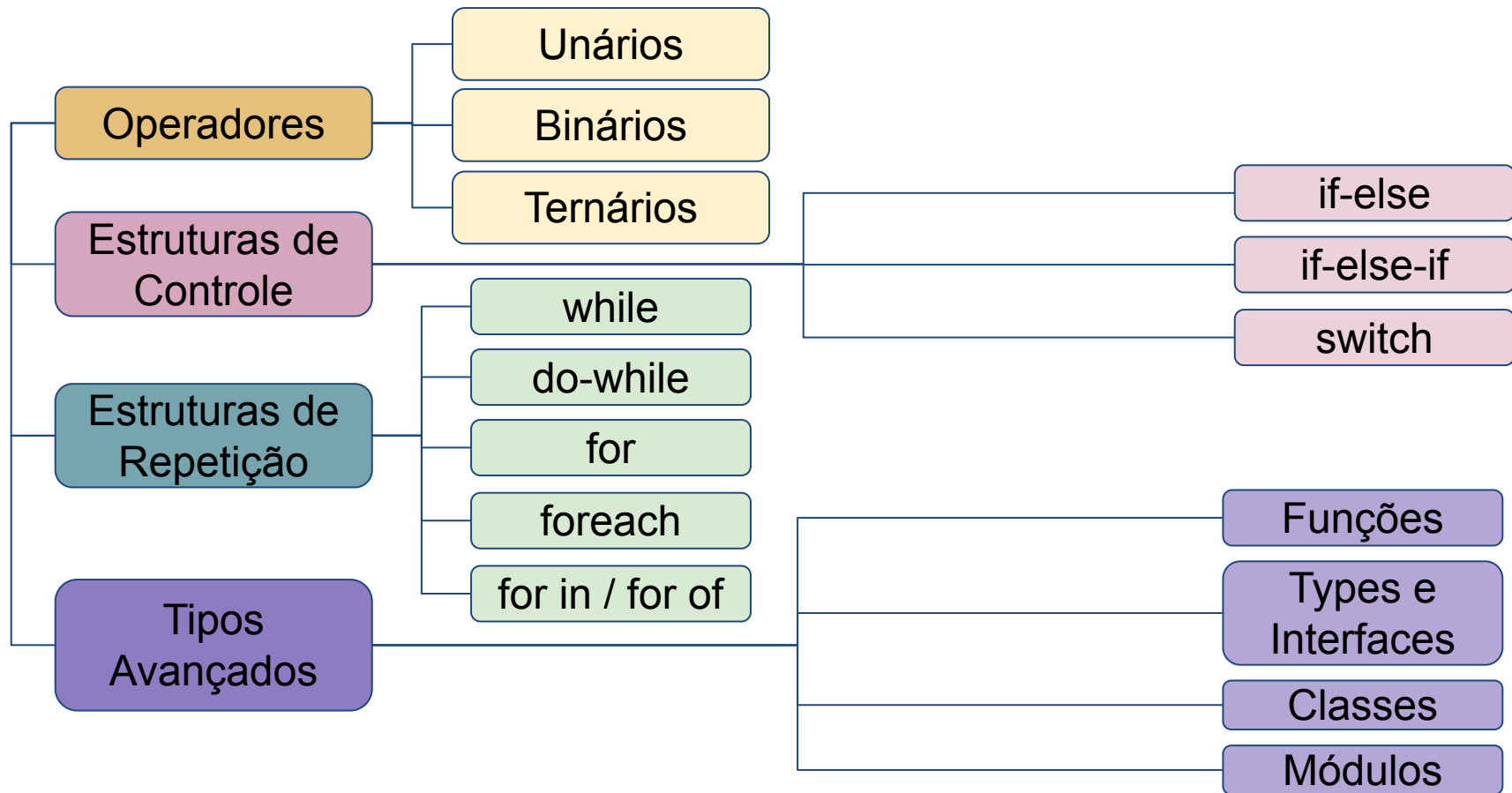




Maiara de Souza Coelho  
Instituto de Computação  
UFAM

# Tópicos da Aula 2



# Operadores

- Unários
- Binários
- Ternários

# Operadores: Unários

Operadores Aritméticos	
++	Incremental, prefixo ou posfixo, semelhante a C.
--	Decremental, prefixo ou posfixo, semelhante a C.
Operadores Bit a Bit	
~	Inverte a cadeia de bits.

# Operadores: Unários

Operadores Especiais	
+	Converte qualquer tipo em number, positivo.
-	Converte qualquer tipo em number, negativo.
!	Transforma qualquer valor em booleano, pode ser combinado consigo mesmo várias vezes. Nega valores Lógicos.
typeof	Pega um operando qualquer e gera o tipo do operando em forma de string.
delete	Deleta um elemento de um array ou propriedade de um objeto, se bem sucedido retorna true, senão, false.

# Operadores: Unários - Exemplos

```
let numero = 100;  
console.log(numero++) //100  
console.log(numero++) //101  
console.log(++numero) //103  
console.log(numero--) //103  
console.log(numero--) //102  
console.log(--numero) //100
```

```
let x = '20';  
let y = '30';  
let z = false;  
console.log(+x); //20  
console.log(+y); //30  
console.log(+z); //0  
console.log(-x); //-20  
console.log(-y); //-30  
console.log(typeof(z)); //"boolean"
```

# Operadores: Binários

Operadores Binários	
*	Multiplicação
/	Divisão
%	Módulo
-	Subtração
**	Exponenciação
+	Soma

# Operadores: Binários

Operadores de Comparação	
>	Retorna true se o operando à esquerda for maior que o da direita, senão, retorna false.
<	Retorna true se o operando à esquerda for menor que o da direita, senão, retorna false.
<=	Retorna true se o operando à esquerda for menor ou igual ao da direita, senão, retorna false.
>=	Retorna true se o operando à esquerda for maior ou igual ao da direita, senão, retorna false.



# Operadores: Binários

Operadores de Comparação	
"=="	Retorna true se o operando da esquerda for igual da direita, senão, retorna false.
!=	Retorna true se o operando da esquerda for diferente do da direita, senão, retorna false.
"==="	Retorna true se o operando da esquerda for igual em valor e em tipo ao da direita, senão, retorna false.
!==	Retorna true se o operando à esquerda for diferente em valor e/ou diferente em tipo, senão, retorna false.

# Operadores: Binários

## Operadores Lógicos

&&

Se os operandos forem expressões, retorna a expressão da esquerda se essa for possível converter para false, senão, a da direita. Caso os operandos sejam valores booleanos, retorna true se ambos forem true, senão, False.

||

Se os operandos forem expressões, retorna a expressão da esquerda se essa for possível converter para true, senão, retorna a direita. Caso os operandos sejam valores booleanos, retorna true se ao menos um for true, senão, False.

# Operadores: Binários

Operadores Relacionais	
in	Retorna true se a propriedade está presente no objeto.
intanceof	Retorna true se o objeto especificado a esquerda for do tipo do objeto a direita.

# Operadores: Binário

## Operadores de Atribuição

=

Atribui o valor a direita ao elemento da esquerda.

+=

Atribui o valor da direita somado com o valor do elemento a esquerda, ao elemento a esquerda.

-=

Atribui o valor da direita subtraído pelo valor do elemento a esquerda, ao elemento a esquerda.

\*=

Atribui o valor da direita multiplicado pelo valor do elemento a esquerda, ao elemento a esquerda.

# Operadores: Binário - Exemplos

```
let x = 8;
```

```
let y = 2;
```

```
let q = 2;
```

```
let z: { p1: number, p2: number } = { p1: 1, p2: 2 };
```

```
let w: { p1: number, p2: number } = { p1: 1, p2: 2 };
```

```
console.log(z === z);
```

```
console.log(y===q);
```

# Operadores: Binário - Exemplos

```
let a = x && y;
```

```
let b = x || y;
```

```
console.log(a); //2 true
```

```
console.log(b); //8 true
```

```
console.log('p3' in w); //false
```

```
console.log(w instanceof Object); //true
```

# Operadores: Ternário

Operadores Condicionais	
exp ? retorno 1 : Retorno 2	Questiona se a exp pode ser atribuída como true, se sim, retorna o retorno 1, senão, o retorno 2.

# Operadores: Ternário - Exemplos

```
let a = { name: "ciclano", idade: 10 };  
let b = a.idade ? a : null;  
console.log(b);
```

```
let perfil = "admin";  
console.log(perfil == "superuser" ? "Super usuário" :  
"Administrador");
```



# Spreads (...)

- Usado para reunir arrays. Faz unions automáticos se os arrays forem diferentes.

```
const musicas = ["Juice", "Shake It Off", "What's Up"];  
const pontuacao = [2, 3, 4];  
const juncao = [...musicas, ...pontuacao]  
console.log(juncao)
```

# Estruturas de Controle

- if-else
- if-else-if
- switch

# Estruturas de Controle: if-else

```
let condition = true;
if (condition) {
  console.log("a variável está com um valor true");
}
else {
  console.log("a variável está com um valor false");
  console.log("a");
}
```

# Estruturas de Controle: if-else-if

```
let perfil = "admin";  
if (perfil == "superuser") {  
  console.log("Super usuário");  
} else if (perfil == "admin") {  
  console.log("Administrador");  
} else {  
  console.log("Usuário comum");  
}
```

# Estruturas de Controle: switch

```
let perfil = "admin";  
switch (perfil) {  
  case "superuser":  
    console.log("Super  
usuário");  
    break;  
  case "manager":  
    console.log("Gerente");  
    break;
```

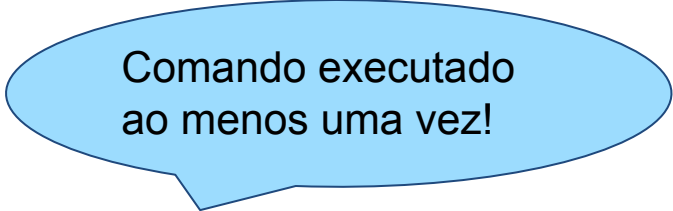
```
  case "admin":  
    console.log("Administrador");  
    break;  
  case "user":  
    console.log("Usuário comum");  
    break;  
  default:  
    console.log("sem perfil");  
    break;  
}
```

# Estruturas de Repetição

- while
- do-while
- for
- foreach
- for in / for of

# Estruturas de Repetição: while / do-while

```
let condicao = true;  
while (condicao) {  
  console.log('Carregando...');  
}
```



Comando executado  
ao menos uma vez!

```
let condicao = false;  
{  
  console.log('Carregando...');  
} while (condicao);
```

# Estruturas de Repetição: for / forEach

```
const languages = [ "C#", "Java", "JavaScript"];  
for (let i = 0; i < languages.length; i++) {  
    console .log(languages[i]);  
}
```

```
const languages = [ "C#", "Java", "JavaScript"];  
languages.forEach(element => {  
    console.log(element);  
});
```



# Estruturas de Repetição: for of / for in

```
const vetor = ["O", "que", "eu", "to", "fazendo", "da",  
"minha", "vida"];  
for (let value of vetor) {  
  console.log(value);  
}
```

```
let obj = { nome: "cão brabo", atack: 50, defesa:  
20, habilidade: "morder" }  
for (let propriedade in obj) {  
  console.log(propriedade);  
}
```

# Funções

- Descrever parâmetros de Callback (parâmetros que são chamados como funções)

```
const musicas = ["Juice", "Shake It Off", "What's Up"];  
function executarMusicas(getSongAt: (index:number) => string) {  
    for(let i=0; i<musicas.length;i++){  
        console.log(getMusica(i));  
    }  
}  
  
function getMusica(index:number): string{  
    return `${musicas[index]}`;  
}  
  
executarMusicas(getMusica);
```

# Funções

- Exemplo de capacidade de atribuição do TypeScript. 3 níveis de detalhes.

```
const musicas = ["Juice", "Shake It Off", "What's Up"];

function executarMusicas(getSongAt: (index:number) => string) {
    for(let i=0; i<musicas.length;i++){
        console.log(getMusica(i));
    }
}

function longMusica(musica:string): string{
    return `${musicas}`;
}

executarMusicas(longMusica);
```

# Funções

obrigatório

- void: funções que não aceitam instrução de retorno ("return")

? opcional

```
function anunciarMusica(musica: string, cantor?:string):void{  
    console.log(`Música: ${musica}`);  
    if(cantor){  
        console.log(`Cantor: ${cantor}`);  
    }  
}  
  
anunciarMusica("Musica");  
anunciarMusica("Música", undefined);  
anunciarMusica("Música", "Cantor");
```

cantor: string|undefined

# Funções: Retorno

- Todos os tipos, tanto primitivos quanto compostos, podem ser usados como retorno e/ou parâmetro de funções, inclusive com uniões.

```
function tamanhoNome(cantor:string) : number{  
    return cantor.length;  
}  
  
console.log("Tamanho: " +  
tamanhoNome("Música"));
```

# Funções: Retorno

```
function tamanhoNome(cantor:string): boolean{  
    if(cantor.length == 6){  
        return true;  
    }  
    return false;  
}  
  
console.log("Tamanho 6?: " +  
tamanhoNome("Música"));
```

# Funções: Retorno

```
function tamanhoNome(cantor:string) : boolean | number {  
    if(cantor.length == 6) {  
        return true;  
    }  
    return cantor.length;  
}  
console.log("Tamanho 6?: " + tamanhoNome("Music"));
```

# Funções: Parâmetro Opcional

- Declarações de variáveis que recebem o tipo função.

```
let anuncioMusica: (musica: string, cantor?:string) => void;  
anuncioMusica = (musica: string, cantor?:string) => {  
    console.log(`Música: ${musica}`);  
    if(cantor){  
        console.log(`Cantor: ${cantor}`);  
    }  
}  
  
anuncioMusica("Musica");
```



# Funções: Parâmetro Padrão

Padrão: recebe um valor no parâmetro

```
function pontuarMusica(musica: string, pontuacao=0) {  
    console.log(`Música: ${musica}`);  
    console.log(`Pontuacao: ${pontuacao}`);  
}  
  
pontuarMusica("Musica");  
pontuarMusica("Música", 5);
```

# Funções: Parâmetros REST

REST: recebe um array de algum tipo ou união de tipos. Pode receber 0 ou mais argumentos

```
function tocarMusicas(cantor: string, ...musicas: string[]) {  
  for(const musica of musicas){  
    console.log(`Cantor: ${cantor}`);  
    console.log(`Música: ${musica}`);  
  }  
}  
  
tocarMusicas("Cantor", "música1", "música2", "música3");
```

# Funções: Never

- Não apenas não retornam nada, como simplesmente não retornam. Servem para lançar exceções e executar loop infinito (tem que ser intencional!)

```
function fail(message:string) : never{  
    throw new Error(`Falha ${message}`);  
}
```

# Funções: Never

```
function workWithUnsafeParam(param: unknown) {  
    if (typeof param !== "string") {  
        fail(`param should be a string, not ${typeof param}`);  
    }  
    param.toUpperCase();  
}  
workWithUnsafeParam(123);
```

# Alias de Função

```
type StringToNumber = (input:string) => number;
```

```
let stringToNumber: StringToNumber;
```

```
stringToNumber = (input) => input.length;
```

```
// stringToNumber = (input) => input.toUpperCase();
```

```
console.log(stringToNumber("Maiara"));
```



Função Seta gorda

# Modificador de Tipos: Assertion

- Semelhante ao cast em outras linguagens.
  - `let codigoNumber: number = codigoAny as number;`
  - `let codigoNumber: number = <number>codigoAny;`

```
function typeAssertions(codigoAny:any) {  
    let codigoNumber: number = codigoAny as number;  
    return codigoNumber * 10;  
}  
  
console.log(typeAssertions("10"));
```

# Interfaces

```
type Poet = {  
  nome?: string,  
  idade: number  
};  
  
let valueLater: Poet;  
  
valueLater = {  
  nome: "Antônio",  
  idade: 30  
}  
  
console.log(valueLater);
```

```
interface Poet {  
  nome?: string,  
  idade: number  
}  
  
let valueLater: Poet;  
  
valueLater = {  
  idade: 30  
}  
  
console.log(valueLater);
```

# Type X Interface

- Diferenças
  - Interfaces são bem úteis para verificação do tipo da estrutura de declarações de classe, o que o type não faz.
  - Com interfaces não dá para fazer unions com tipos primitivos.



# Interface: Funções e Métodos

```
interface
CursoProps{
    id: string;
    nome: string;
    preco: number;

    mostrarPromocao:
(preco:number) =>
void;
}
```

```
const curso: CursoProps = {
    id: "1",
    nome: "Curso Typescript",
    mostrarPromocao: (preco: number):
void => {
        console.log('Preço total: ' +
preco);
    }
}

console.log(curso);
console.log(curso.mostrarPromocao(350));
```

# Extensões de Interface

```
interface JogoProps {  
  id: string;  
  nome: string;  
  descricao: string;  
  plataforma: string[];  
}
```

```
const left4dead: JogoProps = {  
  id: "123",  
  nome: "Lef 4 Dead 2",  
  descricao: "Jogo de ação e tiro",  
  plataforma: ["PS5", "PC"]  
}
```

```
interface DLC extends JogoProps {  
  jogoOriginal: JogoProps;  
  novoConteudo: string[];  
}
```

# Extensões de Interface

```
const left4DeadDLC: DLC = {  
  id: "90",  
  nome: "Left 4 Dead - Novos Mapas",  
  descricao: "4 novos mapas para jogar online",  
  plataforma: ["PS5", "PC"],  
  novoConteudo: ["Modo Coop", "Mais 5 horas de jogo",  
    "Medalhas"],  
  jogoOriginal: left4dead  
}  
console.log(left4DeadDLC);
```

# POO

- Classe: estrutura que abstrai um conjunto de objetos que possuem características e comportamentos similares. Descrevendo os serviços oferecidos e quais informações podem armazenar.
- Objeto: representação de algo no mundo real.
- Exemplo:
  - Classe Produto
  - Atributos: nome, categoria, preco, status
  - Instância a classe produto: shampoo, “Cuidados Pessoais”, 30, “ESGOTADO”.

# Classes e seus Atributos

```
type Status = "EM_ESTOQUE" |  
"ESGOTADO";
```

```
class Produto{
```

Classe

```
  nome: string;
```

```
  categoria: string;
```

```
  preco: number;
```

```
  status: Status;
```

Atributos

```
  constructor(nome: string, categoria:  
string, preco:number, status: Status){  
    this.nome = nome;  
    this.categoria = categoria;  
    this.preco = preco;  
    this.status = status;  
  }  
}
```

Construtor

```
const novoProduto = new  
Produto("Shampoo", "Cuidados Pessoais",  
30, "ESGOTADO");  
console.log(novoProduto.nome);
```

Objeto

# Classes: Funções em classes - Métodos

```
adicionar(): void {  
    this.mudarStatus("EM_ESTOQUE");  
    console.log(`Produto ${this.nome}, categoria:  
${this.categoria}`)  
}  
  
mudarStatus(status: Status): void {  
    if(status === "EM_ESTOQUE") {  
        this.status="EM_ESTOQUE";  
    }else{  
        this.status="ESGOTADO";  
    }  
}
```

# Classes: Funções em classes - Métodos

Instanciação de um novo Objeto Produto

```
const novoProduto = new Produto("Shampoo",  
  "Cuidados Pessoais", 30, "ESGOTADO");  
novoProduto.adicionar();
```

```
console.log(novoProduto.nome);
```

Atributo

```
console.log(novoProduto.status);
```

Método

# Classes: Modificadores de Acesso

Acessível em	public	protected	private
classe	Sim	Sim	Sim
classes filhas	Sim	Sim	Não
Classes instanciadas	Sim	Não	Não



# Classes: Modificadores de Acesso

- Adicionando segurança à classe produto.

```
type Status = "EM_ESTOQUE" | "ESGOTADO";  
class Produto{  
    private nome: string;  
    private categoria: string;  
    private preco: number;  
    private status: Status;
```

# Classes: Modificadores de Acesso

- Assim tais atributos poderão somente ser modificados através dos métodos da própria classe.
- gets e sets são palavras reservadas que podem ser usadas para esse fim.

```
get getNome() {  
    return this.nome;  
}  
  
get getCategoria() {  
    return this.categoria;  
}
```

```
set setNome(nome: string) {  
    this.nome=nome;  
}  
  
set setCategoria(categoria:  
string) {  
    this.categoria=categoria;  
}
```

# Classes: Herança

- Os filhos herdam todos os atributos e métodos da classe pai, além disso possuem seus próprios atributos.
- Permite reutilizar código sem duplicá-lo.
- Palavra reservada “`extends`”.
- Função “`super ( )`” no construtor da classe filha.
- Exemplos:
  - Conta (saldo, numero): PJ, Pessoa Física.
  - Produto (nome e preco): Infantil (faixaEtaria).
  - Usuario (email e senha): Funcionario (dataAdmissao), Cliente (dataUltimaCompra).
  - Midia (titulo, tamanho, duracao, formato): Video (resolucao), Audio (frequencia).

# Classes: Herança

```
class ProdutoInfantil extends Produto{  
    private _faixa_etaria: number;  
  
    constructor(nome: string, categoria: string, preco:number, status:  
Status, faixa_etaria: number){  
        super(nome, categoria, preco, status);  
        this._faixa_etaria = faixa_etaria;  
    }  
}  
  
const novoProdutoInfantil = new ProdutoInfantil("Shampoo", "Cuidados  
Pessoais", 30, "ESGOTADO", 12)
```

# Classes Abstratas

- As classes abstratas não permitem realizar qualquer tipo de instância, elas são utilizadas como modelos para outras classes, que são conhecidas como classes concretas.

```
type DadosConta = {  
  nome: string;  
  numero: string;  
  endereco?: string;  
}
```

```
abstract class ContaBanco{  
  abstract abrirConta(dados:  
    DadosConta): boolean;  
}
```

# Herdando de Classes Abstratas

```
class PessoaFisica extends
ContaBanco{
    abrirConta(dados: DadosConta):
boolean {
    console.log(`Nova conta
P.Fisica criada com sucesso
${dados.nome}`)
    return true;
}
}
```

```
class PessoaJuridica extends
ContaBanco{
    abrirConta(dados:
DadosConta): boolean {
    console.log(`Nova conta
P.Juridica criada com sucesso
${dados.nome}`)
    return true;
}
}
```

# Usando Classes Concretas

```
const joana = new  
PessoaFisica();
```

```
joana.abrirConta({  
  nome: "Joana Silva",  
  numero: "1029-x",  
  endereco: "Rua 15, bairro  
centro"  
})
```

```
const sujeitoprogramador = new  
PessoaJuridica();
```

```
sujeitoprogramador.abrirConta({  
  nome: "Matheus Fraga",  
  numero: "90201-x",  
  endereco: "Avenida dez,  
centro"  
})
```

# Classes & Interfaces

- Dão forma aos dados ou agem como contrato

```
interface Learner{  
    nome: string;  
    estudar(horas:number) : void;  
}  
  
class Estudante implements Learner{  
    nome;  
    constructor(nome: string) {  
        this.nome = nome;  
    }  
}
```



# Classes & Interfaces

```
    estudar(horas: number) {  
        for(let i=0; i<horas; i++){  
            console.log("...estudando...", i);  
        }  
    }  
}  
  
let estudante = new Estudante("Nome");  
estudante.estudar(10);
```

# Exercício 5

- Melhore alguns aspectos das nossas classes Produto e ProdutoInfantil:
  - O produto deve ter um código gerado pela classe e não passado para ela no momento de criação de um novo objeto.
  - Crie métodos gets e sets para todos os atributos das classes.
  - Faça uma função dentro da classe que verifique se a faixa etária é realmente válida, ela deve ser chamada na criação do produto. Produtos infantis devem possuir faixa etária até os 12.
  - Os métodos da classe pai devem ficar protegidos para que somente as classes filhas possam acessá-los.

# Exercício 6

- Crie uma classe para representar as contas de um banco.
  - As contas devem ser do tipo PJ e Pessoa Física. Defina ao menos três atributos para cada classe. A classe pai deve ser uma classe abstrata.
  - A conta pertence à um cliente e o cliente tem seus atributos também, faça uma classe para representá-lo.
  - Faça uma função para adicionar a conta do cliente.
  - Faça validações de saldo a cada saque ou depósito da conta.
  - Armazene as contas de cada cliente em um array e mostre todos os clientes da agência.

Por hoje é só, pessoal!

Obrigada

Dúvidas: Slack

[maiara@icomp.ufam.edu.br](mailto:maiara@icomp.ufam.edu.br)

github: [mayara-msc@hotmail.com](mailto:mayara-msc@hotmail.com)