# Object Oriented Programming, Introduction to Classes

*Software Development Bootcamp*

*Topic*

# Introduction to OOP and Classes

# What Is Object-Oriented Programming?

**Object-Oriented Programming (OOP)** is a programming paradigm based on the concept of "objects", which can contain data and code. Multiple object-oriented languages share common design principles.

- JavaScript is multi-paradigm: object-oriented, functional, and procedural

# Why Use Object Oriented Programming?

- **Organization**: Keep related things together
- **Reuse**: Create Reusable  code structures
- **Clarity**: Make code easier to understand and maintain

# Classes

- Classes are “blueprints” for creating objects
- They define the structure and behavior of objects
- In JavaScript we use the `class` keyword to define a class

# Class: Pizza

- The `Pizza` class definition serves as a blueprint for creating pizza objects
- The `constructor` method initializes two properties: `size` and `toppings`. These represent the characteristics of a pizza

```
// Creating the 'Pizza' class
class Pizza {
 constructor(size, toppings) {
   this.size = size;
   this.toppings = toppings;
 }
 describe() {
   return `This is a ${this.size}
pizza with ${this.toppings.join(",
")}.`;
 }
}
```

# Why Use Classes

- Organization: Classes help keep related things together. All the properties and methods of a `Pizza` are in one place.
- Reuse: Once we have a class, we can make as many pizzas as we want without repeating the same code
- Clarity: It makes our code easier to understand. We can see that `Pizza` is a class for pizzas, and it's clear what properties and methods a pizza has.

# Instantiating Classes

- Use the **`new`** keyword to create instances of a class
- Each instance is a unique object based on the class blueprint

```
// instantiates (creates) a new pizza called pizza1
const pizza1 = new Pizza("large", ["pepperoni", "mushrooms"]);
// instantiates (creates) a new pizza called pizza2
const pizza2 = new Pizza("medium", ["sausage", "peppers"]);
console.log(pizza1.describe()); // This is a large pizza with pepperoni, mushrooms.
console.log(pizza2.describe()); // This is a medium pizza with sausage, peppers.
```

# Factory Methods

A Factory Method is a special type of function inside a class that creates objects of that class

- Uses a method to deal with the problem of creating objects
- Useful when object creation logic is complex, or when you want to centralize object creation

# Factory Method Example

- The `static` keyword is used to define a static method (factory method) on the class

```
class Car {
    constructor(make, model, year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }
    drive() {
        console.log(`${this.make}
${this.model} is driving!`);
    }
    // Factory Method
    static createCar(make, model, year) {
        return new Car(make, model, year);
    }
}
const car1 = Car.createCar('Canyonero',
'F-Series', 1999);
car1.drive();
// Canyonero F-Series is driving!
```

# `extends`, `constructor`, And `super` Keywords

- `extends`: Used to create a class that is a child of another class
- `constructor`: The `constructor` method is called automatically when a new object is created
- `super`: Used to call functions on an object's parent. Commonly used in subclasses to call the `constructor` of the parent class

# extends, constructor, super

- extends
  - Indicates that `Dog` is a subclass (child) of `Animal`
  - `Dog` Inherits properties and methods from `Animal`
- constructor
  - `Animal` constructor sets the `name` property
  - `Dog` constructor sets the `name` and `breed` properties
- super
  - Used in `Dog` constructor as `super(name)`
  - Calls the constructor of the parent class
  - Must be called before using `this` in the constructor

```
class Animal {
 constructor(name) {
   this.name = name; // Constructor
initializes the 'name' property
 }
 speak() {
   return `${this.name} makes a noise.`;
 }
}
class Dog extends Animal {
 constructor(name, breed) {
   super(name); // Call the parent
constructor
   this.breed = breed; // Initialize an
additional property
 }
 speak() {
   return `${this.name} barks.`;
 }
}
let dog = new Dog("Tulip", "Terrier");
console.log(dog.speak()); // Tulip barks.
console.log(dog.breed); // Terrier
```

*Topic*

# OOP Core Concepts

# Object-Oriented Principles

Core Principles:

- **Encapsulation**: Keep data within objects, expose through methods.
- **Inheritance**: Share common behavior using parent and child classes.
- **Abstraction**: Hide complexity by exposing simple interfaces.
- **Polymorphism**: Objects respond to messages based on names and argument types.

# Encapsulation

- Keep data within an object as properties
- Use methods to access and manipulate that data

```
const fido = {
 name: "Fido",
 color: "brown",
 describe() {
   return `Hello! My name is ${this.name}
and I am ${this.color}.`;
 },
};
// RECOMMENDED
console.log(fido.describe());
// DISCOURAGED
console.log(`Hello! My name is ${fido.name}
and I am ${fido.color}`);
```

# Inheritance

- Allows one class to take on properties and methods of another class
- Enables code reuse and establishes a hierarchy between classes

```
// Parent class
class Animal {
 constructor(name) {
   this.name = name;
 }
 speak() {
   return `${this.name} makes a noise.`;
 }
}
// Child class
class Dog extends Animal {
 speak() {
   return `${this.name} barks.`;
 }
}
// Creating an instance of the Dog class
let dog = new Dog('Tulip')
```

# Abstraction

- Allows us to use objects, functions, and classes without worrying about internal implementation
- Hides complexity and exposes only necessary parts of an object

```
class Rectangle {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
    // Area method hides complexity
    area() {
        return this.height * this.width;
    }
}
const shape = new Rectangle(10, 8);
console.log(`The shape's area is:
${shape.area()}`);
```

# Polymorphism

- Allows objects of different classes to be treated as objects of a common superclass.
- Enables methods to do different things based on the object they are called on.

```
class Animal {
 makeSound() {
   console.log("Some generic animal
sound");
 }
}
class Dog extends Animal {
 makeSound() {
   console.log("Woof! Woof!");
 }
}
class Cat extends Animal {
 makeSound() {
   console.log("Meow! Meow!");
 }
}
const myDog = new Dog();
const myCat = new Cat();
myDog.makeSound(); // Output: Woof! Woof!
myCat.makeSound(); // Output: Meow! Meow!
```

*Exercise*

# Walkway Objects

# Linguistic Metaphor for Objects

Naming Conventions:

- **Objects**: Nouns (e.g., Car, Dog).
- **Methods**: Verbs (e.g., drive, bark).
- **Properties**: Adjectives (e.g., color, size).
- **Classes**: Categories (e.g., Vehicle, Animal).