Desenvolvimento para Frontend

Introdução à TypeScript

Universidade Federal do Rio Grande do Norte

Funções

- Em TypeScript, podemos definir tipos tanto para os parâmetros quanto para o retorno das funções.
- O tipo de retorno é especificado após os parâmetros da função. TypeScript infere o tipo de retorno se não for explicitamente declarado.

```
function add(x: number, y: number): number {
    return x + y;
}

let myAdd = function(x: number, y: number): number { return x + y; };

let myAddTyped: (x: number, y: number) => number =
    function(x: number, y: number): number { return x + y; };
```

Funções - Parâmetros Opcionais e Padrão

- Parâmetros opcionais e padrão tornam funções mais flexíveis e evitam a necessidade de sobrecarregar funções para diferentes casos de uso.
- Parâmetros opcionais indicam que um parâmetro não é obrigatório. São definidos com ?.

```
function buildName(firstName: string, lastName?: string): string {
    return firstName + " " + (lastName || "");
}
let result1 = buildName("Bob");  // Ok
let result2 = buildName("Bob", "Adams");  // Ok
```

Funções - Parâmetros Opcionais e Padrão

- São parâmetros com um valor padrão caso não sejam fornecidos.

```
function buildName(firstName: string, lastName = "Smith"): string {
    return firstName + " " + lastName;
}
let result1 = buildName("Bob");  // "Bob Smith"
let result2 = buildName("Bob", undefined);  // "Bob Smith"
let result3 = buildName("Bob", "Adams");  // "Bob Adams"
```

Funções - Rest Parameters e Overloads

- Rest Parameters permitem passar um número variável de argumentos como um array. Definidos com “...”.

```
function buildName(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}
let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

Funções - Rest Parameters e Overloads

- **Overloads**: TypeScript permite definir múltiplas assinaturas de função para lidar com diferentes tipos de argumentos.

```
function pickCard(x: {suit: string; card: number; }[]): number;
function pickCard(x: number): {suit: string; card: number; };
function pickCard(x): any {
    if (typeof x == "object") {
        let pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    else if (typeof x == "number") {
        let suits = ["hearts", "spades", "clubs", "diamonds"];
        let pickedSuit = Math.floor(x / 13);
        return {suit: suits[pickedSuit], card: x % 13};
    }
}

let myDeck = [{suit: "diamonds", card: 2}, {suit: "spades", card: 10}, {suit: "hearts", card: 4}];
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);
let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
```

Funções - Promises

- Uma "promise" **refere-se à expectativa de que algo acontecerá em um determinado momento**, permitindo que sua aplicação use o resultado desse evento futuro para executar certas outras tarefas.
- As palavras-chave async/await são um invólucro para promises, fazendo com que o código assíncrono pareça e se comporte como código síncrono, tornando-o mais fácil de entender.
- Uma função async sempre retorna uma promise.
- Mesmo se você omitir a palavra-chave Promise, o compilador envolverá sua função em uma promise resolvida imediatamente.

Funções - Promises

- A palavra-chave async antes de uma declaração de função para indicar que ela contém código assíncrono.
- Dentro de uma função assíncrona, a palavra-chave await deve ser usada antes de invocar uma promise. Isso pausa a execução da função até que a promise seja resolvida ou rejeitada.

```
async function myAsyncFunction(): Promise<void> {
  // Código com a palavra await
  try {
    const result = await myPromiseFunction();
    // Process result
  } catch (error) {
    // Handle errors
  }
}
```

Funções - Promises

- O método then() das instâncias de Promise aceita até dois argumentos: funções de callback para os casos de **sucesso** e de **rejeição** da Promise.
- O método then() retorna imediatamente um objeto Promise equivalente, permitindo encadear chamadas para outros métodos de promise.

```
then(onFulfilled)
then(onFulfilled, onRejected)
```

```
const promise1 = new Promise((resolve, reject) => {
  resolve('Success!');
});

promise1.then((value) => {
  console.log(value);
  // Expected output: "Success!"
});
```

Generics

- Generics permitem criar componentes que funcionam com uma variedade de tipos em vez de um único tipo específico.
- Aumentam a reutilização de código e a flexibilidade.

```
function identity<T>(arg: T): T {
    return arg;
}

let output1 = identity<string>("myString");  // tipo de T é string
let output2 = identity<number>(100);  // tipo de T é number
```

- A função identity usa um parâmetro de tipo genérico T. O tipo de T é inferido com base no argumento passado na chamada da função

Generics em Funções

- Generics podem ser usados para criar funções flexíveis que trabalham com qualquer tipo de dados.

```
function loggingIdentity<T>(arg: T[]): T[] {
    console.log(arg.length);  // Array tem propriedade 'length'
    return arg;
}

let stringArray = loggingIdentity<string>(["a", "b", "c"]);
let numberArray = loggingIdentity<number>([1, 2, 3]);
```

- A função loggingIdentity aceita um array de qualquer tipo T, logando o tamanho do array e retornando-o.

Generics em Classes

- Classes também podem usar parâmetros de tipo genérico para serem mais flexíveis.

```
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };

let stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function(x, y) { return x + y; };
```

- GenericNumber é uma classe que pode operar com qualquer tipo T, tornando-se reutilizável para diferentes tipos de dado.

Generics em Interfaces

- Interfaces podem ser genéricas para definir contratos flexíveis.

```
interface GenericIdentityFn<T> {
    (arg: T): T;
}

function identity<T>(arg: T): T {
    return arg;
}

let myIdentity: GenericIdentityFn<number> = identity;
```

- A interface GenericIdentityFn define uma função genérica que aceita e retorna um tipo T.

Atividade 1

- Implemente uma função genérica que aceita um array de qualquer tipo e retorna o primeiro elemento desse array.

Atividade 1

- Implemente uma função genérica que aceita um array de qualquer tipo e retorna o primeiro elemento desse array.

```
function pegarPrimeiroElemento<T>(arr: T[]): T | undefined {
    return arr[0];
}

const numeros = [1, 2, 3];
const palavras = ['a', 'b', 'c'];

console.log(pegarPrimeiroElemento(numeros)); // Saída: 1
console.log(pegarPrimeiroElemento(palavras)); // Saída: a
```

Atividade 2

- Crie uma função que aceite um parâmetro que pode ser uma string ou um número. Se for uma string, a função deve retornar a string em maiúsculas. Se for um número, deve retornar o número ao quadrado.

Atividade 2

- Crie uma função que aceite um parâmetro que pode ser uma string ou um número. Se for uma string, a função deve retornar a string em maiúsculas. Se for um número, deve retornar o número ao quadrado.

```
function processarValor(valor: string | number): string | number {
    if (typeof valor === 'string') {
        return valor.toUpperCase();
    }
    return valor * valor;
}

console.log(processarValor('hello')); // Saída: HELLO
console.log(processarValor(4)); // Saída: 16
```

Atividade 3

- Crie uma função assíncrona que simule uma operação de busca de dados com setTimeout e retorne um objeto de usuário. Use await para esperar a resposta e imprima os dados do usuário.

Atividade 3

- Crie uma função assíncrona que simule uma operação de busca de dados com setTimeout e retorne um objeto de usuário. Use await para esperar a resposta e imprima os dados do usuário.

```
function buscarUsuario(): Promise<{ id: number, nome: string }> {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve({ id: 1, nome: 'Maria' });
        }, 2000);
    });
}

async function mostrarUsuario(): Promise<void> {
    const usuario = await buscarUsuario();
    console.log(`ID: ${usuario.id}`);
    console.log(`Nome: ${usuario.nome}`);
}

mostrarUsuario();
// Saída (após 2 segundos):
// ID: 1
// Nome: Maria
```

Referências

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/
- https://www.typescriptlang.org/docs/
- https://www.geeksforgeeks.org/