Linguagens e Ambientes de Programação (2018/2019)

Teórica 18 (15/mai/2019)

Linguagens de scripting. A linguagem Bash.

Elementos da linguagem JavaScript.

Linguagens de scripting

As **linguagens de programação clássicas** são concebidas para criar estruturas de dados e algoritmos a partir do zero, usando os elementos primitivos da linguagem. Exemplos: OCaml, C, C++, Java.

Em constaste, nas **linguagens de scripting** assume-se que já existe uma coleção, pronta a ser usada, de componentes escritas noutras linguagens. As linguagens de scripting são concebidas para permitir ligar e organizar componentes existentes **de forma simples, expedita e flexível**. Exemplos: JavaScript, Bash, Tcl, Perl, PHP, Ruby, Python.

Por exemplo, quando se usa JavaScript para adicionar interatividade a uma página Web, as componentes em causa, são os objetos DOM que representam essa página Web.

Chama-se **script** a um programa escrito numa linguagem de scripting.

Linguagens clássicas versus linguagens de scripting

As linguagens clássicas são geralmente **estaticamente tipificadas** para permitir detetar cedo os erros que podem ocorrer quando se utilizam elementos primitivos na construção de estruturas de dados complexas. Em constaste, as linguagens de scripting são sempre **tipificadas dinamicamente** pois um sistema de tipos estático seria uma complicação burocrática que só serviria para atrasar a escrita de scripts.

Outra diferença é o facto das linguagens de scripting serem geralmente **interpretadas** e não compiladas. Um dos objetivos disso é acelerar o desenvolvimento do código. Outro objetivo é facilitar a geração dinâmica de scripts que possam ser executados imediatamente pelo interpretador.

Nas linguagens de scripting dá-se pouca importância à questão eficiência. Em contrapartida dá-se a máxima importância à simplicidade, flexibilidade de utilização e a um grande poder expressivo que permita escrever scripts compactos. A questão da eficiência é pouco importante porque os scripts tendem a ser pequenos e porque a eficiência da linguagem é dominada pela eficiência das componentes, as quais são normalmente implementadas numa linguagens de programação clássica.

Algumas linguagem de scripting são também concebidas para serem usadas no interior duma aplicação de software, com o objetivo de fornecer ao utilizador um elevado grau de controlo do comportamento da aplicação, incluindo a adição de novas funcionalidades. Se é verdade que o utilizador não pode alterar o código de base da aplicação, ele pode escrever scripts para adaptar a aplicação às suas necessidades. Exemplos: Emacs Lisp é a linguagem de scripting do editor de texto emacs; JavaScript é linguagem de scripting mais usada nos browsers da WEB.

Como escolher entre linguagem clássicas e de scripting

Quando estão em causa aplicações que envolvem acima de tudo a coordenação de componentes já implementadas, podemos escolher programar essa aplicação numa linguagem clássica, e.g. Java, ou numa linguagem de scripting, e.g. JavaScript. Mas estudos mostram que se escolhermos uma linguagem de scripting, tanto o tempo de desenvolvimento como o tamanho da aplicação se reduzem num fator de 5 a 10, em média!

Quando estão em causa aplicações com algoritmos e estruturas de dados complexas, o melhor é usar uma linguagens de programação clássica. Usando uma linguagem de scripting, o script não ficaria mais pequeno, não haveria o benefício da tipificação estática para ajudar a apanhar antecipadamente erros subtis na utilização das estruturas de dados e, no final, o programa correria 10 vezes mais devagar.

Complementaridade dos dois tipos de linguagens

Se puderem ser usadas em conjunto, os dois tipos de linguagens permitem a criação de ambientes de desenvolvimento e execução de programas particularmente poderosos e flexíveis.

As linguagens de scripting sempre tiveram alguma popularidade, mas ultimamente a sua importância tem aumentado. A principal razão é a tendência atual para escrever aplicações baseadas em componentes já disponíveis. É o que se passa, por exemplo, quando está em causa o desenvolvimento de interfaces gráficas e de aplicações que correm sobre a WEB.

Exemplo: linguagem Bash

Bash (bourne-again shell) é uma linguagem de scripting muito usada no Linux na qual as "componentes" são as aplicações disponíveis e o sistema de ficheiros. Em Bash, um script implementa uma nova funcionalidade usando as aplicações do sistema. Uma das construções mais importantes do Bash é o **pipe**, que permite ligar o output duma aplicação ao input de outra aplicação. Em Bash também é possível testar o código-resultado duma aplicação e tomar decisões em conformidade (zero significa que a aplicação terminou sem erro; um valor diferente de zero representa um código de erro particular). Também é possível colocar o output duma aplicação numa variável e processar a seguir o conteúdo dessa variável.

Eis um exemplo de script em bash, retirado daqui. Este script lista na consola o nome de todos os ficheiros HTML que se encontram na diretoria corrente e, além disso, escreve a primeira linha de cada um desses ficheiros num ficheiro chamado FILE_HEADS.

```
#!/bin/bash
# This is a comment
echo "List of files:"
ls -lA

FILE_LIST="`ls *.html`"
echo FILE_LIST: ${FILE_LIST}

RESULT=""
for file in ${FILE_LIST}
do
    FIRST_LINE=`head -1 ${file}`
    RESULT=${RESULT}${FIRST_LINE}$'\n'
done

echo ${RESULT} | cat > FILE_HEADS

echo "${RESULT}"
echo "Script done."
```

No Windows, a linguagem de scripting chama-se PowerShell e foi introduzida em 2006 na versão Windows XP SP2. Antes do PowerShell usava-se a linguagem de scripting implementada pelo programa COMMAND.COM.

Discussão

As linguagens de scripting são geralmente concebidas para a execução de tarefas simples. Contudo, por vezes acabam por ser usadas em projetos grandes e de formas que ultrapassam as intenções originais. As características da linguagem que a tornam fácil de usar acabam por se revelar problemáticas nessas situações.

Quando comunidade de utilizadores é grande, então justifica-se fazer o seguinte:

- Criar ferramentas que ajudam a detetar mais erros antes do programa começar a correr. Exemplos: JSLint, Google Closure Tools.
- Começa a ser melhorada a eficiência da implementação usando técnicas avançadas: V8 JavaScript Engine.

Elementos da linguagem JavaScript

Regressemos ao estudo da linguagem JavaScrip, desta vez para apresentar os elementos de base mais importantes da linguagem.

Comparação do JavaScript com o Java

A primeira é uma linguagens de scripting. A segunda é uma linguagem clássica.

JavaScript	Java

Suporte para programação funcional	Desde o Java 8 suporta programação funcional limitada
Tipificação dinâmica	Tipificação estática
Baseada em protótipos	Baseada em classes
Herança usando o mecanismo dos protótipos	Herança através da hierarquia de classes
Adição dinâmica de novos membros a objetos	Não é possível a adição dinâmica de novos membros
Estilo livre onde a maioria das declarações são opcionais	Estilo rígido para ser possível detetar erros em tempo de compilação

---

Palavras reservadas do JavaScript

Esta é a lista das principais palavras reservadas usadas em JavaScript:

```
break     const      delete   for        import   new      this     void
case      continue   do       function   in       return   typeof   while
default   export     if       else       switch   var      with
```

Note que muitas das palavras anteriores são também palavras reservadas em Java.

As restantes palavras reservadas do Java, que não aparecem na lista anterior, também estão reservadas em JavaScript, apesar de não serem usadas de momento. Todos os interpretadores de JavaScript deveriam proibir a utilização destas palavras. Contudo alguns não o fazem.

Insersão automática dos pontos e vírgula (não usar)

Em JavaScript, o ponto e vírgula é usado como terminador da maioria das instruções e declarações.

```
a = 5 ; v = 6;
```

Mas se tivermos uma sucessão de instruções em linha diferentes, podemos omitir o ponto e vírgula porque o JavaScript insere implicitamente o terminador nesses casos. Exemplo:

```
a = 5
v = 6
```

A **insersão automática dos pontos e vírgula** foi feita para ajudar o programador, mas infelizmente as regras são um pouco complicadas. **O melhor será explicitar sempre os pontos e vírgula**, como estamos habituados em Java e C.

Um exemplo confuso tirado da expecificação do ECMAScript® 2017. No seguinte caso

```
a = b + c
(d + e).print()
```

não é inserido ponto e vírgula, resultando o equivalente a

```
a = b + c(d + e).print()
```

Variáveis

O JavaScript é uma linguagem **dinamicamente tipificada**. Uma consequência disso é o facto das **variáveis não terem tipos associados**. Os tipos ficam associados aos valores e não às variáveis. Exemplo:

```
var x = 34;     // x contém um inteiro
x = "Hello!"    // agora x contém uma string
```

Definição de variáveis

A palavra `var` permite definir variáveis. Factos diversos:

- Dentro duma função, `var` define uma **variável local**.
- O **argumento duma função**, também define implicitamente uma variável local (mas não se excreve a palavra var).
- Usada fora duma função, `var` define uma **variável global**.

- Usa-se **escopo estático** na resolução de nomes.
- Uma variável definida dentro dum bloco têm como âmbito toda a função envolvente, ou seja, **não há âmbito de bloco**. [Na versão mais recente do JavaScript foi introduzida a palavra `let` que permite definir variáveis com escopo de bloco. Para usar, basta trocar `var` por `let`.]

Exemplo com variáveis globais e variáveis locais:

```
var x = 5;
var z = 7;

function f(x) {
    console.log(x);
    console.log(y);   // Não inicializada ainda
    x = 1;
    var y = 2;
    console.log(x);
    console.log(y);
    console.log(z);
    return x;
}

f(6);
console.log(x);

// Output: 6  undefined  1  2  7  5
```

Exemplo com aninhamento de funções:

```
function f(x) {
    function g(y) {
        var x = 10;
        console.log(y);
        console.log(x);    // Imprime o x local
        return 0;
    }
    console.log(x);
    g(1);
    console.log(x);
}

f(6);

// Output: 6  1  10  6
```

Exemplo relativo ao facto de não existir âmbito de bloco em JavaScript:

```
function f() {
    var x = 1;
    {
       var x = 2;
    }
    console.log(x);
}

f();

// Output: 2
```

Variáveis indefinidas

Variáveis definidas num determinado âmbito, ficam com o valor `undefined` enquanto não forem inicializadas.

Exemplos sobre variáveis indefinidas:

```
console.log(zzz);   // Lança a exceção ReferenceError se zzz nao estiver definida
var zzz;
if( zzz === undefined ) {   // Testar se uma variável está indefinida.
    console.log("zzz is undefined");
}
if( !zzz ) {    // undefined comporta-se como false num contexto booleano
    console.log("zzz is undefined");
}
```

Atribuição a variáveis

A atribuição efetua-se usando o operador `=`. É possível efetuar uma atribuição a um nome ainda não definido. Nesse caso é automaticamente criada uma variável global inicializada. Para garantir que a variável fica local, é preciso declará-la usando `var`.

```
x = 5;
y = 7 + 5;
```

Constantes

A palavra `const` permite definir constantes. Mas, cuidado, que isto não faz parte do JavaScript padrão. Contudo está disponível no Rhino e Node.js, por exemplo.

```
const x = 5;
```

Tipos primitivos

Os tipos primitivos são os seguintes:

- **number** - Mistura reais e inteiros. O maior valor em várias implementações é 1.7976931348623157e+308. 0377 é um valor em octal e 0xFF é um valor em hexadecimal.
- **boolean** - Tem os valores `false` e `true`.
- **string** - Por exemplo `""`, `''`, `"Hello"`, `'Hello'`, `"inner 'string' "`, `'inner "string" '`.
- **null** - Este tipo só tem o valor `null` e serve para atribuir a uma variável para indicar que esta não tem valor.
- **undefined** - Este tipo só tem o valor `undefined`, que é valor das variáveis não inicializadas.

Note que em JavaScript, uma string não é um objeto. No entanto também há objetos de tipo String que simplesmente encapsulam valores primitivos de tipo string. Em Javascript as strings são imutáveis, tal como em Java.

Um literal de tipo string é uma porção de texto rodeada, ou por aspas ou plicas. Em código JavaScript, é comum usar plicas entre aspas e aspas entre plicas, como nestes exemplos: "He is called 'Neko-chan'"; 'He is called "Neko-chan"'.

O operador `typeof` pode ser usado para saber o tipo de qualquer valor.

São efetuadas **conversões automáticas de tipo**, entre os tipos primitivos. Exemplos:

```
"The answer is " + 42       // produz "The answer is 42"
42 + " is the answer"       // produz "42 is the answer"
"37" - 7                    // produz 30 ... !?
"37" + 7                    // produz "377" ... !?
true + 7                    // produz 8
```

Operadores

Eis a tabela de operadores do JavaScript ordenada por prioridade decrescente:

```
member                    .  []
call / create instance    ()  new
negation/increment        !  ~  -  +  ++  --  typeof  void  delete
multiply/divide           *  /  %
addition/subtraction      +  -
bitwise shift             <<  >>  >>>   (o último faz shift sem sinal)
relational                <  <=  >  >=  in  instanceof
equality                  ==  !=  ===  !==
bitwise-and               &
bitwise-xor               ^
bitwise-or                |
logical-and               &&
logical-or                ||
conditional               ?:
assignment                =  +=  -=  *=  /=  %=  <<=  >>=  >>>=  &=  ^=  |=
comma                     ,
```

Operadores de igualdade

```
==        Igualdade, produz true se os argumentos forem iguais (após possíveis conversões automáticas de tipo).
```

```
!=          Desigualdade, produz true se os argumentos forem diferentes.
===         Igualdade estrita, produz true se os argumentos forem iguais e do mesmo tipo.
!==         Desigualdade estrita, produz true se os argumentos forem diferentes ou se forem de tipos diferentes.
```

Expressões regulares

O JavaScript suporta expressões regulares semelhantes às da linguagem Perl.

A seguinte expressão regular representa dois "a"s seguidos de zero ou mais dígitos:

```
re = /aa\d*/
re = new RegExp("aa\\d*")          // Equivalente
```

A próxima expressão regular, mais abaixo na caixa, representa um "d" seguido de um ou mais "b"s seguido dum "d". As flags "i" e "g" indicam que o emparelhamento deve ignorar a caixa das letras e que deve ser global.

O método `test` determina se uma string emparelha com a expressão regular.

O método `exec` produz um array com o resultado do emparelhamento na posição 0 do array, mais os resultados dos emparelhamentos das sub-expressões entre parêntesis. Se a expressão regular tiver a flag "g" ligada, então sucessivas chamadas de `exec` produzem sucessivos resultados de emparelhamentos até ser retornado `null`; sem a flag "g" apenas o resultado do primeiro emparelhamento é retornado.

```
var re = /d(b+)(d)/ig;
re = new RegExp("d(b+)(d)", "ig");        // Equivalente
var b = re.test("cdbBdbsbz");             // resultado: true
var arr = re.exec("cdbBdbsdbdz");         // primeiro resultado: ["dbBd", "bB", "d"]
```

As expressões regulares suportam ainda os métodos `match`, `search`, `replace`, `split`.

Arrays

Em JavaScript os arrays podem ser inicializados, pelo menos de duas maneiras diferentes. Exemplo:

```
var colors = ["Red", "Green", "Blue"];
var colors = new Array("Red", "Green", "Blue");    // Equivalente
```

Tal como em Java os índices começam em zero e existe uma propriedade `length`.

```
var len = colors.length;                          // Vale 3
```

Eis um exemplo dum array de comprimento 6 com apenas 4 elementos. Dois elementos estão indefinidos.

```
var colors = ["Red", , , "Green", "Blue","Yellow"];
```

Ao contrário do Java, os arrays crescem automaticamente. Basta atribuir a uma posição inexistente para o array crescer.

```
var colors = [];                                  // Array vazio
colors[2] = "Blue";
var len = colors.length;                          // Vale 3
var t = typeof(colors[0]);                        // Vale undefined
```

Para fazer crescer um array na primeira posição livre, fazer assim:

```
colors[colors.length] = "Yellow";
colors.push("Yellow");                          // Equivalente
```

Para aceder e remover o último elemento dum array fazer:

```
var last = colors.pop();
```

É possível escrever diretamente na propriedade `length` dum array para fazer um array crescer, ou para truncar o array:

```
colors.length = 2;
```

Para percorrer os elementos dum array pode usar-se um `for` clássico, mas também se pode fazer assim:

```
var colors = ["Red", "Green", "Blue"];
colors.forEach(function(c) { console.log(c) });      // Iteração usando função anónima
```

ou assim:

```
var colors = ["Red", "Green", "Blue"];
for( c of colors ) console.log(c);      // Iteração usando for-of
```

Eis um array a duas dimensões, 2x3:

```
var table = [[0, 1, 2],
             [3, 4, 5]];
var r = table[0][2];                              // Vale 2
```

Outros métodos disponíveis para arrays: `join`, `reverse`, `shift`, `slice`, `splice`, `sort`, e muitos outros.

Estão disponíveis úteis funções de ordem superior que podem ser aplicadas a arrays. Exemplos:

```
> var array = [0, 1, 2, 3, 4, 5];
undefined
> array.map(function(x) { return x+1; });
[ 1, 2, 3, 4, 5, 6 ]
> array.map(x => x+1);
[ 1, 2, 3, 4, 5, 6 ]
> array.filter(x => x % 2 == 0);
[ 0, 2, 4 ]
> array.reduce((acc,v) => acc+v, 0);
15
> array.every(x => x >= 0);
true
> array.some(x => x >= 0);
true
> array.find(x => x > 0);
1
```

Funções

O JavaScript suporta o paradigma de programação funcional pois inclui funções anónimas, funções de ordem superior e funções que retornam outras funções.

```
function square(n) { return n * n }              // Função com nome.
var square = function(n) { return n * n };       // Equivalente. Função anónima clássica.
var square = n => n * n;                         // Equivalente. Função anónima simplificada (arrow function).
```

Eis um exemplo duma função de ordem superior, que depois é chamada usando uma função anónima como argumento:

```
function map(f, a) {
    var result = [];
    for( var i = 0 ; i < a.length ; i++ )
        result[i] = f(a[i]);
    return result;
}

var a = map(function(x) { return x * x }, [0, 1, 2, 3]);   // Vale [0, 1, 4, 9]
var a = map(x => x * x, [0, 1, 2, 3]);   // Vale [0, 1, 4, 9]
```

Nas chamadas das funções o número de argumentos não é validado: argumentos a mais na chamada são ignorados; argumentos a menos na chamada ficam indefinidos.

Dentro da cada função (exceto anónimas simplificadas) há um array predefinido chamado `arguments` que representa a sequência de argumentos realmente usados na chamada. Assim é fácil implementar funções com um número variável de argumentos, como no seguinte exemplo:

```
function allAll() {
    var result = 0;
    for( a of arguments )
        result += a;
    return result;
}
```

```
var res = allAll(1,2,3,4,5);                    // Vale 15
```

A passagem de argumentos de tipos primitivos é feita por valor. Os objetos são passados por referência.

Eis algumas funções predefinidas em JavaScript:

- **eval(string)** - Avalia uma string contendo código JavaScript.
- **isFinite(number)** - Testa se um número é finito.
- **isNaN(number)** - Teste se um número é a constante NaN.
- **parseInt(string, radix)** - Converte string em número inteiro.
- **parseFloat(string)** - Converte string em número real.
- **Number(obj)** - Converte um objeto num número.
- **String(obj)** - Converte um objeto numa string.

O argumento this

Todas as funções (exceto anónimas simplificadas) possuem automaticamente um primeiro argumento que não se declara e se chama **this**.

No caso de funções que não fazem parte de objetos, quando elas são chamadas, o argumento **this** fica associado a um objeto que depende da implementação. Por exemplo, no caso do Rhino e do Node.js, **this** fica associado a um objeto especial que guarda as ligações de todos os nomes globais. No seguinte exemplo, se a função `f` abaixo for chamada assim - `f()` - ela imprime-se a si própria de duas maneiras diferentes:

```
function f() {
    console.log(f);
    console.log(this.f);
}
```

No caso duma função `f` ser um método dum objeto `obj` e for chamada assim `obj.f()`, o argumento **this** fica associado ao próprio objeto (aliás, como também acontece em Java).

A operação primitiva call

Ao nível mais básico, em JavaScript todas as chamadas de função são feitas através da operação primitiva **call** que está disponível em todas as funções. Numa chamada de **call**, tem de se explicitar o primeiro argumento, **this**.

Todas as chamadas normais de funções são automaticamente traduzidas para invocações de **call**. Concretamente, a chamada duma função independente é assim traduzida, em Rhino:

```
f(1,2,3)     -> f.call(global(),1,2,3)
```

A chamada do método dum objeto é assim traduzida:

```
obj.f(1,2,3)     -> obj.f.call(obj,1,2,3)
```

Cuidado com a seguinte subtileza. No código abaixo, as duas chamadas não são equivalentes. Só a primeira chamada posiciona **this** igual a **obj**.

```
obj.f(1,2,3);       // uma chamada
var my_f = obj.f;   // Extrai a função f do objeto, e agora o objeto fica esquecido.
my_f(1,2,3);        // outra chamada, que não é equivalente (no valor de this dentro da função)
```

Nesta outra forma, as duas chamadas já são equivalentes:

```
obj.f(1,2,3);       // uma chamada
var my_f = obj => obj.f(1,2,3);
my_f(1,2,3);        // outra chamada, completamente equivalente
```

A operação primitiva apply

Existe ainda outra operação primitiva chamada **apply**, semelhante a **call**, com a única diferença que se passa para ela um array com todos os argumentos que aparecem a seguinte a **this**. Se estiver em causa um método, as seguintes três chamadas são equivalentes:

```
obj.f(1,2,3)   <->   obj.f.call(obj,1,2,3)   <->   obj.f.apply(obj,[1,2,3])
```

Tratamento de exceções

Semelhante ao Java. Exemplo:

```
try {
    undefinedFunction();
    alert('Afinal a função existe');
} catch(e) {
    alert('Erro: ' + e.message);
} finally {
    alert('Chega sempre aqui, independentemente do que aconteceu antes.');
}
```

Objetos

Em JavaScript para além dos tipos primitivos, temos os tipos objeto. Os arrays também são considerados objetos.

Como habitualmente, um **objeto** é um elemento de dados que possui identidade e que interage com outros objetos através da troca de mensagens.

Em JavaScript existem os seguintes objetos predefinidos, que oferecem funcionalidades úteis: Date, Array, Boolean, Function, Math, Number, RegExp e String .

Mas no ambiente de execução envolvente, estão geralmente disponíveis mais objetos predefinidos. Por exemplo, no ambiente dum browser, todos os tipos de objetos previstos no DOM estão disponíveis: Document, Window, Form, Link, etc.

#90