

obj.f(); // Object {name: "Peter"}
```

The Function Context for Objects

SoftUni Foundation

```
let obj = {
  name: 'Todor',
  getName: function () {
    return this.name;  // "this" refers to "obj"
  }
};
console.log(obj.getName()); // Todor
```

```
function Car() {
  console.log(this);
}
let car = new Car(); // Car {}
```

The Function Context with Inner Function

```
function outer() {
  console.log(this); // Object {name: "Peter"}
  function inner() {
    console.log(this); // Window
  }
  inner();
}
let obj = { name: 'Peter', f: outer }
obj.f();
```

The Function Context with Arrow Function

SoftUni Foundation

```
function outer() {
  let inner = () => console.log(this);
  inner();
}
let obj = {
  name: 'Peter',
  f: outer
};
obj.f(); // Object {name: "Peter"}
```

The Function Context for DOM Events

```
<button onclick="alert(this)">Click Me</button>
// Shows "[object HtmlButtonElement]" when clicked
```

```
<button onclick="f(this)">Click Me</button>
function f(btn) { alert(btn); };
// Shows "[object HtmlButtonElement]" when clicked
```

```
<button onclick="f()">Click Me</button>
function f() { alert(this); };
// Shows "[object Window]" when clicked
```

Avoided by using **addEventListener**

Changing the Context: Call

```
let sharePersonalInfo = function (){
    console.log(`Hello, my name is ${this.name}`.);
    console.log(`I'm a ${this.profession}.`);
}

let firstPerson = {name: "Peter", profession: "Fisherman"};
let secondPerson = {name: "George", profession: "Astronaut"};

// Continue on the next slide…
```

Changing the Context: Call

```
sharePersonalInfo.call(firstPerson);
// Hello, my name is Peter.
// I'm a Fisherman.

sharePersonalInfo.call(secondPerson);
// Hello, my name is George.
// I'm a Astronaut.
```

Changing the Context: Apply

SoftUni Foundation

```
let firstPerson = {
    name: "Peter",
    prof: "Fisherman",
    shareInfo: function() {
        console.log(`${this.name} work as ${this.prof}`);
    }
};
let secondPerson = {name: "George", prof: "Astronaut"};
firstPerson.shareInfo.apply(secondPerson);
// George work as Astronaut
```

Changing the Context: Bind

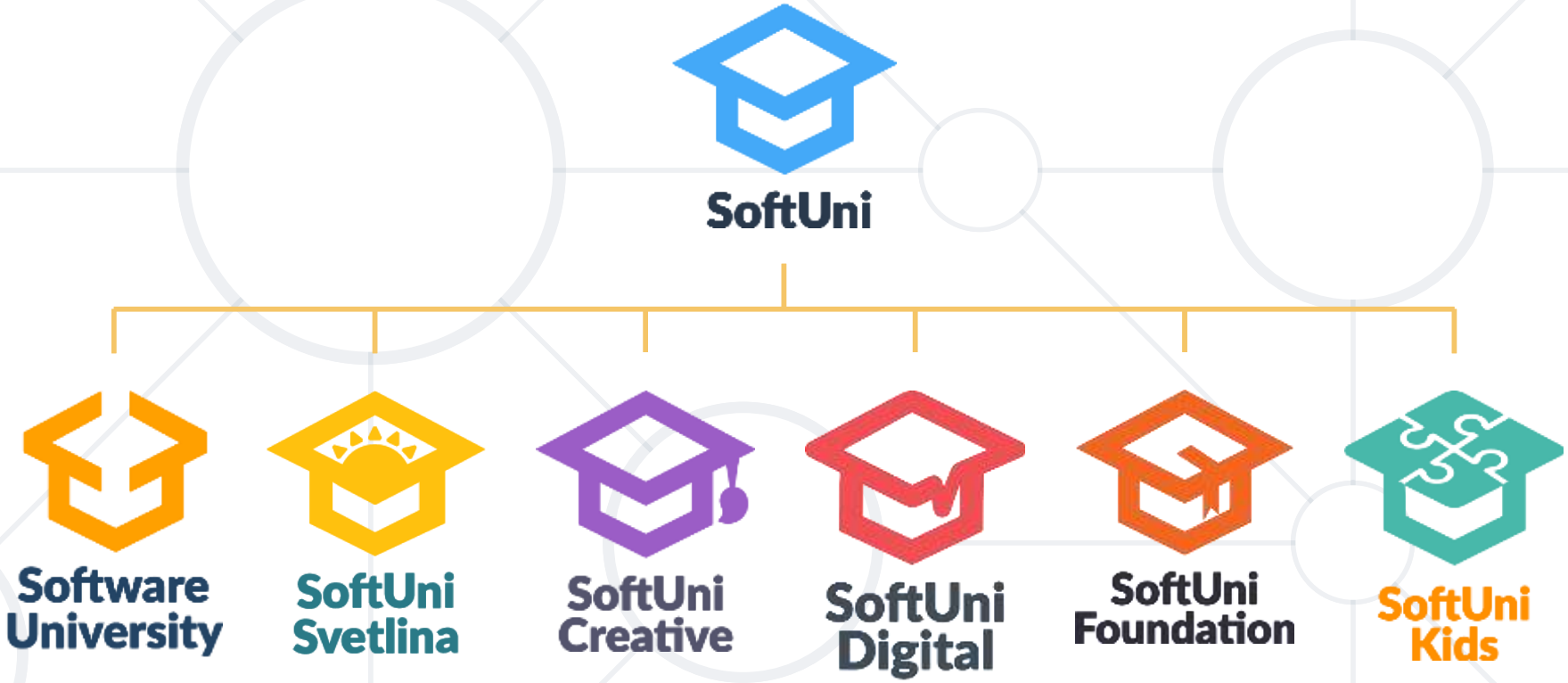
```
let module = {
  x: 42,
  getX: function() return this.x;
};
let unboundGetX = module.getX;
console.log(unboundGetX()); // undefined

let boundGetX = unboundGetX.bind(module);
console.log(boundGetX()); // 42
```

Live Exercises

Summary

- In JS, functions are objects (**first-class functions**)
- **IIFE** is immediately-invoked anonymous function
  - **Encapsulates** JS code + data (state)
- The **function context** "**this**" depends on how the function is invoked
  - Through object, as event-handler, inner function

Questions?



https://softuni.bg/courses/javascript-advanced

SoftUni Diamond Partners

SoftUni Foundation

XSsoftware

SBTech
we know sports

telenor

SoftwareGroup
doing it right

NETPEAK

SmartIT

Postbank
Решения за твоето утре

SUPER HOSTING .BG

INDEAVR
Serving the high achievers

INFRAGISTICS

LIEBHERR

æternity

SoftUni Organizational Partners

SoftUni Foundation

ИНФОРМАЦИОННО ОБСЛУЖВАНЕ

OneBit SOFTWARE

Lukanet.com

WORLD OF MYTHS

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education and Employment Opportunities
  - softuni.bg
- Software University Foundation
  - http://softuni.foundation/
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License

- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license