Tipos

TypeScript

A linguagem de programação TypeScript é um superconjunto de JavaScript que adiciona tipos ao JavaScript usando um conjunto de ferramentas chamado sistema de tipos.

Tipos primitivos

Todos os tipos de dados "primitivos" ou integrados em JavaScript são reconhecidos pelo TypeScript.

```
"Hello, world!"  // string
42               // number
true             // boolean
null
undefined
```

Inferência de tipo TypeScript

A inferência de tipo assume o tipo esperado da variável em todo o programa com base no tipo de dados do valor inicialmente atribuído a ela na declaração. A inferência de tipo registrará uma reclamação se a variável for posteriormente reatribuída a um tipo diferente.

```
let first = 'Anders';

first = 1337; // Type 'number' is not assignable to type 'string'
```

A forma de um objeto

O TypeScript conhece a *forma* de um objeto — quais propriedades de membro ele contém ou não. O TypeScript registrará um erro se o código tentar acessar membros de um objeto conhecido como inexistente. Pode até sugerir possíveis correções.

```
let firstName = 'muriel!';

console.log(firstName.toUppercase());

// error: Property 'toUppercase' does not exist on type
'string'. Did you mean 'toUpperCase'?
```

Datilografe qualquer

Quando uma variável é declarada sem receber um valor inicial, o TypeScript a considera do tipo `any`. Uma variável desse tipo pode ser reatribuída sem gerar nenhum erro do TypeScript.

```
let first;
first = 'Anders';
first = 1337;
```

TypeScript suporta anotações de tipo

Adding a type annotation ensures that a variable can only ever be assigned to that type. This can be useful when declaring a variable without an initial value. Type annotations get removed when compiled to JavaScript. The type declaration is provided by appending a variable with a colon (`:`) and the type (eg. `number`).

```
let first: string;
first = 'Anders';

// Error: Type 'number' is not assignable to type 'string'
first = 1337;
```

Tipos complexos

Tipo TypeScript para array unidimensional

A anotação de tipo para uma matriz unidimensional no TypeScript é semelhante a um tipo de dados primitivo, exceto que adicionamos um `[]` após o tipo.

```
// zipcodes is an array of strings
let zipcodes: string[] = ['03255', '02134', '08002', '03063'];

// Pushing a number to zipcodes will generate an error
// Error: Argument of type 'number' is not assignable to
parameter of type 'string'.
zipcodes.push(90210);
```

Tipo genérico TypeScript para array unidimensional

O tipo para uma matriz unidimensional no TypeScript pode ser anotado com `Array<T>`, onde `T` representa o tipo.

```
// zipcodes is an array of strings
let zipcodes: Array<string> = ['03255', '02134', '08002',
'03063'];

// Pushing a number to zipcodes will generate an error
// Error: Argument of type 'number' is not assignable to
parameter of type 'string'.
zipcodes.push(90210);
```

Tipo TypeScript para array multidimensional

O tipo para uma matriz multidimensional pode ser anotado adicionando um extra `[]` para cada dimensão extra da matriz.

```
// one-dimensional arrays
let zipcodesNH: string[] = ['03255', '03050', '03087', '03063'];
let zipcodesMA: string[] = ['02334', '01801'];

// two-dimensional array
let zipcodes: string[][] = [zipcodesNH];

// Pushing a one-dimensional array to a two-dimensional array
zipcodes.push(zipcodesMA);
console.log(zipcodes); // prints [["03255", "03050", "03087",
"03063"], ["02334", "01801"]]
```

Inicialização de matriz vazia do TypeScript

Um array de qualquer dimensão pode ser inicializado como um array vazio sem gerar nenhum erro.

```
// one-dimensional empty array
let axis: string[] = [];

// two-dimensional empty array
let coordinates: number[][] = [];

axis.push('x');
console.log(axis);        // prints ["x"]

coordinates.push([3, 5]);
coordinates.push([7]);
console.log(coordinates); // prints [[3, 5], [7]]
```

Tipo de Tupla TypeScript

Uma matriz que possui um tamanho fixo de tipos de elementos semelhantes ou diferentes organizados em uma sequência específica é definida como uma tupla no TypeScript.

```
// This is an array
let header: string[] = ['Name', 'Age', 'Smoking', 'Salary'];
// This is a tuple
let profile: [string, number, boolean, number] = ['Kobe', 39,
true, 150000];
```

Sintaxe do tipo de tupla TypeScript

Para anotar uma tupla no TypeScript, adicione dois pontos (`:`) seguido por colchetes (`[...]`) contendo uma lista de tipos separados por vírgula.

```
// This is a tuple
let profile: [string, number, boolean, number] = ['Kobe', 39,
true, 150000];

profile[2] = 'false';   // Error: Type 'string' is not
assignable to type 'boolean'.
profile[3] = null;      // Error: Type 'null' is not assignable
to type 'number'.
```

Tamanho e ordem do tipo de tupla TypeScript

A tuple in Typescript is declared with a fixed number of elements and hence, cannot be assigned to a tuple with a different number of elements. Similarly, a tuple maintains a strict ordering of its elements and therefore, the type for each element is enforced. A transcompiler error will be generated if any of these conditions is violated.

```
let employee: [string, number] = ['Manager', null];
// Error: Type 'null' is not assignable to type 'number'.

let grade: [string, number, boolean] = [ 'TypeScript', 85, true,
'beginner'];
/*
Error: Type '[string, number, true, string]'
is not assignable to type '[string, number, boolean]'.
Source has 4 element(s) but target allows only 3.
*/
```

TypeScript Tuple Array Assignment

Although a tuple may have all elements of the same type and resembles an array, a tuple is still its own type. A tuple cannot expand, while an array can. Hence, assigning an array to a tuple that matches the same type and length will generate an error.

```
// This is a tuple
let eventDate: [string, string] = ['January', '2'];

// This is an array
let newDate: string[] = ['January', '12'];

eventDate = newDate;
/*
Error: Type 'string[]' is not assignable to type '[string,
string]'.
Target requires 2 element(s) but source may have fewer.
*/
```

TypeScript Array Type Inference

When an array variable is declared without an explicit type annotation, TypeScript automatically infers such a variable instance to be an array instead of a tuple.

```
let mixed = ['one', 2, 3, 'four'];
mixed[4] = 5;                   // no error because an array is
expandable
console.log(mixed);             // prints ["one", 2, 3, "four", 5]
```

TypeScript Array Type Inference on Tuple `.concat()`

The JavaScript method, `.concat()` can be called on a TypeScript tuple, and this produces a new array type instead of a tuple.

```
// This is a tuple
const threeWords: [ string, number, string] = ['Won', 5,
'games'];

// Calling .concat() on a tuple returns an array
let moreWords = threeWords.concat(['last', 'night']);

// An array is expandable
moreWords[5] = ('!');

console.log(moreWords);
// This prints ["Won", 5, "games", "last", "night", "!"]
```

TypeScript Function Rest Parameter Any Array Type

A rest parameter inside a function is implicitly assigned an array type of `any[]` by TypeScript.

```
const sumAllNumbers = (...numberList): number => {
  // Error: Rest parameter 'numberList' implicitly has an
'any[]' type.
  let sum = 0;
  for (let i=0; i < numberList.length; i++) {
    sum += numberList[i];
  }
  return sum;
}

// Notice third argument is a string
console.log(sumAllNumbers(100, 70, '30'));
// Prints a string "17030 instead of a number 200
```

TypeScript Function Rest Parameter Explicit Type

Explicitly type annotating a rest parameter of a function will alert TypeScript to check for type inconsistency between the rest parameter and the function call arguments.

```
const sumAllNumbers = (...numberList: number[]): number => {
  let sum = 0;
  for (let i=0; i < numberList.length; i++) {
    sum += numberList[i];
  }
  return sum;
}

console.log(sumAllNumbers(100, 70, '30')); // Error: Argument of type 'string' is not assignable to parameter of type 'number'.
```

TypeScript Tuple Type Spread Syntax

Spread syntax can be used with a tuple as an argument to a function call whose parameter types match those of the tuple elements.

```
function modulo(dividend: number, divisor: number): number {
  return dividend % divisor;
}

const numbers: [number, number] = [6, 4];

// Call modulo() with a tuple
console.log(modulo(numbers));
// Error: Expected 2 arguments, but got 1.
// Prints NaN

// Call modulo() with spread syntax
console.log(modulo(...numbers));
// No error, prints 2
```

TypeScript Enum Type

A programmer can define a set of possible values for a variable using TypeScript's complex type called enum.

```
enum MaritalStatus {
  Single,
  Married,
  Separated,
  Divorced
};

let employee: [string, MaritalStatus, number] = [
  'Bob Jones',
  MaritalStatus.Single,
  39
];
```

TypeScript Numeric and String Enum Types

TypeScript supports two types of enum: numeric enum and string enum. Members of a numeric enum have a corresponding numeric value assigned to them, while members of a string enum must have a corresponding string value assigned to them.

```
// This is a numeric enum type
enum ClassGrade {
  Freshman = 9,
  Sophomore,
  Junior,
  Senior
};

// This is a string enum type
enum ClassName {
  Freshman = 'FRESHMAN',
  Sophomore = 'SOPHOMORE',
  Junior = 'JUNIOR',
  Senior = 'SENIOR'
}

const studentClass: ClassName = ClassName.Junior;
const studentGrade: ClassGrade = ClassGrade.Junior;

console.log(`I am a ${studentClass} in ${studentGrade}th
grade.`);
// Prints "I am a JUNIOR in 11th grade."
```

TypeScript Numeric Enum Type Initializers

By default, TypeScript assigns a value of `0` to the first member of a numeric enum type and auto-increments the value of the rest of the members. However, you can override the default value for any member by assigning specific numeric values to some or all of the members.

```
// This numeric enum type begins with a 1, instead of the
default 0
enum Weekdays {
  Monday = 1,
  Tuesday,
  Wednesday,
  Thursday,
  Friday
}

// This is a numeric enum type with all explicit values
enum Grades {
  A = 90,
  B = 80,
  C = 70,
  D = 60
}

// This numeric enum type has only some explicit values
enum Prizes {
  Pencil = 5,
  Ruler,      // No error: value is 6
  Eraser = 10,
  Pen         // No error: value is 11
};

const day: Weekdays = Weekdays.Wednesday;
const grade:Grades = Grades.B;
const prize:Prizes = Prizes.Pen;
console.log(`On day ${day} of the week, I got ${grade} on my
test! I won a prize with ${prize} points!`);
// Prints "On day 3 of the week, I got 80 on my test! I won
a prize with 11 points!"
```

TypeScript Numeric Enum Variable Assignment

You can assign a valid numeric value to a variable whose type is a numeric enum.

```
enum Weekend {
  Friday = 5,
  Saturday,
  Sunday
};

// Assign a valid value of Weekend
const today: Weekend = 7;       // No error
console.log(`Today is the ${today}th day of the week!`);
// Prints "Today is the 7th day of the week!"
```

TypeScript String Enum Variable Assignment

Unlike a numeric enum type which allows a number to be assigned to its member, a string enum type does not allow a string to be assigned to its member. Doing so will cause a TypeScript error.

```
enum MaritalStatus {
  Single = 'SINGLE',
  Married = 'MARRIED',
  Separated = 'SEPARATED',
  Divorced = 'DIVORCED',
  Widowed = 'WIDOWED'
};

// Assign a string to a string enum type
let eligibility: MaritalStatus;
eligibility = 'SEPARATED';
// Error: Type '"SEPARATED"' is not assignable to type
'MaritalStatus'.

eligibility = MaritalStatus.Separated;  // No error
```

TypeScript Object Type

A JavaScript object literal consists of property-value pairs. To type-annotate an object literal, use the TypeScript object type and specify what properties must be provided and their accompanying value types.

```
// Define an object type for car
let car: {make: string, model: string, year: number};

car = {make: 'Toyota', model: 'Camry', year: 2020}; // No error
car = {make: 'Nissan', mode: 'Sentra', year: 2019};
/*
Error: Type '{make: string; mode: string; year: number;}' is not
assignable to
type '{make: string; model: string; year: number;}'.
Object literal may only specify known properties, but 'mode'
does not exist in
type '{make: string; model: string; year: number;}'.
Did you mean to write 'model'?
*/
car = {make: 'Chevrolet', model: 'Monte Carlo', year: '1995'};
// Error: Type 'string' is not assignable to type 'number'.
```

TypeScript Type Alias

Instead of redeclaring the same complex object type everywhere it is used, TypeScript provides a simple way to reuse this object type. By creating an alias with the `type` keyword, you can assign a data type to it. To create a type alias, follow this syntax:
`type MyString = string;`

```
// This is a type alias
type Student = {
  name: string,
  age: number,
  courses: string[]
};

let boris: Student = {name: 'Boris', age: 35, courses:
['JavaScript', 'TypeScript']};
```

TypeScript Multiple Alias References

You can create multiple type aliases that define the same data type, and use the aliases as assignments to variables.

```
// This is also a type alias with the same type as Student
type Employee = {
  name: string,
  age: number,
  courses: string[]
}

let studentBoris: Student = {name: 'Boris', age: 35, courses:
['JavaScript', 'TypeScript']};
let employeeBoris: Employee = studentBoris;     // No error
console.log(studentBoris === employeeBoris);    // Prints true
```

TypeScript Function Type Alias

In JavaScript, a function can be assigned to a variable. In TypeScript, a function type alias can be used to annotate a variable. Declare a function type alias following this syntax:

```
type NumberArrayToNumber = (numberArray: number[]) => number
```

```
// This is a function type alias
type NumberArrayToNumber = (numberArray: number[]) => number;

// This function uses a function type alias
let sumAll: NumberArrayToNumber = function(numbers: number[]) {
  let sum = 0;
  for (let i=0; i < numbers.length; i++) {
    sum += numbers[i];
  }
  return sum;
}

// This function also uses the same function type alias
let computeAverage: NumberArrayToNumber = function(numbers: 
number[]) {
  return sumAll(numbers)/numbers.length;
};

console.log(computeAverage([5, 10, 15]));   // Prints 10
```

TypeScript Generic Type Alias

In addition to the generic Array type, `Array<T>`, custom user-defined generic types are also supported by TypeScript. To define a generic type alias, use the `type` keyword followed by the alias name and angle brackets `<...>` containing a symbol for the generic type and assign it a custom definition. The symbol can be any alphanumeric character or string.

```
// This is a generic type alias
type Collection<G> = {
  name: string,
  quantity: number,
  content: G[]
};

let bookCollection: Collection<string> = {
  name: 'Nursery Books',
  quantity: 3,
  content: ['Goodnight Moon', 'Humpty Dumpty', 'Green Eggs &
Ham']
};

let primeNumberCollection: Collection<number> = {
  name: 'First 5 Prime Numbers',
  quantity: 5,
  content: [2, 3, 5, 7, 11]
};
```

TypeScript Generic Function Type Alias

With the TypeScript *generic function* type alias, a function can take parameters of generic types and return a generic type. To turn a function into a generic function type alias, add angle brackets, `<...>` containing a generic type symbol after the function name, and use the symbol to annotate the parameter type and return type where applicable.

```
// This is a generic function type alias
function findMiddleMember<M>(members: M[]): M {
  return members[Math.floor(members.length/2)];
}

// Call function for an array of strings
console.log(findMiddleMember<string>(['I', 'am', 'very',
'happy'])); // Prints "very"

// Call function for an array of numbers
console.log(findMiddleMember<number>([210, 369, 102]));     //
Prints 369
```

Tipos de União

Tipo de União TypeScript

O TypeScript permite um tipo flexível chamado any que pode ser atribuído a uma variável cujo tipo não é específico. Por outro lado, o TypeScript permite combinar tipos específicos como um tipo de união.

```
let answer: any;    // any type
let typedAnswer: string | number; // union type
```

Sintaxe do tipo de união TypeScript

O TypeScript permite criar um tipo de união que é uma composição de tipos selecionados separados por uma barra vertical, `|` .

```
let myBoolean: string | boolean;

myBoolean = 'TRUE';  // string type
myBoolean = false;   // boolean type
```

Limitação de tipo de união TypeScript

Como uma variável de um tipo de união pode assumir um dos vários tipos diferentes, você pode ajudar o TypeScript a inferir o tipo de variável correto usando a restrição de tipo. Para restringir uma variável a um tipo específico, implemente um protetor de tipo. Use o `typeof` operador com o nome da variável e compare-o com o tipo que você espera para a variável.

```
const choices: [string, string] = ['NO', 'YES'];
const processAnswer = (answer: number | boolean) => {
  if (typeof answer === 'number') {
    console.log(choices[answer]);
  } else if (typeof answer === 'boolean') {
    if (answer) {
      console.log(choices[1]);
    } else {
      console.log(choices[0]);
    }
  }
}
processAnswer(true);   // Prints "YES"
processAnswer(0);      // Prints "NO"
```

Tipo de União de Retorno da Função TypeScript

O TypeScript infere o tipo de retorno de uma função, portanto, se uma função retornar mais de um tipo de dados, o TypeScript inferirá que o tipo de retorno é uma união de todos os tipos de retorno possíveis. Se você deseja atribuir o valor de retorno da função a uma variável, digite a variável como uma união de tipos de retorno esperados.

```
const popStack = (stack: string[]) => {
  if (stack.length) {
    return stack[stack.length-1]; // return type is any
  } else {
    return null;      // return type is null
  }
};
let toys: string[] = ['Doll', 'Ball', 'Marbles'];
let emptyBin: string[] = [];
let item: string | null = popStack(toys); // item has union type
console.log(item);  // Prints "Marbles"
item = popStack(emptyBin);
console.log(item);  // Prints null
```

União TypeScript de tipos de matriz

TypeScript allows you to declare a union of an array of different types. Remember to enclose the union in parentheses, `(...)`, and append square brackets, `[]` after the closing parenthesis.

```
const removeDashes = (id: string | number) => {
  if (typeof id === 'string') {
    id = id.split('-').join('');
    return parseInt(id);
  } else {
    return id;
  }
}
// This is a union of array types
let ids: (number | string)[] = ['93-235-66', '89-528-92'];
let newIds: (number | string)[] = [];
for (let i=0; i < ids.length; i++) {
  newIds[i] = removeDashes(ids[i]); // Convert string id to
number id
}
console.log(newIds);  // Prints [9323566, 8952892]
```

TypeScript Union Type Common Property Access

As a result of supporting a union of multiple types, TypeScript allows you to access properties that are common among the member types without any error.

```
let element: string | number[] = 'Codecademy';
// The .length property is common for string and array
console.log(element.length);      // Prints 10
// The .match method only works for a string type
console.log(element.match('my')); // Prints ["my"]

element = [3, 5, 1];
// The length property is common for string and array
console.log(element.length);      // Prints 3
// The .match method will not work for an array type
console.log(element.match(5));  // Error: Property 'match' does
not exist on type 'number[]'.
```

TypeScript Union of Literal Types

You can declare a union type consisting of literal types, such as string literals, number literals or boolean literals. These will create union types that are more specific and have distinct states.

```
// This is a union of string literal types
type RPS = 'rock' | 'paper' | 'scissors' ;
const play = (choice: RPS): void => {
  console.log('You: ', choice);
  let result: string = '';
  switch (choice) {
    case 'rock':
      result = 'paper';
      break;
    case 'paper':
      result = 'scissors';
      break;
    case 'scissors':
      result = 'rock';
      break;
  }
  console.log('Me: ', result);
}
const number = Math.floor(Math.random()*3);
let choices: [RPS, RPS, RPS] = ['rock', 'paper', 'scissors'];
play(choices[number]);
```

Tipo de restrição

Limitação de tipo de união TypeScript

Como uma variável de um tipo de união pode assumir um dos vários tipos diferentes, você pode ajudar o TypeScript a inferir o tipo de variável correto usando a restrição de tipo. Para restringir uma variável a um tipo específico, implemente um protetor de tipo. Use o `typeof` operador com o nome da variável e compare-o com o tipo que você espera para a variável.

```
const choices: [string, string] = ['NO', 'YES'];
const processAnswer = (answer: number | boolean) => {
  if (typeof answer === 'number') {
    console.log(choices[answer]);
  } else if (typeof answer === 'boolean') {
    if (answer) {
      console.log(choices[1]);
    } else {
      console.log(choices[0]);
    }
  }
}
processAnswer(true);    // Prints "YES"
processAnswer(0);       // Prints "NO"
```

TypeScript Type Guard

Um protetor de tipo TypeScript é uma instrução condicional que avalia o tipo de uma variável. Ele pode ser implementado com o `typeof` operador seguido do nome da variável e compará-lo com o tipo que você espera para a variável.

```
// A type guard implemented with the typeof operator
if (typeof age === 'number') {
  age.toFixed();
}
```

Tipos aceitos do TypeScript Type Guard com`typeof`

O `typeof` operador pode ser usado para implementar uma proteção de tipo TypeScript para avaliar o tipo de uma variável incluindo `number` , `string` e `boolean` .

TypeScript Type Guard com `in`operador

Se uma variável for do tipo união, o TypeScript oferece outra forma de proteção de tipo usando o `in` operador para verificar se a variável possui uma propriedade específica.

```
/*
In this example, 'swim' in pet uses the 'in' operator to check
if the property .swim is present on pet. TypeScript recognizes
this as a type guard and can successfully type narrow this
function parameter.
*/
function move(pet: Fish | Bird) {
  if ('swim' in pet) {
    return pet.swim();
  }
  return pet.fly();
}
```

`if-else`Instrução de proteção de tipo TypeScript

If a variable is of a union type, TypeScript can narrow the type of a variable using a type guard. A type guard can be implemented as a conditional expression in an `if` statement. If an `else` statement accompanies the `if` statement, TypeScript will infer that the `else` block serves as the type guard for the remaining member type(s) of the union.

```
function roughAge(age: number | string) {
  if (typeof age === 'number') {
    // In this block, age is known to be a number
    console.log(Math.round(age));
  } else {
    // In this block, age is known to be a string
    console.log(age.split(".")[0]);
  }
}
roughAge('3.5');  // Prints "3"
roughAge(3.5);    // Prints 4
```

TypeScript Type Guard `if` Statement Function Return

If a variable is of a union type, TypeScript can narrow the type of a variable using a type guard. A type guard can be implemented as a conditional expression in an `if` statement. If the `if` block contains a `return` statement and is not followed by an `else` block, TypeScript will infer the rest of the code block outside the `if` statement block as a type guard for the remaining member type(s) of the union.

```
function formatAge(age: number | string) {
  if (typeof age === 'number') {
    return age.toFixed(); // age must be a number
  }
  return age; // age must not be a number
}
console.log(formatAge(3.5));    // Prints "4"
console.log(formatAge('3.5'));  // Prints "3.5"
```

Tipos de objetos avançados

Tipo de interface TypeScript

O TypeScript permite que você digite especificamente um objeto usando uma interface que pode ser reutilizada por vários objetos. Para criar uma interface, use a palavra- `interface` chave seguida do nome da interface e do objeto digitado.

```
interface Publication {
  isbn: string;
  author: string;
  publisher: string;
}
```

Interface TypeScript Definir somente objetos

No TypeScript, os aliases de tipo podem definir tipos compostos, como objetos e uniões, bem como tipos primitivos, como números e strings; interface, no entanto, só pode definir objetos. Interface é útil na digitação de objetos escritos para programas orientados a objetos.

```
// Type alias can define a union type
type ISBN = number | string;

// Type alias can define an object type
type PublicationT = {
  isbn: ISBN;
  author: string;
  publisher: string;
}

// Interface can only define an object type
interface PublicationI {
  isbn: ISBN;
  author: string;
  publisher: string;
}
```

Interface TypeScript para Classes

Para aplicar uma interface TypeScript a uma classe, adicione a palavra-`implements` chave após o nome da classe seguida pelo nome da interface. O TypeScript verificará e garantirá que o objeto realmente implemente todas as propriedades e métodos definidos dentro da interface.

```
interface Shape {
  area: number;
  computeArea: () => number;
}

const PI: number = 22/7 ;

// Circle class implements the Shape interface
class Circle implements Shape {
  radius: number;
  area: number;
  constructor(radius: number) {
    this.radius = radius;
    this.area = this.computeArea();
  }
  computeArea = (): number => {
    return PI * this.radius * this.radius;
  }
}

let target = new Circle(3);
console.log(target.area.toFixed(2));  // Prints "28.29"
```

Interface aninhada TypeScript

O TypeScript permite que os aliases de tipo e a interface sejam aninhados. Um objeto tipado com uma interface aninhada deve ter todas as suas propriedades estruturadas da mesma forma que a definição da interface.

```
// This is a nested interface
interface Course {
  description: {
    name: string;
    instructor: {
      name: string;
    }
    prerequisites: {
      courses: string[];
    }
  }
}

class myCourse implements Course {
  description = {
    name: '',
    instructor: {
      name: ''
    },
    prerequisites: {
      courses: []
    }
  }
}
```

Interfaces de aninhamento de TypeScript dentro de uma interface

Since interfaces are composable, TypeScript allows you to nest interfaces within an interface.

```
// Date is composed of primitive types
interface Date {
  month: number;
  day: number;
  year: number
}

// Passport is composed of primitive types and nested with
another interface
interface Passport {
  id: string;
  name: string;
  citizenship: string;
  expiration: Date;
}

// Ticket is composed of primitive types and nested with another
interface
interface Ticket {
  seat: string;
  expiration: Date;
}

// TravelDocument is nested with two other interfaces
interface TravelDocument {
  passport: Passport;
  ticket: Ticket;
}
```

TypeScript Interface Inheritance

Like JavaScript classes, an interface can inherit properties and methods from another interface using the `extends` keyword. Members from the inherited interface are accessible in the new interface.

```
interface Brand {
  brand: string;
}

// Model inherits property from Brand
interface Model extends Brand {
  model: string;
}

// Car has a Model interface
class Car implements Model {
  brand;
  model;
  constructor(brand: string, model: string) {
    this.brand = brand;
    this.model = model;
  }
  log() {
    console.log(`Drive a ${this.brand} ${this.model} today!`);
  }
}

const myCar: Car = new Car('Nissan', 'Sentra');
myCar.log(); // Prints "Drive a Nissan Sentra today!"
```

TypeScript Interface Index Signature

Property names of an object are assumed to be strings, but they can also be numbers. If you don't know in advance the types of these property names, TypeScript allows you to use an index signature to specify the type for the property name inside an object. To specify an index signature, use square brackets, `[...]` , to surround the type notation for the property name.

```
interface Code {
  [code: number]: string;
}
const codeToStates: Code = {603: 'NH', 617: 'MA'};

interface ReverseCode {
  [code: string]: number;
}
const stateToCodes: ReverseCode = {'NH': 603, 'MA': 617};
```

TypeScript Interface Optional Properties

TypeScript allows you to specify optional properties inside an interface. This is useful in situations where not all object properties have values assigned to them. To indicate if a property is optional, append a `?` symbol after the property name before the colon, `:` .

```
interface Profile {
  name: string;
  age: number;
  hobbies?: string[];
}

// The property, hobbies, is optional, but name and age are
required.
const teacher: Profile = {name: 'Tom Sawyer', age: 18};
```