Linguagens de Programação
Roteiro de Laboratório 01 – JavaScript

Os roteiros de laboratório iniciais cobrem o uso de JavaScript tanto como uma linguagem de programação convencional quanto como uma linguagem de *script* que funciona dentro de um navegador.

Parte I
A Linguagem JavaScript

Nessa primeira parte vamos aprender o básico de JavaScript, começando por conceitos fundamentais e evoluindo até aspectos mais avançados. Vamos nos concentrar no núcleo da linguagem aqui, evitando detalhes sobre ambientes específicos.

O conteúdo desse roteiro foi adaptado do **The Modern JavaScript Tutorial**, disponível em `https://javascript.info/`.

1 Introdução

Vamos começar com alguma terminologia e história da linguagem.

O que é JavaScript

JavaScript foi inicialmente criada para prover mais funcionalidades em páginas *web*. Ela inicialmente chamava-se LiveScript, mas por questões comerciais, os seus criadores na Netscape decidiram mudar o nome para JavaScript, apesar da nova linguagem não ter nenhuma relação com Java além de uma sintaxe similar. JavaScript muitas vezes também é chamada pelo nome da sua especificação: ECMAScript.

Os programas da linguagem são chamados de *scripts*. Eles podem ser embutidos diretamente no código HTML das páginas *web* e são executados automaticamente quando a página é carregada.

Hoje em dia, JavaScript (JS) pode ser executada fora do navegador, em qualquer sistema que possua um **motor** (*engine*) JavaScript, também chamado de *JavaScript virtual machine*, que é essencialmente um interpretador da linguagem.

Cada navegador possui um motor diferente:

- **V8** – é o motor usado no Chrome e Opera.
- **SpiderMonkey** – é o motor usado no Firefox.
- **Nitro** – é o motor usado no Safari.

É importante lembrar os nomes acima porque cada motor possui algumas características específicas, então alguns detalhes da linguagem (principalmente I/O) não são padronizados. Assim, se você ler em algum lugar que a funcionalidade X é suportada por **V8**, então ela provavelmente vai funcionar no Chrome e Opera.

Para executar JS fora do navegador, você precisa instalar algum interpretador da linguagem. Isso pode ser feito facilmente através de um gerenciador de pacotes no Linux. (Se você não usa Linux, está na hora de você ponderar seriamente a sua escolha de SO... :P)

Eu vou usar o interpretador `js78`, que usa o motor **SpiderMonkey**. Caso você queria usar o motor **V8**, instale o interpretador `nodejs`. Embora ambos sejam compatíveis com todo o núcleo da linguagem, alguns comandos de I/O diferem entre os interpretadores. Por exemplo, o `js78` provê um comando `print` que não é reconhecido no `nodejs`.

```
$ js78
js> print("Oi");
Oi
$ node
Welcome to Node.js v15.4.0.
> print("Oi");
Uncaught ReferenceError: print is not defined
> console.log("Oi");
Oi
```

Eu vou tentar manter os comandos de todos os exemplos o mais uniformes possíveis, para que eles rodem tanto com o **SpiderMonkey** ou o **V8**. Casos específicos serão destacados no texto. De qualquer forma, caso você use o `nodejs`, as mensagens do seu interpretador podem ficar diferentes, já que eu vou usar o `js78`. (Note que o número 78 indica a versão do interpretador. Diferentes distribuições Linux pode ter versões diferentes desse interpretador.)

Atividade

Instale o interpretador JS que você escolheu na sua máquina.

Linguagens "sobre" JavaScript

Recentemente surgiram novas linguagens que são convertidas (*transpiled*) para JS antes de serem executadas. Ferramentas fazem essa conversão de forma rápida e transparente, permitindo o desenvolvimento nessas novas linguagens e fazendo a tradução de forma automática, efetivamente transformando o interpretador JS na arquitetura alvo da nova linguagem.

Exemplos de linguagens:

- **CoffeeScript** – é uma simplificação de JS que introduz uma sintaxe mais curta.
- **TypeScript** – introduz um sistema estático de tipos em JS para facilitar o desenvolvimento de sistemas mais complexos. Desenvolvida pela Microsoft. (Será vista em um laboratório específico mais para frente.)
- **Flow** – também adiciona tipos mas de outra forma. Desenvolvida pelo Facebook.
- **Dart** – é uma linguagem independente, com o seu próprio motor que roda fora do navegador, mas que pode ser traduzida para JS. Desenvolvida pelo Google.

2 Fundamentos de JavaScript

Vamos ver agora os fundamentos básicos de JS.

Atividade

Procure reproduzir na sua máquina todos os exemplos apresentados nesta seção.

2.1 *Hello, world!*

Como não poderia deixar de ser, vamos começar pelo *Hello, world!*. Para rodar o programa de forma interativa, basta abrir o terminal, chamar o interpretador e executar o programa, como abaixo.

```
$ js78
js> console.log("Hello, world!");
Hello, world!
```

Os *scripts* em JavaScript possuem a extensão `.js`. Podemos então salvar o nosso programa abaixo no arquivo `hello.js`.

```
console.log("Hello, world!");
```

E a seguir executar o *script* diretamente no terminal.

```
$ js78 hello.js
Hello, world!
```

2.2 Estrutura do código

Vamos agora ver os componentes de um código JS.

Comandos (*Statements*)

Statements são construtos sintáticos e comandos que realizam ações. Blocos de comandos podem ser separados por ponto e vírgula:

```
console.log("Hello");
console.log("World!");
```

Ponto e vírgula

O ponto e vírgula pode ser omitido **quase sempre** que houver uma quebra de linha. Assim, esse programa também funciona.

```
console.log("Hello")
console.log("World!")
```

A linguagem interpreta a quebra de linha como um ponto e vírgula "implícito". Isso é chamado de **inserção automática de ponto e vírgula**.

Note que foi dito acima que o ponto e vírgula pode ser omitido **quase sempre**, ou seja, há locais em que ele ainda é necessário. Por exemplo, o código abaixo possui um erro porque o interpretador não insere um ponto e vírgula implícito antes de colchetes `[...]`.

```
console.log("There will be an error")
[1, 2].forEach(console.log)
```

```
There will be an error
  TypeError: can't access property 2, console.log(...) is undefined
```

Por conta de erros como o acima, considera-se uma boa prática de programação em JS **sempre** se usar `;` em todos os comandos, mesmo quando eles sejam opcionais.

Comentários

Comentários em JS são como em C++ e Java.

```
/* An example with two messages.
   This is a multiline comment.
*/
console.log("Hello");
console.log("World!"); // Single line comment.
```

Comentários multilinha não podem ser aninhados. O código abaixo não roda.

```
/*
  /* nested comment ?!? */
*/
console.log("World!");
```

```
test.js:3:0 SyntaxError: expected expression, got '*':
```

2.3 JS moderno, diretriz *"use strict"*

Até 2009, novas funcionalidades foram adicionadas à linguagem mantendo-se a compatibilidade retroativa. No entanto, com a publicação do padrão ECMAScript 5 (ES5) novas características foram adicionadas e algumas já existentes foram modificadas, quebrando a compatibilidade. Para manter a base de código antiga funcionando, a maioria das modificações ficam desligadas por padrão. Você precisa ligá-las explicitamente com a diretiva de compilação `use strict`.

A diretiva é simplesmente uma *string* que deve aparecer no início de um *script*, podendo ser precedida somente por comentários.

```
"use strict";

// This code works the modern way.
...
```

Dado que essa diretiva já foi introduzida há bastante tempo, atualmente uma boa parte da base de código JS existente já está no modo "moderno". Assim, vamos assumir daqui para frente que a diretiva está ligada em todos os exemplos, destacando os casos especiais em que ela ainda faz alguma diferença.

2.4 Variáveis

Variáveis em JS podem ser criadas com as palavras reservadas `let`, `var` e `const`:

- `let` – é a forma **atual** recomendada para declaração de variáveis.
- `var` – é a forma **antiga** para declaração, não devendo ser mais utilizada. Existem diferenças sutis entre `let` e `var`, elas serão explicadas adiante.
- `const` – como `let` mas o valor da "variável" não pode ser modificado.

Usando `let`

Variáveis são declaradas com o comando `let`. É possível declarar mais de uma por linha, além de realizar a sua inicialização ao mesmo tempo que ela é declarada.

```
let message;
let userA = "John";
let ageA = 25;
let userB = "Mary", ageB = 28;
message = "Hello";
```

Nomes de variáveis

Há duas limitações para nomes de variáveis em JavaScript:

1. O nome deve conter somente letras, dígitos ou os símbolos `$` e `_`.
2. O primeiro caractere não pode ser um dígito.

Quando um nome é formado por várias palavras, costuma-se usar `camelCase`, como em Java. (Note que *camel case* é algo maligno que deve sempre ser evitado quando possível. Veja, por exemplo, o artigo **Against Camel Case** em `https://www.nytimes.com/2009/11/29/magazine/29FOB-onlanguage-t.html`.)

Uma curiosidade da linguagem é que os símbolos `$` e `_` não possuem nenhum significado especial, podendo ser usados diretamente como nomes de variáveis:

```
let $ = 1; // Declared a variable with name "$".
let _ = 2; // And now a variable with name "_".
console.log($ + _); // 3
```

Obviamente, somente pessoas insanas criam variáveis com esses nomes... :P

Palavras reservadas

A lista das palavras reservadas de JavaScript pode ser vista aqui: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#keywords`

Note que alguns nomes só são considerados reservados no modo estrito! Por exemplo, o código abaixo dá um erro, como era de se esperar.

```
"use strict";
let = 42;
console.log(let);
```

```
test.js:2:0 SyntaxError: let is a reserved identifier:
test.js:2:0 let = 42;
test.js:2:0 ^
```

Por outro lado, se sairmos do modo estrito, o código abaixo roda! O_o

```
let = 42;
console.log(let);
```

```
$ js78 test.js
42
```

Esse último exemplo também ilustra uma característica adicional do JS antigo. Sem se usar o modo estrito, é possível se criar uma variável simplesmente fazendo-se uma atribuição, como em Python. Isto não é considerado uma boa prática de programação em JS moderno.

```
"use strict";
num = 42; // ReferenceError: assignment to undeclared variable num
```

Constantes

Para declarar constantes basta usar `const`.

```
const century = "01/01/2001";
century = "99/99/9999"; // TypeError: invalid assignment to const
```

Convencionou-se usar identificadores em maiúsculas para valores *hard-coded*, isto é, conhecidos em tempo de compilação, e usar a nomenclatura padrão de nomes para constantes cujos valores são obtidos em tempo de execução.

O velho `var`

Como já dito, o comando `var` não é mais utilizado em *scripts* modernos, mas foi mantido na linguagem para compatibilidade com código antigo. Sendo assim, ainda é importante saber o seu funcionamento. Embora `var` possa ser usado para declarar variáveis, ele tem certas peculiaridades que vamos discutir agora.

Variáveis declaradas com `var` não possuem um escopo de bloco, com a sua visibilidade sendo içada (*hoisted*) para o escopo da função atual, ou para o escopo global, se a variável for declarada fora de uma função. Assim, um código como abaixo funciona.

```
if (true) {
  var test = 42;
}
console.log(test); // 42: the variable lives after if block.
```

Por outro lado, `let` considera o escopo de bloco corretamente.

```
if (true) {
  let test = 42;
}
console.log(test); // ReferenceError: test is not defined
```

Por conta do içamento, uma variável pode ser utilizada antes da sua declaração! O código abaixo roda normalmente. :/

```
function sayHi() {
  msg = "Hi";
  console.log(msg);
  var msg;
}
sayHi();
```

Por fim, convém destacar que `var` também aceita a redeclaração de variáveis.

```
var user = "John";
var user = "Mary"; // Doesn't trigger an error...
console.log(user); // Mary
```

Ao passo que isso é um erro com `let`:

```
let user = "John";
let user = "Mary"; // SyntaxError: redeclaration of let user
```

Por todas estas peculiaridades, fica fácil perceber porque o uso de `var` não é mais recomendado.

2.5 Tipos de dados

Há 8 tipos de dados básicos em JavaScript.

- `number` – para qualquer tipo de número: inteiro ou ponto flutuante. Inteiros ficam limitados no intervalo $\pm(2^{53}-1)$.
- `bigint` – para números inteiros de tamanho arbitrário.

- `string` – para cadeias de zero ou mais caracteres. Não existe um tipo específico para um único caractere.
- `boolean` – para os valores `true` e `false`.
- `null` – tipo unitário para indicar valores desconhecidos.
- `undefined` – tipo unitário para indicar valores indefinidos.
- `object` – para representar as estruturas de dados da linguagem. Este é o único tipo de dados não-primitivo da linguagem.
- `symbol` – para identificadores únicos em um programa.

O operador `typeof` serve para se ver qual é o tipo atualmente armazenado em uma variável.

- Duas formas: `typeof x` ou `typeof(x)`.
- Retorna uma *string* com o nome do tipo.
- Para `null` retorna a *string* `object`. Isto é um erro na linguagem, esse tipo não representa um objeto. (Esta é só mais um exemplo das idiossincrasias de JS... :/)

Vamos agora falar mais um pouco dos tipos primitivos de JS, deixando a discussão sobre objetos mais para adiante.

JavaScript é uma linguagem com **tipagem dinâmica**. Assim, o tipo de uma variável pode mudar ao longo da execução do programa, não sendo amarrado estaticamente. O tipo atual de uma variável corresponde ao tipo do dado armazenado nela no momento, da mesma forma que em Python.

```
let msg = "hello";
console.log(typeof msg);  // string
msg = 42;
console.log(typeof msg);  // number
```

Por atrapalhar consideravelmente a legibilidade de um programa, esse tipo de reúso de variáveis não é recomendado.

Number

O tipo `number` é usado para representar tanto números inteiros quanto de ponto flutuante, com as operações aritméticas usuais suportadas. Além dos números, há três valores especiais que também pertencem ao tipo: `Infinity`, `-Infinity` e `NaN` (*Not a Number*), que é o resultado de uma operação matemática indefinida ou incorreta.

```
console.log(typeof 42);    // number
console.log(typeof 4.2);   // number
console.log( 1 / 0 );      // Infinity
console.log( -1 / 0 );     // -Infinity
console.log( "abc" / 2 ); // NaN
```

Operações matemáticas são "seguras" em JS: um *script* nunca vai falhar por conta de uma divisão por zero ou outra operação inválida. No pior dos casos, obtém-se um `NaN` como resultado.

Como os valores especiais são definidos como pertencendo ao tipo `number`, podemos ver casos engraçados como abaixo, aonde um *Not a Number* é do tipo `number`... :P

```
console.log(typeof NaN); // number
```

BigInt

Em JS, o tipo `number` só consegue armazenar valores inteiros no intervalo $\pm(2^{53}-1)$. Caso sejam necessários valores maiores, deve se usar o tipo `bigint`, capaz de representar inteiros de tamanho arbitrário. Um valor `bigint` é criado acrescentando um `n` no final do número.

```
const x = 123456789n;
console.log(typeof x); // bigint
```

String

Uma *string* em JavaScript é uma sequência de caracteres cercada por aspas. Em JS, podemos usar três tipos de aspas.

1. Aspas duplas: `"`
2. Aspas simples: `'`
3. *Backticks*: `` ` ``

Aspas simples e duplas são usadas para se criar *strings* "simples". Não há nenhuma diferença prática entre elas em JS. Por outro lado, *backticks* permitem a inclusão de expressões dentro de uma *string*, usando-se o comando `${...}`. Seguem alguns exemplos.

```
let str = "Hello";
let msg = 'Single quotes are OK too.';
// Embed a variable.
console.log(`Can embed another ${str}`); // Can embed another Hello
// Embed an expression.
console.log(`The result is ${4 + 2}`); // The result is 6
// No embedding!
console.log('The result is ${4 + 2}'); // The result is ${4 + 2}
```

Valores `null` e `undefined`

Os valores especiais `null` e `undefined` pertencem a dois tipos unitários distintos. Uma variável que foi declarada mas ainda não teve um valor atribuído começa contendo `undefined`. Por outro lado, o valor `null` é usado para indicar que algo é vazio ou tem um valor desconhecido. (Ao contrário de outras linguagens, `null` não deve ser interpretado como um ponteiro ou referência nula, já que JS não possui esses conceitos.)

```
let x;
console.log(x); // "undefined"
x = null;
console.log(x); // "null"
```

Object e Symbol

Todos os tipos de dados vistos até agora são considerados tipos primitivos em JS. O único tipo não-primitivo da linguagem é `object`, que representa qualquer tipo estruturado como vetores e dicionários, bem como instâncias de "classes". Esses conceitos serão discutidos mais para frente.

O tipo `symbol` é usado para se criar identificadores únicos para objetos. Esse tipo também será discutido adiante, quando tratarmos de objetos.

2.6 Interação (I/O básico)

Vamos falar agora dos mecanismos básicos de interação com o usuário. Infelizmente, não há uma padronização em JS de como se fazer I/O no terminal. Como a linguagem foi criada para rodar dentro de navegadores, cada companhia desenvolveu os seus comandos específicos. Essa falta de padronização se estendeu aos interpretadores *stand-alone* também, com comandos diferentes em `js78` e `nodejs`.

Escrevendo em `stdout`

Como já visto anteriormente, para se escrever em `stdout`, basta usar `console.log`.

```
console.log("Hello, world!");
```

Usuários do interpretador `js78` também podem usar o comando `print`.

```
print("Hello, world!"); // Don't use this with Node.js!
```

Lendo de `stdin`

Infelizmente não há uma forma padronizada de leitura do `stdin`. O interpretador `js78` possui o comando `readline`, que é bem simples de usar. O retorno de `readline` é sempre uma *string*. Daqui a pouco vamos ver como realizar conversões de *strings* para números, por exemplo.

```
console.log("What is your name?");
let name = readline();
console.log(`Hello, ${name}!`);
```

```
$ js78 test.js
What is your name?
John
Hello, John!
```

Por outro lado, se tentarmos usar `readline` com o interpretador `nodejs`, obtemos o erro abaixo.

```
$ node test.js
What is your name?
let name = readline();
           ^
ReferenceError: readline is not defined
```

Nesse caso, uma das formas mais simples de se ler de `stdin` no `nodejs` é usar o módulo `prompt-sync`, mas primeiro você deve realizar algumas configurações.

1. Certifique-se de ter instalado o `npm`, que é o gerenciador de módulos do `nodejs`. Utilize o gerenciador de pacotes da sua distribuição Linux para tal.
2. Depois, execute `npm install prompt-sync` no terminal.

Agora, todo *script* que for receber alguma entrada deve importar o módulo `prompt-sync`. Isso é feito em `nodejs` com o comando `require`.

```
const prompt = require('prompt-sync')();
let name = prompt("What is your name? ");
console.log(`Hello, ${name}!`);
```

```
$ node test.js
What is your name? John
Hello, John!
```

Daqui em diante, lembre-se dessa diferença caso esteja usando o `nodejs`. Todos os exemplos no texto vão usar o comando `readline` do `js78`.

2.7 Conversão de tipos

A leitura de `stdin` sempre retorna uma *string*, portanto há casos em que é necessário realizar uma conversão de tipos. Muitos operadores de JS realizam conversão implícita quando necessário, isso será visto nas próximas seções. Vamos ver agora como fazer conversões explícitas.

Os tipos primitivos `string`, `number` e `boolean` podem ser criados pelas funções `String()`, `Number()` e `Boolean()`, respectivamente.

Conversões para *strings* são triviais, como ilustrado abaixo.

```
js> String(true)
"true"
```

Conversões para Booleanos também são simples. Valores que são "intuitivamente vazios", como `0`, `null`, `undefined`, `NaN` e *strings* vazias, viram `false`. Os demais valores viram `true`.

```
console.log( Boolean(0) )    // false
console.log( Boolean("") )   // false : empty string
console.log( Boolean(" ") ) // true  : non-empty string
console.log( Boolean("0") ) // true  : non-empty string
```

A conversão para números segue as regras indicadas abaixo.

Valor	Conversão
`undefined`	`NaN`
`null`	`0`
`true / false`	`1 / 0`
`string`	Tenta interpretar o número contido na *string* ignorando espaços em branco. A *string* vazia vira `0`. Um erro produz `NaN`.

```
js> Number("   42   ");
42
js> Number("42z");
NaN
```

2.8 Operadores básicos

Vamos falar agora dos operadores básicos de JavaScript, começando pelos aritméticos.

As seguintes operações matemáticas são suportadas em JS:

- Adição `+`,
- Subtração `-`,
- Multiplicação `*`,
- Divisão `/`,
- Resto `%`, e
- Exponenciação `**`.

O operador de exponenciação é associativo à direita, enquanto todos os demais são associativos à esquerda. A precedência dos operadores acima e dos demais operadores de JS está indicada na tabela deste *link*: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence#table`. Obviamente, é recomendável o uso de parênteses para melhorar a legibilidade de expressões complexas.

Os operadores `+` e `-` possuem versões unárias, com `-` unário sendo usado da forma usual para definição de números negativos.

```
let x = -1;
```

Já o `+` unário realiza conversões para o tipo `number`, da mesma forma que a função `Number()`.

```
js> +true
1
```

Concatenação de *strings*

O operador `+` binário é sobrecarregado em JS, podendo indicar também a concatenação de duas *strings*. Se um dos operandos for uma *string* e o outro não, ocorre uma conversão implícita.

```
js> 4 + '2';
"42"
js> 2 + 2 + '2';
"42"
```

Somente o operador `+` é sobrecarregado; os demais operadores aritméticos fazem uma conversão implícita dos operandos para `number` antes de realizar a operação.

```
js> '6' / '3';
2
```

Atribuições

Atribuições em JavaScript também são operadores, retornando o resultado da expresssão do lado direito, como em C. Isso permite o encadeamento de atribuições.

```
let a, b, c;
a = b = c = 42;
```

Atribuições compostas, como `+=` e `*=`, por exemplo, também podem ser utilizadas. O seu funcionamento é igual a C e Python. O mesmo vale para os operadores unários de incremento `++` e decremento `--`, que podem ser pré- e pós-fixados, como em C, C++ e Java.

```
let x = 4;
x += 2;
console.log(x++); // 6
console.log(x);   // 7
```

Comparações

JavaScript possui as seguintes operações de comparação, que sempre produzem um `boolean` como resultado.

- Igualdade `==`,
- Desigualdade `!=`,
- Inequalidades `>`, `<`, `>=`, `<=`,
- Igualdade estrita `===`, e
- Desigualdade estrita `!==`.

Excetuando-se `===` e `!==`, os demais operadores certamente já foram vistos em outras linguagens. Alguns comentários:

- Comparação entre *strings* é feita por ordem lexicográfica (alfabética).

- Quando os operandos possuem tipos distintos, eles são convertidos para números.

```
js> '4' > 2
true
```

- A conversão para números também ocorre com o operador de igualdade `==`, o que leva a vários casos "estranhos".

```
console.log( '01' == 1 ); // true, string '01' becomes number 1
console.log( true == 1 ); // true, value true becomes number 1
console.log( false == 0 ); // true, value false becomes number 0
console.log( false == '' ); // true, both values become number 0
console.log( 0 == "0" ); // true
console.log( Boolean(0) == Boolean("0") ); // false :/
```

Para se evitar a conversão de tipos implícita de `==`, foi criado o operador de igualdade estrita `===`, que verifica a igualdade dos operandos sem realizar nenhuma conversão de tipos. Em outras palavras, se `a` e `b` possuem tipos distintos, então `a === b` retorna `false` imediatamente, sem tentar converter os argumentos. O operador de desigualdade estrita `!==` funciona de forma análoga.

As comparações em JS ficam ainda piores quando envolvem os tipos `null` e `undefined`.

```
console.log( null == undefined ); // true
console.log( null === undefined ); // false
console.log( null > 0 ); // false
console.log( null == 0 ); // false
console.log( null >= 0 ); // true, O_o
```

Para manter a sanidade, é indicado seguir as seguintes recomendações.

- Nunca use `==`, use sempre `===`.
- Não use as inequalidades `> < >= <=` com variáveis que podem ser `null` ou `undefined`. Se for o caso, teste contra esses valores primeiro antes de fazer as demais comparações.

Atividade

Veja as referências adicionais abaixo caso queira se aprofundar mais no comportamento bizarro de alguns operadores de JS.

1. **When is it OK to use == in JavaScript?** – `https://2ality.com/2011/12/strict-equality-exemptions.html`. Resposta: Nunca!
2. **Wat** – `https://www.destroyallsoftware.com/talks/wat`. Vídeo engraçado sobre as loucuras de tipos em linguagens com tipagem dinâmica. Recomendado!
3. **JavaScript quirks in one image from the Internet** – `https://dev.to/mkrl/javascript-quirks-in-one-image-from-the-internet-52m7`. Página que explica alguns dos problemas ilustrados no vídeo anterior.

Operadores lógicos

Há três operadores lógicos em JavaScript: `||` (OR), `&&` (AND), e `!` (NOT). Os operadores `||` e `&&` são associativos à esquerda, e funcionam com **curto-circuito**, como em C e Java. A ordem decrescente de precedência é `!`, `&&` e `||`, mas, como sempre, o uso de parênteses em expressões complexas é recomendado para facilitar a leitura.

Como os demais operadores já vistos nesta seção, os operadores lógicos também podem realizar várias conversões implícitas de tipos. Por exemplo, algumas pessoas usam `!!` como um substituto para `Boolean()`.

```
js> Boolean("42")
true
js> !"42"
false
js> !!"42"
true
```

Já outras usam `||` encadeados para encontrar o primeiro valor "verdadeiro".

```
let firstName = "";
let lastName = "";
console.log( firstName || lastName || "Anonymous" ); // Anonymous
```

Novamente, não é recomendável fazer esse tipo de coisa no seu código. Parece óbvio (e é!), mas use os operadores lógicos somente para operações lógicas! Procure evitar a tentação de usar as soluções "espertas" da linguagem e adote um estilo de código que sempre deixa explícita a intenção do programador. Um trecho com `||` como acima pode ser reescrito usando comandos de `if/else`, por exemplo. Isto melhora a legibilidade e facilita muito as futuras manutenções.

2.9 Comandos de seleção

Vamos falar agora dos comandos de seleção de JavaScript, também chamados de comandos de *conditional branching*.

Comando `if`

O principal comando de seleção da linguagem é o `if/else`, que funciona exatamente como em C.

```
console.log("In which year was the ECMAScript-2015 published?");
let answer = readline();
let year = Number(answer);
if (year === 2015) {
  console.log("Well, duh! :P");
} else {
  console.log("Huh?!?");
}
```

É considerada uma boa prática de programação sempre se usar chaves `{ ... }` para delimitar os blocos de `if/else`, mesmo quando elas forem opcionais. Isso evita futuros *bugs* sutis que poderiam surgir caso mais comandos fossem adicionados a um bloco não delimitado de um `if` ou `else`.

Comando `?`

O operador ternário `?` de C e Java também existe em JavaScript, e possui a mesma semântica. Ele é utilizado como uma expressão condicional para retornar um valor. A sintaxe é:

```
let result = condition ? valueIfTrue : valueIfFalse;
```

Alguns *scripts* usam `?` como um substituto do `if` para controle de fluxo do programa.

```
console.log("Which company created JavaScript?");
let company = readline();
(company === "Netscape") ?
    console.log("Right!") : console.log("Wrong.");
```

Novamente, esse tipo de construção não é recomendado. Embora o comando com `?` seja mais curto que um `if` equivalente, ele também é menos legível. Os nossos olhos percorrem o código de forma vertical. Blocos que se espalham por múltiplas linhas são muito mais fáceis de ler do que uma única longa linha horizontal. Um código como abaixo é sempre preferível.

```
console.log("Which company created JavaScript?");
let company = readline();
if (company === "Netscape") {
  console.log("Right!")
} else {
  console.log("Wrong.");
}
```

Mesmo correndo o risco de soar excessivamente repetitivo, eu faço aqui a mesma recomendação. Use cada construto da linguagem para a sua função original: `?` para atribuição de valores e `if` para controle de fluxo.

Comando `switch`

Uma longa sequência de comandos `if/else` pode ser substituída por um comando `switch`. A sua sintaxe é idêntica a C e Java, e a semântica é muito parecida, inclusive com o *fall-through* dos casos sem `break`.

A diferença maior na semântica desse comando em JavaScript é que qualquer expressão pode ser usada como argumento tanto do `switch` quanto do `case`. O valor da expressão do `switch` é comparado por igualdade estrita (`===`) com o valor de cada `case`, até algum deles bater ou se chegar no caso `default`, se houver. Isso contrasta com as versões do comando em C e Java, que são mais restritivas, proibindo valores de ponto flutuante, por exemplo.

O código abaixo ilustra o uso do `switch`.

```
console.log("Enter a digit?");
let digit = readline();
switch (digit) {
    case '0':
    case '1':
        console.log("One or zero");
        break;
    case '2':
        console.log("Two");
        break;
    case 3:
        console.log("Will never run...");
        break;
    default:
        console.log("Four or more, or garbage...");
}
```

Note que o valor de retorno de `readline` é uma *string*, portanto os argumentos dos `case` também devem ser, para que a comparação por igualdade estrita passe. Por esse motivo, o `case 3:` nunca será executado, já que o tipo do argumento é `number`. O primeiro caso ilustra a semântica de *fall-through*.

Atividade

Este roteiro já mencionou várias vezes a necessidade de sempre se escrever código que siga adequadamente algum padrão de qualidade. Você chegou no momento da sua formação em que esse ponto fundamental para a sua vida profissional precisa ser aprendido e praticado.

Uma "filosofia" de programação bastante recomendada atualmente é o Clean Code, proposta por Robert C. Martin no livro de mesmo nome. Veja as breves leituras indicadas abaixo se quiser saber um pouco mais sobre os pontos principais de Clean Code.

- **Clean Code: 5 Essential Takeaways** – `https://medium.com/better-programming/clean-code-5-essential-takeaways-2a0b17ccd05c`.
- **Summary of Clean Code** – `https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29`.

"Clean code is not written by following a set of rules. You don't become a software craftsman by learning a list of heuristics. Professionalism and craftsmanship come from values that drive disciplines." – Robert C. Martin

2.10 Operador `??`

A versão ES11 (ECMAScript 2020) do padrão da linguagem introduziu recentemente o operador de coalescência nula (*nullish coalescing operator*), escrito como dois pontos de interrogação `??`.

O resultado de `a ?? b` é:

- `a`, se `a` não for `null` ou `undefined`; e
- `b`, caso contrário.

Esse novo operador é meramente uma conveniência sintática, não introduzindo nenhuma funcionalidade nova à linguagem. Por exemplo, podemos reescrever `result = a ?? b` usando outros operadores já vistos anteriormente.

```
result = (a !== null && a !== undefined) ? a : b;
```

Um caso de uso comum para `??` é selecionar o primeiro valor de uma sequência que não é `null/undefined`. Foi visto em um exemplo anterior o uso (não recomendado!) de `||` para se selecionar o primeiro valor "verdadeiro". A forma recomendada atualmente é se usar `??`.

```
let firstName = null;
let lastName = null;
console.log( firstName ?? lastName ?? "Anonymous" ); // Anonymous
```

2.11 Comandos de iteração

JavaScript possui três comandos para criação de *loops*: `while`, `do-while` e `for`. Todos possuem a mesma sintaxe e semântica de C e Java, inclusive no que diz respeito às interrupções `break` e `continue`. Além disso, JS suporta estas interrupções com *loops* rotulados, como em Java. Veja o exemplo a seguir.

```
outer:
for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    console.log(`Value at coords (${i},${j})?`);
    let input = readline();
    // If an empty string, then break out of both loops.
    if (input === '') {
        break outer;
    }
    // Do something with the value...
  }
}
console.log('Done!');
```

2.12 Funções

Uma declaração de função em JavaScript tem o seguinte formato.

```
function name(parameters, delimited, by, comma) {
  /* code */
}
```

Como JS possui tipagem dinâmica, não é preciso especificar o tipo do retorno, e nem dos parâmetros. As declarações de funções podem ser aninhadas e isso é uma prática comum na linguagem.

Uma função cria um novo escopo local de nomes, assim variáveis criadas com `let` possuem escopo local à função. Variáveis declaradas em escopos mais externos ficam visíveis dentro de uma função se não forem sombreadas por um mesmo nome mais interno.

Eis um exemplo simples de criação e uso de uma função.

```
function sayHelloTo(name) {
    console.log(`Hello, ${name}!`);
}

sayHelloTo("Mary"); // Hello, Mary!
sayHelloTo();       // Hello, undefined!
sayHelloTo("Mary", "John"); // Hello, Mary!
```

O código acima indica que não há nenhuma verificação de aridade na hora de uma chamada de função. Se um parâmetro não é fornecido, o seu valor se torna `undefined`. Se parâmetros em excesso forem passados, eles são descartados. Além disso, como não há uma especificação estática dos tipos dos parâmetros, não é feita nenhuma verificação de tipos nos argumentos.

É possível se definir valores padrão para os parâmetros ausentes, de forma similar a Ada e Python.

```
function sayHelloTo(name = "stranger") {
    console.log(`Hello, ${name}!`);
}

sayHelloTo(); // Hello, stranger!
```

A passagem de parâmetros em JavaScript é por valor para os tipos primitivos e por referência para objetos, como em Java.

Funções sempre retornam um valor em JS. Uma função sem comandos de retorno ou com um `return` vazio (sem expressão) produz o valor `undefined`.

Expressões de função (*function expressions*)

Em JavaScript, funções são elementos de primeira classe, o que quer dizer que elas podem ser:

- armazenadas em uma variável;
- passadas como um argumento para outra função; e
- retornadas como qualquer outro valor.

A sintaxe de funções vista até agora é dita **declaração de função** (*function declaration*), mas há uma outra forma, chamada **expressão de função** (*function expression*), como abaixo.

```
let sayHello = function() {
  console.log("Hello!");
};
```

Neste caso, a função é criada e atribuída à variável como qualquer outro valor. Para se chamar a função, basta usar o nome da variável com a sintaxe de sempre.

```
sayHello(); // Usual invocation syntax.
```

É possível até imprimir o código da função.

```
console.log(sayHello); // Shows the function code.
```

Uma variável que contém uma função possui o tipo `function`.

```
console.log(typeof sayHello); // function
```

Funções de *callback*

Como funções são elementos de primeira classe, não é necessária nenhuma sintaxe especial para passá-las como argumentos. Isso simplifica o uso de funções de *callback*.

```
function ask(question, yes, no) {
  console.log(question + " [Y/N]");
  let answer = readline();
  if (answer === 'Y') {
    yes();
  } else {
    no();
  }
}

ask(
  "Do you agree?",
  function() { console.log("You agreed."); },
  function() { console.log("You disagreed."); },
);
```

Esse tipo de construção é muito útil na prática. A maior diferença do exemplo acima para o mundo real é que as funções envolvidas normalmente são mais complexas. A ideia do exemplo é mostrar que funções podem ser passadas como argumentos e depois "chamadas de volta" (*called back*) se necessário. No código acima, a primeira expressão de função passada como segundo argumento de `ask` se torna o *callback* do parâmetro formal `yes`. De forma análoga, a segunda função anônima vira o *callback* de `no`.

Arrow functions

Há uma forma ainda mais simples e concisa de se declarar funções além de expressões de funções. Essa forma é chamada de "função flecha" (*arrow function*), por que a sua sintaxe é:

```
let func = (argA, argB, ..., argN) => expression;
```

Este comando cria a função `func` que aceita `N` argumentos, e a seguir avalia a expressão da direita, retornando o resultado. Em outras palavras, é uma versão mais simples para:

```
let func = function(argA, argB, ..., argN) {
  return expression;
};
```

Eis um exemplo concreto:

```
let sum = (a, b) => a + b;
console.log( sum(4, 2) ); // 6
```

Se a função possui um único parâmetro, os parênteses podem ser omitidos.

```
let double = n => n * 2;
console.log( double(5) ); // 10
```

Se a função não possui parâmetros, os parênteses ficam vazios, mas devem estar presentes.

```
let sayHello = () => console.log("Hello!");
sayHello();
```

Funções *arrow* também podem ser multi-linha, sendo declaradas da forma usual, com chaves. Funções assim precisam ter um comando de `return`.

```
let sum = (a, b) => {
  let result = a + b;
  return result;
};

console.log( sum(4, 2) ); // 6
```

Em casos assim não há muita diferença em se declarar uma função *arrow* multi-linha e se usar uma expressão de função. Isso vai do estilo de cada programador. É importante conhecer as duas formas porque elas podem aparecer em diferentes programas. (*Obs.:* Na verdade uma função *arrow* é só uma simplificação sintática da linguagem, sendo considerada uma expressão de função!)

3 Conclusão

Neste primeiro roteiro você aprendeu alguns conceitos básicos de JavaScript. Nos próximos roteiros, vamos começar a ver aspectos mais elaborados da linguagem, como orientação a objetos e funcionamento do motor JavaScript dentro de um navegador.

Para você ir se familiarizando com JS, não deixe de desenvolver os programas pedidos no bloco de Atividade da próxima página, utilizando os elementos da linguagem vistos aqui. Todos os exercícios pedidos são triviais para quem já tem um mínimo de conhecimento de programação. Mesmo que você consiga resolver as questões praticamente de cabeça; sente, escreva as suas soluções e teste. Essa é a única forma de se ganhar experiência com uma nova linguagem.

Atividade

Resolva os exercícios abaixo em JavaScript, executando-os no terminal. Os comandos de I/O devem operar sobre `stdin` e `out`, como visto ao longo do texto. Procure fazer pelo menos um pouco de higiene dos valores lidos do teclado, testando se eles são válidos. Use, por exemplo, a função `isNaN(x)`, que indica se o argumento `x` é um número válido ou não.

1. Leia o tempo de duração de um evento expresso somente em segundos e exiba a duração deste evento expresso em horas, minutos e segundos.
2. Faça um programa que recebe os coeficientes de uma equação do segundo grau (a, b e c), e a seguir calcula e exibe as raízes (x_1 e x_2) usando a fórmula de Bhaskara.
3. Leia três notas de um aluno e calcule a sua média, exibindo-a. Mostre também uma mensagem de "Aprovado", caso a média seja igual ou superior a 7.0; a mensagem "Recuperação", caso a média seja igual ou superior a 5.0 mas inferior a 7.0; ou a mensagem "Reprovado", caso a média seja inferior a 5.0.
4. Escreva um programa que leia um peso em Newtons (N) na Terra e o número de um planeta e imprime o valor do peso no outro planeta. A relação de planetas é dada na tabela abaixo, juntamente com o valor das gravidades relativas à Terra.

No.	Planeta	Gravidade relativa
1	Mercúrio	0.37
2	Vênus	0.88
3	Marte	0.38
4	Júpiter	2.64
5	Saturno	1.15
6	Urano	1.17
7	Netuno	1.18

5. Faça um programa que apresente na tela uma tabela de conversão de graus Celsius para Fahrenheit no intervalo de -100 ^{o}C a 100 ^{o}C com valores igualmente espaçados (de 5 ^{o}C em 5 ^{o}C). *Obs.:*
$$F = \frac{9}{5} * C + 32$$
6. Faça um programa que calcule uma aproximação do cosseno de um número x através de 20 termos da série abaixo:
$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots$$
7. A série de Fibonnacci, dada por $0, 1, 1, 2, 3, 5, 8, 13, 21,$ começa com os termos 0 e 1 e tem a propriedade de que cada termo subsequente é a soma dos dois termos precedentes. Escreva uma função que retorna o n-ésimo número de Fibonnacci, aonde n é o parâmetro da função. Utilizando essa função, exiba os 20 primeiros números da sequência.