Instituto Federal de Ciência, Educação e Tecnologia do Piauí – IFPI

Aluno: Vinícius Gomes Araújo Costa

Disciplina: Programação Orientada a Objetos (POO)

Professor: Ely Miranda

Curso: Análise e Desenvolvimento de Sistemas (ADS) – Módulo II – Tarde

**Semestre: 2023.2**

**Exercício 04**

1. (**V**) Os objetos são modelos para classes;

**Justificativa**: Classes são modelos, objetos são classes em execução/memória (instanciadas); com isso a justificativa de que “os objetos são modelos para classes” baseia-se nos princípios e conceitos fundamentais da Programação Orientada a Objetos(POO). A Programação Orientada a Objetos é um paradigma de programação que organiza o código em torno de objetos, que são instâncias de classes. Como afirmado por um motivo de não redundância de explicações, irei citar as informações em comum que fazem parte desta justificativa: Abstração, Encapsulamento, Herança, Polimorfismo, Reusabilidade e Modularidade são pontos que explicam essa afirmação.

Em síntese, a justificativa de que “objetos são modelos para classes” é uma afirmação central da POO, pois reflete a ideia de que as classes são as estruturas que definem as características e os comportamentos dos objetos. Isso permite a criação de um código mais organizado, modular, reutilizável e adaptável, promovendo uma abordagem eficaz para a construção de software complexo.

(**F**) Os atributos de uma classe devem ser obrigatoriamente inicializados para que as classes compilem;

**Justificativa**: Não necessariamente. Em muitas linguagens de Programação Orientada a Objetos, os atributos de uma classe não precisam ser obrigatoriamente inicializados para que a classe compile. No entanto, isso pode depender do contexto e das regras da linguagem específica que você está utilizando.

Algumas linguagens permitem definir conceitos padrão para os atributos de uma classe, o que significa que se um atributo não for explicitamente inicializado no constructor ou em outro lugar, ele assumirá um valor padrão automaticamente. Isso pode ser útil para evitar erros durante a compilação ou execução.

Outras linguagens podem exigir podem exigir que os atributos sejam inicializados explicitamente no constructor da classe ou em algum ponto antes que sejam acessado. Caso contrário, o compilador pode gerar um erro ou aviso, indicando que a variável pode estar sendo usada antes de ser inicializada.

Em síntese, enquanto a maioria das linguagens permite que os atributos de uma classe não sejam obrigatoriamente inicializados (graças a valores padrão ou inicializações explícitas), a maneira exata como isso é tratado varia de linguagem para linguagem. É importante consultar a documentação da linguagem que você está usando para entender as regras específicas relacionadas à inicialização de atributos em classes.

(**F**) Uma variável declarada dentro de um método deve ser inicializada para que a classe seja compilável;

**Justificativa**: Em muitas linguagens de programação, variáveis locais (ou seja, variáveis declaradas dentro de um método ou função) não precisam ser obrigatoriamente inicializadas no momento da declaração para que a classe seja compilável. No entanto, essas variáveis precisam ser inicializadas antes de serem usadas, caso contrário, erros de compilação podem ocorrer.

A regra geral é que as variáveis locais não têm valores definidos até que sejam atribuídos explicitamente. Se você tentar usar uma variável local antes de atribuir um valor a ela, o compilador pode gerar um erro ou um aviso.

***Vamos considerar um exemplo em TypeScript que demonstra isso:***

class Exemplo {

metodoExemplo() : void {

let x : number; // Variável declarada, mas não inicializada

console.log(x); // Tentativa de usar x sem inicialização

}

}

const exemplo = new Exemplo();

exemplo.metodoExemplo();

Este código será compilado sem erros em TypeScript. No entanto, quando o método `metodoExemplo()` for chamado, você receberá um erro em tempo de execução indicando que a variável `x` está indefinida. Isso corre porque você está tentando usar uma variável que não foi atribuída a nenhum valor.

Embora o código compile, é uma boa prática sempre inicializar suas variáveis locais antes de utilizá-las, para evitar erros de runtime. Se você não tem certeza qual valor atribuir, considere usar um valor padrão ou reavaliar a lógica de seu programa para garantir que as variáveis sejam usadas corretamente.

Em resumo, em TypeScript, variáveis locais não precisam ser inicializadas no momento da declaração para que a classe seja compilável, mas a inicialização adequada é importante para evitar erros em tempo de execução.

(**V**) Uma variável que seja uma classe declarada em um método é automaticamente inicializada com undefined;

**Justificativa**: Sim, em TypeScript, quando você declara uma variável que seja uma instância de uma classe dentro de um método (ou função), ela é automaticamente inicializada com o valor `undefined`. Isso é uma característica comum em linguagens que possuem tipagem estática, como TypeScript.

***Ilustração do exemplo:***

class ExemploClasse {

nome : string;

constructor (nome : string) {

this.nome = nome;

}

mostrarNome() : void {

console.log(this.nome);

}

}

Class Exemplo {

metodoExemplo() : void {

let instancia : ExemploClasse; // Variável declarada, automaticamente inicializada com undefined

instancia.mostrarNome(); // Tentativa de uso da instância

}

}

const exemplo = new Exemplo();

exemplo.metodoExemplo();

Neste exemplo, ao declarar a variável `instancia` dentro do método `metodoExemplo()` da classe `Exemplo`, ela é inicializada automaticamente com `undefined`. Quando você tenta chamar o método `mostrarNome()` da instância, isso resultará em um erro de tempo de execução, pois `instancia` não foi atribuída a uma instância válida da classe `ExemploClasse`.

Portanto, mesmo que a variável seja inicializada automaticamente com `undefined`, você ainda precisa garantir que ela seja atribuída a uma instância válida da classe antes de usar seus métodos ou propriedades.

(**V**) Construtores são rotinas especiais que servem para inicializar e configurar os objetos no momento da instanciação;

**Justificativa**: Sim, isso é verdade. Em Programação Orientada a Objetos, os constructores são rotinas especais que servem para inicializar e configurar objetos no momento da instanciação de uma classe. Um constructor é um método especial que é chamado automaticamente quando um objeto é criado a partir de uma classe.

Os constructores desempenham um papel crucial no processo de criação de objetos, pois permitem que você defina os valores iniciais dos atributos da classe e realize qualquer configuração necessária para que o objeto possa ser usado corretamente. Isso ajuda a garantir que um objeto seja criado em um estado válido e consistente.

***Aqui está um exemplo simples em TypeScript para ilustrar como os constructores funcionam:***

class Pessoa {

nome : string;

idade : number;

constructor (nome : string, idade : number) {

this.nome = nome;

this.idade = idade;

}

saudação() : void {

console.log(`Olá, meu nome é ${this.nome} e eu tenho ${this.idade} anos.`);

}

}

const pessoa1 = new Pessoa(“João”, 25);

pessoa1.saudacao(); // Saída: Olá, meu nome é João e eu tenho 25 anos.

Neste exemplo, a classe `Pessoa` possui um constructor que recebe parâmetros `nome` e `idade`, que são usados para inicializar as propriedades `nome` e `idade` do objeto quando ele é criado. O método `saudação()` pode então usar essas propriedades para gerar uma saudação personalizada.

Em resumo, os constructores são elementos fundamentais na Programação Orientada a Objetos, pois permitem que objetos sejam criados com valores iniciais específicos e configurados corretamente para serem utilizados em sua funcionalidade específica.

(**V**) Constructores não possuem tipo de retorno e podem ou não ter parâmetros;

**Justificativa**: Sim, isso é verdade. Constructores em Programação Orientada a Objetos têm algumas características específicas:

1. **Tipo de retorno**

Constructores não têm um tipo de retorno explícito. Isso significa que eles não retornam um valor diretamente, ao contrário de métodos comuns que podem ter um tipo de retorno especificado.

1. **Parâmetros Opcionais**

Constructores podem te parâmetros, mas eles não são obrigatórios. Você pode definir um constructor com nenhum, um ou vários parâmetros, dependendo das necessidades da classe que está sendo construída. Parâmetros em constructores são usados para passar informações necessárias para inicializar o objeto corretamente.

***Aqui está um exemplo simples em TypeScript:***

class Exemplo {

atributo : string;

constructor (valor : string) {

this.atributo = valor;

}

}

const instancia1 = new Exemplo(“Primeira instância”);

const instancia2 = new Exemplo(“Segunda instância”);

console.log(instancia1.atributo); // Saída: Primeira instância

console.log(instancia2.atributo); // Saída: Segunda instância

Nesse exemplo, a classe `Exemplo` tem um constructor que recebe um parâmetro `valor` e o atribui ao atributo `atributo` da instância. Note que os constructores não têm um tipo de retorno declarado.

Lembrando que, embora os parâmetros não sejam obrigatórios, é importante garantir que a inicialização dos objetos seja feita de maneira consistente e válida. Isso pode envolver a definição de valores padrão ou validações para os parâmetros do constructor.

(**V**) Uma classe pode ter várias instâncias.

**Justificativa**: Sim, em TypeScript (e na maioria das Linguagens de Programação Orientada a Objetos), uma classe pode ser usada como um modelo para criar várias instâncias diferentes. Cada instância é um objeto separado e independente que compartilha a estrutura definida pela classe, mas pode ter valores de propriedades diferentes.

***Vou considerar um exemplo em TypeScript:***

class Pessoa {

nome : string;

idade : number;

constructor (nome : string, idade : number) {

this.nome = nome;

this.idade = idade;

}

saudação() : void {

console.log(`Olá, meu nome é ${this.nome} e eu tenho ${this.idade} anos.`);

}

}

const pessoa1 = new Pessoa (“João”, 25);

const pessoa2 = new Pessoa(“Maria”, 30);

pessoa1.saudacao(); // Saída: Olá, meu nome é João e eu tenho 25 anos.

pessoa2.saudacao(); // Saída: Olá, meu nome é Maria e eu tenho 30 anos.

Neste exemplo, a classe `Pessoa` é usada como um modelo para criar duas instâncias diferentes: `pessoa1` e `pessoa2`. Cada instância têm seus próprios valores para as propriedades `nome` e `idade`, permitindo que você crie e trabalhe com várias instâncias da mesma classe.

Em resumo, uma classe é uma definição que descreve a estrutura e o comportamento dos objetos. As instâncias são objetos individuais criados a partir dessa definição e podem ter propriedades e valores diferentes, permitindo que você modele e manipule dados de maneira mais organizada e eficaz.

**2.** Na questão da classe Hotel, a variável `quantReservas` a qual serve para guardar as solicitações de reservas feitas pelo Hotel encontra-se com os seguintes erros:

O 1º erro seria que: a propriedade `quantReservas` não tem inicializador e não está definitivamente atribuída ao constructor;

O 2º erro seria que: no incremento `quantReservas++` não possui seu instanciador de atribuição em um constructor, isto é, além de não ser possível encontrar o nome `quantReservas`, o correto seria atribuí-lo à referência `this.quantReservas++` em um membro da instância `quantReservas` para o seu constructor;

O 3º erro foi que a função adicionarReserva() : void não segue nenhum parâmetro para ser instanciado na classe Hotel, deixando seu conteúdo vazio e sem argumentos, com isso, o incremento fica sem sentido porquê não envolverá nenhum laço de iteração ou repetição while e for para determinada situação.

O 4º erro, por se tratar da linguagem TypeScript, um **super set “tipado”**, não haverá compilação, pois, a variável inteira não foi inicializada previamente, nem citada nem colocada seus atributos de forma construtiva, organizada e eficaz. Estando equivocada a quantidade de reservas. Questão demonstrada no VS Code juntamente com a 3ª questão.

***A possibilidade seria, em TypeScript:***

class Hotel {

idApartamento : number = 543;

quantReservas : number = 51;

constructor (idApartamento : number, quantReservas : number) {

this.idApartamento = idApartamento;

this.quantReservas = quantReservas;

}

}

this.quantReservas++;

1. Adicione o constructor que aceite um parâmetro inteiro e faça a inicialização do atributo quantReservas. O Código demonstrado corretamente encontra-se no VS Code.

**Justificativa**:

class Hotel {

idApartamento : number = 543;

quantReservas : number = 51;

constructor (idApartamento : number, quantReservas : number) {

this.idApartamento = idApartamento;

this.quantReservas = quantReservas;

}

}

this.quantReservas++;

let hotel : Hotel = new Hotel(2);

console.log(hotel.quantReservas);

O erro dessa questão seria que: a meu modo, configurei a questão adicionando dois atributos a classe Hotel: `idApartamento` e `quantReservas`, como criei uma nova classe hotel, isto é, um new Hotel com parâmetro do tipo number 2, o programa do Visual Studio Code acusará erro de chamada de função para 2 argumentos, porém chamei somente 1, no caso daria undefined. Uma expectativa de 2 argumentos para 1 só. Ocasionando erro de compilação.

1. Justifique o erro de compilação e proponha uma solução, considerando minha classe Radio.

Justificativa: Na minha classe Radio, eu inicializo com o atributo volume e tipo ele para o tipo number, porém na questão não há um valor dado para volume. Ele somente está tipado, sem valor algum. Coloco assim um valor do tipo 12. Crio um constructor com seu respectivo método de instanciação this.volume = volume de acordo com o Paradigma Orientado a Objetos, e por fim, a criação da nova classe let r : Radio = new Radio() não chama nenhum argumento para a nova classe criada.

Com isso, faltou a mensagem de exibição na tela console.log(r.volume) para o valor que deveria ser dado, como dito acima no parágrafo anterior de valor 12, assim ficaria minha nova classe: let r : Radio = new Radio(12); e posteriormente console.log(r.volume) seria printada/exibida corretamente como solução apropriada para a questão da classe Radio. Obs:. Demonstração feita também no Visual Studio Code.

1. Considerando o uso da classe Conta apresentada em aula e seu uso abaixo:
2. Qual o resultado dos dois “prints”? Justifique sua resposta.

**R)** O resultado dos dois “prints” será: `console.log(c1.consutarSaldo())` resulta em 40, porque `c1` agora aponta para a mesma instância de conta que `c2`, e ambas tiveram saques e transferências feitas.

`console.log(c2.consultarSaldo())` resulta em 150, porque `c2` não teve saque após a transferência, e ele recebeu uma transferência de 50 da conta `c1`. Código demonstrado no VS Code.

1. O que acontece com o objeto para o qual a referência c1 apontava?

**R)** O objeto para o qual a referência `c1` apontava originalmente foi perdido. No código disponível pelo VS Code, o programador vê que primeiro `c1` apontava para uma instância de conta com saldo 100. No entanto, após as atribuições `c1 = c2` e `c3 = c1` passou a apontar para a mesma instância que `c2`, e a instância original alocada na primeira linha(`let c1 : Conta = new Conta(“1”, 100)`) não tendo mais referências apontando para ela e, portanto, se torna elegível para coleta de lixo (será destruída). Portanto, a referência original a `c1` não tem mais efeito sobre a conta original, que agora é acessível apenas através de `c3` e `c2`. Código demonstrado no VS Code.

1. Nesta classe, temos os atributos `texto` e `destinatario`, que são inicializados no constructor. O método `obterSaudacao()` concatena esses atributos para criar a saudação desejada. Em seguida, instanciamos a classe `Saudacao` com os valores “Bom Dia” e “João” e testamos o método `obterSaudacao()`, que produz a saudação “Bom Dia, João!” como resultado. Código demonstrado no VS Code.
2. Nesta classe `Triangulo`, os métodos `ehIsosceles()`, `ehEquilatero()` e `ehEscaleno()` verificam o tipo de triângulo com base nas propriedades dos lados após verificar se eles formam um triângulo válido com o método `verificaTriangulo()`. Em seguida, testamos a classe `Triangulo` com três instâncias diferentes que representam triângulos com características diferentes. Questão também demonstrada por código no VS Code.
3. Nesta classe `Equipamento`, os métodos `liga()` e `desliga()` alteram o estado do equipamento e evitam ligar ou desligar novamente se já estiver no estado desejado. O método `inverte()` muda o status atual (se ligado, desliga...se desligado, liga), e o método `estaLigado()` retorna o valor atual do atributo `ligado`. A instância da classe `Equipamento` é criada e seus métodos são testados. Questão também demonstrada por código no VS Code.
4. Demonstração e alterações feitas no VS Code; com uso de operadores lógicos true para saldo em conta positivo, retornando true caso `this.saldo <= valor` no caso em que o método `sacar()` fosse modificado conforme a questão 9ª e tipo false, acusando false caso `this.saldo >= valor` deixando true para saldo positivo e o operador condicional false caso acusasse saldo insuficiente para as 3 contas: `c1`, `c2` e `c3` para `saldo()` em conta corrente. De acordo com a 9ª questão, devemos verificar as alterações e implementar/alterar os respectivos métodos citados: `sacar()` e `transferir()`, realizando seus valores lógicos.
5. Nesta classe `Jogador`, temos os métodos para calcular o ataque, realizar um ataque, receber dano e verificar se o jogador está vivo. Também há um método `toString()` para obter uma representação textual do jogador. O código demonstrado no VS Code instancia dois jogadores, permite que um ataque o outro e verifica qual jogador tem mais pontos durante o ataque. Certifiquei-me de que os valores de força, nível e pontos atuais fossem ajustados conforme necessário para seus testes.
6. Segundo os ensinamentos do professor em sala de aula e diante os slides repassados sobre construtores, inicialização de atributos e objetos x referências é importante salientar que a 5ª questão aborda a classe Conta e a instância de dois métodos `sacar()` e `transferir()` como sendo o principal problema, nela altera-se o saldo do Objeto Conta, instanciando e alocando-o para “pequenas situações” do tipo valor : number; Com isso encontra-se erros de compilação por não existir construtores sem parâmetros. Você, programador, deve colocar para rodar o código e interpretar os erros de inicialização correntes.

Já nos primeiros slides, trata-se somente de um método tratado no slide como dizerAlo(), aonde o (“Alô, “+ this.nome) concatena minhas 2 strings `Alo` e `nome`; Igualmente na 6ª questão quando aborda-se o `obterSaudacao() : string { para retornar `${this.texto}`+ `${this.destinatario}` para então eu testar minha classe, que, no meu código utilizei uma variável `saudacaoExemplo` e mais uma extensão de método para exibir na tela minha `saudação()` chamada “Bom Dia, João!” no console.log().

**Justificativa**: Esses códigos são interessantes e podem ser abordados em Java, C, C++ e C#. Podendo também ser aplicados em Banco de Dados aonde persistem informações relevantes para as situações-problema, chamados de atributos ou funções de minha classe.