

INSTITUTO FEDERAL

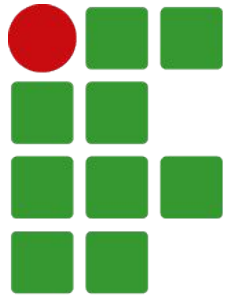
Paraná

Campus Paranaguá

JS - Orientação a Objetos, First Class Functions e IIFE

Desenvolvimento Web

Prof. Diego Stiehl



INSTITUTO FEDERAL

Paraná

Campus Paranaguá

Como funciona a OO e a herança no JavaScript

Orientação a Objetos

Objetos Novamente

- Em JavaScript tudo é objeto
 - QUASE verdade
- Tipos primitivos não são objetos
 - Number, String, Boolean, Undefined e Null
- Todo o resto é objeto
 - Array, Function, Object, Date, ...

Lembram do nosso objeto?

```
let diego = {  
  nome: 'Diego', // string  
  sobrenome: 'Stiehl', // string  
  anoNascimento: 1988, // number  
  admin: true, // boolean  
  cachorros: ['Fritz', 'Franz', 'Berlin'], // Array  
  profissao: { // object  
    cargo: 'Professor',  
    atribuicoes: 'Ensinar web pra piazada'  
  },  
  nomeCompleto: function() { // function  
    return `${this.nome} ${this.sobrenome}`  
  }  
};
```

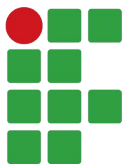
Nosso objetão

- Nosso objetão é muito bonito, mas...
- Ele é único
- E se quisermos fazer outro equivalente?
 - Precisamos repetir tudo?
 - Ctrl+C / Ctrl+V?
 - Como manter a uniformidade?



Protótipo

- Para resolver nosso problema, podemos usar um **protótipo** (prototype)
 - Também chamado de construtor (*constructor*)
- É um objeto qualquer que define uma estrutura comum
 - Semelhante às classes de outras linguagens
- Outros objetos podem ser criados a partir dele (instanciação)



Protótipo

- Na verdade, todo objeto JavaScript sempre tem um protótipo
- Teste estes dois trechos no Console:

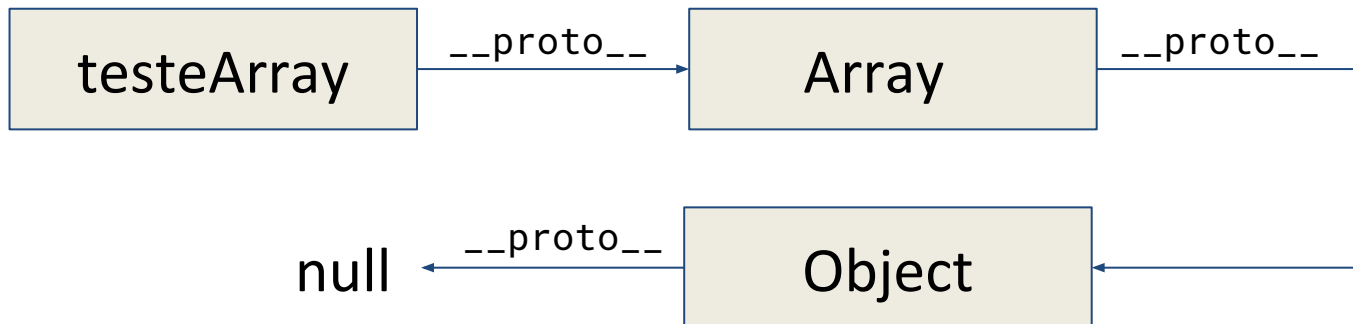
```
let testeObjeto = {  
  atributo: 'valor'  
};
```

```
console.log(testeObjeto);
```

```
let testeArray = [1, 2, 3];  
console.log(testeArray);
```

- Explore os objetos e procure pela propriedade `__proto__` de cada um

Objetos e seus Protótipos



Prototype Chain

- Sempre que criamos um objeto com base em outro dizemos que o objeto-base é o **protótipo** do objeto criado
- Podemos “navegar” na estrutura de protótipos dos nossos objetos

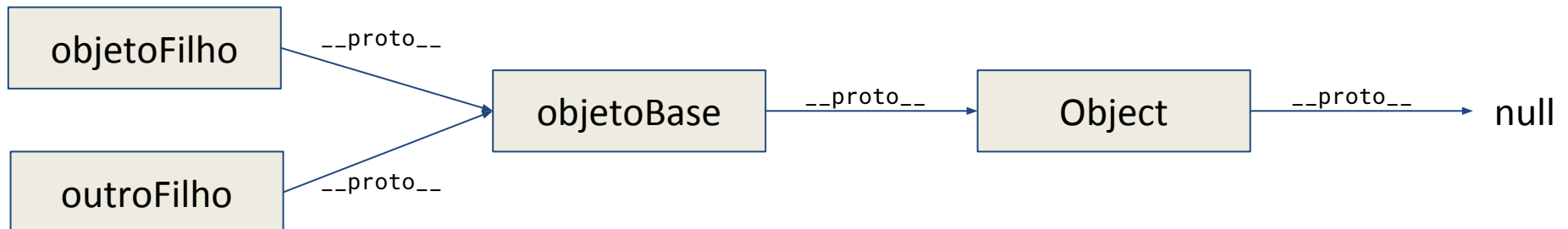
`testeArray.__proto__.__proto__.__proto__`

- Chamamos isto de Prototype Chain
 - É a Herança do JavaScript

Como criar instâncias?

- Opção clássica → `Object.create()`
- Execute e analise no Console:

```
let objetoBase = {  
  atributo: 'valor'  
};  
let objetoFilho = Object.create(objetoBase);  
let outroFilho = Object.create(objetoBase);
```



Como criar instâncias?

- Forma mais comum → Function Constructor

```
const Person = function(nome) {  
    this.nome = nome;  
}
```

- É uma função especial
 - Não serve para ser chamada isoladamente
- Consideramos que um novo objeto será criado e atribuído ao **this**

Function Constructor

- Para chamar, precisamos da palavra new
 - Um objeto vazio é criado → { }
 - O código da Function Constructor é executado
 - O `this` refere-se ao novo objeto
 - O `__proto__` do novo objeto é o prototype da Function Constructor

```
let diego = new Person('Diego');  
diego.__proto__ === Person.prototype // true
```

Prática - Refatorar Objeto

- Refatore nosso objeto

- Utilizar Function Constructor
- Deixar estrutura dinâmica
- Criar duas pessoas para testar

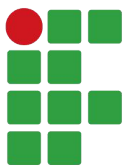
```
let diego = {  
  nome: 'Diego',  
  sobrenome: 'Stiehl',  
  anoNascimento: 1988,  
  admin: true,  
  cachorros: ['Fritz', 'Franz', 'Berlin'],  
  profissao: {  
    cargo: 'Professor',  
    atribuicoes: 'Ensinar web pra piazada'  
  },  
  nomeCompleto: function() {  
    return `${this.nome} ${this.sobrenome}`  
  }  
};
```

Solução (não espiar)

```
const Pessoa = function(nome, sobrenome, anoNascimento,
                        admin, cachorros, profissao) {

    this.nome = nome;
    this.sobrenome = sobrenome;
    this.anoNascimento = anoNascimento;
    this.admin = admin;
    this.cachorros = cachorros;
    this.profissao = profissao;
    this.nomeCompleto = function() {
        return `${this.nome} ${this.sobrenome}`
    };
}

let diego = new Pessoa('Diego', 'Stiehl', 1988, true,
                        ['Fritz', 'Franz', 'Berlin'],
                        {
                            cargo: 'Professor',
                            atribuicoes: 'Ensinar web pra piazada'
                        }
                        );
```

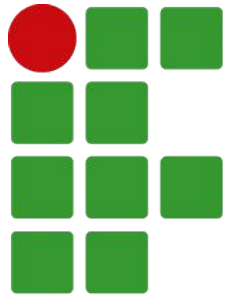


Herança

- A solução cria cópias de todos os atributos
 - Cada objeto tem seus atributos e métodos
- O ideal seria herdarmos os métodos
 - Implementar no protótipo Pessoa
- Remover o método da function e fazer:

```
Pessoa.prototype.nomeCompleto = function() {  
    return `${this.nome} ${this.sobrenome}`  
};
```

- Todas pessoas terão o método via herança



INSTITUTO FEDERAL

Paraná

Campus Paranaguá

JavaScript

First Class Functions

Funções e Objetos

- Funções também são objetos
 - São instâncias de Object (prototype)
 - Se comportam como objetos
 - Conseguimos armazená-las
 - Podemos passá-las como parâmetro para outras funções
 - Podemos retornar uma função de outra função
- Por isto são chamadas de First-Class Functions
 - Funções de primeira classe

Função como parâmetro

- Podemos passar a referência de uma função como parâmetro para outra função
- Objetivo:
 - Queremos que a função invocada invoque a nossa função do parâmetro
 - Uma ou mais vezes
- Chamamos a função passada como parâmetro de Callback Function

Como Funciona

```
function funcaoPrincipal(funcaoCallback) {  
    console.log('PRINCIPAL');  
    funcaoCallback();  
}
```

```
function dizOla() {  
    console.log('CALLBACK');  
    console.log('Olá!');  
}
```

Não passar dizerOla() com
parênteses para não invocar
antes da hora

```
funcaoPrincipal(dizOla);
```



Exemplo: Login

```
function logar(usuario, senha, funcaoSucesso, funcaoFalha) {  
    if (usuario === 'aluno' && senha == '123')  
        funcaoSucesso('Usuário logado.');  
    else  
        funcaoFalha('Login ou senha inválidos');  
}
```

```
logar('aluno', '123',  
    function(mensagem) {  
        console.log(mensagem);  
        console.log('Entrando no sistema...');  
    }, function(mensagem) {  
        console.log(mensagem);  
        console.log('Tente novamente.');
```

Duas funções de callback.
Funções anônimas também
funcionam. =)

Exemplo: Calcular Idade

```
const nascimentos = [1988, 2005, 2008, 1967, 1991];
```

```
function calculaLista(vetor, funcao) {  
  let novos = [];  
  for (const item of vetor) {  
    const calculado = funcao(item);  
    novos.push(calculado);  
  }  
  return novos;  
}
```

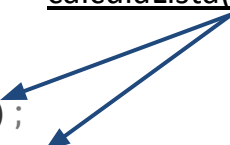
Função de Callback



```
function calculaIdade(anoNascimento) {  
  return 2019 - anoNascimento;  
}
```

```
function verificaMaiorIdade(idade) {  
  return idade >= 18;  
}
```

Diferentes funções de
callback.
calculaLista() é genérica.



```
const idades = calculaLista(nascimentos, calculaIdade);  
const maiores = calculaLista(idades, verificaMaiorIdade);
```

Prática: Filtrar Vetor

- **Crie uma função** que aceitei dois parâmetros
 - Um vetor com números e uma função de callback
- Ela deve retornar um novo vetor
 - O novo vetor terá menos elementos
 - Só restarão os elementos que a função de callback permitir
- A função de callback deve retornar boolean
 - Fazer uma condição
- Crie e chame 3 diferentes funções de callback

Retornando Funções

- Uma **função** pode retornar outra
- O código que invocou a primeira função pode recebê-la e invocá-la
- Útil para dividirmos o processamento em mais partes ou reaproveitarmos parte da invocação

Como Funciona

```
function funcaoPrincipal() {  
    return function(){  
        console.log('Hello from inside!');  
    }  
}
```

```
const funcaoRetornada = funcaoPrincipal();  
funcaoRetornada();
```


Exemplo: Pergunta de Prova

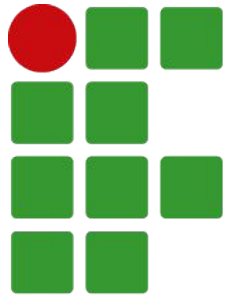
```
function perguntaProva(disciplina) {  
  if (disciplina === 'web') {  
    return function(aluno) {  
      console.log(`${aluno}, explique o que é CSS.`);  
    }  
  } else if(disciplina == 'android') {  
    return function(aluno) {  
      console.log(`Hey ${aluno}! O que é uma Activity?`);  
    }  
  } else {  
    return function(aluno) {  
      console.log(`${aluno}. Quem veio antes: o ovo ou a galinha?`);  
    }  
  }  
}
```

Exemplo: Pergunta de Prova

```
const perguntaWeb = perguntaProva('web');  
perguntaWeb('Diego');  
perguntaWeb('Luiza');
```

```
const perguntaGeral = perguntaProva();  
perguntaGeral('Fritz');
```

```
// Invoca perguntaProva e imediatamente a função retornada  
perguntaProva('android')('Franz');
```



INSTITUTO FEDERAL

Paraná

Campus Paranaguá

Immediately Invoked Function Expression

IIFE

Encapsulamento

- Até então temos escrito nossos códigos JavaScript diretamente na raiz do script
- Pode não ser uma boa ideia
 - Todas nossas variáveis acabam sendo globais
 - Outros scripts podem sobrescrever seus valores
 - Maliciosamente ou não
- Precisamos encapsular nosso código
 - Limitar o contexto de execução

Limitando o Contexto

- Podemos limitar o contexto de execução colocando todo o código em uma função

```
// Todo o código da minha aplicação aqui dentro  
function app() {  
  // Todas que for criado aqui não existe fora da função  
  const valor = 10;  
  let nome = 'Diego';  
  function fazAlgumacoisa(texto) {  
    console.log(`Alguna coisa ${texto}`);  
  }  
  fazAlgumacoisa(nome);  
  console.log(valor);  
}  
app();  
// Não consigo acessar valor, nome e fazAlgumaCoisa() daqui
```



IIFE

- A proposta anterior funciona bem
- Mas foi preciso criar uma função e posteriormente invocá-la
- A comunidade criou um “truque” para abreviar esta chamada
- IIFE
 - *Immediately Invoked Function Expression*
 - Função como expressão invocada imediatamente

Sintaxe

- Primeiro colocamos uma função anônima entre parênteses
 - Isto obriga o JS a executar a expressão e retornar a função

```
(function() {})
```

- Depois, podemos invocar a função retornada

```
(function() {})(  );
```

Invocação

Aplicando IIFE ao Exemplo

```
// Todo o código da minha aplicação aqui dentro
(function() {
  // todas variáveis criadas aqui não existem fora
  const valor = 10;
  let nome = 'Diego';
  function fazAlgumacoisa(texto) {
    console.log(`Alguma coisa ${texto}`);
  }
  fazAlgumacoisa(nome);
  console.log(valor);
})();
// Não consigo acessar valor, nome e fazAlgumaCoisa() daqui
```