Depuração / JAVASCRIPT

Usar o console do JavaScript para verificar o valor de uma variável

O Chrome e o Firefox possui excelentes consoles JavaScript, também conhecidos como DevTools, para depurar seu JavaScript.

Você pode encontrar as Developer tools no menu do seu Chrome ou Web Console no menu do Firefox. Se você estiver usando um navegador diferente, ou um telefone móvel, recomendamos fortemente mudar para o Firefox ou Chrome Desktop.

O método `console.log()`, o qual "imprime" a saída do que está nos seus parênteses no console, provavelmente será a ferramenta de debug mais útil. Colocá-lo em pontos estratégicos no seu código pode mostrar a você os valores intermediários de variáveis. É uma boa prática ter uma ideia do que deveria ser a saída antes de olhar o que é. Ter pontos de verificação para ver o status de seus cálculos ao longo do seu código ajudará a encontrar onde o problema está.

Aqui está um exemplo para imprimir a string `Hello world!` no console:

```
console.log('Hello world!');
```

Exercicio = Use o método `console.log()` para imprimir o valor da variável `a` aonde anotou no código.

```
let a = 5;

let b = 1;

a++;

// Altere apenas o código abaixo desta linha



let sumAB = a + b;

console.log(a);
```

Entender a diferença entre o console do freeCodeCamp e do navegador

Você pode ter percebido que alguns dos desafios do freeCodeCamp incluem seu próprio console. Este console se comporta um pouco diferente do console do navegador.

Existem muitos métodos para usar no `console` para exibir mensagens. `log`, `warn` e `clear` são alguns deles. O console do freeCodeCamp só produzirá mensagens de `log`. Já o console do navegador mostrará todas as mensagens. Quando você fizer alterações em seu código, ele será executado e mostrará automaticamente os logs. O console do freeCodeCamp é limpo cada vez que o código é executado.

Exercicio = Primeiro, abra o console do navegador para poder ver os registros (logs). Para fazer isso, clique com o botão direito na barra de navegação do freeCodeCamp, na parte superior, e clique em `inspect` na maioria dos navegadores. Em seguida, encontre a aba `console` na janela que é aberta.

Depois disso, use `console.log` para registrar a variável `output`. Veja os dois consoles para ver o log. Por fim, use `console.clear` depois do registro para limpar o console do navegador. Veja a diferença entre os dois consoles.

```
let output = "Get this to show once in the freeCodeCamp
console and not at all in the browser console";

console.log(output);

console.clear();
```

Usar typeof para verificar o tipo da variável

Você pode usar `typeof` para verificar a estrutura de dado, ou tipo, de uma variável. Isso é útil na depuração quando trabalhando com diversos tipos de dados. Se você pensar que está adicionando dois números, mas na verdade um é na verdade uma string, o resultado pode ser inesperado. Erros de tipo podem se esconder em cálculos ou chamada de funções. Seja cuidadoso especialmente quando você estiver acessando e trabalhando com dados externos na forma de um objeto JavaScript Object Notation (JSON).

Aqui está alguns exemplos usando `typeof`:

```
console.log(typeof "");
console.log(typeof 0);
console.log(typeof []);
console.log(typeof {});
```

Em ordem, o console exibirá as strings `string`, `number`, `object` e `object`.

JavaScript reconhece sete tipos de dados primitivos (imutáveis): `Boolean`, `Null`, `Undefined`, `Number`, `String`, `Symbol` (novo na ES6) e `BigInt` (novo na ES2020), além de um tipo para itens mutáveis: `Object`. Note que em JavaScript, arrays são tecnicamente um tipo de objeto.

Exercicio = Adicione duas instruções `console.log()` para verificar o `typeof` de cada uma das duas variáveis `seven` e `three` no código.

```
let seven = 7;

let three = "3";

console.log(seven + three);

// Altere apenas o código abaixo desta linha

console.log(typeof seven);

console.log(typeof three);
```

Obs : quando coloco o typeof vai me mostrar o que é o obejto se é uma string... se é um número e etc.

Capturar nomes de variáveis e funções com erros ortográficos

Os métodos `console.log()` e `typeof` são duas formas primárias para verificar valores intermediários e tipos de saída do programa. Agora é hora de conhecer as formas comuns que um bug assume. Um problema de nível de sintaxe que digitadores rápidos podem deixar passar é um simples erro de digitação incorreta.

Caracteres deslocados, faltando ou capitalizados erroneamente em um nome de variável ou função farão com que o navegador procure por um objeto que não existe - e reclame na forma de um erro de referência. Variáveis e funções JavaScript são sensíveis a caracteres maiúsculos e minúsculos.

Corrija os dois erros de ortografia no código para que o cálculo `netWorkingCapital` funcione.

```
let receivables = 10;

let payables = 8;

let netWorkingCapital = receivables - payables;

console.log(`Net working capital is: ${netWorkingCapital}`);
```

Identificar parênteses, colchetes, chaves e aspas sem fechamento

Outro erro de sintaxe para estar ciente é que todas as aberturas de parênteses, colchetes, chaves e aspas têm um par de fechamento. Esquecer um pedaço tende a acontecer quando você está editando um código existente e inserindo itens com um dos tipos de pares. Além disso, tenha cuidado ao aninhar blocos de código em outros, como ao adicionar uma função de callback como um argumento a um método.

Uma maneira de evitar esse erro é, assim que o caractere de abertura é digitado, incluir imediatamente o caractere de fechamento, mover o cursor entre eles e continuar programando. Felizmente, a maioria dos editores de código modernos geram a segunda parte do par automaticamente.

Exercicio = Corrija os dois erros de pares no código.

```
let myArray = [1, 2, 3];

let arraySum = myArray.reduce((previous, current) =>
previous + current);

console.log(`Sum of array values is: ${arraySum}`);
```

Identificar uso misto de aspas simples e duplas

JavaScript nos permite o uso de aspas simples (') e duplas (") para declarar uma string. Decidir qual delas usar geralmente é uma questão de preferência pessoal, com algumas exceções.

Ter duas opções é ótimo quando uma string possui contrações ou outros pedaços de texto que estão entre aspas. Apenas tome cuidado para que você não feche uma string muito cedo, o que causa erro de sintaxe.

Aqui estão alguns exemplos de mistura de aspas:

```
const grouchoContraction = "I've had a perfectly wonderful
evening, but this wasn't it.";
const quoteInString = "Groucho Marx once said 'Quote me as
saying I was mis-quoted.'";
const uhOhGroucho = 'I've had a perfectly wonderful evening,
but this wasn't it.';
```

As duas primeiras estão corretas, mas a terceira não.

Claro, não há problema em usar apenas um estilo de aspas. Você pode escapar as aspas dentro de uma string ao usar o caractere barra invertida (\):

```
const allSameQuotes = 'I\'ve had a perfectly wonderful
evening, but this wasn\'t it.';
```

Exercicio = Corrija a string para que use aspas diferentes para o valor de `href` ou escape as aspas existentes. Mantenha as aspas duplas ao redor de toda a string.

```
let innerHtml = "<p>Click here to <a href=\"#Home\">return home</a></p>";

console.log(innerHtml);
```

Identificar uso do operador de atribuição ao invés do operador de igualdade

Programas de ramificação, ou seja, programas que fazem coisas diferentes se certas condições forem atendidas, dependem de instruções `i`, `else if` e `else` em JavaScript. Às vezes a condição verifica se um resultado é igual a um valor.

Essa lógica é dita da seguinte forma: "se x for igual a y, então ..." o que pode literalmente ser traduzido para código usando o `=`, ou operador de atribuição. Isso leva a um controle de fluxo inesperado no seu programa.

Como abordado nos desafios anteriores, o operador de atribuição (`=`) em JavaScript, atribui um valor para o nome de uma variável. E os operadores `==` e `===` verificam pela igualdade (o triplo `===` testa por igualdade estrita, significando que ambos os valores e o tipo de dado são os mesmos).

O código abaixo atribui o valor 2 para `x`, o que tem como resultado `true`. Quase todo valor por si só em JavaScript é avaliado como `true`, exceto os que são conhecidos como valores "falsy": `false`, `0`, `""` (uma string vazia), `NaN`, `undefined` e `null`.

```
let x = 1;
let y = 2;
if (x = y) {

} else {

}
```

Nesse exemplo, o bloco de código dentro da instrução `if` vai rodar para qualquer valor de `y`, a não ser que `y` seja falso. O bloco de `else`, que nós esperamos que seja executado aqui, não vai rodar de fato.

Exercicio = Corrija a condição para que o programa rode na ramificação correta e o valor apropriado seja atribuído a `result`.

```
let x = 7;

let y = 9;

let result = "to come";



if(x == y) {
  result = "Equal!";
} else {
  result = "Not equal!";
}



console.log(result);
```

Capturar abertura e fechamento de parênteses faltantes após uma chamada de função

Quando uma função ou método não recebe nenhum parâmetro, você pode esquecer de incluir a abertura e fechamento de parênteses (vazio) ao chamá-la. Frequentemente, o resultado de uma chamada de função é salva em uma variável para outro uso em seu código. Esse erro pode ser detectado ao exibir no console os valores das variáveis (ou seus tipos) e verificar que uma variável está definida para uma referência de uma função, ao invés do valor esperado que a função retorna.

As variáveis no seguinte exemplo são diferentes:

```
function myFunction() {
  return "You rock!";
}
let varOne = myFunction;
```

```
let varTwo = myFunction();
```

Aqui `varOne` é a função `myFunction` e `varTwo` é a string `You rock!`.

Exercicio = Corrija o código para que a variável `result` seja definida para o valor retornado da chamada da função `getNine`.

```
function getNine() {
  let x = 6;
  let y = 3;
  return x + y;
}

let result = getNine();
console.log(result);
```

Obs : quando eu colocar apenas o nome da função, só vai retornar ela, mas depois que eu coloco a função e os () ... retorna o valor da função !

Capturar argumentos passados na ordem errada ao chamar uma função

Continuando a discussão sobre chamada de funções, o próximo bug para prestar atenção é quando os argumentos de uma função são fornecidos na ordem incorreta. Se os argumentos forem de diferentes tipos, tal como uma função esperando um array e um inteiro, isso provavelmente lançará um erro de tempo de execução. Se os argumentos são do mesmo tipo (todos os inteiros, por exemplo), então a lógica do código não fará sentido. Certifique-se de fornecer todos os argumentos exigidos, na ordem adequada para evitar esses problemas.

Exercicio = A função `raiseToPower` eleva uma base para um expoente. Infelizmente, não é chamada corretamente - corrija o código para que o valor de `power` seja o 8 esperado.

```
function raiseToPower(b, e) {
  return Math.pow(b, e);
}
```

```
let base = 2;
let exp = 3;
let power = raiseToPower(base, exp);
console.log(power);
```

Capturar erros de fora por um ao usar a indexação

Off by one errors (erros de fora por um) (as vezes chamados de OBOE) surgem quando você está tentando acessar um índice específico de uma string ou array (para fatiar ou acessar um segmento), ou quando você está iterando sobre seus índices. A indexação de JavaScript começa em zero e não um, o que significa que o último índice sempre será o tamanho do item menos 1 (array.length - 1). Se você estiver tentando acessar um índice igual ao tamanho, o programa pode lançar uma referência do erro "index out of range" ou imprimir `undefined`.

Quando você usa métodos de string ou array que recebem intervalos de índices como argumentos, auxilia ler a documentação e compreender se eles são inclusivos (o item no índice especificado é parte do que é retornado) ou não. Aqui estão alguns exemplos de erros de fora por um:

```
let alphabet = "abcdefghijklmnopqrstuvwxyz";
let len = alphabet.length;
for (let i = 0; i <= len; i++) {
  console.log(alphabet[i]);
}
for (let j = 1; j < len; j++) {
  console.log(alphabet[j]);
}
for (let k = 0; k < len; k++) {
  console.log(alphabet[k]);
}
```

O primeiro exemplo itera uma vez a mais (i <= len) e o segundo itera uma vez a menos por começar do índice 1 (let j = 1). O terceiro exemplo está certo.

Exercicio = Corrija os dois erros de índices nas funções seguintes para que todos os números de 1 até 5 sejam exibidos no console.

```
function countToFive() {
  let firstFive = "12345";
  let len = firstFive.length;
  // Altere apenas o código abaixo desta linha
  for (let i = 0; i < len; i++) {
  // Altere apenas o código acima desta linha
    console.log(firstFive[i]);
  }
}

countToFive();
```

Ter cuidado quando reinicializar variáveis dentro de laços

Às vezes é necessário salvar informações, incrementar contadores ou redefinir variáveis dentro de um laço. Um problema em potencial é quando variáveis deveriam ser reinicializadas e, não são, ou vice versa. Isso é particularmente perigoso se você acidentalmente redefinir a variável sendo usada para a a condição de parada, causando um laço infinito.

Imprimir os valores das variáveis em cada ciclo do seu laço usando `console.log()` pode ajudar a descobrir comportamentos com bugs relacionados a reiniciar ou falhar ao reiniciar uma variável.

Exercicio = A seguinte função deveria criar um array de duas dimensões com `m` linhas e `n` colunas de zeros. Infelizmente, não está produzindo a saída esperada porque a variável `row` não está sendo reiniciada (definida de volta para um array vazio) no laço mais externo. Corrija o código para que retorne o array de zeros correto (dimensão: 3x2), que se parece com `[[0, 0], [0, 0], [0, 0]]`.

```
function zeroArray(m, n) {
  let newArray = [];
  for (let i = 0; i < m; i++) {
    let row = []; /* <-----  row has been declared inside the outer loop.
     Now a new row will be initialised during each iteration of the outer loop allowing
     for the desired matrix. */
    for (let j = 0; j < n; j++) {
      row.push(0);
    }
    newArray.push(row);
  }
  return newArray;
```

```
}

let matrix = zeroArray(3, 2);

console.log(matrix);
```

Prevenir laços infinitos com uma condição de término válida

O tópico final é o temido laço infinito. Laços são ótimas ferramentas quando você precisa que o seu programa rode um bloco de código uma quantidade exata de vezes ou até que uma condição seja atendida, mas eles precisam de uma condição de parada que finalize esse laço. Laços infinitos têm alta probabilidade de congelar ou travar o navegador, e causa um caos na execução geral do programa, o que ninguém deseja.

Havia um exemplo de laço infinito na introdução dessa seção - esse laço não tinha uma condição de parada para sair do laço `while` dentro de `loopy()`. NÃO chame essa função!

```
function loopy() {
  while(true) {
    console.log("Hello, world!");
  }
}
```

É trabalho do programador garantir que a condição de parada, a qual avisa ao programa quando sair de um laço, seja eventualmente alcançada. Um erro é incrementar ou decrementar uma variável contadora na direção errada da condição de parada. Outro erro é acidentalmente reiniciar uma variável contadora ou de índice dentro do laço, ao invés de incrementar ou decrementar.

Exercicio = A função `myFunc()` contém um laço infinito porque a condição de parada `i != 4` nunca será `false` (para, então, quebrar o laço) - `i` vai incrementar em 2 a cada iteração, e passa direto por 4 já que `i` é ímpar no

início. Corrija o operador de comparação para que o laço só rode enquanto `i` for menor ou igual a 4.

```
function myFunc() {
  for (let i = 1; i <= 4; i += 2) {
    console.log("Still going!");
  }
}
```