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 02**

1. Levando em consideração o **Paradigma Orientado a Objetos**, tem uma grande preocupação, entre os **desenvolvedores de software**, em esconder o que não é importante e em realçar o que é importante. Neste paradigma, implementa-se um conjunto de classes que definem objetos. Cada classe determina o comportamento (definido nos métodos) e estados possíveis (atributos) de seus objetos, assim como relacionamento entre eles.

Esse é o paradigma mais utilizado em aplicações comerciais e as principais linguagens o implementam: C#, Java, PHP, Ruby, C++, Python, etc...

Deste modo, em especial **TypeScript**, trata-se de um super set (superconjunto) de **JavaScript**, nele se adiciona tipagem estática e explícita; Como se verifica isso?

***Justificativa:*** A principal diferença entre **tipagem estática** e **tipagem dinâmica** está relacionada à forma como as linguagens de programação tratam os tipos de dados durante o desenvolvimento e execução do código.

* ***Tipagem Estática:***

Em linguagens de programação com tipagem estática, os tipos de dados das variáveis são determinados em tempo de compilação e são verificados antes da execução do programa;

As declarações de tipos são obrigatórias, o que significa que você precisa especificar o tipo de uma variável no momento em que ela é declarada;

Os erros de tipos são frequentemente capturados durante a compilação, o que pode ajudar a evitar muitos erros comuns antes mesmo de executar o código;

Exemplos de linguagens com tipagem estática incluem: TypeScript, Java, C++ e C#;

* ***Tipagem Dinâmica:***

Em linguagens de programação com tipagem dinâmica, os tipos de dados das variáveis são verificados em tempo de execução, ou seja, quando o código está sendo executado pelo computador;

As declarações de tipos não são necessárias, pois as variáveis podem mudar de tipo durante a execução do programa;

Os erros de tipo só são revelados durante a execução, o que pode levar a problemas difíceis de diagnosticar, pois os erros só ocorrem quando o código é realmente executado;

Exemplos de linguagens com tipagem dinâmica incluem: JavaScript, Python, Ruby e PHP;

* Ambas as abordagens têm vantagens e desvantagens:
* **Tipagem Estática**: oferece a vantagem de pegar erros de tipo em tempo de compilação, resultando em código mais robusto e seguro. No entanto, pode ser necessário mais esforço para escrever código devido à necessidade de declarar tipos explicitamente;
* **Tipagem Dinâmica**: permite uma flexibilidade maior, pois as variáveis podem ser usadas de maneira mais fluida sem a necessidade de declarações de tipo. No entanto, erros de tipo podem surgir apenas durante a execução, o que pode levar a resultados inesperados.

A escolha entre **tipagem estática** e **tipagem dinâmica** depende das necessidades do projeto, das preferências da equipe de desenvolvimento e do equilíbrio entre segurança e flexibilidade desejado.

Com isso, **TypeScript** é um “sabor” ou uma “variante” do **JavaScript**. O relacionamento entre TypeScript (TS) e JavaScript (JS) é único entre linguagens de programação modernas, então aprender mais sobre esse relacionamento vai te ajudar a entender como TypeScript soma ao JavaScript.

JavaScript (também conhecido como ECMAScript) começou sua vida como uma linguagem de script simples para navegadores. Na época em que foi inventada, sempre foi esperado que fosse usada para pequenos snippets de código embarcados em uma página web - escrevendo mais do que uma dúzia de linhas de código seria algo incomum. Por isso, navegadores jovens da época executavam tal código bem devagar. Entretanto, com o tempo, JS se tornou mais e mais popular e desenvolvedores web começaram a usar ele para criar experiências interativas.

Desenvolvedores de navegadores web responderam a esse uso aumentado de JS otimizando seus motores de execução (compilação dinâmica) e estendendo o que poderia ser feito com ele (adicionando APIs), o que por sua vez fez os desenvolvedores web o usarem ainda mais. Em websites modernos, seu browser frequentemente está rodando aplicações que têm centenas de milhares de linhas de código. Esse é o crescimento longo e gradual da “web”, começando como uma rede simples de páginas estáticas e evoluindo em uma plataforma para aplicações ricas de todos os tipos.

Nós dissemos antes que algumas linguagens não permitiriam esses programas bugados nem serem executados. Detecção de **erros sem execução do código** é chamada de **verificação estática**. Determinar o que é um erro e o que não é baseado nos tipos dos valores sendo operados é chamado de verificação estática de tipos. **TypeScript** verifica um programa por erros antes de sua execução e faz isso baseado nos tipos dos valores, é um **verificador de tipo estático**.

Fonte: <https://www.typescriptlang.org/pt/docs/handbook/typescript-from-scratch.html>

[https://www.treinaweb.com.br/paradimas-de-programacao#](https://www.treinaweb.com.br/paradimas-de-programacao)

1. A **tipagem dinâmica** é um conceito em programação onde o tipo de uma variável é determinado em tempo de execução, ao contrário da tipagem estática, onde o tipo é definido em tempo de compilação. Embora a tipagem dinâmica tenha suas vantagens em termos de flexibilidade e produtividade, também apresenta alguns problemas potenciais:
2. **Erros em tempo de execução:** como os tipos são determinados em tempo de execução, erros relacionados a tipos podem ocorrer somente quando o código está sendo executado. Isso pode levar a falhas inesperadas e difíceis de rastrear, já que o erro pode não ser detectado até que uma determinada parte do código seja alcançada.
3. **Falta de verificação de tipo em tempo de compilação:** a tiagem dinâmica não permite que o compilador detecte erros de tipo antes da execução do programa. Isso significa que problemas relacionados a tipos só são identificados quando o programa é realmente executado, tornando a depuração mais complicada e sujeita a erros.
4. **Código menos legível e previsível:** a ausência de informações claras sobre os tipos das variáveis pode tornar o código mais difícil de entender, uma vez que os desenvolvedores precisam estar constantemente conscientes dos tipos que estão manipulados em cada momento.
5. **Refatoração mais difícil:** quando se trabalha com tipagem dinâmica, a refatoração do código pode ser mais complicada, pois mudanças em um local do código podem afetar indiretamente outras partes que dependem dos tipos das variáveis.
6. **Desempenho:** em muitos casos, a tipagem dinâmica pode resultar em um desempenho inferior, já que a verificação de tipos em tempo de execução pode ter um impacto negativo na velocidade de execução do programa.
7. **Falhas de documentação:** a tipagem dinâmica pode dificultar a criação de documentação precisa e abrangente, pois os tipos das variáveis podem não estar claramente definidos.
8. **Maior potencial para erros sutis:** a natureza dinâmica dos tipos pode levar a erros sutis, onde as conversões de tipo não são tratadas corretamente, causando comportamentos inesperados.

No entanto, é importante ressaltar que a escolha entre tipagem dinâmica e tipagem estática depende das necessidades e das preferências do projeto e da equipe de desenvolvimento. Existem situações em que a tipagem dinâmica pode ser vantajosa, especialmente para prototipagem rápida e ambientes que requerem grande flexibilidade. Nesses casos, as boas práticas de programação, testes abrangentes e revisões de código cuidadosas podem ajudar a mitigar alguns problemas associados à tipagem dinâmica.

1. Um exemplo clássico que ilustra os problemas de tipagem dinâmica pode ser encontrado na linguagem de programação JavaScript. JavaScript é amplamente utilizado para **desenvolvimento web** e é conhecido por sua **tipagem dinâmica**. Vamos considerar um cenário em que a tipagem dinâmica em JavaScript pode **causar problemas**:

Suponha que você esteja desenvolvendo um aplicativo web e tenha o seguinte código em JavaScript:

var total = 0;

var numbers = [10, 20, “30”, 40, “50”];

for (var i = 0; i < numbers.lenght; i++) {

total += numbers[i];

}

console.log(“Total: “, total);

Neste exemplo, a variável ‘numbers’ é um array que contém números e strings representando números. Como JavaScript permite a mistura de tipos em um array e executa conversões automáticas de tipos, o código irá somar os elementos de um array.

No entanto, devido à tipagem dinâmica e à conversão automática de tipos, o resultado será surpreendente:

Total: 1020304050

O resultado não é o que normalmente se espera ao somar os números em um array. Isso ocorre porque o JavaScript converte automaticamente as strings em números durante a operação de adição, resultando em uma concantenação de strings em vez de soma numérica correta.

Nesse caso, a tipagem dinâmica do JavaScript levou a um resultado inesperado e incorreto. Em linguagens de tipagem estática, como Java ou C++, um erro e compilação seria gerado devido à tentativa de adicionar um número e uma string, permitido que o erro fosse identificado antes da execução do programa.

Isso demonstra como a tipagem dinâmica pode tornar difícil a detecção de erros de tipo e levar a comportamentos inesperados em tempo de execução, o que pode ser problemático, especialmente em ambientes de desenvolvimento em que a previsibilidade e a robustez são essenciais.

**Outros exemplos:**

Já outra linguagem como o TypeScript, é open source e desenvolvido pela Microsoft, possui o mesmo criador das linguagens C# e Delphi; Larga adoção sendo utilizado com muitos frameworks conhecidos como Angular, React, Express...

O navegador não entende TypeScript, por isso o código é “transpilado” para JavaScript. O JavaScript, de tipagem dinâmica e implícita, é muito usada nos meios profissionais, com maioria de 64,96 % de aceitação nas empresas como 1º lugar de maior linguagem de programação utilizada no Ecossistema de Tecnologia da Informação atual. Ficando atrás dela, HTML, CSS, SQL e TypeScript ficaria mais ou menos em 6º lugar com 30,19% de aceitação.

A tipagem dinâmica do JavaScript leva a erros de runtime:

function soma (a, b) {

return a + b;

}

console.log(soma(1, 2)); // 3

console.log(soma(1, “2”)); // 12

console.log(soma(1, true)); // 2

console.log(soma(1, null)); // 1

**Por que não apenas Javacript?**

Tipagem estática do TypeScript leva à confiabilidade do software. Menos erros ocorrem em tempos de execução. Com a exposição de warnings e erros durante o projeto/desenvolvimento, assim como menos erros “descobertos” por usuários.

1. A **linguagem de programação C** é conhecida por ter **tipagem estática**, o que significa que os tipos das variáveis são verificados em tempo de compilação e não podem ser alterados durante a execução. No entanto, a linguagem C também é considerada uma linguagem de **tipagem fraca**. A **tipagem fraca** se refere à capacidade de a linguagem realizar automaticamente conversões implícitas entre tipos diferentes, mesmo que essas conversões possam não ser logicamente apropriadas. Isso pode levar a resultados inesperados e comportamentos difíceis de prever.

Um exemplo clássico em C que ilustra a tipagem fraca é a operação de adição entre um úmero e um caractere. Vamos considerar o seguinte código:

#include <stdio.h>

int main() {

int num = 5;

char ch = ‘A’;

int result = num + ch;

printf(“Result: %d\n”, result);

return 0;

}

Neste exemplo, estamos adicionando um número inteiro (‘num’) a um caractere (‘ch’). No entanto, como C possui **tipagem fraca**, ele permitirá a conversão implícita do caractere para seu valor ASCII correspondente antes de realizar a adição. O valor ASCII de ‘A’ é 65. Portanto, a adição será realizada da seguinte forma:

result = num + ch = 5 + 65 = 70;

result = 70;

Embora essa conversão implícita possa ser úti em alguns casos, ela também pode levar a resultados confusos e a erros difíceis de rastrear, especialmente quando não se espera que uma conversão desse tipo ocorra.

Isso exemplifica a tipagem fraca da linguagem C, onde operações entre tipos diferentes podem ser realizadas automaticamente, independente da semântica subjacente da operação. Isso pode levar a resultados inesperados e comportamentos imprevisíveis, o que exige que os programadores sejam muito cuidadosos ao lidar com diferentes tipos de dados em C.

1. Em TypeScript, o tipo ‘**any**’ é usado para representar um valor que pode ter qualquer tipo. Embora seja geralmente recomendado evitar o uso do tipo ‘any’, há casos em que ele pode ser benéfico, **especialmente quando você está trabalhando com código legado, integrações externas ou situações em que** **a estrutura do tipo é desconhecida ou complexa demais** para ser representada adequadamente.

Vamos considerar um exemplo em que o uso do tipo ‘any’ pode ser benéfico:

// Suponha que estamos trabalhando com dados JSON vindos de uma API externa

const jsonData = ‘{“name”: “Alice”, “age”: 30, “isStudent”: true}’;

// Fazendo o parsing do JSON usando o tipo ‘any’

const parsedData: any = JSON.parse(jsonData);

// Agora podemos acessar as propriedades sem erros de tipo

console.log(parseData.name); // Alice

console.log(parseData.age); // 30

console.log(parseData.isStudent); // True

// Realizando operações sem verificar explicitamente os tipos

const doubledAge = parseData.age \* 2;

console.log(doubledAge); // 60

Neste exemplo, estamos lidando com **dados JSON vindos de uma API externa**. Como o formato e a estrutura exata dos dados são desconhecidas em tempo de compilação, **não podemos criar uma interface TypeScript ou um tipo específico** para eles. Nesse caso, o uso do tipo ‘any’ pode ser útil para permitir que os dados sejam analisados e acessados **sem erros de tipo**.

No entanto, é importante notar que **o uso excessivo do tipo ‘any’ pode comprometer a** **segurança de tipos do TypeScript** e anular muitos dos benefícios que a linguagem oferece. Sem a verificação de tipos rigorosa, erros podem passar despercebidos e o código pode se tornar mais difícil de manter e entender.

Portanto, **embora o tipo ‘any’ possa ser útil em situações específicas, é uma boa prática usar tipos mais precisos e específicos** sempre que possível para aproveitar ao máximo as vantagens de segurança de tipos oferecida pelo TypeScript.

***Deste modo, há uma outra análise para o tipo Any:***

Any é a **cláusula de escape** **do Typescript**. Você pode usar any para **declarar uma seção do seu código para ser dinâmica ou semelhante ao JavaScript**, ou para contornar as limitações do sistema de tipos. Um bom caso de uso para any é **o parsing de JSON**:

const myObject = JSON.parse("{}");

O escape TypeScript do tipo Any declara ao Typescript para confiar no seu código como **sendo seguro porque você sabe mais sobre ele**, **mesmo se isso não seja estritamente verdadeiro**.

myObject.x.y.z; // Por exemplo, esse código iria falhar.

Utilizar any **provê a você a habilidade de escrever código próximo ao JavaScript original sem a segurança de tipos**. Any é muito semelhante a um 'tipo coringa' do qual você pode substituir com qualquer tipo (exceto never) para fazer um tipo atribuível a outro.

declare function debug(value: any): void;

debug("a string");

debug(23);

debug({ color: "blue" });

Cada **chamada para debug é permitida porque você pode substituir o tipo any com o tipo do argumento correspondente**. TypeScript irá considerar a posição dos anys de diferentes formas, como, por exemplo, com essas tuplas como argumento da função abaixo:

declare function swap(x: [number, string]): [string, number];

declare const pair: [any, any];

swap(pair);

A chamada de swap é autorizada **porque o argumento pode ser correspondente ao repor o primeiro any do par com um número e o segundo any com uma string**.

Se tuplas são novas para você, veja:

example:tuples;

**Unknown** é um **irmão do tipo any**, **Any** é como dizer "Eu sei o que é melhor", já Unknown é como dizer "Eu não sei o que é o melhor, **então você precisa dizer ao TS o tipo**".

1. **Não**, **porque a tipagem do TypeScript não é fraca, mas** **sim flexível**. O exemplo da variável do tipo ‘number’ aceitar tanto inteiros quanto números de ponto flutuante não se relaciona com tipagem fraca.

A **tipagem fraca** se refere à capacidade de uma linguagem de programação realizar conversões implícitas entre tipos diferentes, mesmo quando essas conversões podem não ser logicamente apropriadas. No exemplo da tipagem fraca que dei anteriormente em C, a adição de um número inteiro e um caractere resultou em uma conversão implícita do caractere para seu valor ASCII antes da operação.

O **TypeScript**, por outro lado, é uma linguagem que busca trazer a verificação de tipos estática para o JavaScript. Embora seja flexível em alguns aspectos, como a conversão de tipos numéricos, isso não se implica necessariamente em tipagem fraca. No caso do TypeScript, essa flexibilidade é mais uma escolha de projeto para melhor acomodar os tipos de dados comuns em JavaScript, onde não há uma distinção rígida entre inteiros e números de ponto flutuante como em algumas outras linguagens.

Em síntese, **a tipagem do TypeScript não é considerada fraca**. Ela é uma abordagem que tenta fornecer segurança de tipos enquanto ainda é capaz de acomodar algumas das nuances e peculiaridades do JavaScript.

1. Feita no VS Code.
2. Feita no VS Code.

***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](http://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 de maneira bem objetiva meus arquivos, utilize 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** ;

* Crie exemplos que mostrem o efeito das alterações no arquivo de configuração:

Não entendi essa parte, atenção falar com o professor da disciplina Ely.

***Fim da atividade!***