Ultimate OOP in JavaScript

This is your **complete guide** to learn Object-Oriented Programming (OOP) in JavaScript (JS). We use simple words, fun examples like toys, pets, and snacks, and lots of practice. It starts easy and gets a little harder to help you grow. Read slowly, try every code, and have fun!

What is OOP?

OOP is a way to write code using "**things**" called **objects**.

- Objects have **stuff** (like name, colour, or size).
- Objects can **do things** (like bark, move, or eat).
- Example: A toy car is an object. It has a colour (red) and can roll or honk.
- OOP makes code neat, like organizing toys in boxes. It's easier to understand, fix, or add new things later.

Try This:

1. Think of a favourite thing (like a phone, pet, or snack).
2. Write 2 things it has (like name, colour).
3. Write 2 things it does (like ring, jump).
4. Share with a friend or teacher. Write it on paper or in a notebook.

Objects

An **object** is like a toy, pet, or snack. It holds stuff (**information**) and can-do things (**actions**). In JS, we use curly braces {} to make objects.

Example 1: Pet Object (Simple)

```
let pet = {
    name: "Fluffy",
    colour: "White",
    bark: function() {
        console.log("Woof Woof!");
    }
};

console.log(pet.name);
console.log(pet.colour);
pet.bark();
```

Example 2: Snack Object

```
let snack = {
  name: "Jalebi",
  taste: "Sweet",
  size: "big",
  eat: function() {
    console.log("Yum! " + this.name + " is " + this.taste + "!");
  }
};

console.log(snack.name);
snack.eat();

```

Example 3: Toy Object

```
let toy = {
  name: "Monkey",
  type: "Animal",
  isSoft: true,
  hug: function() {
    console.log(this.name + " loves banana!");
  },
  talk: function(words) {
    console.log(this.name + " says: " + words);
  }
};

console.log(toy.type);
toy.hug();
toy.talk("Hi!");

```

What's Happening?

- An object is like a **box holding stuff** (name, colour) and **actions** (bark, eat).
- Use this to talk about the object's own stuff (like **this.name** means the object's name).
- Actions can take inputs (like talk(words) uses words to say something).
- Objects let you group related stuff and actions, like keeping a toy's parts together.

Why Use Objects?

Objects make code organized. You can store lots of stuff and actions in one place, like a backpack for school.

Try This

1. Open your browser and press F12 to see the console.
2. Copy and run the pet code. Check what it shows.
3. Copy and run the snack code. Try changing taste to "Salty".
4. Copy and run the toy code. Call talk with your own words (like toy.talk("Hello!")).
5. Make your own object called phone with:
    - Stuff: name (like "MyPhone"), colour, battery (like 100).
    - Do: ring (shows "Ring Ring!"), text (takes a message and shows "[name] sends: [message]").
6. Run it and check the console.
7. Make another object for a book with title, pages, and actions read (shows "Reading [title]!") and flip (shows "Flipping page!").
8. Show your teacher or friend your code.

Classes

A **class** is like a plan or blueprint. It tells JS how to make many objects that are similar. Think of a mold that makes many toy cars.

Example 1: Car Class (Simple)

```
class Car {
  constructor(name) {
    this.name = name;
  }

  move() {
    console.log(this.name + " goes fast!");
  }
}

let car1 = new Car("Zoom");
let car2 = new Car("Speedy");

car1.move();
car2.move();

```

Example 2: Kid Class (More Stuff)

```
class Kid {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  play() {
    console.log(this.name + " is playing!");
  }

  sleep() {
    console.log(this.name + " is sleeping!");
  }
}

let kid1 = new Kid("Anna", 5);
kid1.play();
kid1.sleep();
console.log(kid1.age);

```

Example 3: Pet Class

```
class Pet {
  constructor(name, colour, energy) {
    this.name = name;
    this.colour = colour;
    this.energy = energy;
  }

  run() {
    if (this.energy > 0) {
      console.log(this.name + " runs happily!");
      this.energy = this.energy - 10;
    } else {
      console.log(this.name + " is too tired!");
    }
  }

  rest() {
    this.energy = this.energy + 20;
    console.log(this.name + " rests. Energy: " + this.energy);
  }
}

let pet1 = new Pet("Max", "Brown", 50);
pet1.run();
pet1.run();
pet1.rest();

```

What's Happening?

- class makes a plan (like a recipe for cookies).
- constructor adds stuff (name, colour, energy).
- Actions like move or run use the object's stuff with **this**.
- new makes a new object from the class (like baking a new cookie).
- Complex classes can track things (like energy) and make decisions (like checking if energy > 0).

Why Use Classes?

Classes let you make many objects with the same plan. It saves time and keeps code organized.

Try This

1. Press F12 to open the browser console.
2. Copy and run the Car code. Try making a new car (like new Car("Racer")).
3. Copy and run the Kid code. Add a new action eat (shows "[name] eats lunch!").
4. Copy and run the Pet code. Call run three times, then rest. Check the energy.
5. Make a Toy class with:
    - Stuff: name, type (like "Doll"), batteries (starts at 3).
    - Do: play (shows "[name] plays!" and lowers batteries by 1), charge (adds 2 batteries).
6. Make two Toy objects. Call play twice, then charge.
7. Add a check in play: if batteries is 0, show "No batteries!" instead of playing.
8. Run it and show your teacher.

Inheritance

Inheritance means a class can use stuff and actions from another class. It's like a kid getting toys and clothes from a parent.

Example 1: Pet and Dog

```
class Pet {
  constructor(name) {
    this.name = name;
  }

  eat() {
    console.log(this.name + " eats food.");
  }
}

class Dog extends Pet {
  bark() {
    console.log(this.name + " says Woof!");
  }
}

let puppy = new Dog("Sammy");
puppy.eat();
puppy.bark();
```

Example 2: Pet and Cat (More Actions)

```
class Pet {
    constructor(name) {
        this.name = name;
    }

    eat() {
        console.log(this.name + " eats food.");
    }
}

class Dog extends Pet {
    bark() {
        console.log(this.name + " says Woof!");
    }
}

let puppy = new Dog("Sammy");
puppy.eat();
puppy.bark();

class Cat extends Pet {
    meow() {
        console.log(this.name + " says Meow!");
    }

    jump() {
        console.log(this.name + " jumps high!");
    }
}

let billi = new Cat("Snowy");
billi.eat();
billi.meow();
billi.jump();
```

Example 3: Pet and Bird

```
class Pet {
  constructor(name) {
    this.name = name;
  }

  eat() {
    console.log(this.name + " eats food.");
  }
}


class Bird extends Pet {
  constructor(name, wingColour) {
    super(name);
    this.wingColour = wingColour;
  }

  fly() {
    console.log(this.name + " flies with " + this.wingColour + " wings!");
  }
}

let bird = new Bird("Bulbul", "Yellow");
bird.eat();
bird.fly();

```

What's Happening?

- Pet is the parent class (**like a mom or dad**).
- Dog, Cat, and Bird are child classes (**like kids**).
- extends means the child gets everything from the parent.
- super(**name**) in Bird calls the parent's constructor to set name.
- Children can add their own stuff (**wingColour**) or actions (bark, meow, fly).

Why Use Inheritance?

Inheritance saves time. You write shared stuff (like eat) once, and many classes can use it.
It's like sharing a toy box with siblings.

Try This

1. Open the console (F12).
2. Copy and run the Pet and Dog code.
3. Copy and run the Cat code. Call jump.
4. Copy and run the Bird code. Try a new wingColour.
5. Make a Fish class that uses Pet.
    - Add stuff: finColour.
    - Add action: swim (shows "[name] swims with [finColour] fins!").
6. Make a Fish object and call eat and swim.
7. Add a bubble action to Fish (shows "[name] makes bubbles!").
8. Make two Fish objects with different finColour values.
9. Show your teacher your code.

Encapsulation

Encapsulation hides stuff to keep it safe. It's like a locked toy box—you can add or check toys, but you can't mess with what's inside. Only special actions can touch hidden stuff.

Example 1: Toy Box (Simple)

```
class ToyBox {
  #toys = 0;

  addToy() {
    this.#toys = this.#toys + 1;
    console.log("Toys in box: " + this.#toys);
  }

  showToys() {
    console.log("You have " + this.#toys + " toys!");
  }
}

let box = new ToyBox();
box.addToy();
box.addToy();
box.showToys();
console.log(box.#toys);

```

Example 2: Candy Jar

```
class CandyJar {
  #candies = 0;

  addCandy(number) {
    this.#candies = this.#candies + number;
    console.log("Added " + number + " candies. Total: " + this.#candies);
  }

  eatCandy() {
    if (this.#candies > 0) {
      this.#candies = this.#candies - 1;
      console.log("Ate 1 candy. Left: " + this.#candies);
    } else {
      console.log("No candies left!");
    }
  }
}

let jar = new CandyJar();
jar.addCandy(3);
jar.eatCandy();

```

Example 3: Bank Account

```
class BankAccount {
  #balance = 0;
  #owner;

  constructor(owner) {
    this.#owner = owner;
  }

  deposit(amount) {
    if (amount > 0) {
      this.#balance = this.#balance + amount;
      console.log(this.#owner + " added $" + amount + ". Balance: $" + this.#balance);
    } else {
      console.log("Add a positive amount!");
    }
  }

  withdraw(amount) {
    if (amount <= this.#balance) {
      this.#balance = this.#balance - amount;
      console.log(this.#owner + " took $" + amount + ". Balance: $" + this.#balance);
    } else {
      console.log("Not enough money!");
    }
  }

  checkBalance() {
    console.log(this.#owner + "'s balance: $" + this.#balance);
  }
}

let account = new BankAccount("Anna");
account.deposit(100);
account.withdraw(30);
account.checkBalance();
```

What's Happening?

- #toys, #candies, #balance are hidden (like a locked box).
- Actions like addToy, eatCandy, or deposit let you safely change hidden stuff.
- # makes stuff private (nobody can touch it directly).
- Complex examples add checks (like if (amount > 0)) to make actions smarter.

Why Use Encapsulation?

Encapsulation keeps your code safe. Nobody can break hidden stuff, like locking your toys so nobody steals them.

Try This

1. Open the console (F12).
2. Copy and run the ToyBox code. Call addToy three times.
3. Copy and run the CandyJar code. Try eatCandy when there are no candies.
4. Copy and run the BankAccount code. Try withdraw more than the balance.
5. Make a Backpack class with:
    - Hidden: #books (starts at 0), #owner.
    - Do: addBook (add books, show total), removeBook (remove 1 book if any, show total), checkBooks (show number of books).
6. Make a Backpack object for "Sam". Add 4 books, remove 2, and check books.
7. Add a check in addBook: only add if the number is positive.
8. Try to see #books directly (should show an error).
9. Show your teacher your code.

Abstraction

Abstraction means showing only the easy parts and hiding the hard stuff. It's like a TV remote—you press buttons to change channels, but you don't see the wires or circuits inside.

Example 1: Robot

```
class Robot {
  #power = 100;

  move() {
    console.log("Robot moves forward!");
  }

  stop() {
    console.log("Robot stops!");
  }
}

let myRobot = new Robot();
myRobot.move();
myRobot.stop();
console.log(myRobot.#power);
```

Example 2: Fan (More Actions)

```

class Fan {
  #speed = 0;

  turnOn() {
    this.#speed = 1;
    console.log("Fan spins at speed " + this.#speed + "!");
  }

  turnOff() {
    this.#speed = 0;
    console.log("Fan is off!");
  }
}

let myFan = new Fan();
myFan.turnOn();
myFan.turnOff();

```

Example 3: Game Controller

```
class GameController {
  #battery = 100;
  #connected = false;

  connect() {
    this.#connected = true;
    console.log("Controller is connected!");
  }

  pressButton(action) {
    if (this.#connected && this.#battery > 0) {
      console.log("Action: " + action);
      this.#battery = this.#battery - 5;
    } else {
      console.log("Connect or charge controller!");
    }
  }

  charge() {
    this.#battery = 100;
    console.log("Battery full: " + this.#battery + "%");
  }
}

let controller = new GameController();
controller.connect();
controller.pressButton("Jump");
controller.charge();
```

What's Happening?

- #power, #speed, #battery are hidden (like wires in a remote).
- Actions like move, turnOn, or pressButton are easy to use.
- You don't see how things work inside—that's abstraction!
- Complex examples add logic (like checking #connected or #battery).

Why Use Abstraction?

Abstraction makes code simple to use. You focus on what the object does (like pressing a button), not how it works inside (like circuits).

Try This

1. Open the console (F12).
2. Copy and run the Robot code. Call move and stop.
3. Copy and run the Fan code. Try turnOn twice.
4. Copy and run the GameController code. Try pressButton without connecting.
5. Make a Lamp class with:
    - Hidden: #light (starts at 0), #colour.
    - Do: turnOn (set #light to 1, show "Lamp is on with [colour] light!"), turnOff (set #light to 0, show "Lamp is off!").
6. Make a Lamp object with a colour (like "Yellow"). Call turnOn and turnOff.
7. Add a dim action (if #light is 1, show "Lamp dims!").
8. Try to see #light directly (should show an error).
9. Show your teacher your code.

Polymorphism

Polymorphism means different objects can do the same thing in their own way. It's like a "sing" button that sounds different for a dog, cat, or bird.

Example 1: Animals (Simple)

```
class Animal {
  noise() {
    console.log("Some noise...");
  }
}

class Dog extends Animal {
  noise() {
    console.log("Woof!");
  }
}

class Cat extends Animal {
  noise() {
    console.log("Meow!");
  }
}

let dog = new Dog();
let cat = new Cat();

dog.noise();
cat.noise();
```

Example 2: Bird

```
class Animal {
  noise() {
    console.log("Some noise...");
  }
}

class Dog extends Animal {
  noise() {
    console.log("Woof!");
  }
}

class Cat extends Animal {
  noise() {
    console.log("Meow!");
  }
}

class Bird extends Animal {
  noise() {
    console.log("Chirp!");
  }

  fly() {
    console.log("Flying high!");
  }
}

let bird = new Bird();
bird.noise();
bird.fly();
```

Example 3: Zoo

```
class Animal {
  noise() {
    console.log("Some noise...");
  }
}
class Dog extends Animal {
  noise() {
    console.log("Woof!");
  }
}
class Cat extends Animal {
  noise() {
    console.log("Meow!");
  }
}
class Bird extends Animal {
  noise() {
    console.log("Chirp!");
  }

  fly() {
    console.log("Flying high!");
  }
}
class Zoo {
  makeAllNoises(animals) {
    for (let animal of animals) {
      animal.noise();
    }
  }
}

let zoo = new Zoo();
let animals = [new Dog(), new Cat(), new Bird()];
zoo.makeAllNoises(animals);
```

What's Happening?

- Animal has a noise action.
- Dog, Cat, and Bird change noise to their own sound.
- Polymorphism means the same action (noise) works differently for each object.
- The Zoo class shows how polymorphism lets you use many objects together.

Why Use Polymorphism?

Polymorphism makes code flexible. You can call the same action on different objects, and each does its own thing.

Try This

1. Open the console (F12).
2. Copy and run the Animal, Dog, and Cat code.
3. Copy and run the Bird code. Call noise and fly.
4. Copy and run the Zoo code. Add a new Bird to the animals list.
5. Make a Cow class that uses Animal.
    - Change noise to show "Moo!".
6. Make a Cow object and call noise.
7. Add Cow to the Zoo animals list and run makeAllNoises.
8. Add a dance action to Dog and Cat (shows "[animal] dances!").
9. Show your teacher your code.

Glossary of OOP Words 📖

- **Object**: A thing with stuff and actions (like a pet with a name and bark).
- **Class**: A plan to make objects (like a toy mold).
- **Constructor**: Sets up stuff for a class (like adding a name to a toy).
- **This**: Means "this object's stuff" (like this.name is the object's name).
- **Inheritance**: A class using stuff from another class (like a kid getting toys from a parent).
- **Encapsulation**: Hiding stuff to keep it safe (like a locked box).
- **Abstraction**: Showing only easy parts, hiding hard stuff (like a remote's buttons).
- **Polymorphism**: Same action, different ways (like animals making different noises).
- **Private**: Hidden stuff (like #toys). Use # to hide it.

What You Learned 🎓

- **Objects**: Things with stuff and actions (like a pet or snack).
- **Classes**: Plans to make many objects (like a recipe).
- **Inheritance**: Sharing stuff between classes (like kids and parents).
- **Encapsulation**: Hiding stuff to keep it safe (like a locked box).
- **Abstraction**: Making things simple by hiding hard parts (like a remote).
- **Polymorphism**: Same action, different ways (like animals singing).

Final Challenge

Create a **Mini Zoo Project** to use everything you learned:

1. Make an Animal class with name and noise action.
2. Make Dog, Cat, and Fish classes that use Animal.
    - Each has its own noise ("Woof!", "Meow!", "Bubble!").
    - Add unique actions (like fetch for Dog, purr for Cat).
3. Make a Zoo class with hidden #animals (starts as empty list []).
    - Add actions: addAnimal (add an animal to #animals), makeNoises (call noise for all animals).
4. Create a Zoo object, add 1 Dog, 1 Cat, and 1 Fish, and call makeNoises.