

Desenvolvimento Web

Trilha JavaScript com Angular e Node

Instrutor: Júlio Pereira Machado (julio.machado@pucrs.br)



Classes



Classes

- A partir do ECMAScript 6, o suporte a classes foi adicionado ao JavaScript
- TypeScript suporta a definição de objetos através de classes
- São uma construção sintática para facilitar a definição de classes sobre a estrutura de função construtora + propriedade prototype do JavaScript
- Palavras-chave:
 - class – definição de classes
 - constructor – nome especial para o método construtor
 - get – definição de propriedades de leitura
 - set – definição de propriedades de escrita

Classes

- TypeScript suporta os modificadores usuais de acesso:
 - `public`
 - `private`
 - `protected`
- Se não aparece um modificador explícito, usa *public* implicitamente

Classes

- Exemplo:

```
class Pessoa {  
    private _nome: string  
    private _idade: number  
    constructor(umNome: string, umaIdade: number) {  
        this._nome = umNome;  
        this._idade = umaIdade;  
    }  
    fazAniversario(): void {  
        this._idade = this._idade + 1;  
    }  
    get nome(): string { return this._nome };  
    get idade(): number { return this._idade; }  
}
```

atributos

construtor

método

propriedades de leitura

Classes

- Exemplo:

```
let p: Pessoa = new Pessoa('John Doe', 12);  
p.fazAniversario();  
console.log(p.nome);  
console.log(p.idade);
```

Classes

- TypeScript suporta uma notação que une a definição dos campos do objeto junto com a inicialização do construtor
 - É chamada de *parameter properties*
- Exemplo:

```
class Pessoa {  
  constructor(private _nome: string, private _idade: number) {  
  }  
  ...  
}
```

Classes

- Exemplo:

```
class Pessoa {  
    constructor(private _nome: string, private _idade: number) {  
    }  
    fazAniversario(): void {  
        this._idade = this._idade + 1;  
    }  
    get nome(): string { return this._nome };  
    get idade(): number { return this._idade; }  
}
```

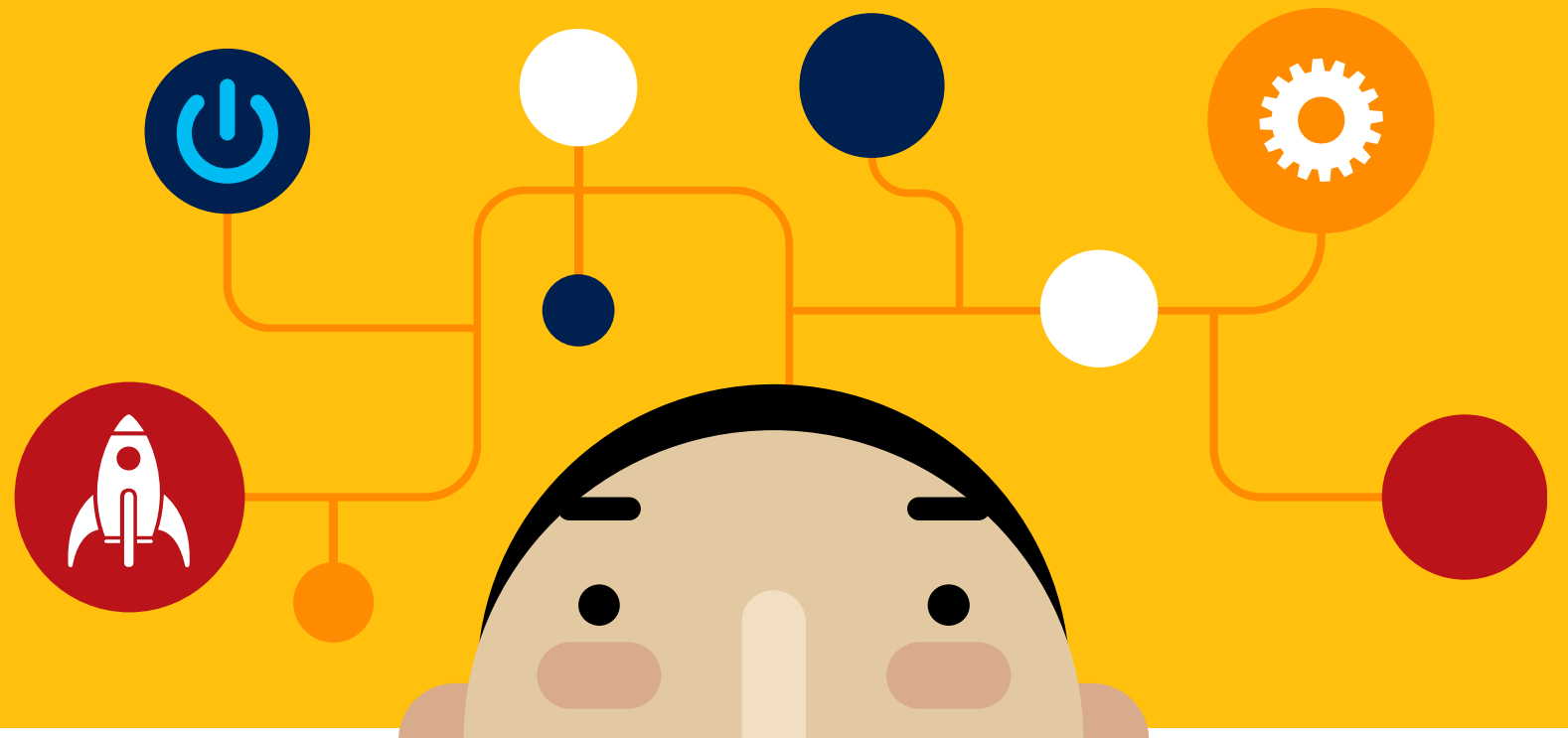

Atributos e Métodos de Classe

Atributos e métodos acessados pelo nome da classe

- Em TypeScript utiliza-se o modificador `static`

Laboratório

- Abra as instruções do arquivo Lab03_TypeScript_Objetos



Classes e Herança



Herança

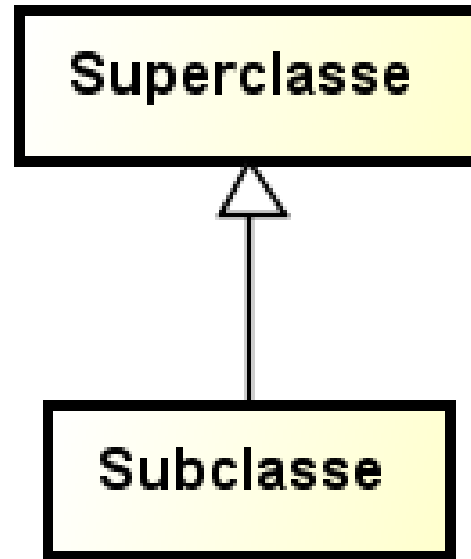
Herança é uma relação de generalização/especialização entre classes

A ideia central de herança é que novas classes são criadas a partir de classes já existentes

- Superclasse: classe já existente
- Subclasse: classe criada a partir da superclasse

Diagrama de Classes UML

Relacionamento de herança:



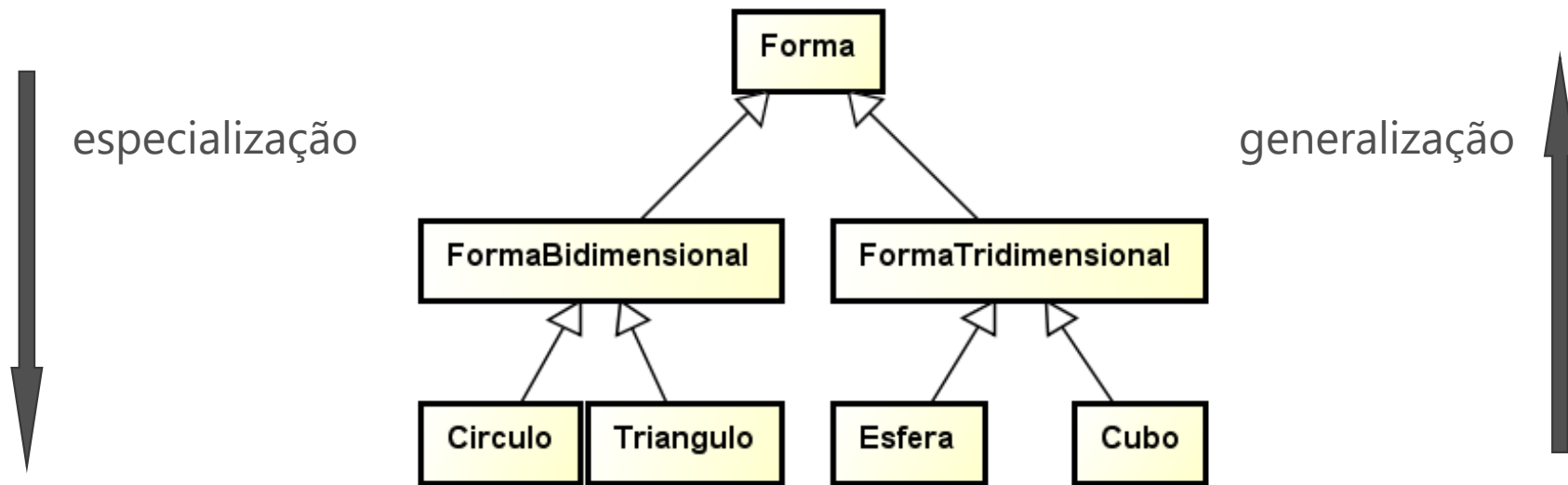
powered by astah* 

Herança

Herança cria uma estrutura hierárquica

Ex.: uma hierarquia de classes para formas geométricas

- Uma forma geométrica pode ser especializada em dois tipos: bidimensional e tridimensional



Herança

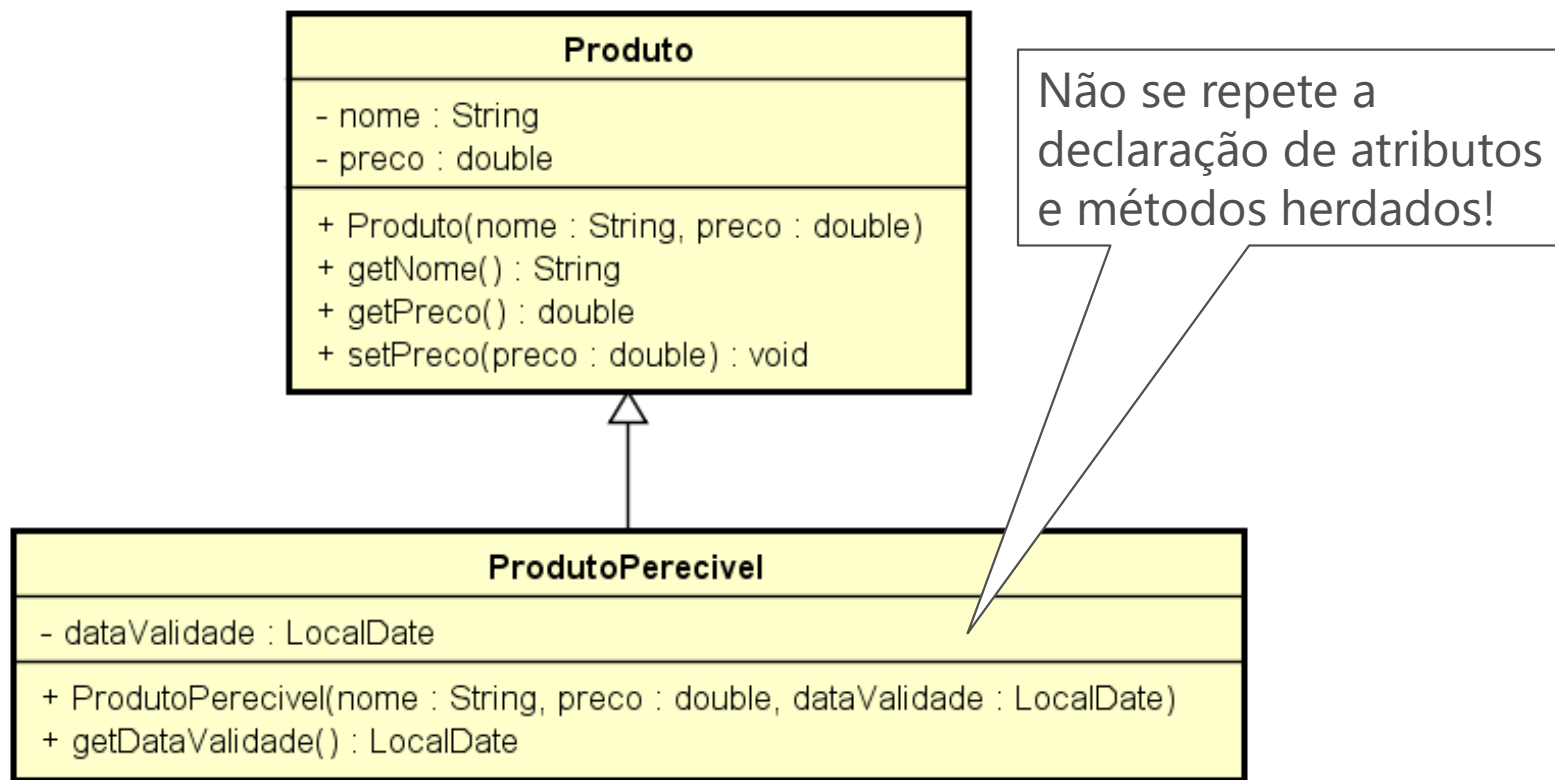
Como implementar herança em TypeScript?

- Utiliza-se a palavra-chave `extends` para definir herança de classes
- Somente é possível herdar de uma única superclasse!

```
class Subclasse extends Superclasse {  
  ...  
}
```

Herança

Exemplo:



Herança

- Exemplo:

```
class Produto {  
    constructor(private _nome: string, private _preco: number) {  
    }  
    get nome() {return this._nome;}  
    get preco() {return this._preco;}  
    set preco(preco: number) {this._preco = preco;}  
}
```

```
class ProdutoPercivel extends Produto {  
    constructor(nome: string, preco: number, private _dataValidade: Date) {  
        super(nome,preco);  
    }  
    get dataValidade() {return this._dataValidade;}  
}
```

Herança

- Exemplo:

```
let p1 = new Produto('produto 1', 1.99);  
let p2 = new ProdutoPercivel('produto 2', 2.50, new Date(2018,10,1));  
console.log(p1.nome);  
console.log(p1.preco);  
console.log(p2.nome);  
console.log(p2.preco);  
console.log(p2.dataValidade);
```

Sobrescrita de Métodos

Uma subclasse pode sobrescrever ("override") métodos da superclasse

- Sobrescrita permite completar ou modificar um comportamento herdado
- Quando um método é referenciado em uma subclasse, a versão escrita para a subclasse é utilizada, ao invés do método na superclasse
- É possível acessar o método original da superclasse: `super.nomeDoMetodo()`

Sobrescrita de Métodos

- Exemplo:

```
class Produto {  
    ...  
    toString(): string {return `[nome=${this._nome}, preco=${this._preco}]`;}  
}
```

```
class ProdutoPercivel extends Produto {  
    ...  
    toString(): string {return super.toString() +  
    `[dataValidade=${this._dataValidade.toDateString()}]`;}  
}
```

Classes e Métodos Abstratos

Em uma hierarquia de classe, quanto mais alta a classe na hierarquia, mais abstrata é sua definição

- Uma classe no topo da hierarquia define o comportamento e atributos que são comuns a todas as classes
- Em alguns casos, a classe nem precisa ser instanciada alguma vez e cumpre apenas o papel de ser um repositório de comportamentos e atributos em comum

Classes e Métodos Abstratos

Classes abstratas são classes que não podem ser instanciadas

São utilizadas apenas para permitir a derivação de novas classes

Identificamos uma classe como abstrata pelo modificador **abstract**

```
abstract class MinhaClasse{  
    ...  
}
```

Em uma classe abstrata, um ou mais métodos podem ser declarados sem o código de implementação

- São os métodos abstratos

Classes e Métodos Abstratos

Métodos abstratos são métodos sem código de implementação

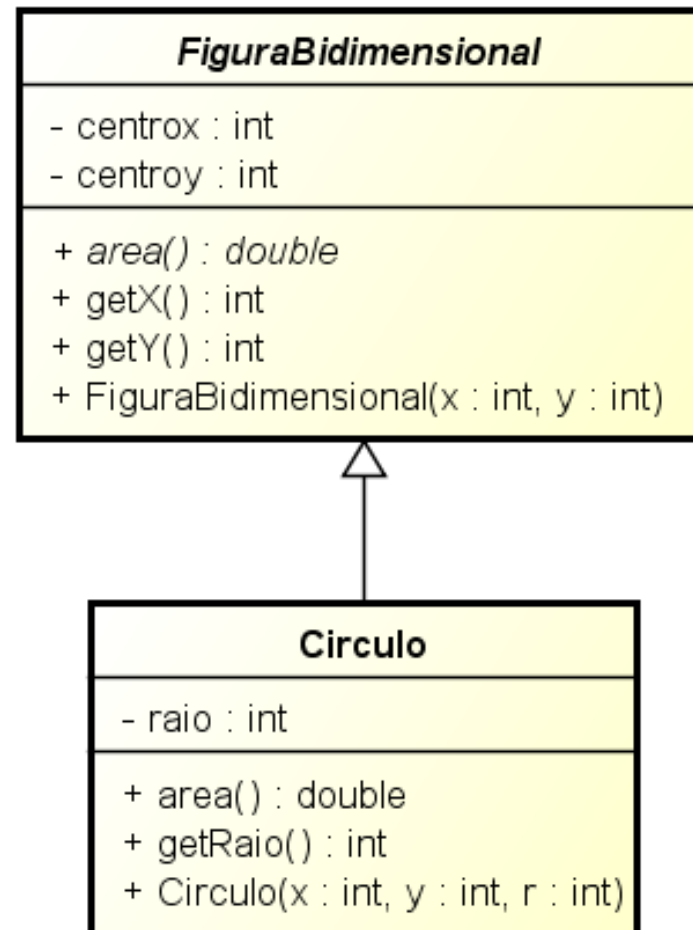
- São prefixados pela palavra `abstract`
- Não apresentam um corpo
- Sua declaração termina com `;` após a declaração dos parâmetros

Um método abstrato indica que a classe não implementa aquele método e que ele deve ser obrigatoriamente implementado nas classes derivadas, pois é um comportamento comum das subclasses

```
abstract class MinhaClasse{  
    abstract metodo(): void;  
}
```

Classes e Métodos Abstratos

Exemplo



Classes e Métodos Abstratos

- Exemplo:

```
abstract class FiguraBidimensional {  
    constructor(private _centrox: number, private _centroy: number){  
    }  
    get x() {return this._centrox;}  
    get y() {return this._centroy;}  
    abstract area(): number;  
}
```

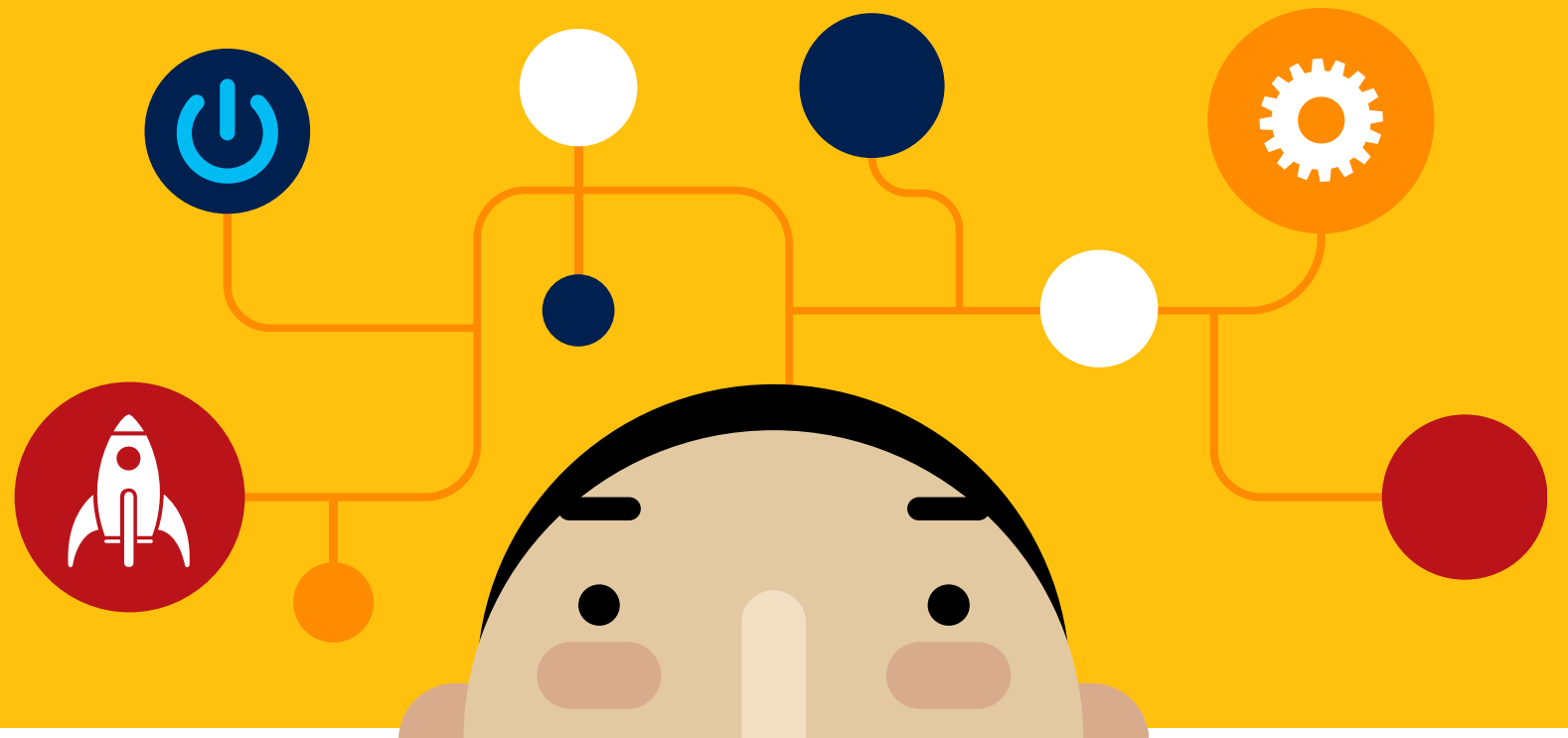
Classes e Métodos Abstratos

- Exemplo:

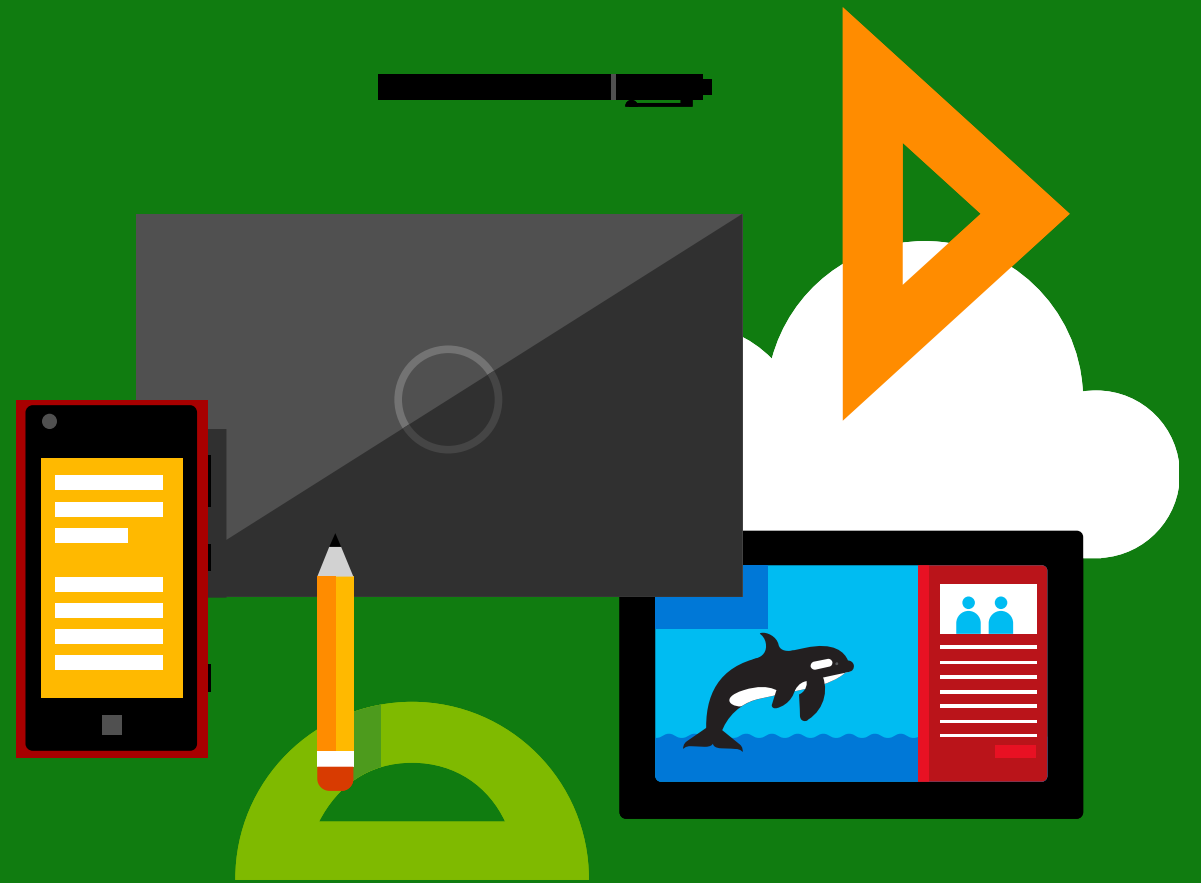
```
class Circulo extends FiguraBidimensional {  
    constructor(x: number, y: number, private _raio: number) {  
        super(x,y);  
    }  
    get raio() {return this._raio;}  
    area(): number {  
        return Math.PI * this._raio ** 2;  
    }  
}
```

Laboratório

- Abra as instruções do arquivo Lab03_TypeScript_Objetos



Objetos Literais



Objetos Literais

- Em JavaScript, objetos literais são coleções indexadas por “nomes”
- Em sua estrutura mais básica, um objeto é uma coleção de pares **nome:valor**, chamados de **propriedades**
 - O nome de uma propriedade é uma String (ou um Symbol)
 - Uma propriedade pode armazenar um valor de qualquer tipo

Definição de Objetos Literais

- Um objeto JavaScript/TypeScript pode ser definido de forma literal com a descrição de propriedades entre chaves { }

```
let pessoa = {  
  nome : 'John Doe',  
  idade : 22  
};  
console.log(pessoa.nome);  
pessoa.idade = 23;
```

Desestruturando Objetos

- Objetos podem também ser desestruturados em suas partes pelo operador de atribuição
 - Ordem não importa, mapeamento é por identificador

```
let pessoa = {  
  nome : 'John Doe',  
  idade : 22  
};  
let {idade, nome} = pessoa;  
console.log(nome);
```

- Identificador pode ser renomeado via ":"

```
let pessoa = {  
  nome : 'John Doe',  
  idade : 22  
};  
let {nome:n, idade} = pessoa;  
console.log(n);
```

Interfaces



Compatibilidade de Tipos

Observe o seguinte exemplo:

```
function printLabel(labelledObj: { label: string }) {  
    console.log(labelledObj.label);  
}  
  
let myObj = {size: 10, label: "Size 10 Object"};  
printLabel(myObj);
```

- TypeScript irá verificar apenas o mínimo necessário para garantir a compatibilidade dos tipos envolvidos em um processo conhecido como *"duck typing"*

Interfaces

Em TypeScript, uma interface é um contrato sintático para um tipo

- Define a “forma” que um objeto terá

Uma interface não pode ser instanciada

- Não se cria objetos a partir de uma interface

CUIDADO: É uma construção extremamente poderosa em TypeScript!

- Não exploraremos todas as formas de uso das interfaces que a linguagem permite, mas somente o essencial

As interfaces “básicas” podem declarar:

- Propriedades
- Propriedades opcionais (usam o símbolo ? ao final do nome da propriedade)
- Propriedades somente de leitura (usam o modificador `readonly` antes do nome da propriedade)
- Métodos abstratos (não necessita repetir o modificador `abstract`)

Interfaces

Exemplo:

```
interface LabelledValue {  
  label: string;  
}  
  
function printLabel(labelledObj: LabelledValue) {  
  console.log(labelledObj.label);  
}  
  
let myObj = {size: 10, label: "Size 10 Object"};  
printLabel(myObj);
```

Interfaces

Exemplo: propriedades opcionais

```
interface SquareConfig {  
  color?: string;  
  width?: number;  
}  
  
function createSquare(config: SquareConfig): {color: string; area: number} {  
  let newSquare = {color: "white", area: 100};  
  if (config.color) {  
    newSquare.color = config.color;  
  }  
  if (config.width) {  
    newSquare.area = config.width * config.width;  
  }  
  return newSquare;  
}  
  
let mySquare = createSquare({ color: "black" });
```

Interfaces

Exemplo: propriedades somente de leitura

```
interface Point {  
  readonly x: number;  
  readonly y: number;  
}  
  
let p1: Point = { x: 10, y: 20 };  
p1.x = 5; // erro!
```

Interfaces

Exemplo: método em interface

```
interface ClockInterface {  
    currentTime: Date;  
    setTime(d: Date): void;  
}
```

Interfaces e Classes

Interfaces são estruturas abstratas que podem ser utilizadas para separar a especificação do comportamento de um objeto de sua implementação concreta

- Trazem uma especificação sem código de implementação
- Ao contrário das classes, definem um novo tipo sem fornecer a implementação

Dessa forma a interface age como um contrato, o qual define explicitamente quais estruturas uma classe deve obrigatoriamente implementar

Interfaces e Classes

Uma interface pode ser implementada por uma classe

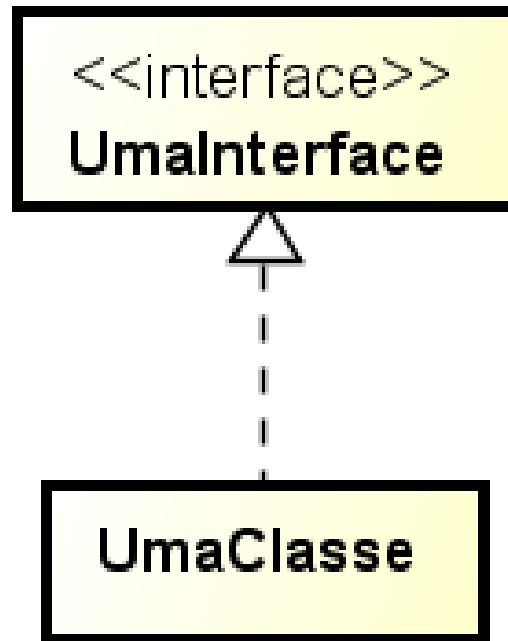
- Uma interface pode ser implementada por diversas classes
 - POLIMORFISMO!
- Uma classe pode implementar diversas interfaces
 - Permite uma classes ser utilizada em diferentes contextos!

```
interface MinhaInterface {  
    ...  
}
```

```
class MinhaClasse implements MinhaInterface {  
    ...  
}
```


Diagrama de Classes UML

Relacionamento de realização de interfaces:



Interfaces e Funções

Interfaces em TypeScript podem ser utilizadas para descrever tipos funcionais

Interface irá declarar uma assinatura para uma função

- Lista de parâmetros e o tipo de retorno

Permite o uso de variáveis cujo tipo é um tipo de função

- Assim, a variável pode ser usada para referenciar qualquer função que esteja de acordo com a assinatura presente na interface

Interfaces e Funções

Exemplo: interfaces e métodos genéricos

```
interface Predicado<T> {  
    (item: T): boolean;  
}
```

```
i => i % 2 === 0
```

Qual o "tipo" dessa
função expressa na
notação lambda?

Interfaces e Funções

Exemplo: interfaces e métodos genéricos

```
interface Predicado<T> {  
    (item: T): boolean;  
}
```

```
function filtrar<T> (array : T[], filtro : Predicado<T>): T[] {  
    let resultado: T[] = [];  
    for(let i = 0; i < array.length; i++) {  
        let valor = array[i];  
        if (filtro(valor)) resultado[resultado.length] = valor;  
    }  
    return resultado;  
}
```

Interfaces e Funções

Exemplo: interfaces e métodos genéricos

```
let pares = filtrar([1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144], i => i % 2 === 0);  
console.log(pares);
```

JSON



JSON

- JSON = JavaScript Object Notation
- Formato textual para serialização de dados
 - É independente de linguagem
 - Muito utilizado para retorno de Serviços Web REST

```
{  
  "productName": "Computer Monitor",  
  "price": "229.00",  
  "specifications": {  
    "size": 22,  
    "type": "LCD",  
    "colors": ["black", "red", "white"]  
  }  
}
```

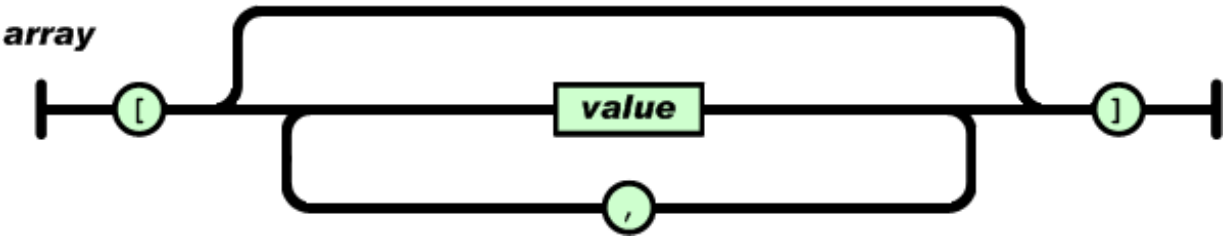
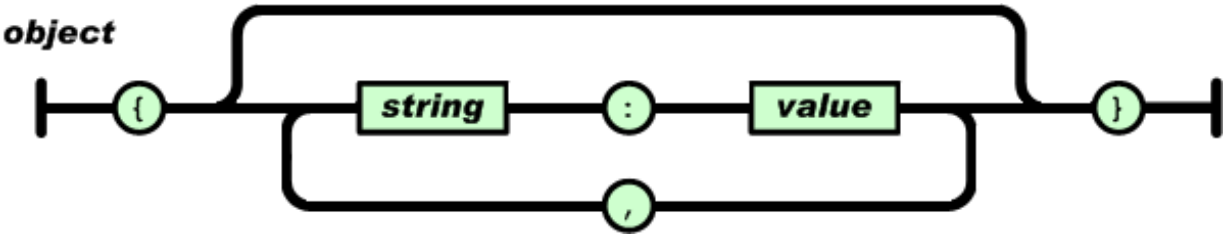
JSON

- Documentação sobre o padrão:
 - <http://www.ecma-international.org/publications/standards/Ecma-404.htm>
 - <https://tools.ietf.org/html/rfc4627>
 - <http://www.json.org/>

JSON

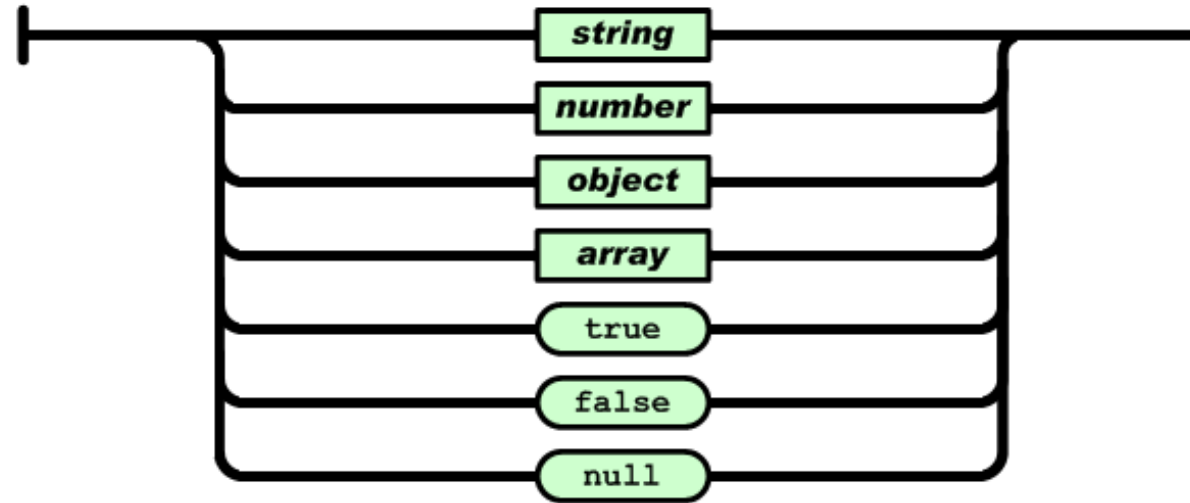
- JSON é capaz de representar:
- Tipos primitivos
 - Strings, números, booleanos, null
- Tipos estruturados
 - Objetos
 - Coleção não-ordenada de zero ou mais pares chave/valor
 - Arranjos
 - Coleção ordenada de zero ou mais valores

JSON

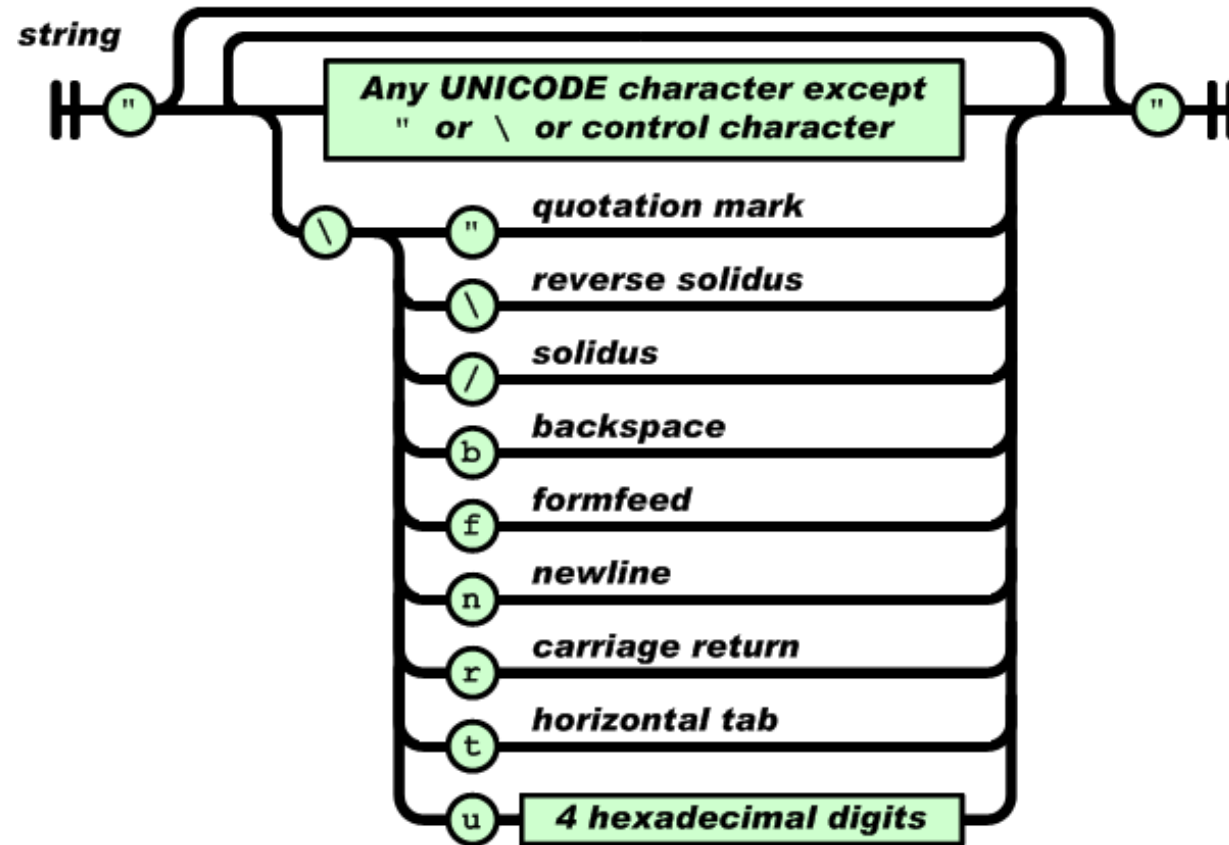


JSON

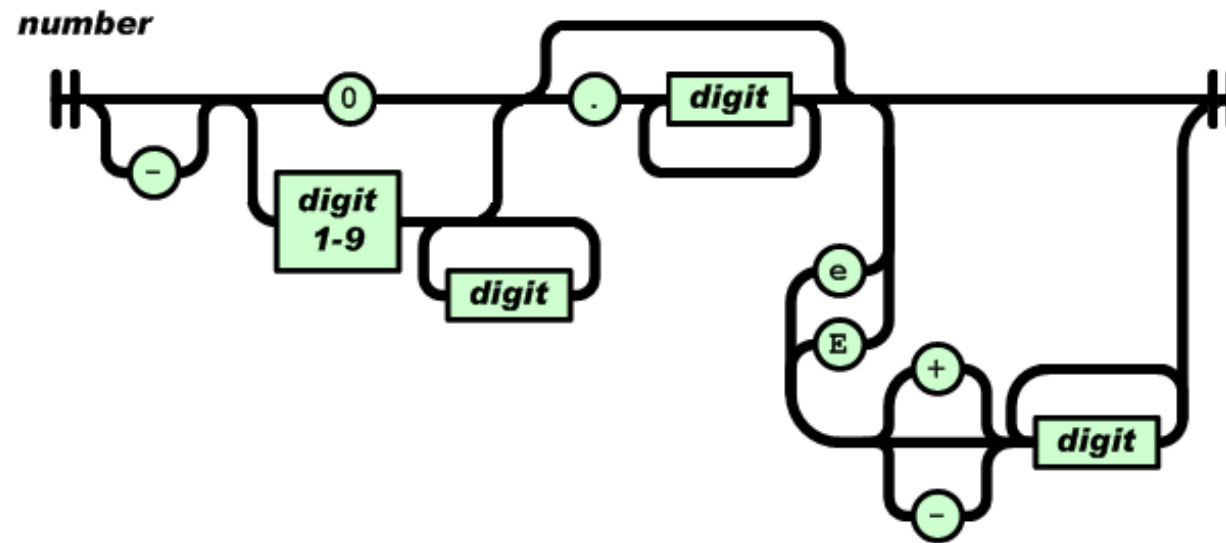
value



JSON



JSON



JSON

- JavaScript provê o método **JSON.stringify** para converter um objeto para o formato JSON
 - Cuidado: não pode existir referências circulares dentro do objeto

```
let student = {  
  name: 'John',  
  age: 30,  
  isAdmin: false,  
  courses: ['html', 'css', 'js'],  
  wife: null  
};  
let json = JSON.stringify(student);  
console.log(json);
```

JSON

- JavaScript provê o método **JSON.parse** para converter uma string no formato JSON em um objeto

```
interface user {  
  name: string;  
  age: number;  
  isAdmin: boolean;  
  friends: number[]  
}
```

```
let json = '{ "name": "John", "age": 35, "isAdmin": false, "friends":  
[0,1,2,3] }';  
let user: user = JSON.parse(json);  
console.log(user.name);
```

JSON

- Cuidado! Não assuma que a conversão suporta qualquer tipo!
 - O exemplo a seguir utiliza um objeto Date, que é serializado como String e, portanto, não é desserializado automaticamente para Date
 - Deve-se utilizar a função *reviver*, que é um dos parâmetros da função JSON.parse

```
interface Person {  
    name: string;  
    birth: Date;  
    city: string;  
}  
let json: string = '{ "name":"John Doe", "birth":"2017-11-  
30T12:00:00.000Z", "city":"Porto Alegre"}';  
let obj: Person = JSON.parse(json, function (key, value) {  
    if (key == 'birth') {  
        return new Date(value);  
    }  
    return value;  
});  
console.log(obj.birth);
```