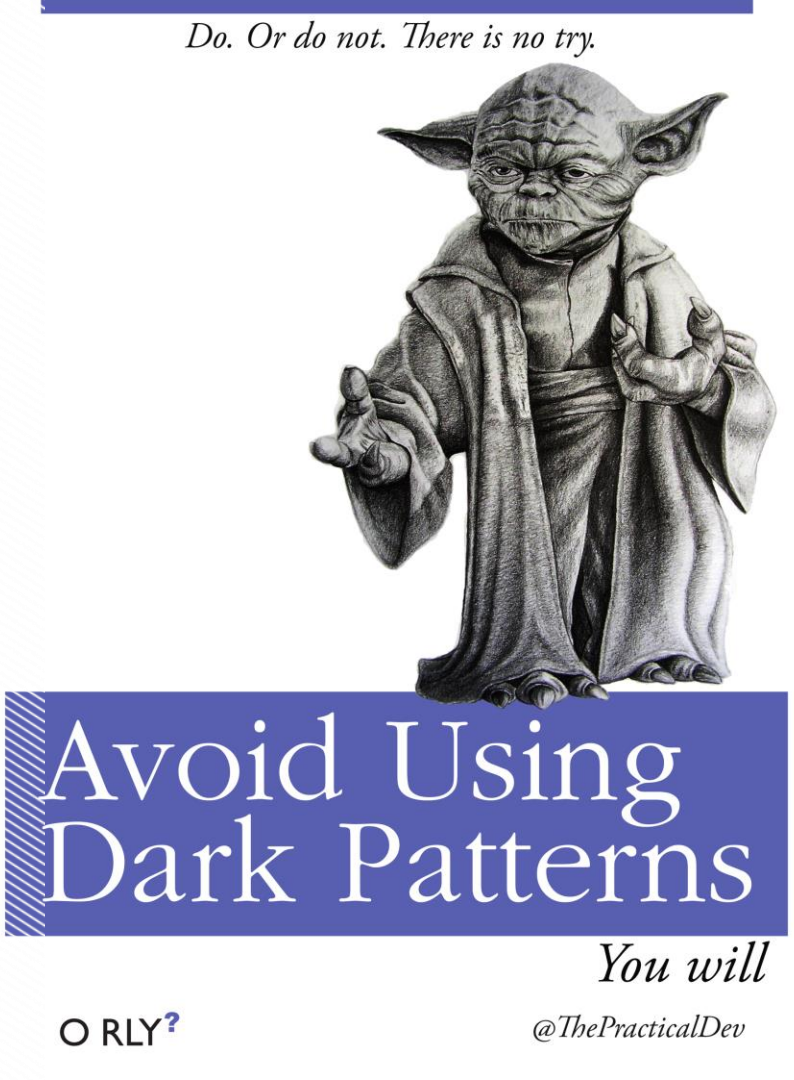
Object-oriented Programming

(or not, that is the question)

Frontend Junior Program - 2022



OOP MENU – PLEASE SELECT OR CLICK TO *START LECTURE*

START LECTURE

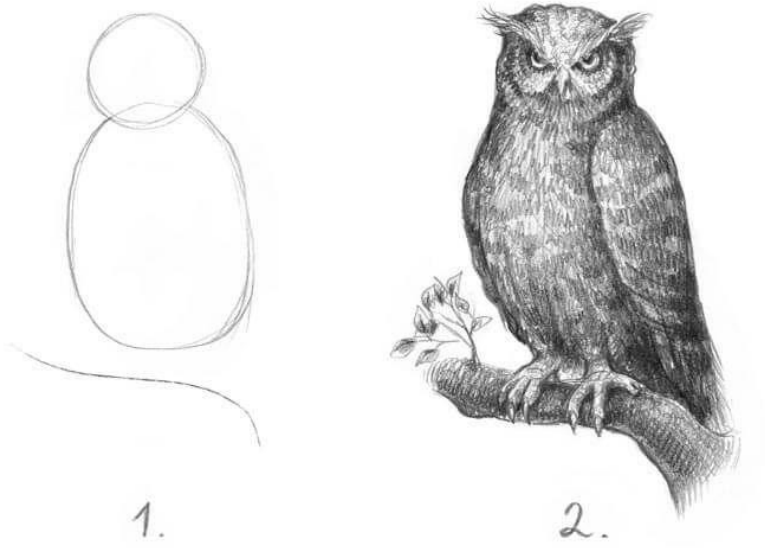
Introduction

How to ~~draw~~ write programs in OOP?

Well, we could follow the old and trustworthy approach…

… or, we could figure it out, what is OOP at all?

While this is a really complex topic, at the end of this lecture you should be prepared not just for interview questions, but probably you can draw the owl yourself!



step 1.

check OOP examples in any programming book…

step 2.

… then add some details in the production code!

Objects can be considered as
data and behavior enclosed together

Objects are analogous to biological cells.

They form a network (another analogy for objects is the computers of a network) and communicates with messages with each other.

That's all: object is merely a concept – objects in any languages are just an implementation of this idea.

a tree traversal object

Object = data + behavior

data (state) →

behavior →

```
const Animal = function(sound) {
  let _sound = sound;

  function speak() {
    setTimeout(() => console.log(_sound), 0);
  }

  return {
    speak
  };
};
```

is this a *pure function*?

Zorg is trying to report on why the unit tests fail again – without any reasonable cause

The subtle detail there about that method is not being pure means a lot when unit tests are considered. And unit tests are always considered. Theoretically, however, it is possible to write always green and therefore perfectly pointless unit tests, but even Zorg is not that evil – you should not be, either.

SUPERGREEN – THE CLASSIC OOP

Supergreen – classic OOP

Let's start with the most useful knowledge you'd need in interviews: the OOP concepts

1. *Encapsulation*
2. *Inheritance*
3. *Polymorphism*

If you open any book *(the good ones, with the owl examples)*, the chance that you'll run into these terms, is 100%. So, let's take aside these first*!

OOP development is always SOLID

no kidding, you should read this

** from this start, as you may already guess, these are not that important for us in JavaScript – still, you will be asked!* ¯_(ツ)_/¯

Encapsulation

There are many different definitions for encapsulation:

a) enclosing and decoupling the data from outside
(protecting from accessing and modifying)

b) bundling the data and the behavior together
(variables and methods)

c) a + b together

← this part requires private fields

objects could be lazy

But decoupling means much more than simply separating the data (and methods) from outside

Let's say an object receives a message that sender would like to access / modify some data: the sender can ask this, but how and when the object reacts to it, it is the concern of the object.

Encapsulation

Encapsulation is a concept

OOP is not about the implementation details. Here, we implemented proper encapsulation with simply a closure*.

No ES6 classes, no constructor functions were involved.

* * *

Encapsulation, however, is a general concept in programming and is not unique for OOP at all.

Any module system, by definition, is an implementation of the encapsulation.

an object *"class"*: data + behavior

data is private and decoupled

it is the concern of the object how and when to react

object *"instances"*

we reveal *speak* as a public method

```
> const Animal = function(sound) {
    let _sound = sound;

    function speak() {
      setTimeout(() => console.log(_sound), 0);
    }

    return {
      speak
    };
  };

  const owl = Animal("HOOT!");
  const pig = Animal("OINK!");

  owl.speak();
  pig.speak();
<- undefined
  HOOT!
  OINK!
```

* *this is the Reveal Module Pattern*

the Reveal Module Pattern is Zorg's favourite question in interviews – look at those eyes, I'd learn this pattern…

Inheritance

With inheritance we can extend the original object with behavior and with data.

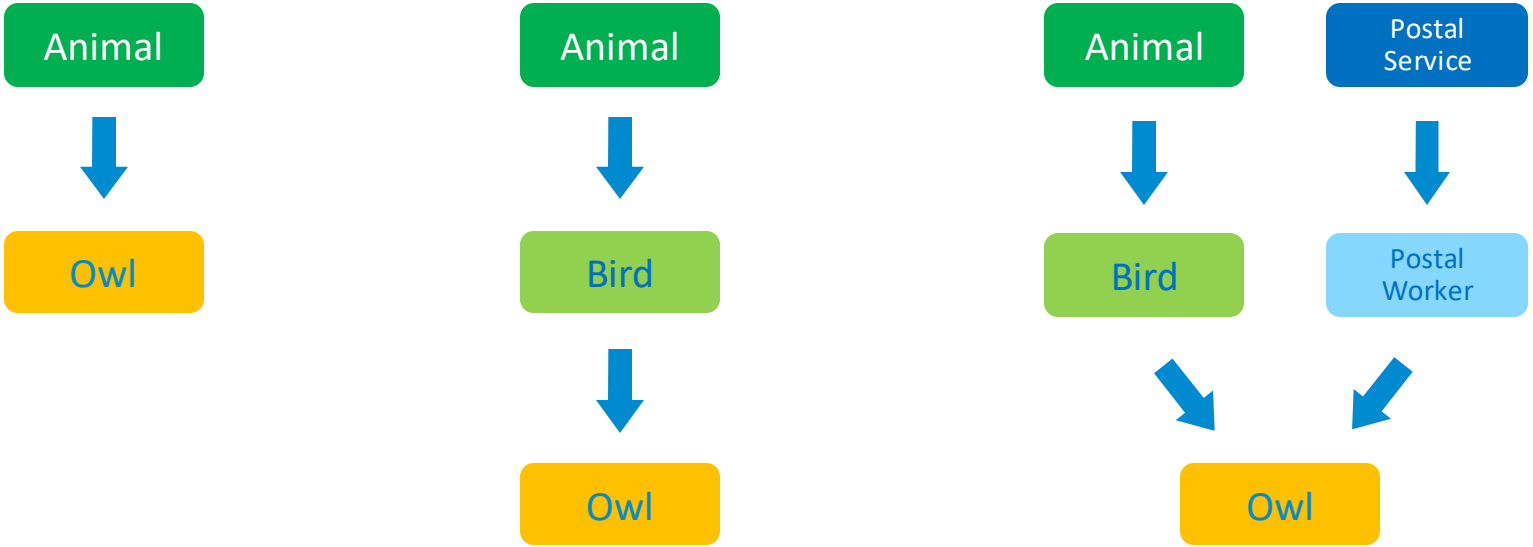
Again, this is *not* the prototypal inheritance! It is just a simple implementation of a concept.

owl inherit everything what Animal has →

now our owl can fly as well →

```
> const Animal = function(sound) {
    let _sound = sound;

    function speak() {
      setTimeout(() => console.log(_sound), 0);
    }

    return {
      speak
    };
  };

  const owl = Animal("HOOT!");
  const pig = Animal("OINK!");

  owl.fly = function() {
    console.log("🦅");
  };

  owl.speak();
  owl.fly();
  🦅
< undefined
  HOOT!
```

Inheritance



single inheritance

multi-level inheritance

multiple inheritance

a developer realising that the project's codebase relies on multi-level inheritance heavily

Remember: as *peace*, inheritance is a concept merely. Whether it is built-in and in what form to the language is another question. While multiple inheritance is not part of the JavaScript, it can be implemented in many different ways. Should be?

Let's ask this way: should you rely on inheritance at all? Or maybe composition would be better?

Polymorphism

Polymorphism is when ~~you believe something about a method, but that will act completely differently~~ a single entity could have different forms

(poly = many, morph = form)

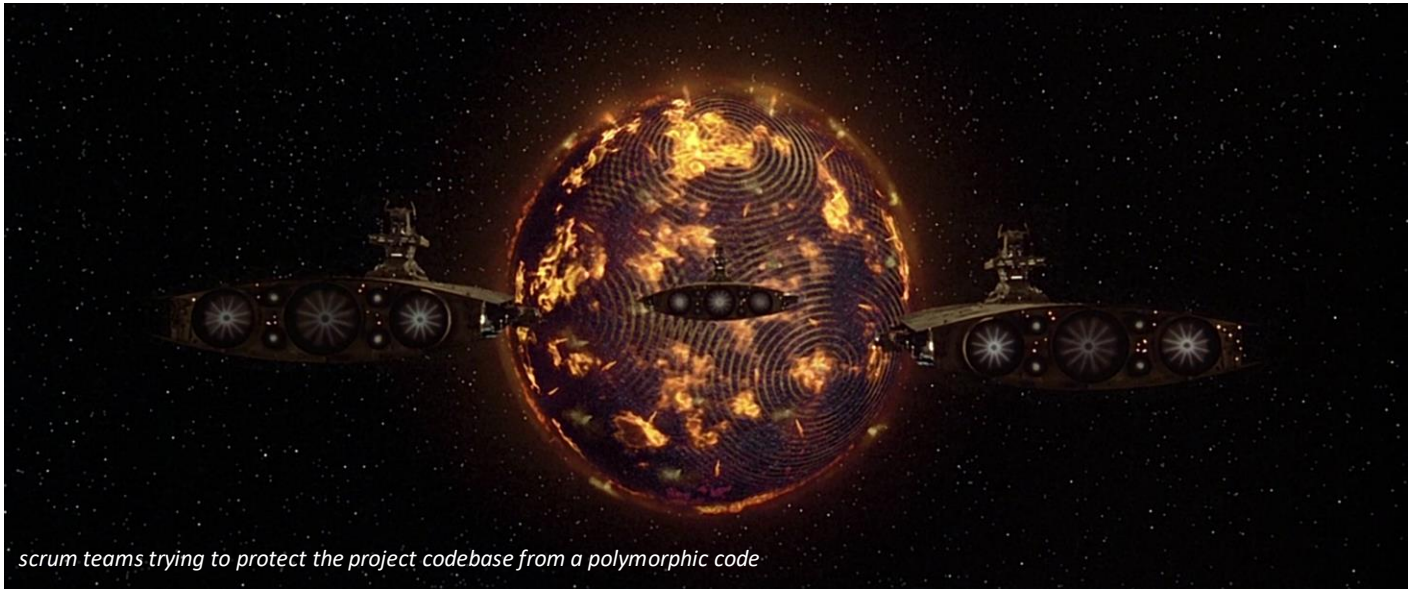
Polymorphism is a powerful concept, however, as the entities could be overridden in run-time, it could lead to a kind of guesswork.

That being said, as a concept, JavaScript relies on it, think about the "+" operator. It behaves completely differently with numbers and strings (operator overloading).

```
> const Animal = function(sound) {
    let _sound = sound;

    function speak() {
      setTimeout(() => console.log(_sound), 0);
    }

    return {
      speak
    };
  };

  const owl = Animal("HOOT!");

  owl.speak = function() {
    console.log("I have no words...");
  };

  owl.speak();
  I have no words...
```

we override the original behavior

no wonder that the owl remains silent: new method cannot access the *_sound* state



Polymorphism, however, is not a problem, if the code does not use inheritance.

Supergreen, the classic OOP – wrap-up

The classic OOP concepts (encapsulation, inheritance and polymorphism) are a bit of strange beings

In one hand, these concepts are not unique to OOP, on the other hand, you'd need to learn and utilize a handful set of principles* as a safeguard, to that extent, where (inheritance being eliminated) there nothing remains from the original concepts and objects will be used simply containers (modules) for encapsulating data.

Let's move to the next part, though, and see why it was invented in the first place?

programming books explaining the benefits of OOP

** like SOLID, composition over inheritance, even the design patterns were introduced to provide workarounds for OOP issues*

DIVINE LIGHT – THE ORIGINS OF OOP

Divine Light – the origins of OOP

This will be short – as nobody thinks about OOP this way…

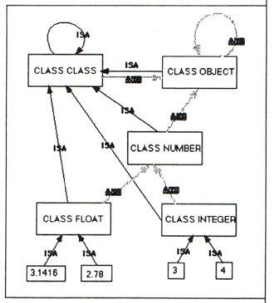
except of the inventor of the term: *Object-Oriented Programming.*

The term: OOP is coined to Alan Kay, however, the original concept was not about inheritance (you see?), but about the messaging*:

> *"OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things."*

```
Class new title: 'Window';
  fields: 'frame';
  asFollows!

This is a superclass for presenting windows on the display. It holds control until the stylus is depressed
outside. While it holds control, it distributes messages to itself based on user actions.

Scheduling
startup                                          : means keyword whose following
  [frame contains: stylus =>                       expression will be sent "by value"
    self enter.
    repeat:                                      : means keyword whose following
      [frame contains: stylus loc =>               expression will be sent "by name"
        [keybard active => [self keyboard]
        stylus down => [self pendown]]
      self outside => []
      stylus down => [^self leave]]]
  ^false]                                        ^ means "send back"
Default Event Responses                          => means "then"
enter [self show]
leave
outside [^false]
pendown
keyboard [keyboard next. frame flash]
Image
show
  [frame outline: 2.
  titleframe put: self title at: frame origin + title loc.
  titleframe complement]
... etc.
```

FIGURE 11.54 Smalltalk-76 Metaphysics




```
Class new title: 'DocWindow';
  subclassof: Window;
  fields: 'document scrollbar edit Menu';
  asFollows!

User events are passed on to the document while the window
is active. If the stylus goes out of the window, scrollbar and
the editMenu are each given a chance to gain control.

Event Responses
enter [self show.editMenu show.scrollbar
show]
leave [document hideselection.editMenu
hide.scrollbar hide]
outside
  [editMenu startup => []
  scrollbar startup => [self showDoc]
  ^flase]
pendown [document pendown]
keyboard [document keyboard]
Image                                            super means delegate message
show [super show.self showDoc]                   to next higher superclass
showDoc [doucument showin: frame at: scrollbar position]
title [^document title]
```

* *"I like to say that JavaScript is Smalltalk's revenge on the world's misunderstanding of OOP."*

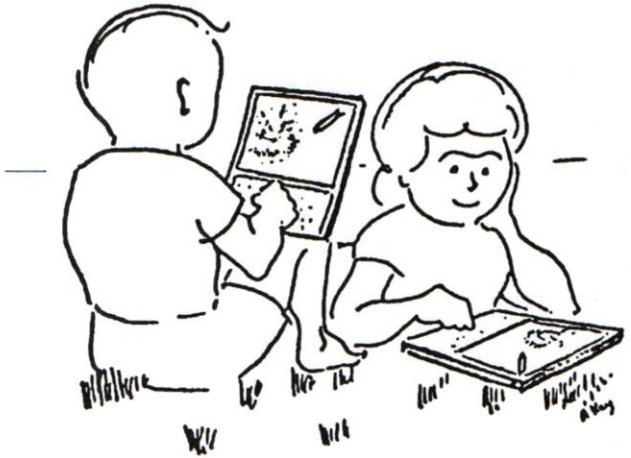
OOP – an idea for building UI

This image explains a lot

Smalltalk was originally designed for UI. And in a UI interface everything is really an object. It even makes sense to have inheritance and polymorphism (think about zillion versions of buttons), and components are communicating via events (messages).

OOP is natural in UI development and is heavily used for that nowadays – for a reason.

FIGURE 11.27 Children with Dynabooks from "A Personal Computer for Children of All Ages" [Kay 1972]



The early history of Smalltalk

this is the real area where OOP shines: on the user interface objects are conceptually real objects

DIVA – OOP IN JAVASCRIPT

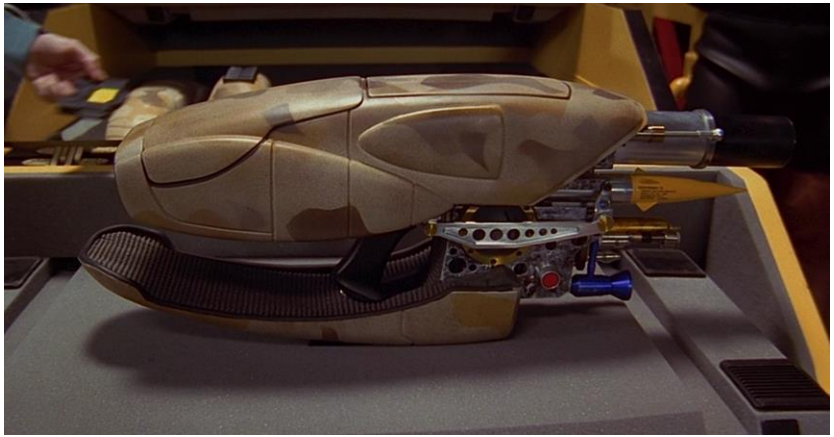
Diva – OOP in JavaScript

This is a simple field – and extremely complex at the same time

From now on we'll avoid all of the dark patters and will focus on the clean JavaScript features. Remember, all these can (and usually) used in different ways, combined with each other, so the possibilities you can go wrong are almost infinite.

However, we have to start with…

THIS



the ~~ZF-1~~ prototypal inheritance in JavaScript

This

Let's be clear: *this* is a nasty beast in JavaScript

The main problem with it that is does have different faces. A real polymorphic creature. It can have completely different meanings in different situations.

For example:

- *in Global context*
- *in Function context*
- *in Function context – strict mode*
- *in a constructor*
- *in a constructor, called without new*
- *in a constructor, called without new – strict mode*
- *in a method call*
- *in a function call (a method assigned to a variable)*
- *in Arrow functions*
- *…*

this character called *ellipsis* (almost like the rest syntax) and it is very useful to refer for an almost infinite list of cases

this – Global and Function context

in Global context
(equals to the global object)

```
> this
◂ ▶ Window {window: Window, self: Window, document:
    document, name: "", location: Location, …}
> function owl() {
    console.log(this);
  }
  owl();
                                                VM27822:2
  ▶ Window {window: Window, self: Window, document:
    document, name: "", location: Location, …}
◂ undefined
> function owl() {
    "use strict";
    console.log(this);
  }
  owl();
    undefined                                   VM27841:3
◂ undefined
```

in Function context
(equals to the global object)

in Function context - strict mode
(equals to undefined)

this – in a constructor

refers to the Object

calling the constructor without *new*

oops!

much better in strict mode

```
> function Owl() {
    console.log(this);
  }

  let owl = new Owl();
▶ Owl {}                                          VM28241:2
< undefined
> Owl();
                                                  VM28241:2
▶ Window {window: Window, self: Window, document:
  document, name: "", location: Location, …}
< undefined
> function Owl() {
    "use strict";
    console.log(this);
  }

  Owl();
  undefined                                       VM28304:3
```

this – in a method call

refers to the Object

assigning a method to a variable

this is a killer one

```
> function Owl() {
    this.speak = function() {
      console.log(this);
    }
  }

  let owl = new Owl();

  owl.speak();
▶ Owl {speak: f}                                    VM28891:3
<· undefined
> let speak = owl.speak;

  speak()

                                                    VM28891:3
▶ Window {window: Window, self: Window, document:
  document, name: "", location: Location, …}
```

this – in Arrow function

Okay, probably that's enough…

Please explore the remaining cases, there is many!

in a method call (with function) *this* refers to the Object

in a method call (with arrow function) it refers to the parent context (here: the Global)

IIFE still a function call, referring to the Global context

in an arrow function it always refers to the parent

```
> const owl = {
    speak: function() {
      console.log(this);
    },
    fly: () => {
      console.log(this);
    },
    eat: function() {
      (function() {
        console.log(this);
      })();
    },
    sleep: function() {
      (() => {
        console.log(this);
      })();
    }
  };

  owl.speak();
▶ {speak: ƒ, fly: ƒ, eat: ƒ, sleep: ƒ}                    VM29796:3
← undefined
> owl.fly();
                                                          VM29796:6
▶ Window {window: Window, self: Window, document: document, name: "", l
  ocation: Location, …}
← undefined
> owl.eat();
                                                          VM29796:10
▶ Window {window: Window, self: Window, document: document, name: "", l
  ocation: Location, …}
← undefined
> owl.sleep();
▶ {speak: ƒ, fly: ƒ, eat: ƒ, sleep: ƒ}                    VM29796:15
```

Function.prototype.bind

Bind returns a new function, where we can set the *this* as we want.

we bind the *this* back to the Object

Object

a function call = Global

whatever we bound it

```
> function Owl() {
    this.speak = function() {
      console.log(this);
    }
  }

  let owl = new Owl();
  let speak = owl.speak;
  let boundSpeak = speak.bind(owl);

  owl.speak();

  speak();

  boundSpeak();
  ▶ Owl {speak: f}                                    VM30687:3

                                                      VM30687:3
  ▶ Window {window: Window, self: Window, document:
    document, name: "", location: Location, …}

  ▶ Owl {speak: f}                                    VM30687:3

<· undefined
```

Function.prototype.call – calls with a new *this*

Call is similar to bind, but it calls the function as well

Please don't mismatch these, these are very frequently asked by Zorg!

of course, we have a third one here: Function.prototype.apply

the difference is in how these accept the arguments, please check the docs!

```
> function Owl() {
    this.speak = function() {
      console.log(this);
    }
  }

  let owl = new Owl();
  let speak = owl.speak;

  speak.call(owl);
  ▶ Owl {speak: f}
< undefined
```

How to create objects

There are many ways to create an object in JavaScript

And while the constructor function and the class declaration are fairly equivalent, the object literal is rarely used in OOP as that is a single instance.

```
> const owl = {
    sound: "HOOT!",
    speak: function() {
      console.log(this.sound);
    }
  };
<- undefined
> owl.speak();
  HOOT!
```

object literal

```
> function Owl() {
    this.sound = "HOOT!",
    this.speak = function() {
      console.log(this.sound);
    }
  }

  let owl = new Owl();
  owl.speak();
  HOOT!
```

constructor function

```
> class Owl {
    constructor() {
      this.sound = "HOOT!";
    }

    speak() {
      console.log(this.sound);
    }
  }

  let owl = new Owl();
  owl.speak();
  HOOT!
```

class declaration

Prototypal inheritance

Objects are inherited through the prototype chain

```
function Animal() {
  this.sound = "";
  this.speak = function() {
    console.log(this.sound);
  }
}

function Owl() {
  this.sound = "HOOT!";
}

Owl.prototype = new Animal();

let owl = new Owl();
owl.speak();
```

```
HOOT!
```

constructor function

```
class Animal {
  constructor() {
    this.sound = "";
  }

  speak() {
    console.log(this.sound);
  }
}

class Owl extends Animal {
  constructor() {
    super();
    this.sound = "HOOT!";
  }
}

let owl = new Owl();
owl.speak();
```

```
HOOT!
```

class declaration

MULTI-PASS – HOW TO USE OOP

We already covered everything here: in Front-end development OOP mainly used as a container for components – heavy OOP code is rarely seen here. It is used to say that it is possible to write good OOP code if you… [*and a long list of criteria presented here*], and that is true: it is *possible*…

That being said, even with simple objects you can do it catastrophically wrong, if you rely on state variables a lot. Please use pure functions and methods even in objects, it will be your *multipass for programming* – the entry point for both worlds.

edu_hu@epam.com