Front-end Advanced

Object-oriented Programming

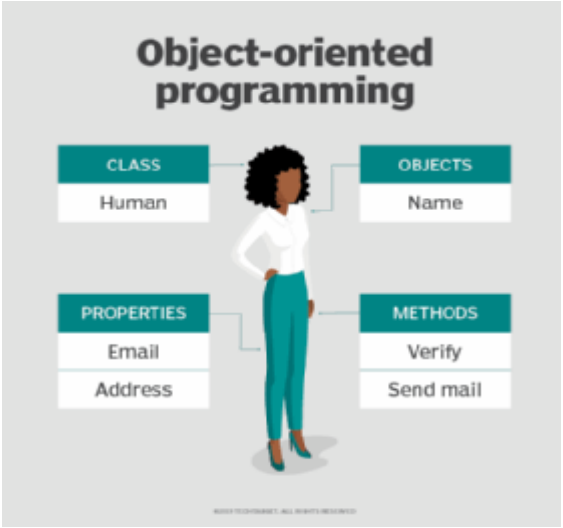
Table of Contents

Section 1

OOP

OOP Overview

- **What is OOP** (Object - oriented programming)?
  - Use objects to model real world things that want to represent inside our programs, and/or provide a simple way to access functionality that would be hard or impossible to make use of



OOP Overview

- **4 major principles:**
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

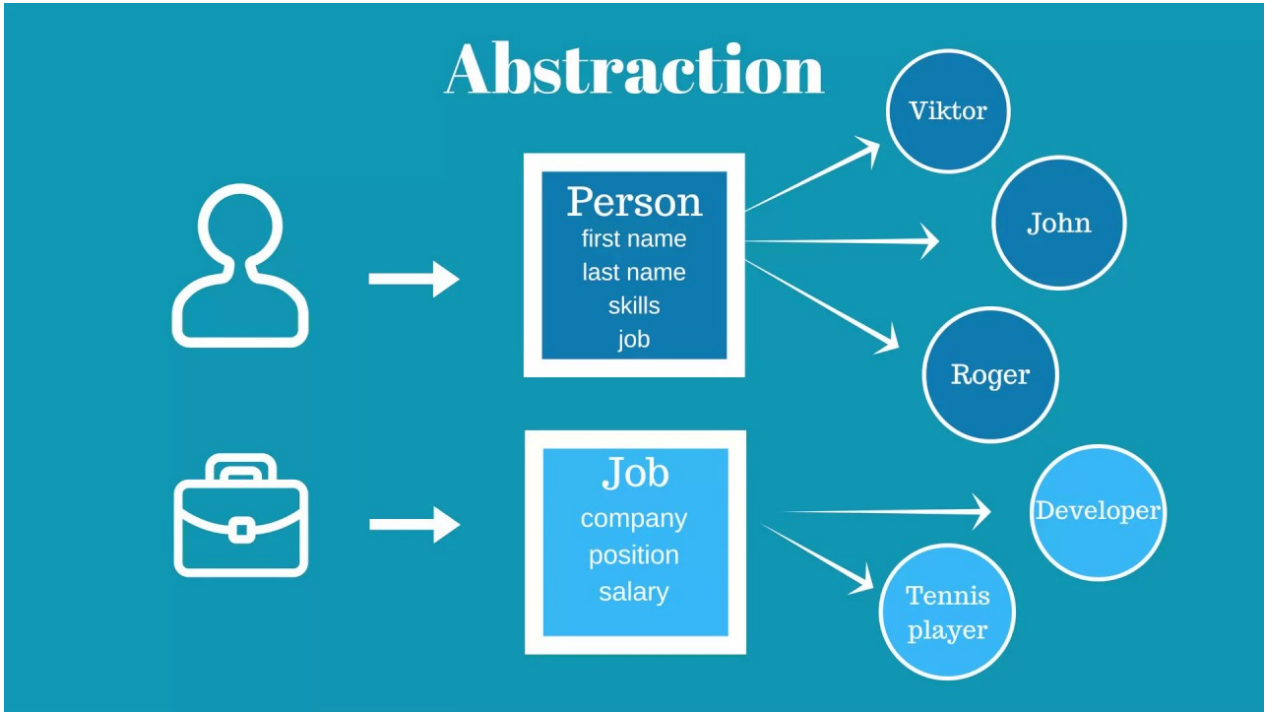
OOP Overview

➢ Abstraction:

- Abstraction is a way of creating a simple model of a more complex real-world entities, which contains the only important properties from the perspective of the context of an application.
- Abstraction allows us to override or extend functionality that should have a different behavior.

OOP Overview

- **Abstraction:**



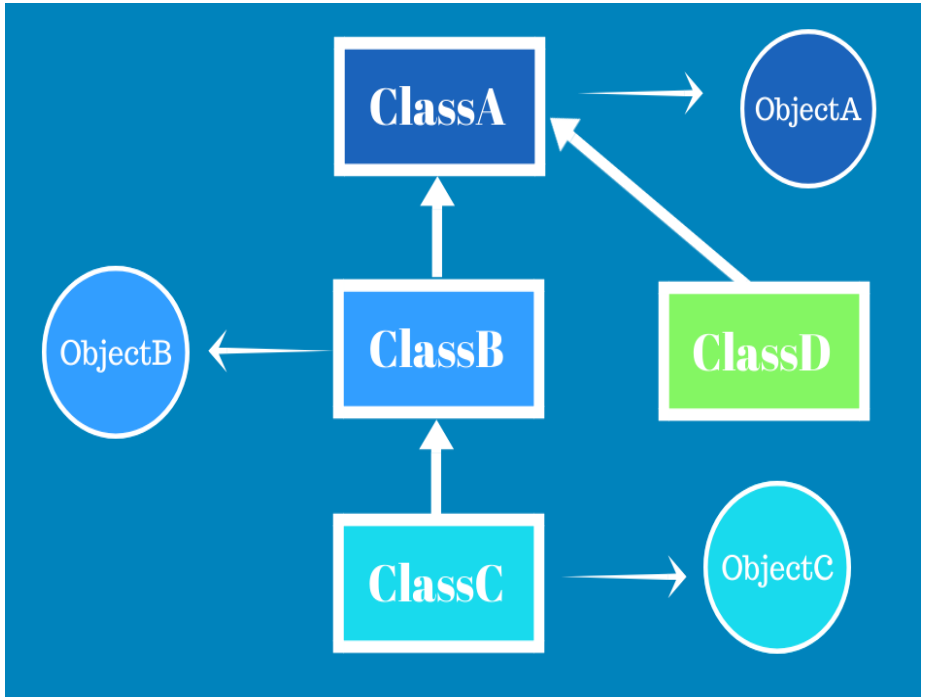
OOP Overview

Inheritance:

- Inheritance is an approach of sharing common functionality within a collection of classes.
- It provides an ability to avoid code duplication in a class that needs the same data and functions which another class already has.
- At the same time, it allows us to override or extend functionality that should have a different behavior.

OOP Overview

➢ Inheritance:



ClassB and ClassD inherit functionality from ClassA

ClassA is called a **super class** or a **parent** class of *ClassB* and *ClassD*, which are called **sub-class**

ClassC is a **child** of ClassB, and its instance has the same functionality as the instance of *ClassB* that includes also *ClassA* functionality.

we achieve inheritence by using **extends** keyword

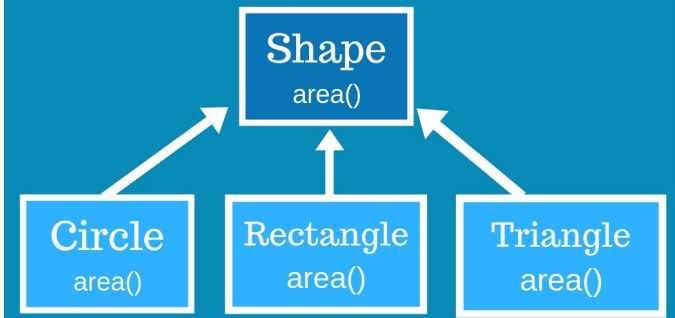
OOP Overview

- **Polymorphism:**
  - Polymorphism is an ability to create a property, a function, or an object that has more than one realization.
  - Polymorphism is an ability to substitute classes that have common functionality in sense of methods and data.
  - Inheritance has a really important relationship with polymorphism.

➢ Polymorphism:

Polymorphism

Create multiple types of shape classes, with same interface, be able to create more classes in future without updating main functionality.




```
1  class Shape {
2      area() {}
3  }
4
5  class Circle extends Shape {
6      constructor(r) {
7          super();
8          this.radius = r;
9      }
10     area() {
11         return Math.PI * this.radius ** 2;
12     }
13 }
14
15 class Rectangle extends Shape {
16     constructor(w, h) {
17         super();
18         this.width = w;
19         this.height = h;
20     }
21     area() {
22         return this.width * this.height;
23     }
24
25 }
26 const shapes = [new Circle(2), new Rectangle(2,3), new Circle(7)];
27 shapes.forEach(item => console.log(item.area()));
```

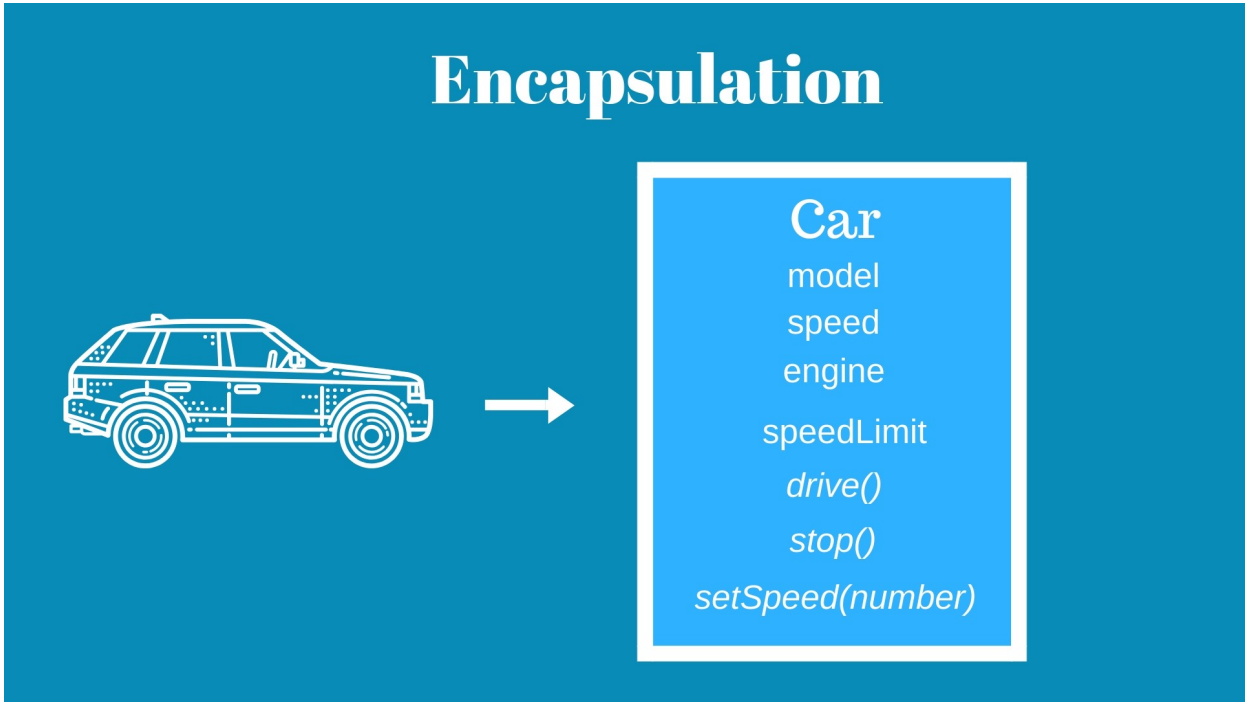
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

vkukurba@vkukurba-lenovo:~/Development/Projects/youtube/js-oop$
```

OOP Overview

- **Encapsulation:**
  - Encapsulation as a concept of bundling data related variables and properties with behavioral methods in one class.
  - Encapsulation is an approach for restricting direct access to some of the data structure elements (fields, properties, methods, etc).

OOP Overview

- **Encapsulation:**

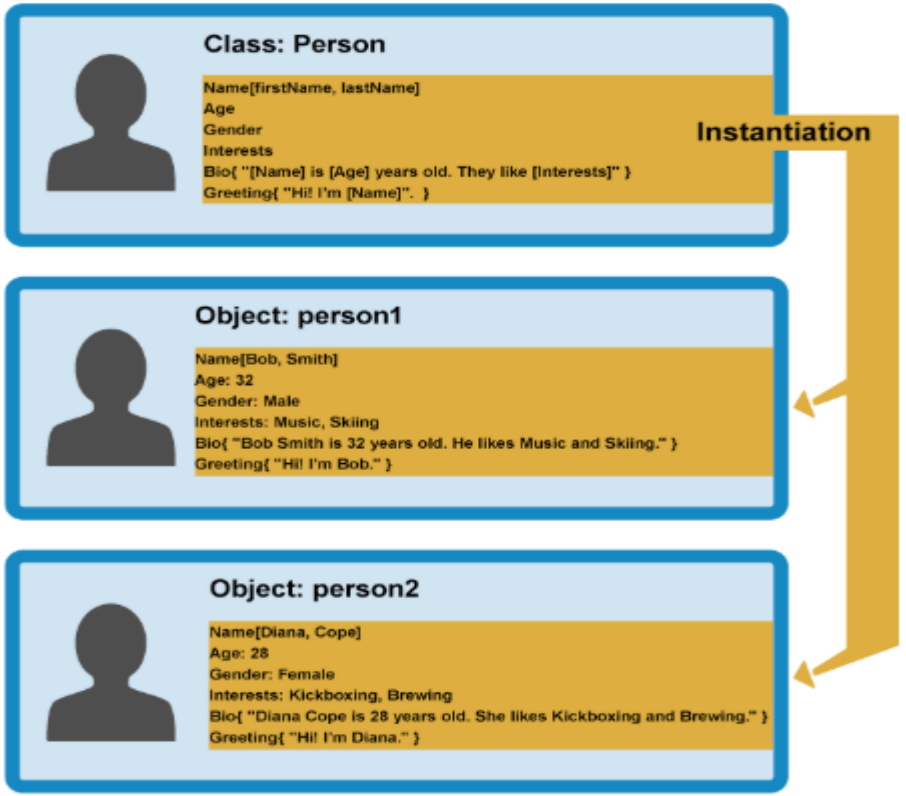


Section 2

Abstraction in JS

Class

- What is Class?
  - A class defines the shape of a type of object - what methods and properties it has
  - Instance of class: Objects that contain the data and functionality defined in the class.



Class - Syntax

- Class syntax
  - Class declarations: Use the class keyword with the name of the class (**Recommended**)
  - Class expressions: Can be named or unnamed.

```
1  class Rectangle {
2    constructor(height, width) {
3      this.height = height;
4      this.width = width;
5    }
6  }
```

Class declaration

```
1  // unnamed
2  let Rectangle = class {
3    constructor(height, width) {
4      this.height = height;
5      this.width = width;
6    }
7  };
```

Class expression

Class - Syntax

- Class is not hoisted

```
1  const p = new Rectangle(); // ReferenceError
2
3  class Rectangle {}
```

- typeof Reactangle === “function”
- Class constructor: Special functions to define and initialize objects and their features.

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

Class - constructor

- To create a new object use **new** keyword:

```
class Person {
  constructor() {
     console.log('contructor');
  }
}

var p = new Person();
```

Class - constructor

- The **constructor** method is special, it is where you initialize properties, it is called automatically when a class is initiated

```
class Person {
    // no constructor
    // default empty block-code constructor will be used
    // constructor() {
    // }
}

var p = new Person();
```

Class - Field

- **Public field declarations**: Allows public properties to initialized at the top of a class outside any constructor

```
1  class Rectangle {
2    height = 0;
3    width;
4    constructor(height, width) {
5      this.height = height;
6      this.width = width;
7    }
8  }
```

Class - Field

- **Private field declarations:** We can define private variables in our class using the hash # symbol.

```
class Rectangle {
  #height = 0;
  #width;
  constructor(height, width) {
    this.#height = height;
    this.#width = width;
  }
}
```

Class - Method

- A **method** represents an action that the entity can performs

```
class Person {
    name;  // declare field (optional)
    constructor(name) {
        this.name = name; // init field name
    }

    speak() { // declare method

    }
}

var p = new Person('AnhNV');
p.speak(); // call method speak
```

Class – Static method

- **Static method:**
  - Defined on the class itself
  - Called without instantiating their class and are also not callable when the class is instantiated.
  - Have no access to data stored in specific objects.
  - ***Syntax:***

```
static methodName() { ... }
```

Class – Static method

➢ **Example:**

```
class Person {
    name;

    static count = 0;

    constructor(name) {
        this.name = name;
        this.count; // will refer to non-static field count
        Person.count += 1; // to access static field use
ClassName.staticField
    }

    speak() {
      console.log('Hello from ', this.name);
    }

    get myName() {
       return 'Halo ' + this.name;
    }
 }

 var p = new Person('AnhNV'); // 1st person
 var b = new Person('Binh'); // 2nd person
 console.log(Person.count); // 2
```

Class - Getters, Setters

- **There are two kinds of object properties:**
  - **Data properties:** All properties that we've been using until now were data properties.
  - **Accessor properties:** They are essentially functions that execute on getting and setting a value, but look like regular properties to an external code.

Class - Getters, Setters

- **ECMAScript 5 (2009) introduced 2 accesor properties - Getter and Setters.**

```
let obj = {
  get propName() {
    // getter, the code executed on getting obj.propName
  },

  set propName(value) {
    // setter, the code executed on setting obj.propName = value
  }
};
```

Class - Getters, Setters

- **Advantages:**
  - You can check if new data is valid before setting a property
  - You can perform an action on the data which you are getting or setting on a property.
  - You can control which properties can be stored and retrieved.

Class - Getters, Setters

- ***get*** - a function without arguments, that works when a property is read.
- ***set*** - a function with one argument, that is called when the property is set.

Section 3

Encapsulation in JS

Encapsulation

- Easy with **private** field and **getter/setter**

```
class Person {
    #name;  // make name as private
    constructor(name) {
        this.#name = name; // this.#name to refer to private field
    }

    speak() { // declare method
      console.log('Hello from ', this.#name); // this.#name to refer to
private field
    }

    get myName() { // getter function
       return 'Halo ' + this.name;
    }
 }

 var p = new Person('AnhNV');
 p.speak(); // Hello from AnhNV
 p.name; // undefined
 p.#name; // not posible
```

Section 4

Inheritance in JS

Inheritance in JS

- Use **extends** keyword

```
class Person {
    name;

    constructor(name) {
        this.name = name;
    }
}

class Student extends Person {
    clazz; // class is reserved keyword

    constructor(name, clazz) {
        super(name); // call the parent constructor

        this.clazz = clazz;
    }
}

var a = new Student('AnhNV', 'ReactJS');
console.log(a); // Student {name: "AnhNV", clazz: "ReactJS"}
```

Inheritance in JS

- Use **extends**

```
class Person {
    name;

    constructor(name) {
        this.name = name;
    }

    speak() {
       console.log('speak');
    }
}

class Student extends Person {
    clazz; // class is reserved keyword

    constructor(name, clazz) {
        super(name); // call the parent constructor

        this.clazz = clazz;
    }

    study() {
        console.log('study');
    }
}

var a = new Student('AnhNV', 'ReactJS');
a.speak(); // "inherits" method from Person
a.study(); // method from its class
```

Inheritance in JS

- Use **instanceof** operator to check if an object is-a subtype of the provided Class or not.

```
class Person {
}

class Student extends Person {
}

class Trainer {}

var student = new Student('AnhNV', 'ReactJS');
console.log(student); // Student {name: "AnhNV", clazz: "ReactJS"}
student instanceof Student; // true: student is-a Student
student instanceof Person; // true: student is-a Person
student instanceof Trainer; // false: student is-not-a Trainer
```

Prototype in JS

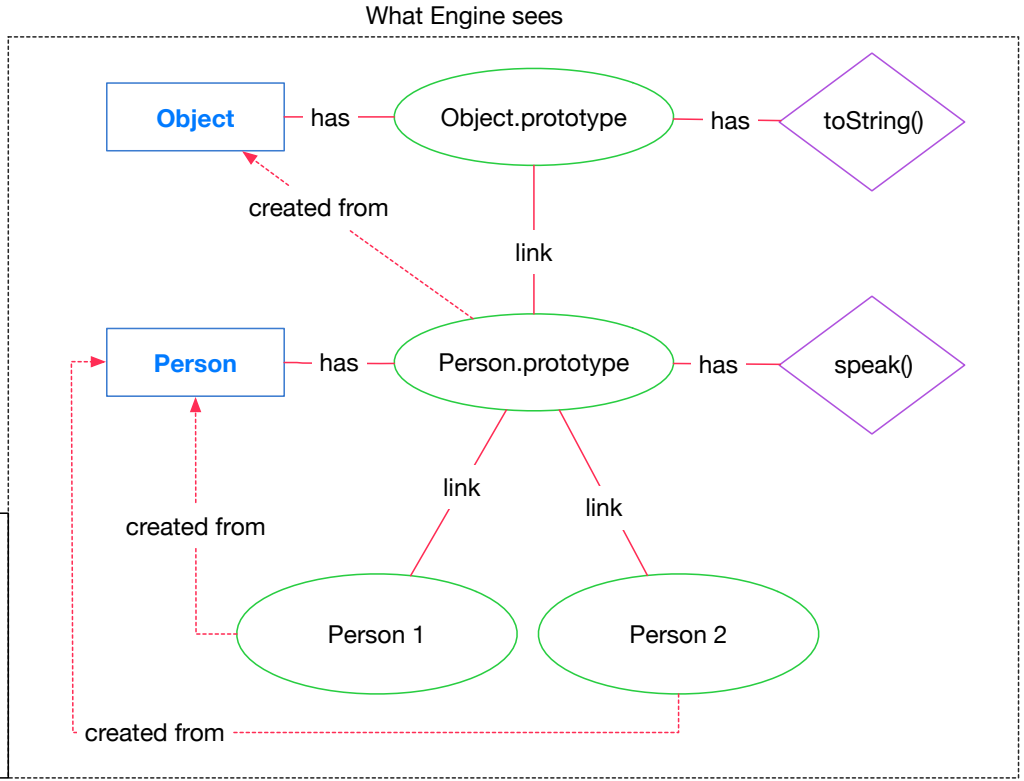
- Under the hook, JS use prototype mechanism to archieve Inheritance.
- Prototype is like a **chain** that link child object to parent object.

```
class Person {
  speak() {
    console.log('speak');
  }
}

class Student extends Person {
  study() {
    console.log('study');
  }
}

var a = new Student();
var b = new Student();

a.speak == b.speak; // true
a.speak === Student.prototype.speak; // true
a.study === Student.prototype.study; // true
a.study === Person.prototype.study; // false
```

Prototype in JS - Explained

What Developer sees

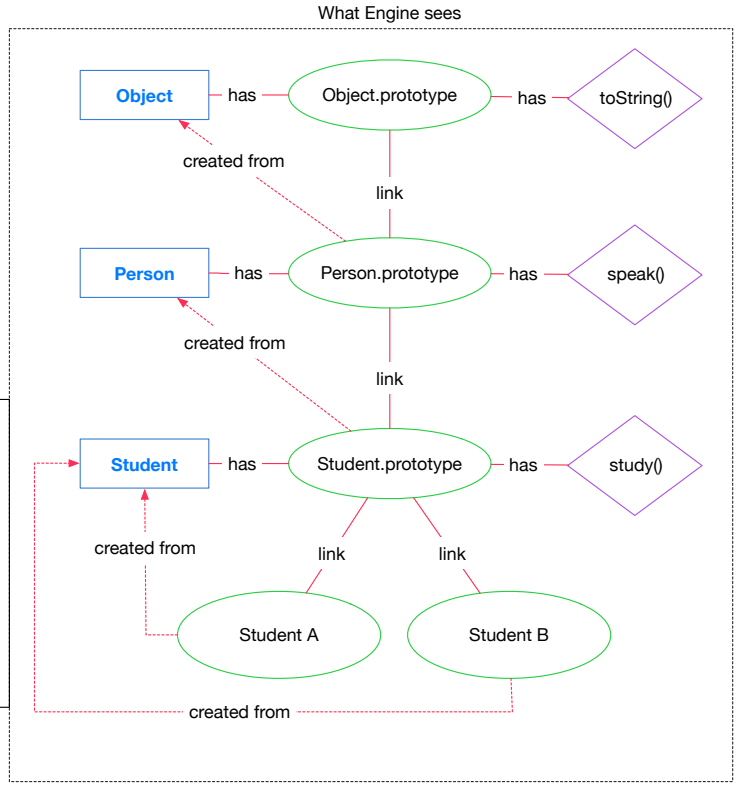
```
class Person {
  speak() {
    console.log('speak');
  }
}

var p1 = new Person();
var p2 = new Person();
```



Prototype in JS - Explained

What Developer sees

```
class Person {
  speak() {
    console.log('speak');
  }
}

class Student extends Person {
  study() {
    console.log('study');
  }
}

var a = new Student();
var b = new Student();
```



Section 5

Polymorphism in JS

Polymorphism

```
class Person {
    name;

    constructor(name) {
        this.name = name;
    }

    speak() {
       console.log('Person speak');
    }
}

class Student extends Person {
    clazz; // class is reserved keyword

    constructor(name, clazz) {
        super(name); // call the parent constructor

        this.clazz = clazz;
    }

    study() {
        console.log('study');
    }

    speak() {
       console.log('Student speak');
    }
}

var s = new Student(); // if we can do: Person s = new Student();
var p = new Person(); // Person p = new Person();
```

Section 6

`this` keyword

this

- In **OOP**, `this` keyword refers to the object containing the currently-executing code

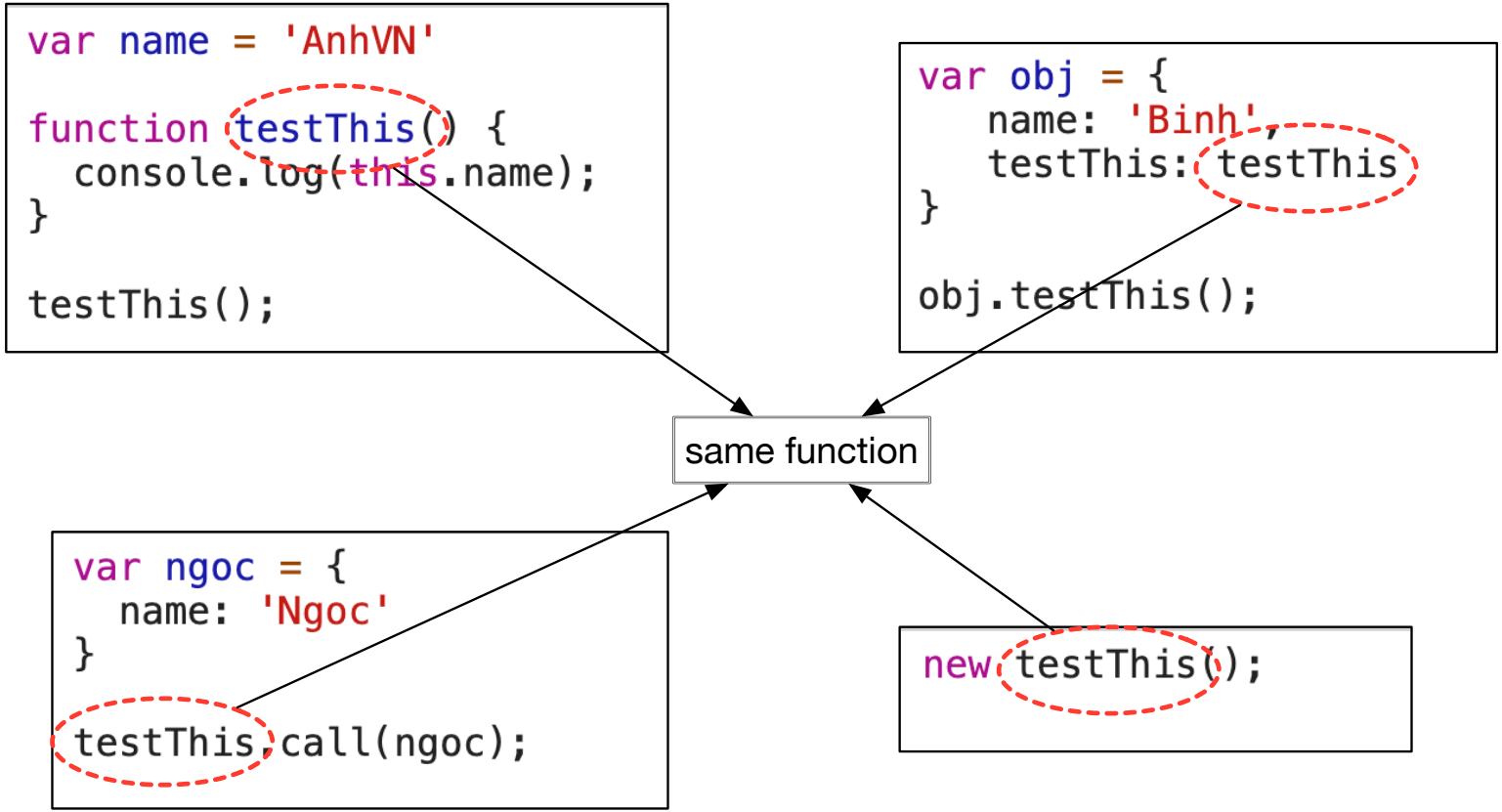
```
class Person {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log('speak' + this.name);
  }
}

var p = new Person();
// when run this inside speak refer to object p
p.speak();
```

this

- In **OOP**, `this` keyword refers to the object containing the currently-executing code

```
class Person {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log('speak' + this.name);
  }
}

var p = new Person();
// when run this inside speak refer to object p
p.speak();
```

this in JS


var name = 'AnhVN'
function testThis() {
console.log(this.name);
}
testThis();
var obj = {
name: 'Binh',
testThis: testThis
}
obj.testThis();
same function
var ngoc = {
name: 'Ngoc'
}
testThis.call(ngoc);
new testThis();

Rule for `this`

Every time you see `this` keyword in a function.
You must determine how that function is called (in 1 of 4 way above):

1. if use new then **this** refer to newly created object
2. If use call//apply then **this** refer to the 1st parameter you provided to call/apply
3. if its called as method then **this** refer to the object on left of '.'
4. normal function then **this** refer to global object (normally window in Browser)

Summary

- Understand OOP and its 4 major principles
- Able to archieve OOP in JS
- Understand Prototype inheritance
- Understand `this` keyword

Thank you!