Na aula passada vimos os conceitos iniciais de Programação Orientada a Objetos (POO) com JavaScript, e hoje vamos ver outros que também usamos nesse paradigma.
Imagine que queremos construir um sistema para a StackX de registro de pessoas, onde teremos alunos e professores, aplicando os conceitos de POO que aprendemos até aqui temos

```
class Aluno {
   constructor(nome, email, matricula) {
        this.nome = nome
        this.email = email
        this.matricula = matricula
   }

 //Método = nosso comportamento
 exibirInfos() {
   return `${this.nome}, ${this.matricula}`;
 }

}
```

e a classe de professor

```
class Professor {
   constructor(nome, email, matricula) {
        this.nome = nome
        this.email = email
        this.matricula = matricula
   }
 //Método = nosso comportamento
 exibirInfos() {
   return `${this.nome}, ${this.matricula}`;
 }

}
```

Percebe-se que temos as duas classes com os mesmos dados, e aí surge a pergunta, será que teríamos como não ficar repetindo código já que são as mesmas propriedades?
Sim, e a maneira que podemos fazer isso é com um conceito de POO muito utilizado, que é **Herança**: Objetos (classes) podem herdar de outros objetos (classes) propriedades e métodos

E vamos ver como funciona isso na prática:

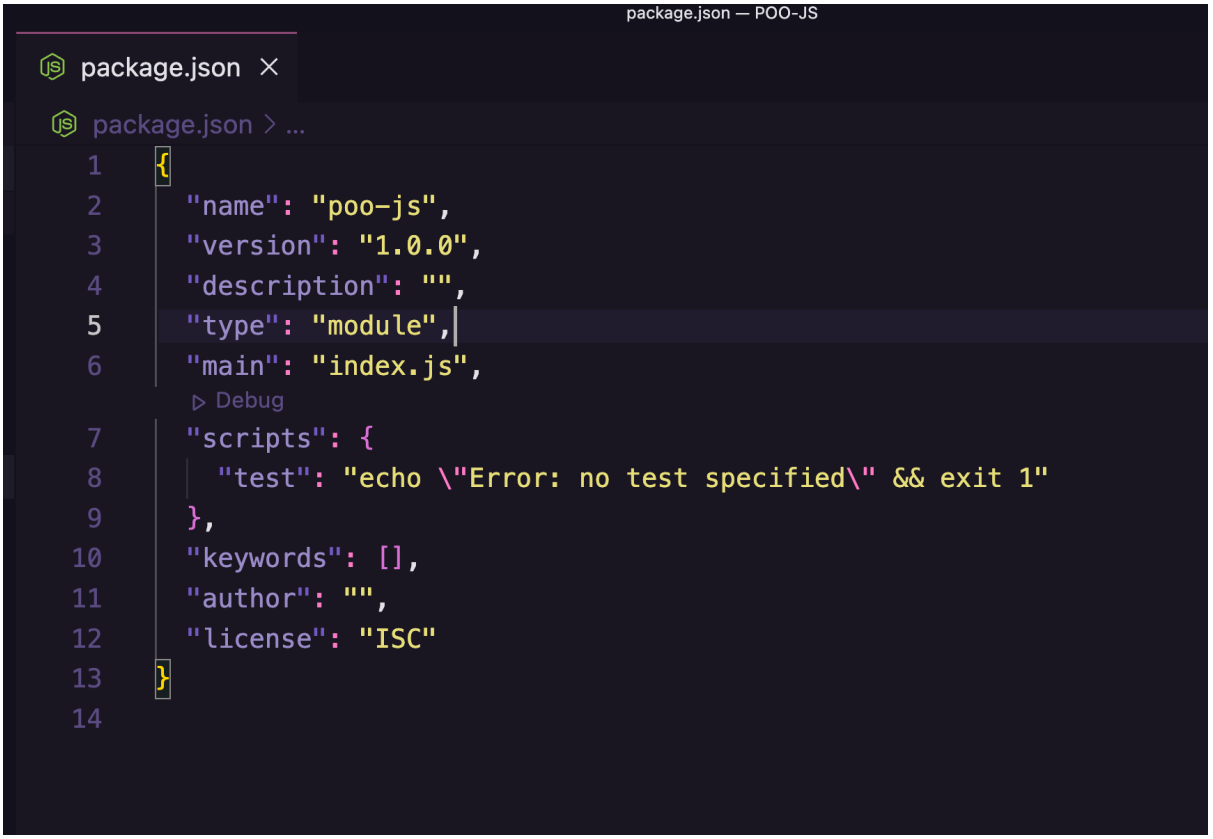
-Temos a nossa classe principal, primeiro devemos falar para o nosso programa que essa classe todo mundo pode usar, então colocamos o expor default antes de class

```
export default class Aluno {
   constructor(nome, email, matricula) {
        this.nome = nome
        this.email = email
        this.matricula = matricula
   }

 //Método = nosso comportamento
 exibirInfos() {
   return `${this.nome}, ${this.matricula}`;
 }

}
```

-Após isso criamos outro arquivo.js que terá a outra classe professor , e antes de começar a codar, abrimos um novo terminal do Visual Studio Code e digitamos npm init -y
Esse comando ele vai gerar um arquivo de configuração para o nosso projeto que é o package.json, que nada mais é do que o arquivo onde mostra nossas configurações, não precisamos entender ele agora, veremos mais sobre ele depois, mas após ele gerar você precisa **adicionar em baixo da linha description**, adicione uma **propriedade chamada type: module**



-Pronto, agora no seu novo arquivo que terá nossa classe professor, vamos apenas importar o nosso arquivo User e usar a palavra de herança no javaScript que é extends, e logo depois declaramos os dados que herdamos da classe user e podemos utilizar normalmente

```
//No arquivo que quero herdar a classe eu importo ela
import Aluno from "./Aluno.js"

class Admin extends User {
   //Parece que nao estamos reaproveitando, geralmente repetimos para dizer o
que queremos
   //Se por exemplo nao precisacessos do dado email para admin era s'o nao
escrever
   constructor(nome, email, nascimento, role = 'admin', ativo = true) {
               //super classe geralmente é como chamamos a classe que ta
fornecendo os parametros e os metodos para outras classe herdar
               //User e a nossa superclasse e a admin é nossa subclasse de
user / ou pai ou filho
               super(nome, email, nascimento, role, ativo)
   }
}
class Professor extends Aluno{
   constructor(nome, email, matricula) {
```

```
        //super classe geralmente é como chamamos a classe que ta fornecendo
os parametros e os metodos para outras classes herdarem
        //Aluno e a nossa superclasse e a professor é nossa subclasse de aluno
/ ou pai ou filho
        super(nome, email, matricula)
    }
}

const novoProfessor = new Professor('Gabriel', 'g2@g.com', '112332')
console.log(novoProfessor)
//Automaticamente herdamos tambem os métodos de aluno
console.log(novoProfessor.exibirInfos())
```

Agora imagine o seguinte, ainda está muito fácil acessar e alterar dados certo? E isso não é muito seguro, para isso vamos aplicar o conceito de atributo privado e método privado. O atributo privado fala que somente podemos acessar ele ou modificá-lo dentro da classe. Mas imagina só, agora temos uma classe (professor) que herda tudo da classe pai (aluno), como eu posso permitir que ela acesse os atributos privados e veja ele até modifique ele com segurança?
Utilizamos isso através de métodos acessores: Métodos que permitem acessar e modificar atributos privados com segurança:
-Métodos Acessores

1)Getter - Serve apenas para lermos atributos privados, ou seja, apenas conseguimos acessar o atributo privado, vamos ver como fazer isso no JS

```
export default class Aluno {
    //Atributo privado
    #matricula
    constructor(nome, email, matricula,){
        this.nome = nome;
        this.email = email;
        this.#matricula = matricula;
    }
    //Getter - Método acessor para retornar dado (serve apenas para leitura)
    get matricula() {
        return this.#matricula
    }

    exibirInformacoes() {
        return `Nome: ${this.nome}, matrícula: ${this.#matricula}`;
    }
}
```

```
const novoAluno = new Aluno('Gabriel', 'gabriel@email.com', '121212');
console.log(novoAluno.matricula);
```

Então aqui como podemos ver adicionamos o nosso getter e acessamos nossa matricula retornando o valor para o usuário, e se quisermos modificar, utilizamos o:

2)Setter - Método acessor que serve para alterar propriedades privadas de forma segura, ele aceita parâmetro. Vamos ver como fazer isso com JS

```
export default class Aluno {
    //Atributo privado
    #matricula
    constructor(nome, email, matricula,){
        this.nome = nome;
        this.email = email;
        this.#matricula = matricula;
    }

    //Getter - Método acessor para retornar dado (serve apenas para leitura)
    get matricula() {
        return this.#matricula
    }

    //Setter - Método acesso para alterar dados (serve para escrita)
    set matricula(novaMatricula) {
        if(novaMatricula === ''){
            throw new Error('Preencha uma matrícula válida')
        }
        this.#matricula = novaMatricula
    }

    exibirInformacoes() {
        return `Nome: ${this.nome}, matrícula: ${this.#matricula}`;
    }
}

const novoAluno = new Aluno('Gabriel', 'gabriel@email.com', '121212');
console.log(novoAluno.matricula);
// chamando o setter (altera o dado)
novoAluno.matricula = '111'
console.log(novoAluno.matricula);
```

Maravilha né? Agora nosso sistema está mais seguro, mas imagine o seguinte, com a herança dos atributos e métodos da classe aluno para professor, estamos retornando no método exibir informacoes esses dados

```
    exibirInformacoes() {
        return `Nome: ${this.nome}, matrícula: ${this.#matricula}`;
    }
```

Mas e se quisesse mudar esse retorno na nossa classe filha (professor), e retornar por exemplo só o nome do professor, podemos fazer isso?
Podemos sim, e é através do **Polimorfismo:** Possibilidade de uma subclasse (classe filha) modificar um comportamento da classe pai
Vamos ver como fazer isso em JS:

```
//Herança - classe possa herdar objetos e propriedades e métodos de outra
classe
import Aluno from './aluno.js';

class Professor extends Aluno{
    constructor(nome, email, matricula,){
        super(nome, email, matricula)
    }

    //Polimorfismo
    exibirInformacoes() {
        return `Professor: ${this.nome}`
    }
}

const novoProfessor = new Professor('Gabriel Professor',
'gabrielProf@email.com', '123');
console.log(novoProfessor.exibirInformacoes());
```

Então como vemos aqui estamos modificando o método exibirInformacoes para ele retornar o nome do professor apenas.

Então é isso pessoal, esses são os principais conceitos de POO com JS que vocês precisam saber, e vamos em frente que temos mais conhecimentos para ver. Até mais e parabéns pelo esforço!! :)