Universidade Federal do Ceará - Campus de Quixadá

Programação para Design

Conceitos Básicos de OO e Encapsulamento

Prof. Anderson Lemos

Introdução

- Os conceitos da orientação objetos foram primeiramente introduzidos em 1960.
  - Foram utilizados na linguagem de programação chamada Simula.
- No entanto, essa técnica só foi aceita com o advento da Smalltalk-80 mais de uma década depois.
- Mais e mais softwares são escritos utilizando essa abordagem.
- A Programação Orientada a Objetos é um dos paradigmas da programação.

Introdução

- O que é um paradigma?
  - É um meio de se classificar as linguagens de programação baseado em suas funcionalidades.
- Paradigmas:
  - Estruturado.
  - Funcional.
  - Lógico.
  - Orientado a Objetos.

Programação Orientada a Objetos

- Conceitos de POO:
  - Artefatos: pacote, classe, objeto, membro, atributo, método, construtores e interface.
- Características da OO:
  - São conhecidas como os **quatro pilares da Programação Orientada a Objetos**:
    - Abstração
    - Encapsulamento
    - Herança
    - Polimorfismo

Abstração

- É utilizada para a definição de entidades do mundo real. Essas entidades são consideradas tudo que é real, tendo como consideração as suas características e ações (abstração de dados).
- Abstração, geralmente está relacionado a habilidade de concentrar-se nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes (abstração de procedimentos).
    - Na programação, um exemplo de abstração, é o uso de bibliotecas de uma linguagem ignorando os detalhes de implementação (ou seja, não estamos interessados em saber como a biblioteca está feita e sim no que ela faz).

Abstração

- Exemplos de abstração do mundo real:
  - Um computador montado.
  - Uma Universidade, seus cursos, alunos e funcionários.
  - Um jogo eletrônico.
  - Um sistema de biblioteca on-line.

Classes

- São especificações para objetos.
- Objetos são instâncias de uma classe.
- Quando um objeto é criado, é alocada quantidade de memória suficiente para aquele objeto.
- Objetos conversam entre si através de “troca de mensagens”.

Classes

- Exemplo: Classe Carro.
  - Quais características compreendem um carro?
    - Podemos dizer que um carro tem uma *marca*, um *ano* e uma *velocidade*, por exemplo.
    - Esses são os **atributos** da classe Carro.
  - Quais ações relacionadas a um carro?
    - Podemos dizer que um carro deve *acelerar*, *parar* e *frear*, por exemplo.
    - Esses são os **métodos** da classe Carro.
  - Quais os tipos de objetos que podemos criar a partir de uma classe carro?
    - Objetos do tipo Carro. Eles representam uma abstração de um Carro no mundo real.
  - Como isso ajuda na abstração do mundo real?
    - Imagine que você está fazendo um sistema para uma concessionária.

Classes

- Classes em TypeScript
  - Utilizamos a palavra reservada **class**.
  - Colocamos após, o nome da classe
    - Em TS, segue o padrão *CamelCase*, do JAVA
    - Inicia com letra maiúscula e segue com letras minúsculas. Se o nome for composto, cada palavra inicia com letra maiúscula.
  - O escopo de uma classe é delimitado por '**{**' e '**}**'.

```
class Carro{
    // códigos da classe aqui
}
```

Classes

- Classes em TypeScript
  - Escrevemos os atributos como variáveis comuns.
    - Não é necessário utilizar as palavras ***var*** ou ***let***.
  - Escrevemos os métodos como funções nomeadas.
    - Não é necessário utilizar a palavra ***function***.

```
class Carro{
    marca : string;
    ano : number;
    velocidade : number;
    acelerar() : void { /* implementação de acelerar */ }
    parar() : void { /* implementação de parar */ }
    frear() : void { /* implementação de frear */ }
}
```

Classes

- Classes em TypeScript
  - Os atributos de uma classe também são chamados de “variáveis de instância”.
  - Podemos acessar e modificar os atributos dentro dos métodos.
  - Para isso, usamos a palavra reservada **this** (do inglês, este).
    - Dentro dos métodos, utilizamos *this.nomeDoAtributo*
    - Semanticamente, significa “pegue **este** objeto e acesse o atributo *nomeDoAtributo*”.

Classes

- Classes em TypeScript

```
class Carro{
    marca : string;
    ano : number;
    velocidade : number;
    acelerar() : void {
        // acelerar aumenta a velocidade em 10 km/h
        this.velocidade += 10;
    }
    parar() : void {
        this.velocidade = 0;
    }
    frear() : void {
        // frear diminui a velocidade em 10 km/h
        this.velocidade -= 10;
    }
}
```

Objetos

- Classes somente são especificações.
- Elas dizem o que um objeto tem e faz.
- Um objeto é uma instância de uma classe.
  - Isto é, uma entidade que possui as características definidas em uma classe.
    - Cada objeto vai ter dentro dele todos os métodos e atributos que estavam definidos em uma classe.
  - Por exemplo, fizemos a classe **Carro**, que especifica o que um carro tem e faz.
  - Instâncias desta classe, seriam os carros propriamente ditos, cada um com seus atributos e métodos independentes um do outro.
    - Por exemplo, o seu carro é uma instância, o carro do vizinho é outra instância, o carro do Felipe Massa é outra instância. Cada instância é um objeto.

Objetos

- Assim, em TypeScript, uma classe pode ser considerada como um novo tipo.
- Uma instância de uma classe (objeto) nada mais é que uma variável do tipo daquela classe.
  - Por exemplo, um objeto carro é uma variável do tipo Carro.
- Tratamos um objeto como uma variável qualquer.
  - Com a diferença de que precisamos instância-lo.
  - Para instanciar um novo objeto, utilizamos a palavra reservada **new** (novo) e a frente, o nome da classe seguido de parênteses.
- Para acessar os métodos ou atributos de um objeto, utilizamos a notação ***nomeDoObjeto.nomeDoAtributoOuMetodo***.

Objetos

```
let c : Carro = new Carro();
c.velocidade = 0;
c.marca = "Fiat";
c.ano = 2018;
console.log(c); // imprime os valores dos atributos do objeto c
                // imprime Carro { velocidade: 0, marca: 'Fiat', ano: 2018 }
c.acelerar(); // aumenta a velocidade de c em 10 km /h
c.acelerar(); // aumenta a velocidade de c em 10 km /h
c.acelerar(); // aumenta a velocidade de c em 10 km /h
console.log(c.velocidade); // como acelerou 3 vezes, imprime 30
c.frear();
console.log(c.velocidade); // como freou, imprime 20
c.parar();
console.log(c.velocidade); // como parou, imprime 0
```

Classes e Objetos

- Note que o que fizemos foi definir uma classe, dizendo o que um objeto daquela classe tem e faz.
- Assim, atributos e métodos são individuais para cada objeto daquela classe.
  - Por exemplo, usando a classe **Carro**, o seu carro tem um atributo ***marca*** e o carro do seu vizinho tem um atributo ***marca*** que pode ser completamente diferente, não está relacionado a marca do seu carro.
  - O método ***acelerar*** do seu carro não está relacionado com o método ***acelerar*** do carro do seu vizinho, isto é, chamar o método ***acelerar*** do carro do seu vizinho não vai mudar a velocidade do seu carro (o que faz todo sentido).

Classes e Objetos

```
let meuCarro : Carro = new Carro();
meuCarro.marca = "Fiat";
meuCarro.velocidade = 0;
let carroDoVizinho : Carro = new Carro();
carroDoVizinho.marca = "Ford";
carroDoVizinho.velocidade = 0;
carroDoVizinho.acelerar(); // não afeta velocidade de meuCarro
console.log("Meu carro: " + meuCarro.marca +
                    ", " + meuCarro.velocidade);
// imprime "Meu carro: Fiat, 0"
console.log("Carro do vizinho: " + carroDoVizinho.marca +
                    ", " + carroDoVizinho.velocidade);
// imprime "Carro do vizinho: Ford, 10"
// podemos ver que são marcas e velocidades diferentes
```

Classes e Objetos

- No entanto, às vezes faz sentido que um atributo ou um método esteja relacionado a uma classe mas não seja algo específico de cada objeto.
  - Por exemplo, imagine que queremos armazenar em uma variável o total de carros do sistema. Essa variável está relacionada a classe **Carro**, mas não é algo específico de cada carro.
  - Outro exemplo, imagine que você quer um método que receba uma lista de carros e conte quantos deles são do ano 2012. Esse método está relacionado a classe **Carro**, mas não é algo específico de cada carro (não faz sentido um carro executar a ação de receber outros carros e contar quais são de um determinado ano).

Classes e Objetos - Operador *static*

- Quando atributos ou métodos estão relacionados a uma classe mas não a objetos específicos, os inserimos na classe e utilizamos o operador ***static*** (do inglês, **estático**).
  - Se diz que é estático, ou seja, não é dinâmico, não muda de acordo com o objeto.
- A grande diferença de um atributo ou método estático, é que não precisamos de um objeto para acessá-los.
  - Para acessar os métodos ou atributos estáticos, utilizamos a notação ***nomeDaClasse.nomeDoAtributoOuMetodo***

Classes e Objetos - Operador *static*

```
class Carro{
    marca : string;
    ano : number;
    velocidade : number;
    static numeroDeCarros = 0;
    static contarCarrosDeUmAno(vetor : Array<Carro>,
        ano : number) : number{
        let cont : number = 0;
        for(let c of vetor){
            if(c.ano == ano) cont++;
        }
        return cont;
    }
    static imprimeNumeroDeCarros() : void{
        console.log(Carro.numeroDeCarros);
    }
}

Carro.numeroDeCarros = 10;
Carro.imprimeNumeroDeCarros();
// imprime 10
```

Classes e Objetos - Operador *static*

- Chamamos atributos e métodos **não estáticos** (ou dinâmicos) de atributos de instância e métodos de instância.
  - Lembrando que uma instância de uma classe nada mais é que um objeto.
- Chamamos atributos e métodos **estáticos** de atributos da classe e métodos da classe.

Construtores

- Note que criamos o objeto e depois inicializamos os seus atributos.
  - Mas há uma forma de inicializar os atributos no momento de criação do objeto?
    - Sim! Através do construtor.
    - Utilizamos a palavra reservada **constructor**.
- O construtor é um método criado dentro da classe que é chamado apenas uma vez, exatamente no momento que o objeto é criado.
- Nele, podemos inicializar os atributos do objeto e fazer quaisquer outras instruções que desejemos.
- Repare que ao criar um objeto do tipo **Carro** fazemos *new Carro()*
  - Estes parênteses representam exatamente os parâmetros que passamos para o construtor.

Construtores

```
class Carro{
    marca : string;
    ano : number;
    velocidade : number;
    constructor(){
        this.velocidade = 0;
        this.marca = "Fiat";
        this.ano = 2018.
    }
    acelerar() : void {
        this.velocidade += 10;
    }
    parar() : void {
        this.velocidade = 0;
    }
    frear() : void {
        this.velocidade -= 10;
    }
}
```

```
let c : Carro = new Carro();
console.log(c);
// imprime :
// Carro { velocidade: 0, marca: 'Fiat', ano: 2018 }

/*  o this dentro da classe é uma forma de
    acessar o objeto dentro da classe. É como
    se disséssemos "este objeto"
*/
```

Construtores

- Ok, mas sempre vamos inicializar os atributos de todos os objetos com os mesmos valores? E se quisermos dizer com o que queremos inicializar?
  - Podemos passar parâmetros para o construtor.
  - E quando chamamos o **new**, passamos dentro dos parênteses tais parâmetros.
  - Por padrão, passamos parâmetros para inicializar todos os atributos, e com os mesmos nomes dos atributos.
    - E como diferenciar?
      - Através do **this**. Se temos um atributo chamado *atr*, passamos por parâmetro uma variável com o mesmo nome, *atr*, e fazemos *this.atr* = *atr*. O TypeScript entende que *this.atr* é o atributo *atr* dentro do objeto *this*, e *atr* é a variável que foi passada no parâmetro do construtor.

Construtores

```
class Carro{
    marca : string;
    ano : number;
    velocidade : number;
    constructor(marca : string, ano : number,
                velocidade : number){
        this.velocidade = velocidade;
        this.marca = marca;
        this.ano = ano;
    }
    acelerar() : void {
        this.velocidade += 10;
    }
    parar() : void {
        this.velocidade = 0;
    }
    frear() : void {
        this.velocidade -= 10;
    }
}
```

```
let c1 : Carro = new Carro("Fiat", 2018, 0);
console.log(c1);
// imprime :
// Carro { velocidade: 0, marca: 'Fiat', ano: 2018 }

let c2 : Carro = new Carro("Ford", 2012, 20);
console.log(c2);
// imprime :
// Carro { velocidade: 20, marca: 'Ford', ano: 2012 }

/*  Repare, que o TypeScript entende que this.marca é
    a variável dentro do objeto e marca é a variável
    passada como parâmetro para o construtor (isso
    serve para todos os atributos). Efetivamente,
    this.marca e marca são variáveis diferentes (até na
    forma de escrever).

    Em let c1 = new Carro("Fiat", 2018, 0), passamos
    os parâmetros na ordem que estão no construtor,
    isso é importante, como em qualquer função.
*/
```

Modularização

- Uma das preocupações da OO é a organização de código.
- Assim, é interessante dividir o programa em módulos.
- Já fazemos isso parcialmente, dividindo em funções e classes.
- Mas será que há um padrão na OO que ajude a modularizar o código?

Modularização

- Pacotes
  - Um **pacote** ou ***package***, em TypeScript, é um conjunto de classes localizadas na mesma estrutura hierárquica de diretórios.
    - Ou seja, separamos as classes por pastas, e deixamos na mesma pasta classes que tem um papel semelhante.
  - Geralmente criamos os pacotes como pastas umas dentro das outras.
    - Primeiramente criamos a pasta do nosso projeto, geralmente com o nome do projeto. Chamaremos de **proj**. Essa é a pasta raiz do nosso projeto.
    - As classes dentro desta pasta estão no pacote **default**, ou padrão.
    - Os outros pacotes são criados dentro da pasta raiz.
    - Por exemplo se criamos a pasta **p1** e dentro dela a pasta **p2**, as classes que estiverem dentro da pasta **p2** farão parte do pacote **p1.p2** (acessamos pacotes dentro de outros pelo ponto).

Modularização

- Nomeando pacotes
  - O padrão que utilizamos para nomear os pacotes, é tentar fazer com que nossa aplicação tenha nomes de pacotes únicos.
  - Geralmente usamos o padrão de localização e de onde o projeto está sendo desenvolvido.
    - Seguindo o padrão: país.com.empresa.espacificacaoDoPacote
  - Por exemplo, para as classes que criamos até agora:
    - `br.com.ufc.dd.poo.exec` , para a classe **Principal**.
    - `br.com.ufc.dd.poo.model` , para a classe **Carro**.

Modularização

- Em OO, por padrão, criamos cada classe em um arquivo separado.
  - Para que essa classe possa ser usada em outros arquivos, utilizamos a palavra reservada ***export***.
  - O nome do arquivo, por padrão, é o mesmo da classe.
  - Para utilizar uma classe de outro arquivo, ela deve ter a palavra ***export*** antes de sua definição e é necessário importá-la no arquivo que queremos utilizá-la.
    - Utilizamos então os comandos ***import*** e ***from***, como no comando abaixo:
      - *import {NomeDaClasse} from "caminhoDoAquivoDaClasse";*
      - Se os arquivos estiverem no mesmo pacote, utilizamos *"./nomeDoArquivo"*.
      - Caso contrário, utilizamos o caminho relativo do arquivo que está a classe.
  - Geralmente há um arquivo principal que não tem nenhuma classe mas que inicia o programa. Basicamente ele é o arquivo que é executado e usa as classes para se tornar uma aplicação.

Modularização

```
/*  Arquivo Carro.ts, pertencente ao pacote:
    br.com.ufc.dd.poo.model */

export class Carro{
    marca : string;
    ano : number;
    velocidade : number;
    constructor(marca : string, ano : number,
                    velocidade : number){
        this.velocidade = velocidade;
        this.marca = marca;
        this.ano = ano;
    }
    acelerar() : void {
        this.velocidade += 10;
    }
    parar() : void{
        this.velocidade = 0;
    }
    frear() : void {
        this.velocidade -= 10;
    }
}
```

```
/*  Arquivo Principal.ts, pertencente ao pacote:
    br.com.ufc.dd.poo.exec */

// importamos a partir do caminho relativo, como abaixo
import {Carro} from '../model/Carro';

let c1 : Carro = new Carro("Fiat", 2018, 0);
console.log(c1);
// Carro { velocidade: 0, marca: 'Fiat', ano: 2018 }

let c2 : Carro = new Carro("Ford", 2012, 20);
console.log(c2);
// Carro { velocidade: 20, marca: 'Ford', ano: 2012 }
```

Exercícios

- Crie uma classe que represente um aluno, com os atributos nome, idade, curso e IRA.
- Crie os seguintes métodos em Aluno:
  - estudar(materia)
  - matricular(disciplina)
- Teste sua classe Aluno usando um arquivo separado.

Encapsulamento

Encapsulamento

- Um objeto deve manter seus dados de instância seguros contra o acesso externo a sua classe.
- Proteger os dados, não oferecendo acesso direto é uma característica do encapsulamento.
- Em TypeScript, o encapsulamento é implementado através dos modificadores de acesso:
  - public
  - private
  - protected
  - readonly

Encapsulamento

- Todo atributo ou método tem um modificador de acesso.
  - Se não os definimos explicitamente, o TypeScript entende que o modificador escolhido é *public*.
    - Ou seja, *public* é o modificador de acesso padrão.
  - Por padrão e legibilidade, mesmo que um atributo ou método deva ser *public*, diremos isso explicitamente.

Encapsulamento

- Exercício:
  - Modifique a classe Aluno (do exercício anterior) tornando seus atributos públicos (colocando explicitamente o modificador *public*).
  - Instancie um objeto Aluno.
  - Modifique os atributos através do objeto, usando o operador de atribuição.
  - Mude o modificador de acesso dos atributos para *private*. Tente acessar os atributos via objeto novamente. O que acontece?

Encapsulamento

- Modificador de acesso público
  - public
  - Se um atributo é público, é permitido que qualquer classe acesse e modifique o valor desse atributo.
  - Se um método é público, é permitido que qualquer classe chame esse método.

Encapsulamento

- Modificador de acesso privado
  - private
  - Se um atributo é privado, só é permitido que a classe dona do atributo acesse e modifique o valor desse atributo. Nenhuma outra classe poderá acessar ou muito menos modificar o valor do atributo.
  - Se um método é privado, só é permitido que a classe dona do método chame esse método. Nenhuma outra classe poderá chamar o método.

Encapsulamento

- Modificador de acesso protegido
  - protected
  - Similar ao modificador privado, mas um pouco menos restrito.
  - Se um atributo é protegido, só é permitido que a classe dona do atributo ou **classes derivadas** da classe dona do atributo acessem e modifiquem o valor desse atributo. Nenhuma outra classe poderá acessar ou muito menos modificar o valor do atributo.
  - Se um método é protegido, só é permitido que a classe dona do método ou **classes derivadas** da classe dona do método chamem esse método. Nenhuma outra classe poderá chamar o método.
  - Fará sentido quando estivermos falando de **herança**.

Encapsulamento

- Modificador de acesso somente leitura
  - readonly
  - Se um atributo é readonly, só é permitido que a classe dona do atributo o modifique. Nenhuma outra classe poderá modificar o valor do atributo. No entanto, qualquer classe pode acessar o valor do atributo.
  - O modificador readonly só faz sentido para atributos, não faz sentido para métodos.

Encapsulamento

- Padrões de encapsulamento
  - Por padrão, os métodos das classes são na maioria das vezes públicos (inclusive o construtor).
  - No entanto, toda classe deve ter seus atributos de instância **privados**!
    - A não ser que a classe tenha classes derivadas. Nesse caso, às vezes, faz sentido ter atributos protegidos (mais uma vez, fará sentido quando falarmos de herança).
  - Cabe ao desenvolvedor implementar métodos chamados “acessores” para os atributos de instância.
  - Em TypeScript, esses métodos acessores são padronizados como **get** e **set**.

Encapsulamento

- Métodos GET e SET
  - São métodos que tem como função retornar (GET) ou modificar (SET) as variáveis de instância.
  - A padronização dos métodos é a seguinte:
    - public get<NomeDoAtributo>() : <TipoDoAtributo> {return this.<NomeDoAtributo>;}
    - public set<NomeDoAtributo>(<nomeDoAtributo> : <TipoDoAtributo>) : void { this.<nomeDoAtributo> = <nomeDoAtributo>;}
  - Esse é o padrão, mas às vezes é interessante colocar restrições para que as outras classes não tenham total acesso aos atributos.
    - Para isso que serve o encapsulamento.

Encapsulamento

- Métodos GET e SET: exemplo da classe carro.

```
// ARQUIVO Carro.ts

export class Carro{
    private marca : string;
    private ano : number;
    private velocidade : number;
    public constructor(marca : string, ano : number,
                velocidade : number){
        this.velocidade = velocidade;
        this.marca = marca;
        this.ano = ano;
    }
    public acelerar() : void {
        this.velocidade += 10;
    }
    public parar() : void {
        this.velocidade = 0;
    }
    public frear() : void {
        this.velocidade -= 10;
    }
```

```
    // continuação da classe Carro
    public getVelocidade() : number {
        return this.velocidade;
    }
    public setVelocidade(velocidade : number) : void {
        this.velocidade = velocidade;
    }
    public getMarca() : string {
        return this.marca;
    }
    public setMarca(marca : string) : void {
        this.marca = marca;
    }
    public getAno() : number {
        return this.ano;
    }
    public setAno(ano : number) : void {
        this.ano = ano;
    }
}
```

Método toString

- O toString tem como função representar um objeto em forma de string, da maneira que o desenvolvedor achar mais apropriado.
- Qualquer classe TypeScript já tem, por padrão, esse método.
  - No entanto ele diz apenas que um objeto é um objeto.
  - Se imprimirmos o objeto em si, ele mostrará os atributos em um formato padrão chamado JSON (JavaScript Object Notation).
- Ex:

```
let c : Carro = new Carro("Ford", 2002, 0);
console.log(c);
// Imprime Carro { velocidade: 0, marca: 'Ford', ano: 2002 }
// É um formato padrão chamado JSON

console.log(c.toString());
// Imprime [object Object]
```

Método toString

- No entanto podemos sobrescrever (escrever por cima) esse método. Ou seja, podemos escrever o método novamente na classe Carro para retornar uma string no formato que queiramos. Por exemplo, se queremos imprimir cada atributo em uma linha:

```
// inserimos o método dentro da classe Carro
public toString() : string {
    let toStr : string = "";
    toStr += "Marca: " + this.marca + "\n";
    toStr += "Ano: " + this.ano + "\n";
    toStr += "Velocidade: " + this.velocidade + "\n";
    return toStr;
}
```

```
let c : Carro = new Carro("Ford", 2002, 0);
console.log(c);
// Imprime Carro { velocidade: 0, marca: 'Ford', ano: 2002 }
// É um formato padrão chamado JSON

console.log(c.toString());
/* Imprime:
Marca: Ford
Ano: 2002
Velocidade: 0
*/
```

Método toString

- A grande vantagem, é caso queiramos imprimir o objeto concatenado com outra coisa.
  - Ele não imprimirá no formato JSON, ele imprimirá o toString.
  - Se você não sobrescrevê-lo, imprimirá algo concatenado com uma string que apenas diz que é um objeto.
  - Se você sobrescrever, ele vai concatenar essa coisa com a string que o toString retornar.

Exercício

- Na classe Aluno:
  - Para todos as variáveis de instância, crie os métodos **get** e **set**.
  - Quais são as vantagens dessa abordagem?
  - Modifique o set de idade para que o mesmo não permita entrada de valores inválidos.
  - Sobrescreva o método toString.

Perguntas?