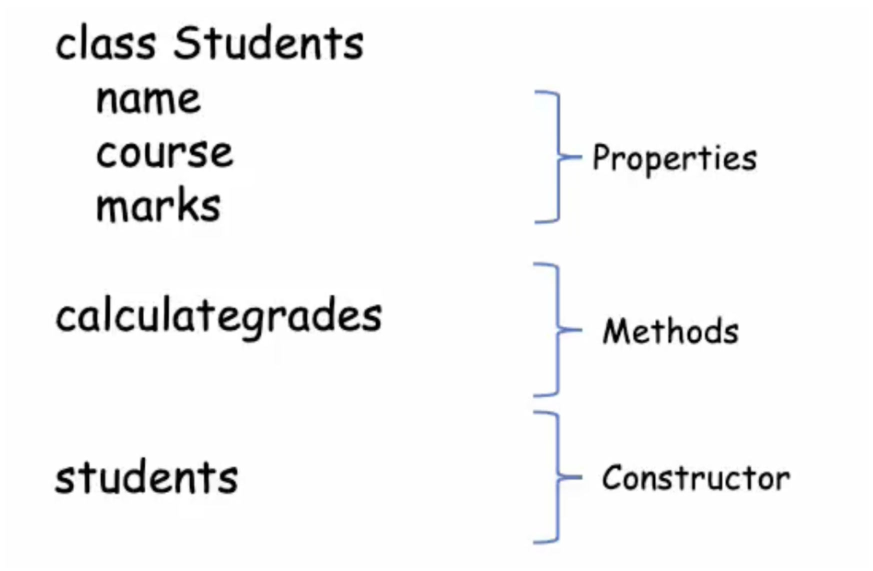
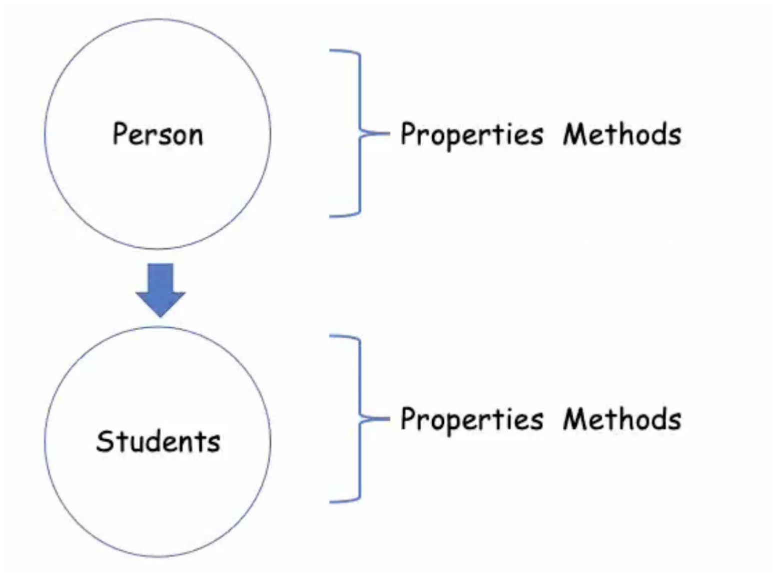
JS: Object Oriented Paradigm (1)

- We model entities in programming using **classes**
    - name, course, marks are all properties of students



- Classes do not exist in memory, however **Instances** of a class do exist in memory.
- Classes have a specific method known as Constructors to initialise values into the Object

Inheritance



- each student belongs to a broader entity called person
    - not all people are students, but all students are people
    - therefore Students inherit all the properties and methods from Person, but then Students have their own unique properties and methods that People might not have

Example in JavaScript:

```
class Students {
    name;

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

    myName() {
        console.log(this.name);
    }


const tom = new Students("Tom Sanders");
```

```
tom.myName();


OUTPUT:

Tom Sanders
```

In JavaScript Objects are *lightweight*

- Objects can be created **without** using a Class instead by using **Object Literals** e.g what you literally tell to the code
- Objects can be created using functions by passing values to a constructor

Examples of These Concepts

Creating an Object Using Literals

```
const Students = {
        name: "Tom Sanders",
        course: "Psychology",
}

console.log(Students.name);
console.log(Students.course);


OUTPUT:
```

```
Tom Sanders
Psychology
```

Creating an Object Using Functions

```
function Students(name) {
        this.name = name;
}

var tom = new Students("Tom Sanders");
console.log(tom.name);


OUTPUT"

Tom Sanders
```

- here we have a function called students, and this serves as a constructor function
    - this is helpful when you have MANY students and you wont want to use literals as this is time consuming and innefficient. You instead would want a function that you can pass in values that will populate students with values

Inheritance

- say for example we want every student to have a common property called “city”
- in JavaScript, **every Object** has a prototype keyword
    - you can therefor assign a value to this keyword
    - this value will apply to **all instances of Student** both *already existing* and *future* instances

```
Students.prototype.city="Bristol";

console.log(tom.name); // tom was created earlier
console.log(tom.city);



OUTPUT:

Tom Sanders
Bristol
```

Object Prototypes:

Understanding Prototypes

Every JavaScript object has a prototype. A prototype is also an object and acts as a template from which the object is inherited. This means that methods and properties defined on a prototype are available on all objects that inherit from that prototype.

Prototype Chain

When you try to access a property or method of an object, JavaScript first looks at the object itself. If it doesn't find the property or method, it looks up the prototype chain: it checks the object's prototype, then the prototype's prototype, and so on, until it finds the property or finds the end of the chain (null, as the prototype of `Object.prototype` is null).

Creating Objects and Setting Prototypes

There are several ways to create objects in JavaScript and set their prototypes:

Constructor Functions

Before ES6, constructor functions were a common way to define reusable object templates. For example:

```
javascriptCopy code
function Person(name, age) {
  this.name = name;
  this.age = age;
}

Person.prototype.greet = function() {
  console.log(`Hello, my name is ${this.name} and I am ${thi
s.age} years old.`);
};

const bob = new Person('Bob', 25);
bob.greet(); // Outputs: Hello, my name is Bob and I am 25 ye
ars old.
```

In this example, `Person` is a constructor function. Any object created using `new Person()` will have `Person.prototype` as its prototype. This includes the `greet` method.

Object.create

The `Object.create()` method creates a new object, using an existing object as the prototype of the newly created object.

```
javascriptCopy code
const prototypeObj = {
  greet: function() {
    console.log(`Hello, my name is ${this.name}`);
  }
};

const alice = Object.create(prototypeObj);
alice.name = 'Alice';
alice.greet(); // Outputs: Hello, my name is Alice
```

This method directly sets the prototype of `alice` to `prototypeObj`.

ES6 Classes

ES6 introduced classes to JavaScript, which is syntactic sugar over JavaScript's existing prototype-based inheritance.

```
javascriptCopy code

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

  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
}

const charlie = new Person('Charlie');
charlie.greet(); // Outputs: Hello, my name is Charlie
```

Here, `Person` is a class, but under the hood, JavaScript uses prototypes. Objects created using `new Person()` will inherit from `Person.prototype`.

Prototypes vs. Class Inheritance

While classes provide a clear and familiar syntax for inheritance, it's fundamentally prototype-based inheritance. Using prototypes directly can offer more control and flexibility but may also lead to more complex code structures.

Modifying Prototypes

It’s important to note that modifying an object’s prototype after objects have been created from it can lead to unpredictable results. Typically, prototypes are set up before any objects are created to ensure consistent behavior.

Conclusion

Prototypes are a powerful feature in JavaScript, enabling object-oriented programming features like inheritance and method sharing. Understanding how prototypes work is crucial for mastering JavaScript, especially for high-performance or complex applications that rely on object-oriented techniques.