Introduction to

JavaScript

the programming language of the Web

User-Defined Objects

OOP

Object-oriented programming (OOP)
- Creating reusable software objects

Object
- Programming code and data that can be treated as an individual unit or component

Data
- Information contained within variables

Object-oriented principles
- Encapsulation
- Inheritance
- Polymorphism

OOP fundamentals

principles of Object-Oriented Programming

- Encapsulation
- Inheritance
- Polymorphism

🤔 *"How do we actually design classes? How do we model real-world data into classes?"*

JS

Encapsulation

NOT accessible from outside the class!

STILL accessible from within the class!

STILL accessible from within the class!

NOT accessible from outside the class!

```
User {
  user
  private password
  private email

  login(word) {
    this.password === word
  }
  comment(text) {
    this.checkSPAM(text)
  }
  private checkSPAM(text) {
    // Verify logic
  }
}
```

Again, NOT actually JavaScript syntax (the `private` keyword doesn't exist)

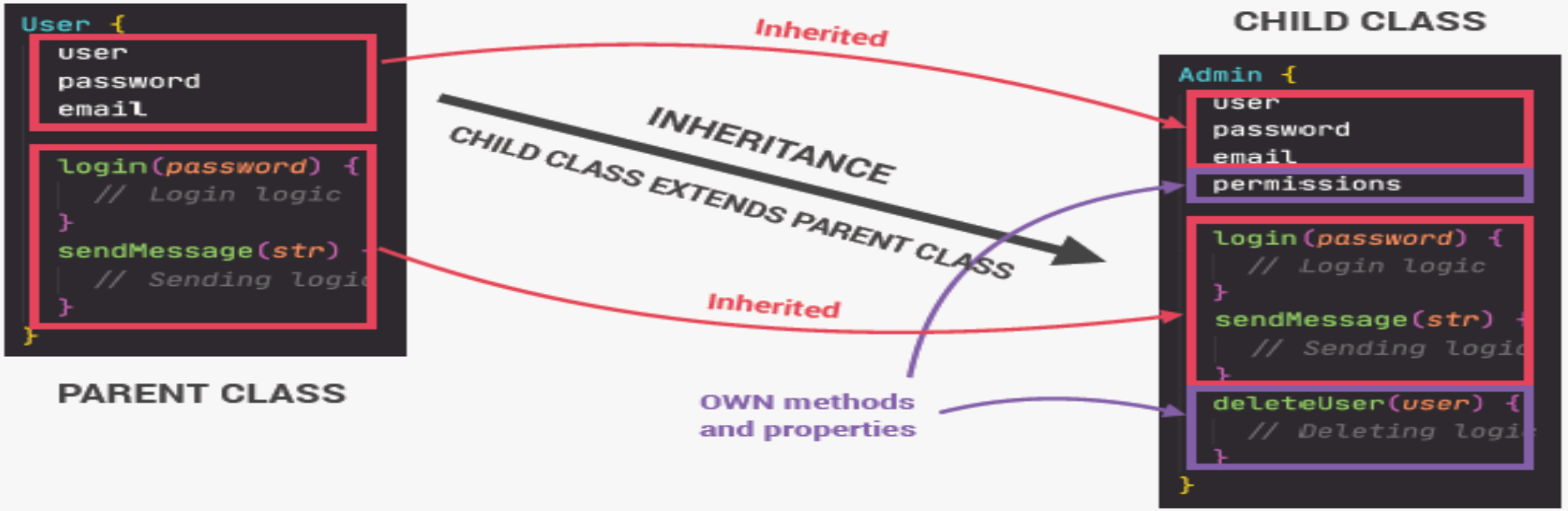
WHY?

- 👉 Prevents external code from accidentally manipulating internal properties/state
- 👉 Allows to change internal implementation without the risk of breaking external code

👉 **Encapsulation:** Keeping properties and methods **private** inside the class, so they are **not accessible from outside the class**. Some methods can be **exposed** as a public interface (API).

JS

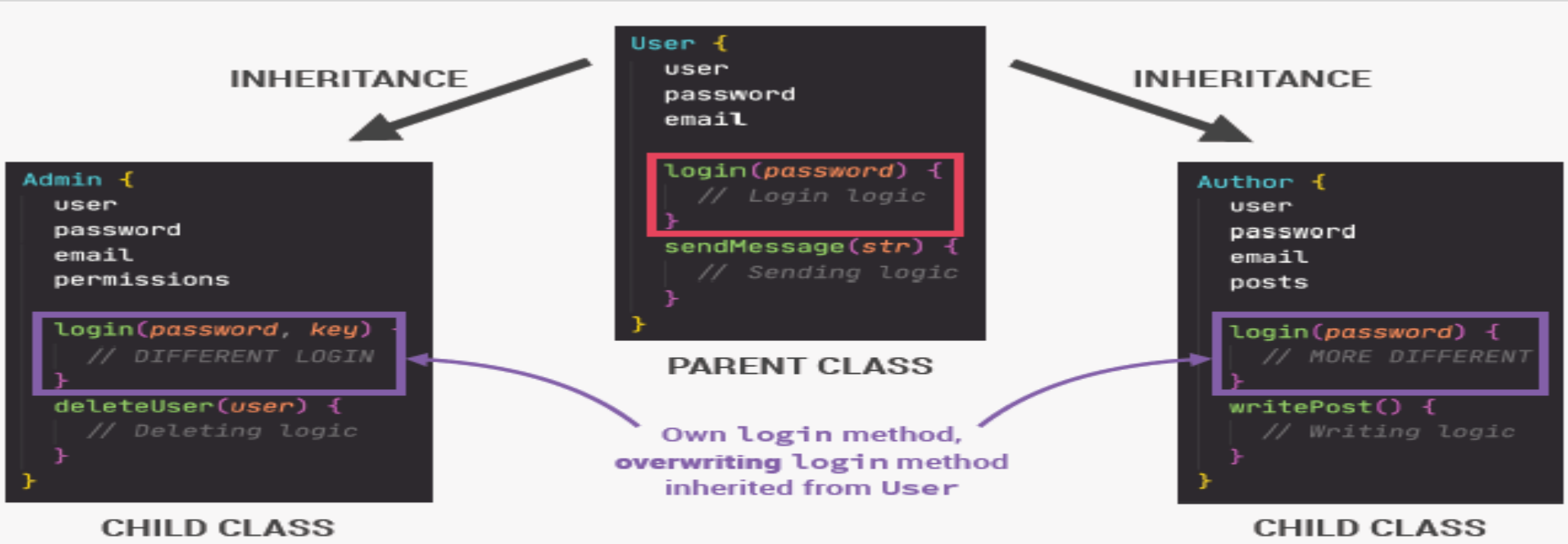
Inheritance



👉 **Inheritance:** Making all properties and methods of a certain class **available to a child class**, forming a hierarchical relationship between classes. This allows us to **reuse common logic** and to model real-world relationships.

JS

Polymorphism



👉 **Polymorphism:** A child class can **overwrite** a method it inherited from a parent class [it's more complex that that, but enough for our purposes].

JS

Instantiating an object

Creating an object from existing class

- Using **new** Operator

```
var today = new Date();
var objname = new Object();
```

To create an array you have two ways

```
var myarr = new Array[3];
//Or//both create array object
var myarr = [1,2,3];
```

From Arrays To Objects

In Arrays :

```
var myarr=[1,2,3];                    //using [] for array
```

In Objects :

```
var myobj={id:10};                    //using {} for object
```

This is an object that has a property called id with value = 10.

```
var a={};                          //a is an empty object
```

Encapsulation

Each object contain Properties & Methods to access these Properties.

```
var Car = { model: 'Toyota',//object has properties
    color: 'Red',
    move: function() {...}
    beep: function(r){ alert(r) }   //and methods
};
```

Accessing Object Members

There are two ways to access object members

- Using [] Square brackets.

```
Car['model'];              //get value from property
Car['color'] = "blue";     //set value to property
Car['move']();             //call a function
```

- Using dot operator.

```
Car.color="black";         //like C++
Car.move();
```

Altering Object Members

Because JavaScript code is parsed not compiled you can add or remove properties of the object.

- Add property to the object.

```
Car.motor="1300 cc";    //adding motor to Car
```

- Remove property from the object.

```
delete Car.model;          //deleting model attribute
Car.model;                 //"undefined"
```

typeof Operator

Now we have the object Car but from which class we instantiated this object.

```
typeof Car;          //Object
```

Each object has a property called **Constructor**

```
Car.constructor;     // Object()
```

The object created using { } it's constructor is **Object ( )**

Classes in JavaScript

- JavaScript doesn't support the notion of *classes as typical OOP languages* do.

- In JavaScript, you **create *functions that can behave*** *just* **like classes** called **Constructor Function.**

- For example, you can call a function, or you can create an instance of that function supplying those parameters.

Constructor Function

```
function Human()
{
    this.name;                    // this refers to the caller object
    this.age;
    this.sayName=function()      // sayName function in human
    {alert(this.name);}
}
```

To take an object from Human

```
var obj = new Human();
obj.name="Mohamed";
obj.age=22;
obj.sayName();                    // alert Mohamed
```

Global Object

If we called Human function without new operator the **this** refers to the global object **Window**

When you say this.age you add age property to the window object.

```
Human();
Window.age=30; //window has no age property
```

C++ & JavaScript Classes

```
Class Table
{
    public:
    int rows,cols;
    Table(int rows,int cols)
    {
        this.rows = rows;
        this.cols = cols;
    }
    int getCellCount()
    {
        return rows * cols;
    }
};
```

```
function Table(rows,cols)
{
    //constructor
    this.rows=rows;
    this.cols=cols;
    //method
    this.getCellCount=function()
    {
        return this.rows * this.cols;
    }
}
```

Private Members

```
function Table(rows,cols)
{
    //constructor
    var _rows = rows;
    var _cols = cols;
    //method
    this.getCellCount=function()
    {
        return _rows * _cols;
    }
}
```

```
//Now if you tries to access
var myTable=newTable(3,2);

myTable.getCellCount();
//it works and returns 6

mytable._rows;
//undefined
```

Object Literals

```
var empty = {};  An object with no properties
var point = { x:0, y:0 };  Two properties
var point2 = { x:point.x, y:point.y+1 };  More complex values
var book = {
    "main title": "JavaScript",  Property names include spaces,
    'sub-title': "The Definitive Guide",  and hyphens, so use string literals
    "for": "all audiences",  for is a reserved word, so quote
    author: {  The value of this property is
        firstname: "David",  itself an object. Note that
        surname: "Flanagan"  these property names are unquoted.
    }
};
```

Javascript Object Notation

JS

Javascript Object Notation

- JSON stands for **J**ava**S**cript **O**bject **N**otation
- Easy to read and write
- JSON is a lightweight format for storing and transporting data
- JSON is often used when data is sent from a server to a web page
- Fast for browser to parse.

```
{
"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]
}
```

Javascript Object Notation

- JSON.parse()
- JSON.stringfy()

```
JSON.stringify({name:"eman",age:20})
'{"name":"eman","age":20}‘
```

```
JSON.parse('{"name":"eman","age":20}')
{name: 'eman', age: 20}
```

Javascript Classes

JS

Introducing JavaScript Classes

Unlike most formal **object-oriented** programming languages, JavaScript didn't support classes and classical inheritance as the primary way of defining similar and related objects when it was created.

When you're exploring **ECMAScript 6 classes**, it's helpful to understand the underlying mechanisms that classes use. ECMAScript 6 classes aren't the same as classes in other languages

Class-Like Structures in ECMAScript 5

ECMAScript 5 and earlier, JavaScript had no classes. The closest equivalent to a class was creating a constructor

```
function Person (name,age)
{   //This is constructor
    this.name=name;
    this.age=age;
    this.sayName=()=>{console.log(this.name)};
}
let person=new Person("Mona",30);
person.sayName(); //Mona
```

Class Declarations

Class declarations begin with the **class** keyword followed by the name of the class. The rest of the syntax looks similar to **concise methods** in object literals but doesn't require **commas** between the elements of the class.

```
class Person
{   constructor(name,age)
    {  this.name=name;
        this.age=age;
    }
    sayName()
    {console.log(this.name); }
};
let person=new Person("Mona",30);
```

Why Use the Class Syntax?

Class declarations, unlike function declarations, are not hoisted.

```
let person=new Person("Mona",30); //error
class Person
{
    constructor(name,age)
    {
        this.name=name;
        this.age=age;
    }
    sayName()
    {
        console.log(this.name);
    }
};
```

Calling the class constructor without new throws an error.

```
let person= Person("Mona",30); //Class constructor Person cannot be invoked
without 'new'
```

Attempting to overwrite the class name within a class method throws an error.

```
class Foo
{
    constructor() {
        Foo = "bar"; // throws an error when executed...
    }
}
// but this is okay after the class declaration
Foo = "baz";
```

Class Expressions

Classes and functions are similar in that they have two forms: declarations and expressions. Function and class declarations begin with an appropriate keyword (function or class, respectively) followed by an identifier. Functions have an expression form that doesn't require an identifier after function; similarly, classes have an expression form that doesn't require an identifier after class. These class expressions are designed to be used in variable declarations or passed into functions as arguments.

Basic Class Expression

```
let Person=Class
{
    constructor(name,age)
    {
        this.name=name;
        this.age=age;
    }
    sayName()
    { console.log(this.name);  }
};
let person=new Person("Mona",30);
console.log(person instanceof Person);//true
console.log(typeof Person); //function
```

Basic Class Expression

ECMAScript 10

```
class Person
{
    name="eman";    //now we can define public here
    age=20;
    #ID=1;          //private member
    constructor(name,age,ID)
    {
        this.name=name;
        this.age=age;
        this.#ID=ID;
    }
};
```

Accessor Properties

Although you should create own properties inside class constructors, classes allow you to define accessor properties on the prototype. To create a **getter**, use the keyword get followed by a space, followed by an identifier; to create a **setter**, do the same using the keyword set

```
class CustomHTMLElement {
    constructor(element) {  this.element = element;  }
    get html() {  return this.element.innerHTML;    }
    set html(value) { this.element.innerHTML = value;    }
}
let element=new CustomHTMLElement(document.getElementsByTagName("div")[0]);
element.html;  //will return innerHTML of the element
element.html="test"; //will change innerHTML of the element
```

Static Members

ECMAScript 6 classes simplify the creation of **static** members by using the formal static annotation before the method or accessor property name. You can use the static keyword on any method or accessor property definition within a class. The only restriction is that you can't use static with the constructor method definition.

```
class Person
{  static count=0;      //new Feature
    constructor(name,age)
    {  Person.count++;
        this.name=name;
        this.age=age;
    }
    static personInfo()   //basic ES6 Fearture
    {
        console.log(`this is person Class`);
    }
};
Person.personInfo();//this is person Class
```

Inheritance

Prior to ECMAScript 6, implementing inheritance with custom types was an extensive process. Classes make inheritance easier to implement by using the familiar **extends** keyword to specify the function from which the class should inherit.

The prototypes are automatically adjusted, and you can access the base class constructor by calling the **super()** method.

```
class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }
    getArea() {
        return this.length * this.width;
    }
}
class Square extends Rectangle {
    constructor(length) {
        super(length, length);
    }
}
var square = new Square(3);
console.log(square.getArea()); // 9
console.log(square instanceof Square); // true
console.log(square instanceof Rectangle); // true
```

Notes on Using super( )

Keep the following key points in mind when you're using super():

✓You can only use super() in a derived class constructor. If you try to use it in a non derived class (a class that doesn't use extends) or a function, it will throw an error.

✓You must call super() before accessing this in the constructor. Because super() is responsible for initializing this, attempting to access this before calling super() results in an error.

Shadowing Class Methods

The methods on derived classes always shadow methods of the same name on the base class.

```
class Square extends Rectangle {
    constructor(length) {
        super(length, length);
    }

    getArea() {
        return this.length * this.length;
    }
}
```

Of course, you can always decide to call the base class version of the method by using the **super.getArea()** method

```
class Square extends Rectangle
{
    constructor(length) {
        super(length, length);
    }

    getArea() {
        return super.getArea();
    }
}
```

Using new.target in Class Constructors

You can also use **new.target** in class constructors to determine how the class is being invoked.

```
class Rectangle {
    constructor(length, width) {
        console.log(new.target.name === 'Rectangle');
        this.length = length;
        this.width = width;
    }
}
// new.target is Rectangle
var obj = new Rectangle(3, 4); // outputs true
```

This code shows that new.target is equivalent to Rectangle when new Rectangle(3, 4) is called. Class constructors can't be called without new, so the new.target property is always defined inside class constructors. But the value may not always be the same.

```
class Rectangle {
    constructor(length, width) {
        console.log(new.target.name === 'Rectangle');
        this.length = length;
        this.width = width;
    }
}
class Square extends Rectangle {
    constructor(length) {
        super(length, length)
    }
}
// new.target is Square
var obj = new Square(3); // outputs false
```

Square is calling the Rectangle constructor, so **new.target** is equal to Square when the Rectangle constructor is called. This is important because it gives each constructor the ability to alter its behaviour based on how it's being called. For instance, you can create an **abstract base** class (one that can't be instantiated directly) by using new.target

```
class Shape {// abstract base class
    constructor() {
        if (new.target.name === 'Shape') {
            throw new Error("This class cannot be instantiated directly.") }
    }
}
class Rectangle extends Shape {
    constructor(length, width) {
        super();
        this.length = length;
        this.width = width;
    }
}
var x = new Shape(); // throws an error
var y = new Rectangle(3, 4); // no error
console.log(y instanceof Shape); // true
```

Thank You