

Typescript

**Classes, herança, interfaces,
módulos e generics**

Ely – elydasilvamiranda@gmail.com

Classes

- As definições de classes em TypeScript são muito semelhantes às das linguagens Java e C#;
- Nelas existem: atributos, construtores e métodos:

```
class Alo {  
    nome: string;  
  
    constructor(nome: string) {  
        this.nome = nome;  
    }  
  
    dizerAlo() {  
        console.log("Alô, " + this.nome);  
    }  
}
```

Classes

- As classes usam como conversão o mesmo padrão CamelCase do Java em suas definições;
- Um construtor usa a palavra reservada **constructor**;
- Acesso a atributos e métodos dentro da própria classe devem ser feitos usando **this**;
- Métodos têm as mesmas convenções de funções, **sem** o uso da palavra reservada **function**.

Classes

- Instanciando e uso de uma classe:

```
let alo = new Alo("Ely");  
alo.dizerAlo();
```

- Para simplificar a definição de atributos usamos modificadores de visibilidade ou a palavra **readonly**:

```
class Alo {  
  
    constructor(public nome: string) {}  
  
    dizerAlo() {  
        console.log("Alô, " + this.nome);  
    }  
}
```

Métodos de acesso

- Métodos de acesso **get** e **set** podem ser criados para ler e escrever em atributos;
- Uma convenção é definir atributos privados com um “_” antes do nome:

```
class Post {  
    constructor(private _text: string) {}  
  
    get text(): string {  
        return this._text;  
    }  
  
    set text(text: string) {  
        this._text = text;  
    }  
}
```

Métodos de acesso

- Os métodos de acesso definem algo como “propriedades” ;
- O acesso ao atributo fica então encapsulado pelo nome dado aos métodos de acesso:

```
let p = new Post("post text");  
p.text = "reviewed text";  
console.log(p.text);
```


Modificador readonly

- O modificador **readonly** cria um atributo somente leitura;
- Nota: não é possível declarar um atributo como **const**:

```
class Post {  
    constructor(private _text: string,  
                 readonly owner: String) {}  
  
    //métodos de acesso de _texto omitidos  
}
```

```
let p = new Post("post text", "Ely");  
p.owner = "new owner";  
console.log(p.text);
```

Cannot assign to 'autor'
because it is a read-only
property



Herança

- A herança permite estender um modelo dando novas características e ações a uma classe;
- TypeScript implementa **herança simples** utilizando a palavra reservada **extends**:

```
class AdvancedPost extends Post {  
    private _likes: number = 0;  
  
    like() {  
        this._likes++;  
    }  
}
```


Conceitos idênticos aos de Java e C#

- Os seguintes conceitos não serão vistos aqui por questão de tempo:
 - Classes abstratas;
 - Sobrecarga;
 - Sobrescritas;
 - Métodos estáticos.

Interfaces

- Uma classe pode implementar diversas interfaces como forma de estabelecer um contrato;
- Implementar uma interface obriga à classe a declarar os métodos da interface;
- Além disso, caso a interface tenha atributos, a classe também deve declará-los:

```
interface Indexable {  
    hashtags : string[];  
    addHashtag(addHashtag: string);  
}
```

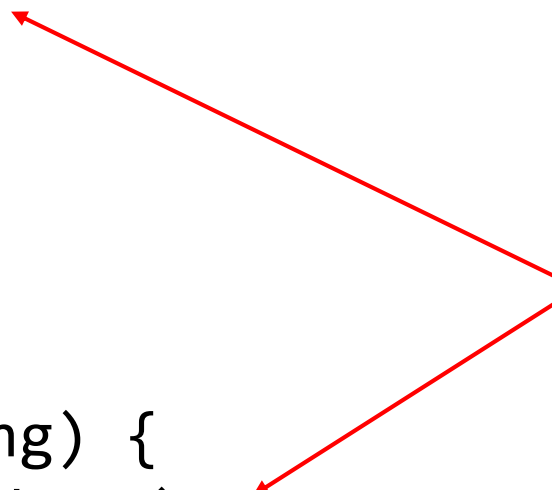
Interfaces

```
class AdvancedPost extends Post
    implements Indexable {

    private _likes: number = 0;
    hashtags: string[] = [];

    like() {
        this._likes++;
    }

    addHashtag(hashtag: string) {
        this.hashtags.push(hashtag);
    }
}
```



Interfaces

```
let p = new AdvancedPost("post text", "Ely");  
p.addHashtag('tbt');  
p.addHashtag('bomdia');  
p.like();  
console.log(`Este post pertence a ${p.owner} e tem  
${p.hashtags.length} hashtag(s).`);
```

Módulos

- Módulos são unidades que podem outros elementos como classes, interfaces e etc;
- São usados de organizar o código e isolar partes de implementações;
- Com isso, elementos de um módulo podem ser exportados e importados quando necessários;
- Na prática um módulo é um arquivo .ts com definições com a opção **export**.

Módulos

```
// arquivo classes.ts
```

```
export class Post {
```

```
    // ...
```

```
}
```

```
interface Indexable {
```

```
    // ...
```

```
}
```

```
export class AdvancedPost extends Post
```

```
    implements Indexable {
```

```
    // ...
```

```
}
```

Módulos

- Uma outra forma de exportar definições é uma seção especificação ao final do arquivo:

```
class Post { /*... */ }
```

```
interface Indexable { /*... */ }
```

```
class AdvancedPost extends Post  
    implements Indexable {
```

```
    // ...
```

```
}
```

```
export { Post, AdvancedPost }
```

Módulos

- Para importar os tipos de um módulo usamos a palavra `import`;
- Os tipos são separados por vírgula; e o caminho é relativo ao arquivo que os importam;
- O nome do arquivo não precisa da extensão:

```
import {Post, AdvancedPost} from "./classes"
```

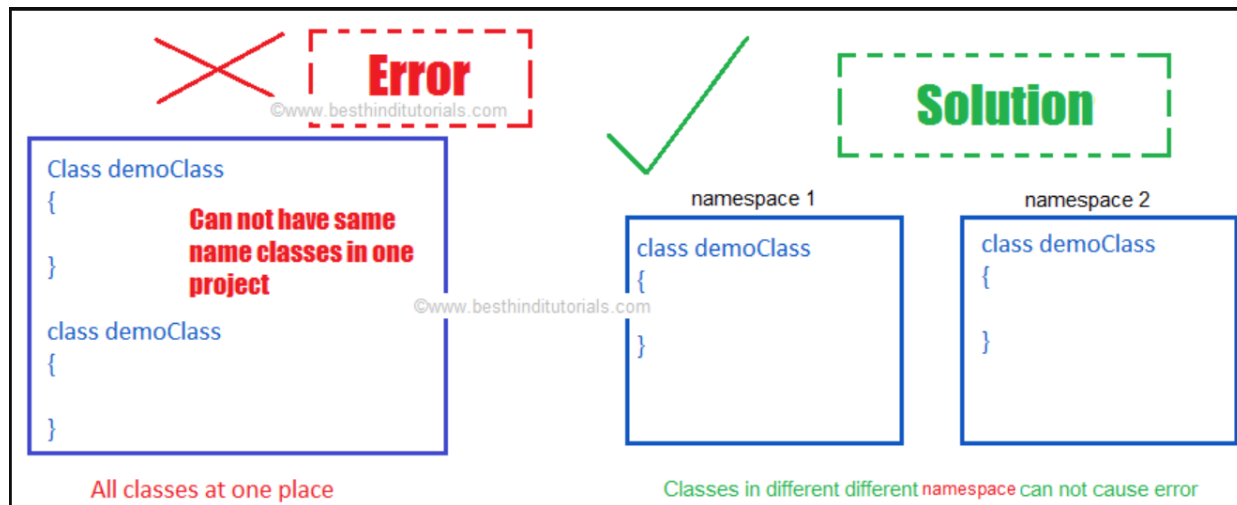
```
let p1 = new Post("ipsum loren", "Joe");
```

```
let p2 = new AdvancedPost("post text", "Ely");
```

```
//...
```


Namespaces

- São formas de organizar o código evitando conflito de nomes;
- Antigamente eram conhecidos como módulos internos;
- Elementos visíveis dentro de um namespace devem ser precedidos de export.



Namespaces

```
namespace Mensagens {  
    export function info(msg) {  
        console.log(msg);  
    }  
    export function erro(msg) {  
        console.error(msg);  
    }  
}  
Mensagens.info("apenas um log tradicional");  
Mensagens.erro("Um erro no console");
```

Generics

- São construções que permitem definir funcionalidades com tipos plugáveis;
- Definem *templates* com funcionalidades que devam funcionar com diferentes tipos;
- Utilizados na construção de componentes e são alternativas à herança ou ao uso do tipo any;
- Em TypeScript podem ser utilizados com classes, interfaces e funções.



Generics

- Dada a definição de uma estrutura de dados de pilha (Last In, First Out) para números:

```
class NumberStack {  
    private _data : number[] = [];  
  
    push(item: number) {  
        this._data.push(item);  
    }  
  
    pop() {  
        return this._data.pop();  
    }  
  
    getData() {  
        return this._data;  
    }  
}
```

Generics

- Usando a classe:

```
let s = new NumberStack();
```

```
s.push(1);
```

```
s.push(2);
```

```
s.push(3);
```

```
console.log(s.getData()); // [1,2,3]
```

```
s.pop();
```

```
console.log(s.getData()); // [1,2]
```

Generics

- Dada outra definição de pilha (Last In, First Out) para strings:

```
class StringStack {  
    private _data : string[] = [];  
  
    push(item: string) {  
        this._data.push(item);  
    }  
  
    pop() {  
        return this._data.pop();  
    }  
  
    getData() {  
        return this._data;  
    }  
}
```

Generics

- Usando a classe:

```
let s = new StringStack ();
```

```
s.push("a");
```

```
s.push("b");
```




```
s.push("c");
```

```
console.log(s.getData()); // ["a","b","c"]
```

```
s.pop();
```

```
console.log(s.getData()); // ["a","b"]
```

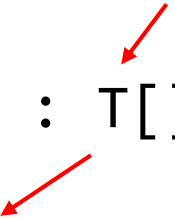
Generics

- As funcionalidades são as mesmas, porém os tipos de dados são diferentes;
- Alternativas para evitar redundância de implementações:
 1. Implementar sem tipos, à maneira JavaScript:
 - Perderíamos a principal vantagem do uso do TypeScript;
 2. Implementar com herança:
 - Teríamos que criar uma hierarquia de tipos;
 3. Utilizar Generics:
 - Teremos um componente reutilizável e ainda uma checagem de tipo.

Generics

- Na implementação com Generics, substituem-se todas as ocorrências do tipo por uma “letra”:

```
class Stack<T> {  
    private _data : T[] = [];  
    push(item: T) {  
        this._data.push(item);  
    }  
    pop() { return this._data.pop(); }  
    getData() {  
        return this._data;  
    }  
}
```



Generics

- Ao criar uma instância de classe que utiliza um tipo genérico, devemos especificar o tipo:

```
let ns = new Stack<number>();
```

```
ns.push(1);
```

```
ns.push("a");
```

← Argument of type '"a"' is not assignable to parameter of type 'number'

```
let ss = new Stack<string>();
```

```
ss.push("a");
```

Typescript

**Classes, herança, interfaces,
módulos e generics**

Ely – elydasilvamiranda@gmail.com