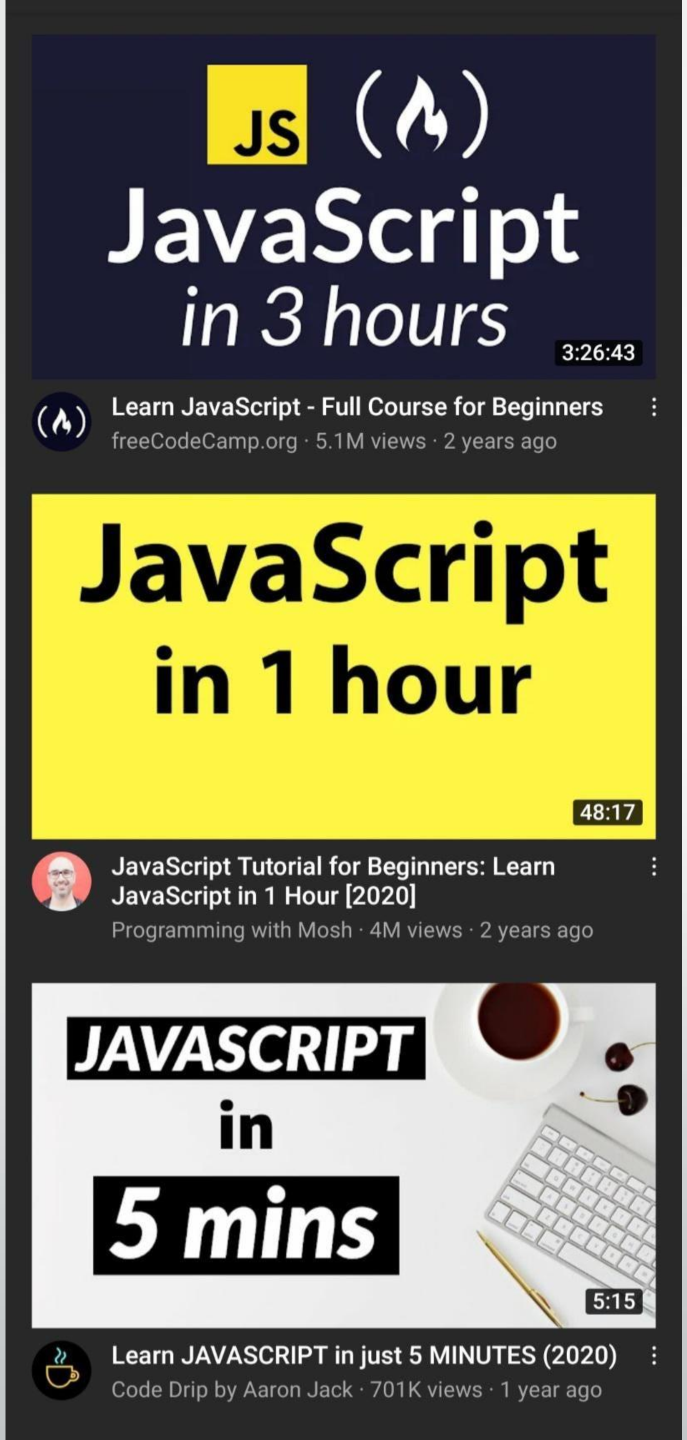
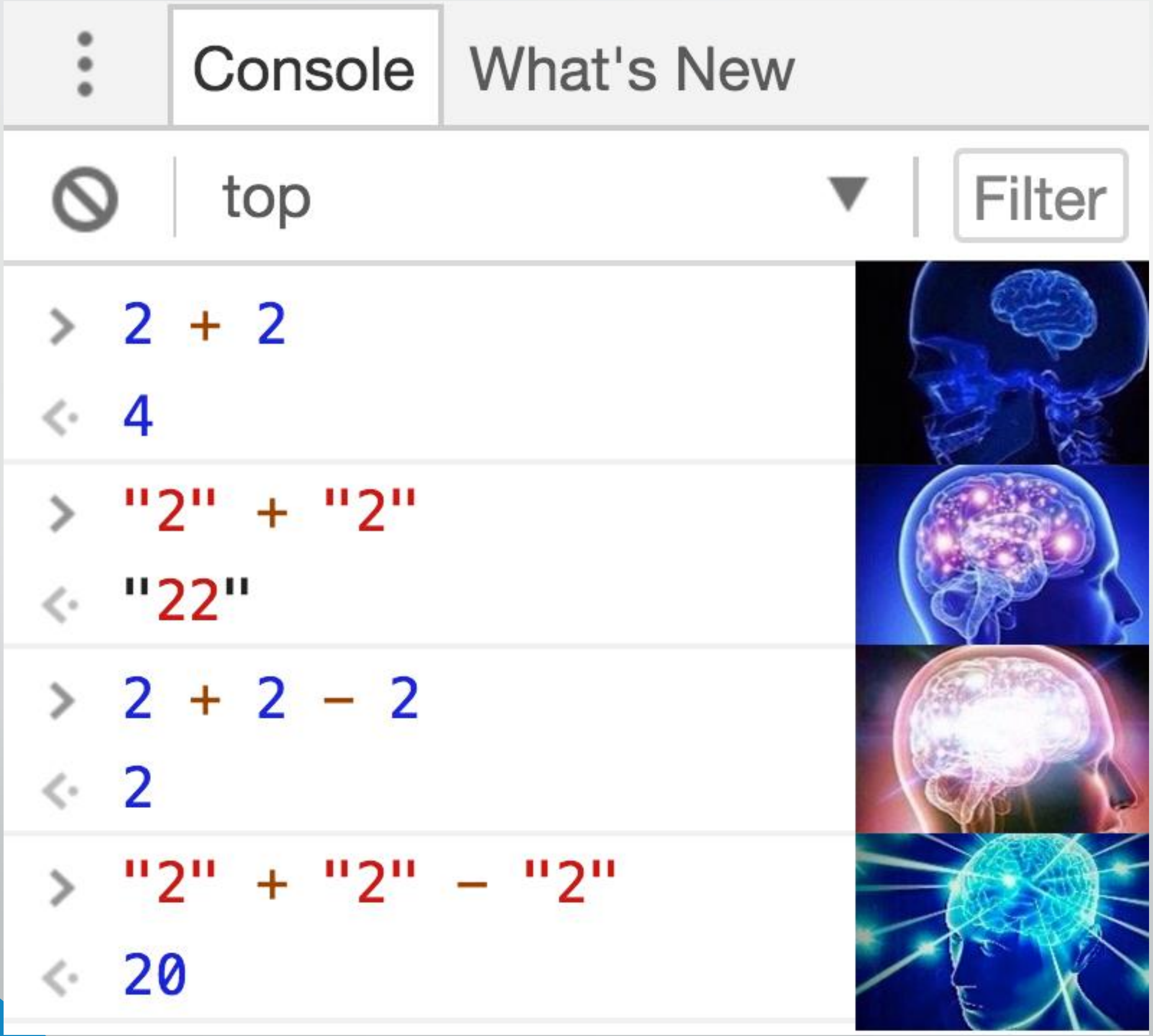


Learn JavaScript - Full Course for Beginners
freeCodeCamp.org · 5.1M views · 2 years ago

JavaScript Tutorial for Beginners: Learn JavaScript in 1 Hour [2020]
Programming with Mosh · 4M views · 2 years ago

Learn JAVASCRIPT in just 5 MINUTES (2020)
Code Drip by Aaron Jack · 701K views · 1 year ago



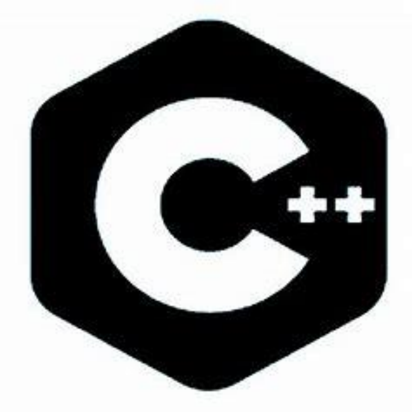
```
> 2 + 2
<· 4
> "2" + "2"
<· "22"
> 2 + 2 - 2
<· 2
> "2" + "2" - "2"
<· 20
```

Good Parts of JavaScript

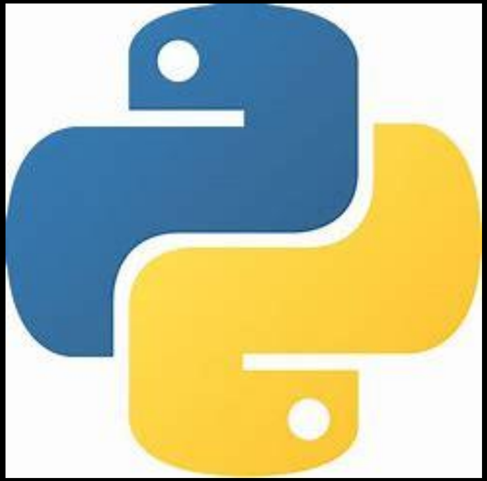
Agenda

- Versatility of JavaScript.
- Why we need to know Good Parts?
- What are Good Parts of JS and why?
- Some Best Practices

My Introduction



C++



Java

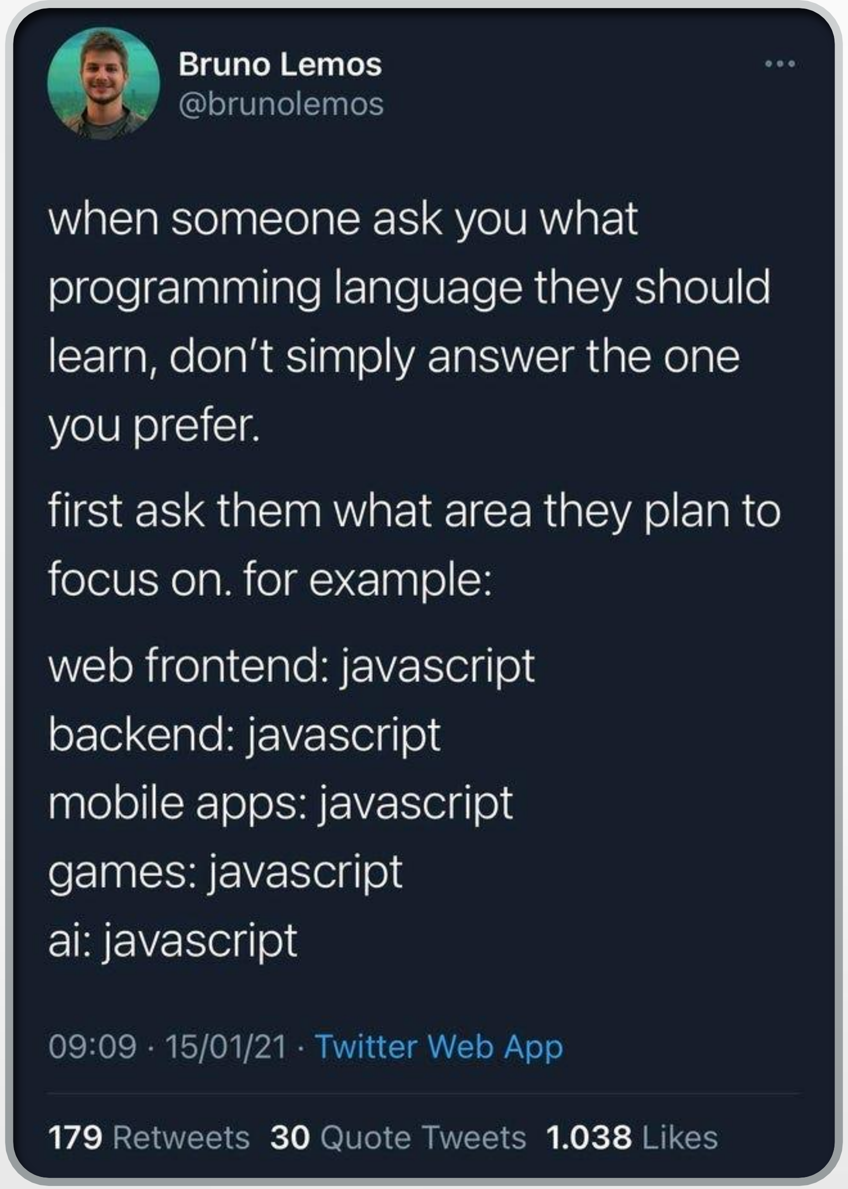
JS
5

JavaScript is the only language that I’m aware of that people feel they don’t need to learn before they start using it.

Douglas Crockford

JavaScript –
A Versatile Language

It is Everywhere



Bruno Lemos
@brunolemos

when someone ask you what programming language they should learn, don't simply answer the one you prefer.

first ask them what area they plan to focus on. for example:

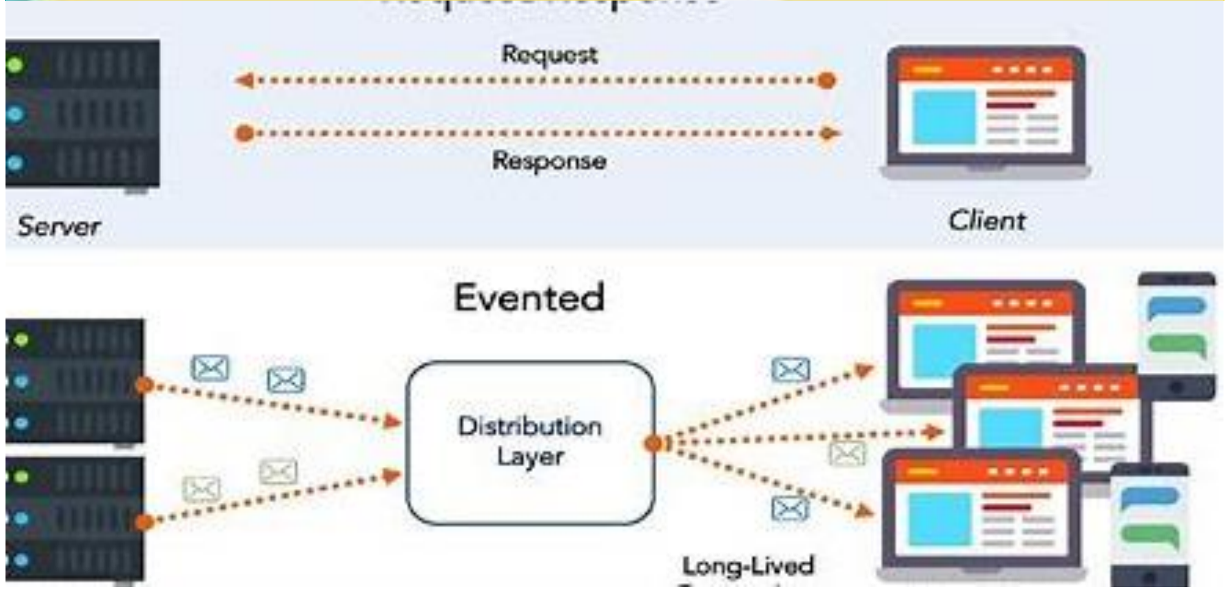
web frontend: javascript
backend: javascript
mobile apps: javascript
games: javascript
ai: javascript

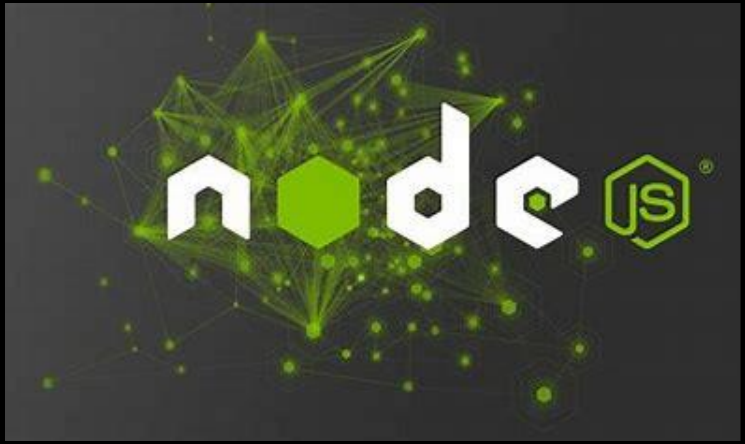
09:09 · 15/01/21 · Twitter Web App

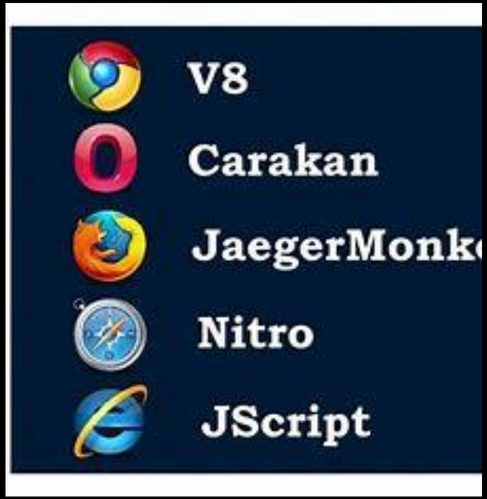
179 Retweets 30 Quote Tweets 1.038 Likes

Web
Application

MOBILE APP
DEVELOPMENT


Request
Response
Server
Client
Evented
Distribution
Layer
Long-Lived



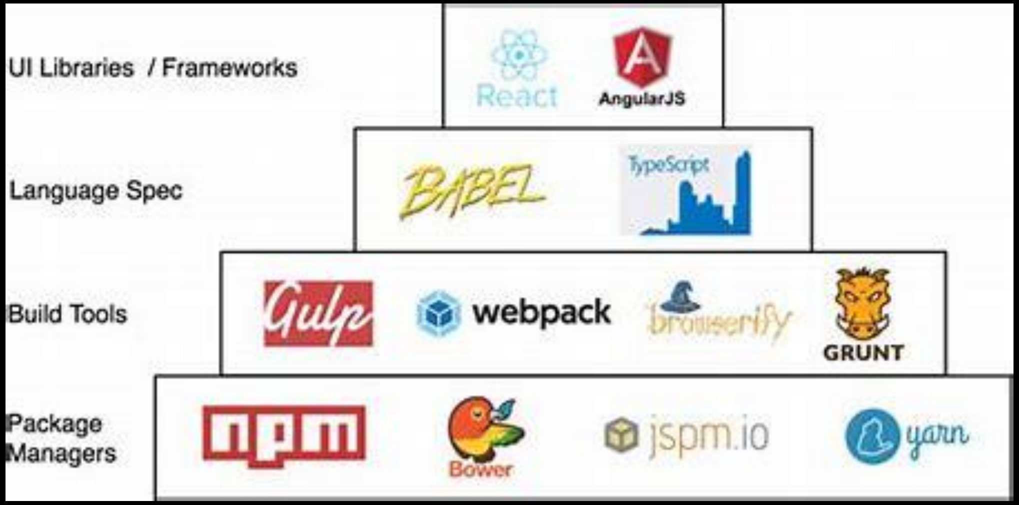

V8

Carakan

JaegerMonk

Nitro

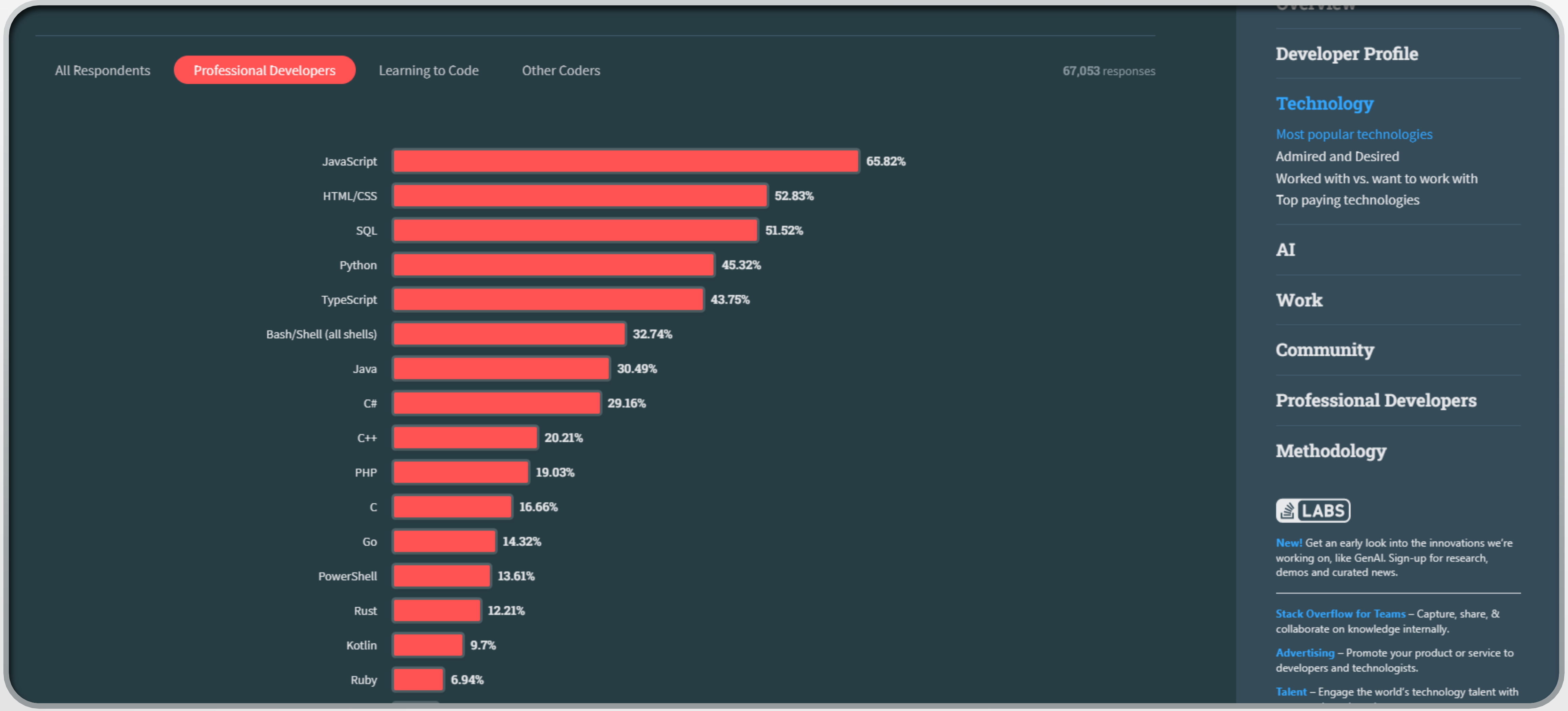
JScript


UI Libraries / Frameworks

Language Spec

Build Tools

Package Managers



All Respondents | Professional Developers | Learning to Code | Other Coders

67,053 responses

Overview

Developer Profile

Technology

Most popular technologies
Admired and Desired
Worked with vs. want to work with
Top paying technologies

AI

Work

Community

Professional Developers

Methodology

LABS

New! Get an early look into the innovations we're working on, like GenAI. Sign-up for research, demos and curated news.

Stack Overflow for Teams – Capture, share, & collaborate on knowledge internally.

Advertising – Promote your product or service to developers and technologists.

Talent – Engage the world's technology talent with

But still Criticised ?

Because …

- Too many smelly things
- Can’t remove bad parts
- Unconventional Concepts
- Difficult to master

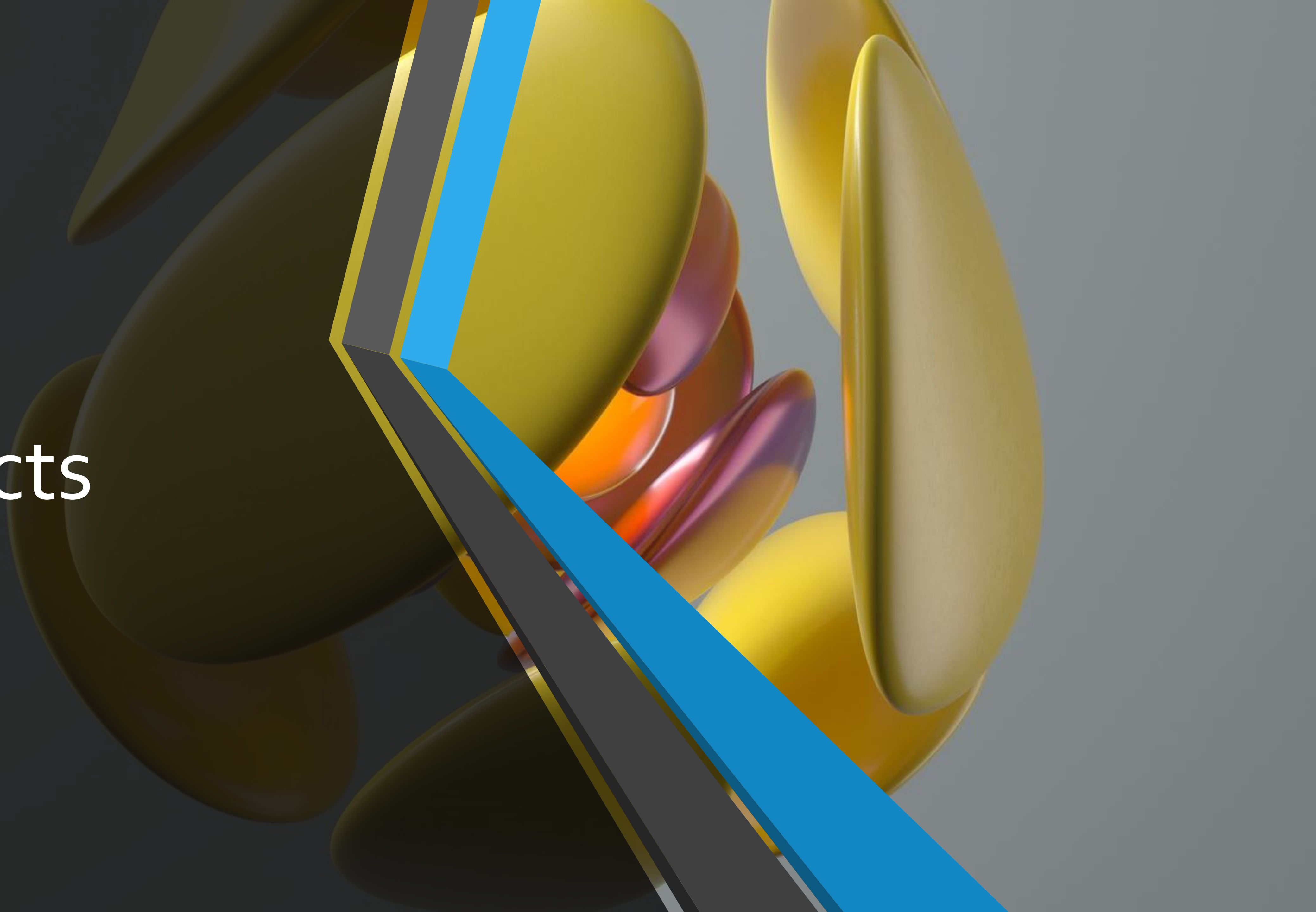
Why to know Good parts ?

The Good Parts

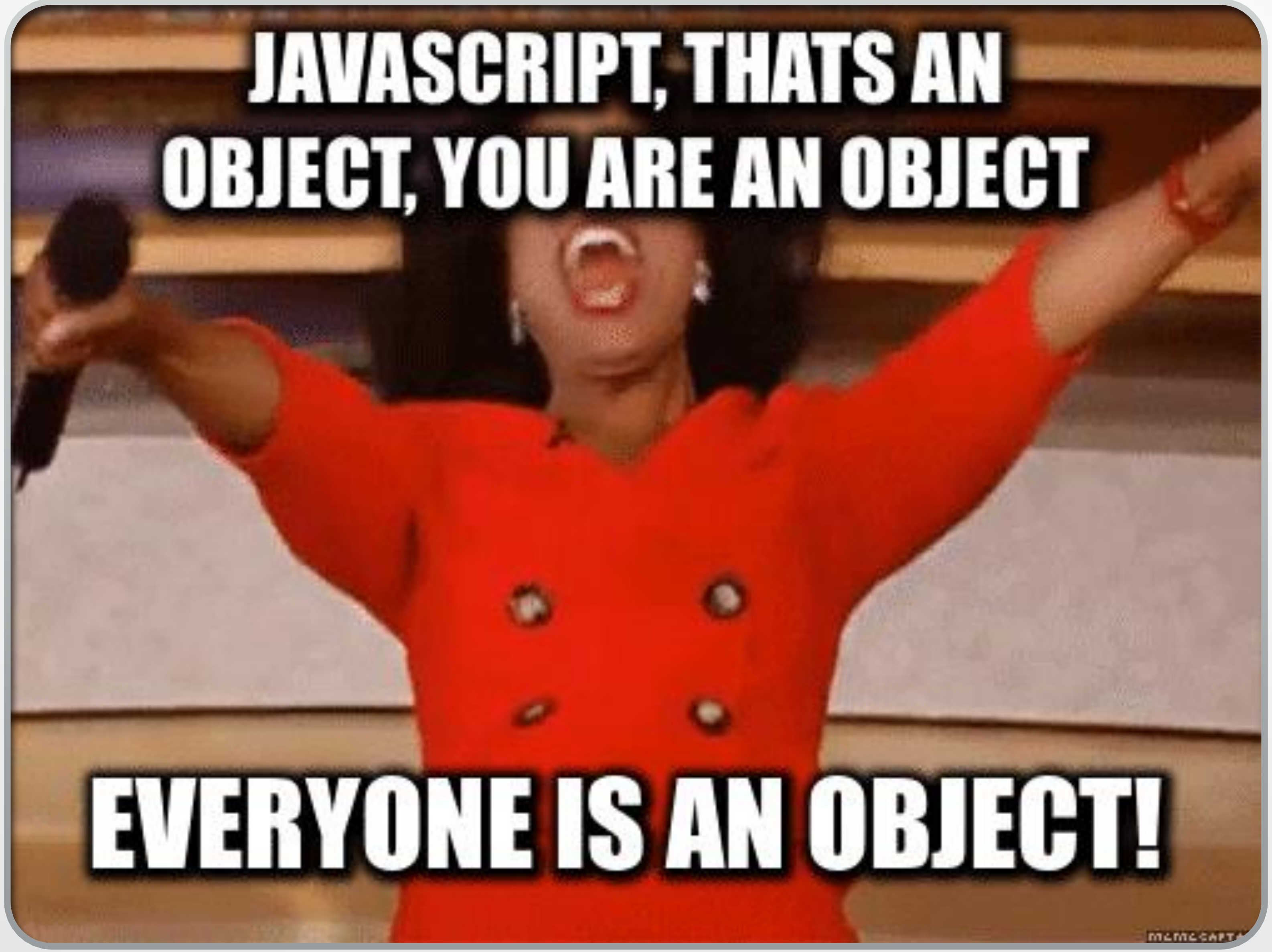
- Objects
- Functions
- Prototypal Inheritance

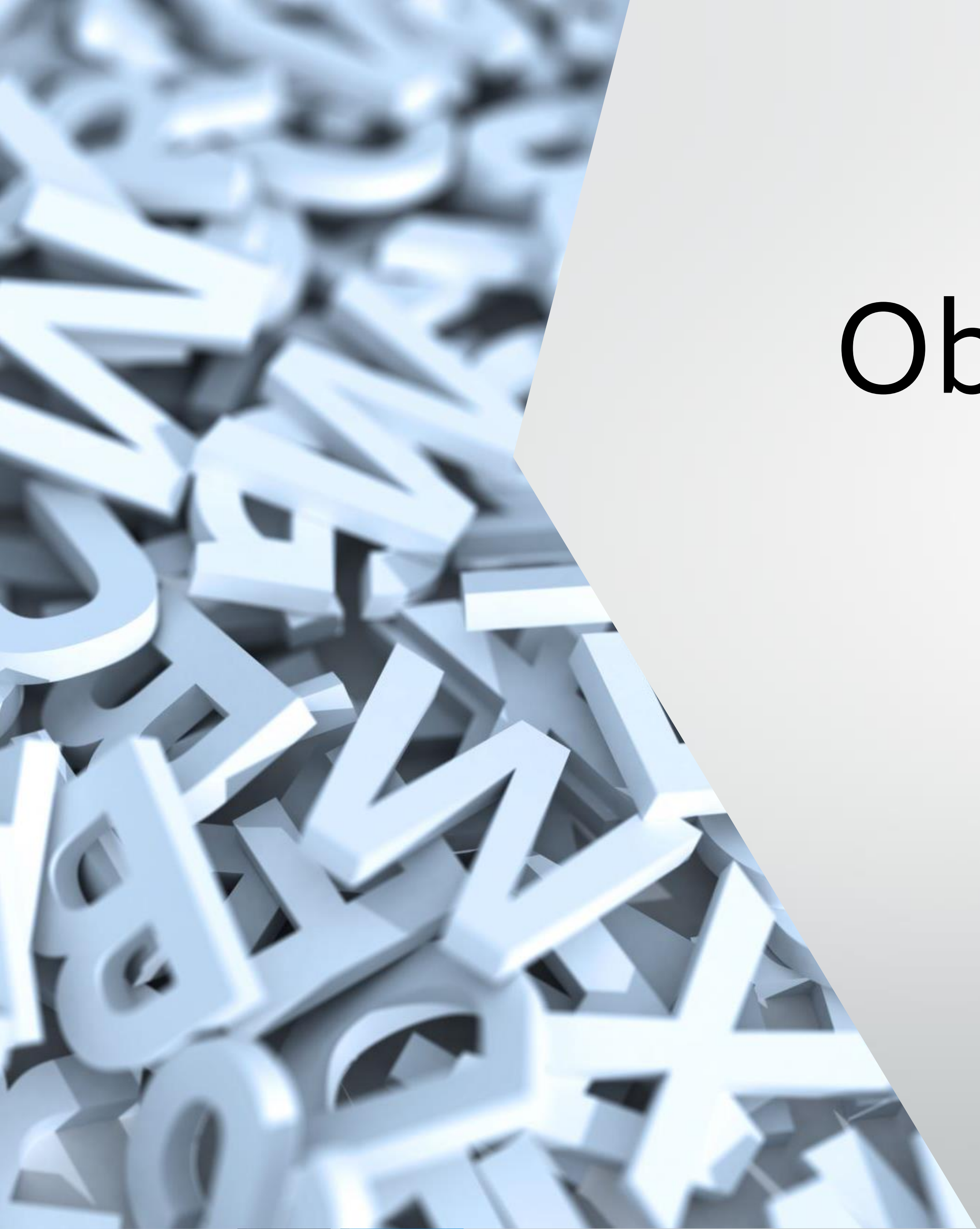
Good Parts ahead >>>

Objects



Not instance of classes


JAVASCRIPT, THATS AN OBJECT, YOU ARE AN OBJECT
EVERYONE IS AN OBJECT!



Object Literals

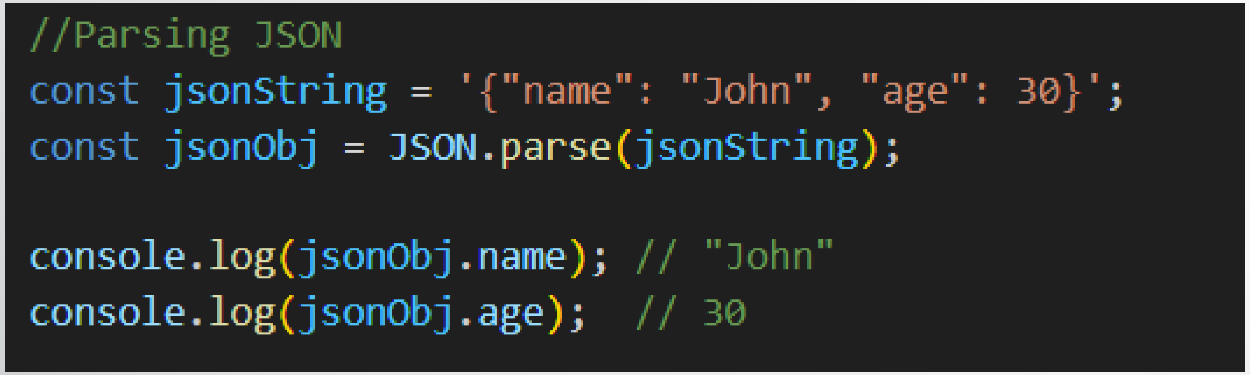
- Simple
- Expressive

JSON

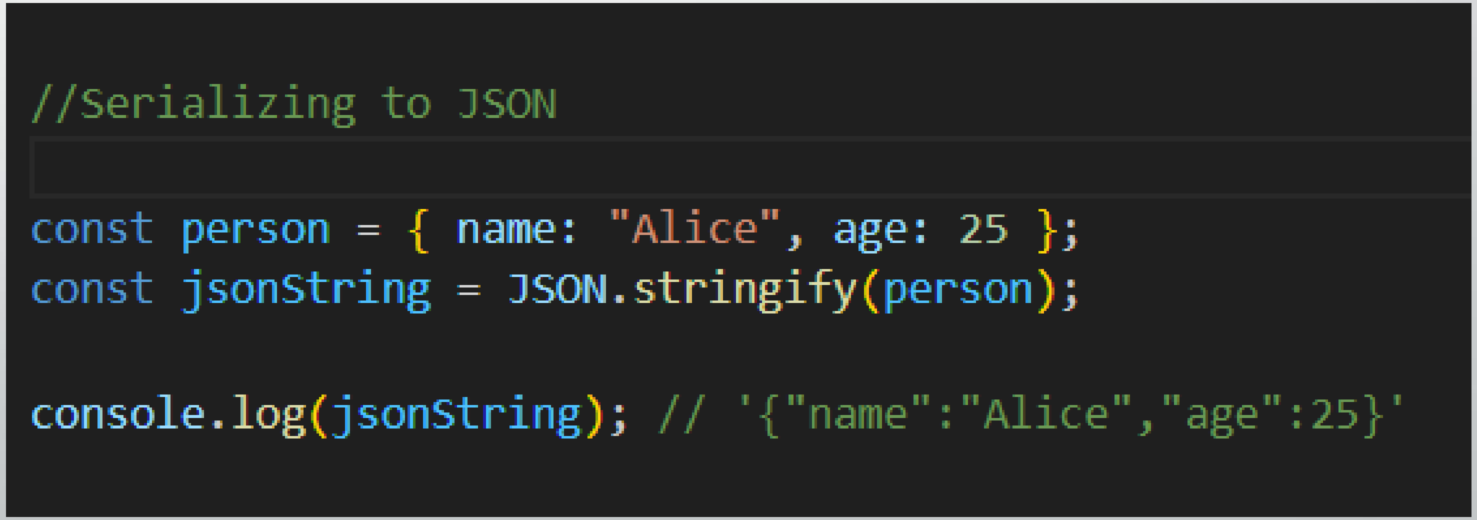
(JavaScript Object Notation)

Frontend Dev
JSON
Backend Dev
imgflip.com

Handling Data Interchange



```
//Parsing JSON
const jsonString = '{"name": "John", "age": 30}';
const jsonObj = JSON.parse(jsonString);

console.log(jsonObj.name); // "John"
console.log(jsonObj.age);  // 30
```



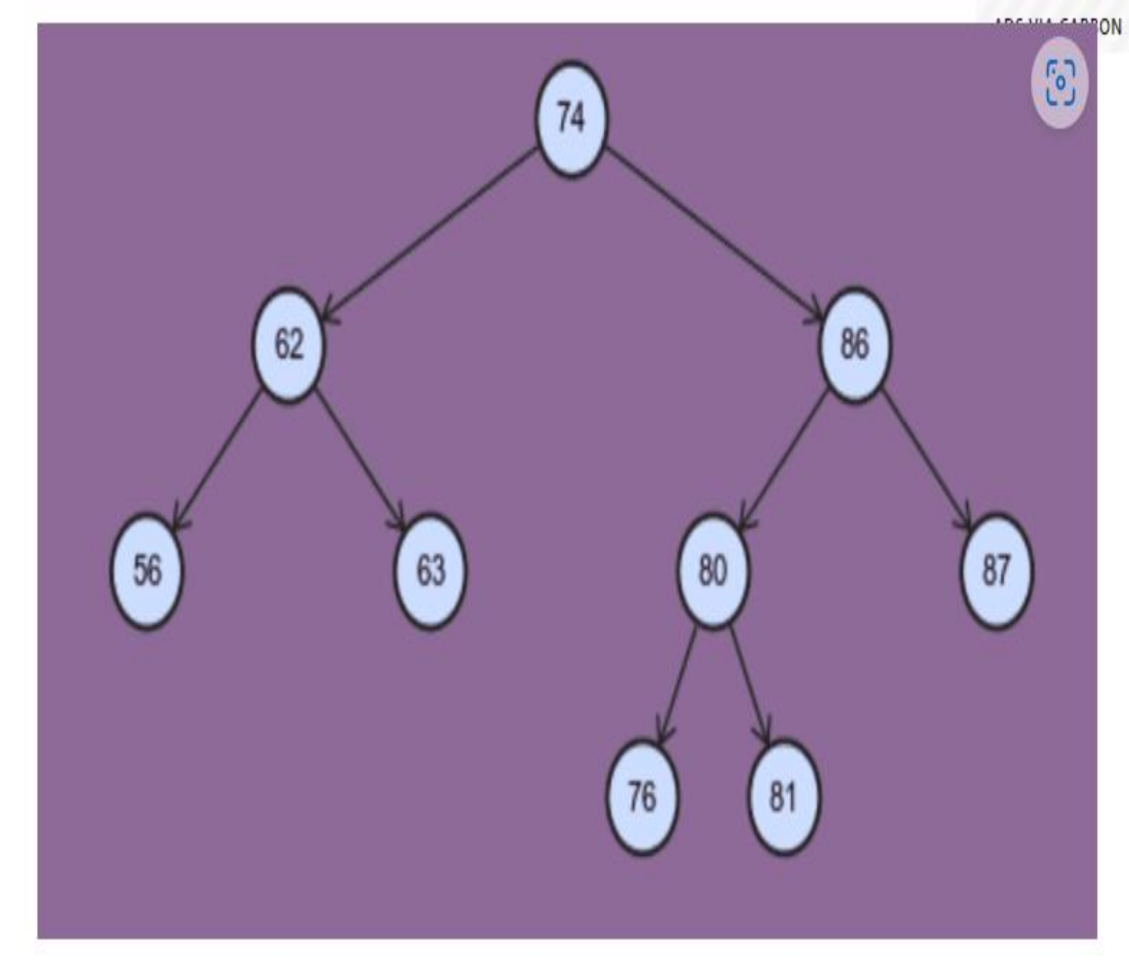
```
//Serializing to JSON

const person = { name: "Alice", age: 25 };
const jsonString = JSON.stringify(person);

console.log(jsonString); // '{"name":"Alice","age":25}'
```

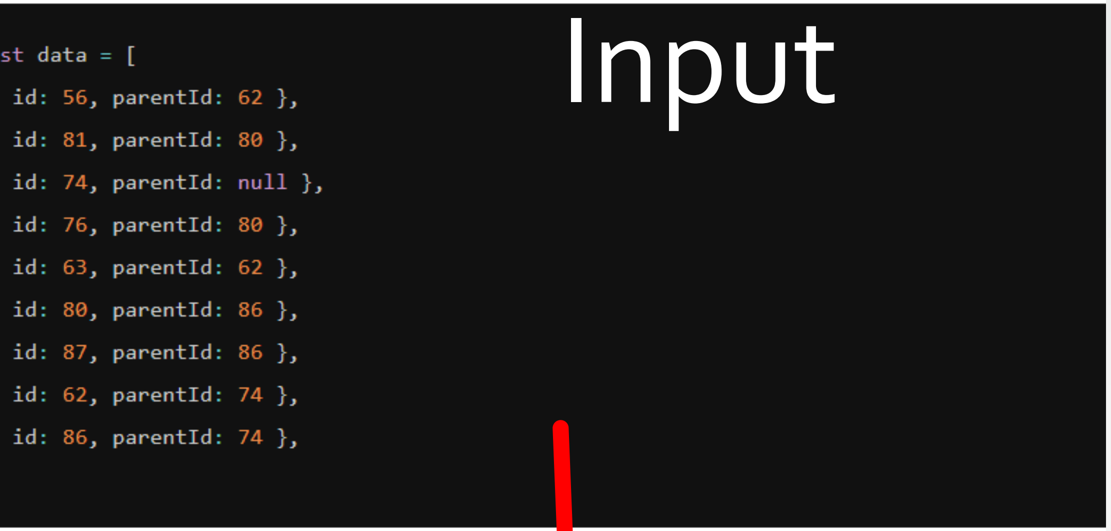
JSON handling in other Programming languages ?

- Using extra libraries function

Mutability of Objects

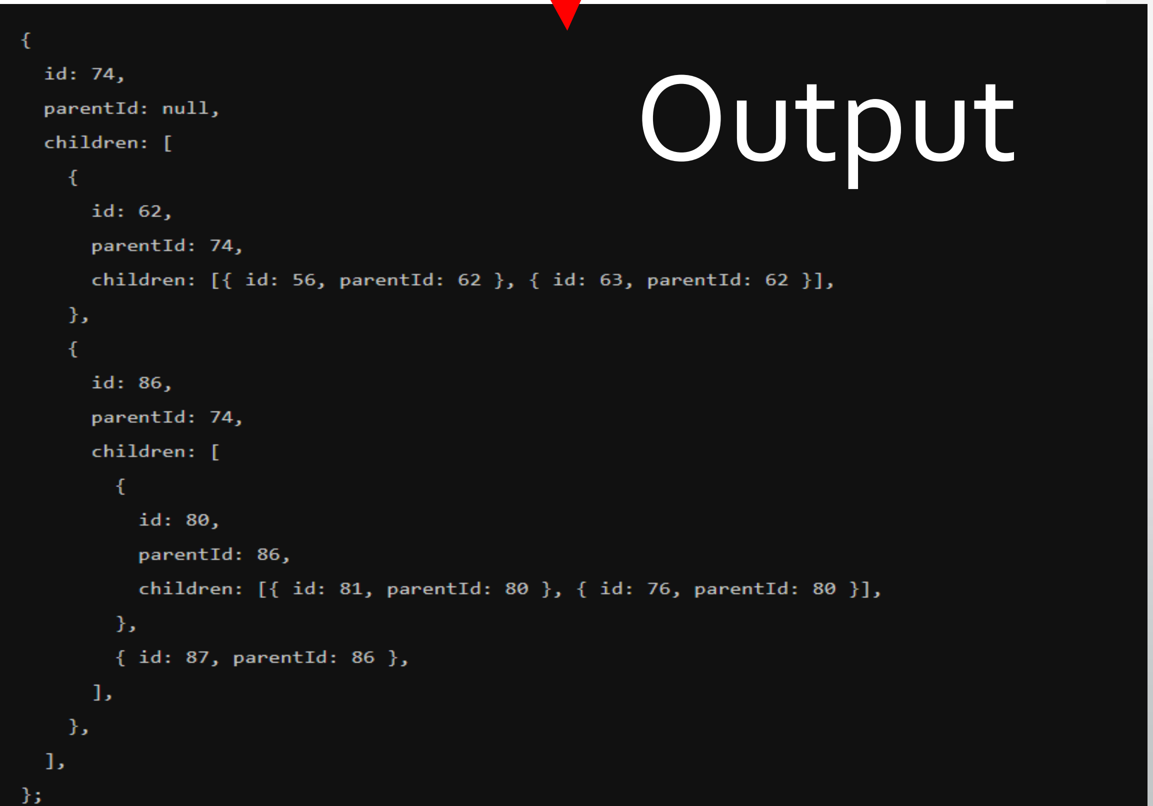
Working with Complex Data Structures





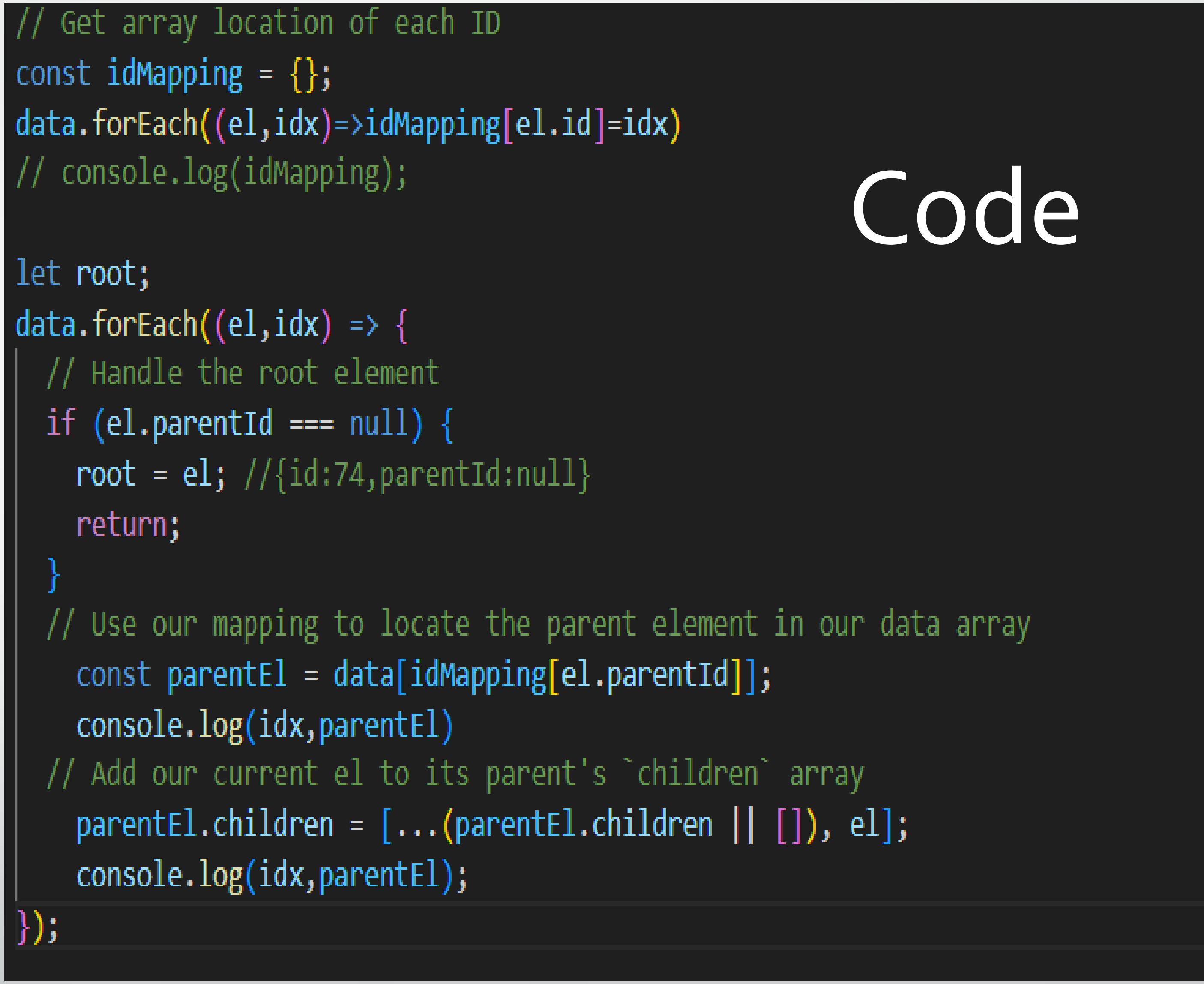
Input

```
st data = [
 id: 56, parentId: 62 },
 id: 81, parentId: 80 },
 id: 74, parentId: null },
 id: 76, parentId: 80 },
 id: 63, parentId: 62 },
 id: 80, parentId: 86 },
 id: 87, parentId: 86 },
 id: 62, parentId: 74 },
 id: 86, parentId: 74 },
```



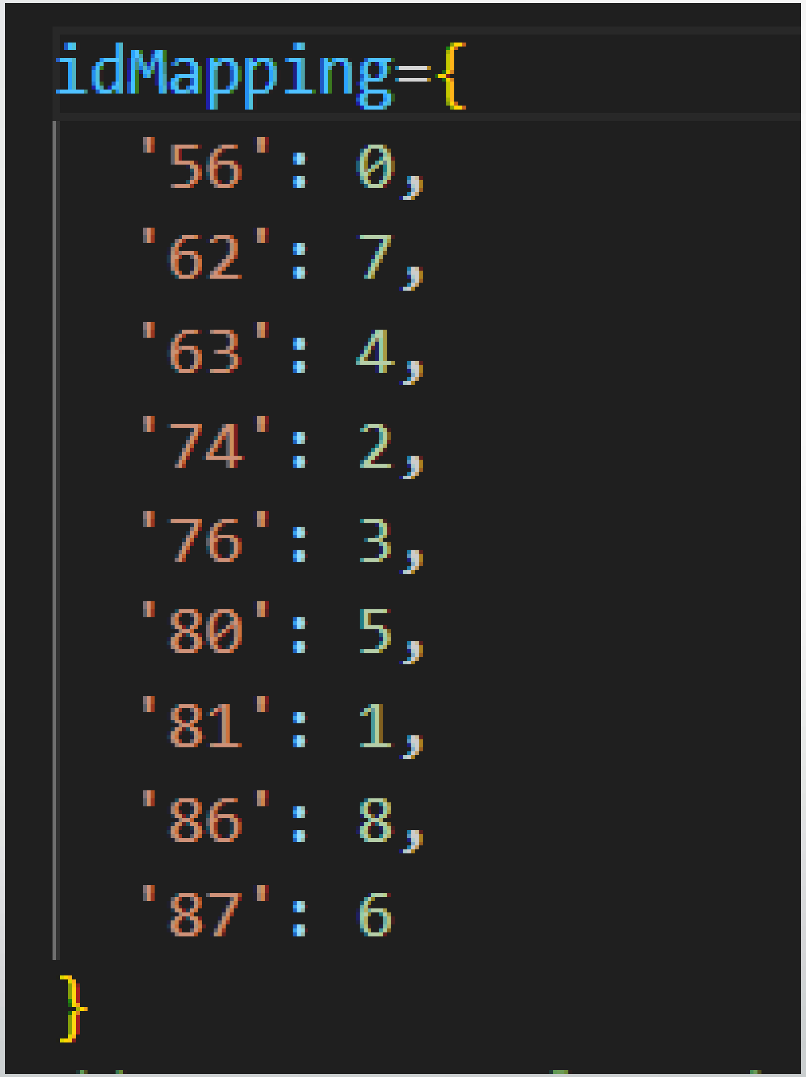
Output

```
{
  id: 74,
  parentId: null,
  children: [
    {
      id: 62,
      parentId: 74,
      children: [{ id: 56, parentId: 62 }, { id: 63, parentId: 62 }],
    },
    {
      id: 86,
      parentId: 74,
      children: [
        {
          id: 80,
          parentId: 86,
          children: [{ id: 81, parentId: 80 }, { id: 76, parentId: 80 }],
        },
        { id: 87, parentId: 86 },
      ],
    },
  ],
};
```



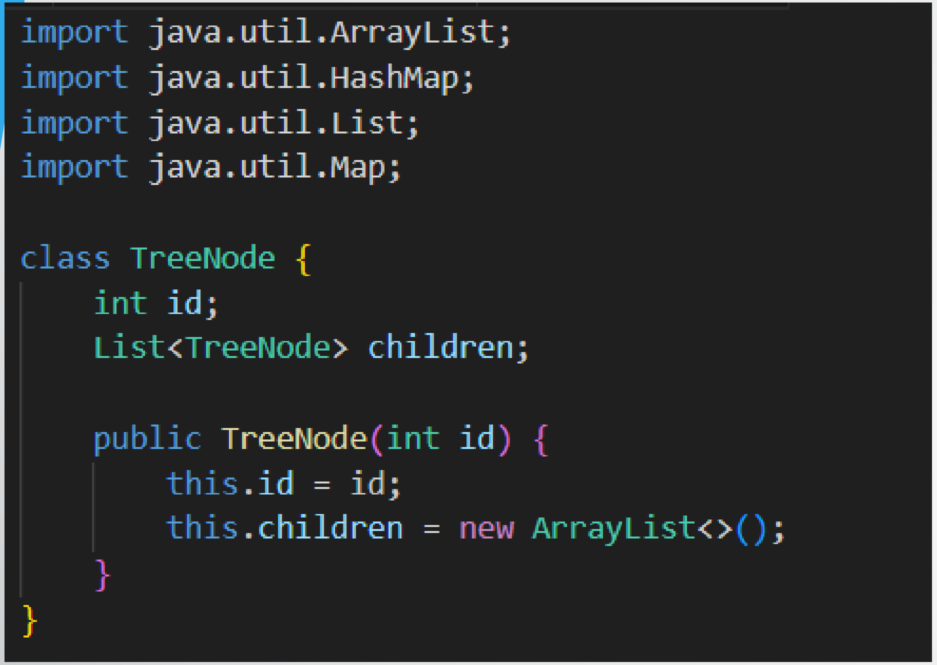
Code

```
// Get array location of each ID
const idMapping = {};
data.forEach((el,idx)=>idMapping[el.id]=idx)
// console.log(idMapping);

let root;
data.forEach((el,idx) => {
  // Handle the root element
  if (el.parentId === null) {
    root = el; //{id:74,parentId:null}
    return;
  }
  // Use our mapping to locate the parent element in our data array
    const parentEl = data[idMapping[el.parentId]];
    console.log(idx,parentEl)
  // Add our current el to its parent's `children` array
    parentEl.children = [...(parentEl.children || []), el];
    console.log(idx,parentEl);
});
```

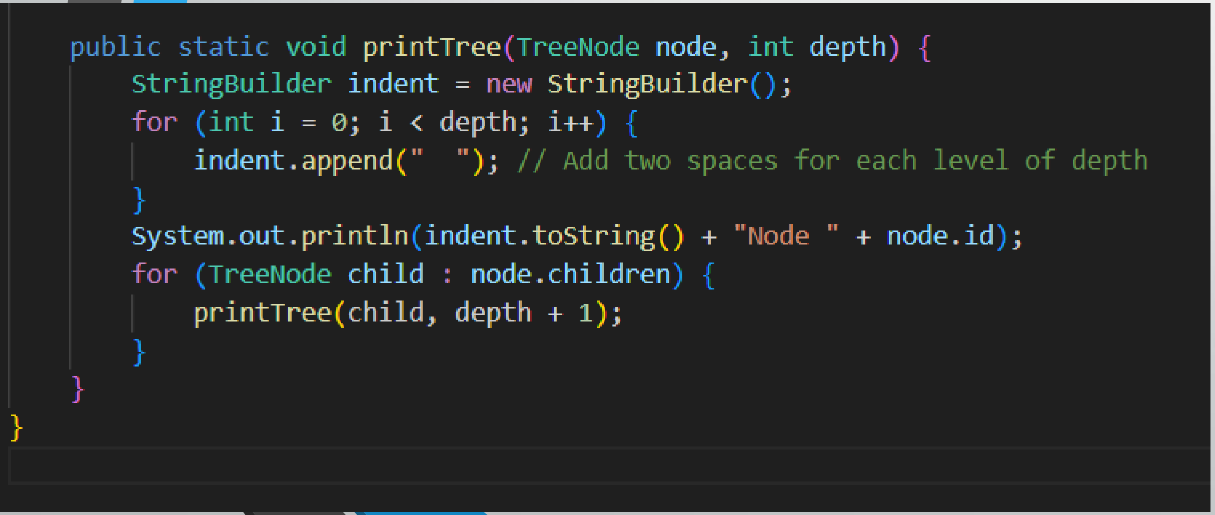


```
idMapping={
    '56': 0,
    '62': 7,
    '63': 4,
    '74': 2,
    '76': 3,
    '80': 5,
    '81': 1,
    '86': 8,
    '87': 6
}
```

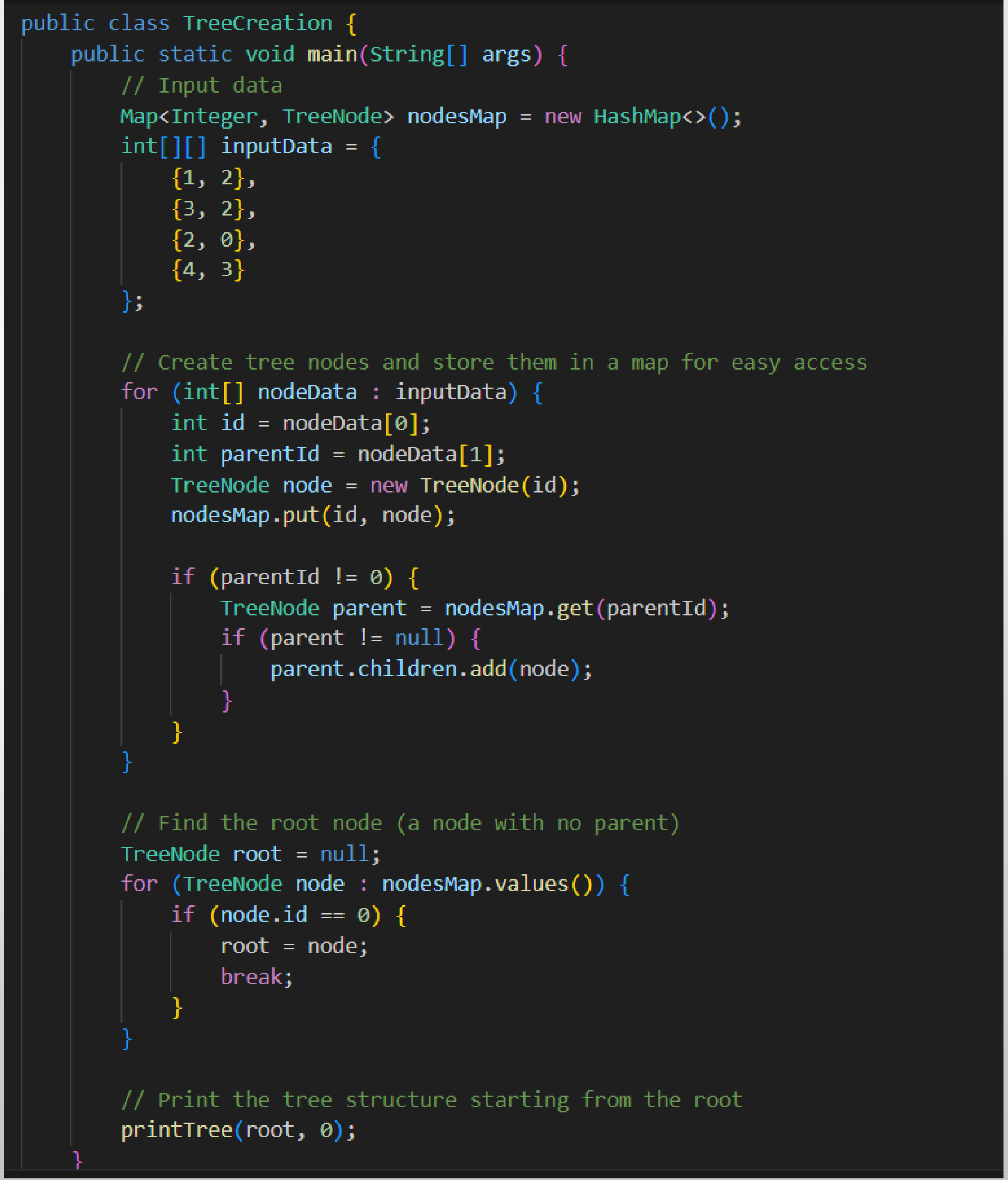
In Java



```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

class TreeNode {
    int id;
    List<TreeNode> children;

    public TreeNode(int id) {
        this.id = id;
        this.children = new ArrayList<>();
    }
}
```



```java
    public static void printTree(TreeNode node, int depth) {
        StringBuilder indent = new StringBuilder();
        for (int i = 0; i < depth; i++) {
            indent.append("  "); // Add two spaces for each level of depth
        }
        System.out.println(indent.toString() + "Node " + node.id);
        for (TreeNode child : node.children) {
            printTree(child, depth + 1);
        }
    }
}
```



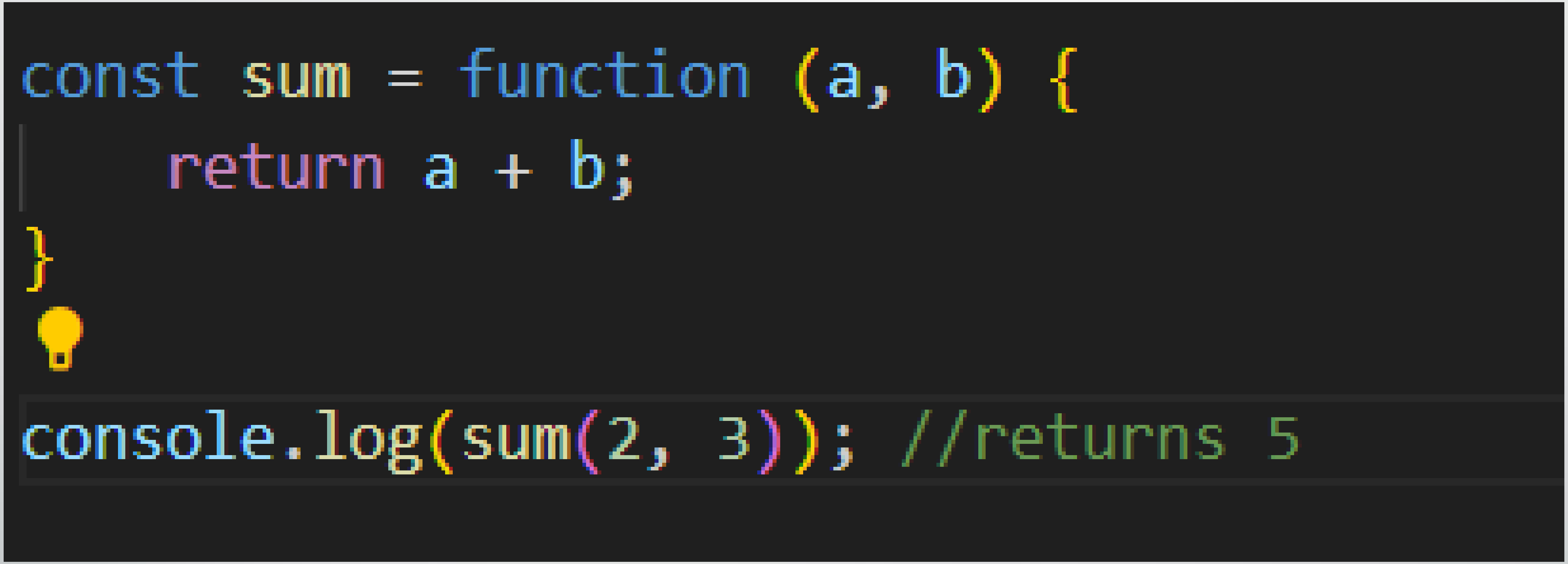
```java
public class TreeCreation {
    public static void main(String[] args) {
        // Input data
        Map<Integer, TreeNode> nodesMap = new HashMap<>();
        int[][] inputData = {
            {1, 2},
            {3, 2},
            {2, 0},
            {4, 3}
        };

        // Create tree nodes and store them in a map for easy access
        for (int[] nodeData : inputData) {
            int id = nodeData[0];
            int parentId = nodeData[1];
            TreeNode node = new TreeNode(id);
            nodesMap.put(id, node);

            if (parentId != 0) {
                TreeNode parent = nodesMap.get(parentId);
                if (parent != null) {
                    parent.children.add(node);
                }
            }
        }

        // Find the root node (a node with no parent)
        TreeNode root = null;
        for (TreeNode node : nodesMap.values()) {
            if (node.id == 0) {
                root = node;
                break;
            }
        }

        // Print the tree structure starting from the root
        printTree(root, 0);
    }
```

To conclude about Objects

- Allows ease in Data Interchange
- Objects are dynamic
- Objects are mutable
- Offers great flexibility

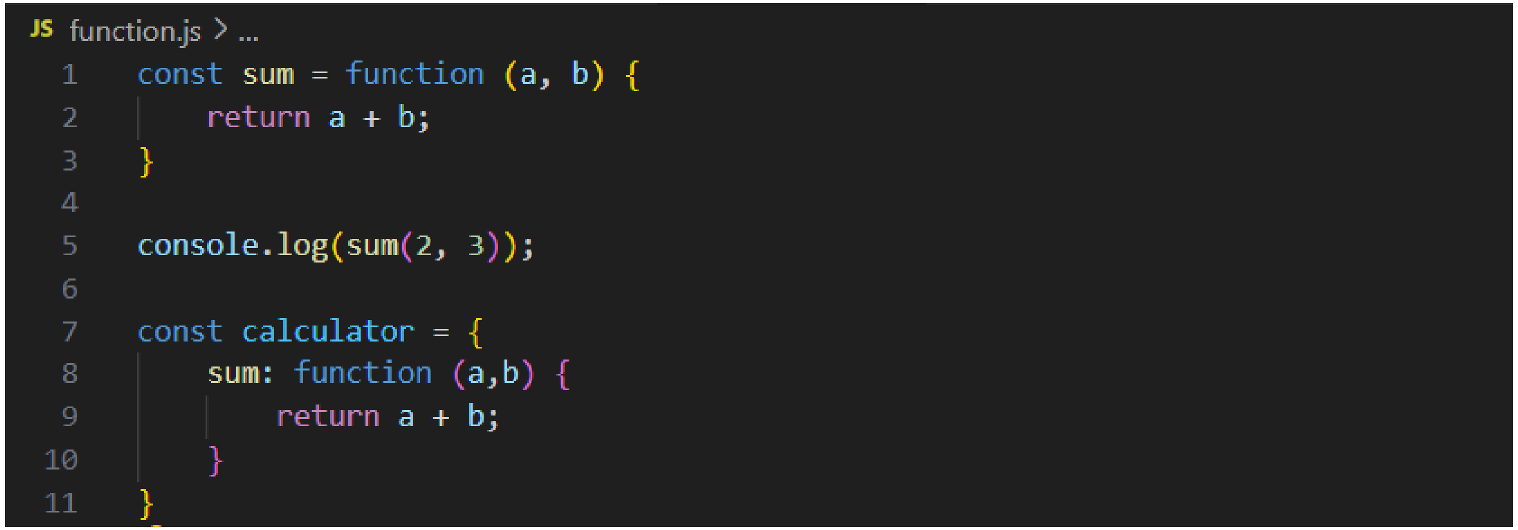
Next Good Part >>>

Functions



```
const sum = function (a, b) {
    return a + b;
}

console.log(sum(2, 3)); //returns 5
```

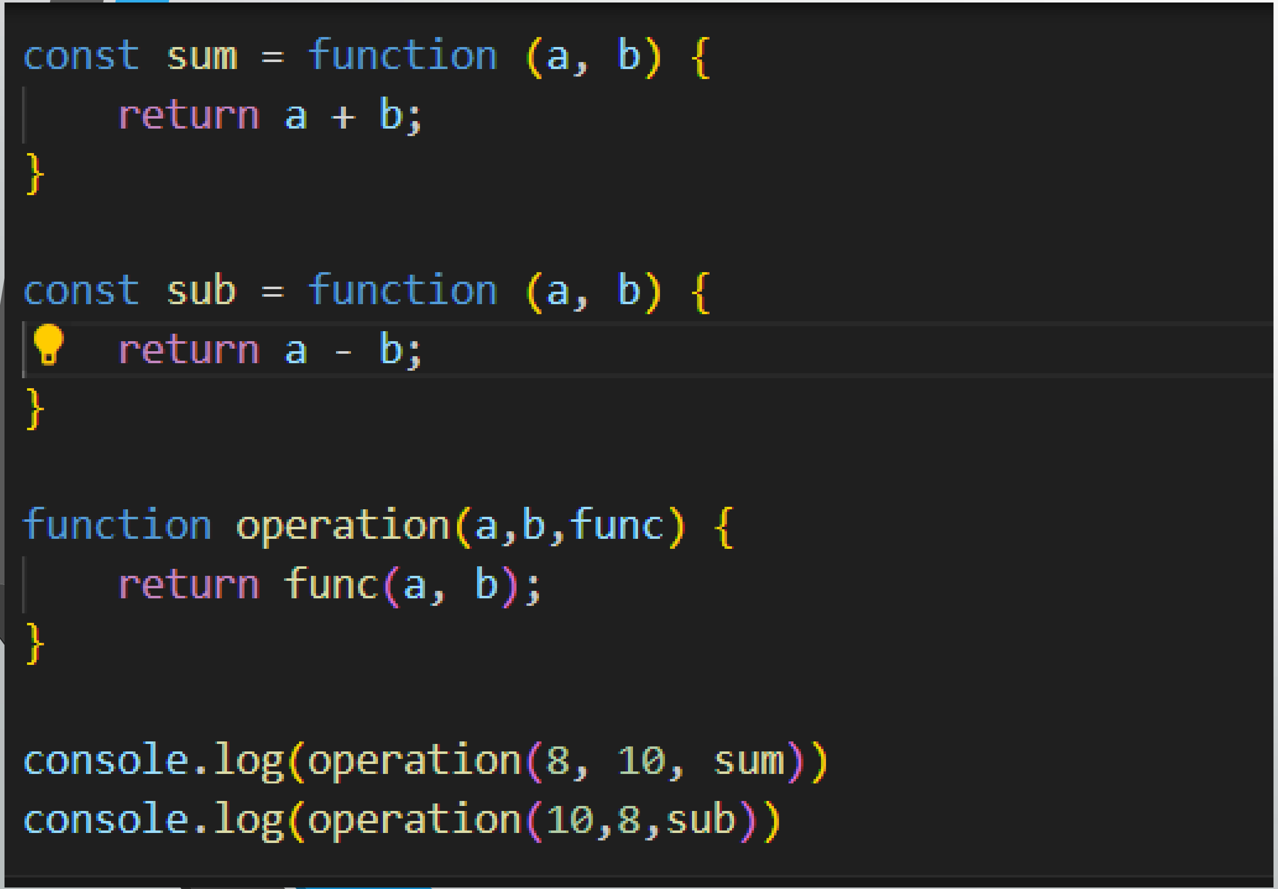
First Class Citizens



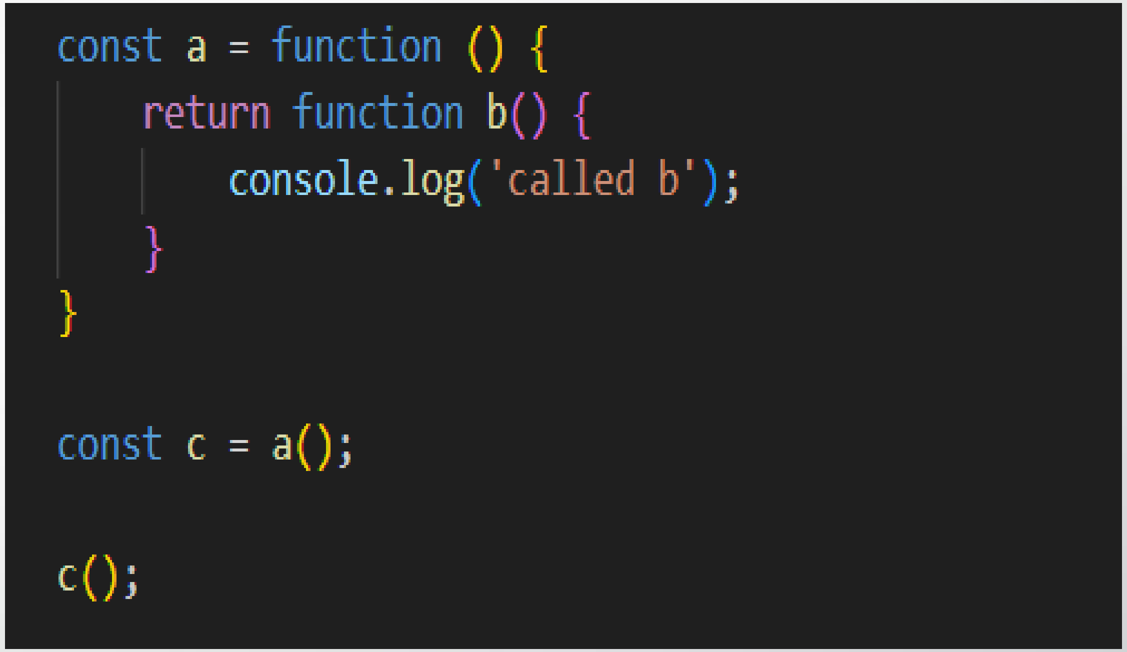
```javascript
const sum = function (a, b) {
    return a + b;
}

console.log(sum(2, 3));

const calculator = {
    sum: function (a,b) {
        return a + b;
    }
}
```

Functions can be stored as a value

Passing function as argument to another function



```
const sum = function (a, b) {
    return a + b;
}

const sub = function (a, b) {
    return a - b;
}

function operation(a,b,func) {
    return func(a, b);
}

console.log(operation(8, 10, sum))
console.log(operation(10,8,sub))
```



```
const a = function () {
    return function b() {
        console.log('called b');
    }
}

const c = a();

c();
```

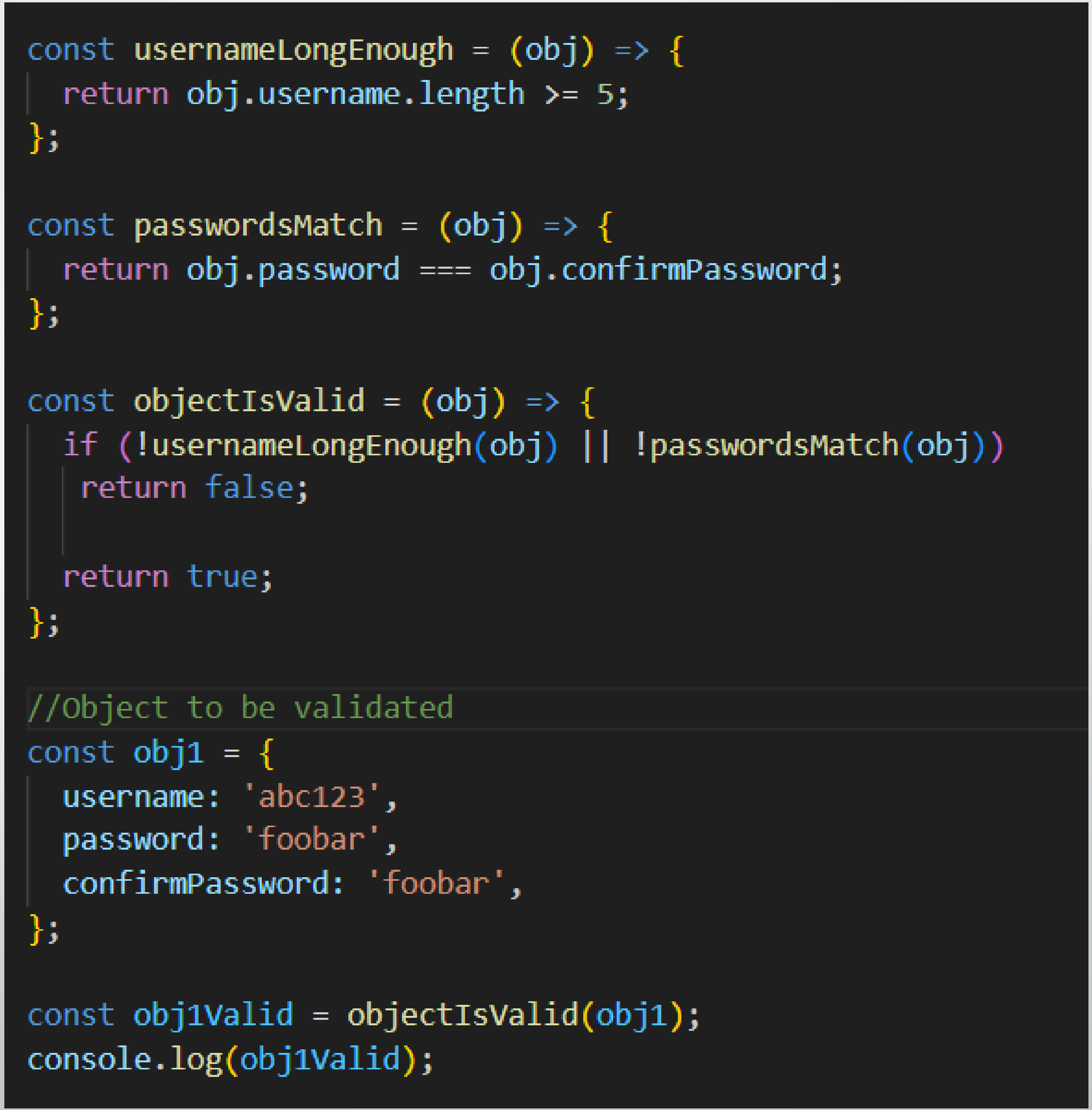
Can be returned from a function

Higher Order Functions

Example

Validate username and password property of an object.

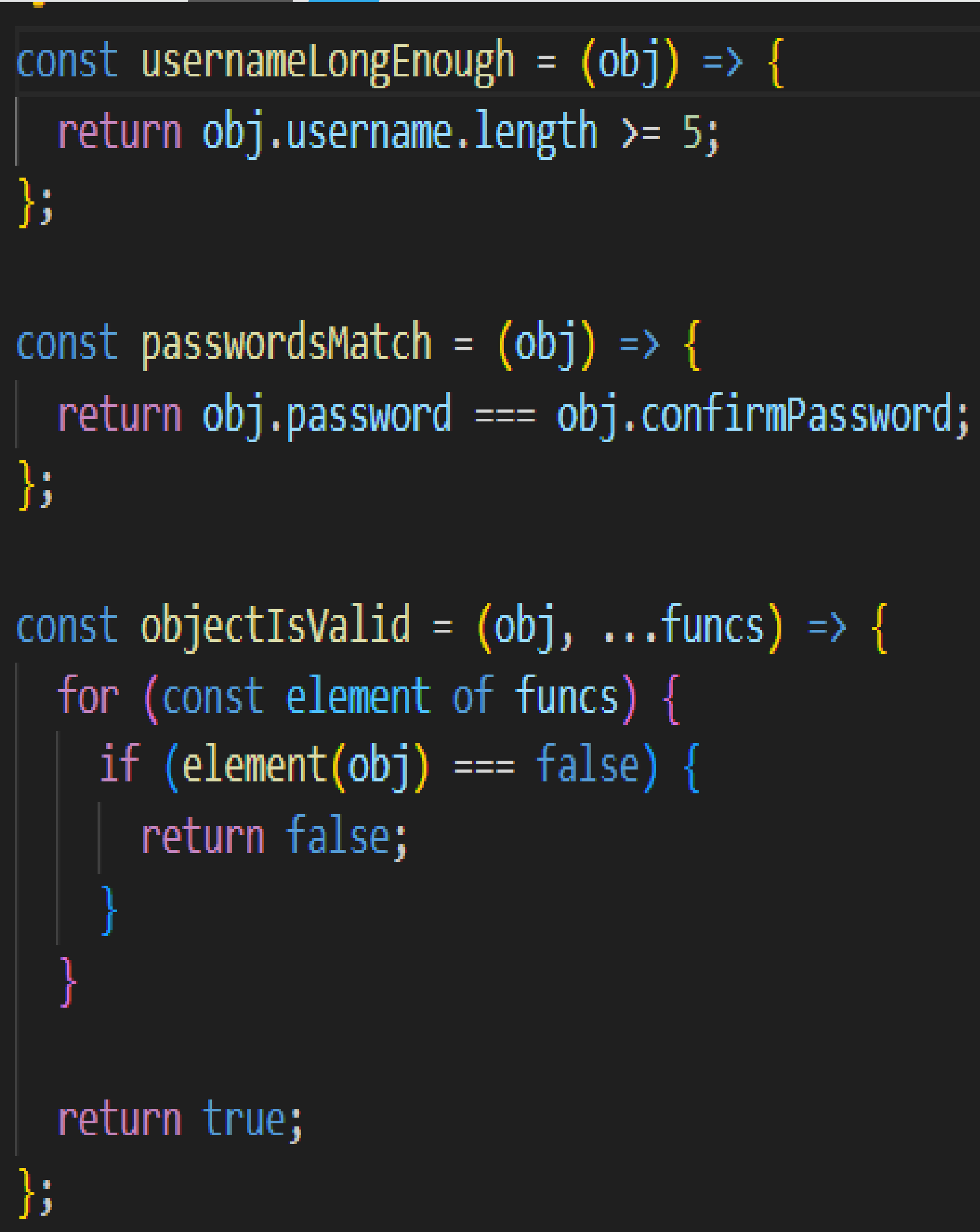
Naive Approach



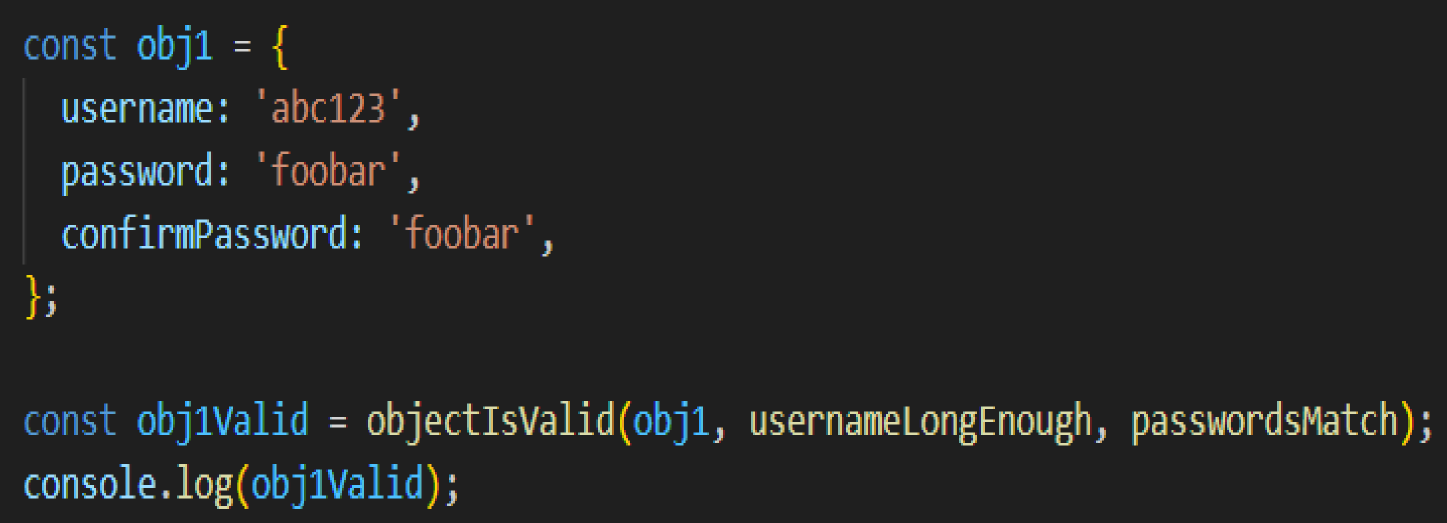
```
const usernameLongEnough = (obj) => {
  return obj.username.length >= 5;
};

const passwordsMatch = (obj) => {
  return obj.password === obj.confirmPassword;
};

const objectIsValid = (obj) => {
  if (!usernameLongEnough(obj) || !passwordsMatch(obj))
   return false;

  return true;
};

//Object to be validated
const obj1 = {
  username: 'abc123',
  password: 'foobar',
  confirmPassword: 'foobar',
};

const obj1Valid = objectIsValid(obj1);
console.log(obj1Valid);
```

No Reusability

Using HOF



```
const usernameLongEnough = (obj) => {
  return obj.username.length >= 5;
};

const passwordsMatch = (obj) => {
  return obj.password === obj.confirmPassword;
};

const objectIsValid = (obj, ...funcs) => {
  for (const element of funcs) {
    if (element(obj) === false) {
      return false;
    }
  }

  return true;
};
```



```
const obj1 = {
  username: 'abc123',
  password: 'foobar',
  confirmPassword: 'foobar',
};

const obj1Valid = objectIsValid(obj1, usernameLongEnough, passwordsMatch);
console.log(obj1Valid);
```

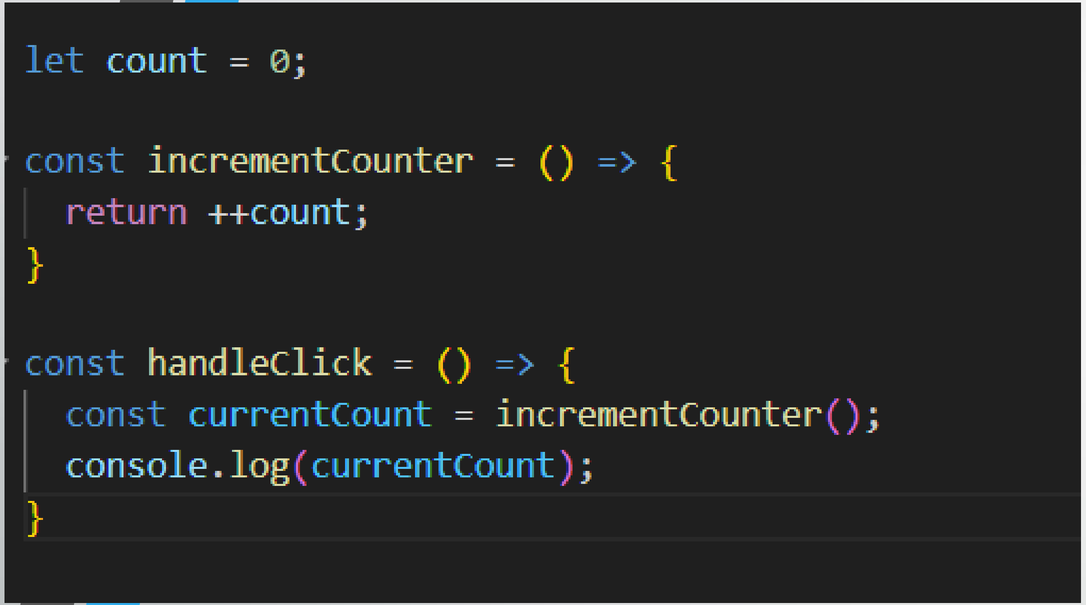
Closures

YO DAWG, I HEARD YOU LIKE CLOSURES
SO WE PUT A JAVASCRIPT FUNCTION INSIDE A JAVASCRIPT FUNCTION
imgflip.com

Example

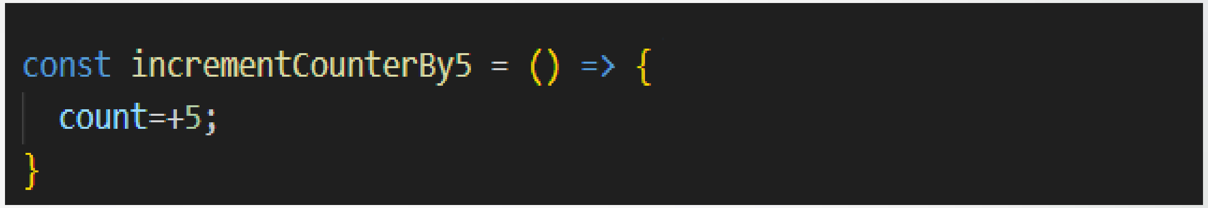
- Count the number of times a button is clicked

Without closure



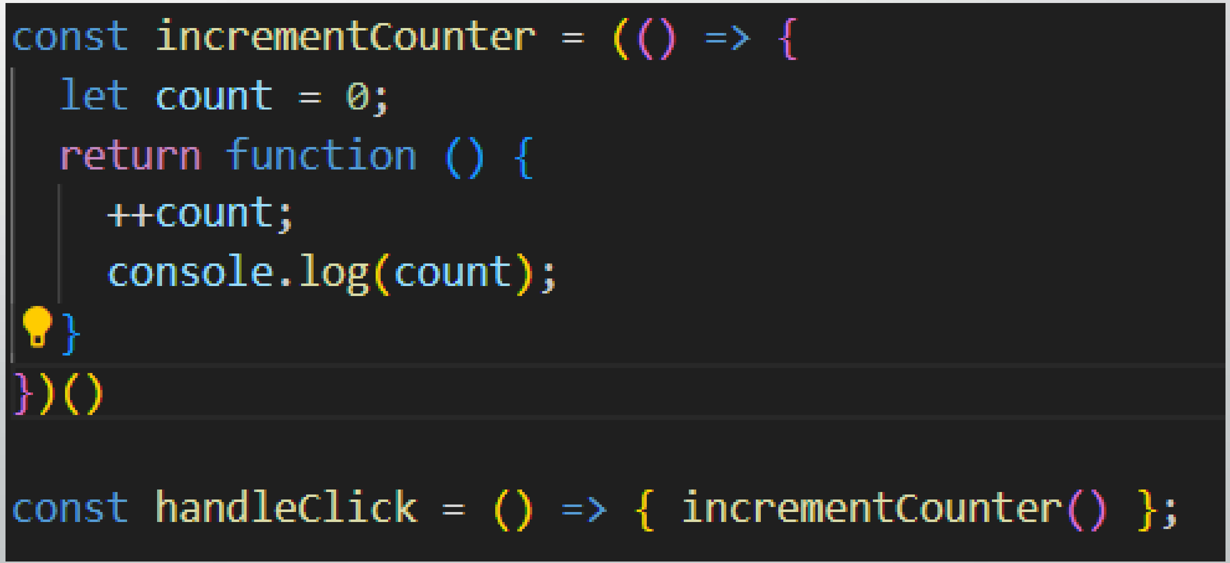
```
let count = 0;

const incrementCounter = () => {
  return ++count;
}

const handleClick = () => {
  const currentCount = incrementCounter();
  console.log(currentCount);
}
```

Possible Problem



```
const incrementCounterBy5 = () => {
  count=+5;
}
```

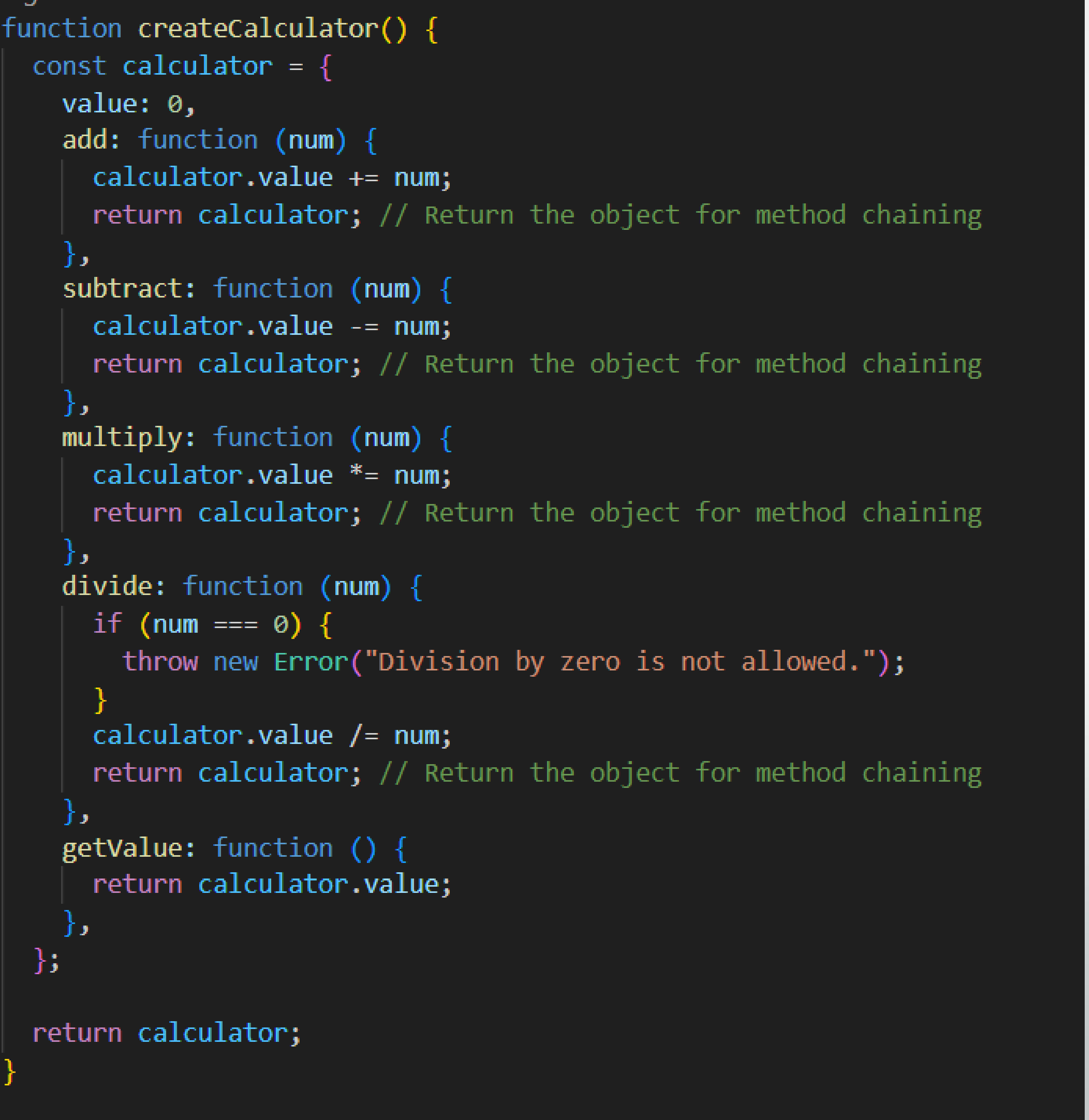
With closures



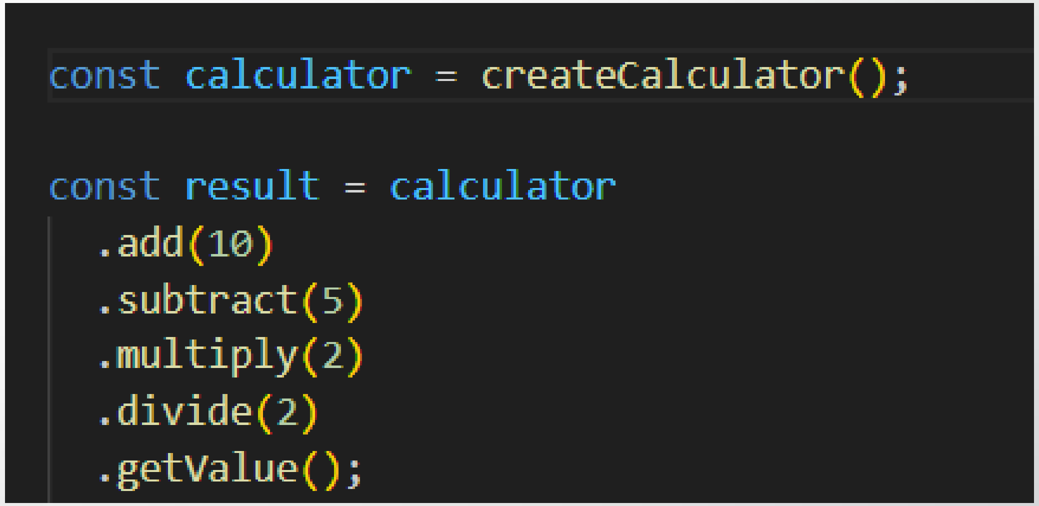
```
const incrementCounter = (() => {
  let count = 0;
  return function () {
    ++count;
    console.log(count);
  }
})()

const handleClick = () => { incrementCounter() };
```

Another Example

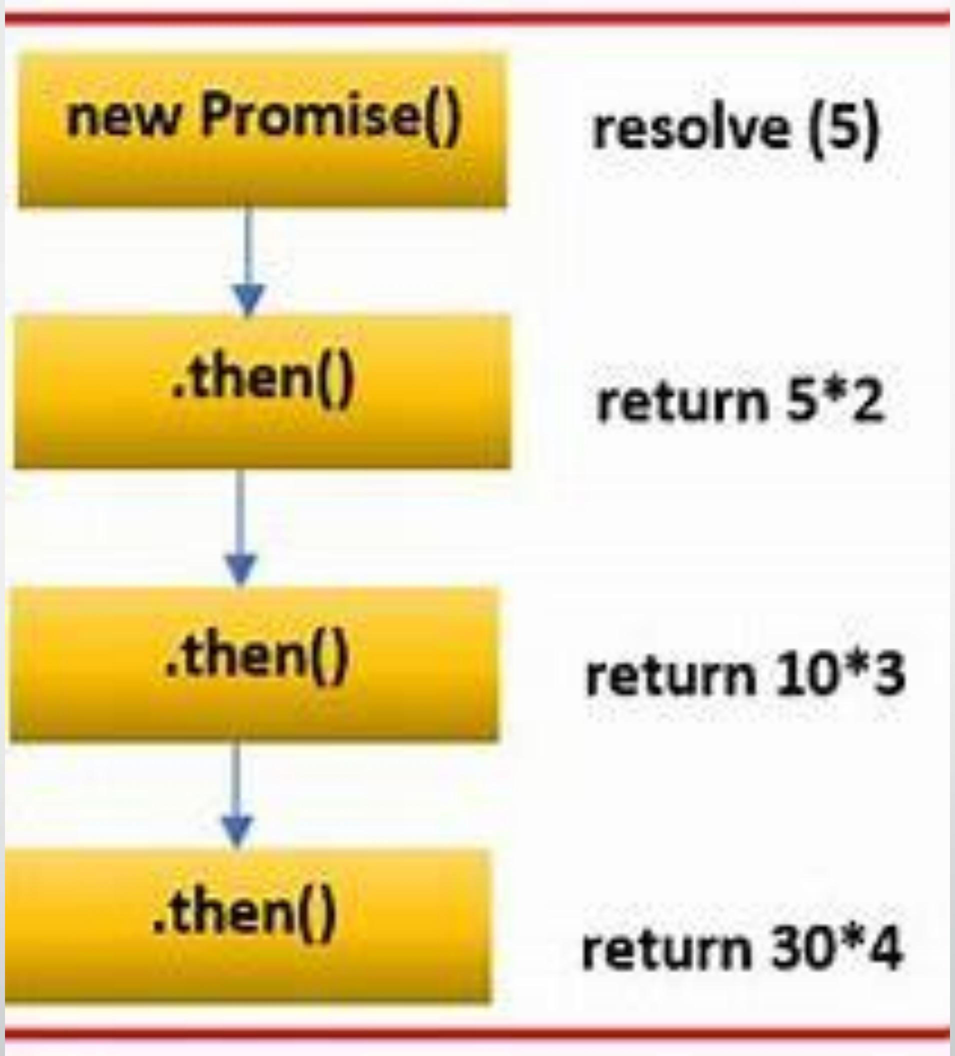
Chaining of Methods



```javascript
function createCalculator() {
  const calculator = {
    value: 0,
    add: function (num) {
      calculator.value += num;
      return calculator; // Return the object for method chaining
    },
    subtract: function (num) {
      calculator.value -= num;
      return calculator; // Return the object for method chaining
    },
    multiply: function (num) {
      calculator.value *= num;
      return calculator; // Return the object for method chaining
    },
    divide: function (num) {
      if (num === 0) {
        throw new Error("Division by zero is not allowed.");
      }
      calculator.value /= num;
      return calculator; // Return the object for method chaining
    },
    getValue: function () {
      return calculator.value;
    },
  };

  return calculator;
}
```



```javascript
const calculator = createCalculator();

const result = calculator
  .add(10)
  .subtract(5)
  .multiply(2)
  .divide(2)
  .getValue();
```



new Promise()
resolve (5)
.then()
return 5*2
.then()
return 10*3
.then()
return 30*4

Benefits

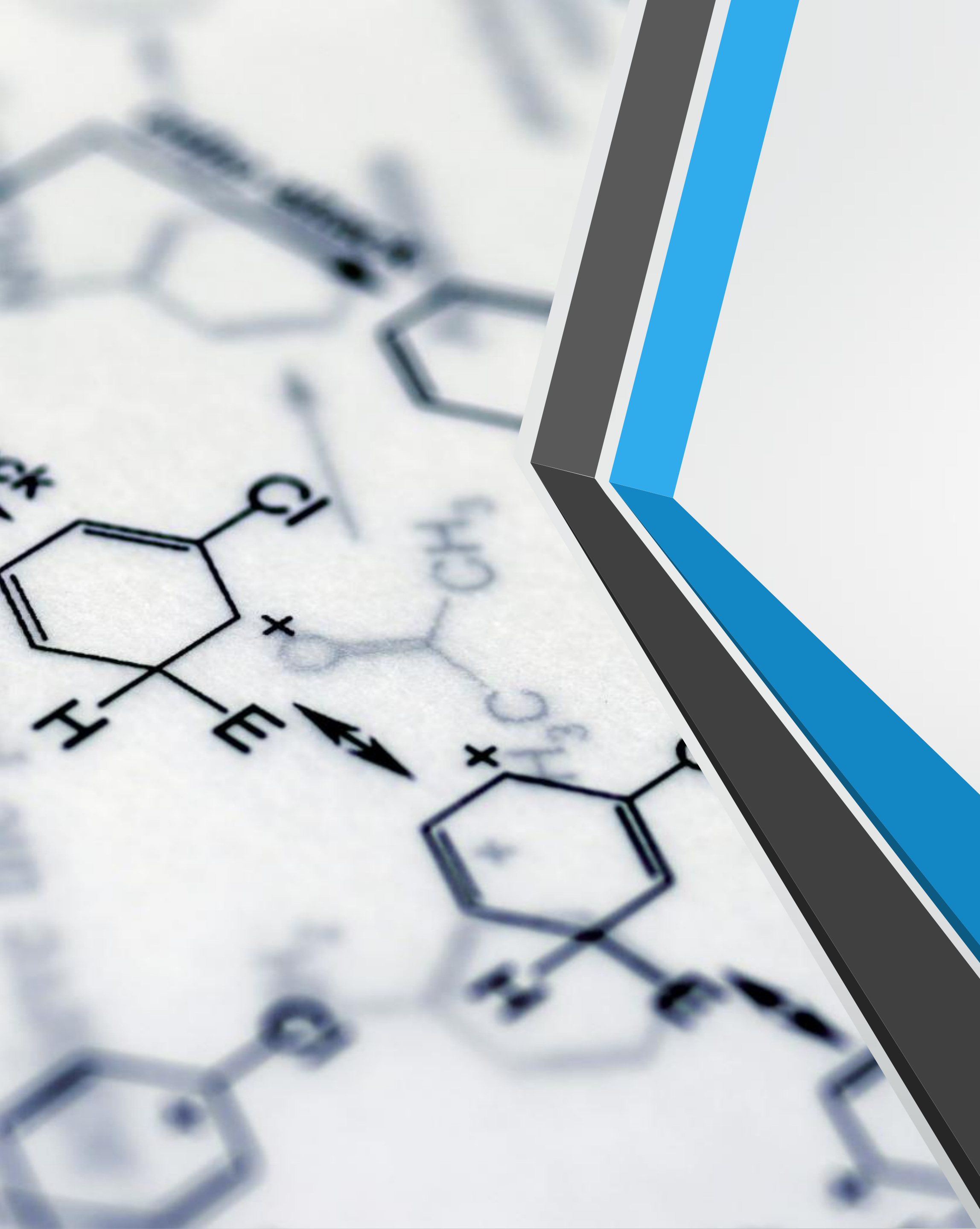
- Asynchronous Operations
- Callbacks
- Debouncing
- Memoization
- Composition
- Separation of Concerns

Base of Functional Programming

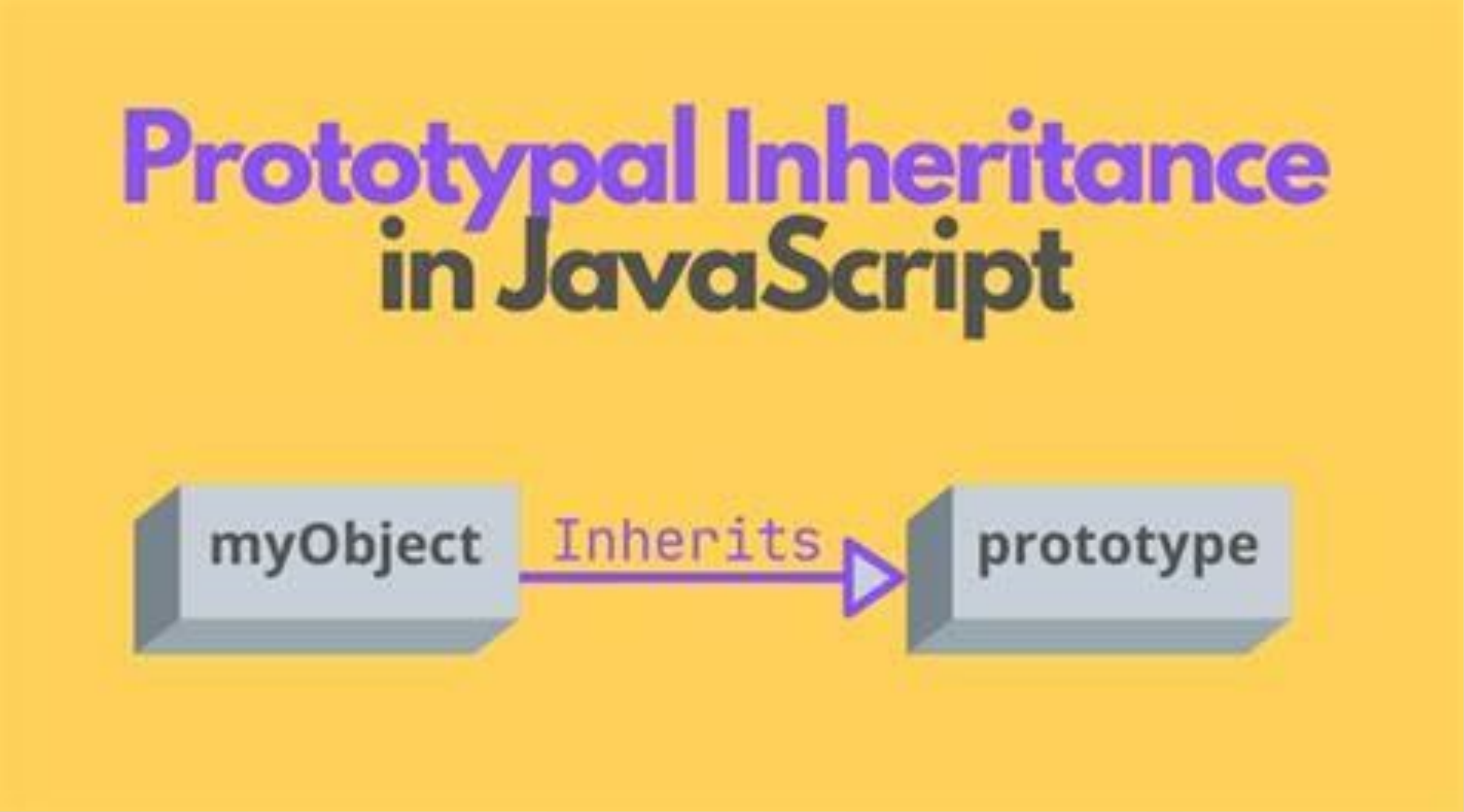
We can conclude

- Functions as first class citizens offers great value
- They brings reusability
- Hides Data
- Build APIs chaining that are easy to consume
- Helps in creating modular applications

One more Good Part ….

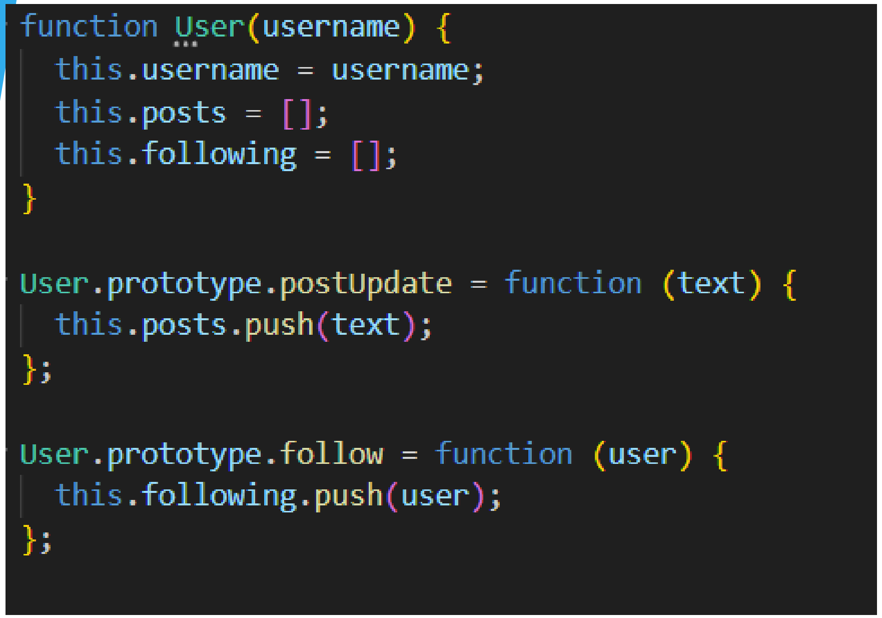


Prototypal inheritance

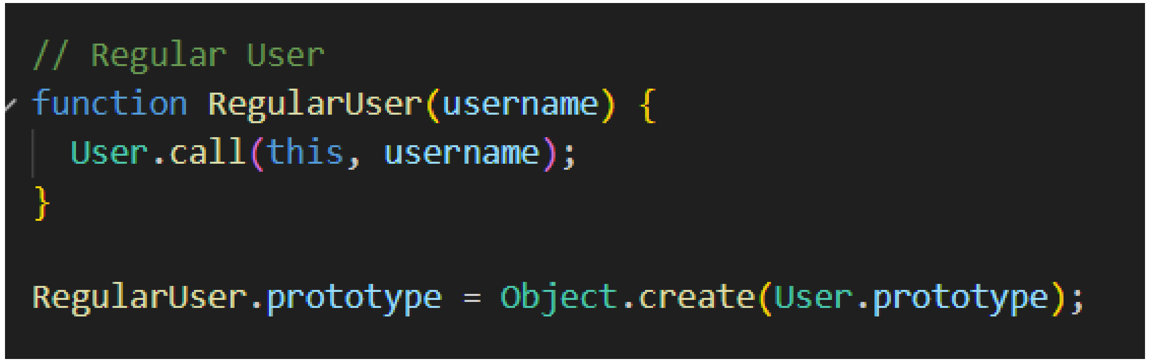


Prototypal Inheritance in JavaScript

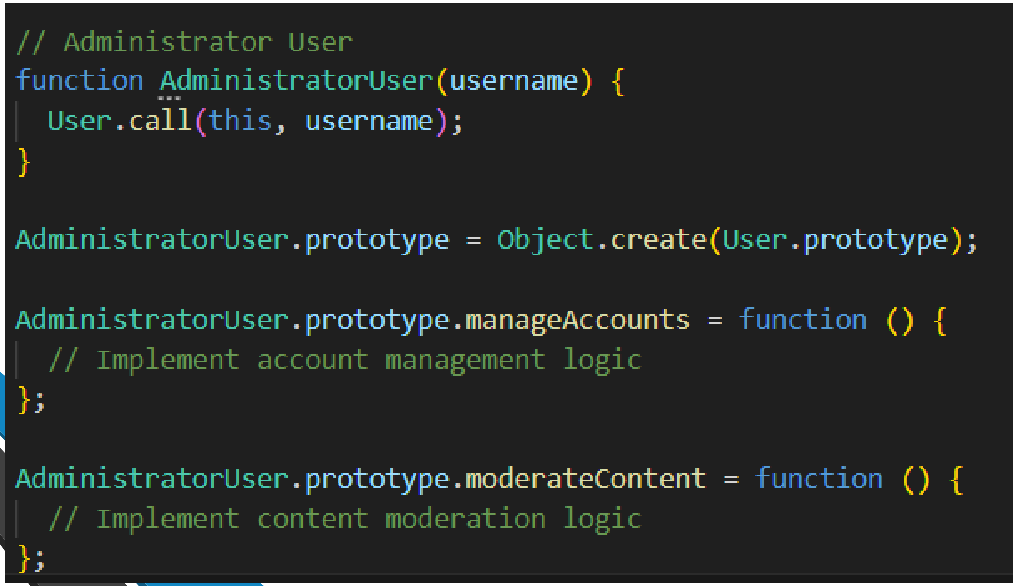
Example of Prototypal Inheritance



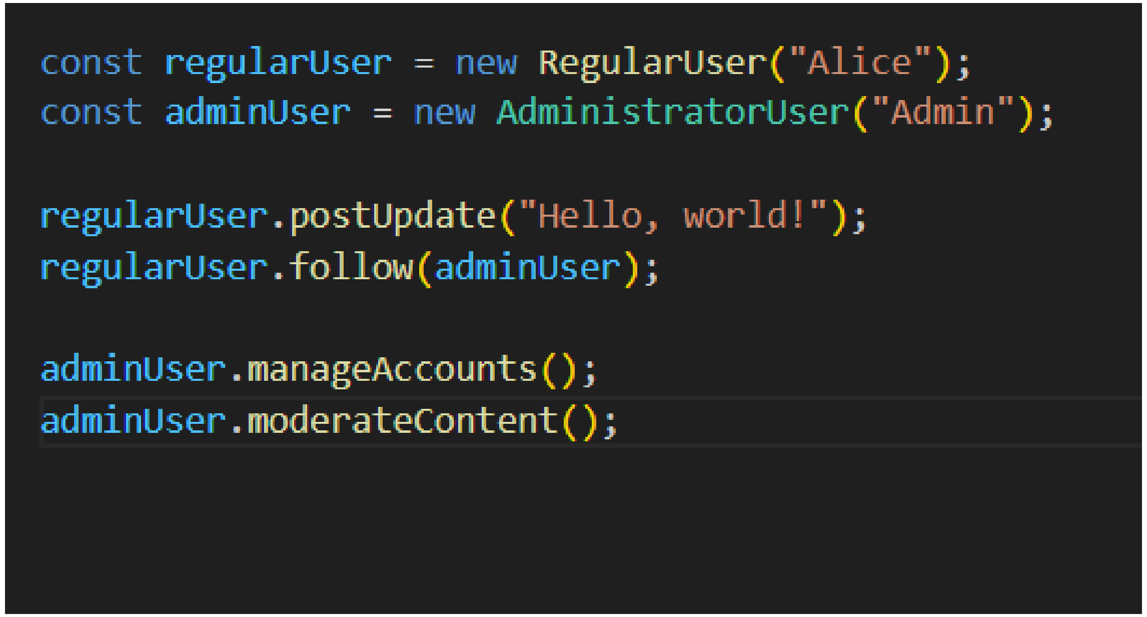
```javascript
function User(username) {
  this.username = username;
  this.posts = [];
  this.following = [];
}

User.prototype.postUpdate = function (text) {
  this.posts.push(text);
};

User.prototype.follow = function (user) {
  this.following.push(user);
};
```



```javascript
// Regular User
function RegularUser(username) {
  User.call(this, username);
}

RegularUser.prototype = Object.create(User.prototype);
```



```javascript
// Administrator User
function AdministratorUser(username) {
  User.call(this, username);
}

AdministratorUser.prototype = Object.create(User.prototype);

AdministratorUser.prototype.manageAccounts = function () {
  // Implement account management logic
};

AdministratorUser.prototype.moderateContent = function () {
  // Implement content moderation logic
};
```



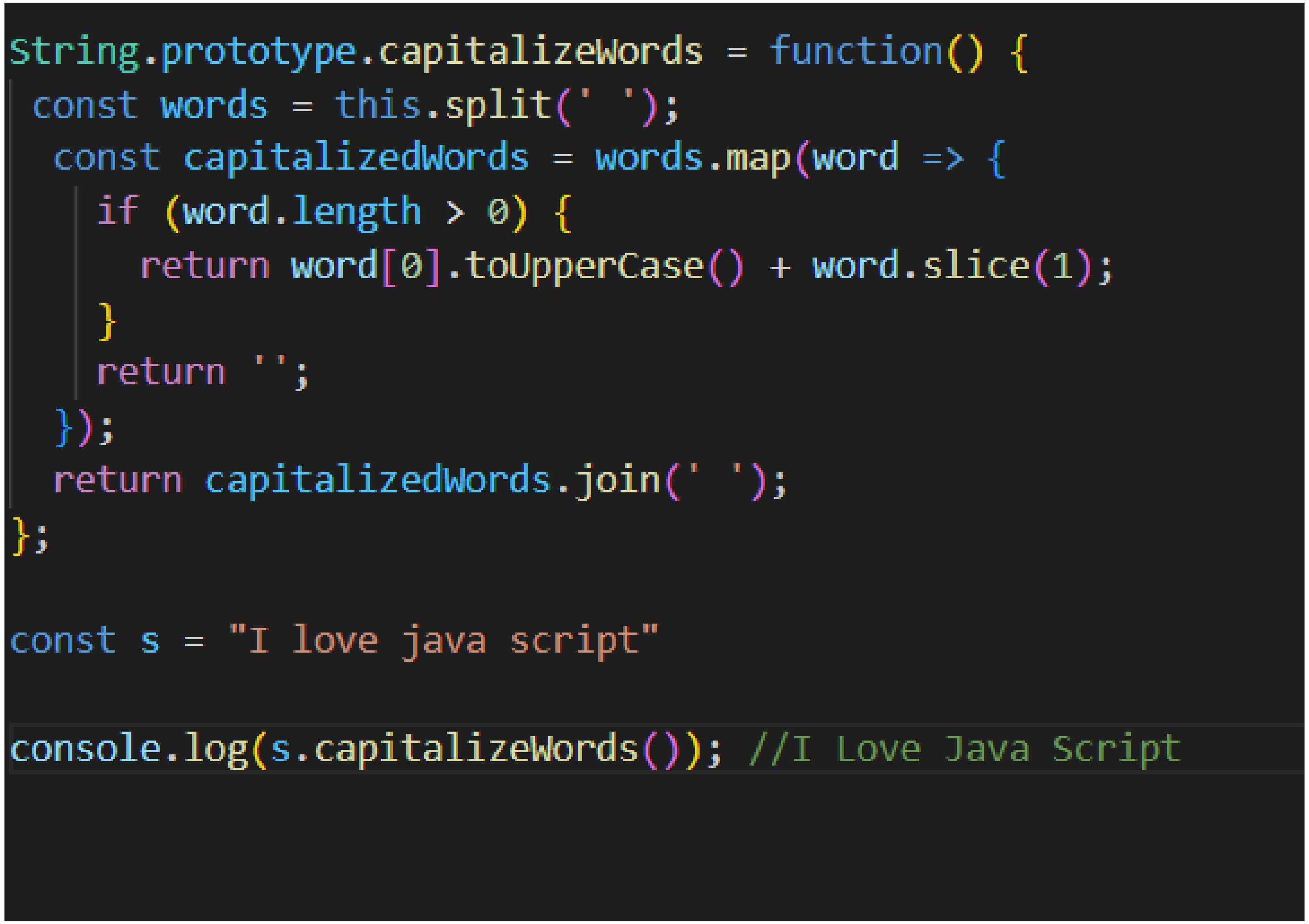
```javascript
const regularUser = new RegularUser("Alice");
const adminUser = new AdministratorUser("Admin");

regularUser.postUpdate("Hello, world!");
regularUser.follow(adminUser);

adminUser.manageAccounts();
adminUser.moderateContent();
```

Why it is different from Classical Inheritance?

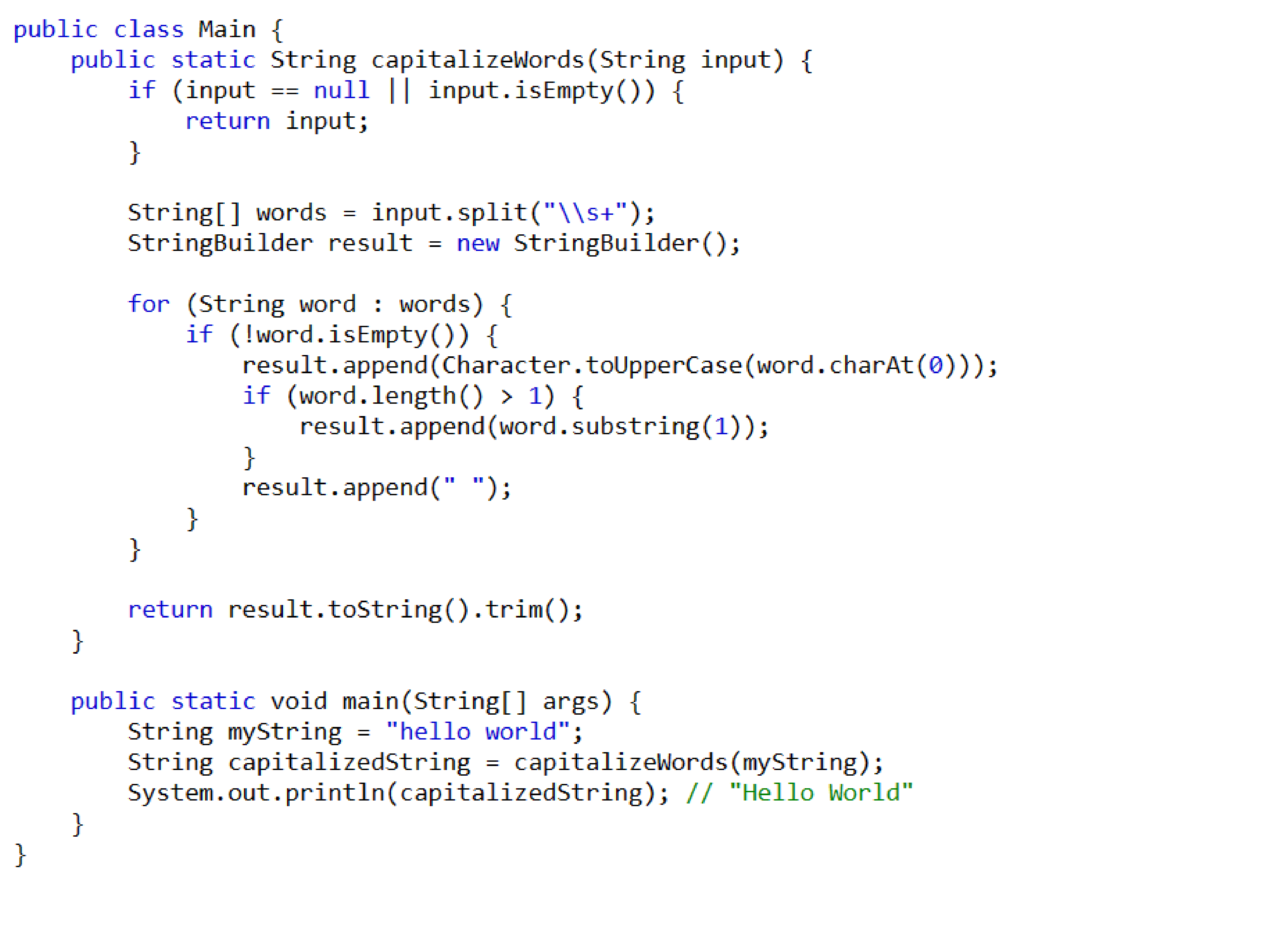
- Clear Hierarchies
- Flexibility
- Reusability
- Memory efficient

Augmenting Types

No inbuilt Function for Capitalizing each word !!??



```
String.prototype.capitalizeWords = function() {
 const words = this.split(' ');
  const capitalizedWords = words.map(word => {
    if (word.length > 0) {
      return word[0].toUpperCase() + word.slice(1);
    }
    return '';
  });
  return capitalizedWords.join(' ');
};

const s = "I love java script"

console.log(s.capitalizeWords()); //I Love Java Script
```



```java
public class Main {
    public static String capitalizeWords(String input) {
        if (input == null || input.isEmpty()) {
            return input;
        }

        String[] words = input.split("\\s+");
        StringBuilder result = new StringBuilder();

        for (String word : words) {
            if (!word.isEmpty()) {
                result.append(Character.toUpperCase(word.charAt(0)));
                if (word.length() > 1) {
                    result.append(word.substring(1));
                }
                result.append(" ");
            }
        }

        return result.toString().trim();
    }

    public static void main(String[] args) {
        String myString = "hello world";
        String capitalizedString = capitalizeWords(myString);
        System.out.println(capitalizedString); // "Hello World"
    }
}
```

‘The joy of JavaScript is in its lack of rigidity and the infinite possibilities that this allows for’

Key Takeaways

- Everything is Object
- Objects are similar to JSON
- Convenience in data interchange
- working with Complex Data Structures made easy

Key Takeaways

- Functions are first class Citizens
- Brings Modularity
- Hides Information
- Base of Functional Programming
- Provides ease in building APIs, asynchronous behavior

Key Takeaways

- Objects can inherit other objects
- Allows Augmenting Built-in Types
- Offers Reusability
- No need to write boilerplate Code

Not over yet

Best Practices

- Keep your functions pure.
- Avoid side effects.
- Focus on code reusability
- Avoid Global variables and objects.
- Be careful while using mutation

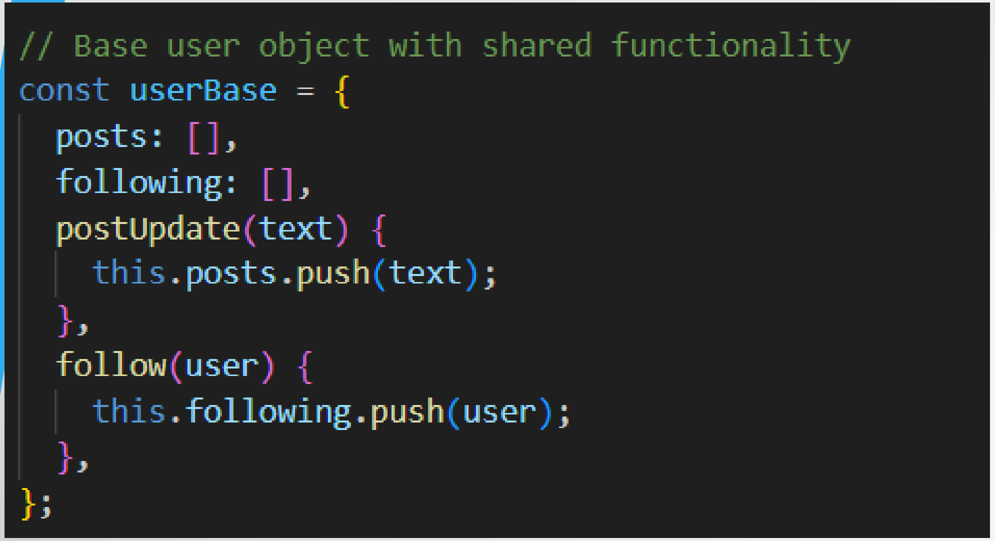
Thanks for Listening

You can connect with me on

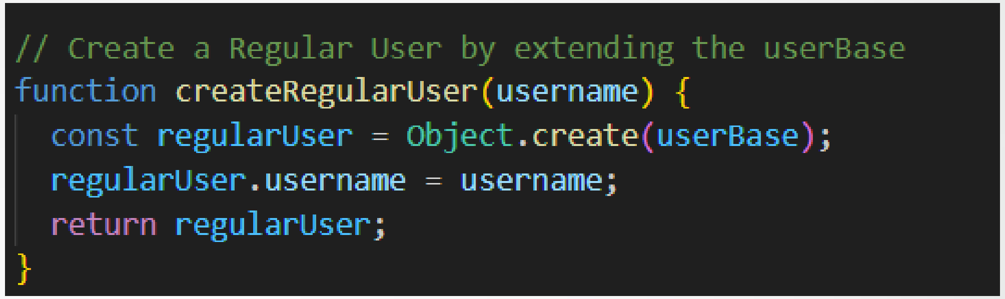
https://www.linkedin.com/in/sonamguptacs/

Questions ?

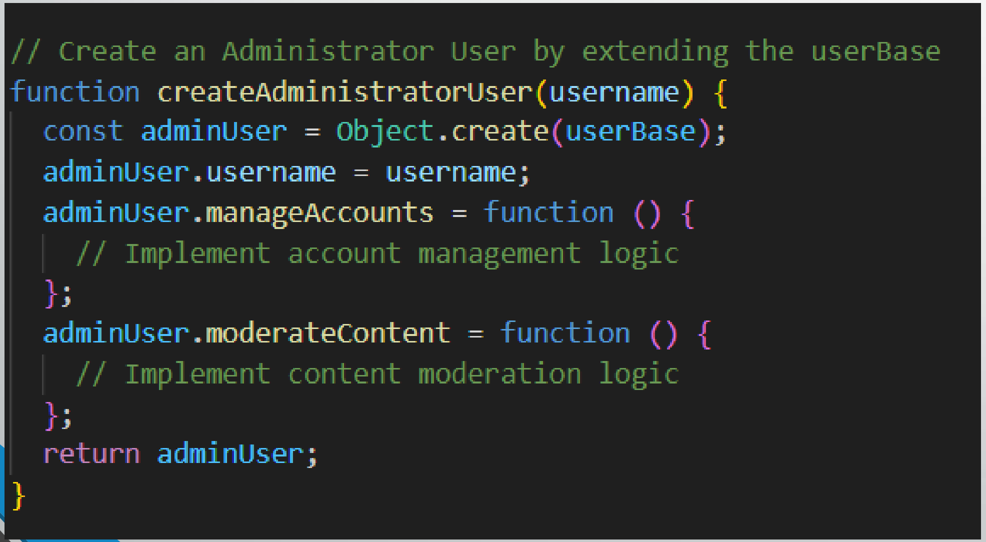
Concatenative Inheritance



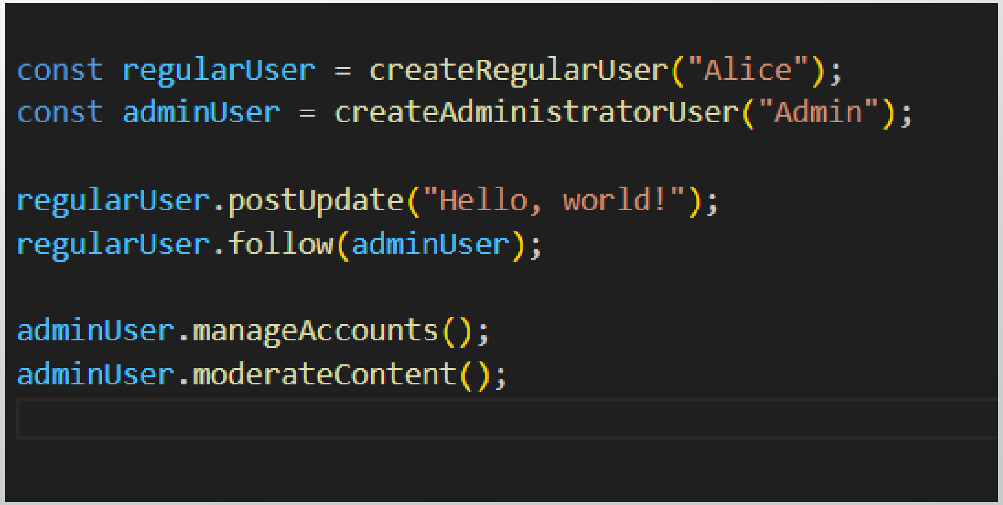
```javascript
// Base user object with shared functionality
const userBase = {
  posts: [],
  following: [],
  postUpdate(text) {
    this.posts.push(text);
  },
  follow(user) {
    this.following.push(user);
  },
};
```



```javascript
// Create a Regular User by extending the userBase
function createRegularUser(username) {
  const regularUser = Object.create(userBase);
  regularUser.username = username;
  return regularUser;
}
```

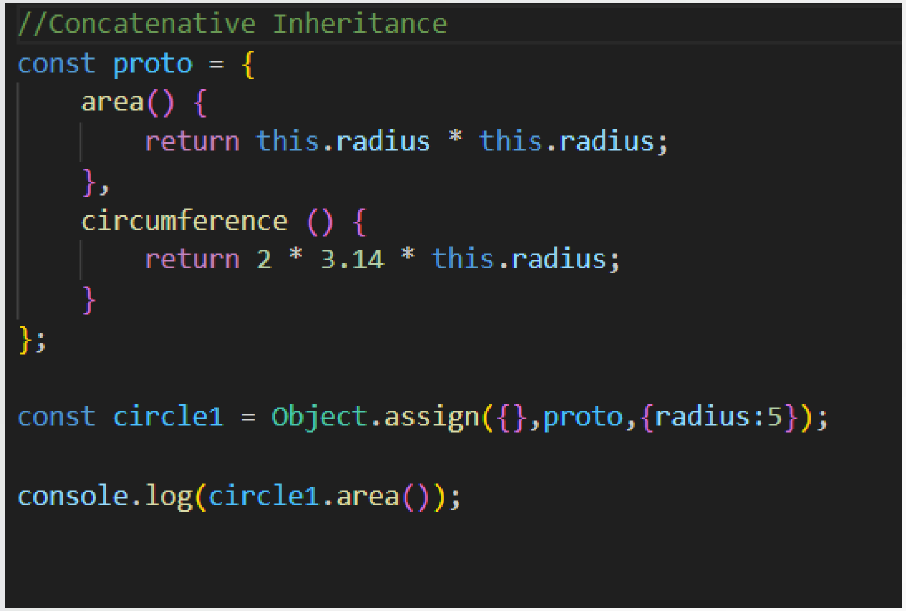


```javascript
// Create an Administrator User by extending the userBase
function createAdministratorUser(username) {
  const adminUser = Object.create(userBase);
  adminUser.username = username;
  adminUser.manageAccounts = function () {
    // Implement account management logic
  };
  adminUser.moderateContent = function () {
    // Implement content moderation logic
  };
  return adminUser;
}
```

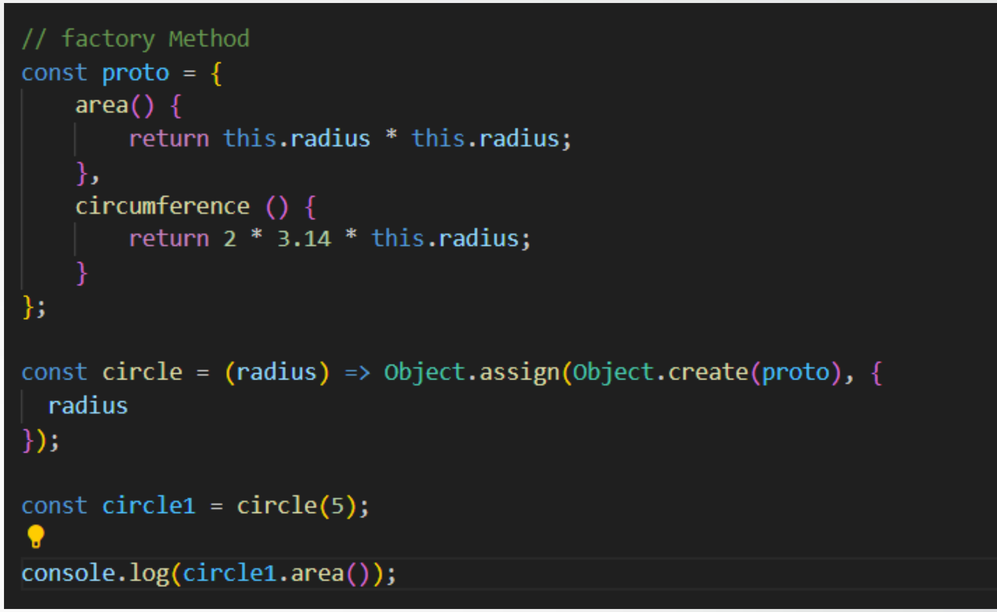


```javascript
const regularUser = createRegularUser("Alice");
const adminUser = createAdministratorUser("Admin");

regularUser.postUpdate("Hello, world!");
regularUser.follow(adminUser);

adminUser.manageAccounts();
adminUser.moderateContent();
```

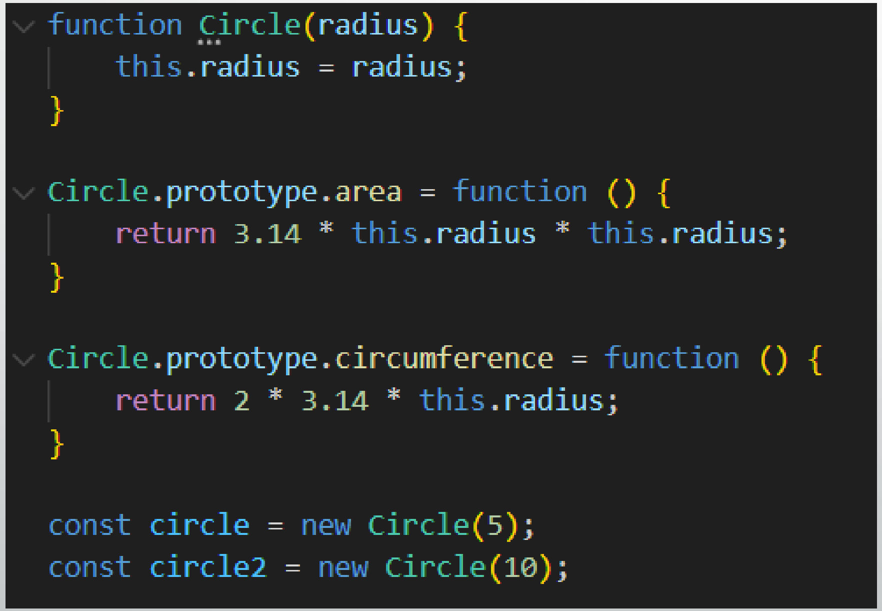
Ways to implement Prototypal Inheritance



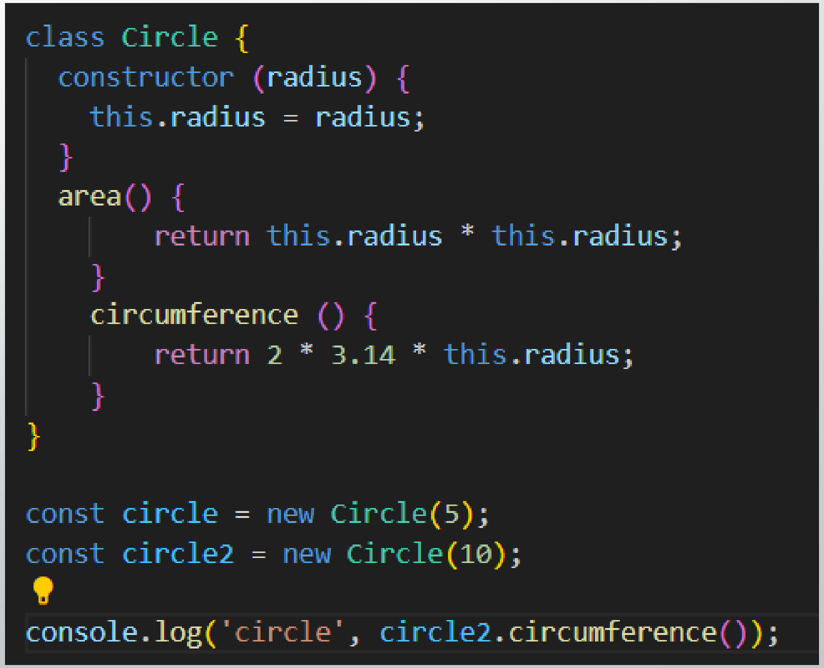
```
//Concatenative Inheritance
const proto = {
    area() {
        return this.radius * this.radius;
    },
    circumference () {
        return 2 * 3.14 * this.radius;
    }
};

const circle1 = Object.assign({},proto,{radius:5});

console.log(circle1.area());
```



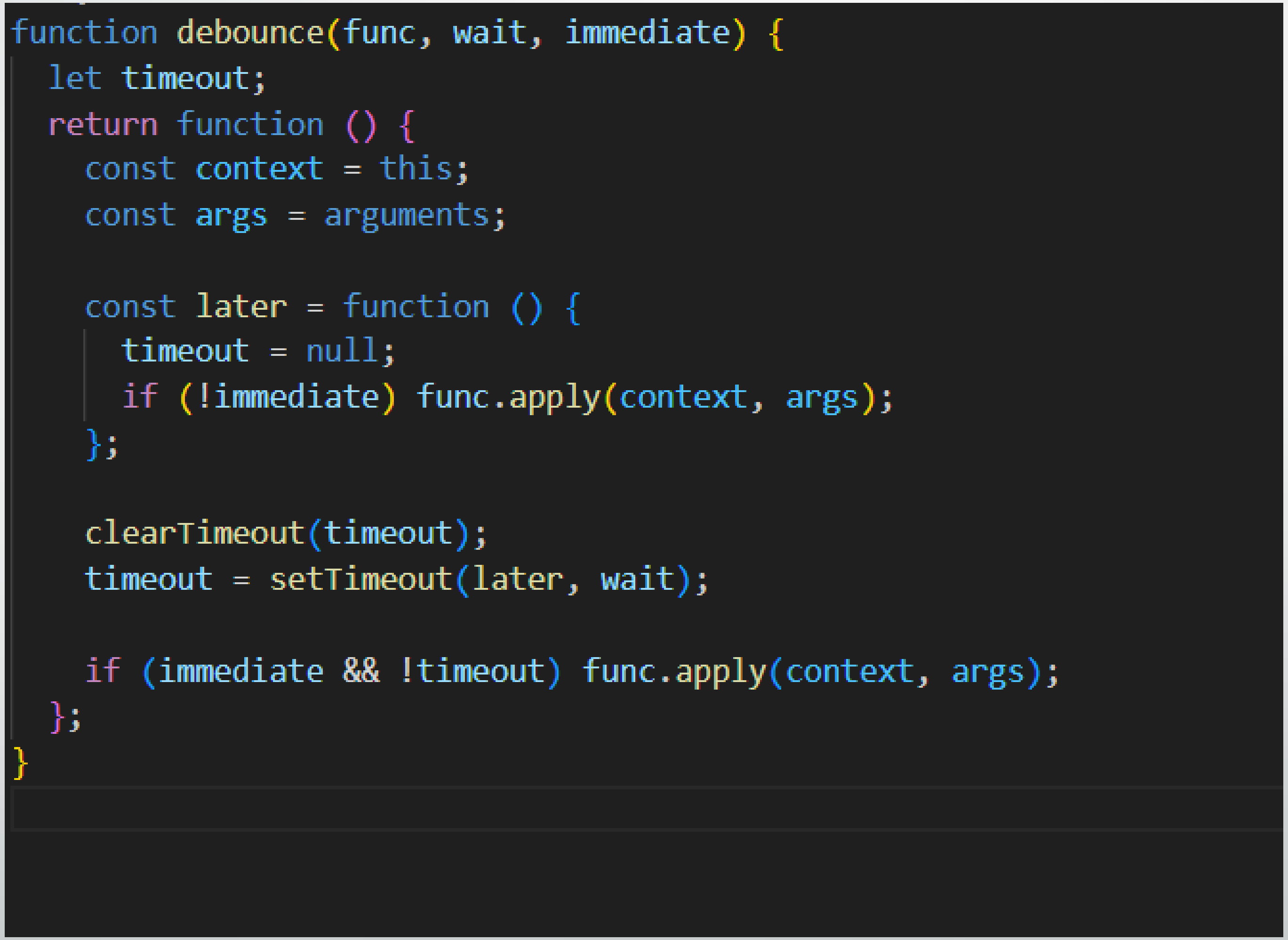
```
// factory Method
const proto = {
    area() {
        return this.radius * this.radius;
    },
    circumference () {
        return 2 * 3.14 * this.radius;
    }
};

const circle = (radius) => Object.assign(Object.create(proto), {
  radius
});

const circle1 = circle(5);

console.log(circle1.area());
```



```
function Circle(radius) {
    this.radius = radius;
}

Circle.prototype.area = function () {
    return 3.14 * this.radius * this.radius;
}

Circle.prototype.circumference = function () {
    return 2 * 3.14 * this.radius;
}

const circle = new Circle(5);
const circle2 = new Circle(10);
```

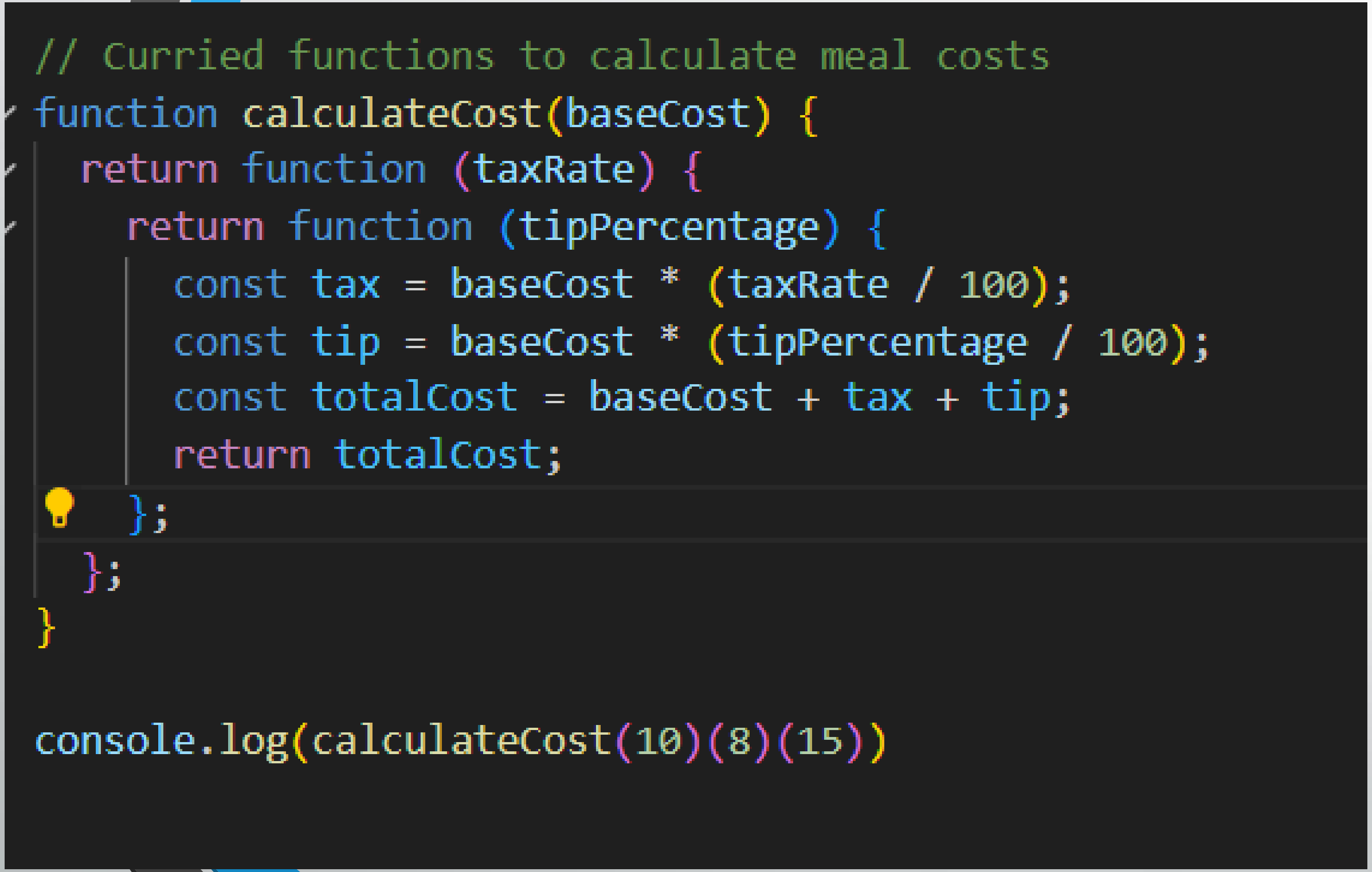


```
class Circle {
  constructor (radius) {
    this.radius = radius;
  }
  area() {
      return this.radius * this.radius;
    }
    circumference () {
      return 2 * 3.14 * this.radius;
    }
}

const circle = new Circle(5);
const circle2 = new Circle(10);

console.log('circle', circle2.circumference());
```

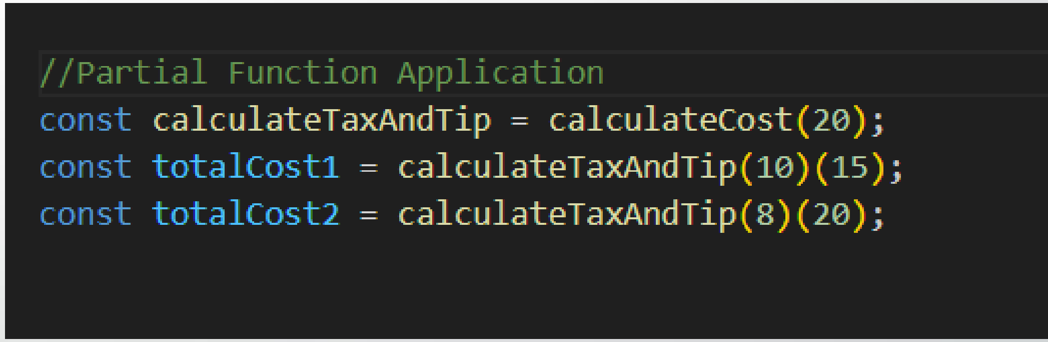


```
function debounce(func, wait, immediate) {
  let timeout;
  return function () {
    const context = this;
    const args = arguments;

    const later = function () {
      timeout = null;
      if (!immediate) func.apply(context, args);
    };

    clearTimeout(timeout);
    timeout = setTimeout(later, wait);

    if (immediate && !timeout) func.apply(context, args);
  };
}
```

Currying



```
// Curried functions to calculate meal costs
function calculateCost(baseCost) {
  return function (taxRate) {
    return function (tipPercentage) {
      const tax = baseCost * (taxRate / 100);
      const tip = baseCost * (tipPercentage / 100);
      const totalCost = baseCost + tax + tip;
      return totalCost;
    };
  };
}

console.log(calculateCost(10)(8)(15))
```



```
//Partial Function Application
const calculateTaxAndTip = calculateCost(20);
const totalCost1 = calculateTaxAndTip(10)(15);
const totalCost2 = calculateTaxAndTip(8)(20);
```