TypeScript

TypeScript = um JavaScript melhorado, tem tudo que o java tem e um pouco mais, ajuda o programador a não errar.
Instalação = pra instalar o typescript basta digitar npm install typescript@versão --save-dev no terminal.
-- save-dev = instala localmente, apenas para o desenvolvimento do código. Não é compartilhado com nada mais.
Compilação = o navegador não entende TypeScript, apenas JavaScript. Por isso, o código pega o que escrevemos em type e compila para java, e então manda pro navegador.
Protected = como o private e o public, é um private que permite os elementos filhos acessarem as características do pai.

Código:

```
//Tsconfig.json = arquivo pra fazer a compilação de Java pra Type. Escrever dentro dele:
{"compilerOptions":
     {"outDir":"arquivo_envio/js", //Diz para onde enviar os arquivos compilados, os js.
    "target":"ES6", //Diz pra qual versão compilar, nesse caso pra ES6.
    "noEmiteOnError": true, //Não permite compilar os arquivos enquanto houver erros.
      "noImplicitAny": true, //Diz pra não colocar implicitamente o tipo any nos atributos, fazer a gente definir.
       "removeComments": true, //Remove qualquer comentário feito no código, na compilação pra Java.
     "strictNullChecks": true} //Diz pra não colocar implicitamente o tipo null nos atributos, fazer a gente definir.
      "include": ["arquivo_raiz/**/*"]} //Diz qual pasta converter, nesse caso a arquivo_raiz e todas suas subpastas.
//E escrever no script do package.json:
"compile": "tsc", //Faz as conversões de type para java.
"watch": "tsc -w" //Faz as mudanças acontecerem simultaneamente na compilação.

class nome_controller //Cria um controller, ponte entre usuário e código. Pega o que o usuário incluir e guarda essas informações.
    {private nome_atributo; //Cria um atributo que não pode ser modificado.
    constructor()
        {this.nome_atributo = document.querySelector("#id_do_input")}
    nome_função() //Faz algo com essas informações, no caso mostra o valor.
        {const controller = new controller
            (this.nome_atributo.value)}}

class nome_controller //Simplifica a primeira parte da classe de cima.
    {constructor
        (private _nome_atributo){}

const controller = new nome_controller(); //Cria novos controllers, de acordo com a
```

```
classe.
const form = document.querySelector("#id_do_form"); //Conecta com o formulário.
form.addEventListener("submit", event => //Pede para escutar as mudanças na página.
      {event.preventDefault(); //Impede que a página recarregue antes de enviar os
dados do formulário.
      controller.nome_função()}) //Executa a função quando há mudanças, nesse caso
mostra o valor.

const exp = /caractere_original/; //Pega esses caracteres no atributo pedido e
substitui pelo outro.
const nova exp = new atributo(this.atributo.value.replace(exp,
"caractere_substituto"))

class controllers //Pega todos os inputs que entrarem e coloca eles em uma lista, a
lista de controllers.
    {private controllers: Array<controller> = []; //Define as propriedades da lista.
nome_função(controller: controller) //Empurra esses inputs pra dentro da lista.
    {this.controllers.push(controller)}
função(): ReadonlyArray<controller> //Retorna o resultado em uma lista que só pode
ser lida, jamais modificada.
    {return this.controllers}}

class controller //Cria uma classe de controller com o atributo bem definido e apenas
leitura, simplificando o anterior.
    {constructor
        (public readonly nome_atributo: tipo_atributo){}

class nome_class //Cria uma classe com um template que retorna o que quisermos.
    private elemento: HTMLElement;
     constructor(seletor: tipo_atributo) //E que incrementa as novas entradas de html
ao conteúdo da página.
        {this.elemento = document.querySelector(seletor)}
    {template(): tipo_atributo
        {return o_que_quiser}
    update(): void //Transformando essas entradas em elementos do dom.
        {this.elemento.innerHTML = this.template()}} //

enum nome //Cria uma enumeration, como um aray em que os itens podem ser apenas
chamados, mas não modificados ou reatribuídos.
        {item 1,
        item 2,
        item 3}

function nome_função () //Cria um decorator, que modifica o comportamento de outra
função, retornando no lugar o acontecimento.
```

```
    {return function
        (target: any,
        propertyKey: string,
        descriptor: PropertyDescriptor)
    {const metodo = descriptor.value;
    descriptor.value = function (...args:any[])
        {let retorno = metodo.apply(this, args);
        acontecimento}
    return descriptor}}

interface nome_interface //Define propriedades com os tipos já definidos. Cria opções, ao chamar pelo nome da interface.
    {propriedade_1: tipo_1;
    propriedade_2: tipo_2;
    propriedade_3: tipo_3;}
```