ntander Coders 2024.1 | Front-End | #1178

aços de repetição condicionais (WHILE DO-WHILE)

Isaias Soares
Id: 1178008

Laços de repetição condicionais (WHILE DO-WHILE)

Muitas coisas na programação são apenas repetições e os computadores são extremamente bons em executar repetições. Para executá-las usamos estruturas chamadas laços ou loops.

Existem no JavaScript três tipos de laços: `while`, `do-while` e `for`. Nesse tópico abordaremos `while` e `do-while`, que são laços condicionais, ou seja, o critério de parada é um teste lógico da mesma forma que fazemos no `if`.

Quando usamos laços condicionais o valor `true` mantém o laço repetindo e o valor `false` encerra o laço. Veremos os laços `for` mais para frente junto com os arrays e matrizes onde eles realmente brilham. Vamos começar com exemplo simples:

Escrevendo no console 10 vezes:

```
let count = 0;

while(count < 10){
    console.log('hello');
    count = count + 1;
}
```

No exemplo, o while é usado em conjunto com uma variável para controlar quantas vezes ele rodará. A variável será incrementada a cada rodada `count = count + 1` e quando seu valor for 10 a condição se torna falsa e o laço para.

Whiles não são a melhor solução para fazer repetições com controle numérico, o `for` é mais adequado para esses casos, mas serve bem como exemplo. Vamos olhar para algo um pouco mais complexo onde o while será a melhor opção:

Jogo de acertar o número:

```
//Função para gerar um número inteiro randômico de acordo com um intervalo
function randomIntFromInterval(min, max) {
  return Math.floor(Math.random() * (max - min + 1) + min);
}

//Programa
const target = randomIntFromInterval(1, 50);
let guess = parseInt(prompt('digite um número entre 1 e 50'));

while(guess !== target){
    console.log('Esse não é o número correto, tente novamente...');
    guess = parseInt(prompt('digite um número entre 1 e 50'));
}

console.log('Você acertou!');
```

No exemplo acima, fizemos:

- Uma função que sorteia um número em um determinado intervalo. Usamos ela para sortear um número de 1 até 50
- Sorteamos o número e guardamos na constante `target`
- Também inicializamos `guess` a partir de um "chute" do usuário
- Comparamos se `guess` e `target` são diferentes e, enquanto forem, o laço continuará rodando

Dentro do laço, solicitamos um número ao usuário até que ele acerte o valor de `target`. Quando ele acerta, o laço para e finalmente a linha abaixo dele poderá rodar, escrevendo "Você acertou!" na tela. Vamos explorar um pouco o laço `do-while` agora.

A única diferença entre `while` e `do-while` é o momento em que a condição é testada. No `while`, como vimos, a condição é testada imediatamente, por isso tivemos que repetir a linha que solicita o número antes do laço e dentro do laço. O que não é conveniente. No `do-while` o teste é feito no final, o que nos dá a oportunidade de já solicitar ao usuário o primeiro valor, dentro do laço, mas antes de validar a condição.

É importante observar que é possível fazer qualquer coisa que faríamos com `do-while` usando `while`, no entanto, em grande parte dos casos isso levaria à repetição de código para compensar o teste logo no início.

Vamos começar reescrevendo o código acima com `do-while`:

Jogo de acertar o número:

```
//Função para gerar um número inteiro randômico de acordo com um intervalo
function randomIntFromInterval(min, max) {
  return Math.floor(Math.random() * (max - min + 1) + min);
}

//Programa
const target = randomIntFromInterval(1, 50);

do{
    var guess = parseInt(prompt('digite um número entre 1 e 50'));
    if(guess !== target)
      console.log('Tente novamente');
}
while(guess !== target);
```

```
console.log('Você acertou!');
```

A diferença fundamental entre o exemplo anterior e esse é que o corpo da repetição está na diretiva `do` que é executada imediatamente (ao menos uma vez) e só depois o teste da condição é feito no `while`.

Viu que usamos `var`? Vou aproveitar esse exemplo para falar um pouco mais de hoisting que deixamos em aberto no tópico de variáveis e aproveitar para falar de escopo. Se declaramos uma variável dentro das chaves de uma expressão, como `if`, `else`, `do`, ou qualquer outro comando que tenha um corpo delimitado por chaves, o ciclo de vida dessa variável termina quando as chaves de fecham. Sendo assim, variáveis declaradas nesse escopo, estarão presentes apenas dentro desse escopo, depois dele elas são eliminadas da memória.

Variáveis declaradas com `var` são exceção, por conta do hoisting sua declaração é elevada ao escopo maior. No exemplo acima, a variável `guess` é lida após o final do escopo onde foi declarada (após o fechamento da chave do `do`), mas o hoisting levanta a declaração dela para antes do `do` para que ela esteja disponível depois do escopo. Na prática, o código acima é interpretado pelo JavaScript como sendo:

Jogo de acertar o número:

```
//Função para gerar um número inteiro randômico de acordo com um intervalo
function randomIntFromInterval(min, max) {
  return Math.floor(Math.random() * (max - min + 1) + min);
}

//Programa
const target = randomIntFromInterval(1, 50);
var guess;

do{
    guess = parseInt(prompt('digite um número entre 1 e 50'));
```

```
    if(guess !== target) console.log('Tente novamente');
}
while(guess !== target);

console.log('Você acertou!');
```

Observe a diferença, declaramos `guess` logo após `target` para que ela use o escopo anterior ao `do` o que permite que ela esteja disponível para nós após o final da expressão. Nessa versão do código não dependemos mais do hoisting porque fizemos manualmente o que ele faz automaticamente. Sendo assim, podemos trocar `var` por `let` sem prejuízos.

Jogo de acertar o número:

```
//Função para gerar um número inteiro randômico de acordo com um intervalo
function randomIntFromInterval(min, max) {
  return Math.floor(Math.random() * (max - min + 1) + min);
}

//Programa
const target = randomIntFromInterval(1, 50);
let guess;

do{
    guess = parseInt(prompt('digite um número entre 1 e 50'));
    if(guess !== target) console.log('Tente novamente');
}
while(guess !== target);

console.log('Você acertou!');
```

Especialmente em laços queremos que nossas variáveis tenham seu escopo controlado, afinal na maioria das vezes não queremos que o valor de uma variável na rodada anterior influencie nas rodadas seguintes. Portanto, a melhor prática é sempre declarar as variáveis no escopo mais restrito possível, para que assim elas saiam da memória o mais rápido possível e possam ser declaradas novamente em outros contextos (sim, uma vez que o escopo terminou você pode declarar variáveis com mesmo nome). Por isso evitamos hoisting usando apenas `let` e `const`.

Vamos a mais um exemplo usando do-while. Há muito tempo, nessa galáxia mesmo, havia jogos de RPG de mesa onde os jogadores interpretavam seus personagens e todos os conflitos eram resolvidos por rolagem de dados. O mais popular desses jogos era o D&D (Dungeons and Dragons). Talvez você conheça por ter jogado, ou por ter assistido Stranger Things, no Netflix. Os dados usados no D&D eram bastante variados, tínhamos dados de 4, 6, 8, 10, 12 e 20 faces. Para simplificar, chamamo-os de D4, D6, D8, D10, D12 e D20.

Por que não aproveitar a função de gerar números aleatórios do exemplo anterior para construir um rolador de dados?

Solicitaremos um dado ao usuário, usaremos um switch para calcular um valor presente no dado escolhido e mostraremos ele na tela.

Exemplo: Rolador de dados

```
//Função para gerar um número inteiro randômico de acordo com um intervalo
function randomIntFromInterval(min, max) {
    return Math.floor(Math.random() * (max - min + 1) + min);
}

//Programa
do {
    const dado = parseInt(prompt('digite 4, 6, 8, 10, 12 ou 20 para rolar o dado:'));
    let resultado = undefined;

    switch (dado) {
        case 4:
            resultado = randomIntFromInterval(1, 4);
            break;
```

```
            case 6:
                resultado = randomIntFromInterval(1, 6);
                break;
            case 8:
                resultado = randomIntFromInterval(1, 8);
                break;
            case 10:
                resultado = randomIntFromInterval(1, 10);
                break;
            case 12:
                resultado = randomIntFromInterval(1, 12);
                break;
            case 20:
                resultado = randomIntFromInterval(1, 20);
                break;
            default:
                resultado = null;
                console.log('Esse não é um valor válido');
                break;
        }
        if (resultado) console.log('Rolagem: ' + resultado);
    }
    while (true);
```

Vamos analisar o código acima.

- Solicitamos um dado ao usuário, e inicializamos resultado como `undefined`, pois existe a chance desse programa de não produzir resultado;

- Depois usamos um `switch` que verifica qual é o dado e chama a função de geração de número aleatório com o parâmetro correto;
- Aproveitamos o `default` do `switch` para tratar valores inválidos;
- No final, se resultado não continuar `undefined` ou ter caído no default (que o transforma em `null`) nós mostramos seu valor;
- E por fim, usamos `while(true)` para manter o programa rodando para sempre.

Flags

Algoritmos têm em sua definição a obrigatoriedade de serem finitos, isso quer dizer que programas não podem rodar para sempre. Sendo assim, temos que dar um jeito de parar nosso programa anterior. Não podemos usar `while(true)` e mantê-lo rodando para sempre.

Para isso, usaremos um conceito chamado "flag", que determina se o programa deve continuar rodando ou não. Esse conceito tem esse nome baseado em uma bandeira (talvez venha de corridas ou algo assim), a bandeira tem dois estados, hasteada ou não. Você decide se quer usar a "flag" para parar seu programa ou para mantê-lo rodando, minha preferência é usá-la para parar. Me parece que a legibilidade é melhor assim. Dissemos que a bandeira tem dois estados, hasteada e não hasteada, sendo assim, ela será representada por uma variável booleana.

Vamos reescrever o exemplo usando uma flag. Quando o usuário não digitar nada vamos hastear nossa bandeira dizendo que o laço precisa parar.

Exemplo: Rolador de dados V.2

```
//Função para gerar um número inteiro randômico de acordo com um intervalo
function randomIntFromInterval(min, max) {
    return Math.floor(Math.random() * (max - min + 1) + min);
}

//Programa
let finished = false;

do {
```

```
const dice = parseInt(prompt('digite 4, 6, 8, 10, 12 ou 20 para rolar o dado:'));

if (dice) {
    result = undefined;
    switch (dice) {
        case 4:
            result = randomIntFromInterval(1, 4);
            break;
        case 6:
            result = randomIntFromInterval(1, 6);
            break;
        case 8:
            result = randomIntFromInterval(1, 8);
            break;
        case 10:
            result = randomIntFromInterval(1, 10);
            break;
        case 12:
            result = randomIntFromInterval(1, 12);
            break;
        case 20:
            result = randomIntFromInterval(1, 20);
            break;
        default:
            result = null;
            console.log('Esse não é um valor válido');
            break;
    }
```

```
            if (result) console.log('Rolagem: ' + result);
        }
        else finished = true;
    }
    while (!finished);
```

Declaramos a flag `finished` como `false` para indicar que o programa deve continuar, no entanto, laços trabalham com o valor `true` para continuar e não `false`, por isso negamos o valor de `finished`: `while(!finished)`.

> Se você ler essa linha em inglês faz todo sentido `while not finished`, isto é, "enquanto não terminado", continue rodando.

Alteramos `finished` para `true` se o usuário colocar um valor avaliado como "falsy", ou seja, `undefined`, `null`, `string vazia`, `0`, ou `NaN` (not a number). Dessa forma, o usuário pode parar o programa não digitando nada no campo e enviando.

Refatoração

Chamamos de refatoração o ato de melhorar um código que já funciona corretamente. Todo código tende a entropia, ou seja, a cada alteração para adicionar funcionalidades, por exemplo, o código fica um pouco pior. Usamos a refatoração para combater essa realidade. Se sempre que passarmos por um código melhorarmos um pouco ele, esse código não tenderá a entropia e se manterá decente por mais tempo.

Existem muitas formas de refatorar, minha favorita é a diminuição, encontrar padrões que indiquem que podemos fazer um código menor e menos propenso aos erros. No código acima, um padrão desses apareceu, o switch testa por exemplo se o dice é 4 e depois usa o 4 como parâmetro. Isso quer dizer que o teste é desnecessário.

Vamos refatorar e remover o switch:

> Exemplo: Rolador de dados V.3

```
//Função para gerar um número inteiro randômico de acordo com um intervalo
function randomIntFromInterval(min, max) {
    return Math.floor(Math.random() * (max - min + 1) + min);
```

```
}

//Programa
let finish = false;

do {
    const dice = parseInt(prompt('digite 4, 6, 8, 10, 12 ou 20 para rolar o dado:'));

    if (dice) {
        if (dice === 4 || dice === 6 || dice === 8 || dice === 10 || dice === 12 || > dice === 20) {
            console.log('Rolagem: ' + randomIntFromInterval(1, dice));
        }
        else console.log('Esse não é um valor válido');
    }
    else finish = true;
}
while (!finish);
```

Veja, fizemos:

- O teste de todos os valores válidos de uma vez no `if`;
- A rolagem direto no `console.log()` o que nos permitiu economizar uma variável.

Tem uma coisa ainda me incomodando: a condição enorme no `if`. Vamos transformá-la em uma função. Veremos funções a fundo mais para a frente então não precisa se preocupar em aprender agora.

Outra coisa que vamos transformar em função será a rolagem, como ela sempre começa em 1 podemos fazer uma função de só um parâmetro com nome de melhor semântica para deixar o código mais claro.

Exemplo: Rolador de dados V.4

```
//Função para gerar um número inteiro randômico de acordo com um intervalo
function randomIntFromInterval(min, max) {
    return Math.floor(Math.random() * (max - min + 1) + min);
}

//Função rolar dado
function roll(dice) {
    return randomIntFromInterval(1, dice);
}

//Função de validação dos dados
function isValidDice(dice) {
    return dice === 4  || dice === 6  || dice === 8 ||
           dice === 10 || dice === 12 || dice === 20;
}

//Programa
let finish = false;

do {
    const dice = parseInt(prompt('digite 4, 6, 8, 10, 12 ou 20 para rolar o dado:'));
    if (dice) {
        if (isValidDice(dice)) console.log('Rolagem: ' + roll(dice));
        else console.log('Esse não é um valor válido');
    }
    else finish = true;
```

```
    }
    while (!finish);
```

Observe que criamos uma função chamada `roll` que recebe um `dice` como parâmetro, assim usamos nomenclaturas que fazem parte do domínio de nosso programa, muito melhor do que `randomIntFromInterval`. Fizemos uma função de validação que retorna a mesma coisa que antes era a condição do `if` chamada `isValidDice`.

Referências e Materiais Complementares

- While

Próximo Tópico