Curso de TypeScript

Uma introdução ao superset

O que é TypeScript?

- TypeScript é um **superset** para a linguagem JavaScript;
- Ou seja, adiciona funções ao JavaScript, como a **declaração de tipos** de variável;
- Pode ser utilizado com frameworks/libs, como: **Express e React**;
- Precisa ser **compilado em JavaScript**, ou seja, não executamos TS;
- Desenvolvido e mantido pela **Microsoft**;

Por que TypeScript?

- Adiciona **confiabilidade** ao programa (tipos);
- Provê novas funcionalidades a JS, como **Interfaces**;
- Com TS podemos **verificar os erros antes da execução** do código, ou seja, no desenvolvimento;
- Deixa JavaScript **mais explícito**, diminuindo a quantidade de bugs;
- Por estes e outros motivos **perdemos menos tempo com debug**;

Instalando o TypeScript

- Para instalar o TypeScript vamos utilizar o **npm**;
- O nome do pacote é **typescript**;
- E vamos adicionar de forma global com a **flag -g**;
- A partir da instalação temos como **executar/compilar** TS em qualquer local da nossa máquina, com o comando **tsc**;

Primeiro programa

- Vamos criar nosso **primeiro programa em TS**;
- O intuito é entender como **compilar e executar** o arquivo gerado pelo processo de compilação;
- Vamos a prática!

Live Server

- A cada vez que geramos o arquivo **temos que recarregar a página**, o que pode se tornar chato ao longo tempo;
- A extensão **LiveServer** faz com que a cada alteração de código, o navegador atualize automaticamente;
- **Inclusive o compile** provoca a atualização;
- Vamos instalar!

Como tirar o melhor proveito do curso

- **Programar junto**, e não ficar só assistindo;
- Criar seus **próprios exemplos** (para os conteúdos das aulas);
- **Projetos próprios** ao final do curso ou durante;
- Realizar os **exercícios** propostos;
- **Responder** ou criar novas perguntas no Q&A;
- Dica extra: ver depois fazer;

Introdução ao TS

Conclusão

Desafio 1

1. Vamos criar um novo projeto para treinar a compilação e execução de código;
2. O desafio é criar uma função que aceita dois argumentos, somente do tipo numérico (number);
3. E exibe a soma entre os dois;

Fundamentos do TS

Introdução

O que são tipos?

- Em TypeScript a principal função é **determinar tipos para os dados**;
- Isso vai garantir a **qualidade do código**;
- Além de fazer o TS nos **ajudar na hora do desenvolvimento**;
- Ou seja, precisamos **definir corretamente o tipo** das variáveis, dos retornos das funções, das manipulações de dados;
- Consequentemente teremos um software melhor programado e é este o principal intuito do TS;

Tipos primitivos

- Há diversos tipos em TS, vamos começar pelos **primitivos**;
- Que são: **number**, **string** e **boolean**;
- Todos estes são inseridos com **letras minúsculas**;
- E por mais óbvio que pareça, eles servem para: números, textos e booleanos;

Conhecendo o number

- O tipo **number** garante que o dado seja um número;
- Logo, podemos **inserir apenas números** na variável;
- E também mudar o valor para outro número;
- O TypeScript **possibilita também a inserção de métodos numéricos** apenas;
- Vamos ver na prática;

Conhecendo o string

- O tipo **string** garante que o dado seja um texto;
- Logo, podemos **inserir apenas texto** na variável;
- E também mudar o valor para outro texto;
- O TypeScript **possibilita também a inserção de métodos de texto** apenas;
- Vamos ver na prática;

Conhecendo o boolean

- O tipo **boolean** garante que o dado seja um booleano (true ou false);
- Logo, podemos **inserir apenas booleans** na variável;
- E também mudar o valor para outro boolean;
- Vamos ver na prática;

TS e a aplicação

- Talvez já tenha ficado claro, mas programar com TS é como um **pair programming**;
- Temos sempre alguém para nos avisar se algo é **feito errado**;
- Depois da compilação **o TS não tem mais efeito**, ele não pode mais nos ajudar;
- Por isso há **uma trava de compilação com erros**;
- Além de erros, o TS também proporciona **avisos**;

Type annotation e Type inference

- Estes dois conceitos vão nos acompanhar em **todo o processo do desenvolvimento** de aplicações;
- **Annotation** é quando definimos o tipo de um dado manualmente;
- **Inference** é quando o TS identifica e define o tipo de dados para nós;
- Futuramente entraremos em mais alguns detalhes sobre estes recursos;
- Vamos ver na prática!

Gerando arquivo de configuração

- O TS pode ser configurado de **muitas maneiras**;
- Mas para isso precisamos do **arquivo de configuração**;
- Para criar ele utilizamos: **tsc --init**
- Agora podemos mudar várias opções em relação ao que o TS executa e também feito o compile;

Compilar TS automaticamente

- Estamos **sofrendo** muito, não é?
- Para gerar a compilação automática podemos utilizar o comando: **tsc -w**
- O **nosso output será gerado automaticamente sempre que salvarmos** o projeto;
- Agora vai!

Fundamentos do TS

Conclusão

Desafio 2

1. Crie uma variável que recebe um número;
2. Converta este número para string em uma nova variável;
3. Esta variável de conversão deve estar tipada por inferência;
4. Imprima este número em uma string maior, utilizando o recurso de template strings ou concatenação;

Avançando em tipos

Introdução

Arrays

- Podemos especificar um **array** como tipo também;
- Se temos um array de números: **number[]**
- Se é um array de string: **string[]**
- Isso acontece pois **geralmente os arrays possuem apenas um único tipo** de dado entre seus itens;
- Vamos ver na prática!

Outra sintaxe de arrays

- Os tipos de array possuem duas sintaxes;
- **Obs:** a da primeira aula é a mais utilizada;
- Podemos também fazer desta maneira: **Array<number>**
- Vamos ver na prática!

O tipo any

- O **any** transmite ao TS que qualquer tipo satisfaz a variável;
- Devemos **evitar ao máximo evitar este tipo**, pois vai contra os princípios do JavaScript;
- **Dois casos de uso:** o tipo do dado realmente não importa e arrays com dados de múltiplos tipos;
- Vamos ver na prática!

Tipo de parâmetro de funções

- Podemos **definir o tipo de cada parâmetro de uma função**;
- Assim condicionamos o seu uso correto;
- A sintaxe é: function minhaFuncao(**nome: string**) { }
- Agora podemos passar o parâmetro nome, **apenas como texto**;
- Vamos ver na prática!

Tipo de retorno de funções

- Os **retornos** também podem ser definidos por nós;
- Para isso utilizamos a sintaxe: function myFunction()**: number { }**
- Esta função **retorna um número**;
- Vamos ver na prática!

Funções anônimas em TS

- O TypeScript consegue nos **ajudar também em funções anônimas**;
- **Fazendo uma validação do código digitado**, nos fornecendo dicas de possíveis problemas;
- Exemplo: métodos não existentes;
- Vamos ver na prática!

Tipos de objetos

- Podemos determinar tipos para objetos também;
- A sintaxe é: **{prop: tipo, prop2: tipo2}**
- Ou seja, estamos determinando quais tipos as **propriedades de um objeto possuem**;
- Vamos ver na prática!

Propriedades opcionais

- Nem sempre os objetos possuem todas as propriedades que poderiam possuir;
- Por isso temos as **propriedades opcionais**;
- Para ter esse resultado devemos colocar uma interrogação: **{nome: string, sobrenome?: string}**
- Vamos ver na prática!

Validação de props opcionais

- Quando a propriedade é opcional, **precisamos criar uma validação**;
- Isso acontece **por que o TypeScript não nos ajuda mais**, já que ele deixa de controlar o valor que recebemos;
- Para isto utilizamos uma **condicional if**, e conseguimos resolver a situação;
- Vamos ver na prática!

Union types

- O **Union type** é uma alternativa melhor do que o any;
- Onde podemos **determinar dois tipos** para um dado;
- A sintaxe: **number | string**
- Vamos ver na prática!

Avançando com Union Types

- Podemos utilizar **condicionais** para validação do tipo de union types;
- Com isso é possível **trilhar rumos diferentes**, baseado no tipo de dado;
- Para checar o tipo utilizamos **typeof**;
- Vamos ver na prática!

Type alias

- **Type alias** é um recurso que permite criar um tipo e determinar o que ele verifica;
- Desta maneira **temos uma maneira mais fácil de chamá-lo** em vez de criar expressões complexas com Union types, por exemplo;
- Vamos ver na prática!

Introdução às interfaces

- Uma outra maneira de **nomear um tipo de objeto**;
- Podemos **determinar um nome** para o tipo;
- E também escolher **quais as propriedades e seus tipos**;
- Vamos ver na prática!

Diferença entre type alias e interfaces

- Na maioria das vezes **podemos optar entre qualquer um dos recursos** sem problemas;
- A única diferença é que **a Interface pode ser alterada ao longo do código**, já o alias não;
- Ou seja, se pretendemos mudar como o tipo se conforma, devemos escolher a Interface;
- Vamos ver na prática!

Literal types

- **Literal types** é um recurso que permite colocar valores como tipos;
- Isso restringe o uso a não só tipos, **como também os próprios valores**;
- Este recurso é **muito utilizado com Union types**;
- Vamos ver na prática!

Non-null Assertion Operator

- Às vezes o TypeScript pode evidenciar um erro, **baseado em um valor que no momento do código ainda não está disponível**;
- Porém se sabemos que este valor será preenchido, podemos evitar o erro;
- Utilizamos o caractere **!**
- Vamos ver na prática!

Bigint

- Com o tipo **bigint** podemos declarar números com valores muito altos;
- Podemos utilizar a **notação literal**, exemplo: 100n
- Para este recurso precisamos **mudar a configuração do TS**, para versão mínima de ES2020
- Vamos ver na prática!

Symbol

- De forma resumida, o **Symbol** cria uma referência única para um valor;
- Ou seja, mesmo que ele possua o mesmo valor de outra variável, **teremos valores sendo considerados diferentes**;
- Vamos ver na prática!

Avançando em tipos

Conclusão

Narrowing

Introdução

O que é narrowing?

- **Narrowing** é um recurso de TS para identificar tipos de dados;
- Dando assim uma direção diferente a execução do programa, **baseada no tipo que foi identificado**;
- Há situações em que **os tipos podem ser imprevisíveis**, e queremos executar algo para cada uma das possibilidades;
- Isso é fundamental também para **evitar erros do compilador**, identificando e resolvendo os possíveis erros na hora do desenvolvimento;

Typeof type guard

- O **type guard** é basicamente uma validação do tipo utilizando o typeof;
- Desta maneira podemos **comparar o retorno do operador** com um possível tipo;
- **Todos os dados vem como string**, exemplo: “string”, “number”, “boolean”
- E a partir disso realizamos as bifurcações;
- Vamos ver na prática!

Checando se valor existe

- Em JavaScript podemos **colocar uma variável em um if**, e se houver algum valor recebemos um true;
- Quando não há valor um false;
- **Desta maneira conseguimos realizar o narrowing também**, é uma outra estratégia bem utilizada;
- Vamos ver na prática!

Operador instanceof

- Para além dos tipos primitivos, podemos trabalhar com **instanceof**;
- Checando se um dado pertence a uma determinada classe;
- E ele vai servir até para as **nossas próprias classes**;
- Vamos ver na prática!

Operador in

- O operador in é utilizado para **checar se existe uma propriedade no objeto**;
- Outro recurso interessante para o narrowing;
- Pois propriedades também podem ser **opcionais**;
- Vamos ver na prática!

Narrowing

Conclusão da seção

Desafio 3

1. Vamos criar uma função que recebe review dos usuários, precisamos utilizar o narrowing para receber o dado;
2. As possibilidades são boolean e number;
3. Serão enviados números de 1 a 5 (estrelas), prever uma mensagem para cada uma destas notas;
4. Ou false, que é quando o usuário não deixa uma review, prever um retorno para este caso também;

Aprofundando em funções

Introdução da seção

Funções que não retornam nada

- Podemos ter funções que não retornam valores;
- Qual seria o **tipo de dado** indicado para essa situação?
- Neste caso utilizamos o **void**!
- Ele vai dizer ao TS que **não temos um valor de retorno**;
- Vamos ver na prática!

Callback como argumento

- Podemos inserir uma **função de callback** como argumento de função;
- Nela conseguimos **definir o tipo de argumento aceito pela callback**;
- E também o **tipo de retorno** da mesma;
- Vamos ver na prática!

Generic functions

- É uma estratégia para quando **o tipo de retorno é relacionado ao tipo do argumento**;
- Por exemplo: um item de um array, pode ser string, boolean ou number;
- Normalmente são utilizadas **letras como T ou U** para definir os generics, é uma convenção;
- Vamos ver na prática!

Constraints nas Generic Functions

- As Generic Functions podem ter seu **escopo reduzido por constraints**;
- Basicamente **limitamos os tipos que podem ser utilizados** no Generic;
- Isso faz com que nosso escopo seja menos abrangente;
- Vamos ver na prática!

Definindo tipo de parâmetros

- Em Generic functions **temos que utilizar parâmetros de tipos semelhantes**, se não recebemos um erro;
- Porém há a possibilidade de **determinar o tipo também dos parâmetros aceitos**, com uma sintaxe especial;
- Isso faz com que a validação do TS aceite os tipos escolhidos;
- Vamos ver na prática!

Parâmetros opcionais

- Nem sempre utilizamos **todos os parâmetros** de uma função;
- Mas se ele for opcional, precisamos **declarar isso para o TS**;
- E ainda deixar ele no **fim da lista** de parâmetros;
- Vamos ver na prática!

Parâmetros default

- Os **parâmetros default** são os que já possuem um valor definido;
- Se não passarmos para a função, é utilizado esse valor;
- Para este recurso, **a aplicação em TS é igual JS**;
- Vamos ver na prática!

O tipo unknown

- O **tipo unknown** é utilizado de forma semelhante ao any, ele aceita qualquer tipo de dado;
- Porém a diferença é que ele **não deixa algo ser executado** se não houver validação de tipo;
- Ou seja, adiciona uma **trava de segurança**;
- Vamos ver na prática!

O tipo never

- O **never** é um tipo de retorno semelhante ao void;
- Porém é utilizado quando a função **não retorna nada**;
- Um exemplo: retorno de erros;
- Vamos ver na prática!

Rest parameters

- Em JavaScript ES6 podemos utilizar o **Rest Operator**;
- Para aplicá-lo em parâmetros em TS é fácil, basta **definir o tipo de dado com a sintaxe de Rest** (...);
- Vamos ver na prática!

Destructuring em parâmetros

- O **Destructuring**, outro recurso de ES6, também pode ser aplicado com TS;
- Precisamos apenas **determinar o tipo de cada dado que será desestruturado**;
- Desta maneira o TS valida o Destructuring;
- Vamos ver na prática!

Aprofundando em funções

Conclusão da seção

Object Types

Introdução da seção

O que são Object Types?

- São os dados que tem como o tipo objeto, por exemplo: **object literals** e **arrays**;
- Temos **diversos recursos** para explorar sobre estes tipos;
- Como: **Interfaces**, **readonly**, **tupla** e outros;
- Isso nos dá possibilidades a mais para o JavaScript;
- Nesta seção focaremos nestes detalhes que são **super importantes para o TypeScript;**

De tipo para Interface

- Um caso de uso para interfaces é **simplificar os parâmetros de objeto** de funções;
- Ou seja, em vez de passar parâmetros de um objeto longo para n funções, **passamos apenas a interface**;
- Vamos ver na prática!

Propriedades opcionais em interfaces

- As interfaces podem conter **propriedades de objeto opcionais**;
- Basta adicionar a **interrogação** no nome da propriedade;
- Exemplo: **nome?: string**
- Vamos ver na prática!

Propriedades readonly

- As propriedades **readonly** podem ser alteradas apenas uma vez, na criação do novo dado;
- É uma forma de criar um **valor constante** em um objeto;
- Podemos adicionar as **interfaces**;
- Vamos ver na prática!

Index Signature

- Utilizamos o **Index Signature** quando não sabemos o nome das chaves, **mas já sabemos quais os tipos de um objeto ou array**;
- Isso **restringe** a tipos que não devem ser utilizados;
- Uma barreira de segurança a mais na nossa variável;
- Vamos ver na prática!

Extending Types

- Utilizamos **Extending Types** como uma herança para criar tipos mais complexos por meio de uma interface;
- Ou seja, uma interface pode **herdar as propriedades e tipos já definidos** de outra;
- Isso acontece por meio da instrução **extends**;
- Vamos ver na prática!

Intersection Types

- **Intersection Types** são utilizados para criar tipos mais complexos a partir de duas interfaces, por exemplo;
- Podemos concatenar os tipos com **&**;
- Vamos ver na prática!

Extending x Intersection

- Qual devemos utilizar?
- Os dois recursos **chegam no mesmo ponto**, implementando tipos mais complexos;
- Ou seja, **vai da sua escolha** optar por um deles;
- Minha opinião: o Extending tem uma **sintaxe parecida com OOP**;
- Além disso a sintaxe é mais enxuta, não precisamos **criar um Type**, por exemplo;

ReadOnlyArray

- O **ReadOnlyArray** é um tipo para arrays, que deixa os itens como readonly;
- Podemos **aplicar um tipo para os itens do array**, além desta propriedade especial;
- **A modificação de itens pode ser feita através de método**, mas não podemos aumentar o array, por exemplo;
- Vamos ver na prática!

Tuplas

- **Tupla é um tipo de array**, porém definimos a quantidade e o tipo de elementos;
- Basicamente **criamos um novo type**, e nele **inserimos um array com os tipos necessários**;
- Cada tipo conta também como um elemento;
- Vamos ver na prática!

Tuplas com readonly

- Podemos criar **tuplas com a propriedade de readonly**;
- É um tipo de dado **super restrito** pois:
- Limita quantos itens teremos, qual o tipo de cada um e também não são modificáveis;
- Vamos ver na prática!

Object Types

Conclusão da seção

Criação de Tipos

Introdução da seção

Sobre a criação de novos tipos

- Há a possibilidade de **gerar novos tipos em TypeScript**, já vimos isso com **Generics** (vamos nos aprofundar neste recurso também);
- Porém existem outros operadores que visam facilitar nossa vida neste assunto;
- A ideia deste recurso é **deixar a manutenção do projeto mais simples**;
- Ou seja, gastar mais tempo na estruturação dos tipos e menos na busca de possíveis bugs depois;

Generics

- Utilizamos **Generics** quando uma função **pode aceitar mais de um tipo**;
- É uma prática **mais interessante do que o any**, que teria um efeito semelhante;
- Porém com Generics
- Vamos ver na prática!

Constraint em Generics

- As constraints nos ajudam a **limitar os tipos aceitos**;
- **Como em Generic podemos ter tipos livres**, elas vão filtrar os tipos aceitos;
- Adicionando organização quando queremos aproveitar a liberdade dos Generics;
- Vamos ver na prática!

Interfaces com Generics

- Com **Interfaces** podemos criar tipos complexos para objetos;
- **Adicionando Generics** podemos deixá-los mais brandos;
- Aceitando tipos diferentes em ocasiões diferentes;
- Vamos ver na prática!

Type parameters

- **Type parameters** é um recurso de Generics;
- Utilizado para dizer que **algum parâmetro de uma função**, por exemplo, **é a chave de um objeto**, que também é parâmetro;
- Desta maneira conseguimos criar uma **ligação entre o tipo genérico e sua chave**
- Vamos ver na prática!

keyof Type Operator

- Com o **keyof Type Operator** podemos criar um novo tipo;
- Ele aceita **dados do tipo objeto**, como object literals e arrays;
- E pode criar o tipo baseado nas **chaves do objeto** passado como parâmetro;
- Vamos ver na prática!

typeof Type Operator

- Com o **typeof Type Operator** podemos criar um novo tipo;
- Este tipo é será **baseado no tipo de algum dado**;
- Ou seja, é interessante para quando queremos criar uma variável com o mesmo tipo da outra, por exemplo;
- Vamos ver na prática!

Indexed Access types

- A abordagem **Indexed Access types** pode criar um tipo baseado em uma chave de objeto;
- Ou seja, conseguimos reaproveitar o tipo da chave para outros locais, como funções;
- Vamos ver na prática!

Conditional Expressions Type

- O **tipo por condição** permite criar um novo tipo com base em um if/else;
- Mas não são aceitas expressões tão amplas;
- Utilizamos a sintaxe de **if ternário**;
- Vamos ver na prática!

Template Literals Type

- Podemos criar tipos com **Template literals**;
- É uma forma de customizar tipos **de maneiras infinitas**;
- Pois o texto que formamos pode depender de variáveis;
- Vamos ver na prática!

Criação de Tipos

Conclusão da seção

Classes

Introdução da seção

Campos em classes

- Em TS podemos **adicionar novos campos a uma classe**, ou seja, iniciar a classe com campos para os futuros dados dos objetos;
- Que serão as **propriedades** dos objetos instanciados;
- Estes campos podem ser **tipados** também;
- Note que um campo sem valor inicial, deve ser declarado com **!**;
- Vamos ver na prática!

Constructor

- **Constructor** é uma função que nos dá a possibilidade de iniciar um objeto com valores nos seus campos;
- Isso faz com que **não precisemos mais da !**;
- Provavelmente **todos os campos serão preenchidos** na hora de instanciar um objeto;
- Vamos ver na prática!

Campos readonly

- Podemos iniciar o campo com valor e torná-lo **readonly**;
- Ou seja, será um **valor só para consulta**;
- Não poderemos alterar este valor ao longo do programa;
- Vamos ver na prática!

Herança e super

- Para gerar uma herança utilizamos a palavra reservada **extends**;
- Depois vamos **precisar passar as propriedades da classe pai para ela**, quando instanciamos um objeto;
- Isso será feito com a **função super**;
- Vamos ver na prática!

Métodos

- **Métodos** são como funções da classe;
- **Podemos criá-los junto das classes** e os objetos podem utilizá-los;
- É uma maneira de **adicionar funcionalidades** as classes;
- Vamos ver na prática!

O this

- A palavra reservada this serve para **nos referirmos ao objeto em si**;
- Ou seja, conseguimos **acessar as suas propriedades**;
- Esta funcionalidade funciona da mesma forma que em JavaScript;
- Vamos ver na prática!

Utilizando getters

- Os **getters** são uma forma de retornar propriedades do objeto;
- Porém **podemos modificá-las neste retorno**, isso é muito interessante;
- Ou seja, é um método para ler propriedades;
- Vamos ver na prática!

Utilizando setters

- Os getters leem propriedades, os **setters** as modificam/atribuem;
- Logo, **podemos fazer validações antes de inserir** uma propriedade;
- Os setters também **funcionam como métodos**;
- Vamos ver na prática!

Herança de interfaces

- Podemos herdar de interfaces também com a instrução **implements**;
- A ideia é bem **parecida de extends**;
- Porém esta forma é mais utilizada para **criar os métodos que várias classes terão em comum**;
- Vamos ver na prática!

Override de métodos

- O **override** é uma técnica utilizada para substituir um método de uma classe que herdamos algo;
- Basta **criar o método com o mesmo nome** na classe filha;
- Isso caracteriza o override;
- Vamos ver na prática!

Visibilidade

- Visibilidade é o conceito de expor nossos métodos de classes;
- **public**: visibilidade default, pode ser acessado em qualquer lugar;
- **protected**: acessível apenas a subclasses da classe do método, para acessar uma propriedade precisamos de um método;
- **private**: apenas a classe que declarou o método pode utilizar;
- Veremos exemplos de todos eles a seguir!

Visibilidade: public

- O **public** é o modo de visibilidade default;
- Ou seja, **já está implícito** e não precisamos declarar;
- Basicamente **qualquer método ou propriedade** de classe pai, estará acessível na classe filha;
- Vamos ver na prática!

Visibilidade: protected

- Os itens **protected** podem ser utilizados apenas em subclasses;
- Uma propriedade **só pode ser acessada por um método**, por exemplo;
- O mesmo acontece com métodos;
- Adicionando uma camada de segurança ao código criado em uma classe;
- Vamos ver na prática!

Visibilidade: private

- Os itens **private**, propriedades e objetos, só podem ser acessados na classe que os definiu;
- E ainda **precisam de métodos** para serem acessados;
- Esta é a **maior proteção** para propriedades e métodos;
- Vamos ver na prática!

Static members

- Podemos criar propriedades e métodos **estáticos** em classes;
- Isso faz com que o acesso ou a utilização **não dependam de objetos**;
- Podemos utilizá-los a partir **da própria classe**;
- Vamos ver na prática!

Generic class

- Podemos criar classes com **tipos genéricos** também;
- Ou seja, as propriedades dos argumentos podem ter os tipos definidos **na hora da criação da instância**;
- Isso nos permite **maior flexibilidade** em uma classe;
- Vamos ver na prática!

Parameters properties

- **Parameters properties** é um recurso para definir a privacidade, nome e tipo das propriedades no constructor;
- Isso **resume um pouco a sintaxe** das nossas classes;
- Vamos ver na prática!

Class Expressions

- **Class Expressions** é um recurso para criar uma classe anônima;
- Podemos também utilizar **generics** junto deste recurso;
- Vamos encapsular a classe em uma **variável**;
- Vamos ver na prática!

Abstract class

- **Abstract Class** é um recurso para servir como molde de outra classe;
- **Todos os métodos dela devem ser implementados** nas classes que a herdam;
- E também **não podemos instanciar objetos** a partir destas classes;
- Vamos ver na prática!

Relações entre classes

- Podemos criar um **objeto com base em outra classe**;
- **Quando as classes são idênticas** o TS não reclama sobre esta ação;
- Precisamos que as duas sejam exatamente iguais;
- Vamos ver na prática!

Classes

Conclusão da seção

Trabalhando com Módulos

Introdução da seção

Introdução aos módulos

- Os módulos são a forma que temos para importar código em arquivos;
- Podemos exportar código com **export default**;
- E importar com **import**;
- **Criaremos os arquivos com .ts**, mas importaremos como .js;
- Isso nos dá mais flexibilidade, podendo separar as responsabilidades em arquivos;
- Utilizaremos o **Node.js** para executar os arquivos com módulos;

Importando arquivos

- Para começar **vamos criar um arquivo simples e importar seu conteúdo**;
- Basta criar um **arquivo .ts**, desenvolver o código e exportar;
- Depois no arquivo principal vamos importar o arquivo anterior, com a **extensão .js**;
- Vamos ver na prática!

Importando variáveis

- Podemos **exportar e importar variáveis** também;
- A sintaxe é um pouco mais simples, vamos utilizar **apenas o export**;
- No arquivo de importação vamos importar os valores com **destructuring**;
- Vamos ver na prática!

Múltiplas importações

- Podemos **exportar múltiplas variáveis e funções**;
- Isso pode ser realizado no **mesmo arquivo**;
- Para esta modalidade utilizamos **export para todos os dados**;
- E todos devem ser importados com destructuring;
- Vamos ver na prática!

Alias para importações

- Podemos criar um **alias** para importações;
- Ou seja, **mudar o nome** do que foi importado;
- Podendo tornar este novo nome, uma **forma mais simples de chamar o recurso**;
- Vamos ver na prática!

Importando tudo

- Podemos **importar tudo que está em um arquivo** com apenas um símbolo;
- Utilizamos o * para isso;
- Os dados virão em um **objeto**;
- Vamos ver na prática!

Importando tipos

- Importar **tipos ou interfaces** também é possível;
- Vamos exportar como se fossem **variáveis**;
- E no arquivo que os recebe, utilizamos **destructuring**;
- Depois podemos implementar no projeto;
- Vamos ver na prática!

Trabalhando com Módulos

Conclusão da seção

Decorators

Introdução da seção

O que são os decorators?

- Decorators podem **adicionar funcionalidades extras** a classes e funções;
- Basicamente criamos novas funções, que são adicionadas a partir de um **@nome**;
- Esta função será chamada assim que o item que foi definido o decorator **for executado**;
- Para habilitar precisamos adicionar uma configuração no **tsconfig.json**;

Primeiro decorator

- Vamos criar um decorator como uma **function**;
- Ele pode trabalhar com argumentos especiais que são: **target**, **propertyKey** e **descriptor**;
- Estes são os **grandes trunfos** do decorator, pois nos dão informação do local em que ele foi executado;
- Vamos ver na prática!

Múltiplos decorators

- Podemos utilizar **múltiplos decorators** em TS;
- O primeiro a ser executado é o que está **mais ao topo do código**;
- Desta maneira é possível criar operações mais complexas;
- Vamos ver na prática!

Decorator de classe

- O decorator de classe está ligado ao **constructor**;
- Ou seja, sempre que este for executado, **teremos a execução do decorator**;
- Isso nos permite acrescentar algo a criação de classes;
- Vamos ver na prática!

Decorator de método

- Com este decorator podemos **modificar a execução de métodos**;
- Precisamos inserir o decorator **antes da declaração do método**;
- Ele é executado antes do método;
- Vamos ver na prática!

Accessor decorator

- Semelhante ao decorator de método;
- Porém este serve apenas para os **getters e setters**;
- Podemos alterar a execução antes de um set ou get;
- Vamos ver na prática!

Property decorator

- O **property decorator** é utilizado nas propriedades de uma classe;
- Ou seja, na hora da definição da mesma podemos **ativar uma função**;
- Isso nos ajuda a modificar ou validar algum valor;
- Vamos ver na prática!

Exemplo real: Class Decorator

- Com **Class Decorator** podemos influenciar o constructor;
- Neste exemplo vamos criar uma função para inserir **data de criação dos objetos**;
- Vamos ver na prática!

Exemplo real: Method Decorator

- Com **Method Decorator** podemos alterar a execução dos métodos;
- Neste exemplo vamos **verificar se um usuário pode ou não fazer uma alteração** no sistema;
- A alteração seria o método a ser executado;
- Vamos ver na prática!

Exemplo real: Property Decorator

- Com o **Property Decorator** conseguimos verificar uma propriedade de um objeto;
- Vamos criar uma **validação de número máximo de caracteres** com decorators;
- Vamos ver na prática!

Decorators

Conclusão da seção

TS com React

Introdução da seção

React com TS

- Adicionar **TypeScript ao React** nos dá mais possibilidades;
- Seguindo a mesma linha de que em JS, **temos uma forma mais padronizada** para programar;
- Como **tipos para componentes** ou mapeamento de **props por meio de interface**;
- Isso dá mais **confiabilidade** ao projeto e está sendo cada vez mais utilizado hoje em dia;

Instalando React com TS

- Para instalar o **TS junto do React** é simples;
- Vamos começar com **create-react-app** e adicionar a flag **–template** com o valor de typescript;
- **Um novo projeto é criado**, agora com arquivos **.tsx**;
- Podemos inicializá-lo normalmente;
- Vamos ver na prática!

Estrutura de React com TS

- A estrutura de React quando adicionamos TS **não muda muito**;
- Temos as pastas clássicas como: **node_modules**, **src** e **public**;
- Em src que as coisas ficam diferentes, temos a criação de **arquivos .tsx**;
- **Que são arquivos jsx** porém com a possibilidade de aplicação das funcionalidades de TS;
- Podemos executar o projeto com **npm run start**;
- Vamos ver estes arquivos!

Criação de variáveis em componentes

- Podemos **criar variáveis** dentro dos componentes;
- E elas podem receber os tipos que já vimos até este momento do curso;
- Isso nos permite **trabalhar com JSX** com apoio destas variáveis e seus tipos;
- Vamos ver na prática!

Criação de funções em componentes

- Podemos também criar **funções em componentes**;
- Estas funções recebem **parâmetros**, que **podem ser tipados**;
- E o seu retorno também;
- Ou seja, podemos aplicar os mesmos conceitos que já vimos de TS;
- Vamos ver na prática!

Criação de novos componentes

- Geralmente criamos os componentes em uma pasta chamada **components**;
- O arquivo deve ser **.tsx**;
- E um retorno comum utilizado é o **JSX.Element**;
- O resto fica semelhante ao React sem TS;
- Vamos ver na prática!

Extensão para React com TS

- A extensão que vamos utilizar é a **TypeScript React code snippets**;
- Ela nos ajuda com **atalhos** para programar mais rápido;
- Como o **tsrafce**, que cria um componente funcional;
- Isso torna o nosso dia a dia mais simples;
- Vamos ver na prática!

Importando componentes

- A importação de componente **funciona da mesma forma que sem TypeScript**;
- Porém temos que nos atentar aos **valores e tipos das props** de cada componente;
- O TS interage de forma mais sucinta na parte da importação;
- Vamos ver na prática!

Destructuring nas props

- O **destructuring** é um recurso de ES6, que **permite separar um array ou objeto** em várias variáveis;
- Aplicamos esta técnica nas **props**, para não precisa repetir o nome do objeto sempre;
- Podemos fazer desta maneira em TS também;
- Vamos ver na prática!

O hook useState

- o **useState** é um hook muito utilizado em React;
- Serve para **consultar e alterar o estado** de algum dado;
- **Atrelamos uma função set a um evento**, como mudança de dado em input e alteramos o valor da variável base;
- Podemos adaptar este recurso para TS também;
- Vamos ver na prática!

Enum

- O **Enum** é uma forma interessante de formatar um objeto com chaves e valores;
- Onde podemos **utilizar como props**;
- Passando a chave pela prop, imprimimos o valor dela no componente;
- Vamos ver na prática!

Types

- Além das interfaces, podemos criar estruturas de tipos com o **type**;
- Isso nos permite criar dados com **tipos de dados fixos**;
- Ou até tipos customizados, como quando utilizamos o **operador |**
- Vamos ver na prática!

Context API

- A **Context API**, é uma forma de transmitir dados entre componentes no React;
- A ideia principal é que podemos **determinar quais componentes recebem estes dados**;
- Ou seja, fazem parte do **contexto**;
- Podemos aplicar TS a esta funcionalidade também;
- Vamos ver na prática!

Utilizando o dado de contexto

- Para utilizar os dados do contexto vamos precisar de um **hook**;
- Que é o **useContext**;
- A partir dele conseguimos **extrair os dados** e utilizar em um componente;
- Vamos ver na prática!

TS com React

Conclusão da seção

TS com Express

Introdução da seção

Inicialização

- Para iniciar um projeto com Express e TypeScript precisamos criar o projeto com **npm init**;
- E também iniciar o TS com **npx tsc –init**;
- Após estes dois passos **vamos instalar as dependências**, algumas são de de dev (como os tipos) e outras não (como o Express);
- E por fim criamos um **script** e iniciamos a aplicação!

Utilizando o Express

- Para utilizar o express vamos **importar o pacote**;
- E criar ativá-lo em uma nova variável, geralmente chamada de **app**;
- Podemos **criar uma rota** que retorna uma mensagem;
- Definir uma **porta** para a aplicação;
- E verificar o resultado no navegador;

Roteamento

- Podemos utilizar **qualquer verbo HTTP nas rotas** do Express;
- Vamos criar uma que funciona a base de **POST**;
- Para isso precisamos trafegar dados em **JSON**, podemos fazer isso ativando um **middleware**;
- Iremos realizar os testes com o **Postman**;

Rota para qualquer verbo

- Utilizando o método **all**, podemos criar uma rota que aceita qualquer verbo;
- É interessante para quando um endpoint **precisa realizar várias funções**;
- Podemos criar um tratamento para entregar a resposta;
- Vamos ver na prática!

Interfaces do Express

- Para alinhar nossa aplicação ao **TypeScript** vamos adicionar novos tipos;
- As request podem utilizar o tipo **Request**;
- E as respostas o **Response**;
- Vamos ver na prática!

JSON como respostas

- Na maioria das vezes enviamos **JSON para endpoints de API**;
- Para fazer isso com Express é facil!
- Basta enviar o JSON no **método json** de response;
- Vamos ver na prática!

Router parameters

- Podemos pegar parâmetros de rotas com Express;
- Vamos utilizar **req.params**;
- A rota a ser criada precisa ser **dinâmica**;
- Ou seja, os parâmetros que estamos querendo receber precisam estar no padrão: **:id**;

Rotas mais complexas

- Podemos ter rotas com **mais de um parâmetro**;
- Todos os dados continuam em **req.params**;
- O padrão é: /algo/**:param1**/outracoisa/**:param2**
- Teremos então: param1 e param2 em req;
- Vamos ver na prática!

Router handler

- **Router handler** é um recurso muito interessante para o Express;
- Basicamente retiramos a função anônima de uma rota e **externalizamos em uma função**;
- Podemos reaproveitar essa função, ou estrutura nossa aplicação desta maneira;
- Vamos ver na prática!

Middleware

- **Middleware** é um recurso para executar uma função entre a execução de uma rota, por exemplo;
- O nosso **app.use de json** é um middleware;
- Podemos utilizar middleware para **validações**, por exemplo;
- Vamos ver na prática!

Middleware para todas as rotas

- Para criar um middleware que é executado em todas as rotas vamos utilizar o método **use**;
- **Criamos uma função** e atrelamos ao método;
- Desta maneira **todas as rotas** terão ação do nosso middleware;
- Vamos ver na prática!

Request e Response generics

- Podemos estabelecer os **argumentos que vem pelo request e vão pela response**;
- Para isso vamos utilizar os **Generics** de Response e Request;
- Que são as **Interfaces** disponibilizadas pelo Express;
- Vamos ver na prática!

Tratando erros

- Para tratar possíveis erros utilizamos **blocos try catch**;
- Desta maneira **podemos detectar algum problema** e retornar uma resposta para o usuário;
- Ou até mesmo **criar um log** no sistema;
- Vamos ver na prática!

TS com Express

Conclusão da seção