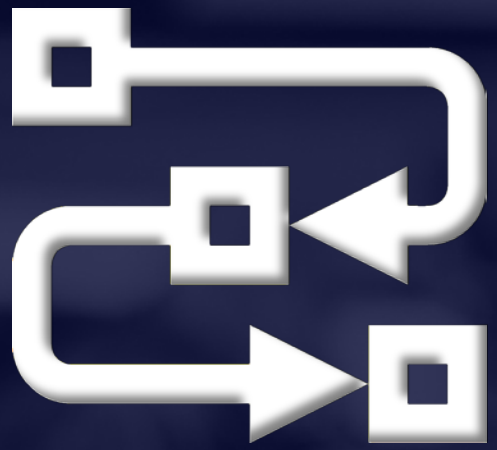
KINGSLAND UNIVERSITY

Functions

Table of Contents



Functions Overview

Declaring and Invoking Functions

Functions in JS

- A **function** is a **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 camel-case

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**

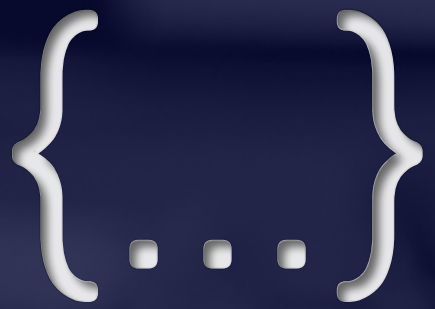
- Using existing functions several times

Function Without Parameters

- **Executes** the code between the brackets

```
function multiplyNumbers() {
  let result = 5 * 5;
  console.log(result);
}
multiplyNumbers();   // Expected Output: 25
```

Prints result on the console



Declaring and Invoking Functions

Declaring Function

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

Name

Parameters

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

Body

Declaring Function

- Functions can be declared in two ways:
  - **Function declaration**

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

  - **Function expression**

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

Invoking a Function

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

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

Function Declaration

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

```
function main(){
  printHeader();
}
```

Function Invocation

Invoking a Function (2)

- A **function** can be invoked from:
  - Other functions

```
function printHeader() {
  printHeaderTop();
  printHeaderBottom();
}
```

Function invoking functions

  - Itself (**recursion**)

```
function crash() {
  crash();
}
```

Function invoking itself

Problem : Grades

- Write a function that **receives a grade** a grade between 2.00 and 6.00 and **prints** the **corresponding grade** in **words**
  - Between **2.00** and **2.99** - '**`Fail`**'
  - Between **3.00** and **3.49** - '**`Poor`**'
  - Between **3.50** and **4.49** - '**`Good`**'
  - Between **4.50** and **5.49** – '**`Very good`**'
  - Between **5.50** and **6.00** - '**`Excellent`**'

Solution: Grades

```
function solve(grade) {
    if (grade >= 2.00 && grade <= 2.99) {
      return 'Fail';
    } else if (grade >= 3.00 && grade <= 3.49) {
       return 'Poor';
    }
    // TODO: Add other conditions
}
```

Problem : Math Power

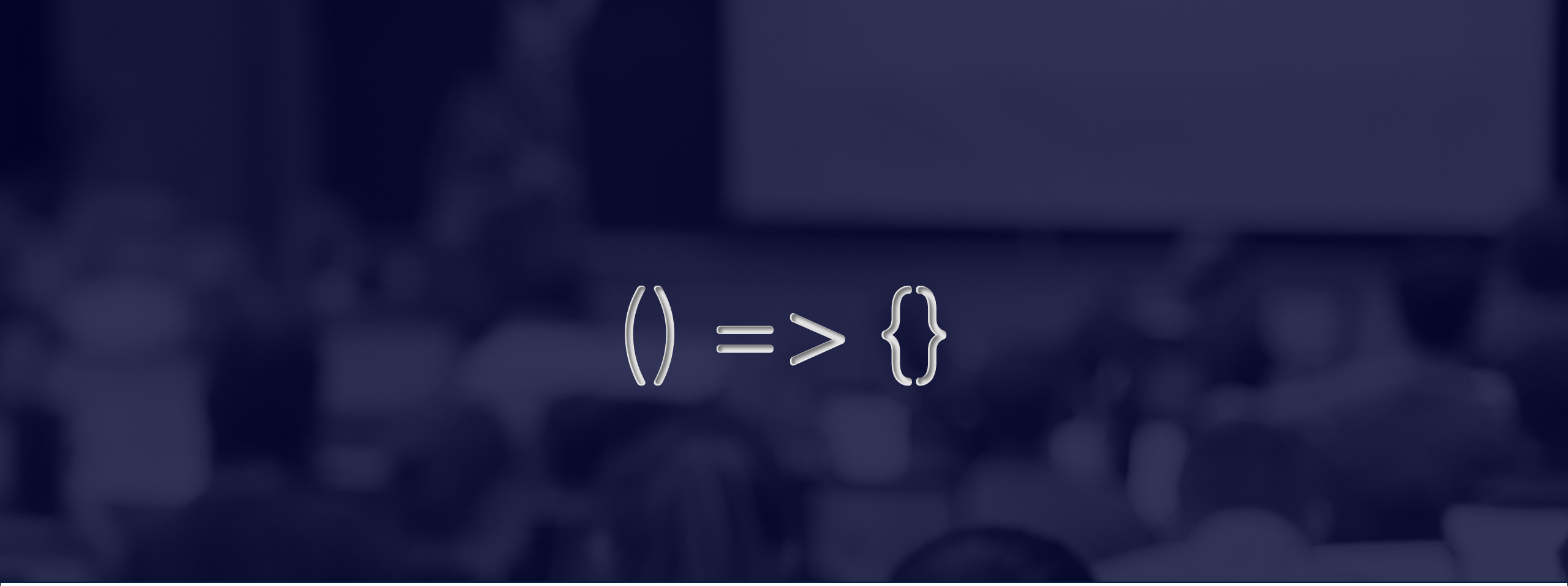
- Create a function that **calculates** the value of a number
  - The number should be **raised to a given power**
  - **Return** its value

Input	Output
2, 8	256
3, 4	81

Solution: Math Power

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

    return pow;
}
```



Arrow Functions

Arrow Functions

- These are functions with their own special syntax
- They accept a fixed number of arguments
- They operate in the **context** of their **enclosing scope**

```
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

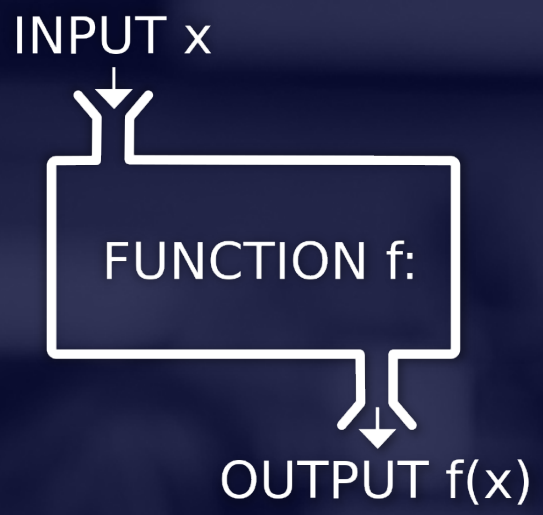
Problem: Simple Calculator

- Write a function that **receives three parameters** and write an **`arrow`** function, that calculates a result depending on 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'`	25

Solution: Simple Calculator

```
function solve(a, b, operator) {
  switch (operator) {
    case 'multiply':
      let multiply = (a, b) => a * b;
      console.log(multiply(a, b));
      break;
    case 'divide':
        //TODO: Divide the two numbers
    case 'add':
        //TODO: Add the two numbers
    case 'subtract':
        //TODO: Subtract the two numbers
    }
}
```



Nested Functions

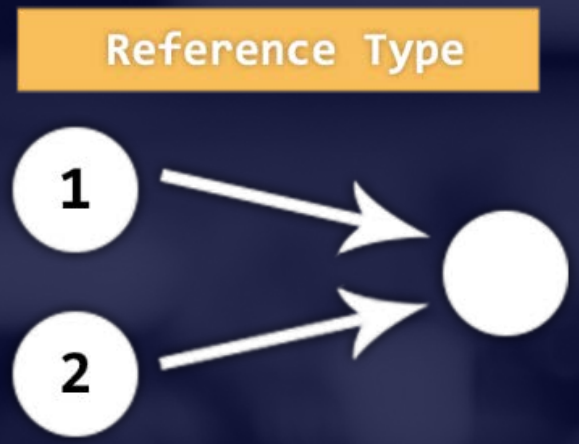
Nested Functions: Example

- Functions in can be **nested**, i.e. hold other functions
- Inner functions **have access to variables from their parent**

Main `function`

```
function drawDiamond(size) {
  drawTop(size / 2)
  drawBottom(size / 2)
}
```

Nesting the functions



Value vs. Reference Types

Memory Stack and Heap

Reference vs. Value Types

- JavaScript has 5 data types that are copied by **value**:
  - **Boolean**, **String**, **Number**, **null**, **undefined**
  - These are **primitive types**
- JavaScript has 3 data types that are copied by having their **reference** copied:
  - **Array**, **Objects** and **Functions**
  - These are all technically Objects, so we'll refer to them collectively as Objects

Example: Reference vs. Value Types

pass by reference	*pass by value*
`cup =` 	`cup =` 
`fillCup( )`	`fillCup( )`

Value Types

- If a primitive type is assigned to a variable, we can think of that variable as **containing** the primitive value

```
let a = 10;                    let c = a;
let b = 'abc';                 let d = b;
```

- They are **copied by value**

```
console.log(a, b, c, d);
// a = 10 b = 'abc' c = 10 d = 'abc'
```

Reference Types

- Variables that are assigned a non-primitive value are given a reference to that value

```
let arr = [];
let arrCopy = arr;
```

  - That reference points to a location in memory
  - Variables don't actually contain the value but lead to the location

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, sine
```

Self explaining

Puzzling

```
Method1, DoSomething, handleStuff, DirtyHack
```

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

Naming Function Parameters

Function parameters 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
```

Functions – Best Practices

- 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** to several shorter functions

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

Self documenting and easy to test

Code Structure and Code Formatting

- Make sure to use correct **indentation**
- Leave a **blank line** between functions, after **`loops`** and after
  `if` statements
- Always use **curly brackets** for loops and if statements bodies
- Avoid long lines and complex expressions

```
function main() {
    // some good…
    // example code…
}
```

```
function Main()
 {
            // some bad
// example code…
}
```

Summary

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?

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © Kingsland University – https://kingslanduniversity.com

KINGSLAND UNIVERSITY

THANK YOU