Universidade Federal do Ceará - Campus de Quixadá

Programação para Design

Comandos básicos do TypeScript

Prof. Anderson Lemos

Introdução

- TypeScript é um superconjunto do JavaScript.
- Assim, os comandos parecidos com a maioria das linguagens derivadas de C
  - Como C++, Java, C#, etc.
  - Inclusive, toda instrução tem que terminar com ponto e vírgula, como em C.
  - E também como em C, escopo dos blocos são delimitados por abre chaves '**{**' e fecha chaves '**}**'.
    - Diferentemente do Python que usa indentação para delimitar escopos.

Introdução

- TypeScript tem:
  - Comentários.
  - Sentenças de atribuição.
  - Variáveis tipadas.
  - Comandos de entrada e saída.
  - Operadores aritméticos, relacionais e lógicos.
  - Estruturas de decisão e de repetição.
  - Funções.
  - Inferência de tipos.

Execução do TypeScript

- Após o projeto configurado (mostrado no Slide ***01. TypeScript***), usaremos a linha de comando para compilar e executar os arquivos TS (TypeScript).
  - Prompt de comando no Windows.
  - Terminal no Linux.
  - Terminal (console) do VS Code.
  - Obviamente, a linha de comando deve estar aberta na pasta do projeto.
- Compilaremos e executaremos utilizando ECMAScript 6.
- Compilar o arquivo TS (transformando-o em um arquivo JavaScript).
  - `tsc -t ES6 Principal.ts`
  - Esse deve ser o arquivo principal da sua aplicação, o que está indicado no index.js.
  - São gerados automaticamente arquivos JS (JavaScript) com os mesmos nomes dos arquivos compilados.
- Executa-se o arquivo ***index.js*** utilizando Node.js.
  - `node index.js`

Comentários

- Podem ser feitos em apenas uma linha, utilizando //.
- Podem ser feitos em blocos delimitados por /* e */.

```
// comentário em uma linha

/*
bloco de comentários.
pode haver várias linhas.
inclusive só uma.
*/
```

Sentenças de atribuição

- Uma sentença de atribuição ocorre quando atribuímos um valor a uma variável.
- O operador de atribuição é o sinal de igualdade (**=**).
  - Para não causar ambiguidade a comparação é feita com '**==**'.
- Atribuição em TypeScript é feita copiando os valores.
- Se são objetos (conceito que veremos quando estivermos vendo OO), as cópias são feitas "por referência".
  - Se copia o valor do endereço de memória do objeto.

Variáveis

- Representam uma abstração de uma célula de memória do computador e do valor que está contido na célula.
- Variáveis são como caixinhas que guardam valores que podem mudar durante a execução do programa.
- Em TypeScript declaramos variáveis de duas formas:
  - **var**
  - **let**

Variáveis

- ***var***
  - Modo tradicional do JavaScript.

```
var a = 10;
```

Variáveis

- ***var***
  - Modo tradicional do JavaScript.
  - **Escopo da variável vaza.**

```
function f(shouldInitialize : boolean){
    if(shouldInitialize){
        var x = 10;
    }
    return x;
}

f(true); // retorna 10
f(false); // retorna undefined
```

Variáveis

- ***var***
  - Modo tradicional do JavaScript.
  - **Escopo da variável vaza.**
  - **Permite declarar duas vezes.**

```
var a;
var a;

// roda sem problemas
```

Variáveis

- ***let***
  - Nova característica do ECMA 6/2015.

```
let a = 10;
```

Variáveis

- ***let***
  - Nova característica do ECMA 6/2015.
  - **Escopo da variável funciona como o esperado - escopo de bloco.**

```
function f(shouldInitialize : boolean){
    if(shouldInitialize){
        let x = 10;
    }
    return x;
    // Error: cannot find name 'x'.
    // Erro indica que x não está declarado nesse escopo.
}
```

Variáveis

- ***let***
  - Nova característica do ECMA 6/2015.
  - **Escopo da variável funciona como o esperado - escopo de bloco.**
  - **Não podemos declarar duas vezes.**

```
let a;
let a;
// Error: Cannot redeclare block-scoped variable 'a'.
```

Variáveis

- ***let***
  - Nova característica do ECMA 6/2015.
  - **Escopo da variável funciona como o esperado - escopo de bloco.**
  - **Não podemos declarar duas vezes.**
  - Em TypeScript é sempre preferível usar o ***let***.

Variáveis

- Variáveis podem assumir valores especiais como:
  - *null*: indica que o valor da variável é nulo.
  - *undefined*: indica que o valor da variável é indefinido.
- Quando queremos uma variável com valor fixo (que não mude), não faz sentido o termo “variável”.
  - Criamos então uma constante.
  - Para isso, usamos a palavra reservada ***const***.

```
const a = 10;
a = 11;
// Error: Cannot assign to 'a' because it is a constant
// Ou seja, não é possível mudar o valor de 'a' porque ela é uma constante
```

Tipagem

- É possível definir tipo de variáveis.
  - Caso variável receba valor do tipo errado, compilador acusa erro.
  - Por padrão, sempre declaramos os tipos das variáveis.
- Existem muitos tipos no TypeScript.
  - Tipos primitivos:
    - number (inteiros, float).
    - string (cadeias de caracteres).
    - booleanos (só pode ser **true** ou **false**).
  - Tipos especiais:
    - any
    - void
  - Coleções (que veremos mais adiante):
    - tuplas, listas (arrays), conjuntos (sets) e mapas (maps).

Tipagem

- Exemplos de tipos primitivos e listas:

```
// Declarando tipo primitivos
let isDone: boolean = false;
let decimal: number = 6;
let fullName: string = "Tony Stark";

// modos de declarar lista (inicializadas ou vazias)
let list1: number[] = [1, 2, 3];
let list2: Array<number> = [1, 2, 3];
let list3: Array<number> = new Array<number>();
let list4: number[] = [];
```

Tipagem

- Tipo ***any***
  - Descrever qualquer tipo de variável.
  - Tipo do valor pode mudar dinamicamente.
  - Em POO, por padrão, não costumamos utilizar o tipo ***any***.

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false;
```

- Tipo ***void***
  - Descreve a ausência de valor.
  - Muito usado para indicar que função retorna nada.

```
function warnUser(): void {
    console.log("This is my warning message");
}
```

Entrada e Saída

- Comandos de entrada de dados:
  - readline.question
    - Serve para ler dados digitados na linha de comando.
    - Igual ao *raw_input* do Python 2 ou *input* do Python 3.
    - Necessita do pacote *readline-sync* do node.
- Comando de saída de dados:
  - console.log
    - Serve para imprimir dados na linha de comando.
  - alert
    - Serve para abrir uma janela com informações no navegador.

Operadores aritméticos

- Operadores matemáticos:
  - **+** adição
  - **-** subtração
  - ***** multiplicação
  - **/** divisão
  - **%** módulo (resto)
  - ****** potência
- Atribuição com operação:
  - **+=**, **-=**, ***=**, **/=**, **%=**
    - Por exemplo `a += 10` é o mesmo que `a = a + 10`.
    - O mesmo serve para os outros operadores e atribuição com operação.

Operadores aritméticos

- Precedência
  - De acordo com as regras matemáticas.
    - `let x : number = 4 + 2*3 + 9/3;`
  - Use parênteses para definir a precedências.
    - `let x : number = (4+2)*(3+9)/3;`
  - Ordem de precedência (da maior para a menor):
    - Expressões entre parênteses.
    - Potência.
    - Divisões e multiplicações.
    - Adições e subtrações.
  - Expressões são analisadas da esquerda para direita.

Operadores aritméticos

- Sobrecarga de operadores:
  - O operador de adição ‘+’, quando aplicado a duas strings, concatena (junta) as duas.
  - Qualquer coisa, quando concatenada a uma string, é convertida em string e depois é feita a concatenação.

```
let a : string = "UFC ";
let b : string = "Quixadá";
let c : string = a + b;
console.log(c);
// c vale "UFC Quixadá", que é o valor impresso
```

Operadores aritméticos

- Incremento e decremento (em uma unidade)
  - Há várias formas de fazê-lo.
  - Incremento

```
let x = 10;
x = x + 1;
x += 1;
x++; // pós incremento: faz a instrução, depois o incremento.
++x; // pré incremento: faz o incremento, depois a instrução.
```

  - Decremento

```
let x = 10;
x = x - 1;
x -= 1;
x--; // pós decremento: faz a instrução, depois o decremento.
--x; // pré decremento: faz o decremento, depois a instrução.
```

Operadores relacionais

- Operadores relacionais:
  - **==** igual
  - **!=** diferente
  - **<** menor
  - **<=** menor ou igual
  - **>** maior que
  - **>=** maior ou igual
- Resultam em true ou false.

Operadores lógicos

- Operadores lógicos:
  - **&&** e
  - || ou
  - **!** não
- Implementam a funcionalidade de curto-circuito (short-circuit).

Operador ternário

- Operador Ternário:
  - Retorna um valor ou outro, dependendo do resultado da expressão booleana.
  - `variável = expressão ? retorno_true : retorno_false;`
- Exemplo:
  - `let x : number = (y!=0) ? 50 : 500;`
  - Qual o valor de *x*?
    - Depende de *y*.

Controle de execução

- Seleção
  - if-else
  - switch-case
- Repetição
  - for
  - while
  - do-while
- Desvios (só funcionam em estruturas de repetição):
  - continue
  - break

Estruturas de decisão

- if-else
  - Necessita de chaves para delimitar o escopo se houver mais de uma instrução no mesmo.

```
if(expressaoBooleana)
    // instrução_simples;
else {
    // instruções
}
```

```
if(expressaoBooleana) {
    // instruções
} else if(expressaoBooleana){
    // não tem elif como python. Aqui é else if
    // instruções
} else {
    // instruções
}
```

Estruturas de decisão

- switch-case
  - Testa qual caso é igual a expressão.
  - Se não for igual a nenhum caso, entra no *default*.
  - Pode ter quantos *cases* o programador quiser.
  - Necessita de um break para delimitar o escopo do case.

```
switch (expressao) {
    case valor1:
        // code...
        break;
    case valor2:
        // code...
        break;
    case valor3:
        // code...
        break;
    default:
        // code...
}
```

Estruturas de repetição

- while e do-while
  - Necessitam de chaves para delimitar o escopo se houver mais de uma instrução no mesmo.
  - A diferença entre um e outro é que o do-while sempre executa uma vez antes de testar se a condição é verdadeira.

```
while(expressaoBooleana){
    // instruções
}
```

```
do{
    // instruções
} while(expressaoBooleana);
```

Estruturas de repetição

- for
  - Necessitam de chaves para delimitar o escopo se houver mais de uma instrução no mesmo.
  - Diferente do for do Python.
  - Tem três campos, separados por ponto e vírgula:
    - inicialização: acontece apenas uma vez, antes da primeira iteração do laço.
    - condição: condição que verifica se o laço deve entrar na iteração (como a condição do while).
    - incremento: acontece ao fim de cada iteração, após todas as instruções do escopo do laço.

```
for(let i=0; i<10; i++){
    // instruções
}
```

Estruturas de repetição

- for .. in
  - Necessitam de chaves para delimitar o escopo se houver mais de uma instrução no mesmo.
  - Parecido com for do Python.
  - Percorre os índices dos elementos de uma lista.
    - O for do python percorre os elementos.
    - Similar a usar um *for in range*

```
let lista : Array<number> = [5, 8, 9, 2];
for(let i in lista){
    console.log(i);
}
// imprime 0 1 2 3, cada valor em uma linha.
```

Estruturas de repetição

- for .. of
  - Necessitam de chaves para delimitar o escopo se houver mais de uma instrução no mesmo.
  - Igual ao for do Python.
  - Percorre os elementos de uma lista

```
let lista : Array<number> = [5, 8, 9, 2];
for(let i of lista){
    console.log(i);
}
// imprime 5 8 9 2, cada valor em uma linha.
```

Desvios

- Utilizados dentro do escopo de laços de repetição.
- Servem para desviar o fluxo do programa.
- Dois comandos:
  - **break**
    - Quebra o laço mais interno que o contém.
    - O laço termina sua execução, não importa em qual iteração esteja.
  - **continue**
    - Quebra apenas a iteração do laço.
    - Se a iteração quebrada não for a última, a próxima iteração é iniciada.

Funções

- Função nomeada ECMAScript 5

```
function sum(x, y){
    return x+y;
}
```

- Função anônima ECMAScript 5 (atribuída a uma variável)

```
let mySum = function(x, y){
    return x + y;
}
```

Funções

- Função Arrow ECMAScript 6/2015

```
let mySum = (x, y) => { return x+y; };
```

- Função Arrow ECMAScript 6/2015
  - Uma única expressão, a qual o retorno da mesma é retornado pela função
  - Útil para o paradigma funcional

```
let mySum = (x, y) => (x+y);
```

Funções

- Funções com parâmetros e retorno tipados
  - Por padrão, sempre colocamos os tipos dos parâmetros e do retorno.
  - Mais confiável.

```
// tipo de cada parâmetro colocado após o parâmetro
// tipo de retorno depois do ')' dos parâmetros
function sum(x : number, y : number) : number {
    return x+y;
}

// função anônima com tipos:
let mySum = function(x : number, y : number) : number {
    return x+y;
}
```

Funções

- Funções com parâmetros e retorno tipados
  - Inferência de tipos
    - Os tipos dos parâmetros e do retorno são verificados e, se não correspondem, ocorre um erro em tempo de transpilação.

```
function teste(texto : string) : void {}
let numero : number = 1;
teste(numero); // dá erro
```

Perguntas?