Class 25 - OOPS 2 - classes and constructors

title: Constructor Functions

1. What are Constructor Functions and Why Use Them?

1. Constructor functions in JavaScript are a way to create multiple objects with the same structure, methods, and properties. They act like a blueprint for creating instances of a particular type of object.
2. In JavaScript, constructor functions are special functions that help create and initialize objects based on a common template. They're like a blueprint for building multiple objects that share similar properties and behaviors.
3. In JavaScript, a constructor function is similar to a pizza recipe. It defines the essential components of an object (like dough, sauce, and cheese are for pizza), and it allows for customization through parameters that represent different toppings.

The Blueprint (Constructor Function)

The recipe for making a pizza acts as your "Pizza" constructor function. It specifies:

Toppings: cheese, pepperoni, vegetables, etc.
Size: small, medium, large.

Crust Type: thin, thick, stuffed.

```
function Pizza(toppings, size, crustType) {
 this.toppings = toppings;
 this.size = size;
 this.crustType = crustType;

 this.describe = function () {
   console.log(
     `A ${this.size} pizza with ${this.toppings.join(", ")} on a ${
       this.crustType
     } crust.`
   );
 };
}

const customerOrder1 = new Pizza(["cheese", "pepperoni"], "medium",
"thin");

const customerOrder2 = new Pizza(["veggies", "pepperoni"], "small",
"thick");

customerOrder1.describe(); // Output: A medium pizza with cheese,
pepperoni on a thin crust.

customerOrder2.describe(); // Output: A small pizza with veggies,
pepperoni on a thick crust.
```

Step by Step Breakdown of the Code -

This piece of code defines a constructor function named Pizza and then uses it to create an instance representing a customer's pizza order.

1. Define the Pizza Constructor Function:
    a. The Pizza function is designed to create pizza objects. It takes three parameters: toppings, size, and crustType.
    b. Inside the function, this refers to the new object that will be created when the function is called with new. **The new keyword will make the this keyword point to newly created object**
    c. this.toppings = toppings; assigns the array of toppings passed as an argument to the toppings property of the new object.
    d. this.size = size; assigns the size argument (a string) to the size property of the new object.
    e. this.crustType = crustType; assigns the crust type argument (a string) to the crustType property of the new object.
    f. this.describe = function() {...}; defines a method named describe on the new object. This method, when called, will log a string describing the pizza to the console, using the object's size, toppings, and crustType properties.
2. Method describe Inside the Pizza Constructor:

    a. This method constructs and logs a string to the console that describes the pizza. It accesses the pizza's properties (this.size, this.toppings, and this.crustType) to create a descriptive string.
    b. this.toppings.join(", ") combines all elements in the toppings array into a single string, separated by commas. This makes the array of toppings into a readable list.
3. Create an Instance of Pizza:
    a. var customerOrder1 = new Pizza(['cheese', 'pepperoni'], 'medium', 'thin'); creates a new Pizza object with the specified toppings ('cheese', 'pepperoni'), size ('medium'), and crust type ('thin').
    b. Now with using the same constructor function you can create another object
    c. var customerOrder2 = new Pizza(['veggies', 'pepperoni'], 'small', 'thick'); creates a new Pizza object with the specified toppings ('veggies', 'pepperoni'), size ('small'), and crust type ('thick').
    d. The new keyword is used to call the Pizza function, creating a new instance of a pizza object, which is then assigned to the variable customerOrder1.
4. Call the describe Method of customerOrder1 and customerOrder2 :
    a. You can create methods inside constructor functions and call them with different Objects

b. customerOrder1.describe(); calls the describe method on the customerOrder1 object. This results in logging the description of the pizza to the console
c. customerOrder2.describe(); calls the describe method on the customerOrder2 object. This results in logging the description of the pizza to the console

Basically this is how you can work with a Constructor function

title: Classes in JS

1. Classes in ES6 (ECMAScript 2015) introduced a new syntax for creating objects and dealing with inheritance in JavaScript. This new syntax is more readable and concise.

1. Syntax and Structure of Classes in ES6

```
class MyClass {
 constructor() {
   // Initialization of properties
 }

 myMethod() {
   // Method implementation
 }
}
```

1. class keyword is used to declare a class.
2. MyClass is the name of the class.

3. The constructor is a special method for creating and initializing objects created with the class. There can only be one constructor method in a class.
4. myMethod() is an example of a class method that can be added to the class.

2. Class Constructor, Properties, and Methods

1. Constructor: The constructor method is called automatically when a new instance of a class is created. It usually initializes class properties.
2. Properties: Properties are essentially variables that belong to an object. properties are initialized inside the constructor method.
3. Methods: Methods are functions that belong to the class. They can be used to define behavior for class instances or to manipulate class data.

3. Static Methods and Properties

1. Static methods and properties are associated with the class itself, not instances of the class. This means you can call a static method without creating an instance of the class.
2. In other words, static methods are functions defined on a class that are not available on instances of the class but rather on the class itself. You can think of them as utility functions that belong

to the class and not to any individual object created from the class.

```
class MyClass {
 static myStaticMethod() {
   console.log("This is a static method.");
 }
}

MyClass.myStaticMethod(); // This is a static method.
```

3. Key characteristics
    a. Not Tied to Instances:
        i. It’s accessible directly from the class.
    b. Usage:
        i. Static methods are used when you need a function that is relevant to all instances of a class or when the method does not require data from any specific instance.
        ii. They are often used to implement utility functions that operate on input arguments and return a result without altering or requiring data from instances.

4.Comparing Classes to Constructor Functions

1. Classes in ES6 offer a more straightforward syntax for creating objects and handling inheritance. They essentially provide syntactic sugar over JavaScript's existing prototype-based

inheritance and do not introduce a new object-oriented inheritance model.

2. Constructor Functions were the traditional way to create objects and implement inheritance in ES5 and earlier versions.

Rewriting Pizza constructor function

```
class Pizza {
 constructor(toppings, size, crustType) {
   this.toppings = toppings;
   this.size = size;
   this.crustType = crustType;
 }

 describe() {
   console.log(
     `A ${this.size} pizza with ${this.toppings.join(", ")} on a ${
       this.crustType
     } crust.`
   );
 }
}

var customerOrder1 = new Pizza(["cheese", "pepperoni"], "medium",
"thin");
var customerOrder2 = new Pizza(["veggies", "pepperoni"], "small",
"thick");

customerOrder1.describe(); // Output: A medium pizza with cheese,
pepperoni on a thin crust.
customerOrder2.describe(); // Output should be: A small pizza with
veggies, pepperoni on a thick crust.
```

title: Inheritance JS

1. Extending the example of the Pizza class, let's demonstrate classical inheritance in ES6 using the extends and super keywords.
2. Inheritance allows a class to inherit properties and methods from another class.
3. The extends keyword is used in class declarations or class expressions to create a class as a child of another class.
4. The super keyword is used to access and call functions on an object's parent.

Extending the Pizza Class

1. Suppose we want to create a specialized type of pizza, StuffedCrustPizza, which extends the basic Pizza class. This new class has an additional property for the type of stuffing and a method to describe the stuffing.

```
class StuffedCrustPizza extends Pizza {
  constructor(toppings, size, crustType, stuffingType) {
    super(toppings, size, crustType); // Call the parent class constructor with super
    this.stuffingType = stuffingType; // New property specific to StuffedCrustPizza
  }
   describeStuffing() {
    console.log(`This pizza has ${this.stuffingType} stuffing in the crust.`);
```

```
  }
   // Overriding the describe method
  describe() {
    super.describe(); // Call the describe method from the parent class
    this.describeStuffing(); // Additional description for the stuffing
  }
}
```

2. extends is used to define a class as a child of another class.
3. super (within the constructor) calls the parent class' constructor with the parameters required by the parent class. It's necessary to call super() before using this in a constructor, as it will initialize the parent's properties in the child class.
4. // extra - Without this step, the this reference in the subclass wouldn't be able to access properties or methods defined in the parent class because its prototype chain would not be properly established.
5. super is also used to call parent class methods. In the overridden describe method, super.describe() is used to include the parent's description logic.

```
class Pizza {
 constructor(toppings, size, crustType) {
   this.toppings = toppings;
   this.size = size;
   this.crustType = crustType;
 }

 describe() {
   console.log(
     `A ${this.size} pizza with ${this.toppings.join(", ")} on a ${
       this.crustType
```

```
    } crust.`
  );
 }
}

class StuffedCrustPizza extends Pizza {
  constructor(toppings, size, crustType, stuffingType) {
    super(toppings, size, crustType); // Call the parent class
constructor with super
    this.stuffingType = stuffingType; // New property specific to
StuffedCrustPizza
  }
   describeStuffing() {
    console.log(`This pizza has ${this.stuffingType} stuffing in the
crust.`);
  }
   // Overriding the describe method
  describe() {
    super.describe(); // Call the describe method from the parent
class
    this.describeStuffing(); // Additional description for the
stuffing
  }
 }

 const specialOrder = new StuffedCrustPizza(['cheese', 'mushrooms'],
'large', 'thick', 'cheese and tikki');

 specialOrder.describe();
 // Expected output:
 // A large pizza with cheese, mushrooms on a thick crust.
 // This pizza has cheese and garlic stuffing in the crust.
```

6. Explanation
    a. We create a new class StuffedCrustPizza that extends the Pizza class.
    b. The constructor of StuffedCrustPizza adds an additional parameter for stuffingType. It uses super to pass the toppings, size, and crustType to the parent class constructor.
    c. We added a new method describeStuffing to describe the unique feature of the stuffed crust pizza.
    d. We override the describe method to include the stuffed crust description. The method first calls the parent's describe method using super.describe(), then calls this.describeStuffing() to add the specific description of the stuffing.

Fun Fact - Classical Inheritance doesn't even exists in JS , Everything is a paradigm of Prototypal Inheritance which we will learn in the next class

Below is the combined code that includes the original Pizza class and the new StuffedCrustPizza class that extends Pizza to demonstrate classical inheritance in ES6.

```
// Define the base Pizza class
```

```
class Pizza {
 constructor(toppings, size, crustType) {
   this.toppings = toppings;
   this.size = size;
   this.crustType = crustType;
 }

 describe() {
   console.log(
     `A ${this.size} pizza with ${this.toppings.join(", ")} on a ${
       this.crustType
     } crust.`
   );
 }
}

// Define the StuffedCrustPizza class that extends Pizza
class StuffedCrustPizza extends Pizza {
 constructor(toppings, size, crustType, stuffingType) {
   super(toppings, size, crustType); // Call the parent class constructor with super
   this.stuffingType = stuffingType; // New property specific to StuffedCrustPizza
 }

 describeStuffing() {
   console.log(`This pizza has ${this.stuffingType} stuffing in the crust.`);
 }

 // Overriding the describe method
 describe() {
   super.describe(); // Call the describe method from the parent class
   this.describeStuffing(); // Additional description for the stuffing
 }
}

// Creating instances and calling methods to demonstrate functionality
const customerOrder1 = new Pizza(["cheese", "pepperoni"], "medium", "thin");
customerOrder1.describe(); // Output: A medium pizza with cheese, pepperoni on a thin
crust.

const customerOrder2 = new Pizza(["veggies", "pepperoni"], "small", "thick");
customerOrder2.describe(); // Output: A small pizza with veggies, pepperoni on a thick
crust.
```

```
const specialOrder = new StuffedCrustPizza(
 ["cheese", "mushrooms"],
 "large",
 "thick",
 "cheese and garlic"
);
specialOrder.describe();
// Expected output:
// A large pizza with cheese, mushrooms on a thick crust.
// This pizza has cheese and garlic stuffing in the crust.
```

title: Static methods

1. Let's further extend our example with a static method, let's introduce a static method in the Pizza class.
2. A static method is a function that is associated with the class, not with instances of the class.
3.
4. This means you can call a static method without creating an instance of the class. Static methods are often used for utility functions that do not require an instance of the class to work.

Adding a Static Method

Let's add a static method calculateTotalPizzasMade that keeps track of the total number of pizzas made. To accomplish this, we'll also need to add a static property to keep count of every time a Pizza or StuffedCrustPizza is instantiated.

```
// Define the base Pizza class
class Pizza {
 static totalPizzasMade = 0; // Static property to keep count

 constructor(toppings, size, crustType) {
   this.toppings = toppings;
   this.size = size;
   this.crustType = crustType;
   Pizza.totalPizzasMade++; // Increment the count each time a new pizza is made
 }

 describe() {
   console.log(
     `A ${this.size} pizza with ${this.toppings.join(", ")} on a ${
       this.crustType
     } crust.`
   );
 }
 // Static method
 static calculateTotalPizzasMade() {
   console.log(`Total pizzas made: ${Pizza.totalPizzasMade}`);
 }
}

// Define the StuffedCrustPizza class that extends Pizza
class StuffedCrustPizza extends Pizza {
 constructor(toppings, size, crustType, stuffingType) {
   super(toppings, size, crustType); // Call the parent class constructor with super
   this.stuffingType = stuffingType; // New property specific to StuffedCrustPizza
 }

 describeStuffing() {
   console.log(`This pizza has ${this.stuffingType} stuffing in the crust.`);
 }

 // Overriding the describe method
 describe() {
   super.describe(); // Call the describe method from the parent class
   this.describeStuffing(); // Additional description for the stuffing
 }
}
```

```
// Creating instances and calling methods to demonstrate functionality
const customerOrder1 = new Pizza(["cheese", "pepperoni"], "medium", "thin");
customerOrder1.describe(); // Output: A medium pizza with cheese, pepperoni on a thin
crust.

const customerOrder2 = new Pizza(["veggies", "pepperoni"], "small", "thick");
customerOrder2.describe(); // Output: A small pizza with veggies, pepperoni on a thick
crust.

const specialOrder = new StuffedCrustPizza(
 ["cheese", "mushrooms"],
 "large",
 "thick",
 "cheese and garlic"
);
specialOrder.describe();
// Expected output:
// A large pizza with cheese, mushrooms on a thick crust.
// This pizza has cheese and garlic stuffing in the crust.
Pizza.calculateTotalPizzasMade(); // Output: Total pizzas made: 3
```

Explanation of Static Methods

1. Static Methods belong to the class rather than any particular object instance. They can be called directly on the class itself.
2. Use Cases: Static methods are useful for utility functions, such as factory methods (methods that create instances of the class), or for operations that relate to the class as a whole rather than to individual instances.

3. Accessing Static Properties: Within static methods, you can access static properties using the class name (e.g., Pizza.totalPizzasMade).
4. In our example, calculateTotalPizzasMade is a static method that outputs the total number of pizzas made. This demonstrates how static properties and methods can be used to maintain and access data that is relevant to the class as a whole, rather than to individual instances.

How would you create private properties for a class in JavaScript?

1. Private properties ensure that class data is encapsulated and not directly accessible from outside the class. This is a fundamental principle of object-oriented programming, improving security and data integrity.
2. In JavaScript ES6 and later versions, you can create private class properties by prefixing the property name with a # symbol.
3. Private properties are accessible only within the class that defines them, making them useful for encapsulating class-specific data and behaviors that shouldn't be directly accessible from outside the class.

4. Let's modify our Pizza and StuffedCrustPizza classes to include private properties. For illustration, we'll make the toppings, size, and crustType properties of Pizza private, and add a private property to StuffedCrustPizza as well.

```
// Define the base Pizza class
class Pizza {
 static totalPizzasMade = 0; // Static property to keep count
 #toppings; // Private property
 #size;     // Private property
 #crustType; // Private property

 constructor(toppings, size, crustType) {
   this.#toppings = toppings;
   this.#size = size;
   this.#crustType = crustType;
   Pizza.totalPizzasMade++; // Increment the count each time a new pizza is
made
 }

 describe() {
   console.log(`A ${this.#size} pizza with ${this.#toppings.join(", ")} on a
${this.#crustType} crust.`);
 }
 // Static method
 static calculateTotalPizzasMade() {
   console.log(`Total pizzas made: ${Pizza.totalPizzasMade}`);
 }
}

// Define the StuffedCrustPizza class that extends Pizza
class StuffedCrustPizza extends Pizza {
   #stuffingType; // Private property

 constructor(toppings, size, crustType, stuffingType) {
   super(toppings, size, crustType); // Call the parent class constructor with
super
   this.#stuffingType = stuffingType;
```

```
 }

 describeStuffing() {
   console.log(`This pizza has ${this.#stuffingType} stuffing in the crust.`);
 }

 // Overriding the describe method
 describe() {
   super.describe(); // Call the describe method from the parent class
   this.describeStuffing(); // Additional description for the stuffing
 }
}

// Creating instances and calling methods to demonstrate functionality
const customerOrder1 = new Pizza(["cheese", "pepperoni"], "medium", "thin");
customerOrder1.describe(); // Output: A medium pizza with cheese, pepperoni on
a thin crust.

const customerOrder2 = new Pizza(["veggies", "pepperoni"], "small", "thick");
customerOrder2.describe(); // Output: A small pizza with veggies, pepperoni on
a thick crust.

const specialOrder = new StuffedCrustPizza(
 ["cheese", "mushrooms"],
 "large",
 "thick",
 "cheese and garlic"
);
specialOrder.describe();
// Expected output:
// A large pizza with cheese, mushrooms on a thick crust.
// This pizza has cheese and garlic stuffing in the crust.
Pizza.calculateTotalPizzasMade(); // Output: Total pizzas made: 3
// Attempting to access a private property from outside the class
console.log(specialOrder.toppings); // Undefined, as toppings is private
// console.log(specialOrder.#toppings); // Syntax error: Private field
'#toppings' must be declared in an enclosing class
```