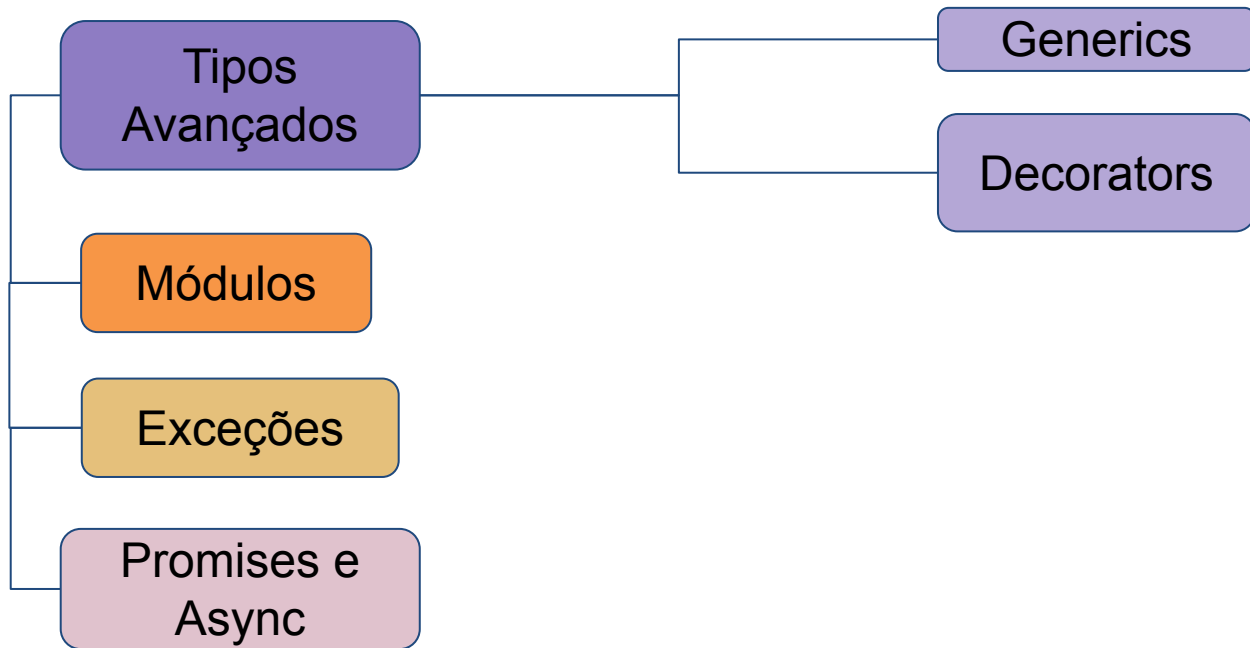




Maiara de Souza Coelho
Instituto de Computação
UFAM

Tópicos da Aula 3



Generics

- Um trecho de código pode operar com tipos diferentes dependendo de como for chamado.
- Se usar any, vai dar para prever todos os assertions necessários??
- Uso de convenções para parâmetros de tipo como T e U ou Key e Value entre parâmetros angulares <>.
- Funções Genéricas
- Interfaces Genéricas
- Classes Genéricas
- Aliases de Tipo Genéricos

Generics: Funções

- Inserção de um alias de parâmetro de tipo T entre parâmetros angulares, antes dos parênteses dos parâmetros

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

```
const stringy = identity("Eu"); // Tipo Eu  
const numeric = identity(123); // Tipo 123
```

```
const identity = <T>(input:T) => input;  
identity(123); // Tipo 123
```

Generics: Pode ter mais de um parâmetro

- Argumento e retorno genérico. Definir explicitamente na chamada.

```
function fun<T, U, V> (args1:T, args2: U, args3: V) : V {  
    return args3;  
}  
  
console.log(fun<string, number, boolean>('teste', 1, true))  
//retorna true
```

Generics: Interfaces

- Os tipos são inferidos da mesma forma como acontece nas funções.

```
interface Box<T>{  
    inside: T  
}  
  
let stringBox: Box<string> = {  
    inside: "abc"  
}  
  
let numberBox: Box<number> = {  
    inside: 123  
}
```

Generics: Classes

- Assim como em funções e interfaces, podem-se declarar mais de um parâmetro genérico e depois utilizar para propriedades, tipos de parâmetros em construtor, tipos de parâmetros dos métodos e tipo de retorno.

Generics: Classes

```
class Secret<Key, Value>{  
    key: Key;  
    value: Value;  
  
    constructor(key:Key,  
value: Value){  
        this.key = key;  
        this.value = value;  
    }  
}
```

```
    getValue(key:Key):  
Value|undefined{  
        return this.key ===  
key?this.value:undefined;  
    }  
}  
  
const storage = new  
Secret(12345, "lugae");  
storage.getValue(1997);
```


Generics: Extensões de Classes

- A subclasse deve especificar explicitamente o tipo que será utilizado por ela na classe base.

```
class Quote<T>{  
    lines: T;  
    constructor(lines: T) {  
        this.lines = lines;  
    }  
}
```

```
class SpokenQuote extends  
    Quote<string[]>{  
    speak() {  
        console.log(this.lines.join("\n"));  
    }  
}
```

```
new SpokenQuote([ "greed is so destructive", "it destroys  
everything" ]).lines;
```

Generics: Implementação de Interfaces

- Funcionam semelhantemente ao extends.
- Qualquer parâmetro de tipo da interface deve ser declarado pela classe.

```
interface ActingCredit<Role>{  
    role: Role;  
}
```

```
class MovePart implements  
ActingCredit<string>{  
    role: string;  
    speaking: boolean;
```

```
    constructor(role: string,  
speaking:boolean) {  
        this.role = role;  
        this.speaking = speaking;  
    }  
}  
  
const part = new  
MovePart("Miranda", true);  
part.role;
```

Generics: Aliases de Tipo

- Geralmente são usados com funções para descrever o tipo de uma função genérica

Entrada: string

Saída: number

```
type CreatesValue<Input,Output> = (input: Input) => Output;  
let creator: CreatesValue<string,number>;  
creator = text => text.length
```

Generics: Exemplo de utilização Array

- Array usa Generics

```
interface Array<T> {  
    includes(searchElement: T, fromIndex?: number): boolean;  
}
```

T é uma string

```
const nomes: Array < string> = [  
    'pessoa1', 'pessoa2', 'pessoa3'];
```

Agora é um number

```
const dias: Array< number> = [5, 25, 28]
```

Decorators

- Nos permite configurar de forma dinâmica uma classe
- Descomentar a linha em tsconfig.js: "experimentalDecorators" : true,'
- Definido pelo caracter '@'.
- Podemos definir decorators a partir de 3 parâmetros:
 - target (alvo) - função construtora de uma classe.
 - propertyKey (chave) - nome do membro da instância que será utilizado, propriedade, por ex.
 - descriptor (descritor) - a propriedade descriptor do membro da instância, chamando o método.

Decorators: Método Decorator

```
class Conta {  
    numeroDaConta: number;  
    titular: string;  
    saldo: number;  
    constructor(numeroDaConta: number, titular: string, saldo:  
number) {  
        this.numeroDaConta = numeroDaConta;  
        this.titular = titular;  
        this.saldo = saldo;  
    }  
}
```

Decorators: Método Decorator

```
function analisaSaldo(target: any, key: any, descriptor: any) {  
  //implementação  
}
```

```
class ContaBancaria {  
  ...  
  @analisaSaldo  
  consultaSaldo(): string {  
    return `O seu saldo atual é: ${this.saldo}`;  
  }  
}
```

Decorators: Decorator de Propriedade

- Função com target e propertykey, onde key é o nome da propriedade.

```
function validaTitular(target: any, propertyKey: any) {  
    //implementação de decorator de propriedade  
}  
  
class Conta {  
    numeroDaConta: number;  
    @validaTitular  
    titular: string  
    ...  
}
```


Decorators: Decorator de Parâmetro

- Recebe 3 parâmetro: target, propertyKey que é o nome do método que contém o parâmetro e parameterIndex que indica a posição do parâmetro.

```
function saldo() {  
    return (target: any,  
            propertyKey: string,  
            parameterIndex: number) => {  
        console.log('target'+ target);  
        console.log('property key', propertyKey);  
        console.log('parameter index', parameterIndex);  
    }  
}
```

Decorators: Decorator de Parâmetro

```
class ContaBancaria {  
    ...  
    adicionaSaldo(@saldo() saldo: number): void {  
        this.saldo+=saldo;  
    }  
}
```

Decorators: Decorator de Classe

- Recebe apenas o construtor da classe.

```
function log(ctor: any) {  
    console.log(ctor)  
}  
  
@log  
class ContaBancaria { ... }
```

Decorators: Decorator Factory

- Quando se quer uma interação entre classe e decorator.

```
function analisaConta(tipoConta: any) {  
  return (_target: any) => {  
    console.log(` ${tipoConta} - ${_target}`);  
  }  
}  
  
@analisaConta( 'PJ')  
class ContaBancaria {}
```

Decorators: Múltiplos

- Um ou mais decorator para a mesma classe

```
@log  
@analisaConta( 'PJ' )  
class ContaBancaria {}
```

Módulos

- export - na importação o mesmo nome, sem chaves
- export default: um por arquivo

```
//conta.ts
export class Conta {
    numeroDaConta: number;
    titular: string;
    saldo: number;
    /* outras implementações */
}
```

Módulos

```
//contaInvestimento.ts
```

```
import { Conta } from "../Conta";
```

```
export class ContaInvestimento extends Conta { }
```

```
//index.ts
```

```
import { ContaInvestimento } from "../contaInvestimento";
```

```
let novaContaInvestimento = new ContaInvestimento();
```

Namespaces

```
namespace Banco {
```

```
  export class Conta {
```

```
  }
```

```
}
```

```
namespace Banco {
```

```
  export class ContaPF extends Conta { }
```

```
}
```

```
namespace Banco {
```

```
  export class ContaPJ extends Conta { }
```

```
}
```

```
let conta = new Banco.Conta();
```


Exceções

- A classe Erro
- Bloco try / catch /finally
- Tipos de erros definidos pela LP

Exceções: Error

```
//Isso:
```

```
const erro1 = Error('Criado por uma chamada de função');
```

```
//É o mesmo que isso:
```

```
const erro2 = new Error('Criado com a palavra chave new');
```

```
//Lançando o erro
```

```
//throw erro2;
```

```
class DivisaoPor0 extends Error{
```

```
    constructor() {
```

```
        super("Erro: Divisão por 0");
```

```
    }
```

```
}
```

Exceções: Error

```
function f1 () {  
    throw new Error("Erro em f1");  
}  
  
function f2 () {  
    throw "me lançaanaa";  
}  
  
function f3 () {  
    return 3  
}  
  
throw f3(); //throw f3(); ^ 3
```

Exceções: Bloco try / catch /finally

```
function getNomedoMes (mes:number) {  
    enum meses {Janeiro = 1,  
                Fevereiro,Marco,Abril,Maio,Junho,Julho,  
                Agosto,Setembro,Outubro,Novembro,Dezembro};  
    if (mes>=1 && mes<=12 ) {  
        return meses[mes];  
    }  
    else {  
        throw Error("Mês invalido!");  
    }  
}
```

Exceções: Bloco try / catch /finally

```
try{  
    console.log(getNomedoMes(13));  
}  
catch(e) {  
    //if(e instanceof Error)  
    console.log(e);  
}  
finally{  
    console.log("Sempre serei notado.")  
}
```

Erros Definidos pela LP

- `InternalError`: Lançado quando um erro interno ocorre, como excesso de recursão “too much recursion”
- `RangeError`: Quando uma variável ou parâmetro numérico está fora de intervalo válido.
- `ReferenceError`: É lançado ao tentar referenciar uma variável que não foi declarada.
- `SyntaxError`: Quando ocorre erro de sintaxe.
- `TypeError`: Lançado quando uma variável ou parâmetro não é de um tipo válido.

Funções Assíncronos

- São funções que não seguem o fluxo normal síncrono. Ou seja, em sua execução o programa pode optar por tomar outras ações.
- Possibilidade de erros, como funções que precisam do valor de uma função assíncrona serem terminadas antes do valor ser retornado pela função assíncrona.
- Funções assíncronas retornam promises.
- Palavra chave “async”.

Promises

- Possui 5 estados, Pendente, Resolvida, Rejeitada, Realizada e Estabelecida.
- Como o nome diz, é uma promessa de uma ação, se essa ação for bem sucedida, teremos uma promise resolvida e poderemos usufruir de seu resultado.
- `Promise.all`

Promises

```
async function corrida(x: number) {  
    setTimeout(() => console.log(x, " está  
correndo..."), Math.random() * x);  
    return "terminou " + x;  
}  
  
let p = Promise.all([corrida(1), corrida(2),  
corrida(3), corrida(4), corrida(5)]);  
console.log(p);  
setTimeout(() => console.log(p), 10);
```

Exercício 6

- Faça um programa para gerenciar as turmas de um ou mais cursos.
- A turma deve ter:
 - id que não pode ser alterado
 - descrição
 - turno como um enum (manhã, tarde e noite)
 - curso: descrição e área (humanas, biológicas e exatas)
- As turmas devem ser armazenadas em um vetor
 - Implemente métodos para adicionar, excluir, alterar, buscar e imprimir as turmas.
- Complete o exercício usando decoradores na classe para validar tamanho de caracteres.
 - O número mínimo de caracteres deve ser 10.

Fim!

Obrigada

Dúvidas: Slack

maiara@icomp.ufam.edu.br

github: mayara-msc@hotmail.com