Classes e Módulos em Javascript

Rafael Escalfoni

adaptado de

Progressive Web Apps – Construa aplicações progressivas com React. Guilherme Pontes

Learning React. Kirupa Chinnathambi

React and Libraries. Elad Elrom

Orientação a Objetos - OO

- Paradigma de programação amplamente difundido
  - Imperativa, funcional, lógica, OO...
  - Reúso, facilidades para realização de testes, interoperabilidade
- Classe – molde de um elemento
- Objeto – uma instância do elemento
- Encapsulamento, Herança e Polimorfismo
  - Encapsulamento – os detalhes de implementação ficam protegidos
  - Herança – permite criar classes especializadas
  - Polimorfismo – permite a criação de diferentes procedimentos a serem executados para um mesmo método

Até a ECMAScript 5.1...

- Exemplo de Carros:
- Imagine que estamos modelando um carro. Todos os carros possuem as seguintes características em comum:
  - **Modelo**
  - **Chassi**
  - **Quantidade de portas**
  - Tipo de combustível
  - Tração
  - Transmissão

Criando um objeto Carro

```
function Carro(modelo, chassi, qtdPortas) {
  this.modelo = modelo;
  this.chassi = chassi;
  this.qtdPortas = qtdPortas;
}
//testando
const prototipo = new Carro('protótipo', '12903811209', 2);
console.log(prototipo.modelo);    // protótipo
console.log(prototipo.chassi);    // 12903811209
console.log(prototipo.qtdPortas); // 2
```

Adicionando funcionalidades - andar

```
Carro.prototype.andar = function() {
  console.log("vrum vrum");
}

const prototipo = new Carro('protótipo', '12903811209', 2);
prototipo.andar(); // vrum vrum

const sonix = new Carro('Sonix', '9120219', 4);
console.log(sonix.modelo); // Sonix
sonix.andar(); // vrum vrum
```

Herança em ECMAScript 5

Considerando agora que Sonix é um carro com algumas propriedades a mais

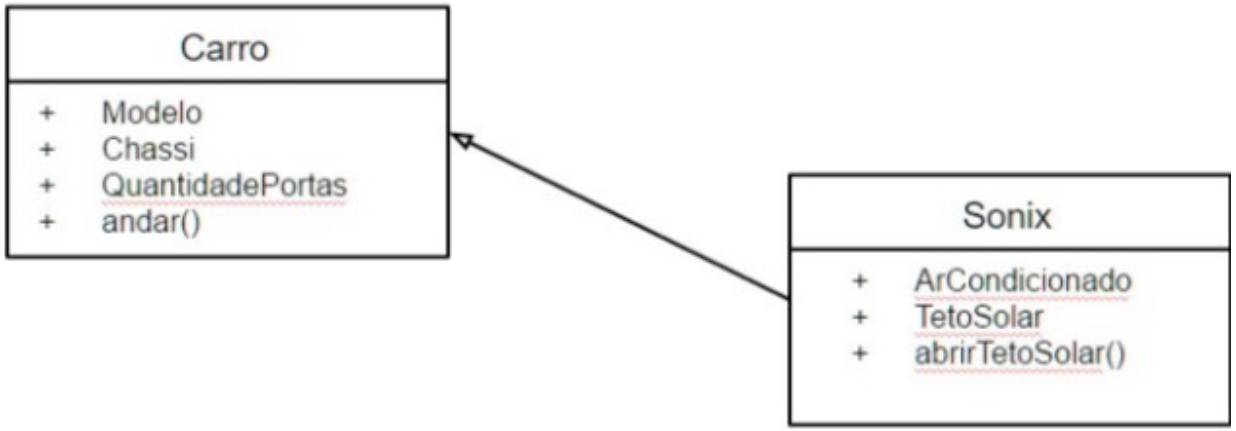


Figura 16.1: Relação entre os modelos de carros

```
function Sonix(modelo, chassi, qtdPortas) {
  Carro.call(this, modelo, chassi, qtdPortas);
}
Sonix.prototype = Object.create(Carro.prototype);
Sonix.prototype.constructor = Sonix;

const modeloBasico = new Sonix('Sonix', '9120219', 4);
console.log(modeloBasico.modelo); // Sonix
modeloBasico.andar(); // vrum vrum
```

Estendendo a classe Carro:

```
Sonix.prototype.abrirTetoSolar = function() {
  console.log('abrindo teto solar');
}

const modeloSport = new Sonix('Sonix', '9122222', 4);
sonix.abrirTetoSolar(); // abrindo teto solar
```

Criando classes em ECMAScript 6

- Palavra reservada **class**

```
class Carro {
  // implementação do carro
}
```

- Função construtora:

```
class Carro {
  constructor(modelo, chassi, qtdPortas) {
    this.modelo = modelo;
    this.chassi = chassi;
    this.qtdPortas = qtdPortas;
  }
}
```

Adicionando métodos

- Função andar:

```
class Carro {
  constructor(modelo, chassi, qtdPortas) {
    this.modelo = modelo;
    this.chassi = chassi;
    this.qtdPortas = qtdPortas;
  }
  andar() {
    console.log("vrum vrum");
  }
}
const basico = new Carro('Básico', '123123', 2);
console.log(basico.modelo);    // Básico
console.log(basico.qtdPortas); // 2
basico.andar();                // vrum vrum
```

Estendendo classes em ECMAScript 6

```
class Sonix {
  abrirTetoSolar() {
    console.log('abrindo teto solar');
  }
}
```

Estendendo:

```
class Sonix extends Carro {
  abrirTetoSolar() {
    console.log('abrindo teto solar');
  }
}

const basico = new Sonix();
basico.abrirTetoSolar(); // abrindo teto solar
basico.andar();          // vrum vrum
```

Fazendo referência à classe pai - super

```
class Sonix extends Carro {
  constructor(modelo, chassi, qtdPortas) {
    super(modelo, chassi, qtdPortas);
  }
  abrirTetoSolar() {
    console.log('abrindo teto solar');
  }
}

const basico = new Sonix('Sonix e', '91212', 2);
basico.abrirTetoSolar();     // abrindo teto solar
basico.andar();              // vrum vrum
console.log(basico.modelo); // Sonix e
```

Métodos estáticos

- São métodos que podem ser executados sem precisar instanciar uma classe

```
class Casa {
  static abrirPorta() {
    console.log('abrindo porta');
  }
}
Casa.abrirPorta(); // abrindo porta
```

Tornando atributos privados - Weakmap

- Impedir acesso indevido à uma propriedade (encapsulamento)

```
const propriedades = new WeakMap();

class VideoGame {
  constructor(nome, controles, saida, midia) {
    propriedades.set(this, {
      nome, controles, saida, midia
    });
  }
}
const caixa360 = new VideoGame('caixa360', 4, 'hdmi', 'dvd');
console.log(caixa360.nome); // undefined
```

```
const propriedades = new WeakMap();

class VideoGame {
  constructor(nome, controles, saida, midia) {
    propriedades.set(this, {
      nome, controles, saida, midia
    });
  }
  getProp(propName) {
    return propriedades.get(this)[propName];
  }
}
const caixa360 = new VideoGame('caixa360', 4, 'hdmi', 'dvd');
console.log(caixa360.getProp('nome')); // caixa360
```

Modularização de Sistemas

Dividir e conquistar

- Projetos grandes requerem estratégias para tornar a gestão do desenvolvimento mais simples
- Modularizar significa **dividir** o código em partes que representam uma **abstração funcional** e **reaproveitável** da aplicação
- Modularizar permite organizar e reaproveitar o código

Representação de um sistema em módulos

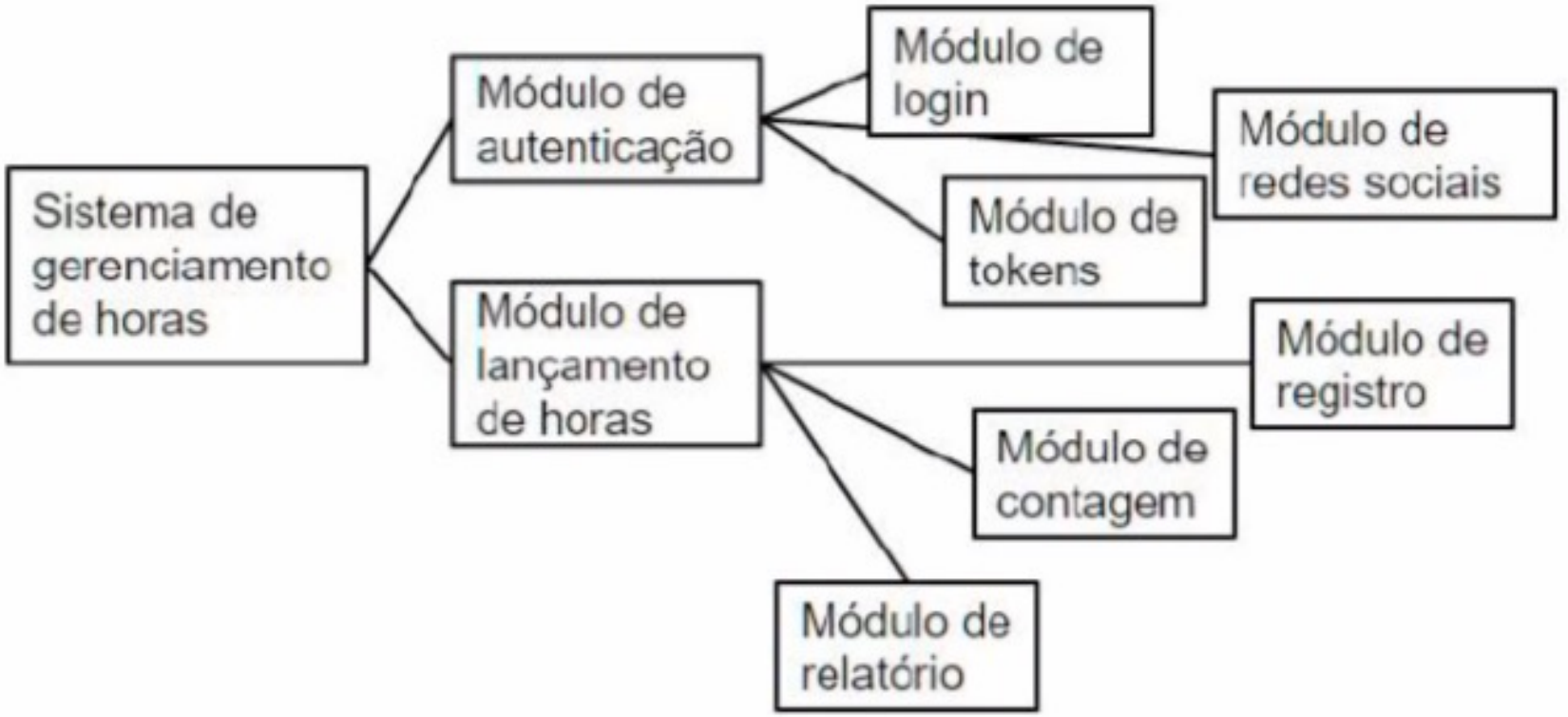


Figura 17.1: Simulação de modularização de um sistema

Limitações do ECMAScript 5.1

- Não havia suporte nativo a módulos
- Bibliotecas de suporte - CommonJS e AMD
- CommonJS
  - API síncrona, muito boa para lado servidor
- AMD (*Asynchronous Module Definition*)
  - Mecanismo de definição de módulos e dependências *assíncrono*
  - Boa saída para lado cliente

Gerindo Módulos em ECMAScript 6

- Importando e exportando módulos
- **Tudo é interpretado como um módulo em ECMAScript 6**
- Cada módulo é armazenado em um arquivo JavaScript
- Palavras reservadas `import` e `export`

Exportando módulos

- Tipos de exportação: padrão x nomeado
- Padrão
  - para caso em que temos algum valor primário (função, variável, etc)
  - apenas um por módulo
- Nomeado
  - para quando o módulo possa ser consumido em pedaços
  - usado várias vezes por módulo

Módulo de cálculos com circunferências

circunferencia.js

```
export const PI = 3.14;

function comprimento(raio) {
  return 2 * PI * raio;
}

function area(raio) {
  return PI * (raio * raio);
}

export default comprimento; // padrão
export {area, PI};          // nomeado
```

padrão ou nomeado???

matematica.js :

```
import comprimento from ‘./circunferencia’;
comprimento (2);  // 12.5
```

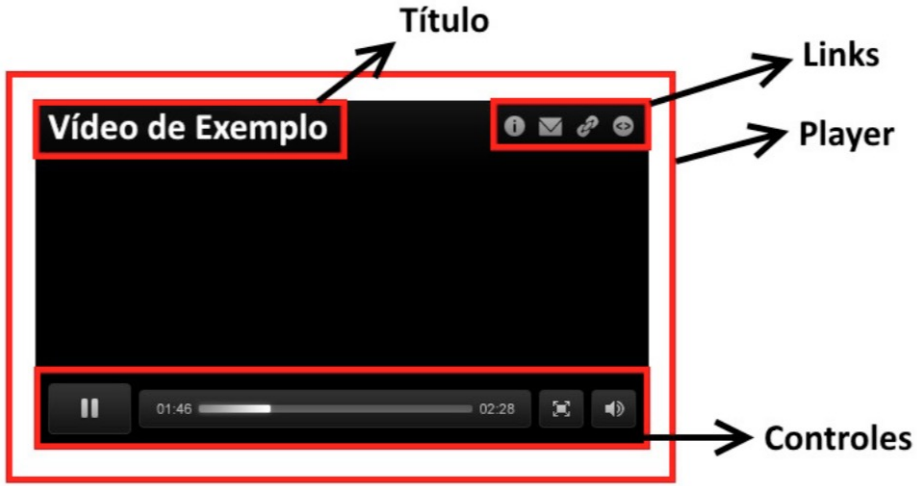
No caso dos nomeados:

```
import {raio, PI}  from ‘./circunferencia’;
comprimento (2);  // 12.5
raio(2);
console.log(PI); // 3.14
```

Exportando Classes

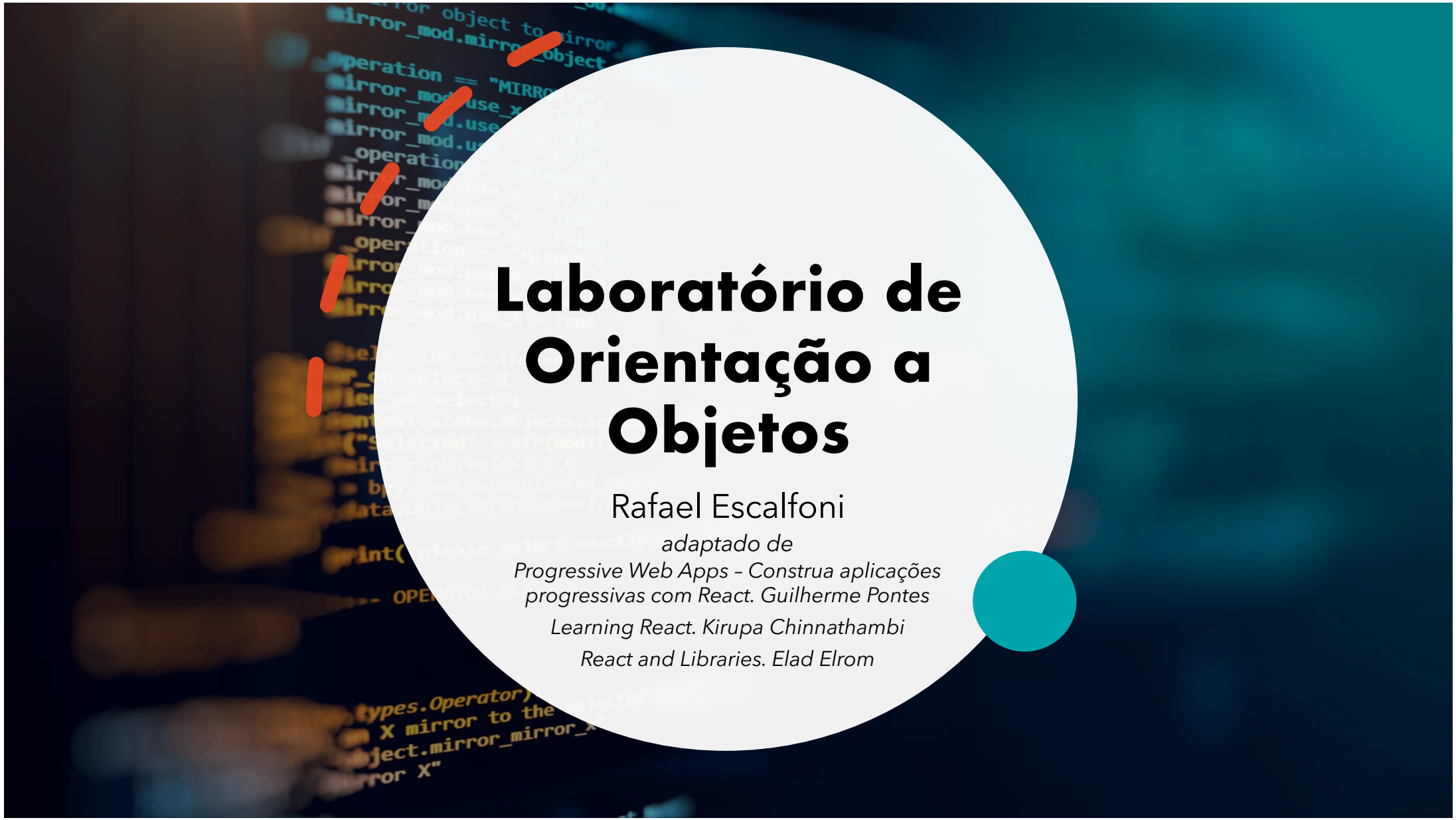
- Vários componentes no player

```
class Controles {
// implementação do botão
}
export default Controles;
```




```
import Controles from './Controles';
import Titulo from './Titulo';
import Links from './Links';
class Player {
// implementação do player
}
```

- Módulos são singletons
  - Mesmo que um módulo seja importado múltiplas vezes dentro de um projeto, só vai existir uma instância dele
- Módulos podem importar coisas de outros módulos
  - É possível aproveitar dentro de um módulo coisas que foram importadas de outros módulos que ele usa
- Importações de módulos são hoisted
  - Tudo que é importado é movido internamente para o topo



Laboratório de Orientação a Objetos

Rafael Escalfoni

adaptado de

Progressive Web Apps – Construa aplicações progressivas com React. Guilherme Pontes

Learning React. Kirupa Chinnathambi

React and Libraries. Elad Elrom

Sistema de Gestão de Eventos – Nova Friburgo

Eventos

- Propriedades comuns:
  - Local, título, descrição, horário, data de início, data de término, fotos de divulgação
- Podem ser:
  - Eventos Gastronômicos
    - Restaurantes/bares envolvidos, cozinha
  - Eventos Culturais
    - Artistas envolvidos, estilo

Eventos

- Métodos