

# JavaScript Classes

Constructor, Properties, Methods, Getters, Setters



**SoftUni Team**  
Technical Trainers



**SoftUni**



**Software University**

<https://softuni.bg>

1. Defining Classes
  - Constructor and Methods
  - Accessor Properties
2. DOM Classes
  - Review of DOM
  - Methods and Properties
3. Built-in Collections





sli.do

#js-advanced



# Defining Classes

Constructor, Properties, Accessors

# Class Definition

- **Structure** for objects
- Classes define:
  - **Data** (properties, attributes)
  - **Actions** (behavior)
- One class may have **many instances** (objects)
- Unlike functions, class declarations are **not hoisted!**



# Class Body

- The class body contains **method definitions**

```
class Circle {  
    constructor(r) {  
        this.r = r;  
    }  
};
```



- The **constructor** is a special method for **creating** and **initializing** an object created with a class
- Instance **properties** are defined inside the **constructor**

# Class Methods

- A class may have **methods**, which will be available to its **instances**

```
class Rectangle {  
  constructor(height, width) {  
    this.height = height;  
    this.width = width;  
  }  
  // Method  
  calcArea() { return this.height * this.width; }  
}  
const square = new Rectangle(10, 10);  
console.log(square.calcArea()); // 100
```



- **this** refers to the **instance** of the class

```
class Person {  
  constructor(firstName, lastName) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
  }  
  displayName() {  
    console.log(`Name: ${this.firstName} ${this.lastName}`);  
  }  
};  
  
const person = new Person("John", "Doe");  
person.displayName(); // Name: John Doe
```



# Problem: Person

- Write a **class** that represent a personal record
- It needs to have the following properties:
  - **firstName**, **lastName**, **age** and **email**
- And a **toString()** method

```
let person = new Person('Anna', 'Simpson', 22, 'anna@yahoo.com');  
console.log(person.toString());  
// Anna Simpson (age: 22, email: anna@yahoo.com)
```

# Solution: Person

```
class Person {  
    constructor(fName, lName, age, email) {  
        this.firstName = fName;  
        this.lastName = lName;  
        this.age = age;  
        this.email = email;  
    }  
    toString() {  
        return `${this.firstName} ${this.lastName}  
                (age: ${this.age}, email: ${this.email})`  
    }  
}
```

# Instanceof Operator

- The **instanceof** operator returns **true** if the given object is an **instance** of the specified class



```
const circle = new Circle(5);  
  
console.log(circle instanceof Circle); // true  
console.log(circle instanceof Object); // true  
console.log(circle instanceof String); // false  
console.log(circle instanceof Number); // false
```

- The **static** keyword defines a **static method** for a class

```
class MyClass {  
    static staticMethod() { return 'Static call'; }  
}
```

- Static methods are **part of the class** and not of its instances

```
console.log(MyClass.staticMethod())
```

- They can **only** access other static methods via **this** context

```
static anotherStaticMethod() {  
    return this.staticMethod() + ' from another method';  
}
```

# Problem: Point Distance


- Write a **class** representing a **Point** in the plane
  - Properties **x** and **y**, set through the **constructor**
  - **Static** method **distance()**
    - Takes **two parameters** of type **Point**
    - Returns **Euclidian distance** between them

# Solution: Point Distance

```
class Point {  
    constructor(x, y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    static distance(p1, p2) {  
        const dx = p1.x - p2.x;  
        const dy = p1.y - p2.y;  
        return Math.sqrt(dx ** 2 + dy ** 2);  
    }  
}
```

# Accessor Properties

- Accessor properties are **methods** that **mimic values**
  - Keywords **get** and **set** with **matching identifiers**
  - They can be **accessed** and **assigned** to like properties



```
class Circle {  
  constructor(r) { this.radius = r; }  
  
  get area() {  
    return Math.PI * (this.radius ** 2);  
  }  
}
```

```
const circle = new Circle(5);  
console.log(circle.area); // 78.5398...
```

Accessing value  
without brackets

# Accessor Properties Example

Property getter

Property setter

Read-only  
property **area**

```
class Circle {  
  constructor(radius) { this.radius = radius; }  
  get diameter() { return 2 * this.radius; }  
  set diameter(value) {  
    this.radius = value / 2;  
  }  
  get area() {  
    return Math.PI * (this.radius ** 2);  
  }  
}  
  
let c = new Circle(2);  
c.diameter = 1.6;  
console.log(`Radius: ${c.radius}`); // 0.8  
console.log(`Diameter: ${c.diameter}`); // 1.6  
console.log(`Area: ${c.area}`); // 2.0106...
```



- Accessors are often used for **validation**
  - The **setter** can verify that a **given value** meets requirements

```
set diameter(value) {  
  if (value <= 0) {  
    throw new Error('Diameter must be positive');  
  }  
  this.radius = value / 2;  
}
```

- Properties **without** a setter are **read-only** (cannot be assigned)
- Getters can be used for a **validated** or **calculated** property



# DOM Classes

Methods and Attributes

- All DOM objects are **instances** of standard DOM classes
  - Always created via **factory functions**, instead of with **new**

```
const divElement = document.createElement('div');  
console.log(divElement instanceof HTMLDivElement); // true
```

- They provide many useful methods and properties
  - Already seen: **addEventListener**, **appendChild**, **remove**, **children**, **parentNode**, **textContent**, **value**, etc.

- **cloneNode(*deep*)** create a **duplicate** of the selected element
  - If *deep* is true, a **deep-copy** is created

```
const duplicate = divElement.cloneNode(true);
```

- **replaceWith()** replaces selected element with another

```
const span = document.createElement('span');  
divElement.replaceWith(span);
```

- **before()** insert element before selected node
- **after()** insert element after selected node

- **classList** - is a read-only property that returns a collection of the class attributes of specified element

```
<div id="myDiv" class="container div root"></div>
```

```
const element = document.getElementById('myDiv').classList;  
// DOMTokenList(3)  
["container", "div", "root", value: "container div root"]
```

- **classList Methods**

```
<div id="myDiv" class="container div root"></div>
```

- **add()** - Adds the specified class values

```
document.getElementById('myDiv').classList.add('testClass');
```

- **remove()** - Removes the specified class values

```
document.getElementById('myDiv').classList.remove('container');
```

```
<div id="myDiv" class="div root testClass"></div>
```

- **getAttribute()** - returns the value of attributes of specified HTML element

```
<input type="text" name="username"/>  
<input type="password" name="password"/>
```

```
const inputEle = document.getElementsByTagName('input')[0];  
inputEle.getAttribute('type'); // text  
inputEle.getAttribute('name'); // username
```

- **setAttribute()** - sets the value of an attribute on the specified HTML element

```
<input type="text" name="username"/>  
<input type="password"/>
```

```
const inputPassEle = document.getElementsByTagName('input')[1];  
inputPassEle.setAttribute('name', 'password');
```

```
<input type="text" name="username"/>  
<input type="password" name="password"/>
```



- **removeAttribute()** - removes the attribute with the specified name from an HTML element

```
<input type="text" name="username" placeholder="Username..." />  
<input type="password" name="password" placeholder="Password..." />
```

```
const inputPassEle = document.getElementsByTagName('input')[1];  
inputPassEle.removeAttribute('placeholder');
```

```
<input type="text" name="username" placeholder="Username..." />  
<input type="password" name="password" />
```

- **hasAttribute()** - method returns true if the specified attribute exists, otherwise it returns false

```
<input type="text" name="username" placeholder="Username..."/>  
<input type="password" name="password" id="password"/>
```

```
const passwordElement = document.getElementById('password');  
passwordElement.hasAttribute('name'); // true  
passwordElement.hasAttribute('placeholder'); // false
```

- **dataset** obtain **DOMStringMap** of custom **data attributes**

- Classes can be used to **encapsulate** elements and behavior
  - Store **references** to DOM elements
  - Provide **event handlers**
  - **Methods** that **manipulate** the elements
- This is called the **Component Pattern**
  - Used in many **JS frameworks**, such as React, Vue, Angular
  - Used in the **Custom Web Component API**



# **Live Demonstration**

## **Classes Interacting with DOM**



# **Build-in Collections**

## **Set, Map, WeakSet, WeakMap**

# What is a Map?

- A **Map** collection stores its elements in **insertion order**
- A for-of loop returns an array of **[key, value]** for each iteration
- Pure **JavaScript objects** are like **Maps** in that both let you:
  - Assign **values** to **keys**
  - Detect whether something is stored in a key
  - Delete keys



- **.set**(key, value) – adds a new key-value pair

```
let map = new Map();  
map.set(1, "one"); // key - 1, value - one  
map.set(2, "two"); // key - 2, value - two
```

- **.get**(key) – returns the value of the given key

```
map.get(2); // two  
map.get(1); // one
```

- **.size** – **property**, holding the number of stored entries

- **.has(key)** - checks if the map has the given key

```
map.has(2); // true  
map.has(4); // false
```

- **.delete(key)** - removes a key-value pair

```
map.delete(1); // Removes 1 from the map
```

- **.clear()** - removes all key-value pairs



- **.entries()** - returns Iterator - array of **[key, value]**
- **.keys()** - returns Iterator with all the **keys**
- **.values()** - returns Iterator with all the **values**

```
let entries = Array.from(map.entries());  
// [ [1, 'one'], [2, 'two'] ]  
let keys = Array.from(map.keys()); // [1, 2]  
let values = Array.from(map.values()); // ['one', 'two']
```

These methods return an Iterator,  
transform it into an Array

- To print a map simply use one of the **iterators** inside a **for-of**

```
let iterable = phonebookMap.keys();  
for(let key of iterable) {  
  console.log(` ${key} => ${phonebookMap.get(key)} `);  
}
```

```
for(let [key, value] of phonebookMap) {  
  console.log(` ${key} => ${value} `);  
}
```


- To **sort** a Map, first transform it into an **array**
- Then use the **sort()** method

```
let map = new Map();
map.set("one", 1);
map.set("eight", 8);
map.set("two", 2);
let sorted = Array.from(map.entries())
                  .sort((a, b) => a[1] - b[1]);
for (let kvp of sorted) {
  console.log(`${kvp[0]} -> ${kvp[1]}`);
}
```

Sort ascending by value

# What is a Set?

- Store **unique values** of any type, whether **primitive** values or **object** references
- Set objects are **collections** of values



```
let set = new Set([1, 2, 2, 4, 5]);  
// Set(4) { 1, 2, 4, 5 }  
set.add(7)); // Add value  
console.log(set.has(1));  
// Expected output: true
```

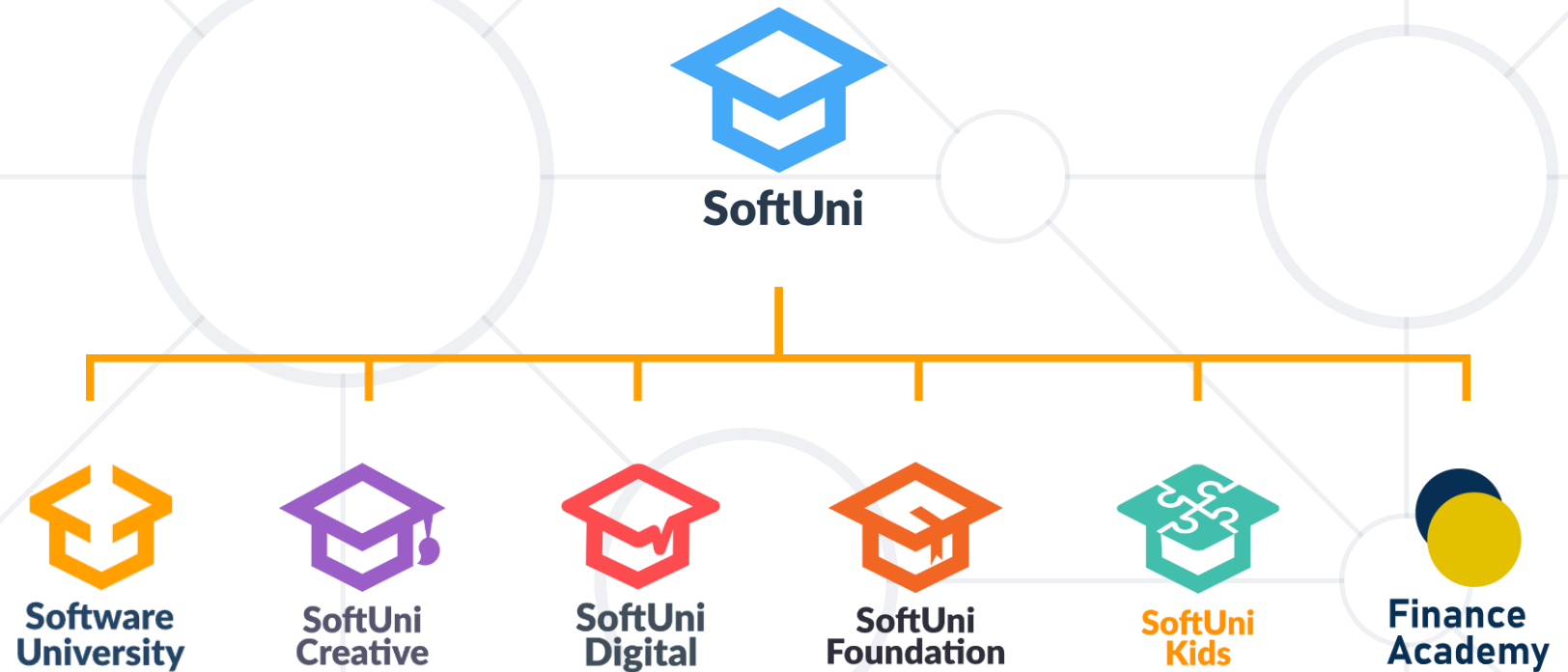
- Can **iterate** through the elements of a set in **insertion** order

- **Special variants** of Map and Set
- Their elements **do not** count as **active** references
  - Reference types visible (in **scope**) in the program stack are **active**
  - Active references **remain in memory**
  - **Out-of-scope** references are **removed** by the **garbage collector**
- These collections are used in **memory-intensive** applications

- Classes - **structure** for objects, that may define
  - Constructors & Parameters
  - Methods & Properties
  - Getters & Setters
- DOM Classes: review and more
- Build-in Collections
  - Map & Set



# Questions?



# SoftUni Diamond Partners

**SUPER  
HOSTING  
.BG**



**Coca-Cola HBC  
Bulgaria**

 **Flutter**<sup>TM</sup>  
International

**INDEAVR**  
Serving the high achievers



**AMBITIONED**

 **DRAFT  
KINGS**



**BOSCH**

 **Postbank**  
*Решения за твоето утре*

 **PHAR  
VISION**



**SmartIT**

**DXC**  
TECHNOLOGY

**createX**



- 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>



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

- [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.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)

