

# Functions and Statements

$$f(x)$$

SoftUni Team

Technical Trainers



**SoftUni**



Software University

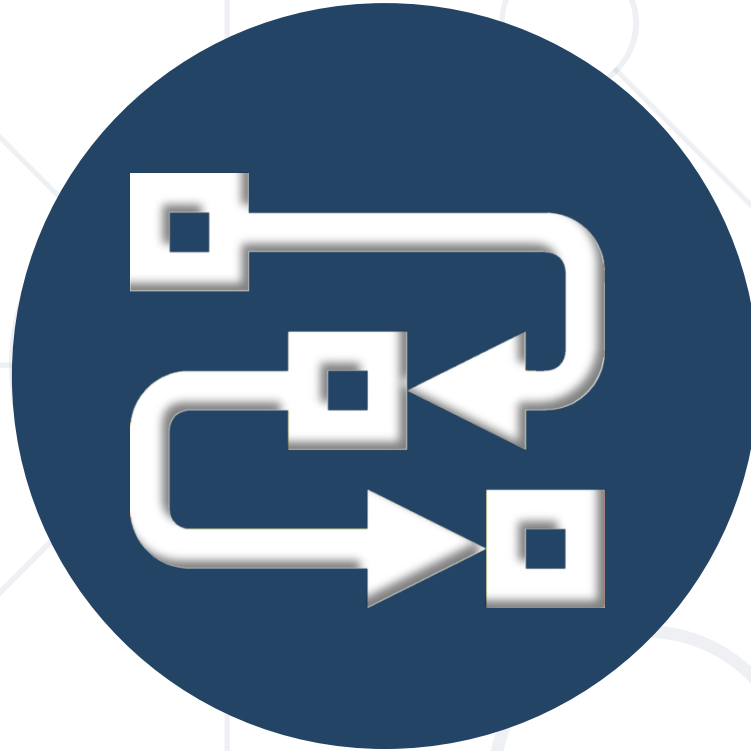
<https://softuni.bg>

1. Declaring and Invoking Functions
2. Nested Functions
3. Value vs Reference Types
4. First-class Functions
5. Arrow Functions
6. Naming and Best Practices



sli.do

**#js-front-end**



# Functions Overview

Definition and Objectives

# Functions in JS

- A **function** is a **named subprogram** designed to perform a particular task
  - Functions are executed when they are called. This is known as **invoking** a function
  - Values can be **passed** into functions and used within the function

Use **camelCase**

Parameter

```
function printStars(count) {  
  console.log("*".repeat(count));  
}
```



# Why Use Functions?

- More **manageable programming**
  - **Splits** large problems into small pieces
  - Better **organization** of the program
  - Improves code **readability** and **understandability**
- Avoiding **repeating code**
  - Improves code maintainability
- Code **reusability**
  - Using existing functions several times





# Declaring and Invoking Functions

# Declaring Function

- Functions can be declared in two ways:
  - **Function declaration** (recommended way)

```
function printText(text){  
    console.log(text);  
}
```

- **Function expression** (useful in functional programming)

```
let printText = function(text){  
    console.log(text);  
}
```





# Declaring Function

- Functions can have **parameters**
- Functions **always** return a value (custom or default)



```
function printText(text){  
  console.log(text);  
}
```

Name

Parameters

Body

# Invoking a Function

- Functions are first **declared**, then **invoked** (many times)

```
function hLine() {  
  console.log("-----");  
}
```

Function  
Declaration

- Functions can be **invoked (called)** by their name

```
hLine();
```

Function  
Invocation

# Invoking a Function (2)

- Invocation from another function:

```
function printDocument() {  
    printLabel();  
    printContent();  
}
```

Function invoking  
functions

- Self-invocation (**recursion**):

```
function countdown(x) {  
    console.log(x);  
    if (x > 0) { countdown(x - 1); }  
}
```

Function invoking  
itself

# Functions Without Parameters

- Does **not** receive arguments when invoked
- The result is **always the same** (unless it reads data from outside)

```
function printHeader() {  
    console.log('~~~-    {@}    -~~~');  
    console.log('~- Certificate -~');  
    console.log('~~~- ~---~ -~~~');  
}  
printHeader();    // Output is always the same
```

# Functions With Parameters

- Can receive **any number** and **type** of arguments when invoked

```
function multiply(a, b) {  
  console.log(a*b);  
}  
multiply(5, 7); // 35
```

Pass two numbers

```
function printName(nameArr) {  
  console.log(nameArr[0] + ' ' + nameArr[1]);  
}  
printName(['John', 'Smith']); // John Smith
```

Pass array of strings

# Problem : Format Grade

- Write a function that **receives a grade** between 2.00 and 6.00 and prints a formatted line with **grade and description**
  - Grade  $< 3.00 \rightarrow$  **Fail**
  - Grade  $\geq 3.00$  and  $< 3.50 \rightarrow$  **Poor**
  - Grade  $\geq 3.50$  and  $< 4.50 \rightarrow$  **Good**
  - Grade  $\geq 4.50$  and  $< 5.50 \rightarrow$  **Very good**
  - Grade  $\geq 5.50 \rightarrow$  **Excellent**

Input	Output
3.33	Poor (3.33)
4.50	Very good (4.50)
2.99	Fail (2)

# Solution: Format Grade

```
function formatGrade(grade) {  
  if (grade < 3.00) {  
    console.log('Fail (2)');  
  } else if (grade < 3.5) {  
    console.log(`Poor (${grade})`);  
  }  
  // TODO: Add other conditions  
}
```

# Problem : Math Power

- Create a function that **calculates** the result of a number, raised to the given power
  - **Print** the result to the console

Input	Output	Details
2, 8	256	$2^8 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 256$
3, 4	81	$3^4 = 3 * 3 * 3 * 3 = 81$



# Solution: Math Power

```
function pow(num, power){  
  let result = 1;  
  // loop exponent times  
  for(let i = 0; i < power; i++){  
    //multiply the base value  
    result *= num;  
  }  
  console.log(result);  
}
```



**Returning Values**

# The Return Statement

- The **return** keyword immediately **stops the function's execution**
- **Returns** the specified value to the caller



```
function readFullName(firstName, lastName) {  
    return firstName + " " + lastName;  
}
```

```
const fullName = readFullName("John", "Smith");  
console.log(fullName) //John Smith
```

# Using the Return Values

- Return value can be:
  - Assigned** to a variable

```
let max = getMax(5, 10);
```

- Used** in expression

```
let total = getPrice() * quantity * 1.20;
```

- Passed** to another function

```
multiply(getMax(5,10), 20)
```



# Returning Values: Examples

- Check if **array index** is valid:

```
function isValid(index, arr) {  
  if (Number.isInteger(index) && index >= 0 && index < arr.length) {  
    return true;  
  } else {  
    return false;  
  }  
}
```

- Does the student pass the exam:

```
function pass(grade) {  
  return grade >= 3;  
}
```

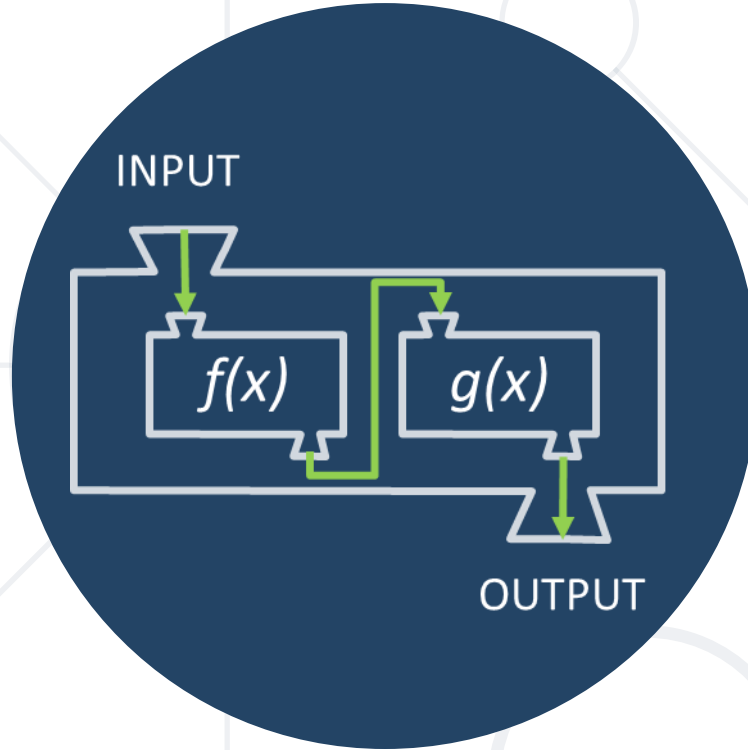
# Problem : Repeat String

- Create a function that takes a **string** and a **number n** and returns the string repeated **n times**
  - **Return** the result as a string

Input	Output
"abc", 3	abccabccabc
"String", 2	StringString

# Solution: Repeat String

```
function repeat(str, n) {  
  let result = '';  
  for (let i = 0; i < n; i++) {  
    result += str;  
  }  
  
  return result;  
}
```



# Nested Functions



# Nested Functions: Example

- Functions can be **nested**, i.e. hold other functions

```
function swapElements(arr) {  
  for (let i = 0; i < arr.length/2; i++) {  
    swap(arr, i, arr.length - 1 - i);  
  }  
  console.log(arr.join(' '));  
  function swap(elements, i, j) {  
    let temp = elements[i];  
    elements[i] = elements[j];  
    elements[j] = temp;  
  }  
}
```

Nested function

# Problem: Print Certificate

- Write a function that receives a **grade** and an **array**, containing two strings and **prints** a formatted certificate
  - If the student failed, **print** "Student does not qualify"

```
printCertificate(5.25, ['Peter', 'Carter']);  
// ~~- {@} ~~-  
// ~- Certificate ~-  
// ~~- ~---~ ~~-  
// Peter Carter  
// Very good (5.25)
```

# Solution: Print Certificate

- Use the functions we declared in **earlier examples**:

```
function printCertificate(grade, nameArr) {  
  if (pass(grade)) {  
    printHeader();  
    printName(nameArr);  
    formatGrade(grade);  
  } else {  
    let msg = `${nameArr[0]} ${nameArr[1]} does not qualify`;  
    console.log(msg);  
  }  
}
```




*f(x)*

# Functional Programming in JS

First Class

# 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

- 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

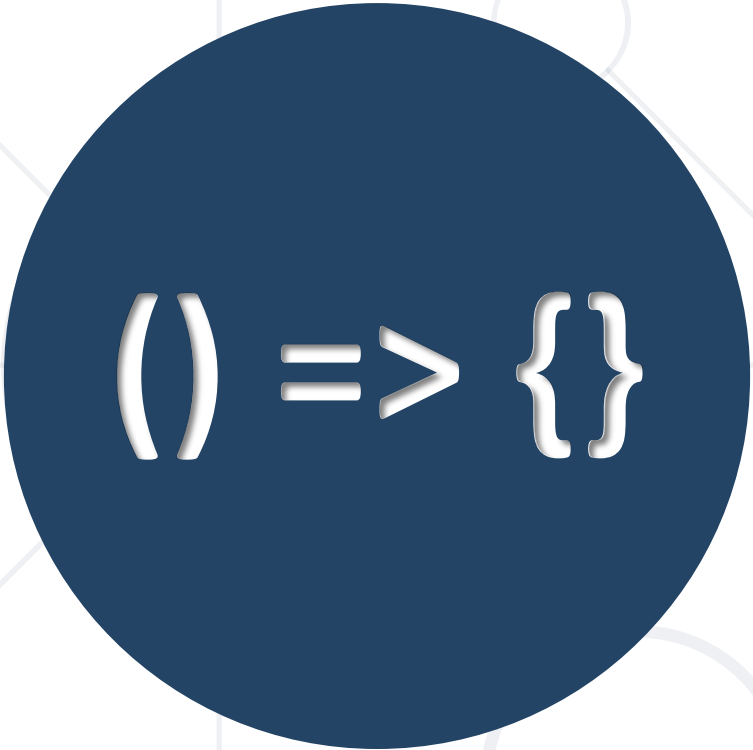
- Can be assigned as a **value** to a **variable**



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

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





`() => {}`

**Arrow Functions**

# Arrow Functions

- Special **shorthand syntax** for declaration
- They operate in the **context** of their **enclosing scope**
- Useful in **functional programming**

```
let increment = x => x + 1;  
console.log(increment(5)); // 6
```

```
let increment = function(x) {  
  return x + 1;  
}
```

```
let sum = (a, b) => a + b;  
console.log(sum(5, 6)); // 11
```

"=>" (**arrow**)

This is the same as  
the function **above**





# **Naming and Best Practices**

# Naming Functions

- Use **meaningful** names
- Should be in **camelCase**
- Names should answer the question:

- **What does this function do?**

`findStudent, loadReport, add`

`Method1, DoSomething, handleStuff, DirtyHack`

Self explaining

Puzzling

- If you cannot find a good name for a function, think about whether it has a **clear intent**



- Function parameter names:
  - Preferred form: [**Noun**] or [**Adjective**] + [**Noun**]
  - Should be in **camelCase**
  - Should be **meaningful**

`firstName, report, speedKmH,  
usersList, fontSizeInPixels, font`

- Unit of measure should be obvious

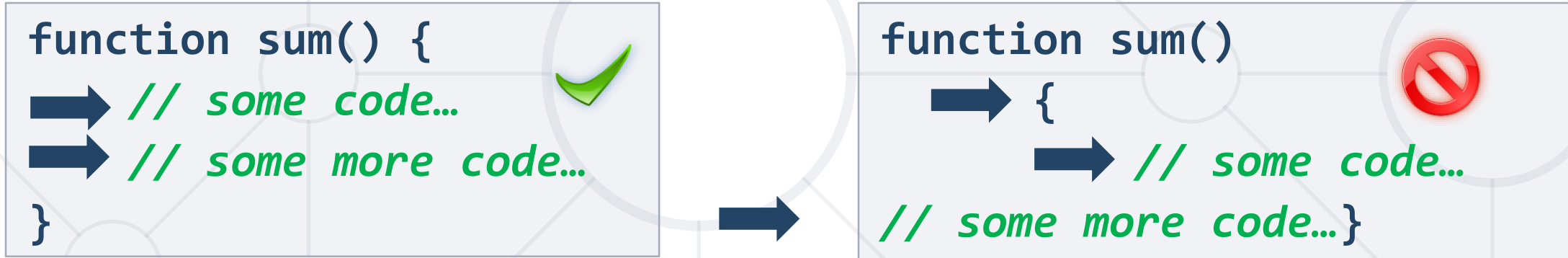
`p, p1, p2, populate, LastName, last_name, convertImage`

- Each **function** should perform a **single**, well-defined task
  - A name should **describe that task** in a clear and non-ambiguous way
- **Avoid** functions **longer than one screen**
  - **Split them** into several shorter functions

```
function printReceipt(){  
    printHeader();  
    printBody();  
    printFooter();  
}
```

**Self documenting  
and easy to test**

- Make sure to use correct **indentation**



- Leave a **blank line** between **functions** and after **blocks**
- Always use **curly brackets** for **conditional** and **loop bodies**
- Avoid long lines** and **complex expressions**

# Problem: Simple Calculator

- Write a function that **receives three parameters** and calculates the result, depending on a given operator
- The operator can be '**multiply**', '**divide**', '**add**', '**subtract**'
- The input comes as three parameters - two **numbers** and an operator as a **string**

Input	Output
5, 10, 'multiply'	50

- **Bonus task:** use **arrow functions** for the solution



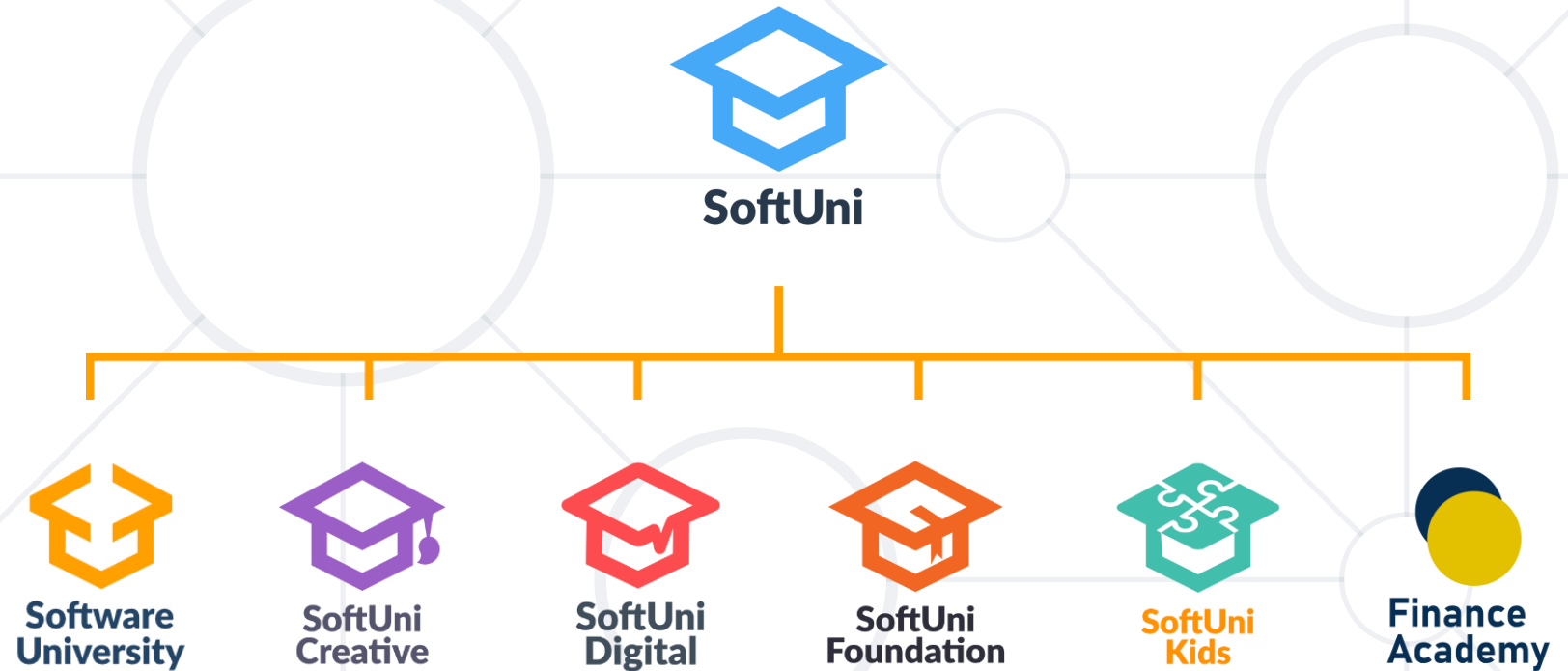
# Solution: Simple Calculator

```
function solve(a, b, operator) {  
  switch (operator) {  
    case 'multiply':  
      multiply(a, b);  
      break;  
      //TODO: other cases  
  }  
  function multiply(a, b) { // ...body }  
  //TODO: other operations  
}
```

- **Functions:**
  - Break large programs into simple functions that solve small sub-problems
  - Consist of **declaration** and **body**
  - Are invoked by their **name**
  - Can accept **parameters**



# Questions?



# SoftUni Diamond Partners

**SUPER  
HOSTING  
.BG**



**Coca-Cola HBC  
Bulgaria**



**POKERSTARS**  
POKER | CASINO | SPORTS  
a Flutter International brand

**INDEAVR**  
Serving the high achievers



**AMBITIONED**

 **DRAFT  
KINGS**



**SOFTWARE  
GROUP**

createX



**Postbank**  
Решения за твоето утре

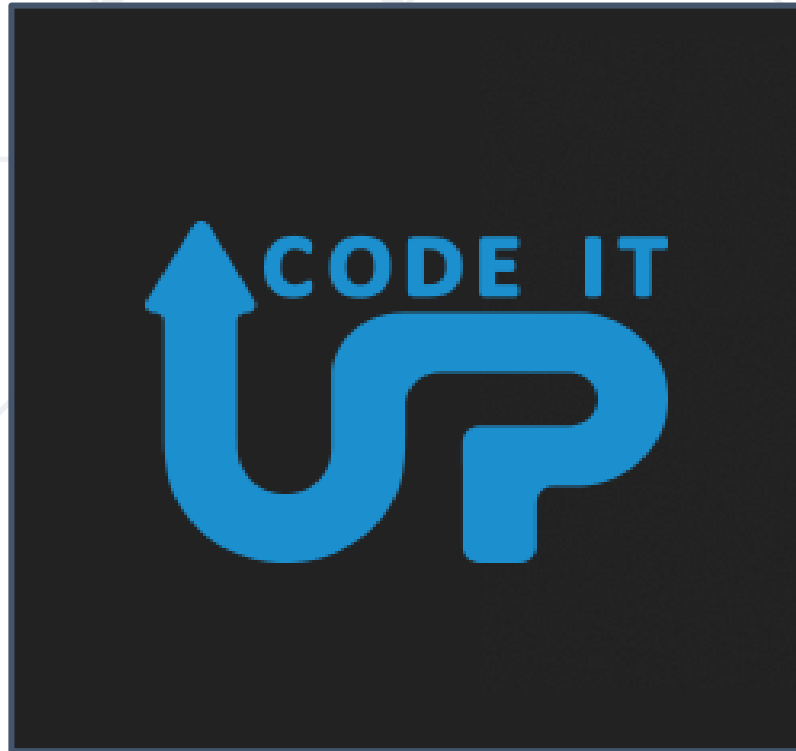


**BOSCH**

**DXC**  
TECHNOLOGY



**SmartIT**



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg)
  - Software University Foundation
    - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



- 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>

