Javascript Reintroduction (2)

Advanced Web Design

Variable declarations: var, let and const

	Scope	Reassignable	Mutable
const	block	No	Yes
let	block	Yes	Yes
var	function	Yes	Yes

Functions

- **Function statements (1):** Declared with the function keyword and can be called before it's defined.

- **Function expressions (2):** A function stored in a variable, must be defined before use.

Functions

(1)

```
sayWelcome(); // works – hoisted

function sayWelcome() {
    console.log("Welcome to AWD!");
}
```

(2)

```
// greet(); Error - not hoisted
let greet = function sayWelcome() {
    console.log("Welcome to AWD!");
}
greet(); // works
```

References and values in JS

- Every primitive type is passed by value (copied).
- Every non-primitive type is passed by reference.

References and values in JS

- Pass by value:

```
let num1 = 10;

function changeValue(value) {
    value = 20;        // change the local copy
    console.log("Inside function:", value); // value = 20
}

changeValue(num1);
console.log("Outside function:", num1); // value = 10
```

References and values in JS

- Pass by reference:

```
let student = {
    name: "John",
    idNum: 203043
};

function changeName(student) {
    student.name = "Jane"
    console.log("Inside function:", student)
}

changeName(student);
console.log("Outside function:", student)
```

Functions as arguments

- In JavaScript, functions are “first-class citizens”
- They can be assigned to variables or passed to other functions as arguments
- Passing functions as arguments is extremely common in jQuery.

Functions as arguments

- Example

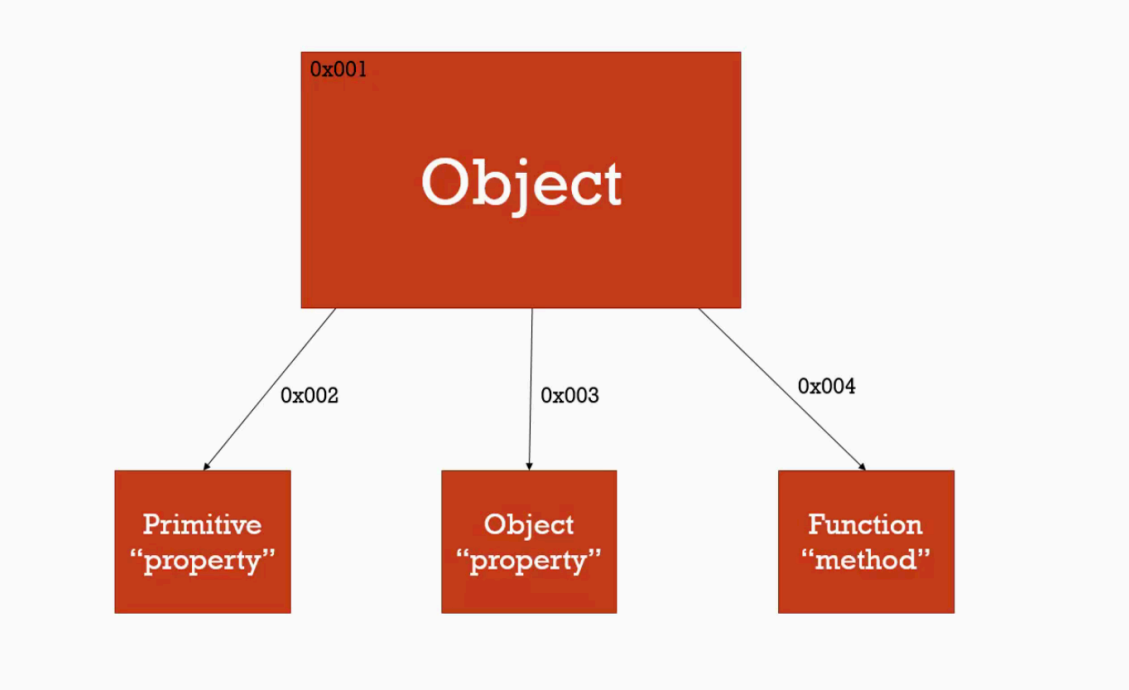
```
function sayHello(){
    console.log("Hello students, welcome!")
}

function greetStudents(someFunction) {
    console.log("Preparing to greet the
students...")
    someFunction()
    console.log("How are you?")
}

greetStudents(sayHello)
```

Objects in depth

- Collections of key-value pairs
- The value portion can be any type of value:
  - number
  - string
  - array
  - function
  - another object

Objects in depth

- The values do not live inside the object, but the object keeps references to them.



Closures

- A **closure** happens when a function **remembers** variables from the place it was **created**.

- Inner functions can still **access outer variables**, even after the outer function is finished.

Closures

- Example

```
function outer(){
    let faculty = "FINKI"
    function inner(){
        console.log(`Hello students! Welcome to ${faculty}!`);
    }
    return inner;
}

let fn = outer();
fn();
```

Closures

- Example

```
function makeButtonFactory(color) {
    let count = 0; // private variable

    return function(label) {
        count++;
        const btn = document.createElement("button");
        btn.innerHTML = `${label} (${count})`;
        btn.style.backgroundColor = color;
        document.body.appendChild(btn);
    };
}

// create two independent factories
const redButtons = makeButtonFactory("red");
const blueButtons = makeButtonFactory("blue");

// generate buttons
redButtons("Delete");
redButtons("Remove");
blueButtons("Save");
blueButtons("Upload");
```

Why do we use closures?

- To keep data private (encapsulation).
- To avoid global variables and accidental changes.
- To create function factories or custom behaviors.

Example 1

- Write a function dropUntil that takes 2 arguments:
  - array: an array of elements
  - predicate: a function to be executed on each element

  The function should drop elements from an array until the predicate is true.

input:

```
let users = [
    { "user": "barney", "active": false },
    { "user": "fred", "active": false },
    { "user": "pebbles", "active": true }
]
```

output:

```
[ { user: 'pebbles', active: true } ]
```

Solution

```
let users = [
    { "user": "barney", "active": false },
    { "user": "fred", "active": false },
    { "user": "pebbles", "active": true }
]


function dropUntil(array, isUserActive){
    return array.filter(user => isUserActive(user));
}

function isUserActive(user){
    return user.active;
}

console.log(dropUntil(users, isUserActive))
```

Useful array functions

Method & Description
concat()Returns a new array comprised of this array joined with other array(s) and/or value(s).
every()Returns true if every element in this array satisfies the provided testing function.
filter()Creates a new array with all of the elements of this array for which the provided filtering function returns true.
forEach()Calls a function for each element in the array.
indexOf()Returns the first (least) index of an element within the array equal to the specified value, or -1 if none is found.
join()Joins all elements of an array into a string.
lastIndexOf()Returns the last (greatest) index of an element within the array equal to the specified value, or -1 if none is found.
map()Creates a new array with the results of calling a provided function on every element in this array.
pop()Removes the last element from an array and returns that element.
push()Adds one or more elements to the end of an array and returns the new length of the array.
reduce()Apply a function simultaneously against two values of the array (from left-to-right) as to reduce it to a single value.
reduceRight()Apply a function simultaneously against two values of the array (from right-to-left) as to reduce it to a single value.
reverse()Reverses the order of the elements of an array -- the first becomes the last, and the last becomes the first.
shift()Removes the first element from an array and returns that element.
slice()Extracts a section of an array and returns a new array.
some()Returns true if at least one element in this array satisfies the provided testing function.
toSource()Represents the source code of an object
sort()Sorts the elements of an array
splice()Adds and/or removes elements from an array.
toString()Returns a string representing the array and its elements.
unshift()Adds one or more elements to the front of an array and returns the new length of the array.

Testing type with *typeof*

- **typeof** checks what type of value a variable holds.
- Always returns a **string** describing the type.
- Works on numbers, strings, booleans, objects, and more.
- Great for quick checks in conditionals or debugging.

Arrow functions

- A shorter way to write function using the => syntax
- Automatically return the expression if written in one line.
- Great for callbacks and inline logic

Comparison: Regular function vs Arrow function

(1)

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


// Arrow function
const addArrow = (a, b) => a + b;
```

(2)

```
// Regular function
function isEven(num) {
    return num % 2 === 0;
}


// Arrow function
const isEvenArrow = num => num % 2 === 0;
```

Example 2

- Create a script that filters out products that are out of stock and uses array functions to:
  - list the names of the available items
  - calculate the total quantity of all available items

```
[{ id: 1, name: "Keyboard", price: 30, qty: 2, inStock: true },
 { id: 2, name: "Mouse", price: 15, qty: 1, inStock: false },
 { id: 3, name: "Monitor", price: 120, qty: 1, inStock: true },
 { id: 4, name: "Headphones", price: 45, qty: 3, inStock: true },
 { id: 5, name: "USB Cable", price: 8, qty: 5, inStock: false }]
```

Solution

```
let products = [
    { id: 1, name: "Keyboard", price: 30, qty: 2, inStock: true },
    { id: 2, name: "Mouse", price: 15, qty: 1, inStock: false },
    { id: 3, name: "Monitor", price: 120, qty: 1, inStock: true },
    { id: 4, name: "Headphones", price: 45, qty: 3, inStock: true },
    { id: 5, name: "USB Cable", price: 8, qty: 5, inStock: false }
]

let filteredProducts = products.filter(product => product.inStock)

let filteredProductsNames = products.map(product => product.name)

let totalQuantity = 0

filteredProducts.forEach(product => totalQuantity+=product.qty)

console.log(filteredProductsNames)
console.log("Total quantity: ",totalQuantity)
```

Example 3

- For the given an array of user objects, write a function that removes duplicate users while keeping only the first occurrence of each email. After cleaning the array, extract only the users' names, sort them alphabetically, and print the result as a single comma-separated string.

```
[
    { id: 1, name: "Ana", email: "ana@example.com" },
    { id: 2, name: "Marko", email: "marko@example.com" },
    { id: 3, name: "Ana", email: "ana@example.com" },
    { id: 4, name: "Mila", email: "mila@example.com" },
    { id: 5, name: "John", email: "john@example.com" },
    { id: 6, name: "Jane", email: "jane@example.com" }
]
```

Solution

```
const users = [
    { id: 1, name: "Ana", email: "ana@example.com" },
    { id: 2, name: "Marko", email: "marko@example.com" },
    { id: 3, name: "Ana", email: "ana@example.com" },
    { id: 4, name: "Mila", email: "mila@example.com" },
    { id: 5, name: "John", email: "john@example.com" },
    { id: 6, name: "Jane", email: "jane@example.com" }
];

const emails = []
const filteredUsers = []

users.forEach(user => {
    if(!emails.includes(user.email)){
        filteredUsers.push(user);
    }
    emails.push(user.email)
})

let filteredUsersNames = filteredUsers
    .map(user => user.name)
    .sort()
    .join(",")

console.log(filteredUsersNames)
```

Example 4

- Compute total grade sum for students above 10 by composing map, filter and reduce:

```
const students = [
    { name: "Nick", grade: 10 },
    { name: "John", grade: 15 },
    { name: "Julia", grade: 19 },
    { name: "Nathalie", grade: 9 },
];
```

Solution

```
const students = [
    { name: "Nick", grade: 10 },
    { name: "John", grade: 15 },
    { name: "Julia", grade: 19 },
    { name: "Nathalie", grade: 9 },
];

let totalGrade = students
    .filter(student => student.grade > 10)
    .map(student => student.grade)
    .reduce((prev, next) => prev+next, 0)

console.log(totalGrade)
```

Destructuring objects and arrays

- *Destructuring* is a convenient way of creating new variables by extracting some values from data stored in objects or arrays.

- To name a few use cases, *destructuring* can be used to destructure function parameters or *this.props* in React projects for instance.

Destructuring objects and arrays

```
const person = {
   firstName: "Nick",
   lastName: "Anderson",
   age: 35,
   sex: "M"
}
```

- Without destructuring:

```
const first = person.firstName;
const age = person.age;
const city = person.city || "Paris";
```

- With destructuring, all in one line:

```
const { firstName: first, age, city = "Paris" } = person;
```

Spread operator "..."

- The spread operator `...` is used to expand elements of an iterable (like an array) into places where multiple elements can fit

```
const arr1 = ["a", "b", "c"];
const arr2 = [...arr1, "d", "e", "f"]; // ["a", "b", "c", "d", "e", "f"]
```

Promises

- A Promise is an object that represents the result of an asynchronous operation - a value that you'll get in the future, once the task is finished.
- Used for asynchronous operations - like API calls, file reading, timers, etc.
- You handle the result using *.then()*, *.catch(),* and *.finally()*

Promises

```
fetch("https://jsonplaceholder.typicode.com/users")
.then(response => {
    // convert the response to JSON
    return response.json();
})
.then(data => {
    // handle the data
    console.log("Fetched users:", data);
})
.catch(error => {
    // handle errors (e.g., network issue)
    console.error("Error fetching data:", error);
})
.finally(() => {
    console.log("Request completed.");
});
```

Promises

- *fetch()* starts an asynchronous operation - it returns a promise, not the data right away
- *then()* runs when the promise resolves successfully
- *catch()* runs if something goes wrong
- *finally()* runs no matter what

async / await

- ***async*** and ***await*** make working with Promises look like regular code
- ***async*** marks a function that will always return a Promise
- ***await*** pauses the function until a Promise is resolved or rejected

async / await

```
async function loadUsers() {
    try {
        // wait for the API to respond
        const response = await fetch("https://jsonplaceholder.typicode.com/users");

        // wait for the response to be converted to JSON
        const data = await response.json();

        console.log("Fetched users:", data);
    } catch (error) {
        // handle possible errors
        console.error("Error fetching data:", error);
    } finally {
        console.log("Request completed.");
    }
}

loadUsers();
```

Example 5

- Fetch data for products from https://dummyjson.com/products (GET request) and perform useful data transformations: extract all existing categories and print them alphabetically. Print out the three products with the highest average review rating and the three with the lowest. If a product has no reviews, treat its average as 0. Build a summary per category containing the number of products, the total inventory value (price * stock summed). Print all products that are low stock (stock < 10) sorted by stock ascending. Print the top 5 discounted products by discount percentage (descending) showing Title - Price - Discount%. Finally, produce a quick availability report counting products by availability status, print everything in a readable format and handle missing/empty arrays appropriately.