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 III – Tarde

Turma Especial – 2024.1

Exercício 02

1. A diferença entre **tipagem dinâmica e tipagem estática em Programação Orientada a Objetos (POO)** refere-se à forma como os tipos de dados são tratados em um programa. Irei mostrar cada conceito:

- **TIPAGEM ESTÁTICA:**

Na tipagem estática, **os tipos de dados são verificados em tempo de compilação**. O tipo de uma variável é determinado em tempo de declaração e não pode ser alterado durante a execução do programa. Erros de tipo são detectados durante a compilação, antes mesmo de o programa ser executado. Exemplos de Linguagens de Programação com **Tipagem Estática** incluem Java, C++, Swift, C# e TypeScript;

- **TIPAGEM DINÂMICA:**

Na tipagem dinâmica, **os tipos de dados são verificados em tempo de execução**. O tipo de uma variável é associado aos dados durante a execução e pode ser alterado durante a vida do programa. Erros de tipo só são detectados durante a execução do programa. Exemplos de Linguagens de Programação com **Tipagem Dinâmica** incluem Python, JavaScript, Ruby e PHP;

- **Em RESUMO:**

Em resumo, a principal diferença está no momento em que os tipos são verificados: tipagem estática faz a verificação em tempo de compilação, **enquanto a tipagem dinâmica realiza a verificação em tempo de execução**. **Cada abordagem tem suas vantagens e desvantagens**, e a escolha entre elas depende das necessidades e preferências do desenvolvedor e das características da Linguagem de Programação utilizada.

2. Como foi dito na questão anterior, principalmente **erros de tempo de execução**.

Aqui estão alguns exemplos em **JavaScript**:

- **TypeError: Cannot read property ‘x’ of undefined**:

```
var obj = { };

console.log(obj.x); // Erro: obj.x is not undefined.
```

- **TypeError: Cannot assign to read only property ‘x’ of ‘object’ #<Object>**’:

var obj = { };

Object.freeze(obj); // Torna o objeto imutável

Obj.x = 42; // Erro: Tentando atribuir uma propriedade de objeto imutável

- **TypeError: someFunction is not a function**:

var someFunction = “Não é uma função”;

someFunction(); // Erro: Tentando chamar uma variável como função

- **ReferenceError: variableName is not defined**:

console.log(undefinedVariable); // Erro: Tentando acessar uma variável não definida

- **TypeError: Cannot convert null to object**:

var nullValue = null;

console.log(nullValue.x); // Erro: Tentando acessar uma propriedade de null

- **TypeError: Cannot read property ‘length’ of null**:

var element = document.getElementById(“nonexistent”);

console.log(element.length); // Erro: Tentando acessar a propriedade ‘length’ de null

Lembrando que, em **JavaScript**, é possível mitigar alguns desses erros usando técnicas como verificação de nulos **(‘null’ ou ‘undefined’**), **‘try-catch’** para lidar com **exceções e verificações de tipos antes de realizar operações**. O desenvolvedor deve ser consciente dos tipos de dados que está manipulando e implementar boas práticas para evitar esses erros durante a execução do programa.

3. Em se tratando de uma linguagem **dinamicamente tipada**, como o **JavaScript**, segundo estudos na **stackoverflow.co/2023/#technology,** um Banco de Dados aonde se verifica seu Ecossistema de uso contínuo na atualidade, tem grande importância acadêmica e conceitual para desenvolvimento web.

Exemplificação:

```
function somar(a, b) {
        return a + b;
}
console.log (somar (5, 10);  // saída: 15.
console.log (somar (“5”, 10);  // saída: 510
console.log (somar (5, true); // saída: 5.
console.log (somar (5, null);  // saída: 5.
```

console.log (somar (5, false); // saída: 5.

Pois a tipagem dinâmica do JavaScript leva a erros de runtime:

A tipagem dinâmica em linguagens de programação pode levar a situações problemáticas quando não há uma clara compreensão dos tipos de dados que estão sendo manipulados. Vamos considerar um exemplo em JavaScript, uma linguagem de tipagem dinâmica:

```javascript
function somar(a, b) {
    return a + b;
}

console.log(somar(5, 10));  // Saída: 15
console.log(somar("5", 10));  // Saída: "510"
```

Neste exemplo, a função `somar` é projetada para somar dois valores, mas como JavaScript é dinamicamente tipado, ela não verifica os tipos dos argumentos em tempo de compilação. Isso significa que você pode chamar a função com diferentes tipos de dados.

Na primeira chamada, `somar(5, 10)`, os argumentos são números e a função retorna a soma correta, que é `15`. No entanto, na segunda chamada, `somar("5", 10)`, o primeiro argumento é uma string e o segundo é um número. JavaScript tentará converter dinamicamente a string para um número e realizar a adição, resultando em uma concatenação de strings em vez de uma soma aritmética, retornando `"510"`.

Isso pode ser problemático porque a função `somar` foi projetada para operar em números, mas não há verificação de tipo para garantir que os argumentos sejam sempre do tipo esperado. Isso pode levar a bugs difíceis de identificar, especialmente em programas mais extensos e complexos, nos quais é mais difícil rastrear as origens de problemas causados por tipos de dados inesperados.

Em linguagens com tipagem estática, esse tipo de erro seria detectado em tempo de compilação, proporcionando uma detecção precoce e ajudando os desenvolvedores a corrigirem potenciais problemas antes mesmo de executar o programa.

4. **A Linguagem C** é frequentemente citada como tendo **tipagem estática**, mas **tipagem fraca**. Isso se refere ao fato de que, embora os tipos de dados sejam verificados em tempo de compilação (tipagem estática), a linguagem permite operações que, em outras linguagens permite operações com tipagem mais forte, seriam consideradas mais restritivas ou gerariam erros.

Um exemplo disso é **a capacidade de realizar operações de ponteiro de forma mais flexível do que em C**, que pode levar a resultados inesperado se não for manuseada com cuidado.

Vamos considerar o seguinte exemplo como imagem de código:

```c
#include <stdio.h>

int main() {
    int numero_inteiro = 10;
    float numero_flutuante = 5.5;

    // Ponteiro Genérico (void pointer)
    void *ponteiro_generico;

    // Atribui o valor da variável inteira ao ponteiro genérico
    ponteiro_generico = &numero_inteiro;
    printf("Valor inteiro: %d\n", *(int*) ponteiro_generico);

    // Atribuir o valor da variável decimal ao ponteito genérico
    ponteiro_generico = &numero_decimal;
    printf("Valor decimal : %f\n", *(float*) ponteiro_generico);

    return 0;

}
```

Neste exemplo, usamos um ponteiro genérico (void pointer) em C para armazenar o endereço de uma variável inteira e, em seguida, o endereço de uma variável decimal. O ponteiro genérico permite que você armazene o endereço de qualquer tipo de dado, e **você pode fazer a desreferência do ponteiro usando um tipo diferente, como int ou float**.

Embora isso **ofereça flexibilidade**, também pode levar a erros sutis se não for tratado com cuidado, uma vez que a linguagem **não impõe restrições estritas sobre essas operações de ponteiro e desreferência**. Em linguagens com tipagem mais forte, como Java e C#, **essas operações seriam mais restritas e a tentativa de desreferenciar um ponteiro de forma incompatível geraria erros em tempo de compilação**.

Portanto, quando dizemos que C possui tipagem fraca, estamos nos referindo à flexibilidade que a linguagem oferece em operações de tipos, especialmente no contexto de ponteiros, em comparação com linguagens com sistemas de tipos mais fortes.

5. Pesquisando em uma ferramenta da Internet, como mecanismo fundamental de busca, **um tipo ‘any’** seria benéfico caso em TypeScript, o tipo **‘any’** for usado para representar uma variável que pode ter qualquer tipo. Embora seja muitas vezes recomendado evitar o uso excessivo de **‘any’**, há situações em que ele pode ser benéfico, **especialmente quando você está migrando um código JavaScript para TypeScript** e ainda não tem informações suficientes sobre os tipos de certos valores.

   Aqui embaixo está um exemplo onde um tipo **‘any’** pode ser útil:

   // Suponha que esteja integrando um API JavaScript existente.

   // e não tem informações detalhadas sobre os tipos retornados.

```
// Função que chama um API JavaScript e retorna um resultado desconhecido
function chamarAPI() : any {
// Simulando uma chamada de API que retorna um resultado desconhecido
    return Math.random() > 0.5 ? “Sucesso” : 42;
}

//  Utilizando o resultado da chamada da API
const resultado = chamarAPI();

// Você pode atribuir o resultado a qualquer tipo sem gerar um erro de tipo
const comprimentoDaString : number = resultado.length;  // Funciona bem para strings
const multiplicação : number = resultado * 2;  // Funciona bem para números
```

Neste exemplo, a função **‘chamarAPI’ retorna um valor que pode ser uma string ou um número, dependendo de uma condição simulada**. Usar o tipo **‘any’** permite que você armazene o resultado em uma variável sem ter que especificar um tipo específico. Isso pode ser útil quando você está trabalhando um código JavaScript existente e ainda não tem informações suficientes sobre os tipos de retorno.

No entanto, é importante notar que **o uso excessivo de ‘any’ pode comprometer a segurança de tipos do TypeScript**, já você perde a verificação de tipo forte que a linguagem oferece. Sempre que possível, é recomendado utilizar tipos mais específicos para manter **a robustez do sistema de tipos** do TypeScript.

6. Não. A tipagem do TypeScript não é considerada fraca por permitir que as variáveis do tipo **‘number’** aceitem tanto inteiros quanto números de ponto flutuante. Na verdade, isso é uma característica da tipagem fraca.

A tipagem forte refere-se à restrição estrita dos tipos em uma linguagem de programação, onde certas operações ou atribuições são permitidas apenas entre tipos compatíveis. Por exemplo, em uma linguagem com tipagem forte, tentar somar uma string a um número seria geralmente um erro de tipo.

Já a tipagem fraca permite uma maior flexibilidade em relação aos tipos. No caso do TypeScript, o tipo **‘number’** é usado para representar tanto inteiros quanto números de ponto flutuante. **Isso não significa que o TypeScript tem uma tipagem fraca em geral, pois ainda impõe verificações de tipo durante a compilação e fornece um sistema de tipos robusto**.

Portanto, a capacidade do TypeScript de tratar números como um tipo único, abrangendo tanto inteiros quanto números de ponto flutuante, é uma escolha de design que oferece flexibilidade **sem sacrificar completamente a segurança de tipos**. No entanto, essa característica específica não torna o TypeScript uma linguagem de tipagem fraca.

7. Demonstrada no VS Code, em Linguagem TypeScript, transpilado para JavaScript.

8. Configuração de arquivo **TypeScript**:

Salvo aqui algumas observações:

- O que é ***BUILD***?

Build é o termo usado para **identificar uma versão compilada de um programa** ou Sistema Operacional. No Windows 10 e 11, a Microsoft lança uma série de builds com o objetivo de **melhorar a segurança do sistema e trazer novos recursos**. São as tradicionais atualizações oferecidas pelo **Windows Update**.
Através da questão 8, quando se digita **npm install typescript –g** , o programa VS Code **instala e exibe para mim no teclado para jogar na tela** a versão global da linguagem de programação **TypeScript**.

- O que é o **OutDir**?

OutDir é um setting (uma configuração) do **TSConfig** para saída do diretório no programa. O navegador não irá entender **TypeScript**, por isso, ele transpila para **JavaScript**. Quando se digita **tsc –v** ou **tsc –w** ativa o **watch mode** (modo assistir de exibição do conteúdo) e o **diretório automaticamente reconhecerá OutDir**.

- Significado de estudar essa ferramenta?

Exemplo:
const myName : string = “Vinicius”;
console.log(“Hello, ”+ myName);
No **prompt de comando** ou **cmd** digite a seguinte linha de código:
tsc (typescript compiler) **nome_arquivo.ts**
node (javascript compiler) **nome_arquivo.js**

Isto é:
tsc index.ts
node index.js

- Há dois tipos de **JSON (Java Script Object Notation)**:

PACKAGE-JSON e o TSCONFIG.JSON;

- **allowUnreachableCode : true**

É um tipo de **warning** da documentação www.typescriptlang.org/pt/tsconfig e significa **permissão de código inacessível**, coloco true para verdadeiro;

- **noImplicitAny : true**

É, na tradução livre, **valor não implícito de qualquer tipo**, como dito no item anterior, é desconhecido por boa parte da pesquisa acadêmica de desenvolvedores de projetos com pesquisas relacionadas à área da computação. Faz parte do

TSCONFIG.JSON. Coloco true para verdadeiro na opção de deixar implícito nesse caso.

- **Target como ES3**, além disso, utilize a classe do exercício anterior e veja como ela é **transpilada para JS**:

Alvo, **localização do meu arquivo**, utilizado no **padrão internacional Ecma Script versão 3.0**, localizo e coordeno o arquivo, utilizando geralmente para fins didáticos as **versões mais atuais** a fim de evitar **bugs**;

- **strictNullChecks** para false:

Verificações nulas estritas, isto é, eu **desabilito esta função** em casos de configuração de código para false, gerando assim ***.ts** e o ***.js** ;

Fim da atividade!