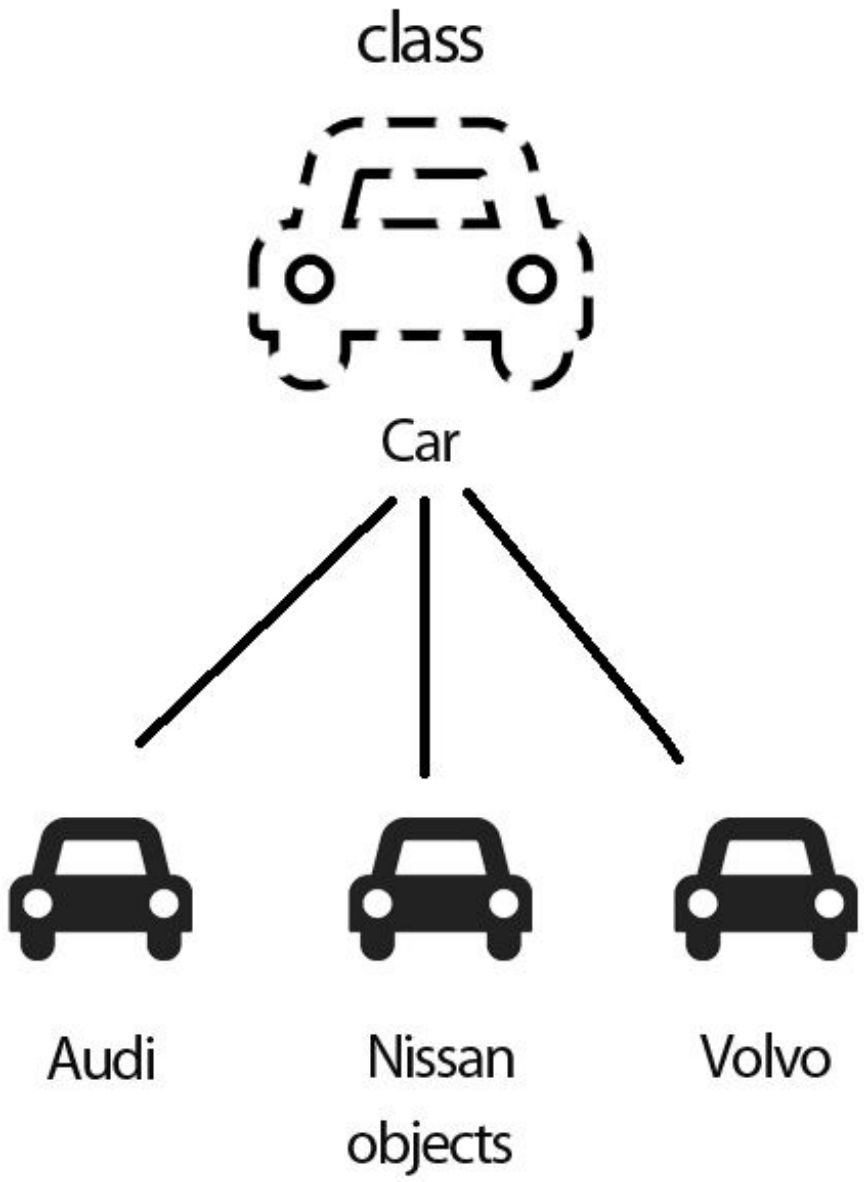
Agenda

- What is OOP?
- ES Classes and instance(object)
- this, bind, apply, call methods
- Prototype, Inheritance & Class Extends
- SOLID principles

What is OOP?

- Object-Oriented Programming is a way of writing code that allows you to create different objects from a common object.
- The common object is usually called a *blueprint,* while the created objects are called *instances.*
- Each instance has properties that are not shared with other instances.
- There are certain features or mechanisms which make a Language Object-Oriented, like: **Object, Classes, Encapsulation** and **Inheritance.**

Example: If you have a Car blueprint, you can create car instances with different names.



Note: JavaScript is not a class-based object-oriented language. But it still has ways of using object-oriented programming (OOP).

- An Object is a **unique** entity that contains **property** and **methods**.
- For example “car” is a real-life Object, which has some characteristics like colour, type, model, horsepower and performs certain actions like driving.
- The characteristics of an Object are called Property, in Object-Oriented Programming and the actions are called methods.
- An Object is an **instance** of a class.
- Objects are everywhere in JavaScript almost every element is an Object whether it is a function, array, and string.

Note: A Method in javascript is a property of an object whose value is a function.

Let’s Create our First Object.

```
let person = {
    first_name: 'John',
    last_name: 'Doe',

    //method
```

```
    getPersonDetails: function() {
        return `The name of the person is
                ${person.first_name} ${person.last_name}`
    },
}

console.log(person.first_name);
console.log(person.getPersonDetails());
```

ES Classes and instance(object)

Classes are in fact "*special functions*", and just as you can define function expressions and function declarations, the class syntax has two component class expressions and class declarations.

class declaration

One way to define a class is using a class declaration. To declare a class, you use the class keyword with the name of the class.

Create a class Rectangle using class declaration.

```
class Rectangle {
    constructor (height, width) {
        this.height = height;
        this.width = width;
    }
}
```

class expression

A class expression is another way to define a class. Class expressions can be named or unnamed. The name given to a named class expression is loc to the class's body. However, it can be accessed via the name property. Create a class Rectangle using class expression.

```
let Rectangle = class {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
};
```

Object Instance

- JavaScript is an object-based language based on prototypes, rather than being class-based.
- Because of this different basis, it can be less apparent how JavaScript allows you to create hierarchies of objects and to have inheritance of properties and their values.
- The term instance has a specific technical meaning in class-based languages.
- In these languages, an instance is an individual instantiation of a class and is fundamentally different from a class.
- In JavaScript, "instance" does not have this technical meaning because JavaScript does not have this difference between classes and instances.
- However, in talking about JavaScript, "instance" can be used informally to mean an object created using a particular constructor function.
- So, in this example, you could informally say that jane is an instance of Engineer.
- Similarly, although the terms parent, child, ancestor, and descendant do not have formal meanings in JavaScript; you can use them informally to refer to objects higher or lower in the prototype chain.

`this` Keyword in JavaScript

In JavaScript, the this keyword refers to an object.

Which object depends on how this is being invoked (used or called).

The this keyword refers to different objects depending on how it is used:`

- In an object method, this refers to the 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(), apply(), and bind() can refer this to any object.

```
fullName : function() {
    return this.firstName + " " + this.lastName;
}
```

this in a Method

When used in an object method, this refers to the object.

In the given code, this refers to the person object.

Because the fullName method is a method of the person object.

Bind

The bind() function creates a new bound function, which is an exotic function object (a term from ECMAScript 2015) that wraps the original functic object.

Calling the bound function generally results in the execution of its wrapped function.

- the JS engine is creating a new pokemonName instance and binding pokemon as its this variable. It is important to understand that it copies the pokemonName function.
- After creating a copy of the pokemonName function it is able to call logPokemon(), although it wasn't on the pokemon object initially. It will now recognizes its properties (Pika and Chu) and its methods.

And the cool thing is after we bind() a value, we can use the function just like it was any other normal function. We could even update the function accept parameters, and pass them

```
var pokemon = {
    firstname: 'Pika',
    lastname: 'Chu ',
    getPokeName: function() {
        var fullname = this. firstname + ' ' + this. lastname;
        return fullname;
    }
};
var pokemonName = function() {
    console.log(this.getPokeName() + 'I choose you!');
};

var logPokemon = pokemonName.bind(pokemon); // creates new object and binds pokemon.
//'this' of pokemon = pokemon now
logPokemon(); // 'Pika Chu I choose you!!
```

Call & apply

The call() method calls a function with a given this value and arguments provided individually.

What that means, is that we can call any function, and explicitly specify what this should reference within the calling function. Really similar to the binc method! This can definitely save us from writing hacky code.

The main differences between bind() and call() is that the call() method:

1. Accepts additional parameters as well
2. Executes the function it was called upon right away.
3. The call() method does not make a copy of the function it is being called on.

call() and apply() serve the exact same purpose. The only difference between how they work is that call() expects all parameters to be passed individually, whereas apply() expects an array of all of our parameters.

```
var pokemon = {
    firstname: 'Pika',
    lastname: 'Chu ',
    getPokeName: function() {
        var fullname = this.firstname ++ this.lastname;
        return fullname;
    }
};

var pokemonName = function(snack, hobby) {
    console.log(this.getPokeName()+' loves '+ snack + ' and ' + hobby);
};

pokemonName.call(pokemon, 'sushi', 'algorithms'); // Pika Chu loves sushi and algorithms
pokemonName.apply(pokemon, ['sushi', 'algorithms']); // Pika Chu loves sushi and algorithms
```

JavaScript Prototype

All JavaScript objects inherit properties and methods from a prototype:

- Date objects inherit from Date.prototype
- Array objects inherit from Array.prototype
- Person objects inherit from Person.prototype

The Object.prototype is on the top of the prototype inheritance chain:

Date objects, Array objects, and Person objects inherit from Object.prototype.

The JavaScript prototype property allows you to add new properties to object constructors:

```
function Person(first, last, age, eyecolor){
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
}

Person.prototype.nationality = "English";

Person.prototype.name = function() {
    return this.firstName + " " this.lastName;
};
```

JavaScript Inheritance

JavaScript does not have classes like other languages. It uses the concept of prototypes and prototype chaining for inheritance. In this post, we w
discuss how we can achieve inheritance in JavaScript using Prototypes.

Prototype Chaining

- Prototype chaining means an object's *dunder proto* or *proto* property will point to another object instead of pointing to the constructor function *prototype*.
- If the other object's *dunder proto* or *proto* property points to another object it will result in the chain.
- This is called Prototype Chaining.

Creating a prototype chaining.

The code defines two constructor functions, SuperType and SubType.

```
// SuperType constructor function
function SuperType() {
    this.name = "John"
}
```

```
//SuperType prototype
SuperType.prototype.getSuperName = function(){
    return this.name
}
//SubType prototype function
function SubType(){
    this.age 26
}

//Inherit the properties from SuperType
SubType.prototype = new SuperType();
// Add new property to SubType prototype
SubType.prototype.getSubAge = function(){
    return this.age;
}

//Create a SubType object
var subTypeObj = new SubType();
console.log(subTypeObj.name); //Output: John
console.log(subTypeObj.age); //Output: 26
console.log(subTypeObj.getSuperName()); //Output: John
console.log(subTypeObj.getSubAge()); //Output: 26
```

By default, SubType.prototype has a constructor **function which points to the constructor *function* itself and proto property which inherits the defau object properties.

Class Extends

The extends keyword is used to create a child class of another class (parent).

The child class inherits all the methods from another class.

Inheritance is useful for code reusability: reuse properties and methods of an existing class when you create a new class.

Note: From the example above; The *super()* method refers to the parent class. By calling the super() method in the constructor method, we call th parent's constructor method and gets access to the parent's properties and methods.

Create a class named "Model" which will inherit the methods from the "Car" class.

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return 'I have a ' + this.carname;
  }
}

class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a ' + this.model;
  }
}

myCar = new Model("Ford", "Mustang");
document.getElementById("demo").innerHTML = myCar.show();
```

SOLID principles in JavaScript

The invention of SOLID principles began in the late 80s. Robert C. Martin started to develop these principles while arguing the principle of softwa design on USENET (an early kind of Facebook). After addition and subtraction, Robert C. Martin formulated the principles in the early 2000s. It was un

2004 that the principles were arranged and called SOLID principles. It is an acronym that stands for five specific design principles.

- S represents the Single Responsibility principle
- O represents the Open Closed principle
- L represents the Liskov Substitution principle
- I represents the Interface Segregation principle
- D represents the Dependency Inversion principle

The S.O.L.I.D. principles can be very useful to write code:

- That is easy to understand
- Where things are where they're supposed to be
- Where classes do what they were intended to do
- That can be easily adjusted and extended without bugs
- That separates the abstraction from the implementation
- That allows to easily swap implementation (DB, API, frameworks, ...)
- Easily testable

Note: The SOLID principles are useful when constructing both individual modules or larger architectures. So, we're going to explore each princip alongside examples in JavaScript.

S - Single Responsibility Principle

Any function must be responsible for doing only ONE thing.

Example: Let's say we want to validate a form, then create a user in a Postgres Database

Incorrect

```
/* A function with such a name is a symptom of ignoring the Single Responsibility Principle
*  Validation and Specific implementation of the user creation is strongly coupled.
*  That's not good
*/
validateAndCreatePostgresUser = (name, password, email) => {

  //Call an external function to validate the user form
  const isFormValid = testForm(name, password, email);

  //Form is Valid
  if(isFormValid){
    User.Create(name, password, email) //Specific implementation of the user creation!
  }
}
```

Correct

```
//Only Validate
validateRequest = (req) => {

  //Call an external function to validate the user form
  const isFormValid = testForm(name, password, email);

  //Form is Valid
  if(isFormValid){
    createUser(req); // The user creation will be implemented in another function
  }
}

//Only Create User in the Database
createUser = (req) => User.Create(req.name, req.password, req.email)
```

This seems a pretty little change, but think about strongly coupling all the methods, than you have to change the Database for any reason

O - Open-Closed Principle

- The OCP states the following: "*Software entities (classes, modules, functions, and so on) should be open for extension, but closed for modification"* by Meyer, Bertrand*.*
- This principle advises us to refactor the system so that further changes of that kind will not cause more modification.
- Most developers recognize the OCP as a principle that guides them in the design of classes and modules.

There are two primary attributes in the OCP, they are.

- Open for extension — we are able to extend what the module does.
- Closed for modification — extending the behaviour of a module does not result in changes to the source or binary code of the module.

It would seem that these two attributes are at odds with each other because the normal way to extend the behavior of a module is to make changes the source code of that module.

If we have something like this:

```
const roles = ["ADMIN", "USER"]
checkRole = (user) => {
  if(roles.includes(user.role)){
    return true;
  }else{
    return false
  }
}

//Test role
checkRole("ADMIN"); //true
checkRole("Foo"); //false
```

And we want to add a superuser, for any reason, instead of modifying the existing code (or maybe we just can't modify it),

we could do it in another function.

```
//UNTOUCHABLE CODE!!!
const roles = ["ADMIN", "USER"]
checkRole = (user) => {
  if(roles.includes(user.role)){
    return true;
  }else{
    return false
  }
}
//UNTOUCHABLE CODE!!!

//We can define a function to add a new role with this function
addRole(role){
  roles.push(role)
}

//Call the function with the new role to add to the existing ones
addRole("SUPERUSER");

//Test role
checkRole("ADMIN"); //true
checkRole("Foo"); //false
checkRole("SUPERUSER"); //true
```

L - Liskov Substitution Principle

Barbara Liskov's famous definition of subtypes, from 1988 in a conference keynote address titled Data Abstraction and Hierarchy. In short, this princip says that to build software systems from interchangeable parts, those parts must adhere to a contract that allows those parts to be substituted one f another.

- One of the classic examples of this principle is a rectangle having four sides.
- A rectangle's height can be any value and width can be any value.
- A square is a rectangle with equal width and height.
- So we can say that we can extend the properties of the rectangle class into square class.
- In order to do that you need to swap the child (square) class with parent (rectangle) class to fit the definition of a square having four equal sides but a derived class does not affect the behavior of the parent class so if you will do that it will violate the Liskov Substitution Principle.

```
class Job {
  constructor(customer) {
    this.customer = customer;
    this.calculateFee = function () {
      console.log("calculate price"); //Add price logic
    };
  }
  Simple(customer) {
    this.calculateFee(customer);
  }
  Pro(customer) {
    this.calculateFee(customer);
    console.log("Add pro services"); //additional functionalities
  }
}

const a = new Job("Francesco");
a.Simple();
//Output:
//calculate price

a.Pro();
//Output:
//calculate price
//Add pro services...
```

I - Interface Segregation Principle

NOTE: This principle advises software designers to avoid depending on things that they don't use.

- Suppose you enter a restaurant and you are a pure vegetarian.
- The waiter in that restaurant gave you the menu card, which includes vegetarian items, non-vegetarian items, drinks, and sweets.
- In this case, as a customer, you should have a menu card which includes only vegetarian items, not everything which you don't eat in your food.
- Here the menu should be different for different types of customers.
- The common or general menu card for everyone can be divided into multiple cards instead of just one.
- Using this principle helps in reducing the side effects and frequency of required changes.

In JavaScript it means one must prevent classes from relying on modules or functions that they don't need.

We don't have interfaces in Javascript, but let's try with an example

Incorrect

```
//Validate in any case
class User {
  constructor(username, password) {
    this.username = username;
    this.password = password;
    this.initiateUser();
  }
  initiateUser() {
    this.username = this.username;
    this.validateUser()
```

```
  }

  validateUser = (user, pass) => {
    console.log("validating...");
  }
}
const user = new User("Francesco", "123456");
console.log(user);
// validating...
// User {
//   validateUser: [Function: validateUser],
//   username: 'Francesco',
//   password: '123456'
// }
```

Correct

```
//ISP: Validate only if it is necessary
class UserISP {
  constructor(username, password, validate) {
    this.username = username;
    this.password = password;
    this.validate = validate;

    if (validate) {
      this.initiateUser(username, password);
    } else {
      console.log("no validation required");
    }
  }

  initiateUser() {
    this.validateUser(this.username, this.password);
  }

  validateUser = (username, password) => {
    console.log("validating...");
  }
}

//User with validation required
console.log(new UserISP("Francesco", "123456", true));
// validating...
// UserISP {
//   validateUser: [Function: validateUser],
//   username: 'Francesco',
//   password: '123456',
//   validate: true
// }

//User with no validation required
console.log(new UserISP("guest", "guest", false));
// no validation required
// UserISP {
//   validateUser: [Function: validateUser],
//   username: 'guest',
//   password: 'guest',
//   validate: false
// }
```

D - Dependency Inversion Principle

The Dependency Inversion Principle (DIP) tells us that the most flexible systems are those in which source code dependencies refer only to abstraction not to concretions. Rather, details should depend on policies.

Looking at a real-life example

- You can consider the real-life example of a TV remote battery.
- Your remote needs a battery but it's not dependent on the battery brand.
- You can use any XYZ brand that you want and it will work.
- So we can say that the TV remote is loosely coupled with the brand name.
- Dependency Inversion makes your code more reusable.

Incorrect

```
//The HTTP Request depends on the setState function, which is a detail
http.get("http://address/api/examples", (res) => {
 this.setState({
  key1: res.value1,
  key2: res.value2,
  key3: res.value3
 });
});
```

Correct

```
//HTTP request
const httpRequest = (url, setState) => {
 http.get(url, (res) => setState.setValues(res))
};

//State set in another function
const setState = {
 setValues: (res) => {
  this.setState({
    key1: res.value1,
    key2: res.value2,
    key3: res.value3
  })
 }
}

//HTTP request, state set in a different function
httpRequest("http://address/api/examples", setState);
```

Interview Questions

> Explain call(), apply() and, bind() methods.

1. call():

- It's a predefined method in javascript.
- This method invokes a method (function) by specifying the owner object.
- Example 1:

```
function sayHello(){
  return "Hello " + this.name;
}

var obj = {name: "Sandy"};

sayHello.call(obj);

// Returns "Hello Sandy"
```

- call() method allows an object to use the method (function) of another object.

2.**apply()**

- The apply method is similar to the call() method. The only difference is that,
- **call() method takes arguments separately whereas, apply() method takes arguments as an array.**

```
function saySomething(message){
  return this.name + " is " + message;
}
var person4 = {name:  "John"};
saySomething.apply(person4, ["awesome"]);
```

3.**bind():**

- This method returns a new function, where the value of **"this"** keyword will be bound to the owner object, which is provided as a parameter.
- Example with arguments:

```
var bikeDetails = {
    displayDetails: function(registrationNumber,brandName){
    return this.name+ " , "+ "bike details: "+ registrationNumber + " , " + brandName;
  }
}

var person1 = {name:  "Vivek"};

var detailsOfPerson1 = bikeDetails.displayDetails.bind(person1, "TS0122", "Bullet");

// Binds the displayDetails function to the person1 object


detailsOfPerson1();
// Returns Vivek, bike details: TS0452, Thunderbird
```

> What are object prototypes?

All JavaScript objects inherit properties from a prototype. For example,

- Date objects inherit properties from the Date prototype
- Math objects inherit properties from the Math prototype
- Array objects inherit properties from the Array prototype.
- On top of the chain is **Object.prototype.** Every prototype inherits properties and methods from the Object.prototype.
- **A prototype is a blueprint of an object. The prototype** allows us to use properties and methods on an object even if the properties and methods do not exist on the current object.

> Difference between prototypal and classical inheritance

Programmers build objects, which are representations of real-time entities, in traditional OO programming. Classes and objects are the two sorts abstractions. A class is a generalization of an object, whereas an object is an abstraction of an actual thing. A Vehicle, for example, is a specialization of Car. As a result, automobiles (class) are descended from vehicles (object).

Classical inheritance differs from prototypal inheritance in that classical inheritance is confined to classes that inherit from those remaining classes, b prototypal inheritance allows any object to be cloned via an object linking method. Despite going into too many specifics, a prototype essentially serve as a template for those other objects, whether they extend the parent object or not.