Advanced Functions

Function Context, First-Class Functions, Referential Transparency, Currying, IIFE, Closure

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#js-advanced

Table of Contents

Execution Context

Execution Context Review

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 method: `object.function()`
- DOM Event: `element.addEventListener()`
- Using `call()` / `apply()` / `bind()`

Inner Method Context

Software University

`this` variable is accessible only by the outer method

```
const obj = {
  name: 'Peter',
  outer() {
    console.log(this); // Object {name: "Peter"}

    function inner() { console.log(this); }

    inner();
  }
}

obj.outer(); // Window
```

Arrow Function Context

Software University

`this` retains the value of the **enclosing lexical context**

```
const obj = {
  name: 'Peter',
  outer() {
    console.log(this); // Object {name: "Peter"}

    const inner = () => console.log(this);

    inner();
  }
}

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

Explicit Binding

Occurs when `call()`, `apply()`, or `bind()` are used on a function

Forces a **function** call to **use** a particular **object** for this binding

```
function greet() {
  console.log(this.name);
}

let person = { name:'Alex' };
greet.call(person, arg1, arg2, arg3, ...); // Alex
```

Changing the Context: Call

Calls a function with a given **this** value and **arguments** provided individually

```
const sharePersonalInfo = function (...activities) {
  let info = `Hello, my name is ${this.name} and`+
           + `I'm a ${this.profession}.\n`;
  info += activities.reduce((acc, curr) => {
      let el  = `--- ${curr}\n`;
      return acc + el;
  }, "My hobbies are:\n").trim();
  return info;
}
// Continues on the next slide...
```

Changing the Context: Call

```
const firstPerson = { name: "Peter", profession: "Fisherman" };
console.log(sharePersonalInfo.call(firstPerson, 'biking',
'swimming','football'));
// Hello, my name is Peter.
// I'm a Fisherman.
// My hobbies are:
// --- biking
// --- swimming
// --- football
```

Changing the Context: Apply

Calls a function with a **given this value**, and **arguments** provided as an **array**

`apply()` accepts a **single array** of arguments, while `call()` accepts an **argument list**

If the first argument is **undefined** or **null** a similar outcome can be achieved using the array **spread syntax**

Apply() –Example

Software University

```
const firstPerson = {
  name: "Peter",
  prof: "Fisherman",
  shareInfo: function () {
    console.log(`${this.name} works as a ${this.prof}`);
  }
};
const secondPerson = { name: "George", prof: "Manager" };
firstPerson.shareInfo.apply(secondPerson);
// George works as a Manager
```

Changing the Context: Bind

Software University

The `bind()` method creates a **new function**

Has its `this` keyword **set** to the **provided** value, with a given sequence of arguments preceding any provided when the **new function** is called

Calling the bound function generally results in the **execution** of its **wrapped function**

Bind –Example

Software University

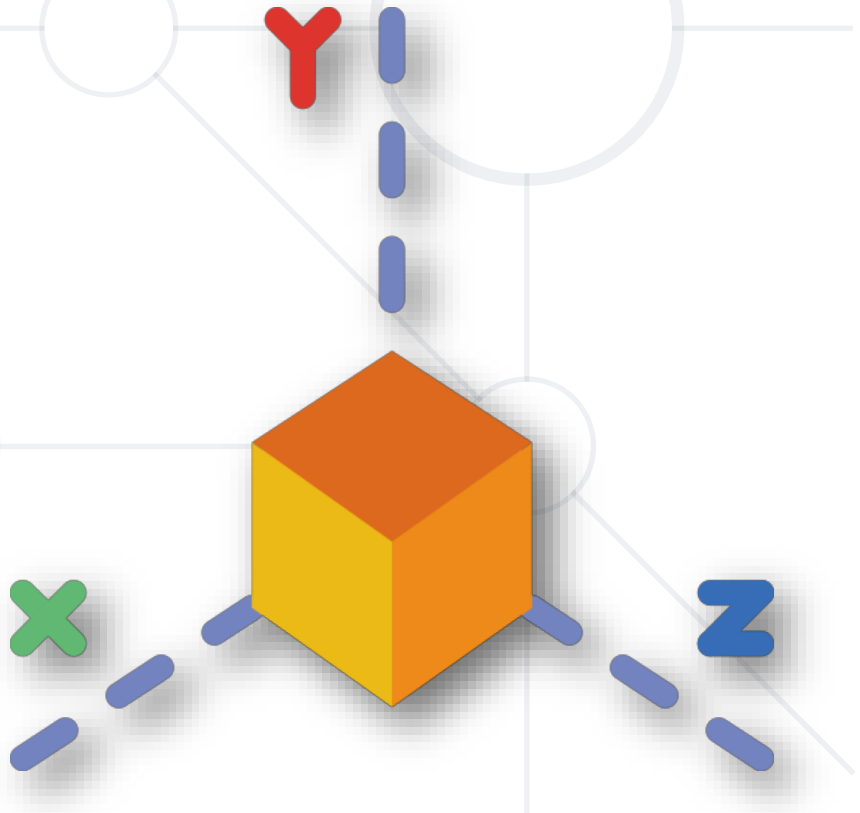
```
const x = 42;
const getX = function () {
  return this.x;
}
const module = {x , getX };
const unboundGetX = module.getX;
console.log(unboundGetX()); // undefined
const boundGetX = unboundGetX.bind(module);
console.log(boundGetX()); // 42
```

Problem: Area and Volume Calculator

The functions **area** and **vol** are **passed as parameters** to your function

```
function area() {
  return Math.abs(this.x * this.y);
};
```

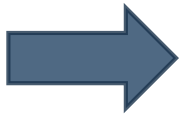
```
function vol() {
  return Math.abs(this.x * this.y *
this.z);
};
```



Problem: Area and Volume Calculator

Calculate the **area** and the **volume** of figures, which are defined by their coordinates (**x**, **y** and **z**), **using** the **provided functions**

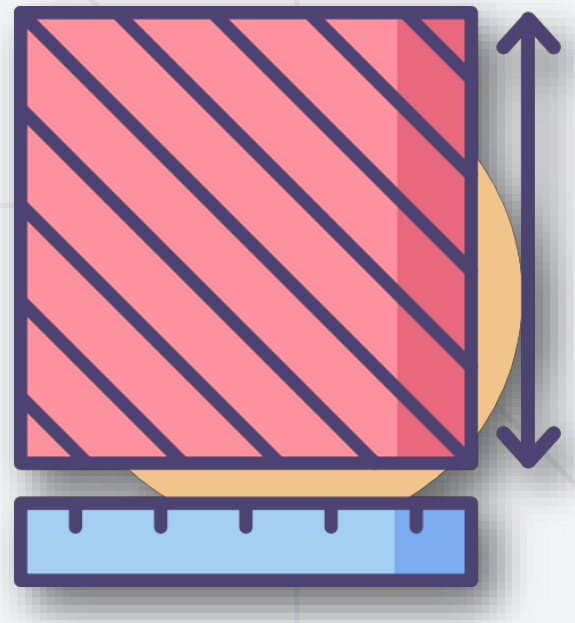
```
`[
{"x":"1","y":"2","z":"10"},
{"x":"7","y":"7","z":"10"},
{"x":"5","y":"2","z":"10"}
]`
```



```
[
  { area: 2, volume: 20 },
  { area: 49, volume: 490 },
  { area: 10, volume: 100 }
]
```

Solution: Area and Volume Calculator

```
function solve(area, vol, input) {
  let objects = JSON.parse(input);
  function calc(obj) {
    let areaObj = Math.abs(area.call(obj));
    let volumeObj = Math.abs(vol.call(obj));
    return { area: areaObj, volume: volumeObj }
  }
  return objects.map(calc);
}
```



Object Methods as Browser Event Handlers

Software University

```
const person = {
  name: "Peter",
  respond() {
    alert(`${this.name} says hello!`);
  }
}
const boundRespond = person.respond.bind(person);

documet.getElementById('callBtn')
       .addEventListener('click', person.respond);

documet.getElementById('callBtn')
       .addEventListener('click', boundRespond);
```

```
<body>
  <button id="callBtn">Call Person</button>
</body>
```

Unwanted result

Works as intended

Functional Programming in JS

First-Class Functions

First-class functions are treated like any other variable

- Passed as an **argument**
- **Returned** by another function
- Assigned as a **value** to a **variable**

> The term "first-class" means that something is just a value. A first-class function is one that can go anywhere that any other value can go - there are few to no restrictions.
>
> *Michael Fogus, Functional Javascript*

First-Class Functions

Software University

Can be passed as an **argument** to another function

```
function sayHello() {
    return "Hello, ";
}
```

```
function greeting(helloMessage, name) {
    return helloMessage() + name;
}
```

```
console.log(greeting(sayHello, "JavaScript!"));
// Hello, JavaScript!
```

First-Class Functions

Can be **returned** by another function

We can do that, because we treated functions in JavaScript as a **value**

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

First-Class Functions

Software University

Can be assigned as a **value** to a **variable**

```
const write = function () {
    return "Hello, world!";
}
```

```
console.log(write());
// Hello, world!
```

Higher-Order Functions

Software University

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

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

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

Predicates

Any function that returns a **bool based** on evaluation of the **truth** of an **assertion**

Predicates are often found in the form of **callbacks**

```
let found = array1.find(isFound);

function isFound(element) {
    return element > 10; //True or false
}

console.log(found); // 12
```

predicate

Built-in Higher Order Functions

Array.prototype.map

Array.prototype.filter

Array.prototype.reduce

```
users = [ { name: 'Tim', age: 25 },
          { name: 'Sam', age: 30 },
          { name: 'Bill', age: 20 } ];

getName = (user) => user.name;
usernames = users.map(getName);
console.log(usernames) // ["Tim", "Sam", "Bill"]
```

Pure Functions

Returns the **same result** given **same parameters**

Execution is **independent** of the state of the system

```
// impure function:
let number = 1;
const increment = () => number += 1;
increment(); // 2

// pure function:
const increment = n => n + 1;
increment(1); // 2
```

Referential Transparency

Software University

An **expression** that can be **replaced** with its corresponding **value** without **changing** the program's behavior

Expression is **pure** and its evaluation must have no **side effects**

```
function add(a, b) { return a + b };
function mult(a, b) { return a * b};
let x = add(2, mult(3, 4));
// mult(3, 4)) can be replaced with 12
```

Closure

Closure

Software University

One of the most **important features** in JavaScript

The **scope** of an inner function **includes** the scope of the outer function

An **inner** function retains **variables** being used from the **outer** function scope even after the parent function has **returned**

Functions Returning Functions

Software University

A **state** is preserved in the outer function (**closure**)

```
const 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

Write a program, which:

- Keeps a string **inside its scope**
- Can execute different **commands** that modify a string:
  - `append()` - add **str** to the end of the internal string
  - `removeStart()` - **remove** the **first n** characters
  - `removeEnd()` - remove the **last n** characters
  - `print()` - print the stored string

Solution: Command Processor

```
function solution() {
    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)
    }
}
```

Review: DOM Problems

Software University

Attempt to solve problems **Sections**, **Locked Profile** and **Furniture** from **previous exercises**, by using **closures** to store local **state** and **references**

What is IIFE?

Immediately-Invoked Function Expressions (IIFE)

Define **anonymous** function expression

Invoke it **immediately** after declaration

```
(function () { let name = "Peter"; })();
// Variable name is not accessible from the outside scope
console.log(name); // ReferenceError
```

```
let result = (function () {
    let name = "Peter";
    return name;
})();
// Immediately creates the output:
console.log(result); // Peter
```

Function Decoration

Partial Application

Set some of the **arguments** of a function, **without executing** it

Pass the **remaining arguments** when a result is needed

The partially applied function can be **used multiple times**

It will **retain** all fixed arguments, **regardless of context**

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

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

```
Math.pow(x,y)
```

```
sqr = (x) => Math.pow(x,2)
```

Problem: Currency Format

Software University

Receive **three primitives** and a **function `formatter`**

The **`formatter`** function takes **4 arguments**

Use the first **three parameters** of your solution to create and return a **partially applied** function that only takes 1 parameter

Sample usage:

```
const dollarFormatter =
  createFormatter(',', '$', true, currencyFormatter);
console.log(dollarFormatter(5345));   // $ 5345,00
console.log(dollarFormatter(3.1429)); // $ 3,14
console.log(dollarFormatter(2.709));  // $ 2,71
```

Solution: Currency Format

```
function createFormatter(separator,
                         symbol,
                         symbolFirst,
                         formatter) {
  return (value) => formatter(separator,
                              symbol,
                              symbolFirst,
                              value);
}
```

Partially applied arguments

Currying

Currying is a technique for **function decomposition**

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

Supply arguments **one at a time**, instead of at once

- They may come from **different sources**
- Execution can be **delayed** until it's needed

Currying Usage

Software University

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

Currying vs Partial Application

Software University

Currying always produces nested unary functions

Partial application produces functions of arbitrary number of arguments

Currying is **NOT** partial application

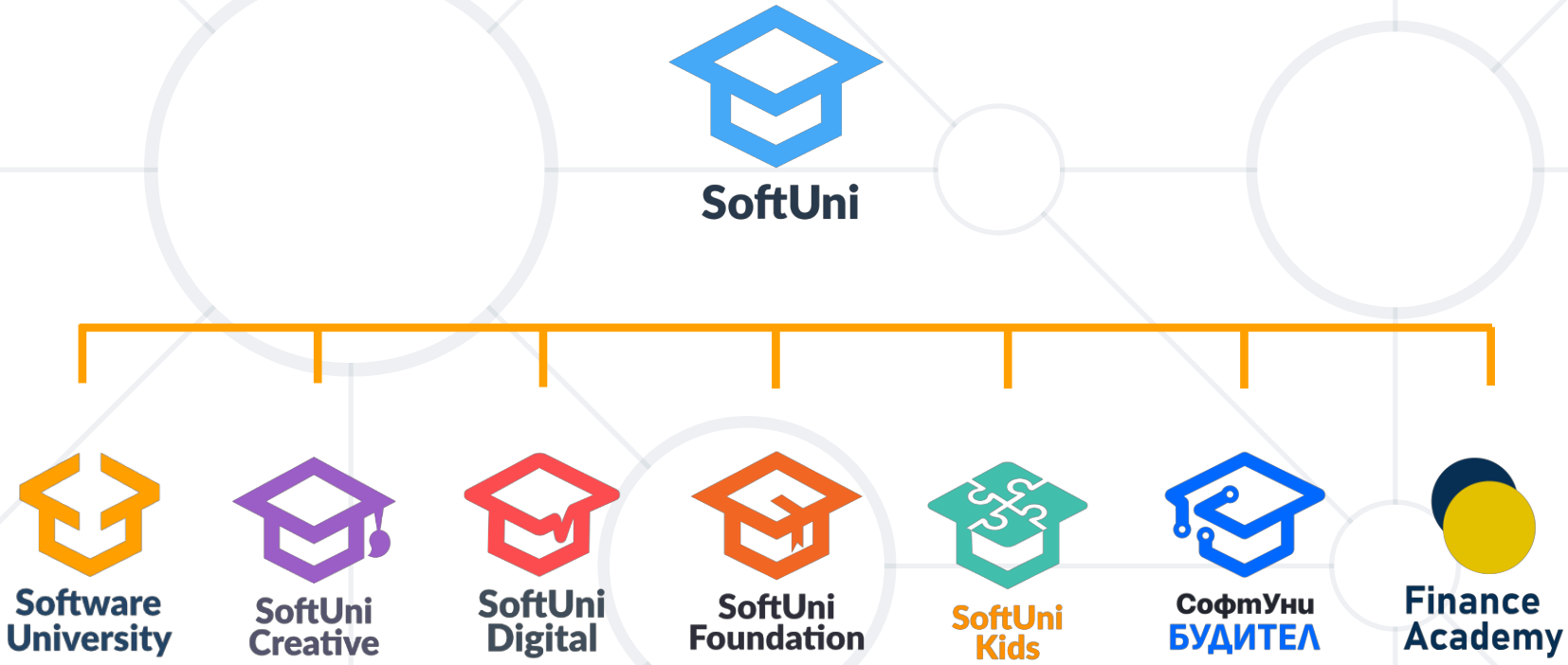
- It can be implemented using partial application

Summary

Software University

- The execution context of a function can be changed using bind, apply and call
- JavaScript supports many aspects of the functional programming paradigm
- Closures allow a function to maintain state
  - They are powerful and flexible
- Partial application can be used to decorate and compose functions and to delay execution

Questions?



SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

Решения за твоето утре

DRAFT KINGS

VIVACOM

AMBITIONED

INDEAVR

Serving the high achievers

SOFTWARE GROUP

PHAR VISION

License

This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**

Unauthorized copy, reproduction or use is illegal

© SoftUni – https://about.softuni.bg/

© Software University – https://softuni.bg

Trainings @ Software University (SoftUni)

Software University –High-Quality Education, Profession and Job for Software Developers

softuni.bg, about.softuni.bg

Software University Foundation

softuni.foundation

Software University @ Facebook

facebook.com/SoftwareUniversity