Análise técnica de JavaScript - respostas às perguntas

1. Como instalar e começar a usar?
JavaScript é suportado nativamente pela maioria esmagadora dos navegadores modernos há muitos anos. Portanto, para utilizar a linguagem, basta ter um editor de texto e um navegador. O código pode ser carregado com um arquivo HTML cliente ou rodado diretamente no console do navegador em modo de desenvolvedor. É possível utilizar o interpretador Node.js (instalado separadamente) para executar códigos no terminal.

2. Quais são os processos de tradução utilizados?
Hoje em dia, pode-se dizer que JavaScript não é uma linguagem puramente interpretada; em seus primórdios, era comum interpretar diretamente o código-fonte da linguagem. No entanto, a fim de otimizar aplicações, diversos motores foram criados, como o V8 da Google, um motor de JavaScript amplamente usado que lança mão de otimizações JIT (*Just-in-time*). Como exemplo, o motor passa um arquivo JavaScript por um analisador léxico para construir a árvore sintática abstrata, que é prontamente utilizada pelo interpretador a fim de começar a execução rapidamente, ainda que sem otimizações. O Profiler observa o código em busca de possíveis pontos passíveis de otimização, cujos trechos de código são passados para o compilador gerar código de máquina — neste caso, bytecode a ser interpretado — que substituirá os trechos de código não otimizado.

3. Em que paradigmas se encaixa?
Multiparadigma, apesar de possuir fortes influências da programação funcional.

4. Os nomes são sensíveis à capitalização?
Sim. Por exemplo: O nome `func` é diferente e não conflita com o nome `Func`.

5. Quais os caracteres aceitos em um nome?
Quaisquer caracteres Unicode que sejam símbolos referentes a letras[1] (ex: "A", "Ý", "Ὼ", "א", "ツ", etc).

6. Existe alguma restrição de tamanho para nomes?
O ECMAScript (especificação para padronização de JavaScript, entre outras coisas) não menciona quaisquer limites de tamanho, e por testes verifica-se que o tamanho é praticamente ilimitado.

7. Como é a questão das palavras-chave x palavras reservadas?
Não existem palavras-chave, apenas palavras reservadas. Um bom número de palavras reservadas não são utilizadas pela linguagem.

8. É possível definir uma variável anônima? Mostre exemplo.
Não, entretanto é possível definir funções anônimas, que podem ser simplificadas com o uso de Arrow Functions.

9. A vinculação de tipos (tipagem) é estática ou dinâmica?

[1] https://stackoverflow.com/a/9337047/7841675

A vinculação é dinâmica, ocorrendo somente em tempo de execução. JavaScript só é estaticamente tipado se forem usadas ferramentas de linguagem como TypeScript para transpilar o código.

10. Quais categorias de variável (Sebesta, Seção 5.4.3) apresenta? Mostre exemplos.

- Estáticas
  Não, uma vez que JavaScript é dinamicamente tipada e uma variável só poderá ser carregada em tempo de execução;

- Dinâmica de pilha
  Apenas com tipos primitivos (Number, Boolean, referências a objetos, etc).

- Dinâmica de monte explícita
  Não. Todas as dinâmicas de monte são implícitas.

- Dinâmicas do monte implícitas
  A maioria das variáveis de JavaScript são dinâmicas do monte implícitas, isso se não forem todas e eu estiver na Disney;
  Ex: `highs = [74, 84, 86, 90, 71];`
  é uma variável dinâmica do monte implícita e pode mudar de tamanho ou tipo a qualquer momento durante a execução.

11. Permite ocultamento de nomes (variáveis) em blocos aninhados? Mostre exemplo.
Sim:

```
function func() {
    let text = 'fora';
    if (true) {
        let text = 'dentro';
        console.log(text); // dentro
    }
    console.log(text); // fora
}
func();
```

12. Permite definir constantes? Vinculação estática ou dinâmica? Mostre exemplos.
Sim, é possível definir constantes em JavaScript. O tipo de vinculação é dinâmica.
Ex:

```
const x = 200;
const nome = “Vítor”;
```

13. Quais os tipos oferecidos? Mostre exemplos de definição de variáveis de cada tipo.
Em JavaScript, temos 6 tipos primitivos de dados: `Number`, `Boolean`, `String`, `Undefined`, `Null` e `Symbol`, além do tipo `object`. Qualquer variável que tenha tipo diferente dos 6 primitivos é um `object`. Podemos checar o tipo dos exemplos usando o operador `typeof`:

```
typeof infinity; // “number”
typeof true; // “boolean”
typeof ‘a’; // “string”
var undef;
typeof undef // “undefined”, valor não foi inicializado.
typeof [1, 2, 4] // “object”
var Simb = Symbol(“Símbolo”)
typeof Simb // “symbol”
var lista = [‘a’, ‘b’, ‘c’]
typeof lista // “object”
```

14. Existe o tipo função? São cidadãs de primeira classe? Mostre exemplo.

Sim, o que significa que podem ser:

- Armazenadas em uma variável, objeto ou vetor
- Passadas como argumento em outra função
- Retornada em uma função

Ex.:

```
function add(x, y){
     return x + y;
}
console.log(typeof add); // function
```

Em JavaScript também é possível definir funções anônimas (também em forma de arrow functions) e closures:

```
var add = (x,y) => x + y;
console.log(add(2, 4)); // 6
```

15. Possui ponteiros ou referências? Permite aritmética de ponteiros?

JavaScript não possui ponteiros. Variáveis do tipo object têm sua referência passada por cópia para as funções.

16. Oferece coletor de lixo? Se sim, qual a técnica utilizada?

Sim. Os principais algoritmos ou técnicas usados são **referência de contagem** e **varredura e rotulação**.

No primeiro, um objeto pode ser coletado pelo *garbage collector* se não existir referência apontando para este objeto, tendo como principal desvantagem o fato de objetos com referências circulares não serem coletados (ainda que a execução não precise mais destes objetos).

O segundo é utilizado nativamente por todos os navegadores modernos desde 2012, com implementações melhoradas seguindo a mesma base de raciocínio: guarda-se uma lista de objetos chamada `roots`, a qual o coletor percorrerá encontrando todos os objetos referenciados. Os objetos não encontrados nesta busca serão coletados.

17. É possível quebrar seu sistema de tipos (forçar erro de tipo)? Mostre exemplo.

As variáveis são dinâmicas e podem mudar de tipo a qualquer momento. JavaScript possui o erro `TypeError`, lançado em situações como a seguinte:

```
var numero = 1;
```

```
try {
num.toUpperCase();    // Tenta converter número para caixa alta e
lança exceção
} catch(err) {
console.log(err.name); // 'TypeError'
}
```

18. Quais os operadores oferecidos? Mostre exemplo de uso de cada operador.

- Operadores de atribuição
  O operador de atribuição atribui um valor à direita do operador ao operando à esquerda.
  Ex: `x = y`

  Daí, surgem diversos operadores compostos:
  - Atribuição de adição
    `x += y` equivalente a `x = x + y`
  - Atribuição de subtração
    `x -= y` equivalente a `x = x - y`
  - Atribuição de multiplicação
    `x *= y` equivalente a `x = x * y`
  - Atribuição de divisão
    `x /= y` equivalente a `x = x / y`
  - Atribuição de resto
    `x %= y` equivalente a `x = x % y`
  - Atribuição exponencial
    `x **= y` equivalente a `x = x ** y`
  - Atribuição bit-a-bit por deslocamento à esquerda
    `x <<= y` equivalente a `x = x << y`
  - Atribuição bit-a-bit por deslocamento à direita
    `x >>= y` equivalente a `x = x >> y`
  - Atribuição de bit-a-bit deslocamento à direita não assinado
    `x >>>= y` equivalente a `x = x >>> y`
  - Atribuição AND bit-a-bit
    `x &= y` equivalente a `x = x & y`
  - Atribuição XOR bit-a-bit
    `x ^= y` equivalente a `x = x ^ y`
  - Atribuição OR bit-a-bit
    `x |= y` equivalente a `x = x | y`

- Operadores de comparação
  - Operadores de igualdade
    == para verificar igualdade de valor
    === para verificar igualdade de valor e de tipo
    Ex.: x == y
    x === y

    - Operadores de diferença
      != para verificar diferença de valor
      !== para verificar diferença de valor e de tipo
      Ex.: x != y
           x !== y
    - Operador maior que >
      Ex.: x > y
    - Operador menor que <
      Ex.: x < y
    - Operador maior ou igual que >=
      Ex.: x >= y
    - Operador menor ou igual que <=
      Ex.: x <= y

- Operadores aritméticos
    - Operador de soma +
      Ex.: x = y + z
    - Operador de subtração -
      Ex.: x = y - z
    - Operador de multiplicação *
      Ex.: x = y * z
    - Operador de exponenciação **
      Ex.: x = y ** z
    - Operador de divisão /
      Ex.: x = y / z
    - Operador de módulo %
      Ex.: x = y % z

- Operadores bit a bit
    - Operador AND &
      Ex.: x = y & z
    - Operador OR |
      Ex.: x = y | z
    - Operador NOT ~
      Ex.: x = ~y
    - Operador XOR ^
      Ex.: x = y ^ z
    - Operador shift para esquerda com 0 à esquerda <<
      Ex.: x = y << 1
    - Operador shift para direita com sinal >>
      Ex.: x = y >> 1
    - Operador shift para direita com 0 à direita >>>
      Ex.: x = y >>> 1

- Operadores lógicos
    - Operador AND &&

Ex.: x && y
    - Operador OR ||
      Ex.: x || y
    - Operador NOT !
      Ex.: !x

- Operador de string
  O operador de soma e a atribuição de adição podem ser usados em strings para concatenar strings. Se somar uma string com um valor numérico, o valor será implicitamente convertido para string.
  Ex.: x = “a” + “b”
  x+= “b”
  x = “a” + 2

- Operador condicional (ternário)
  Único operador com três operandos. Se condição for verdadeiro, retorna valorSe. Caso contrário, retorna valorCasoContrario.
  `condicao ? valorSe : valorCasoContrario`

- Operador vírgula
  O operador vírgula avalia o valor de seus operandos (da esquerda para a direita) e retorna o valor do último operando. O uso mais comum desse operador é suprir múltiplos parâmetros em um loop for.
  expr1, expr2, expr3…
  for (**var i = 0, j = 9**; i <= 9; **i++, j--**)

- Operadores unários
    - Operador de incremento ++
      Ex.: x++
    - Operador de decremento --
      Ex.: x--

19. Permite sobrecarga de operadores? Mostre exemplo.
Não. Existem operadores sobrecarregados disponíveis ao usuário, mas não é possível definir novos.

20. Quais operadores funcionam com avaliação em curto-circuito?
`&&` e `||`, em que o operador `&&` realiza uma operação de AND entre duas expressões; se a primeira possuir valor equivalente a `false`, a segunda expressão sequer é avaliada. Analogamente com o operador ||; este realiza a operação OR entre duas expressões: se a primeira for equivalente a `true`, a segunda operação não é avaliada.

21. O operador de atribuição funciona como uma expressão?
Sim. Exemplo:
```
const a = b = c = 1;
```

```
console.log(a); // 1
console.log(b); // 1
console.log(c); // 1
```

22. Quais as estruturas de controle (seleção, iteração) oferecidas? Mostre exemplos.
JavaScript conta com todas as estruturas de controle usuais de linguagens de programação, além da `with`, descrita posteriormente:

if:

```
if (a === b){
  console.log("a");
}
```

while:

```
while (a != ".") {
      ...
}
```

do; while:

```
do {
      ...
} while (a < 5);
```

for:

```
for (let i = 0; i < n; i++) {
      ...
}
for (element in object) {
      ...
}
```

switch:

```
switch (variable) {
      case 'a': …
            break;
      case "ab": ...
            break;
}
```

with:

```
with (Math) {
      ...
}
```

No exemplo, with abre o escopo da biblioteca Math dentro do bloco. Nela é possível usar seus métodos e variáveis.

23. Quais sentenças de desvio incondicional oferecidas? Mostre exemplos.
`Return`, `break`, `continue` e (fora das convenções da ECMAScript) `goto`. Exemplos:

```
function func() {
  console.log("a");
  return;
  console.log("b");
}
func() // a

function fun() {
    for(let i = 0; i < 5; i++) {
        console.log(i);
        break;
    }
}
fun() // 0

function f1() {
     let x = 0;
     for(let i = 0; i < 5; i++) {
          if (i % 2 == 0) {
               continue;
          }
          x += i;

     }
     console.log(x);
}

f1(); // 4
```

24. Quais os métodos de passagem de parâmetros oferecidos? Mostre exemplos.
Em JavaScript, a passagem por referência só acontece para valores compostos como Arrays ou Objetos. Para os demais tipos, a passagem é por valor.

Exemplo:

```
let x = {'nome': 'Philipe', 'idade': '21'};
let y = 3;
let z = [1,2,3]

function f(x) {
     y = 2;
     x['nome'] = 'Tiago';
     z[2] = 4;
```

```
}
console.log(y); // 2
console.log(x); // { nome: 'Philipe', idade: '21' }
console.log(z); // [1,2,3]
f(x, y, z);
console.log(y); // 3
console.log(x); //{ nome: 'Tiago', idade: '20' }
console.log(z); // [1,2,4]
```

25. Permite sobrecarga de subprogramas? Mostre exemplo.

Não, apenas sobrescrita:

```
function adiciona(produto) {
  console.log(produto);
}
adiciona(1)

function adiciona(produto, categoria) {
  console.log(categoria);
 // essa função "sobrescreve" a anterior
}
adiciona(1, 2)
```

Porém, é possível forjar uma sobrecarga (pois é possível chamar funções mesmo sem passar todos parâmetros):

```
function foo(a, b, args) {
   if (args['test']) { console.log(args); } // se test existir,
faça algo
}

foo(1, 2, {"method":"add"});
foo(3, 4, {"test":true, "foo":"bar"});
```

26. Permite subprogramas genéricos? Mostre exemplo.

Todo subprograma é genérico por padrão, uma vez que JavaScript é uma linguagem fracamente tipada. Exemplo:

```
function func(x, y) {
    return x + y
}

console.log(func(2, 1)) // 3
console.log(func("a", 1)) // "a1"
```

27. Como é o suporte para definição de Tipos Abstratos de Dados? Mostre exemplo.

Tem suporte por meio do uso de classes (a partir do ECMAScript 15). No entanto, não há modificadores de acesso para ocultamento de informações, como `private`. Portanto os

atributos podem ser modificados diretamente. Porém é possível com o uso de funções realizar um encapsulamento em que os valores não são modificados nem vistos.

Classe:

```
class Stack {
  constructor() {
    this.height = 0;
    this.data = [];
  }
  insert(object) {
     this.height;
    this.data.push(object);
  }
}

let s = new Stack();
s.insert({'nome': 'Philipe'});
s.insert({'nome': 'Tiago'});
s.insert({'nome': 'Atílio'});
console.log(s.data); // [ { nome: 'Philipe' }, { nome:
'Tiago' }, { nome: 'Atílio' }
```

Função “construtora”:

```
function Stack(){
  var data = [];
  var height = 0;
  this.insert = (object) => {
    data.push(object);
    height++;
  }
  this.pop = (object) => {
    data.pop();
    height--;
  }
  this.getData = () => {
    return data;
  }
  this.getHeight = () => {
    return height;
  }
}

let s = new Stack();
s.insert({'nome': 'philipe'});
s.insert({'nome': 'Tiago'});
s.insert({'nome': 'Atílio'});
```

```
console.log(s.data); // undefined
console.log(s.height); // undefined
console.log(s.getData());   [{'nome':   'philipe'},   {'nome':
'Tiago'}, {'nome': 'Atílio'}]
console.log(s.getHeight()); // 3
```

28. Permite TADs genéricos/parametrizáveis? Mostre exemplo.
Permite. O exemplo acima é um tipo de TAD genérico que é capaz de ocorrer devido a tipagem dinâmica do JavaScript. Qualquer elemento de qualquer tipo que se tentar inserir com o método push será aceito pela linguagem.

29. Quais as construções de encapsulamento oferecidas? Mostre exemplos.
Definição de classes e divisão de código em múltiplos arquivos. Não possui suporte a ocultamento de informação por meio de modificadores de acesso. Podemos, no entanto, criar variáveis locais em classes e funções com `var`, `let` e `const`, não sendo acessíveis em outro escopo.

Exemplo de classe:
```
class Retangulo {
  constructor(altura, largura) {
    this.altura = altura;
    this.largura = largura;
  }
}
```
Note que `altura` e `largura` são acessíveis publicamente.

Variáveis locais ocultas:
```
function f() {
  var a = 1;
  const b = 2;
  let c = 3;
  d = 4;
}
f()
console.log(a); // ReferenceError: a is not defined
console.log(b); // ReferenceError: b is not defined
console.log(c); // ReferenceError: c is not defined
console.log(d); // 4
```

30. Quais tipos de polimorfismo suporta? Mostre exemplos.
Ad-hoc apenas. Por coerção:
```
x = 1;
y = '1';
console.log(x + y); // '11'
```
Por sobrecarga:

Em JavaScript, é possível chamar funções mesmo não passando todos parâmetros, comportando-se como um polimorfismo ad-hoc de sobrecarga. Exemplo:

```
function f(a) {
  console.log(a);
}
f(1) // 1
f(1, 2) // 1
f() // undefined
```

31. Permite herança de tipos? Herança múltipla? Mostre exemplo.

Permite herança simples facilmente, exemplo:

```
class Animal {
      constructor(nome) {
      this.nome = nome;
      }
}

class Gato extends Animal {
      constructor(nome, raça) {
      super(nome);
      this.raça = raça;
      }
}
```

A herança múltipla é consideravelmente mais complicada e necessita de muitas voltas com as funcionalidades da linguagem para ser implementada.

32. Permite sobrescrita de subprogramas? Mostre exemplo.

Sim. Exemplo:

```
class A {
  speak() {
    console.log("Sou A");
  }
}
class B extends A {
  speak() {
    super.speak();
    console.log("Sou B");
  }
}

var a = new A();
a.speak(); // Sou A

var b = new B();
b.speak();
// Sou A
// Sou B
```

33. Permite a definição de subprogramas abstratos? Mostre exemplo.
Talvez.
Com sobrescrita em herança (a sobrescrita não é obrigatória, como em outras linguagens):

```
class A {
  speak() { // subprograma abstrato
  }
}
class B extends A {
  speak() {
    console.log("Sou B");
  }
}
```

JavaScript não obriga a sobrescrita da função, fazendo com que se aproxime mais de uma função virtual do que de uma abstrata.

34. Oferece mecanismo de controle de exceções? Mostre exemplo.
O programador pode criar blocos de `try` para controlar exceções lançadas no escopo, tratando-as no bloco `catch`.

```
try {
      var x = y;
} catch(err) {
      console.log(err); // ReferenceError: y is not defined
}
```

35. Possui hierarquia de exceções controlada, como em Java? Qual a raiz?
JavaScript possui hierarquia de erros, lançadas em tempo de execução e originadas da raiz `Error`. Tipos mais comuns: `SyntaxError`, `ReferenceError`, etc.

Exemplo:

```
try {
    const x = y;
}   catch (err) {
    console.log(err);
}
```

Retorno ao executar o programa: "`ReferenceError: y is not defined`"

36. Categoriza as exceções em checadas e não checadas? Como?
Não, visto que o código não é compilado antes da execução, ou seja, todas as exceções são lançadas em tempo de execução e que a linguagem não obriga a declaração de exceções lançadas. Portanto, todas as exceções são não checadas.

37. Obriga a declaração de exceções lançadas para fora de um subprograma?
Não, uma vez que não é feita nenhuma verificação de exceções antes de execução e que código JavaScript não é compilado antes da execução.

38. Como você avalia a LP usando os critérios do Sebesta (Seção 1.3)?

- Simplicidade

Pode-se dizer que JavaScript perde simplicidade por conta de seus operadores sobrecarregados, múltiplas construções que alcançam um mesmo fim e funcionamentos inusitados (ver a seguir).

- Ortogonalidade
  JavaScript possui diversos momentos em que apresenta comportamento confuso. Estes momentos partem inclusive do uso operações básicas definidas pela linguagem. Portanto, podemos dizer que possui baixa ortogonalidade.Alguns exemplos:

```
>0.1+0.2==0.3          >{}+[]                  >91-"1"
false                  0                       90

>typeof NaN            >[10,      1,           >Math.min()
"number"               3].sort()               Infinity
                       [1, 10, 3]
```

- Tipos de dados
  Possui todos os tipos de dados usuais de linguagens de programação moderna, como booleanos, listas, dicionários, etc.

- Projeto de sintaxe
  Possui sintaxe concisa e permite nomes com caracteres especiais. O uso de `var` vs `let` pode causar algumas dúvidas.

- Suporte para abstração
  Possui bom suporte para abstração, sendo capaz de definir classes (a partir do ECMAScript 2015), funções construtoras, subprogramas e subprogramas aninhados.

- Expressividade
  Possui diversos operadores poderosos, como "+" que opera com diferentes tipos, e várias estruturas de controle, métodos, etc.

- Verificação de tipos
  Não existe em JavaScript (é possível checar tipos em tempo de execução através da função `typeof`). A alternativa é a utilização de TypeScript.

- Tratamento de exceções
  Possui suporte com `try-catch`, mas deixa passar muitos potenciais erros sem interromper a execução do programa, podendo causar futuras dores de cabeça.

- Apelidos restritos
  Possibilita utilização de apelidos com objetos. Ex:

```
> a = {'a': 2}
{ a: 2 }
> b = a
{ a: 2 }
```

```
> b['a'] = 3
3
> b
{ a: 3 }
> a
{ a: 3 }
```

39. Como você avalia a LP usando os critérios do Varejão?

- Legibilidade
  Por não ser uma linguagem verbosa, a legibilidade de JavaScript já possui um ponto a seu favor. No entanto, existem muitas formas de se executar uma mesma ideia, seja por diferentes funções/métodos *built-in* ou *runtimes* (JavaScript "puro" em navegadores vs Node.js vs alternativas), o que prejudica a legibilidade se o leitor não possuir muito conhecimento acerca da linguagem.

- Redigibilidade (facilidade de escrita)
  Escrever JavaScript é muito simples graças aos tipos de dados oferecidos e os métodos prontos para manipulá los. Um dos motivos de JavaScript ser considerada uma linguagem de alto nível é sua facilidade de escrita.

- Confiabilidade
  A confiabilidade de JavaScript é significativamente reduzida devido à falta de checagem de tipos e, também, pela coerção de tipos realizada pela linguagem, fazendo com que muitas operações indevidas e que em outras linguagens trariam erros em tempo de compilação - ou ao menos mais cedo em execução - sejam levadas adiante no programa. Isso faz com que seja muito mais difícil depurar e/ou identificar problemas simples.

- Ortogonalidade
  Ver questão 38.

- Eficiência
  Apesar de ser uma linguagem interpretada, JavaScript consegue ter uma eficiência considerável, ainda mais ao considerar-se o uso da V8 engine e otimizações através de compilação *just-in-time*.

- Facilidade de aprendizado
  JavaScript possui com uma curva de aprendizado bem suave, podendo ser uma boa porta de entrada para novos programadores. Alguns conceitos que poderiam ser impedimentos são abstraídos e a linguagem conta com muitas funcionalidades (métodos, bibliotecas) prontas para uso *out-of-the-box*.

- Reusabilidade
  Código JavaScript pode ser facilmente reutilizado através da modularização de arquivos ou até mesmo do uso de pacotes de terceiros, através de, por exemplo, um gerenciador de pacotes como `npm` (com Node.js).

- Modificabilidade
  Um sistema escrito em JavaScript pode ser facilmente modificado pois ele permite a separação de interface e implementação, além de possuir um bom suporte a POO.

- Portabilidade
  Alta, pois um mesmo código JavaScript pode ser rodado em diversos sistemas operacionais e plataformas diferentes e até mesmo em navegadores com pouca ou nenhuma modificação de código sendo necessária.