Introduction to OOP

IT TALENTS
Training Camp

Object Oriented Programming

- OOP is a programming paradigm based on the concept of "objects“
- It enable software engineers to write reusable, easy for understanding and maintaining code
- The heart of OOP consist of objects, fields and methods.

What are objects?

- Software objects are used to model the real-world and abstract objects that you find in everyday life
- Real-world objects share two characteristics: They all have state and behavior
- Each person has name, age, personal number... (state)
- Each person can eat, sleep, walk... (behavior)
- Mobile phone – Have memory, has color, is switched on or of. Can ring, can send SMS, can be switched of

Object person - example

- Name: Ivan
- Years: 25
- Gender: male
- Weight: 80.5 kg
- Can walk
- Can speak
- Can eat

- Name: Maria
- Years: 35
- Gender: female
- Weight: 55 kg
- Can walk
- Can speak
- Can eat

Creating a simple object

- Each object is separated with {} literal. The {} means „empty object“.
- The properties are called ‘**fields**’. And the functions are called ‘**methods**’.
- Each property is created using the **key : value** syntax

```
const player = {
    // fields:
    name: 'Hristomir’,
    points: 341,
    level: 10,
    isSuperPlayer: true,
    friends: ['Misho', 'Mimi’],
    // methods:
    shoot: function() {
        console.log("Will shoot to " + this.friends[0]);
    }
};
```

Accessing Object Properties

- You can access object properties in two ways:
  - Using the . Symbol

```
objectName.propertyName
```

  - Using square brackets

```
objectName["propertyName"]
```

  - The square brackets syntax is commonly used when the properties cannot be accessed with .
    - Examples: objectName[5], objectName['hello world']

Removing Object Properties

- You can remove properties on objects using the „**delete**“ keyword.

```
let mimi = {
        name : "Mimi",
        egn : 8303121232
}
console.log(mimi.egn); // 8303121232
delete mimi.egn;
console.log(mimi.egn); // undefined
```

- The delete operator deletes both the value of the property and the property itself.
- The delete operator is designed to be used on object properties. It has no effect on variables or functions.
- **Note:** The delete operator should not be used on predefined JavaScript object properties. It can crash your application.

Common Object methods

- Object.keys() -> Returns an array containing the names of all of the given object's own enumerable string properties.
- Object.values() -> Returns an array containing the values that correspond to all of a given object's own enumerable string properties.
- Object.entries() -> Returns an array containing all of the [key, value] pairs of a given object's own enumerable string properties.
- Object.freeze() -> Freezes an object. Other code cannot delete or change its properties.

Iterating over object properties

- Using the **for.. in** statement. It loops through the properties of an object:

```
let mimi = {
        name : "Mimi",
        egn : 8303121232
}
for(let key in mimi) {
        console.log(`key: ${key} | value: ${mimi[key]}`)
}
```

- Using Object.keys() and then iterate with forEach or just a simple for loop

Checking for object properties

- You can verify for property existence using „in“. Same used for iterating all properties of object.

```
let mimi = {
        name : "Mimi",
        egn : 8303121232
}
console.log(mimi.egn);
delete mimi.egn;

console.log(mimi.egn); // undefined
console.log('name' in mimi); // true
console.log('egn' in mimi); // false
```

Property descriptors

- The property descriptor simply holds the configuration of the property. Each Object property can be:
  - Configurable -> Whether the property can be deleted from the corresponding object.
  - Enumerable -> Whether the property is going to be visible during “enumeration”. Example for … in loop.
  - Writable -> Whether the property can be changed in the future

Defining properties in ES 5

```
let hristo = {};
Object.defineProperty(hristo, "name", {
        writable: false,
        configurable: false,
        enumerable: true,
        value: 'Hristomir'
});
console.log(hristo.name); // Hristomir
hristo.name = 'Mristo';
console.log(hristo.name); // still Hristomir
```

The “this” keyword

- Functions inside given object has reference called „this“ that points to current object, the function is called on. Thus you can refer properties of the object inside the function.

```
const person1 = {
	name: "Hristomir",
	whoAmI: function() {
		console.log("Hi! I'm " + this.name + " nice to meet you.")
	}
};
person1.whoAmI(); // I'm Hristomir nice to meet you.
person1.name = 'Jestislav';
person1.whoAmI(); // I'm Jestislav nice to meet you.
```

More on “this”

- It has different values depending on where it is used:
  - In a method, **this** refers to the owner object.
  - Alone, **this** refers to the global object.
  - In a function, **this** refers to the global object.
  - In a function, in strict mode, **this** is undefined.
  - In an event, **this** refers to the element that received the event.
  - Methods like call(), and apply() can refer **this** to any object.

Call and Apply

- You can „borrow“ a function from other object and use as it`s yours. You can use call or apply to do this.
- First parameter is the object (or the context) on which to apply the function.
- Next, you supply the arguments of the function separated with “,“ (using call) or as an array (using apply)

Demo – call() vs apply()

IT TALENTS
Training Camp

Constructor Pattern

IT TALENTS
Training Camp

The Problem

- Imagine a scenario where you have to create 50 objects for dogs. Each dog should have the following fields:
  - Name
  - Age
  - Walk: function() {…}
  - Eat: function() {…}
  - Drink: function() {…}

You would have to write 50 times the logic for all of the methods. What if we want to change the behaviour?

Constructor Pattern

- The constructors act as the template for building objects.
- They can define the properties and the methods for the objects they build.
- They give us the **blueprint** for creating many objects of the same “type”.
- The constructors are just a normal function which make use of this keyword

Constructor Pattern

- The constructor is responsible for creating an object.
- Constructors don't return – they always return the newly created object.
- In fact, they are just a normal function modifying „this“ reference.
- Constructors should have a body
- Constructor functions should start with uppercase letter!
- Objects created with constructor functions should use the **new** keyword

Constructor Pattern

```
function Person(name, age) {
	this.name = name;
	this.age = age;
}
const tisho = new Person('Tisho', 30);
const misho = new Person('Misho', 28);
console.log(tisho.name + " " + tisho.age);
console.log(misho.name + " " + misho.age);
```

Prototype

IT TALENTS
Training Camp

Prototypes

- Prototypes are the mechanism by which JavaScript objects inherit features from one another.
- Prototype allows to add methods and fields to constructor before you create an object. So, your new objects will have this built-in property.
- It is really useful for functions, in order not to duplicate code in memory.

Example

```
function Person(name, age, weight) {
    this.name = name;
    this.age = age;
    this.weight = weight;
}

Person.prototype.sayHi = function() {
    console.log("Hi, I'm " + this.name);
}

Person.prototype.looseWeight = function(howMuch) {
    this.weight -= howMuch;
}

const pesho = new Person('Pesho', 25, 75);
pesho.looseWeight(10);
console.log(pesho.weight); // 65
pesho.sayHi(); // Hi, I'm Pesho
```

Summary

- What is OOP?
- How do we create objects?
- Access/Modify/Delete object fields
- Most used Object methods
- How to iterate object fields?
- What are property descriptors?
  - How can we define property that cannot be modified/deleted/enumerated?
- The scary **“this”** keyword
- Call vs apply
- Constructor Pattern
- Prototypes

IT TALENTS
Training Camp