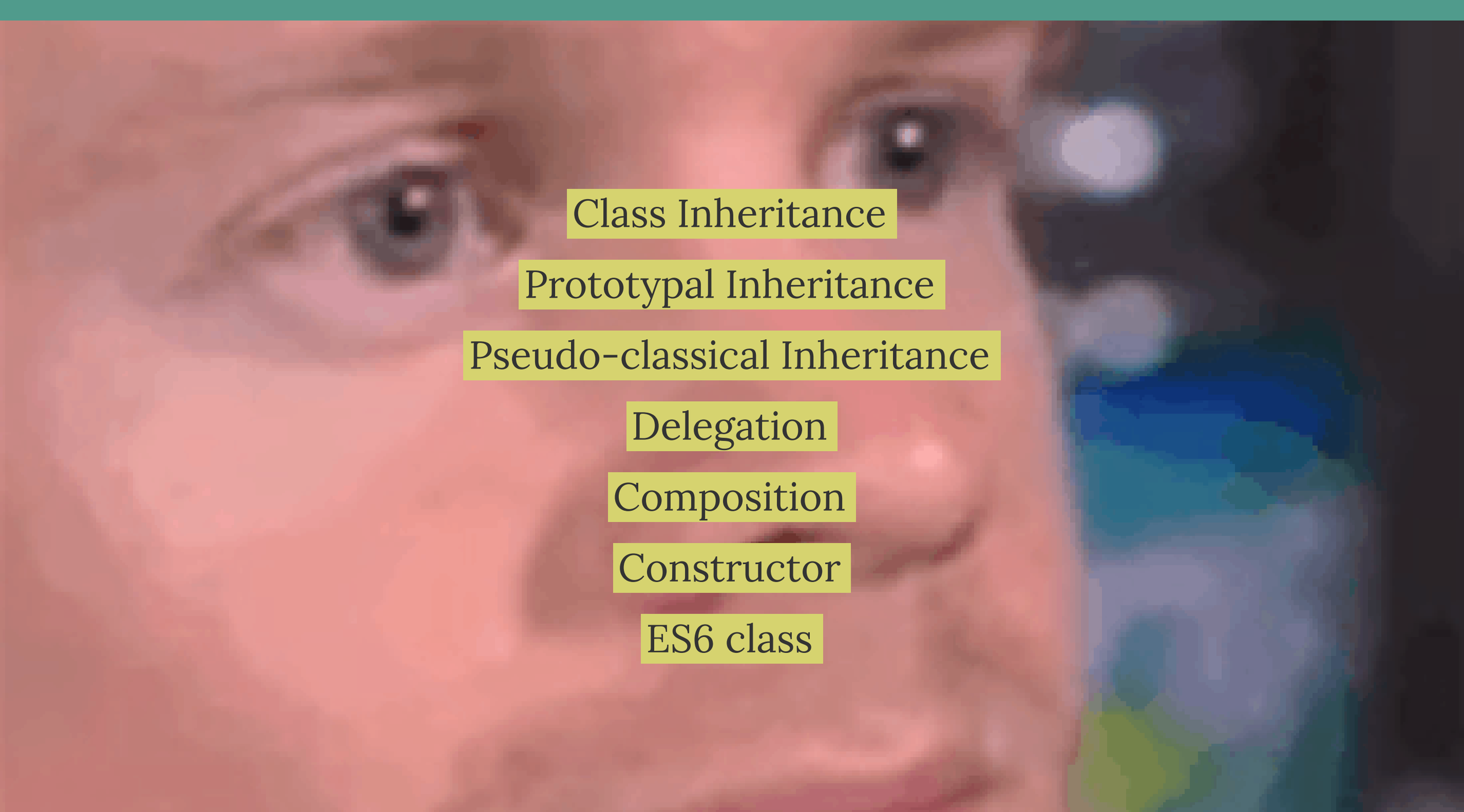
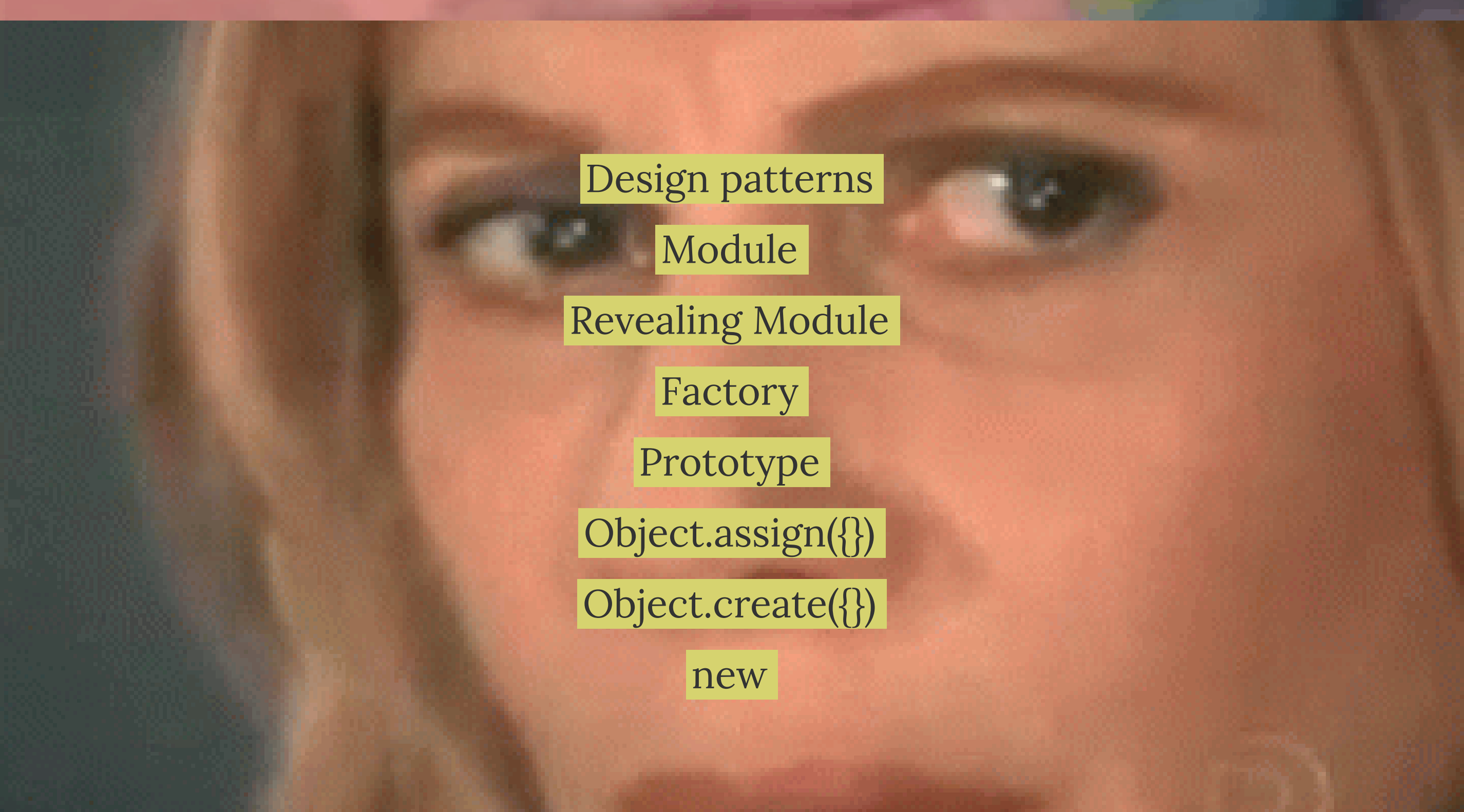
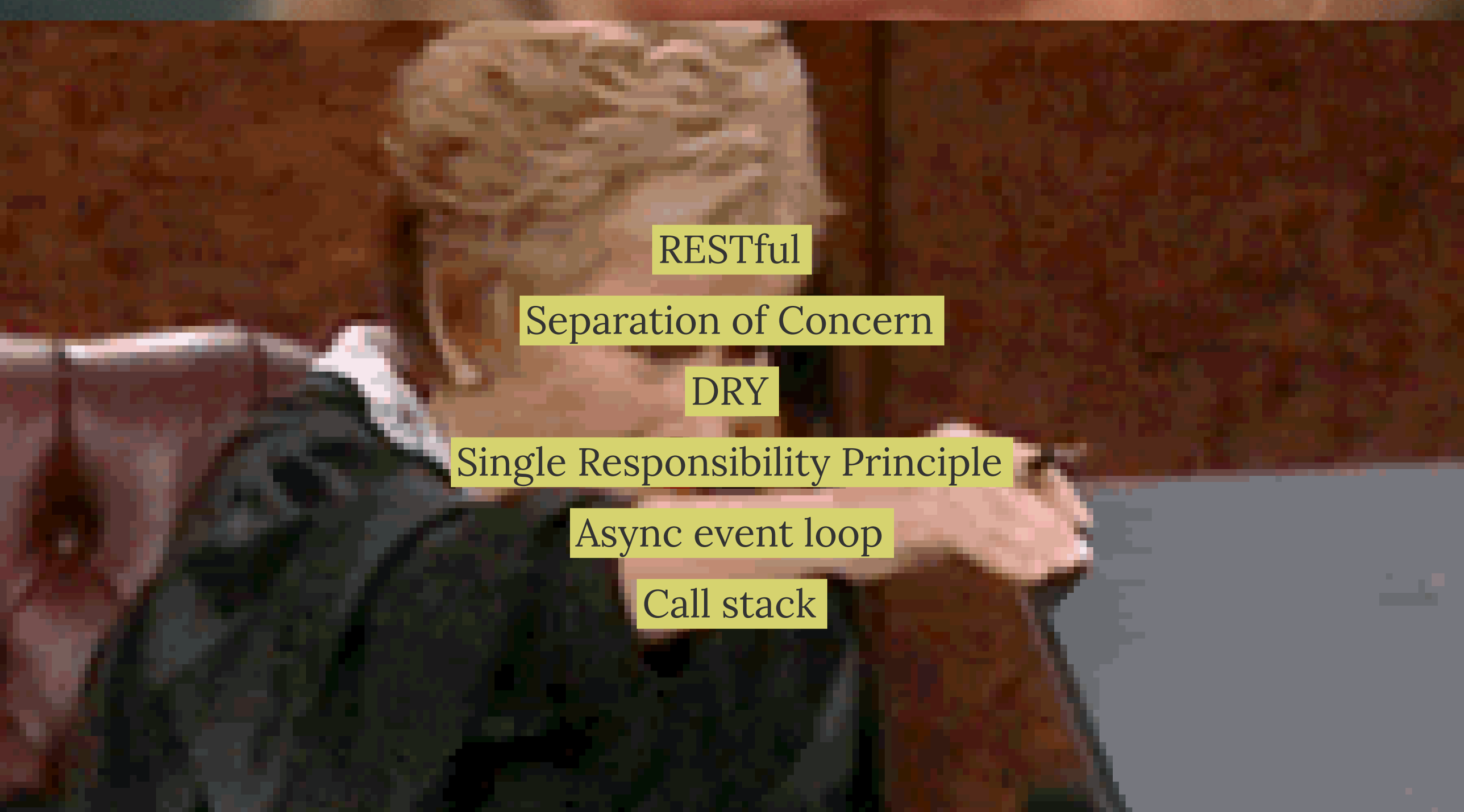
JAVASCRIPT

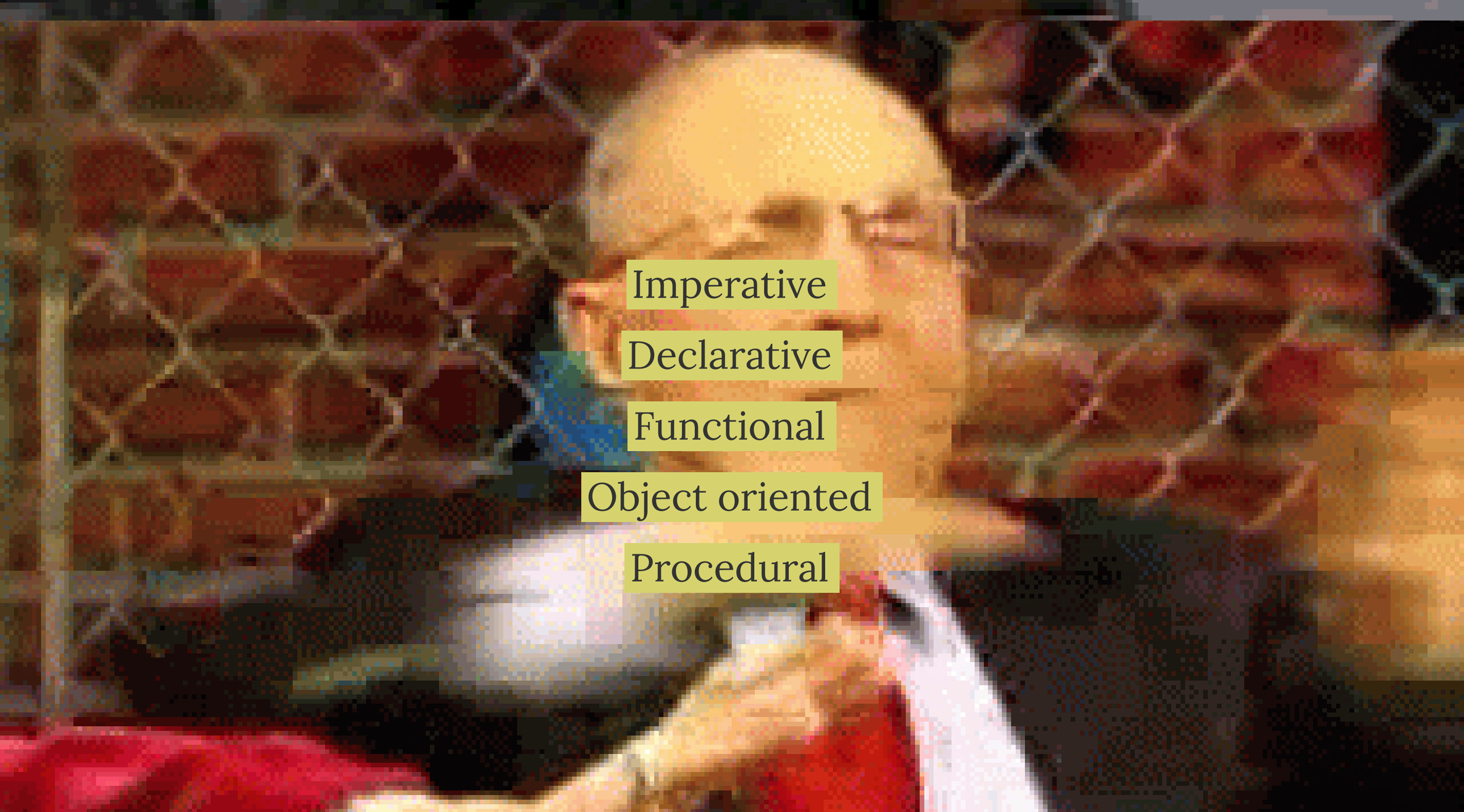
Eller: hur jag lärde mig massa konstiga termer och blev hipp

Eller: korvstoppning


Class Inheritance
Prototypal Inheritance
Pseudo-classical Inheritance
Delegation
Composition
Constructor
ES6 class


Design patterns
Module
Revealing Module
Factory
Prototype
Object.assign({})
Object.create({})
new


RESTful
Separation of Concern
DRY
Single Responsibility Principle
Async event loop
Call stack


Imperative
Declarative
Functional
Object oriented
Procedural


node
npm
gulp
webpack
yarn
bower

Intervjumaterial?

Områden att fördjupa sig i

Utgår ifrån:

10 interview questions every JavaScript Developer Should Know

Heavly opinionated

Snubben vill ju såklart kränga böcker och kurser

TABLE OF CONTENT

10 Interview questions

Funfunfunctions - Composition > Inheritance @ YouTube

Dave Atchley - Understanding Prototypes, Delegation & Composition

Wiki: Comparison of Programming Paradigms

Making Sense of Front end build tools

Todomvc.com

Vanilla MVC Implementation

PROGRAMMING PARADIGMS

Programming paradigms *are a way to classify programming languages according to the style of computer programming*

Ref: Wiki - Programming Paradigm

Imperative programming

Imperativ programmering fokuserar på HUR programmeringen sker.

Vi definierar exakt vad som ska hända och när det ska hända.

Ref: Wiki - Imperative

```
sum = 0;
for(let i = 0; array.length; i++){
  sum += array[i];
}
console.log(sum)
```

Vi säger exakt vilka steg som ska tas för att loopa igenom arrayen.

Ta värde på index i, lägg till det i sum etc.

Declarative programming

Deklarativ programmering fokuserar på VAD koden ska göra.

Vi beskriver övergripande vad som ska hända, men inte specifikt hur programmet ska göra det.

Ref: Wiki - Declarative Programming

```
array.map(movie => movie.title);

array.reduce((p,c) => p + c);
```

Fungerar som en loop, men hur den egentligen loopar vet vi inte. Vi säger bara vad vi ska göra.

Functional programming

Functional programming (often abbreviated FP) is the process of building software by composing pure functions, avoiding shared state, mutable data, and side-effects. *Functional programming is declarative rather than imperative, and application state flows through pure functions*

Ref: What is Functional Programming

```
function calculate(a,b){
  return a + b;
}
```

Gör inget annat än att ta in två värden, beräkna och skicka ut ett nytt skapat värde.

Inga ändringar utanför funktionens scope, inga sidoeffekter. Vi muterar heller inte data som skickas in.

```
var globalArray = [];

function calculate(a,b){
  var c = a + b;
  //Side-effect, manipulate data outside of scope
  globalArray.push(c);
  return c;
}
```

Inte längre en ren funktion.

```
function calculate(a,b){
  var c = a + b;
  //Side-effect
  $('#id').html(c);

  return c;
}
```

Inte längre en ren funktion.

Object Oriented programming

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects" *, which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.*

Oftast när vi ser nyckelordet Class har vi ett Object Oriented Language

Instansierar objekt baserade på klasser. En majoritet av de stora språken har detta upplägg.

Och JavaScript halvt.

OCH JAVASCRIPT ÄR VADÅ?

Javascript kan skrivas funktionellt, deklarativt, imperativt samt objektorienterat.

Det ena paradigment utesluter inte det andra.

Funktionella språk: Haskell, Lisp, F.

Objektorienterade: C#, Java, PHP

WIKI: COMPARISON OF PROGRAMMING PARADIGMS

INHERITANCE

composition & delegation

INHERITANCE

Man brukar övergripande prata om arv

Objekt ärver egenskaper från andra objekt.

Jag fastnar ofta i att jag tycker OOP-tänket är lättare eftersom jag började med ett OOP-språk. Men JavaScript är inte OOP i grunden.

Basic Inheritance in JavaScript

Class Inheritance : A class is like a blueprint—a description of the object to be created. Classes inherit from classes and create subclass relationships: hierarchical class taxonomies.

Prototypal Inheritance : A prototype is a working object instance. Objects inherit directly from other objects.

What's the Difference Between Class & Prototypal Inheritance?

Class Inheritance: Här har du en ritning på ett hus, skapa det här huset utifrån ritningen

Prototypal Inheritance: Här har du ett hus, skapa ett nästan likadant hus baserat på det här huset.

ES6 Class

```
class Animal{
  constructor(){}
}

class Cat extends Animal{
  constructor(){}
}
```

I OOP-språk så skapar vi instanser av en klass, objekt som läggs i minnet när det skapas.

I JavaScript skapar vi egentligen inte instanser. Detta imiterar bara OOP.

```
function Animal(){}

function Cat(){}

Cat.prototype = new Animal();
```

Enligt många så imiterar vi här OOP och dess klass-struktur men vi implementerar strukturen med hjälp av Prototyper

Big nono ¯_(ツ)_/¯

VARFÖR INTE OOP?

Skapar hierarkier som är svåra att omstrukturera om koden måste ändras

Hierarkierna är tight kopplade, de blir beroende av varandra.

Lätt att en klass får för många ansvarsområden. Svårt att strukturera vad klassen ska göra och inte göra.

Skapar en struktur som språket inte är uppbyggt av.

Ref: Eric Elliott

I Am Devloper
@iamdevloper

Following

manager: we need to design an admin system for a veterinary centre

dev: ok, this is it, remember your training

class Dog extends Animal {}

RETWEETS 6,309 LIKES 8,046

3:35 PM - 4 May 2016

58 6.3K 8.0K

DELEGATION & COMPOSITION

Delegation - when you access a property on your object, if it's not directly defined on it, javascript will search the prototype chain and delegate that behavior to the first object it finds that defines that property.

Dave Atchley - Understanding Prototypes, Delegation & Composition

Kort sagt: Prototype chain

```
function Foo(){}
Foo.prototype.x = "x";

function Bar(){}
Bar.prototype = new Foo();

Bar.x === 'x' //Has the prop
```

Vi kopplar egenskaperna från Foo till Bar

INHERITANCE

```
class Animal{
  constructor(){}
}

class Cat extends Animal{
  constructor(){}
}
```

Svårt att förutse vilka egenskaper varje superklass ska ha. Vad som ska vara gemensamt.

Pseudo-classical Inheritance

```
function Animal(){}

function Cat(){}

Cat.prototype = new Animal();
```

Funfunfunctions - Composition > Inheritance @ YouTube

COMPOSITION

Inheritance skapar **is-a**: Cat is-a(n) Animal

Composition skapar has-a: Cat has-a tail

In other words, use can-do, has-a, *or* uses-a *relationships instead of* ***is-a*** *relationships.*

Ref: Eric Elliott

DESIGN PRINCIPLES

DRY

Don't Repeat Yourself

Skapa kod som kan återanvändas.

Var lat, programmerare ska vara lata. På ett bra sätt.

Upprepa dig så lite som möjligt.

Ref: Wiki - DRY

SINGLE RESPONSIBILITY PRINCIPLE

Every function you write should do exactly one thing. It should have one clearly defined goal.

Ref: Wiki - SRP

SRP

Vad som definieras som one thing är en tolkningsfråga.

```
function calculateTax(money){
  return money * 0.3;
}
```

En ren funktion som gör just en grej

```
function appendHtml(data){
  let el = document.getElementById('list');
  let html = '';
  for(let thing in data){
    html += `<li> ${thing.name} <li>`;
  }
  list.innerHTML = html;
}
```

Gör fortfarande bara en grej

Uppdaterar UI:t

```
function appendHtml(data){
  let el = document.getElementById('list');
  let html = '';
  for(let thing in data){
    html += `<li> ${thing.name} <li>`;
  }
  list.innerHTML = html;
  //ANOTHER THING???
  globalVariable = thing.id;
}
```

SEPARATION OF CONCERNS

Vad är en concern?

Business logic är en viss concern

Business logic == koden bakom

UI/Interface är en concern

Ref: Wiki - SOC

PRINCIPLE OF LEAST PRIVILEGE

For example, a user account for the sole purpose of creating backups does not need to install software: hence, it has rights only to run backup and backup-related applications.

Varför ska en funktion ha tillgång till en variabel som den inte behöver?
(Undvik global scope)

Ref: Wiki - Principle of Least Privilege

```
//Does code outside of Module need
//access to 'Shh'?
const Module = (function(){
  var private = "Shh";
  return {
    private: private
  }
})();
```

NÄR SKA JAG ANVÄNDA VAD?

Nästan alla designprinciper går in i varandra.

Jobbar man utifrån Separation of Concerns och Single Responsibility får man oftast DRY kod.

Jobbar man funktionellt så borde vissa designprinciper komma typ automatiskt

TOOLS YOU FOOLS

MAKING SENSE OF FRONT END BUILD TOOLS @ MEDIUM

BUILD TOOLS

Task Runners: Gulp, Grunt etc.

Module bundler: Webpack, systemjs

BUILD TOOLS

- Automatisera
- Minifera
- Transpilera (Omvandla kod)
- Linta kod (kolla efter fel)
- Döpa om och flytta filer
- Ladda om efter ändringar

Oberoende av verktyg syftar alla på att lösa dessa problem.

```
//export.js
export const sum = ( a, b ) => a+b;

//main.js
import {sum} from './export.js';
```

Måste skötas med Browserify

Detta är webpacks grej

Oavsett verktyg är målet

Automatisera utvecklingsprocessen

&

Göra vår kod produktionsfärdig

Inget är det bästa, välj något som du förstår och är bekväm med.

- `dist (public)`
- `src`
- `index.html`
- `.gitignore`

Våra filer i `src` ska slutligen konverteras till produktionsfärdig kod i `dist`

PACKAGE MANAGERS

npm: följer med node. Uppgift att installera dessa verktyg.

Yarn: samma som npm fast snabbare.

Bower: Pakethanterare för frontend-paket.

Yeoman: Scaffolding. Skapa strukturer för projekt. Lite som att klona ner en boilerplate från GitHub.

npm löser de flesta av dina problem. Allt annat är bara fluff.

WHAT'S NEXT?

- Prototypes
- Context
- Async Patterns
- Scope
- ES6 => beyond
- Performance

FRAMEWORKS

PROBLEM MED VANILLA

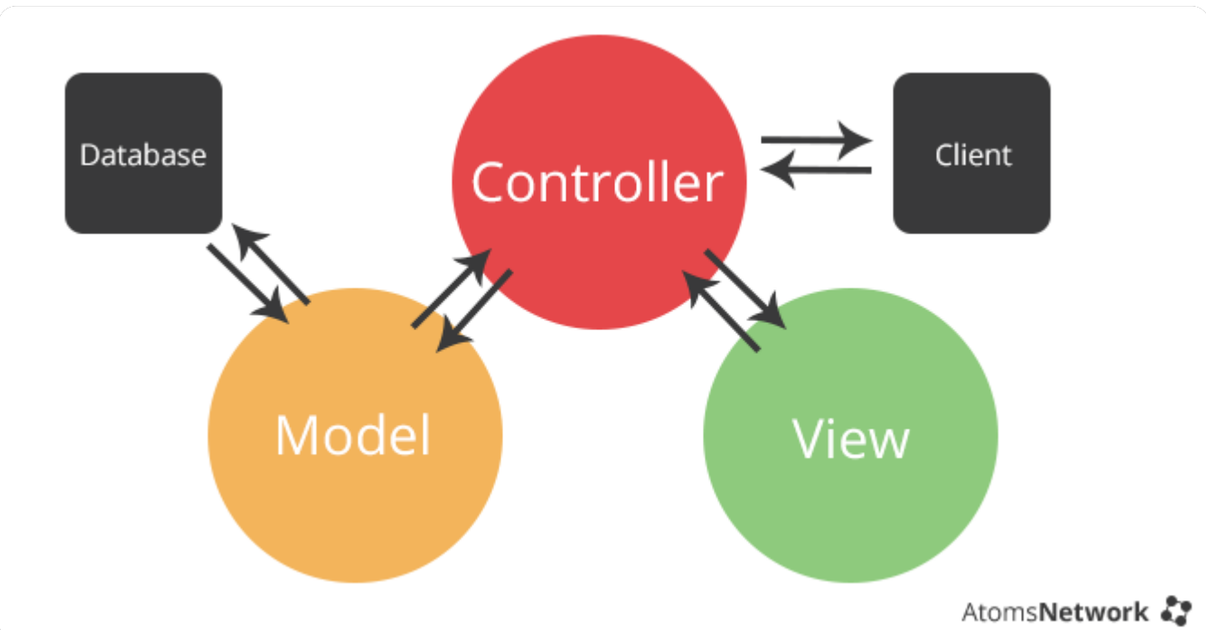
Omständig DOM-hantering

Omständig data-binding

Implementera MV* ?

MV*-RAMVERK

MVW - Model-View-Whatever

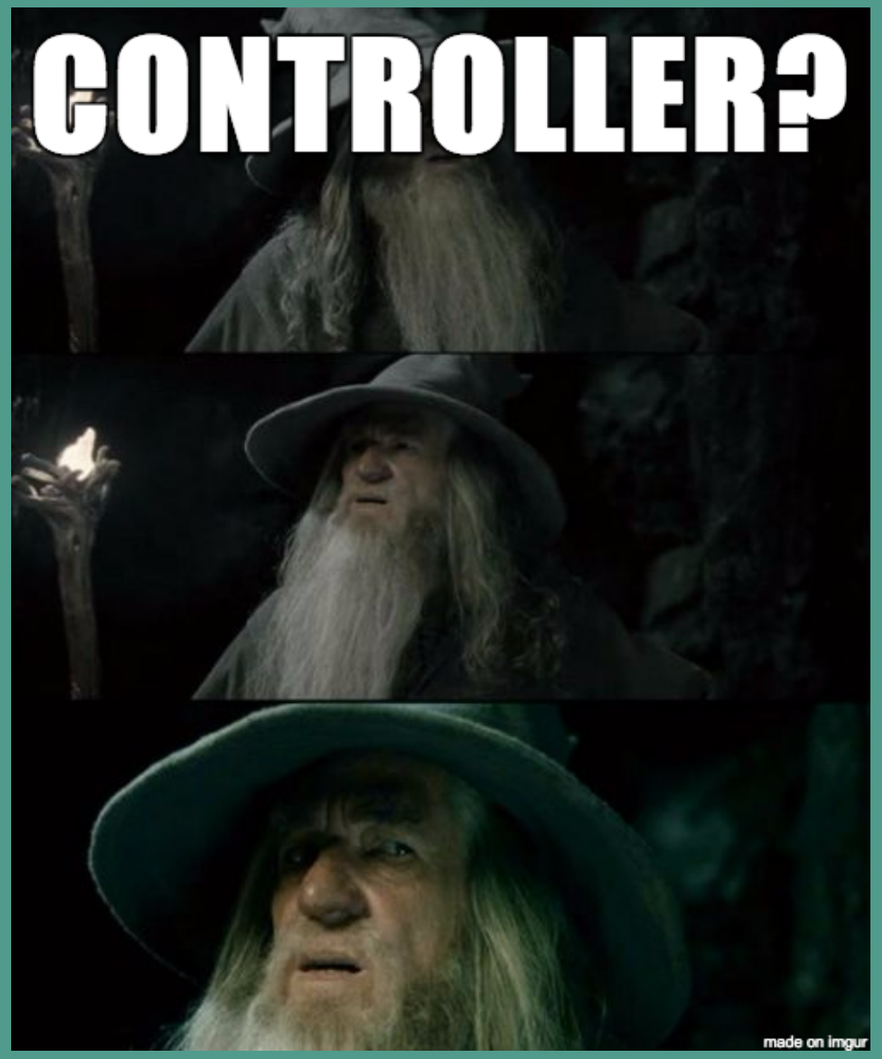


Separation of Concerns

"Skilj på de logiska funktionerna från DOM-manipulationen"

Skilj model från view

Skilj logiken (model) från det vi ser utåt (view)



Controllern är inte alltid helt separerad. I er Movie Database kallar DOM-funktionerna på de funktioner som manipulerar arrayen. View kallar på model, ingen explicit controller.

Controller kan i många fall "skippas"

React är enbart View

Vue är MVVM: Model-View-ViewModel

Angular är MVVM: Model-View-Whatever

Vanilla är ingenting, du kan skapa MV*-mönster men det är upp till dig.

Samma med jQuery, bara ett bibliotek. Du kan dock skapa ditt eget MV*-mönster med jQuery.

TODOMVC.COM

En todolista skapad i de flesta ramverk som finns med tillhörande källkod. Exempel på hur man kan göra.