Programação Orientada a Objetos com

Cláudio Luís V. Oliveira (claudio.oliveira@fatec.sp.gov.br)

Olá TypeScript

O TypeScript pode ser considerado como um super conjunto do JavaScript, adicionando uma série de funcionalidades que não estão disponíveis ou que não apresentam uma sintaxe clara, dificultando o desenvolvimento e o entendimento dos programas. Neste cenário, podemos citar a tipagem de dados e a orientação a objetos.

A instalação do TypeScript, considerando que o Node.js já está presente no computador, pode ser realizada através do gerenciador de pacotes npm. Desta forma, digite no prompt de comandos do sistema operacional:

```
npm install -g typescript
npm install -g ts-node
```

Com o intuito de testar a instalação dos módulos, vamos implementar um programa simples. Grave o arquivo com o nome “ola.ts”.

```
let nome: string = 'Cláudio';
console.log ('Olá, ' + nome + '!');
```

ola.ts

A execução do programa que criamos pode ser realizada através do compilador TypeScript (**tsc**) que irá realizar a verificação sintática e gerar um arquivo JavaScript que, na sequência, poderá ser executado através do node.

```
tsc ola.ts
node ola.js
```

Existe também a possibilidade de executar diretamente o arquivo com o código-fonte em TypeScript. Neste caso, use o módulo **ts-node** que também já se encontra instalado.

```
ts-node ola.ts
```

Os tipos de dado mais comuns são **string**, **number** e **boolean**. Cabe salientar que todos os valores numéricos no TypeScript são de ponto flutuante. Além disso, há o tipo de dado **any**, que pode ser usando quando não há uma definição sobre o tipo de dado a ser usado em determinada variável. Também podemos agrupar conjuntos de dados em vetores (arrays). Desta forma, podemos declarar as variáveis da seguinte maneira:

```
let descricao: string = 'Geladeira';
let preco: number = 2000.00;
let disponivel: boolean = true;
let garantia: any = 12;
garantia = '12 meses';
let historicoPreco: number[] = [1990.00, 2100.00, 2050.00, 1990.00];
```

variaveis.ts

O acesso ao vetor pode ser realizado através da instrução **for**, conforme mostrado no programa a seguir.

```
let historicoPreco: number[] = [1990.00, 2100.00, 2050.00, 1990.00];

for (let i in historicoPreco) {
  console.log(historicoPreco[i]);
}
```

vetor.ts

Como mencionado anteriormente, outro recurso valioso implementado no TypeScript é a orientação a objetos. Com o intuito de exemplificar este conceito, vamos criar um novo arquivo e nele definimos a classe **Saudacao** que, por sua vez, vai conter o atributo nome e os métodos construtor e **getOla()**.

```
class Saudacao {
  private nome: string;

  constructor (nome: string) {
    this.nome = nome;
  }

  getOla(): string {
    return "Olá, " + this.nome + "!";
```

```
  }
}

let saudacao: Saudacao = new Saudacao('Cláudio');
console.log(saudacao.getOla());
```

saudacao.ts

Conceitos Básicos de Orientação a Objetos

O paradigma da Orientação a Objetos trouxe uma nova forma de projetar e de desenvolver programas de computadores. Enquanto o projeto e a programação estruturada focam a redução dos problemas computacionais a apenas três estruturas básicas, que é sequência, decisão e interação, e na qual, procedimentos, funções e estruturas de dados apresentam ligações e relacionamentos tênues, por outro lado, a Orientação a Objeto estabelece um forte relacionamento entre os diversos componentes de um determinado problema computacional através da adoção de dois conceitos chave: classe e objeto.

Na programação orientada a objetos implementa-se um conjunto de classes que permitem a definição dos objetos presentes no projeto do sistema. Cada classe tem por objetivo determinar o comportamento, que são definidos pelos métodos, e estados possíveis, descritos pelos atributos que compõem e caracterizam os objetos, além de permitir a definição do relacionamento com outros objetos.

Conforme mencionado acima, a orientação a objeto, pode ser entendida através da definição de dois dos elementos chave:

- O primeiro é o conceito de classe, que pode ser entendido como a descrição de um ou mais objetos através de um conjunto uniforme de atributos e métodos.

- O segundo é o próprio objeto, que pode ser definido como uma abstração de algo que existe dentro do domínio de um problema ou na sua implementação, sendo que todo objeto é a instância de uma classe. Ilustrando estes conceitos, considere uma determinada classe chamada pessoa, os objetos dessa classe são os indivíduos que a compõem, por exemplo, José e Maria entre outros.

Em síntese, uma **classe** determina o comportamento dos objetos, que são definidos pelos métodos, e estados possíveis, descritos pelos atributos que compõem e caracterizam esses objetos. Já o **objeto** pode ser entendido como uma abstração de algo que existe dentro do domínio de um problema ou na sua implementação, sendo que todo objeto é a instância de uma classe. Para um perfeito entendimento dos conceitos básicos da orientação a objeto torna-se necessário definir os seguintes conceitos:

Abstração é a capacidade de permitir a concentração apenas nos aspectos essenciais de um contexto, mascarando as características menos importantes. Na modelagem orientada a objetos, uma classe pode ser entendida como uma abstração das entidades existentes no domínio da aplicação.

Atributo: define um conjunto padrão de características específicas para uma determinada classe. Os valores (estados) que estes atributos recebem, quando a classe é instanciada, permitem caracterizar um objeto. Considerando uma determinada classe pessoa, os atributos poderiam ser nome, endereço e telefone, entre outros.

Método: solicitação a um objeto invocando um de seus métodos, método este que ativa um determinado comportamento descrito pela classe que deu origem ao objeto em questão. Usando como exemplo a classe pessoa, poderíamos definir métodos para alterar o nome, o endereço ou o telefone de determinado indivíduo.

Método Construtor: é um método especial, que é executado somente quando o objeto é criado, ou seja, quando uma classe é instanciada.

A partir dos conceitos de classes, atributos e métodos é possível realizar a definição de uma classe utilizando, para isso, a sintaxe em TypeScript:

```
class Pessoa {
  private nome: string = '';
  private endereco: string = '';
  private telefone: string = '';

  constructor () {
  }

  public setNome(nome: string) {
    this.nome = nome;
  }

  public getNome(): string {
    return this.nome;
  }

  public setEndereco(endereco: string) {
    this.endereco = endereco;
  }

  public getEndereco(): string {
    return this.endereco;
  }

  public setTelefone(telefone: string) {
    this.telefone = telefone;
  }

  public getTelefone(): string {
    return this.telefone;
  }
}
```

pessoa-1.ts

Palavra-reservada this: é utilizada para referenciar um atributo ou método da própria classe evitando ambiguidade em relação aos parâmetros ou variáveis declaradas dentro de um método da classe. Por exemplo, no método **setNome**, **this.nome** faz a referência ao atributo nome enquanto **nome** identifica o parâmetro que foi passado na chamada do método.

```
public setNome(nome: string) {
  this.nome = nome;
}
```

Encapsulamento: consiste na separação de características internas e externas de um determinado objeto. Este recurso é comumente utilizado para evitar o acesso direto aos atributos desse objeto, permitindo, desta forma, que apenas os métodos consigam alterar esses atributos. O conceito de **encapsulamento** é implementado através da definição da "visibilidade" de atributos e métodos através das palavras-reservadas **public**, **private** ou **protected**. Quando não definido na declaração do método ou atributo o encapsulamento público é adotado por padrão. No trecho de programa a seguir definimos que o atributo **nome** é privado, ou seja, apenas pode ser acessado por métodos da própria classe.

```
private nome: string;
```

Por outro lado, o método **getNome**, mostrado a seguir é público, sendo acessível por qualquer rotina, independente desta pertencer à classe ou não.

```
public getNome(): string {
  return this.nome;
}
```

Instância: Como abordado anteriormente, uma classe define o comportamento de um objeto através dos métodos e atributos. Porém, ela não realiza o armazenamento dos atributos e a execução dos métodos. Desta forma, torna-se necessário a criação de objetos a partir da classe definida, ou seja, um objeto pode ser entendido como uma instância de uma determinada classe, sendo que uma classe pode instanciar inúmeros objetos.

```
let pessoa: Pessoa = new Pessoa();
```

É importante observar que o método construtor define a maneira como o objeto será instanciado, ou seja, como o objeto será criado. Assim sendo, após a definição da classe, podemos instanciar um objeto e definir o estado (valor) de cada atributo:

```
let pessoa: Pessoa = new Pessoa();
pessoa.setNome('Cláudio Oliveira');
pessoa.setEndereco('Rua das Flores, 100');
pessoa.setTelefone('(11) 4567-8901');
```

```
console.log(pessoa.getNome() + '\n' +
  pessoa.getEndereco() + '\n' +
  pessoa.getTelefone());
```

Getters e **Setters:** dentro do conceito de **encapsulamento** é comum definirmos os atributos como privados, desta forma, para os atributos que precisam receber valores provenientes de fora da classe é necessário implementar um método de atribuição, normalmente conhecido como **Setter**. Por outro lado, atributos que precisam ter o seu conteúdo consultado por outras classes deverão implementar um método **Getter**.

A implementação dos métodos **Getters** e **Setters** pode ser realizada conforme mostrado no exemplo anterior, mas também é possível declarar os métodos usando as palavras reservadas **get** e **set**.

```
class Pessoa {
  private nome: string = '';
  private endereco: string = '';
  private telefone: string = '';

  constructor () {
  }

  public get nome(): string {
    return this._nome;
  }
  public set nome(value: string) {
    this._nome = value;
  }

  public get endereco(): string {
    return this._endereco;
  }
  public set endereco(value: string) {
    this._endereco = value;
  }

  public get telefone(): string {
    return this._telefone;
  }
  public set telefone(value: string) {
    this._telefone = value;
  }
}
```

pessoa-2.ts

Observando o trecho de programa a seguir, observe que os métodos podem ser acessados como se fossem atributos. Ou seja, usamos o operador de atribuição (=) para receber um valor, neste caso será executado o método **set**.

```
let pessoa: Pessoa = new Pessoa();
pessoa.nome = 'Cláudio Oliveira';
pessoa.endereco = 'Rua das Flores, 100';
pessoa.telefone = '(11) 4567-8901';
```

Por outro lado, para obter o retorno através do método **get**, basta evocar o nome do método, conforme mostra o seguinte código-fonte.

```
console.log('Nome: ' + pessoa.nome +
  '\nEndereço: ' + pessoa.endereco +
  '\nTelefone: ' + pessoa.telefone);
```

Polimorfismo: permite que um determinado método declarado em uma classe funcione de maneira diferente de acordo com o contexto no qual o objeto está inserido. Observe no exemplo o uso de polimorfismo no método **somar** possibilitando, desta forma, que o método possa ser usado recebendo dois números ou duas strings como parâmetros.

```
class Calculo {
  somar(num1: number, num2: number): number;

  somar(num1: string, num2: string): number;

  somar(num1: any, num2: any): any {
    return (parseFloat(num1) + parseFloat(num2));
  }
}

let calc: Calculo = new Calculo();
console.log(calc.somar(5.6, 6.9));
console.log(calc.somar('10.3', '21.4'));
```

calculo.ts

Observe no próximo programa que o **polimorfismo** também pode ser aplicado no método construtor.

```
class Ponto {
  private x: number;
  private y: number;

  constructor(x: string, y: string);
  constructor(x: number, y: number);
  constructor(x: string);
  constructor(x: any, y?: any) {
    this.x = parseFloat(x);
```

```
    if (!y)
      y = -1;
    this.y = parseFloat(y);
  }

  getPonto() {
    return this.x + ', ' + this.y;
  }
}

let pt1: Ponto = new Ponto(3, 5);
console.log('x, y = ' + pt1.getPonto());
```

ponto.ts

Associação: este conceito pode ser entendido como um recurso através do qual um objeto consegue utilizar componentes de um outro objeto. Por exemplo, todo veículo (objeto) possui um motor (objeto). Desta maneira, podemos saber a informação da potência de um determinado veículo a partir de uma informação (atributo) que está disponível no motor instalado nele.

```
class Motor {
  private litros: number;
  private cilindros: number;
  private potencia: number;
  private combustivel: string;

  constructor(litros: number, cilindros: number, potencia: number,
    combustivel: string) {
    this.litros = litros;
    this.cilindros = cilindros;
    this.potencia = potencia;
    this.combustivel = combustivel;
  }

  toString(): string {
    return '\n\tLitros: ' + this.litros +
      '\n\tCilindros: ' + this.cilindros +
      '\n\tPotência: ' + this.potencia +
      '\n\tCombustível: ' + this.combustivel;
  }
}

class Veiculo {
  private modelo: string;
  private fabricante: string;
  private anoFabricacao: number;
  private motor: Motor;

  constructor(modelo: string, fabricante: string, anoFabricacao: number,
```

```
    motor: Motor) {
    this.modelo = modelo;
    this.fabricante = fabricante;
    this.anoFabricacao = anoFabricacao;
    this.motor = motor;
  }

  toString(): string {
    return 'Modelo: ' + this.modelo +
      '\nFabricante: ' + this.fabricante +
      '\nAno de Fabricação: ' + this.anoFabricacao +
      '\nMotorização: ' + this.motor.toString();
  }
}

let m1vw: Motor = new Motor(1.0, 3, 70, 'Flex');
let gol: Veiculo = new Veiculo('Gol', 'Volkswagen', 2020, m1vw);
console.log(gol.toString());
```

veiculo.ts

Exercícios

1) Considerando o funcionamento de uma lâmpada, projetar uma classe que represente os estados possíveis e respectivos métodos.

2) Uma determinada loja precisa armazenar as seguintes características de seus produtos: código, descrição e preço. Especificar uma classe, com atributos e métodos, que permita realizar essa representação.

3) Uma empresa seguradora necessita descrever as seguintes informações dos veículos segurados: número do chassi, número da placa, fabricante, modelo, ano de fabricação e cor. Especificar uma classe, com atributos e métodos, que permita caracterizar essa necessidade.

4) Considerando um DVD especificar uma classe que permita representá-lo.

5) Representar uma classe chamada matemática que permita realizar uma das quatro operações básicas sobre dois números de ponto flutuante retornando, em seguida, o resultado.

Herança

O conceito de herança pode ser entendido como o recurso pelo qual uma determinada classe, denominada subclasse ou classe filha, pode receber atributos e métodos presentes em outra classe, neste caso chamada de superclasse ou classe pai.

Uma das características fundamentais da orientação a objetos é a capacidade de reaproveitamento de código, a qual pode ser implementada através da adoção do conceito de herança. O exemplo discutido a seguir mostra uma possibilidade de aplicação com herança.

Considerar como cenário uma loja, que mantém um registro de todos os produtos que são vendidos. Estes produtos possuem alguns atributos em comum como, por exemplo, código, descrição e preço. Porém, alguns produtos possuem algumas informações específicas:

- DVD: duração, região, idioma e legenda;
- Eletrônicos: tensão de operação e garantia;
- Livros: autor, número de páginas e idioma.

A primeira etapa consiste na definição da classe **Produto**. Para isso, inicialmente crie o arquivo "produto.ts", após criar a classe é importante observar que os atributos deverão ser declarados como **protected** para que possam ser acessados posteriormente pelas subclasses. Observe também o uso da palavra reservada **export** na declaração da classe para indicar que ela poderá ser acessada a partir de outros programas, mediante o uso da declaração **import**.

```
export class Produto {
  protected codigo: number;
  protected descricao: string;
  protected preco: number;

  constructor();
  constructor(codigo: number, descricao: string, preco: number);
  constructor(codigo?: any, descricao?: any, preco?: any) {
    this.codigo = codigo;
    this.descricao = descricao;
    this.preco = preco;
  }

  public setCodigo(codigo: number) {
    this.codigo = codigo;
  }

  public getCodigo(): number {
    return this.codigo;
  }

  public setDescricao(descricao: string) {
    this.descricao = descricao;
  }
```

```
  public getDescricao(): string {
    return this.descricao;
  }

  public setPreco(preco: number) {
    this.preco = preco;
  }

  public getPreco(): number {
    return this.preco;
  }

  public toString(): string {
    return "Código: " + this.codigo +
      "\nDescrição: " + this.descricao +
      "\nPreço: " + this.preco;
  }
}
```

heranca-1/produto.ts

Agora, como exemplo, criaremos o arquivo "eletronico.ts" que irá conter a classe **Eletronico** a qual, a partir do mecanismo de herança, receberá todos os atributos da superclasse Produto sendo necessária, desta forma, apenas a definição dos atributos e métodos específicos. Porém, para poder usar a classe **Produto** devemos realizar a sua importação, conforme pode ser notado na primeira linha do código-fonte apresentado a seguir.

```
import { Produto } from "./produto";

export class Eletronico extends Produto {
  private tensao: number;
  private garantia: string;

  constructor();
  constructor(codigo: number, descricao: string, preco: number, tensao:
number, garantia: string);
  constructor(codigo?: any, descricao?: any, preco?: any, tensao?: any,
garantia?: any) {
    super(codigo, descricao, preco);
    this.tensao = tensao;
    this.garantia = garantia;
  }

  public setTensao(tensao: number) {
    this.tensao = tensao;
  }

  public getTensao(): number {
    return this.tensao;
```

```
  }

  public setGarantia(garantia: string) {
    this.garantia = garantia;
  }

  public getGarantia(): string {
    return this.garantia;
  }

  public toString(): string {
    return super.toString() +
      "\nTensão: " + this.tensao +
      "\nGarantia: " + this.garantia;
  }
}
```

heranca-1/eletronico.ts

Como pode-se observar no código acima, no método construtor da classe **Eletrônico**, inicializamos os atributos da superclasse (**Produto**), ou seja, o código, a descrição e o preço, juntamente com os atributos da subclasse filha que são a tensão e a garantia.

O TypeScript oferece a palavra reservada **super** que permite realizar diretamente a chamada a um método construtor da superclasse. Assim sendo, o construtor da classe Eletrônico poderia ser reescrito conforme mostrado no código abaixo simplificando, desta forma, a implementação do método.

```
constructor(codigo?: any, descricao?: any, preco?: any, tensao?: any,
garantia?: any) {
  super(codigo, descricao, preco);
  this.tensao = tensao;
  this.garantia = garantia;
}
```

Note que os parâmetros **codigo**, **descricao** e **preco** são passados diretamente para o método construtor da classe Produto através da palavra reservada **super**. Também é importante salientar que este procedimento pode ser adotado para executar qualquer método da superclasse dentro da subclasse.

O último passo na elaboração deste exemplo consiste em criar uma classe que irá utilizar a classe **Eletrônico**. Neste caso, é necessário apenas a instanciação de um objeto da classe **Eletrônico**. Observe que os métodos da classe **Produto** estarão disponíveis graças ao mecanismo de herança:

```
import { Eletronico } from "./eletronico";

let eletr: Eletronico = new Eletronico();
```

```
eletr.setCodigo(11);
eletr.setDescricao('Televisor');
eletr.setPreco(2000);
eletr.setTensao(127);
eletr.setGarantia('12 meses');
console.log (eletr.toString());
```

heranca-1/principal-1.ts

Note no código-fonte a seguir que, como aplicamos o conceito de polimorfismo no método construtor, também podemos instanciar um objeto já passando diretamente o conjunto de valores que irão caracterizar o objeto.

```
import { Eletronico } from "./produto";

let eletr: Eletronico = new Eletronico(11, 'Televisor', 2000, 127,
  '12 meses');
console.log (eletr.toString());
```

heranca-1/principal-2.ts

A implementação dos métodos **Getters** e **Setters** também pode ser realizada usando as palavras reservadas **get** e **set**. Desta forma, a classe **Produto** pode ser reescrita e implementada conforme apresentado a seguir.

```
export class Produto {
  private _codigo: number;
  private _descricao: string;
  private _preco: number;

  constructor();
  constructor(codigo: number, descricao: string, preco: number);
  constructor(codigo?: any, descricao?: any, preco?: any) {
    this._codigo = codigo;
    this._descricao = descricao;
    this._preco = preco;
  }

  get codigo(): number {
    return this._codigo;
  }

  set codigo(value: number) {
    this._codigo = value;
  }

  get descricao(): string {
```

```
    return this._descricao;
  }

  set descricao(value: string) {
    this._descricao = value;
  }

  get preco(): number {
    return this._preco;
  }

  set preco(value: number) {
    this._preco = value;
  }

  public toString(): string {
    return "Código: " + this.codigo +
      "\nDescrição: " + this.descricao +
      "\nPreço: " + this.preco;
  }
}
```

heranca-2/produto.ts

Por sua vez, a classe **Eletronico** pode ser desenvolvida usando **set** e **get** da seguinte maneira:

```
import { Produto } from "./produto";

export class Eletronico extends Produto {
  private _tensao: number;
  private _garantia: string;

  constructor();
  constructor(codigo: number, descricao: string, preco: number, tensao:
number, garantia: string);
  constructor(codigo?: any, descricao?: any, preco?: any, tensao?: any,
garantia?: any) {
    super(codigo, descricao, preco);
    this._tensao = tensao;
    this._garantia = garantia;
  }

  public get tensao(): number {
    return this._tensao;
  }

  public set tensao(value: number) {
    this._tensao = value;
```

```
  }

  public get garantia(): string {
    return this._garantia;
  }

  public set garantia(value: string) {
    this._garantia = value;
  }

  public toString(): string {
    return super.toString() +
      "\nTensão: " + this.tensao +
      "\nGarantia: " + this.garantia;
  }
}
```

heranca-2/eletronico.ts

A seguir temos um exemplo de criação de um objeto a partir da classe **Eletronico**.

```
import { Eletronico } from "./eletronico";

let eletr: Eletronico = new Eletronico();
eletr.codigo = 11;
eletr.descricao = 'Televisor';
eletr.preco = 2000;
eletr.tensao = 127;
eletr.garantia = '12 meses';
console.log (eletr.toString());
```

heranca-2/principal.ts

Exercícios

1) Implementar as demais classes descritas no cenário proposto: DVD e Livro.

2) Uma empresa de aluguel de veículos mantém um banco de dados para todos os veículos de sua frota. Para todos os veículos são armazenados o número do chassi, o número da placa, o fabricante, o modelo, o ano de fabricação e a cor. Algumas informações são incluídas para determinados tipos de veículos: - caminhões: capacidade de carga, número de eixos; - carros esportes: potência, capacidade de aceleração; ônibus: quantidade de passageiros, número de eixos; - off road: altura em relação ao solo, tipo de tração. Realizar a definição e implementação das classes, com respectivos atributos e métodos, com base no exposto acima.

3) Considerando os diversos tipos de pessoas que trabalham em uma empresa, definir um conjunto de classes, utilizando o mecanismo de herança, que caracterize os seguintes tipos de funcionários:

a) Todos os funcionários possuem atributos de número de registro, nome, função, salário e situação.
b) Funcionários efetivos possuem atributos de data de admissão e demissão.
c) Os funcionários temporários possuem data de início do contrato e a duração em meses dele.
d) Os funcionários terceirizados apresentam data de início, duração do contrato e o nome da empresa prestadora de serviços.

Interfaces e Implementações

Em sistemas de computação, o conceito de interface pode ser aplicado de várias formas. Por exemplo, uma interface com o usuário é a camada de um sistema que tem como finalidade permitir e facilitar a interação entre o usuário e os diversos processos da aplicação. O conceito de interface de programação ou API (*Applications Programming Interface*) refere-se a um conjunto de funções que podem ser evocadas por programas aplicativos para utilizar os serviços oferecidos pelo Sistema Operacional. A utilização de interfaces bem definidas permite a interoperabilidade entre os diversos componentes de um sistema.

O conceito de interfaces na orientação a objetos é utilizado para definir um tipo abstrato de dados que descreve o comportamento que é visível externamente de uma classe, objeto ou qualquer outra entidade. No caso de uma classe ou objeto, a interface inclui um conjunto de operações e suas respectivas assinaturas de métodos. Deste modo, é possível conceituar interface como a especificação de um tipo, que é formado por um nome e um conjunto de métodos, os quais não podem conter nenhuma implementação.

Normalmente, os nomes de interfaces são criados da mesma maneira que os nomes de classes, ou seja, utilizando-se substantivos simples ou adjetivados, no singular, com cada palavra que compõe o nome da interface iniciando com letra maiúscula. Uma característica essencial da utilização das interfaces é a ausência de implementação, tanto em relação aos métodos quanto as estruturas de dados, sendo que a única exceção admitida é a definição de constantes. Todas as operações de uma interface são públicas, não sendo exigido o modificador **public** nas assinaturas dos métodos. A seguir é mostrado um exemplo de interface:

```
interface ICliente {
  setCPF(cpf: string): void;
  getCPF(): string;
  setNome(nome: string): void;
  getNome(): string;
  setTelefone(telefone: string): void;
  getTelefone(): string;
}
```

interface1/cliente.ts

A implementação de um tipo especificado por uma interface deve sempre ser realizada através da definição de uma classe. Sendo que uma classe pode implementar um número qualquer de interfaces. Por outro lado, diferentes classes podem realizar a implementação de maneiras diferentes de uma mesma interface através de construções distintas para os métodos definidos na interface. A função principal de uma interface é definir os componentes visíveis (públicos).

No exemplo a seguir é mostrado como realizar uma das possíveis implementações para a interface **Cliente** definida anteriormente, sendo importante observar que, neste momento, o funcionamento de cada método é explicitado.

```
class Cliente implements ICliente {
```

```
  private cpf: string = '';
  private nome: string = '';
  private telefone: string = '';

  public setCPF(cpf: string): void {
    this.cpf = cpf;
  }

  public getCPF(): string {
    return this.cpf;
  }

  public setNome(nome: string): void {
    this.nome = nome;
  }

  public getNome(): string {
    return this.nome;
  }

  public setTelefone(telefone: string): void {
    this.telefone = telefone;
  }

  public getTelefone(): string {
    return this.telefone;
  }

  public toString(): string {
    return "CPF: " + this.cpf +
      "\nNome: " + this.nome +
      "\nTelefone: " + this.telefone;
  }
}
```

interface1/cliente.ts

Desta forma, a implementação consiste na definição das operações especificadas pelas interfaces, através de métodos que devem obrigatoriamente apresentar as mesmas assinaturas definidas pela respectiva interface. Por último, no código-fonte a seguir, é instanciado um objeto da classe **Cliente**.

```
let cli: Cliente = new Cliente();
cli.setCPF('111.111.111-11');
cli.setNome('Cláudio Oliveira');
cli.setTelefone('(11) 4567-8901');
console.log(cli.toString());
```

A implementação dos métodos Getters e Setters também pode ser realizada usando as palavras reservadas **get** e **set**. Desta forma, a interface ICliente e a classe Cliente podem ser reescritas e implementadas conforme mostrado a seguir.

```
interface ICliente {
  set cpf(valor: string);
  get cpf(): string;
  set nome(valor: string);
  get nome(): string;
  set telefone(valor: string);
  get telefone(): string;
}

class Cliente implements ICliente {
  private _cpf: string = '';
  private _nome: string = '';
  private _telefone: string = '';

  public set cpf(valor: string){
    this._cpf = valor;
  }

  public get cpf(): string {
    return this._cpf;
  }

  public set nome(valor: string) {
    this._nome = valor;
  }

  public get nome(): string {
    return this._nome;
  };

  public set telefone(valor: string) {
    this._telefone = valor;
  }

  public get telefone(): string {
    return this._telefone;
  }

  public toString(): string {
    return "CPF: " + this._cpf +
      "\nNome: " + this._nome +
      "\nTelefone: " + this._telefone;
  }
}
```

```
let cli: Cliente = new Cliente();
cli.cpf = '111.111.111-11';
cli.nome = 'Cláudio Oliveira';
cli.telefone = '(11) 4567-8901';
console.log(cli.toString());
```

interface2/cliente.ts

Entrada de Dados

Quando é necessário solicitar alguma interação com o usuário, podemos realizar a entrada de dados através do console (teclado). Para isso, devemos utilizar a interface **readLine**, sendo que previamente devemos instalar o módulo **@types/node** que adiciona suporte a definição de tipo de dado.

```
npm i --save-dev @types/node
```

Agora podemos implementar o programa simples que solicitará ao usuário para digitar o seu nome e depois exibirá uma mensagem de saudação, conforme ilustra o exemplo a seguir.

```
import readline from 'readline';

const teclado = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

teclado.question('Digite o seu nome: ', (nome: string) => {
  console.log('Olá ' + nome);
  teclado.close();
});
```

teclado.ts

Observe que após a criação do objeto teclado, devemos usar o método **question** para possibilitar que o usuário realize a digitação, sendo que o texto digitado será armazenado uma string. No exemplo seguinte vamos solicitar ao usuário para digitar dois números e, em seguida, iremos calcular e exibir o valor da soma. Note que os dados fornecidos sempre serão armazenados em variáveis do tipo string, sendo, se necessário, realizar a conversão para o tipo de dado desejado.

```
import readline from 'readline';

const teclado = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

teclado.question('Digite o primeiro valor: ', (valor1: string) => {
  teclado.question('Digite o segundo valor: ', (valor2: string) => {
    let soma = parseFloat(valor1) + parseFloat(valor2);
```

```
    console.log("A soma é " + soma);
    teclado.close();
  });
});
```

somar.ts

Aplicações para a Internet

Vamos usar o módulo **express** para implementar um servidor de páginas web através do Node.js/TypeScript. Desta maneira, o primeiro passo consiste em instalar o express que é projetado de modo a facilitar a criação de aplicações para a web. A instalação deve ser feita na pasta criada para a aplicação, onde o npm deverá ser executado da maneira indicada a seguir. Também instale o módulo **@types/node** que adiciona suporte a definição de tipo de dado.

```
npm install express
npm i --save-dev @types/node
```

Em seguida, vamos elaborar o programa que irá criar um servidor local executando da porta 8080.

```
let fs = require('fs');
let express = require('express');
let app = express();

let servidor = app.listen(8080, function() {
  let porta = servidor.address().port;
  console.log("Servidor executando na porta %s", porta);
});

app.get('/', function (req: any, res: any) {
  fs.readFile('ola.html', function(erro: any, dado: any) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(dado);
    res.end();
  });
});
```

servidor-ola.ts

Analisando o programa, observe que o método **app.listen** irá criar um servidor na porta TCP indicada, enquanto **app.get** determina a ação que será tomada de acordo com o caminho passado que, no nosso exemplo, é a raiz do site (caractere '/'). Então o método **fs.readFile** irá abrir o arquivo HTML mostrado a seguir que, então, será enviado para o navegador através do método **res.write**.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Olá TypeScript</title>
    <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.mi
n.css">
  </head>
  <body>
    <div class="container">
      <div class="jumbotron">
        <h1>Olá TypeScript</h1>
      </div>
    </div>
  </body>
</html>
```

ola.html

Utilize o TypeScript para executar o arquivo servidor-ola.ts conforme mostrado a seguir.

```
ts-node servidor-ola.ts
```

Então, no navegador, abra o site local na porta 8080, isto é, digite localhost:8080/ na barra de endereço e observe que a página HTML será exibida, conforme ilustra a Figura 1.

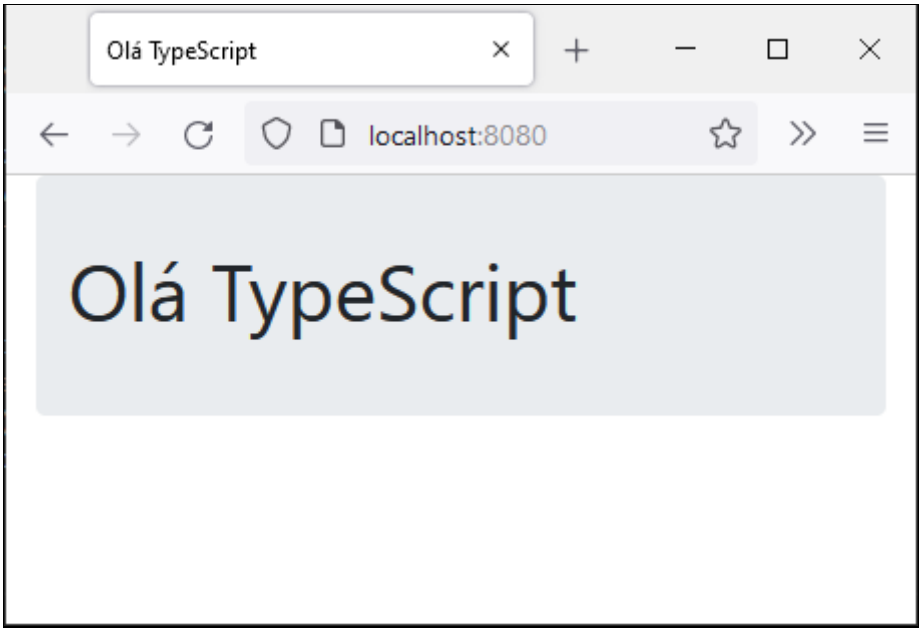

Figura 1: Página carregada

Quando desejar, no terminal, utilize a combinação de teclas “Ctrl C” para finalizar a execução do servidor.

Interação com HTML

Através do Express podemos interagir com elementos HTML e com outros arquivos frequentemente presentes em projetos de aplicações para a Internet como, por exemplo, outros arquivos HTML, folhas de estilo (CSS), imagens e Javascript. No exemplo seguinte ilustramos este recurso, possibilitando carregar aleatoriamente uma imagem que será exibida em uma página HTML.

Antes de implementar o programa, crie uma pasta chamada img e copie para ela imagens representando cada uma das faces de um dado.

```
let express = require('express');
let app = express();

let servidor = app.listen(8080, function() {
  let porta = servidor.address().port;
  console.log("Servidor executando na porta %s", porta);
});

app.use(express.static('img'));
app.get('/', function (req: any, res: any) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  let face = Math.floor(Math.random() * 6 + 1);
  res.end('<h1>Jogo de Dados</h1><img src="Dado_' + face + '.jpg">');
});
```

dados/dados.ts

Note que o programa é bastante parecido com o que foi apresentado no exemplo anterior. Mas, neste caso, usamos o método **app.use** para informar ao servidor que iremos usar um conteúdo estático, ou seja, os arquivos que estão na pasta img. Em seguida, sorteamos um valor entre 1 e 6 que corresponderá à face do dado que deverá ser exibida. Por fim, o médodo **res.end** será usado para exibir a respectiva imagem, conforme ilustra a Figura 2.

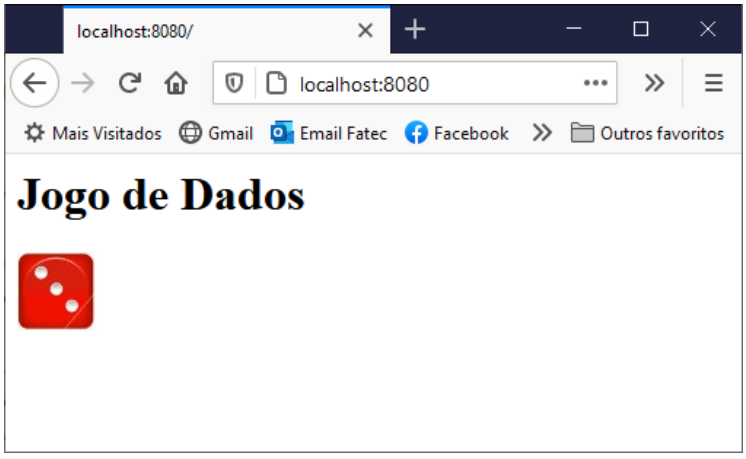

Figura 2: Página exibindo a face sorteada

Experimente recarregar a página várias vezes e observe que uma nova face será mostrada.

Exercícios

1) Considerando um dado de jogo sendo lançado 50 vezes, criar uma aplicação para a Internet que mostre todos os lançamentos e, ao final, a quantidade de vezes que cada um dos lados foi sorteado.

2) Considerando uma moeda sendo lançada 30 vezes, criar uma aplicação web que mostre o resultado de todos os lançamentos e, ao final, a porcentagem de caras e de coroas que ocorreram.

Utilização de Formulários

Através do TypeScript é bastante simples trabalhar com os dados provenientes de formulários HTML. Esta próxima aplicação irá demonstrar como podemos obter os valores dos campos de formulários e processá-los no servidor.

Começamos criando a página HTML que irá implementar o formulário, vamos usar apenas dois campos, o nome da pessoa e o ano que ela nasceu, além de um botão para realizar a submissão (envio) do formulário.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Idade</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.mi
n.css">
  </head>
  <body>
    <div class="container">
      <div class="jumbotron">
        <h1>Idade</h1>
      </div>
      <form id="form" method="POST" action="/idade">
        <div class="form-group">
          Nome: <input type="text" name="nome" class="form-control" />
        </div>
        <div class="form-group">
          Ano de Nascimento: <input type="text" name="anonasc"
class="form-control" />
        </div>
        <input type="submit" value="Calcular" class="btn btn-primary" />
      </form>
    </div>
  </body>
</html>
```

idadeweb/idade-form.html

Observe no trecho de código-fonte destacado a seguir, que os dados do formulário serão submetidos (enviados) para o endereço “/idade” do servidor local, que será posteriormente implementado.

```
<form id="form" method="POST" action="/idade">
```

Em seguida, já vamos criar a página HTML que irá receber os dados processados pelo servidor. Observe que os dados que serão posteriormente enviados para esta página estão identificados como **{{nome}}**, **{{anonasc}}** e **{{idade}}**.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Idade</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.mi
n.css">
  </head>
  <body>
    <div class="container">
      <div class="jumbotron">
        <h1>Idade</h1>
      </div>
      Nome: {{nome}}<br>
      Ano de Nascimento: {{anonasc}}<br>
      Idade: {{idade}} anos.
    </div>
  </body>
</html>
```

idadeweb/idade-res.html

Uma vez criadas as páginas com o conteúdo HTML passaremos para a criação da aplicação em TypeScript.

```
let fs = require('fs');
let http = require("http");
let express = require('express');
let app = express();
let bodyParser = require('body-parser');
let urlencodedParser = bodyParser.urlencoded({ extended: true });

let servidor = app.listen(8080, function() {
  let porta = servidor.address().port;
  console.log("Servidor executando na porta %s", porta);
});

app.get('/', function (req: any, res: any) {
  fs.readFile('idade-form.html', function(erro: any, dado: any) {
    res.writeHead(200, {'Content-Type': 'text/html'});
```

```
    res.write(dado);
    res.end();
  });
});

app.post('/idade', urlencodedParser, function (req: any, res: any){
  fs.readFile('idade-res.html', function(erro: any, dado: any) {
    let hoje: Date = new Date();
    let nome: string = req.body.nome;
    let anonasc: number = parseInt(req.body.anonasc);
    let idade: number = hoje.getFullYear() - anonasc;
    dado = dado.toString().replace("{{nome}}", nome);
    dado = dado.toString().replace("{{anonasc}}", anonasc);
    dado = dado.toString().replace("{{idade}}", idade);
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(dado);
    res.end();
  });
});
```

idadeweb/servidor-idade.ts

Utilize o TypeScript para executar o arquivo servidor-idade.ts, conforme mostrado a seguir.

```
ts-node servidor-idade.ts
```

No navegador, abra o site digitando localhost:8080/ na barra de endereço. Preencha o formulário de maneira similar à mostrada na Figura 3.

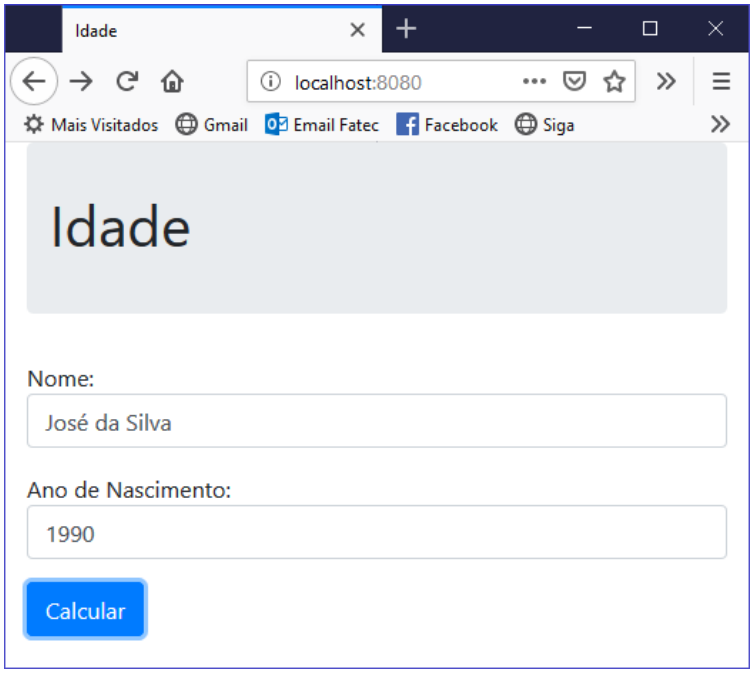

Figura 3: Formulário para cálculo da idade

Quando pressionar o botão “Calcular” o servidor irá processar os dados e enviar o resultado para a página “idade-res.html”, conforme ilustra a Figura 4.

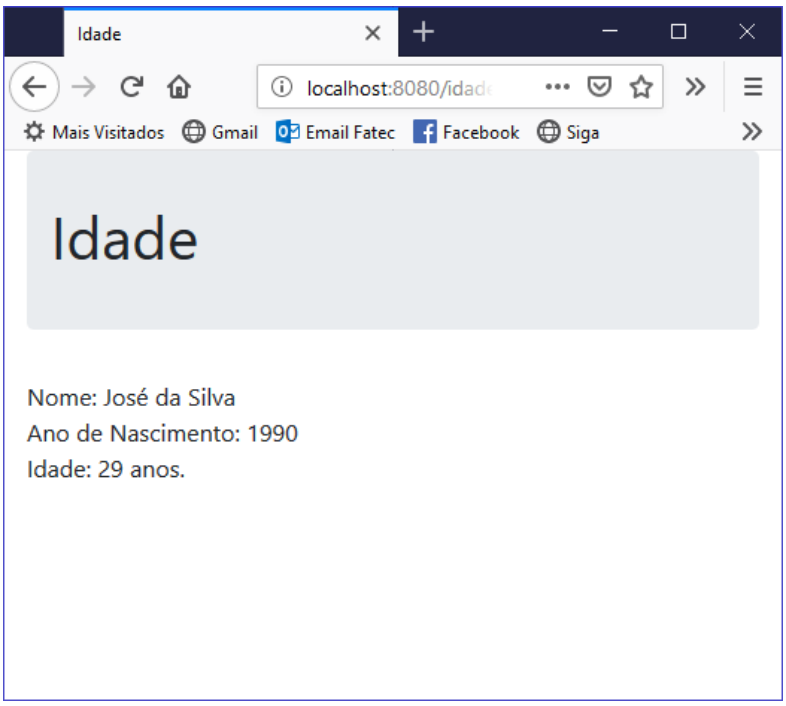

Figura 4: Exibição dos dados processados

Após a execução desta aplicação vamos analisar, em detalhes, o seu funcionamento. O primeiro bloco de programa, mostrado a seguir, consiste em realizar a carga dos módulos e definir a variável app e as variáveis para realizar o parse, ou seja, a obtenção dos elementos que compõem o endereço da página (URL) e os dados submetidos.

```
let fs = require('fs');
let http = require("http");
let express = require('express');
let app = express();
let bodyParser = require('body-parser');
let urlencodedParser = bodyParser.urlencoded({ extended: true });
```

A seguir, observe no trecho a seguir, que ele será o responsável pela criação do servidor que irá receber as solicitações do navegador através da porta TCP 8080.

```
let servidor = app.listen(8080, function() {
  let porta = servidor.address().port;
  console.log("Servidor executando na porta %s", porta);
});
```

Quando o servidor receber a requisição do navegador para acessar a raiz do site, o trecho de programa mostrado a seguir será responsável por enviar o conteúdo do arquivo "idade-form.html" para o navegador.

```
app.get('/', function (req: any, res: any) {
  fs.readFile('idade-form.html', function(erro: any, dado: any) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(dado);
    res.end();
  });
});
```

Concluindo esta aplicação, no trecho de código-fonte, apresentado a seguir, quando o servidor receber a requisição "/idade" este irá carregar o arquivo "idade-res.html" e obter os dados que foram enviados através da submissão do formulário. Para obter os dados, usamos a propriedade **req.body.nome**, onde nome corresponde ao valor da propriedade id do campo do formulário.

Em seguida, usamos o método **replace** para trocar os marcadores **{{nome}}**, **{{anonasc}}** e **{{idade}}** pelos dados obtidos e processados pelo servidor.

```
app.post('/idade', urlencodedParser, function (req: any, res: any){
  fs.readFile('idade-res.html', function(erro: any, dado: any) {
    let hoje: Date = new Date();
    let nome: string = req.body.nome;
    let anonasc: number = parseInt(req.body.anonasc);
    let idade: number = hoje.getFullYear() - anonasc;
    dado = dado.toString().replace("{{nome}}", nome);
    dado = dado.toString().replace("{{anonasc}}", anonasc);
    dado = dado.toString().replace("{{idade}}", idade);
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(dado);
    res.end();
  });
});
```

Exercícios

1) Desenvolver uma aplicação Internet em TypeScript que receba quatro números reais digitados pelo usuário através de um formulário HTML e, em seguida, calcule e exiba a valor da média dos números em uma nova página HTML.

2) Desenvolver uma aplicação web em TypeScript para uma determinada loja que precisa calcular o preço de venda de um produto. O cálculo deverá ser efetuado através da multiplicação do

preço unitário pela quantidade vendida e, posteriormente, subtrair o valor do desconto. Considerar todas as variáveis do tipo de dado real, que serão digitadas pelo usuário através de um formulário HTML.

3) Considerando o desenvolvimento de uma aplicação TypeScript para a Internet e sendo que a Lei de Ohm define que a resistência (R) de um condutor é obtida através da divisão da tensão aplicada (V) dividida pela intensidade de corrente elétrica (A). Desta forma, a partir de uma tensão e corrente, digitadas pelo usuário através de um formulário HTML, calcule e mostre o valor da resistência.

4) Desenvolver um servidor, usando TypeScript, que apresente uma página HTML contendo a data e hora atuais.

5) Desenvolver um servidor, usando TypeScript, que simule um uma página HTML um dado de jogo sendo lançado aleatoriamente por 50 vezes. A mesma página deverá exibir quantas vezes cada uma das faces foi sorteada.

6) Desenvolver um servidor, usando TypeScript, que simule um uma página HTML uma moeda sendo lançada aleatoriamente por 200 vezes. A mesma página deverá exibir a porcentagem de caras e coroas que foram sorteadas.

7) Considerando que a aprovação de um aluno em determinada disciplina requer uma média final maior ou igual a 6,0 (seis). Elaborar uma aplicação TypeScript para a web que receba através de um formulário HTML, duas notas, realize o cálculo da média, exiba o valor calculado e uma imagem indicando se o aluno está aprovado ou reprovado.

8) Realizar o desenvolvimento de um servidor TypeScript que disponibilize uma página HTML que irá receber as informações através de um formulário e realizar a validação dos dados digitados, considerando que uma seguradora de veículos precisa calcular o valor da apólice com base nas seguintes informações: nome, sexo e ano de nascimento do segurado, marca, modelo, ano de fabricação, valor do veículo e porcentagem do bônus. As seguintes validações deverão ser realizadas na própria página HTML que contém o formulário:
   a) O campo sexo deverá aceitar apenas F (Feminino) ou M (Masculino).
   b) O campo ano de nascimento deve aceitar um valor entre 2001 e 1901.
   c) O campo ano de fabricação deverá ser um valor inteiro positivo.
   d) O campo valor do veículo deve ser um número real positivo.
   e) O campo porcentagem do bônus deverá ser um número real entre 0 e 25.

   Quando o formulário for submetido o servidor deverá determinar o valor da apólice, a partir dos seguintes critérios para cálculo:
   a) Para veículos 2010 ou mais recentes o valor da apólice é de 1,25% do valor do veículo, veículos entre 2009 e 2000 o valor da apólice é de 1,75% do valor do veículo, veículos entre 1999 e 1980 o valor da apólice é de 2,00% e para os demais anos de fabricação devemos utilizar 2,50% como base de cálculo.
   b) Caso o segurado seja do sexo feminino aplicar um desconto 10% sobre o valor calculado no item a, caso contrário, acrescer 5% ao valor calculado no item a.
   c) Se o segurado possuir menos de 30 anos ou mais de 60 anos, acrescentar 20% ao valor da apólice após os cálculos realizados no item a e no item b.

d) A partir do valor apurado nos itens a, b e c aplicar o desconto com base na porcentagem de bônus informada pelo usuário.

Por fim, uma página HTML deverá ser exibida apresentando o valor da apólice.

Arquitetura em Camadas

Aplicações orientadas a objeto comumente adotam o conceito de projeto de software em múltiplas camadas, no qual cada camada tem uma responsabilidade específica, por exemplo: apresentação (interface com o usuário), regras de negócio e acesso a dados entre outras, conforme ilustra a Figura 5 a seguir.

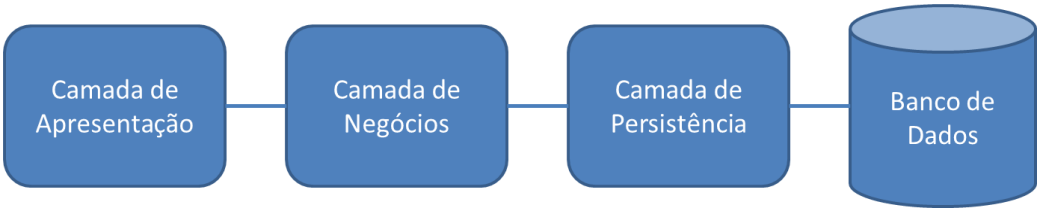

Figura 5: Modelo em camadas

A ideia de se desenvolver um software em camadas está fundamentada sobre o conceito de separação de tarefas e responsabilidades: diferentes serviços disponibilizados pelo programa, classificados de acordo com a sua função, são desenvolvidos de forma separada. A implementação do conceito de camadas através do TypeScript é realizada através da separação das tarefas através da criação e utilização de classes.

Como exemplo, vamos desenvolver a mesma aplicação para cálculo da idade. Porém, aplicando o conceito de camadas iremos isolar os elementos da interface com o usuário das regras de negócios. Desta forma, crie a pasta idadewebclasse e copie os arquivos idadeweb/idade-form.html e idadeweb/idade-res.html usados no exemplo anterior.

Em seguida crie a classe **Idade** que irá conter os atributos para nome e ano de nascimento, além de realizar o cálculo da idade.

TS

```
export class Idade {
  private _nome: string = '';
  private _anoNasc: number= 0;

  public get nome(): string {
    return this._nome;
  }

  public set nome(value: string) {
    this._nome = value;
  }

  public get anoNasc(): number {
    return this._anoNasc;
  }

  public set anoNasc(value: number) {
    this._anoNasc = value;
  }
```

```
  public getIdade(): number {
    let hoje = new Date();
    let idade =  hoje.getFullYear() - this._anoNasc;
    return idade;
  }
}
```

idadewebclasse/idade.ts

O passo seguinte consiste em criar o servidor que será responsável pelo envio das páginas para o navegador (cliente). Também irá obter os dados do formulário e repassá-los para a classe **Idade**, conforme mostrado no código-fonte apresentado a seguir.

```
import {Idade} from './idade';

let fs = require('fs');
let http = require("http");
let express = require('express');
let app = express();
let bodyParser = require('body-parser');
let urlencodedParser = bodyParser.urlencoded({ extended: true });

let servidor = app.listen(8080, function() {
  let porta = servidor.address().port;
  console.log("Servidor executando na porta %s", porta);
});

app.get('/', function (req: any, res: any) {
  fs.readFile('idade-form.html', function(erro: any, dado: any) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(dado);
    res.end();
  });
});

app.post('/idade', urlencodedParser, function (req: any, res: any){
  fs.readFile('idade-res.html', function(erro: any, dado: any) {
    let idade: Idade = new Idade();
    idade.nome = req.body.nome;
    idade.anoNasc = parseInt(req.body.anonasc);
    dado = dado.toString().replace("{{nome}}", idade.nome);
    dado = dado.toString().replace("{{anonasc}}", idade.anoNasc);
    dado = dado.toString().replace("{{idade}}", idade.getIdade());
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(dado);
    res.end();
  });
});
```

idadewebclasse/servidor-idade.ts

Utilize o TypeScript para executar o arquivo servidor-idade.ts.

```
ts-node servidor-idade.ts
```

No navegador, abra o site digitando localhost:8080/ na barra de endereço e preencha o formulário.

Botões de Rádio e Caixas de Seleção

Botões de rádio e caixas de seleção, possibilitam a escolha de um item dentro de um conjunto de opções. Com o intuito de ilustrar este conceito, no exemplo a seguir, vamos criar uma página que permita realizarmos a conversão de temperatura a partir das possibilidades mostradas na tabela a seguir.

De	Para	Fórmula
Celsius	Fahrenheit	°F = °C × 1,8 + 32
Fahrenheit	Celsius	°C = (°F − 32) / 1,8
Celsius	Kelvin	K = °C + 273,15
Kelvin	Celsius	°C = K − 273,15
Fahrenheit	Kelvin	K = (°F + 459,67) / 1,8
Kelvin	Fahrenheit	°F = K × 1,8 - 459,67

A aplicação para a Internet irá usar o conceito de camadas, desta forma, além do programa em TypeScript que irá implementar o servidor e das páginas em HTML que realizarão a interface com o usuário, será criada uma classe que realizará os cálculos indicados na tabela.

Desta forma, na página que contém o formulário vamos criar um campo para que a temperatura seja digitada e, como existem seis opções de conversão, vamos criar um botão de rádio para cada possibilidade.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Temperatura</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.mi
n.css">
  </head>
  <body>
    <div class="container">
      <div class="jumbotron">
        <h1>Conversão de Temperatura</h1>
      </div>
      <form id="form" method="POST" action="/temperatura">
        <div class="form-group">
          Nome: <input type="text" name="valor" class="form-control" />
        </div>
        Escolha uma opção:
        <div class="form-check">
          <input type="radio" name="conversao" value="1" class="form-
check-input" checked>Celsius para Fahrenheit<br />
          <input type="radio" name="conversao" value="2" class="form-
check-input">Fahrenheit para Celsius<br />
```

```
          <input type="radio" name="conversao" value="3" class="form-
check-input">Celsius para Kelvin<br />
          <input type="radio" name="conversao" value="4" class="form-
check-input">Kelvin para Celsius<br />
          <input type="radio" name="conversao" value="5" class="form-
check-input">Fahrenheit para Kelvin<br />
          <input type="radio" name="conversao" value="6" class="form-
check-input">Kelvin para Fahrenheit<br /><br />
        </div>
        <input type="submit" value="Calcular" class="btn btn-primary" />
      </form>
    </div>
  </body>
</html>
```

temperatura/temp-form.html

Na Figura 6 podemos observar a página criada, sendo relevante observar que os botões de rádio são definidos com o mesmo nome de forma a atuarem em conjunto, ou seja, quando uma opção e seleciona a anterior automaticamente e desmarcada, garantindo que sempre, apenas uma única escolha é realizada.

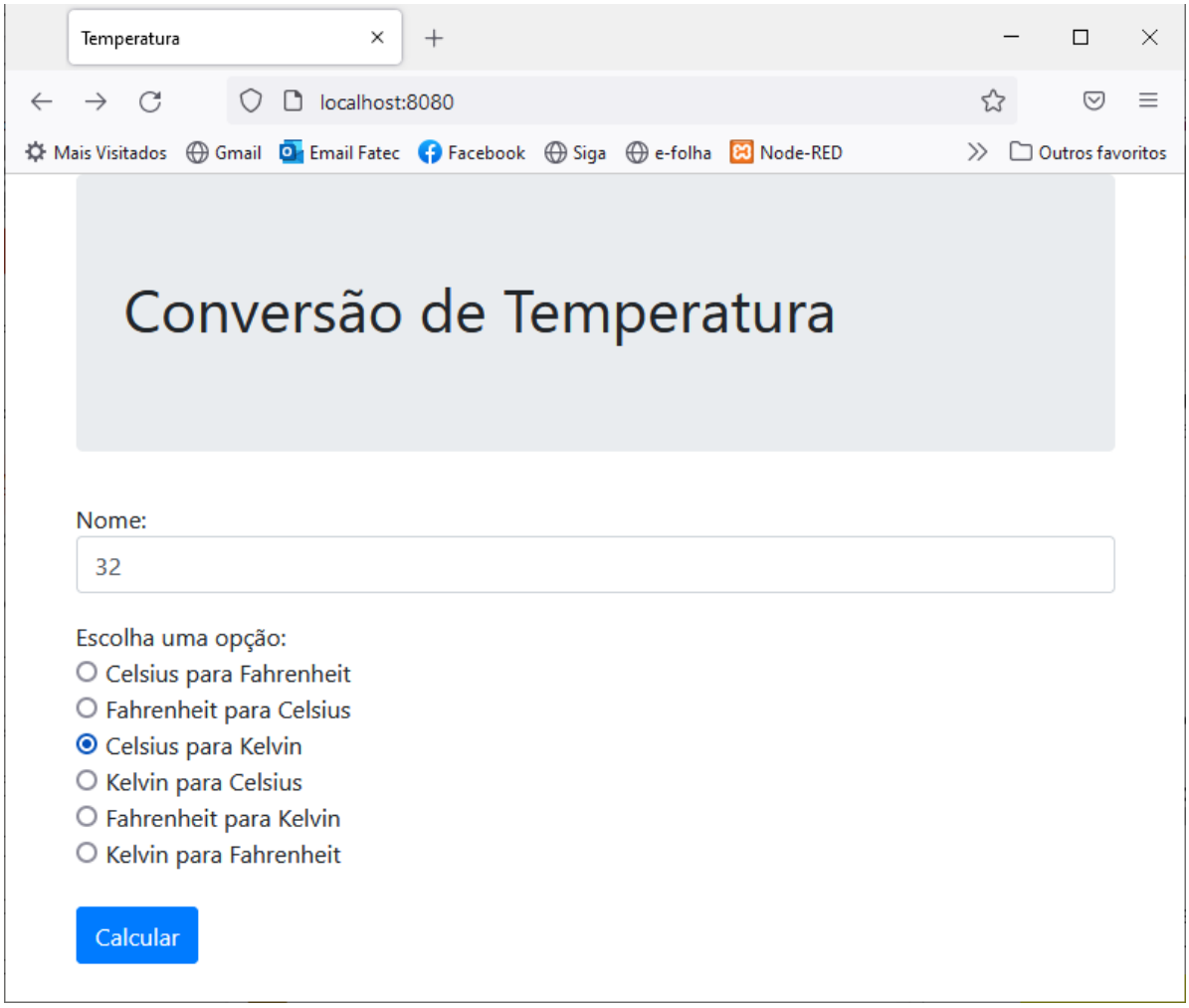

Figura 6: Botões de Rádio

Na sequência, também será criada uma página que será usada para exibir o resultado.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Temperatura</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.mi
n.css">
  </head>
  <body>
    <div class="container">
      <div class="jumbotron">
        <h1>Conversão de Temperatura</h1>
      </div>
      {{mensagem}}
    </div>
  </body>
</html>
```

temperatura/temp-res.html

Após a criação da interface com o usuário (camada de apresentação), passamos para o desenvolvimento do servidor. Além das funcionalidades abordadas nos exemplos anteriores, é importante notar que será obtida, através da obtenção dos dados do formulário, a temperatura e a opção de conversão escolhida pelo usuário e passadas para a classe **Temperatura**, onde serão aplicadas as devidas regras de negócio e realizados os cálculos pertinentes (camada de negócios).

```
import {Temperatura} from './temperatura';

let fs = require('fs');
let http = require("http");
let express = require('express');
let app = express();
let bodyParser = require('body-parser');
let urlencodedParser = bodyParser.urlencoded({ extended: true });

let servidor = app.listen(8080, function() {
  let porta = servidor.address().port;
  console.log("Servidor executando na porta %s", porta);
});

app.get('/', function (req: any, res: any) {
  fs.readFile('temp-form.html', function(erro: any, dado: any) {
```

```
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(dado);
    res.end();
  });
});

app.post('/temperatura', urlencodedParser, function (req: any, res: any){
  fs.readFile('temp-res.html', function(erro: any, dado: any) {
    let temp: Temperatura = new Temperatura();
    let mensagem: string = temp.converter(parseFloat(req.body.valor),
parseInt(req.body.conversao));
    dado = dado.toString().replace("{{mensagem}}", mensagem);
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(dado);
    res.end();
  });
});
```

temperatura/servidor-temp.ts

Por último, a classe **Temperatura**, com base nos dados recebidos e usando a estrutura de seleção **switch**, irá aplicar a respectiva fórmula e retornar uma mensagem contendo o resultado do processamento.

```
export class Temperatura {
  public converter(valor: number, opcao: number): string {
    let res: number = 0.0;
    let msg: string = '';
    switch (opcao){
      case 1:
        res = valor * 1.8 + 32;
        msg = valor + '°C equivale a ' + res + '°F';
        break;
      case 2:
        res = (valor - 32) / 1.8;
        msg = valor + '°F equivale a ' + res + '°C';
        break;
      case 3:
        res = valor + 273.15;
        msg = valor + '°C equivale a ' + res + 'K';
        break;
      case 4:
        res = valor - 273.15;
        msg = valor + 'K equivale a ' + res + '°C';
        break;
      case 5:
        res = (valor + 459.67) / 1.8;
        msg = valor + '°F equivale a ' + res + 'K';
        break;
```

```
      default:
        res = valor * 1.8 - 459.67;
        msg = valor + 'K equivale a ' + res + '°F';
    }
    return msg;
  }
}
```

temperatura/temperatura.ts

Ao executar a aplicação, preencher e submeter o formulário, a página de resultados será exibida de maneira similar à apresentada na Figura 7.

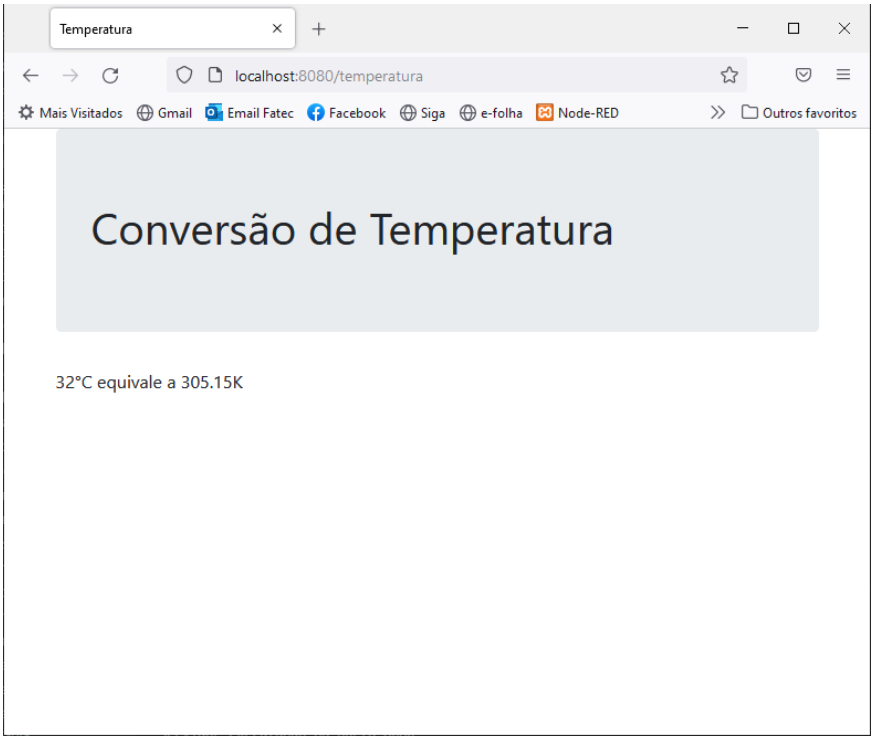

Figura 7: Página de resultados

As caixas de seleção têm funcionamento análogo aos botões de rádio. Desta forma, para exemplificar o seu uso, alteramos apenas a página que implementa o formulário da maneira indicada no código-fonte apresentado a seguir.

</>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Temperatura</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.mi
n.css">
  </head>
```

```
  <body>
    <div class="container">
      <div class="jumbotron">
        <h1>Conversão de Temperatura</h1>
      </div>
      <form id="form" method="POST" action="/temperatura">
        <div class="form-group">
          Nome: <input type="text" name="valor" class="form-control" />
        </div>
        <div class="form-group">
          Escolha uma opção:
          <select name="conversao" class="form-control">
            <option value="1">Celsius para Fahrenheit</option>
            <option value="2">Fahrenheit para Celsius</option>
            <option value="3">Celsius para Kelvin</option>
            <option value="4">Kelvin para Celsius</option>
            <option value="5">Fahrenheit para Kelvin</option>
            <option value="6">Kelvin para Fahrenheit</option>
        </select>
        </div>
        <input type="submit" value="Calcular" class="btn btn-primary" />
      </form>
    </div>
  </body>
</html>
```

temperatura/temp-form.html

Na Figura 8 temos a página com a caixa de seleção. Note que não é necessário alterar nenhuma outra rotina do programa.

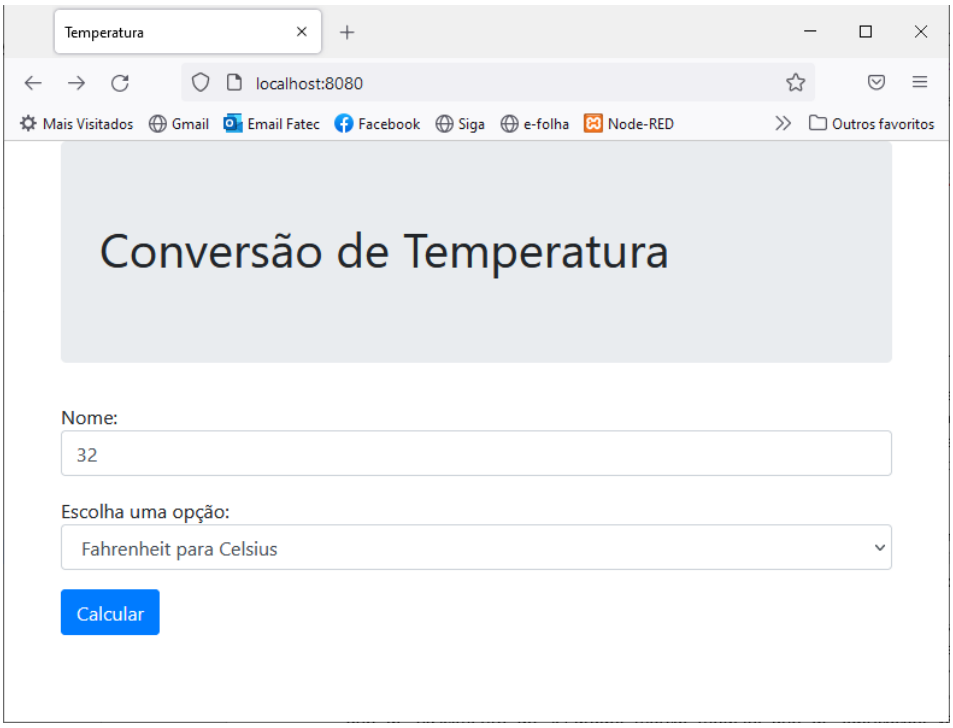

Figura 8: Caixa de Seleção

Exercícios

1) Considerando o desenvolvimento de uma aplicação TypeScript para a Internet, aplicando o conceito de arquitetura em camadas e sendo que a Lei de Ohm define que a resistência (R) de um condutor é obtida através da divisão da tensão aplicada (V) dividida pela intensidade de corrente elétrica (A). Desta forma, a partir de uma tensão e corrente, digitadas pelo usuário através de um formulário HTML, calcule e mostre o valor da resistência.

2) Desenvolver uma aplicação para a Internet, que adote arquitetura em camadas e realize uma das quatro operações aritméticas básicas (soma, subtração, multiplicação e divisão) sobre dois números reais. As operações aritméticas devem ser selecionadas a partir de botões de rádio.

3) Realizar o desenvolvimento de um servidor TypeScript que, aplicando a arquitetura em camadas, disponibilize uma página HTML que irá receber as informações através de um formulário e realizar a validação dos dados digitados, considerando que uma seguradora de veículos precisa calcular o valor da apólice com base nas seguintes informações: nome, sexo e ano de nascimento do segurado, marca, modelo, ano de fabricação, valor do veículo e porcentagem do bônus. As seguintes validações deverão ser realizadas na própria página HTML que contém o formulário:
    a) O campo sexo consiste em botões de rádio que apresentam as opções F (Feminino) ou M (Masculino).
    b) O campo ano de nascimento deve aceitar um valor entre 2001 e 1901.
    c) O campo ano de fabricação deverá ser um valor inteiro positivo.
    d) O campo valor do veículo deve ser um número real positivo.
    e) O campo porcentagem do bônus consiste em uma caixa de seleção com as seguintes opções: 0% (Sem bônus), 5%, 10% e 20%.

    Quando o formulário for submetido o servidor deverá determinar o valor da apólice, a partir dos seguintes critérios para cálculo:
    a) Para veículos 2010 ou mais recentes o valor da apólice é de 1,25% do valor do veículo, veículos entre 2009 e 2000 o valor da apólice é de 1,75% do valor do veículo, veículos entre 1999 e 1980 o valor da apólice é de 2,00% e para os demais anos de fabricação devemos utilizar 2,50% como base de cálculo.
    b) Caso o segurado tenha o sexo feminino aplicar um desconto 10% sobre o valor calculado no item a, caso contrário, acrescer 5% ao valor calculado no item a.
    c) Se o segurado possuir menos de 30 anos ou mais de 60 anos, acrescentar 20% ao valor da apólice após os cálculos realizados no item a e no item b.
    d) A partir do valor apurado nos itens a, b e c aplicar o desconto com base na porcentagem de bônus informada pelo usuário.

    Por fim, uma página HTML deverá ser exibida apresentando o valor da apólice.

JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) ou traduzindo Notação de Objetos JavaScript é uma notação leve para de troca de dados. É de fácil compreensão, leitura e escrita. Adota formato texto e é compvaramente independente de linguagem.

Uma especificação em JSON está constituída em duas estruturas. A primeira delas, é chamada de objeto e consiste em uma coleção de pares (nome e valor) o que pode caracterizar um objeto, registro, dicionário ou arrays associativos. Um item da coleção deve ser delimitado por chaves. Cada nome é seguido pelo caractere ':' (dois pontos) e, em seguida, o valor deve ser especificado. Vários pares (nome/valor) devem ser separados por vírgula, conforme ilustra o próximo exemplo.

{JSON}

```
{ codigo: 10, descricao: "Televisor", preco: 1990.00 }
```

Desta forma, analisando a estrutura, podemos entender que estamos especificando um par que apresenta o nome 'codigo' que possui o valor 10. Depois temos o par cujo nome é 'descricao' e que tem o valor "'Televisor" e o par chamado 'preco' que tem o valor 1990.00. Desta forma, usando TypeScript podemos ter o seguinte programa que permitirá manipular uma estrutura JSON como um objeto.

```
interface IProduto {
  codigo: number;
  descricao: string;
  preco: number;
}

let produto: IProduto = JSON.parse('{ "codigo": 10, "descricao":
"Televisor", "preco": 1990.00 }');
console.log('Código: ' + produto.codigo);
console.log('Descrição: ' + produto.descricao);
console.log('Preço: ' + produto.preco);
```

json/produto.ts

Sendo que também é possível declarar diretamente a estrutura JSON, sem precisar realizar o **parse**, da maneira que mostrada no programa a seguir.

```
interface IProduto {
 codigo: number;
 descricao: string;
 preco: number;
}
```

```
let produto: IProduto = { codigo: 10, descricao: "Televisor", preco:
1990.00 };
console.log('Código: ' + produto.codigo);
console.log('Descrição: ' + produto.descricao);
console.log('Preço: ' + produto.preco);
```

json/produto.ts

A segunda estrutura do JSON é chamada de array e consiste em uma lista ordenada de valores que pode caracterizar os conceitos de vetor, lista ou sequência e que basicamente pode ser entendido como um conjunto de dados que permite acesso à um único elemento através de um índice ou chave. Um array é delimitado por colchetes onde os valores são separados por vírgula, conforme podemos observar no exemplo a seguir.

{JSON}

```
[ 10, 20, 30, 40, 50 ]
```

Como neste exemplo, se trata de um conjunto de valores, em TypeScript definimos uma interface que irá conter o índice (chave) e o valor, conforme mostra o seguinte código-fonte.

```
interface ILista {
  [chave: number]: number;
}

let numero: ILista = JSON.parse('[ 10, 20, 30, 40, 50 ]');

for (let i in numero) {
  console.log(numero[i]);
}
```

json/vetor.ts

Um array pode ser uma coleção objetos, desta maneira, podemos realizar a seguinte notação:

{JSON}

```
[ { codigo: 11, descricao: "Geladeira", preco: 2990.00 }, { codigo: 12,
descricao: "Fogao", preco: 840.00 }, { codigo: 13, descricao:
"Computador", preco: 3500.00 } ]
```

Manipular objetos e arrays JSON em TypeScript é uma tarefa bastante simples. Com o intuito de ilustrar como ocorre a interação, vamos implementar um exemplo que mostrará como acessar e exibir os atributos de todos os objetos que foram armazenados.

```
interface IProduto {
  codigo: number;
```

```
  descricao: string;
  preco: number;
}

interface IListaProduto {
  [chave: number]: IProduto;
}

let produto: IListaProduto = JSON.parse('[ \
  { "codigo": 11, "descricao": "Geladeira", "preco": 2990.00 }, \
  { "codigo": 12, "descricao": "Fogao", "preco": 840.00 }, \
  { "codigo": 13, "descricao": "Computador", "preco": 3500.00 } ]');

for (let item in produto) {
  console.log('Código: ' + produto[item].codigo);
  console.log('Descrição: ' + produto[item].descricao);
  console.log('Preço: ' + produto[item].preco.toFixed(2));
  console.log('---');
}
```

json/vetor-produto.ts

Lembrando que também é possível declarar diretamente o array:

```
let produto: IListaProduto = [
  { codigo: 11, descricao: "Geladeira", preco: 2990.00 },
  { codigo: 12, descricao: "Fogao", preco: 840.00 },
  { codigo: 13, descricao: "Computador", preco: 3500.00 }
];
```

No TypeScript os arrays tem comportamento dinâmico, isto é, não tem um tamanho pré-determinado, podendo durante a execução da aplicação ter seu tamanho ampliado ou reduzido conforme a necessidade. No próximo exemplo demonstramos o uso dos métodos **push** (que permite adicionar elementos ao final de um array) e **splice** (que remove elementos).

```
interface IProduto {
  codigo: number;
  descricao: string;
  preco: number;
}

interface IListaProduto extends Array<IProduto> {
  [chave: number]: IProduto;
}

let produto: IListaProduto = [
```

```
  { codigo: 11, descricao: "Geladeira", preco: 2990.00 },
  { codigo: 12, descricao: "Fogao", preco: 840.00 },
  { codigo: 13, descricao: "Computador", preco: 3500.00 }
];

produto.push({ codigo: 14, descricao: "Mesa", preco: 400.00 });
produto.splice(2, 1);

for (let item in produto) {
  console.log('Código: ' + produto[item].codigo);
  console.log('Descrição: ' + produto[item].descricao);
  console.log('Preço: ' + produto[item].preco.toFixed(2));
  console.log('---');
}
```

json/vetor-produto.ts

Observe, no programa, que o método **push** irá inserir um elemento ao final da lista. Enquanto o método **splice** irá remover 1 elemento a partir do índice 2 do array, que neste exemplo, corresponde ao objeto que possui código igual a 13 (Computador). O método splice também pode ser usado para inserir um elemento em determinada posição do array:

```
produto.splice(1, 0, { codigo: 14, descricao: "Mesa", preco: 400.00 });
```

Neste outro exemplo o item com código 14 (Mesa) seria inserido no índice 1 do array deslocando os elementos subsequentes.

Caixas de Verificação

Diferente do funcionamento dos botões de rádio e das caixas de seleção, onde o usuário pode apenas escolher um item dentro de um conjunto de opções, a caixa de verificação permite realizar múltipla escolha. Vamos ilustrar esse conceito considerando uma pizzaria hipotética que deseja oferecer ao usuário a opção de livre escolha dos ingredientes para sua pizza.

A aplicação será desenvolvida para a Internet e aplicando a arquitetura em camadas. Desta forma, temos a página principal que implementa o formulário.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Pizzaria</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.mi
n.css">
  </head>
  <body>
    <div class="container">
      <div class="jumbotron">
        <h1>Pizzaria</h1>
      </div>
      <h3>Monte sua pizza!</h3>
      <form id="form" method="POST" action="/pizza">
        Ingredientes:
        <div class="form-check">
          <input type="checkbox" name="ingr_1" value="1" class="form-
check-input">Mussarela<br />
          <input type="checkbox" name="ingr_2" value="2" class="form-
check-input">Calabresa<br />
          <input type="checkbox" name="ingr_3" value="3" class="form-
check-input">Catupiry<br />
          <input type="checkbox" name="ingr_4" value="4" class="form-
check-input">Aliche<br />
          <input type="checkbox" name="ingr_5" value="5" class="form-
check-input">Frango<br />
          <input type="checkbox" name="ingr_6" value="6" class="form-
check-input">Milho<br />
          <input type="checkbox" name="ingr_7" value="7" class="form-
check-input">Atum<br />
          <input type="checkbox" name="ingr_8" value="8" class="form-
check-input">Tomate<br />
          <input type="checkbox" name="ingr_9" value="9" class="form-
check-input">Presunto<br /><br />
        </div>
```

```
        <input type="submit" value="Calcular" class="btn btn-primary" />
      </form>
    </div>
  </body>
</html>
```

pizzaria/pizza-form.html

Na Figura 9 temos a visualização da página criada que implementa o formulário com as caixas de verificação.

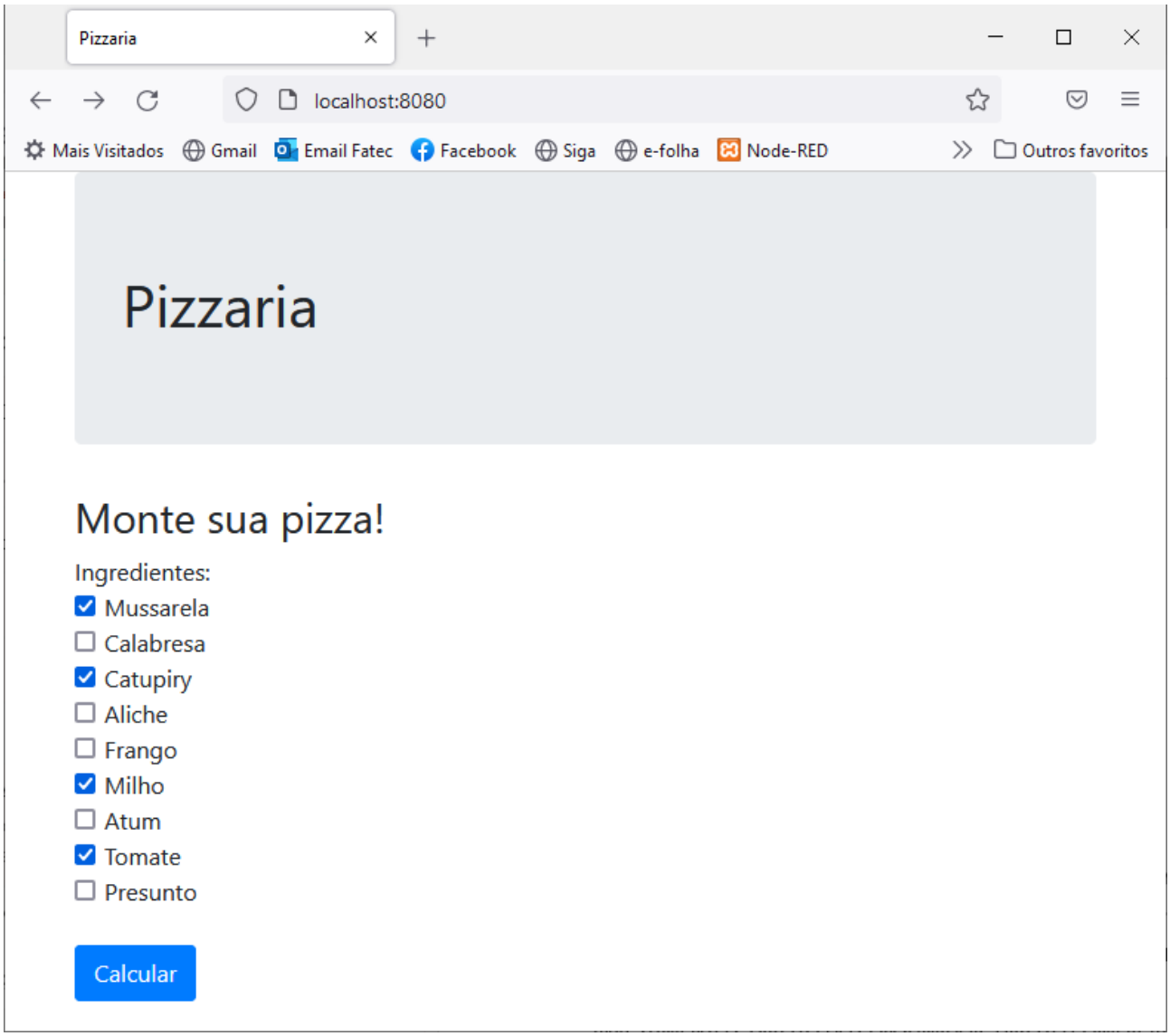

Figura 9: Caixa de Verificação

Na sequência, também será criada a página que irá exibir o resultado.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Pizzaria</title>
```

```
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.mi
n.css">
  </head>
  <body>
    <div class="container">
      <div class="jumbotron">
        <h1>Pizzaria</h1>
      </div>
      {{mensagem}}
    </div>
  </body>
</html>
```

pizzaria/pizza-res.html

Após a criação da interface com o usuário através das páginas HTML, passamos para o desenvolvimento do servidor.

```
import { Pizza } from "./pizza";

let fs = require('fs');
let http = require("http");
let express = require('express');
let app = express();
let bodyParser = require('body-parser');
let urlencodedParser = bodyParser.urlencoded({ extended: true });

let servidor = app.listen(8080, function() {
  let porta = servidor.address().port;
  console.log("Servidor executando na porta %s", porta);
});

app.get('/', function (req: any, res: any) {
  fs.readFile('pizza-form.html', function(erro: any, dado: any) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(dado);
    res.end();
  });
});

app.post('/pizza', urlencodedParser, function (req: any, res: any){
  fs.readFile('pizza-res.html', function(erro: any, dado: any) {
    let ingr: string[] = req.body;
    let pizza: Pizza = new Pizza();
    dado = dado.toString().replace("{{mensagem}}", pizza.montar(ingr));
```

```
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(dado);
    res.end();
  });
});
```

pizzaria/servidor-pizza.ts

Por último, a classe **Pizza**, com base nos dados recebidos irá retornar uma mensagem contendo os ingredientes escolhidos e o preço total. Observe o uso dos arrays que irão conter o nome e o valor de cada ingrediente, além do array que foi declarado como parâmetro do método **montar** e que contém apenas os ingredientes que foram selecionados pelo usuário no formulário.

```
export class Pizza {
  private nome: string[] = ['(Nenhum)', 'Mussarela',
    'Calabresa', 'Catupiry', 'Aliche',
    'Frango', 'Milho', 'Atum', 'Tomate',
    'Presunto'];
  private valor: number[] = [0.00, 9.00, 5.50, 6.50,
    8.00, 6.00, 4.50, 7.00, 5.00, 8.00];

  public montar(ingr: string[]): string {
    let msg: string = 'A sua pizza de ';
    let preco: number = 0.00;
    for (var chave in ingr) {
      let i: number = parseInt(ingr[chave]);
      msg += this.nome[i] + ', ';
      preco += this.valor[i];
    }
    msg += 'irá custar R$ ' + preco.toFixed(2);
    return msg;
  }
}
```

pizzaria/pizza.ts

Ao executar a aplicação, preencher e submeter o formulário, a página de resultados será exibida de maneira similar à apresentada na Figura 10.

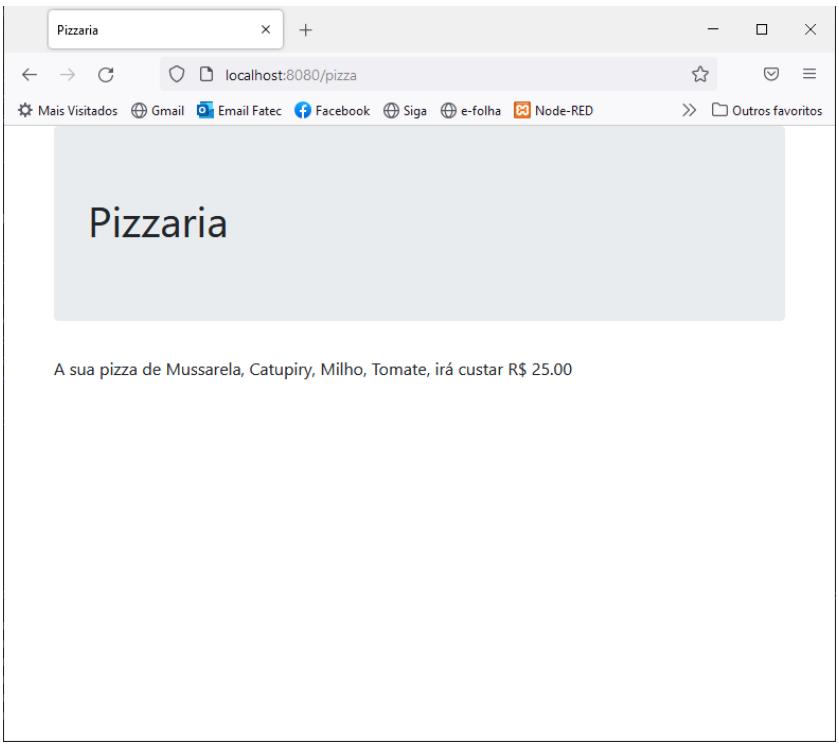

Figura 10: Resultado