Object Oriented Programming

An Easy Guide for Beginners

Which one is better?

Let's start with some code right away.

```
cat.eat(food)
cat.sleep(amountOfHours)
cat.drink(water)
```

```
eat(cat, food)
sleep(cat, amountOfHours)
drink(cat, water)
```

Which one is better?

Let's compare the first lines.

```
cat.eat(food)
cat.sleep(amountOfHours)
cat.drink(water)
```

```
eat(cat, food)
sleep(cat, amountOfHours)
drink(cat, water)
```

Which one is better?

Let's compare the first lines.

```
cat.eat(food)
cat.sleep(amountOfHours)
cat.drink(water)
```

object-oriented

```
eat(cat, food)
sleep(cat, amountOfHours)
drink(cat, water)
```

procedural

Again, which one is better?

```
cat.eat(food)
cat.sleep(amountOfHours)
cat.drink(water)
```

```
eat(cat, food)
sleep(cat, amountOfHours)
drink(cat, water)
```

Better in what?

Both can be a valid solution.

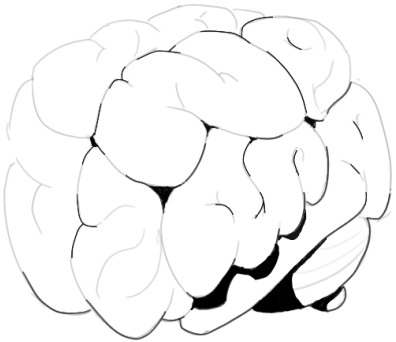
So why do we need oop?

1. Question

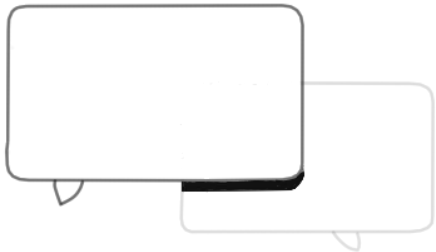
Why do we need object oriented programming?

Why?

The idea of object-oriented programming is quite natural.



think



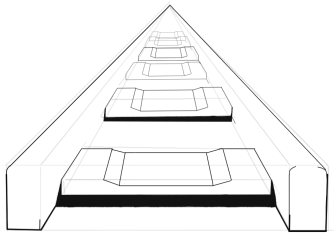
communicate

Why?

It is a paradigm. A set of rules we can follow.

This leads us to an consistent code base.

Rules give us fewer possibilities to break out of consistency.



rails

2. Question

What is an Object?

What is an Object?

An object is something that groups data and actions together.

Data

A cat is tired.

A cat is hungry.

Actions

A cat can sleep.

A cat can eat.

UML Class-Diagram

What is an Object?

An object is something that groups data and actions together.

Data
A cat is tired.
A cat is hungry.

Actions
A cat can sleep.
A cat can eat.

UML Class-Diagram

How does it look in javascript?

Cat
tiredness hunger
sleep eat

Code

```
class Cat {}
```

How does it look in javascript?

Cat
tiredness hunger
sleep eat

Code

```
class Cat {
    tiredness
    hunger
}
```

How does it look in javascript?

Cat
tiredness hunger
sleep eat

Let's create a cat that is very tired but not so hungry.

Code

```
class Cat {
    tiredness
    hunger
}
```

```
const tiredCat = new Cat()
tiredCat.tiredness = 10
tiredCat.hunger = 3
```

How does it look in javascript?

Cat
tiredness hunger
sleep eat

Let's create a cat that is very tired but not so hungry.

Let's create a cat that is very hungry but not so tired.

Code

```
class Cat {
    tiredness
    hunger
}
```

```
const tiredCat = new Cat()
tiredCat.tiredness = 10
tiredCat.hunger = 3
```

```
const hungryCat = new Cat()
hungryCat.tiredness = 2
hungryCat.hunger = 10
```

How does it look in javascript?

Cat
tiredness hunger
sleep eat

A `constructor` simplifies the creation.

Code

```
class Cat {
    tiredness
    hunger

    constructor(tiredness, hunger) {
        this.tiredness = tiredness
        this.hunger = hunger
    }

}
```

```
const tiredCat = new Cat(10, 3)
const hungryCat = new Cat(2, 10)
```

How does it look in javascript?

Cat
tiredness hunger
sleep eat

A `constructor` can keep our objects safe.

Code

```
class Cat {
    tiredness
    hunger

    constructor(tiredness, hunger) {
        if (tiredness > 10) { throw new Error() }
        this.tiredness = tiredness
        this.hunger = hunger
    }

}
```

```
const tiredCat = new Cat(100, 3) // throws an Error
```

How does it look in javascript?

Cat
tiredness hunger
sleep eat

Code

```
class Cat {
    tiredness
    hunger

    constructor(tiredness, hunger) {
        if (tiredness > 10) { throw new Error() }
        this.tiredness = tiredness
        this.hunger = hunger
    }

    sleep() {
        this.tiredness -= 5
    }

    feed() {
        this.hunger -= 5
    }
}
```

How does it look in javascript?

Cat
tiredness hunger
sleep eat

The tired cat needs some sleep.

Code

```
class Cat {
    tiredness
    hunger

    constructor(tiredness, hunger) {
        if (tiredness > 10) { throw new Error() }
        this.tiredness = tiredness
        this.hunger = hunger
    }

    sleep() {
        this.tiredness -= 5
    }

    feed() {
        this.hunger -= 5
    }
}

const tiredCat = new Cat(10, 0)
tiredCat.sleep()
```

3. Question

How to program object oriented?

How?

OOP is based upon 4 concepts.

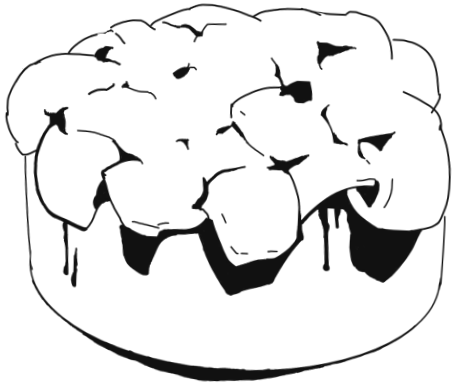
Or a pie.

Abstration

Polymorphism

I nheritance

Encapsulation



a pie

4. Question

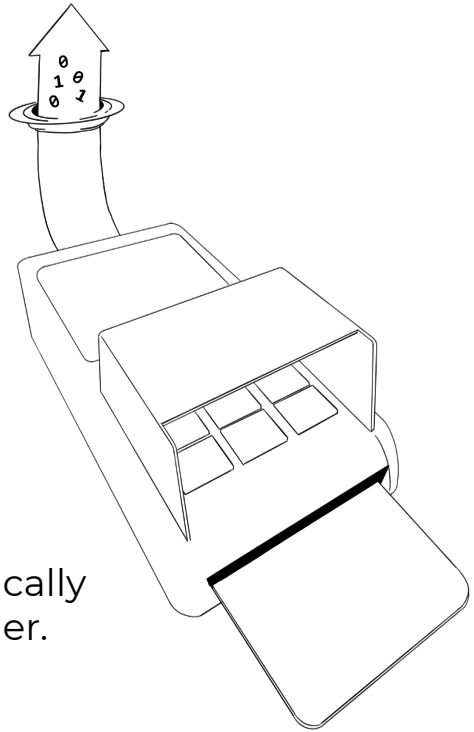
What is Abstraction?

Encapsulation
Inheritance
Polymorphism

What is Abstraction?

It is the omission of details.

We are interested in what and not how something is done.



How the payment works technically is abstracted for us as a customer.

Without Abstraction

```
cat.openMouth()
cat.stickOutTongue()
cat.takeWithTongue(water)
```

With Abstraction

```
cat.drink(water)
```

Without Abstraction

```
cardReader.decrypt(card)
cardReader.validat(pin)
cardReader.encrypt(userInformation)
cardReader.send(userInformation)
```

With Abstraction

```
cardReader.upload(transaction)
```

5. Question

What is Encapsulation?

Abstraction

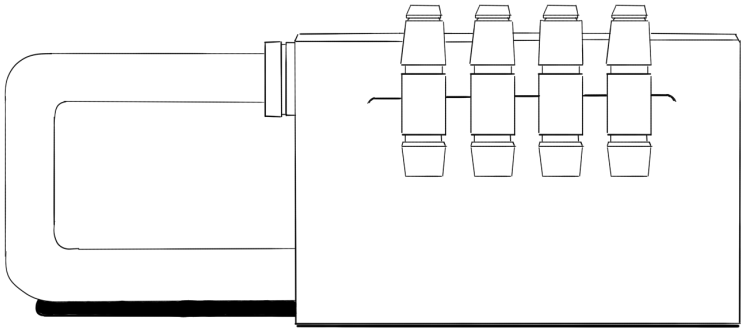
Inheritance

Polymorphism

What is Encapsulation?

It is the hiding of the internals.

We want to control how an object behaves.



Nobody gets the PIN through the lock.

Without Encapsulation

```
person.walkOnTailOf(cat)
// cat is angry
cat.happiness += 900
```

With Encapsulation

```
person.walkOnTailOf(cat)
// cat is angry
cat.pet()
```

Without Encapsulation

```
// wrong attempt
lock.openWith(1234)

// change the pin
lock.pin = 1234
lock.openBy(1234)
```

With Encapsulation

```
// wrong attempt
lock.openWith(1234)

// change the pin
lock.pin = 1234
```

Do not allow to modify objects directly.

Without Encapsulation

```
person.walkOnTailOf(cat)
// cat is angry
cat.happiness += 900
```

Do not allow to modify your objects directly.

With Encapsulation

```
person.walkOnTailOf(cat)
// cat is angry
cat.pet()
```

Without Encapsulation

```
person.walkOnTailOf(cat)
// cat is angry
cat.happiness += 900
```

With Encapsulation

```
person.walkOnTailOf(cat)
// cat is angry
cat.pet()
```

Objects should modify their internals by them self.

6. Question

What is Inheritance?

Abstraction

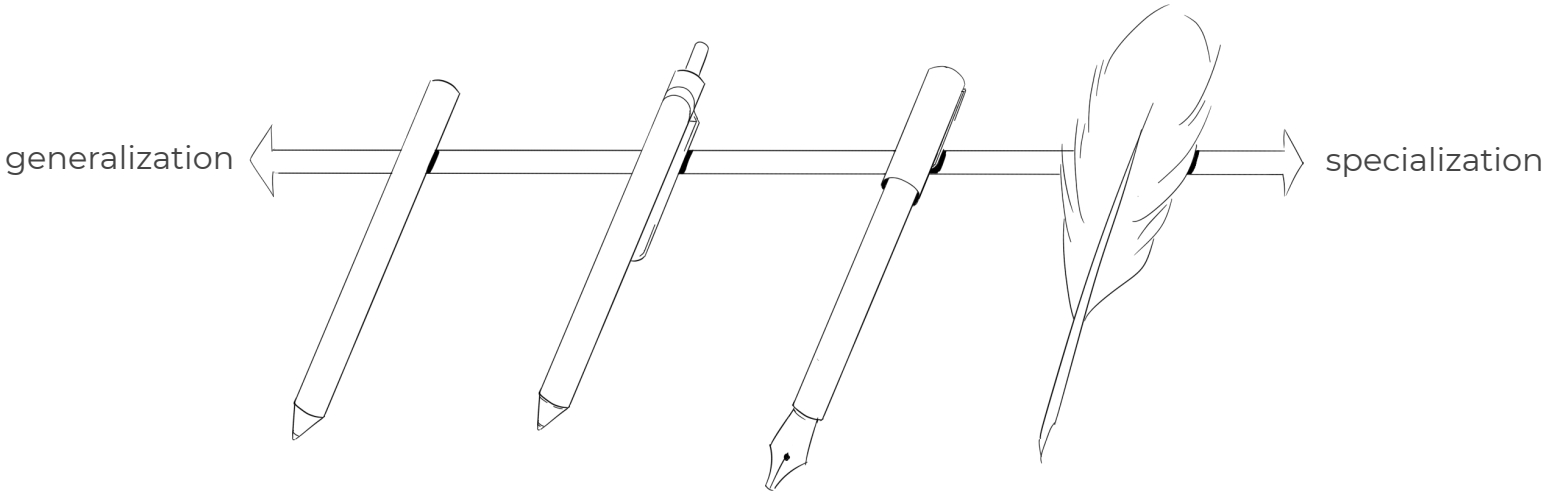
Encapsulation

Polymorphism

What is Inheritance?

A derived class specializes the behavior of the general base class.

We can share and alter behavior.



Without Inheritance

```
class BritishShorthair {
    #_hunger = 0

    feed() {
        this.#_hunger -= 10
    }
}
```

```
class MaineCoon {
    #_hunger = 0

    feed() {
        this.#_hunger -= 5
    }
}
```

With Inheritance

```
class Cat {
    #_hunger = 0
    #_hungerCount

    constructor(hungerCount) {
        this.#_hungerCount = hungerCount
    }

    feed() { this.#_hunger -= this.#_hungerCount }
}
```

```
class BritishShorthair extends Cat {
    constructor() {
        super(10)
    }
}
```

```
class MaineCoon extends Cat {
    constructor() {
        super(5)
    }
}
```

Without Inheritance

```
class Pencil {
    #_thickness = 0.3

    draw(from, to) {
      // ...
    }
}
```

```
class Ballpen {
    #_thickness = 0.5

    ink(start, end) {
        // ...
    }
}
```

With Inheritance

```
class Pen {
    #_thickness

    constructor(thickness) {
        this.#_thickness = thickness
    }
    draw() { /* common logic */ }
}
```

```
class Pencil extends Pen {
    constructor() {
        super(0.3)
    }
    draw() {/* special logic */ }
}
```

```
class Ballpen extends Pen {
    constructor() {
        super(0.5)
    }
    draw() {/* special logic */ }
}
```

Without Inheritance

```
class BritishShorthair {
    #_hunger = 0

    feed() {
        this.#_hunger -= 10
    }
}
```

```
class MaineCoon {
    #_hunger = 0

    feed() {
        this.#_hunger -= 5
    }
}
```

If the logic changes in one class, the logic must also change in the other.

With Inheritance

```
class Cat {
    #_hunger = 0
    #_hungerCount

    constructor(hungerCount) {
        this.#_hungerCount = hungerCount
    }

    feed() { this.#_hunger -= this.#_hungerCount }
}
```

```
class BritishShorthair extends Cat {
    constructor() {
        super(10)
    }
}
```

```
class MaineCoon extends Cat {
    constructor() {
        super(5)
    }
}
```

Without Inheritance

```
class BritishShorthair {
    #_hunger = 0

    feed() {
        this.#_hunger -= 10
    }
}
```

```
class MaineCoon {
    #_hunger = 0

    feed() {
        this.#_hunger -= 5
    }
}
```

The logic will change for all.

With Inheritance

```
class Cat {
    #_hunger = 0
    #_hungerCount

    constructor(hungerCount) {
        this.#_hungerCount = hungerCount
    }

    feed() { this.#_hunger -= this.#_hungerCount }
}
```

```
class BritishShorthair extends Cat {
    constructor() {
        super(10)
    }
}
```

```
class MaineCoon extends Cat {
    constructor() {
        super(5)
    }
}
```

Abstraction
Encapsulation
Inheritance

7. Question

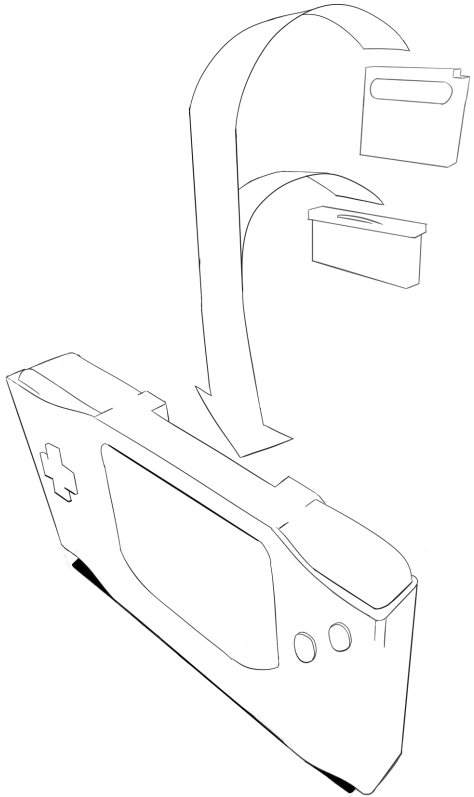
What is Polymorphism?

What is Polymorphism?

There can be many different forms of an executable unit.

We can alter behavior.

Games that have different shapes are playable.



Without Polymorphism

```
person.buyBritishShorthair()
person.buyMaineCoon()
```

With Polymorphism

```
person.buy(britishShorthair)
person.buy(maineCoon)
```

Without Polymorphism

```
gamboy.playSmallerFormat(pokemon)
gamboy.playBiggerFormat(mario)

/*
for every format we need a new
method
*/
```

With Polymorphism

```
person.play(pokemon)
person.play(mario)
```

Without Polymorphism

```
person.buyBritishShorthair()
person.buyMaineCoon()
```

A new method must be created for each additional breed.

With Polymorphism

```
person.buy(britishShorthair)
person.buy(maineCoon)
```

Without Polymorphism

```
person.buyBritishShorthair()
person.buyMaineCoon()
```

With Polymorphism

```
person.buy(britishShorthair)
person.buy(maineCoon)
```

This variant can be expanded infinitely without changing the person.

Project Example

The Cat

Write yourself a virtual cat - animals with a CLI are so much nicer than ones with fur.

- Create an object that represents a cat. It should have **properties** for tiredness, hunger, loneliness and happiness

- Next, write **methods** that increase and decrease those properties. Call them something that actually represents what would increase or decrease these things, like "**feed**", "**sleep**", or "**pet**".

- Last, write a method that prints out the cat's status in each area. (Be creative e.g. Paws is really hungry, Paws is VERY happy.)

https://github.com/gSchool/JS-Intro-OOP-Exercises#the-cat

We will compare a procedural with an object-oriented approach.

1. Step

Build the Cat

procedural

Task

Create an object that represents a cat.

It should have **properties** for **tiredness**, **hunger** and **happiness**.

Code

Task

Create an object that represents a cat.

It should have **properties** for tiredness, hunger and happiness.

Code

```
const cat = {}
```

Task

Create an object that represents a cat.

It should have **properties** for tiredness, hunger and happiness.

Code

```
const cat = {tiredness: 10, hunger: 10, happiness: 0}
```

Violations

Violations

Abstraction

We know how a cat's feelings are represented.

Code

```
const cat = {tiredness: 10, hunger: 10, happiness: 0}
```

Violations

Abstraction

We know how a cat's feelings are represented.

When we interact with the cat, we use this representation.

Code

```
const cat = {tiredness: 10, hunger: 10, happiness: 0}
```

```
function sleep(cat) {
    cat.tiredness -= 5
}
```

Violations

Abstraction

We know how a cat's feelings are represented.

When we interact with the cat, we use this representation.

If the representation is changed, all interactions must also be changed.

Code

```
const cat = {tiredness: "very", /* ... */ }
```

```
function sleep(cat) {
    if (cat.tiredness === “very”) {
      cat.tiredness = “low”
    }
    /* … */
}
```

2. Step

Build the Functions

procedural

Task

Next, write **methods** that increase and decrease those properties.

Call them something that actually represents what would increase or decrease these things, like "**feed**", "**sleep**", or "**pet**"

Code

```
const cat = {tiredness: 10, hunger: 10, happiness: 0}

function sleep(cat) {
    cat.tiredness -= 5
}

function feed(cat) {
    cat.feed -= 5
}

function pet(cat) {
    cat.happiness += 5
}
```

Violations

Violations

Encapsulation

If the functions are in different files, it will be difficult to find them.

Code

```
const cat = {tiredness: 10, hunger: 10, happiness: 0}

function sleep(cat) {
    cat.tiredness -= 5
}

function feed(cat) {
    cat.feed -= 5
}

function pet(cat) {
    cat.happiness += 5
}
```

Violations

Encapsulation

If the functions are in different files, it will be difficult to find them.

You don't know how someone changes the cat.

Code

```
const cat = {tiredness: 10, hunger: 10, happiness: 0}

function sleep(cat) {
    cat.tiredness -= 5
}

function feed(cat) {
    cat.feed -= 5
}

function pet(cat) {
    cat.happiness += 5
}



// somewhere in the code that you don't control
function bathe(cat) {
    cat.happiness + 1000
}
```

1. Step

Build the Cat

object-oriented

Task

Create an object that represents a cat.

It should have **properties** for **tiredness**, **hunger** and **happiness**.

Code

Task

Create an object that represents a cat.

It should have **properties** for tiredness, hunger and happiness.

Code

```
class Cat {}
```

Task

Create an object that represents a cat.

It should have **properties** for tiredness, hunger and happiness.

Code

```
class Cat {
    tiredness
    hunger
    happiness

    constructor(tiredness, hunger, happiness) {
        this.tiredness = tiredness
        this.hunger = hunger
        this.happiness = happiness
    }
}
```

Violation

We still haven't created a level of abstraction. This approach is similar to the procedural version.

We need a way to hide the properties?

Code

```
class Cat {
    tiredness
    hunger
    happiness

    constructor(tiredness, hunger, happiness) {
        this.tiredness = tiredness
        this.hunger = hunger
        this.happiness = happiness
    }
}


const aCat = new Cat(10, 10, 0)
aCat.tiredness = 1000
```

Syntax

The # makes properties *privat*.

The _ is a convention for private variables.

Code

```
class Cat {
    #_tiredness
    #_hunge
    #_happiness

    constructor(tiredness, hunger, happiness) {
        this.#_tiredness = tiredness
        this.#_hunger = hunger
        this.#_happiness = happiness
    }
}
```

Syntax

Causes an `Error`

Code

```
class Cat {
    #_tiredness
    #_hunge
    #_happiness

    constructor(tiredness, hunger, happiness) {
        this.#_tiredness = tiredness
        this.#_hunger = hunger
        this.#_happiness = happiness
    }
}


const aCat = new Cat(10, 10, 0)
aCat.#_tiredness = 1000
```

2. Step

Build the Methods

object-oriented

Task

Next, write **methods** that increase and decrease those properties.

Call them something that actually represents what would increase or decrease these things, like "**feed**", "**sleep**", or "**pet**"

Code

```
class Cat {
    #_tiredness
    #_hunger
    #_happiness

    constructor(tiredness, hunger, happiness) {
        this.tiredness = tiredness
        this.hunger = hunger
        this.happiness = happiness
    }

    sleep() {
        this.#_tiredness -= 5
    }

    feed() {
        this.#_hunger -= 5
    }

    pet() {
        this.#_happiness += 5
    }
}
```

Thank you!

Keep on coding.