




Instruções

Para iniciar nossa aula sobre “Instruções” em JavaScript, é essencial compreendermos o conceito fundamental que sustenta o funcionamento de qualquer programa: as instruções. Elas são responsáveis por orientar o computador a executar ações específicas, permitindo que o código responda de maneira dinâmica às interações do usuário. Ao longo desta aula, abordaremos diferentes tipos de instruções, desde as básicas, como condicionais e laços de repetição, até as mais avançadas, como o uso de rótulos. Através de exemplos práticos, veremos como essas instruções são implementadas em JavaScript, tornando nossos programas mais eficientes e adaptáveis.

O que são instruções?

Nesta primeira parte, vamos nos concentrar em entender o conceito de instruções no contexto de desenvolvimento de software, especificamente utilizando JavaScript. As instruções são fundamentais para o funcionamento de qualquer programa, pois são elas que orientam o computador a realizar determinadas ações ou operações. Em JavaScript, uma instrução pode ser vista como uma declaração que o navegador interpreta e executa, sendo responsável por definir como o código irá se comportar em diferentes cenários.

As instruções são essenciais para permitir o controle de fluxo em nossos programas. Por exemplo, ao interagir com uma aplicação web, como ao clicar em um botão ou inserir dados em um formulário, o programa precisa reagir a essas ações. Essa reação é possível por causa das instruções, que determinam como o código deve responder a essas interações. Imagine, por exemplo, que você deseja exibir uma mensagem quando um botão for clicado. Para isso, você precisará de uma instrução que detecte o clique e execute o código responsável por exibir a mensagem.


Além disso, as instruções em JavaScript podem envolver diversos elementos, como palavras-chave, operadores, expressões, e funções. Cada um desses elementos desempenha um papel específico e permite a construção de lógica complexa dentro do código. Por exemplo, considere o uso de instruções condicionais, como `if` e `else`, que possibilitam a tomada de decisões dentro do programa. Se uma determinada condição for atendida, o código dentro do bloco `if` será executado; caso contrário, o bloco `else` será executado, permitindo que diferentes caminhos sejam seguidos conforme a necessidade. 

Outro exemplo importante são as instruções de repetição, como `for` e `while`, que permitem a execução repetida de um bloco de código enquanto uma condição for verdadeira. Isso é útil, por exemplo, quando você precisa processar uma lista de itens, aplicando a mesma operação a cada item. As instruções de repetição tornam o código mais eficiente e evitam a necessidade de escrever o mesmo bloco de código várias vezes.

Instruções condicionais

Na segunda parte da nossa aula, vamos nos aprofundar no tema das instruções condicionais, que são fundamentais para controlar o fluxo de execução dos programas. Em essência, uma instrução condicional é uma estrutura que permite ao código tomar decisões com base em determinadas condições. Isso é crucial para criar programas dinâmicos e responsivos, que podem executar diferentes blocos de código conforme as variáveis mudam ou conforme as interações dos usuários ocorrem.

O comando `if` é a base das instruções condicionais. Ele avalia uma condição e, se essa condição for verdadeira, executa um bloco específico de código. Por exemplo, em JavaScript, podemos verificar se o valor de uma variável “fruta” é igual a “banana” usando a sintaxe `if (fruta === 'banana')`. Se a condição for atendida, o código dentro do bloco `if` será executado, como exibir uma mensagem no console: `console.log('Minha fruta é uma banana.')`. Caso a condição não seja satisfeita, podemos utilizar a instrução `else` para definir o que deve ser feito em alternativa.

Assim, podemos adicionar `'else { console.log('Minha fruta não é uma banana.')} ?`, garantindo que o código sempre tenha uma saída, independentemente  condição inicial.

Além do if-else, também temos o switch, que é particularmente útil quando há múltiplas condições possíveis para uma variável. Por exemplo, se tivermos uma variável “fruta” que pode conter “banana,” “uva,” ou “maçã,” podemos usar um switch para comparar o valor da variável e executar o código correspondente a cada caso. A sintaxe básica seria `switch(fruta) { case 'banana': console.log('Minha fruta é uma banana.');` `break;` `case 'uva': console.log('Minha fruta é uma uva.');` `break;` `case 'maçã': console.log('Minha fruta é uma maçã.');` `break;` `default:` `console.log('Minha fruta não está nas opções válidas.');` `}`. O default é executado se nenhum dos casos for atendido, funcionando como uma rede de segurança para garantir que o código trate todas as possíveis entradas.

Um ponto importante ao trabalhar com essas instruções é garantir que as comparações sejam feitas de maneira consistente, especialmente em relação à sensibilidade a maiúsculas e minúsculas. Utilizar métodos como `toLowerCase()` pode ajudar a padronizar os valores das variáveis antes de compará-los, evitando erros que possam surgir devido a diferenças na capitalização. Dessa forma, garantimos que nosso código funcione de forma robusta e sem surpresas.

Instruções de laço e repetição

Agora vamos nos aprofundar nas instruções de laço e repetição, que são essenciais para a execução repetida de blocos de código em situações onde a repetição é necessária. Essas estruturas permitem que o código seja executado várias vezes, controlando o número de iterações com base em condições definidas pelo programador.

A estrutura mais comum de laço é o for, que segue uma sintaxe específica em JavaScript. Por exemplo, `for (let i = 0; i <= 10; i++) { console.log("Valor de i é " + i); }` inicializa a variável i com 0, verifica se i é menor ou igual a 10, e, caso positivo, executa o código dentro do laço, incrementando i em 1 a cada iteração. O for é

ideal quando sabemos antecipadamente o número de vezes que queremos que o laço seja executado.



Outra estrutura importante é o `while`, que executa um bloco de código repetidamente enquanto uma condição for verdadeira. Por exemplo, `let i = 0; while (i <= 5) { console.log("Valor de i dentro do while é " + i); i++; }` continua a execução enquanto `i` for menor ou igual a 5. É fundamental garantir que a condição eventualmente se torne falsa para evitar laços infinitos que podem travar o sistema.


Por fim, temos o `do while`, que garante que o bloco de código seja executado pelo menos uma vez, independentemente da condição inicial. Sua sintaxe é `do { i++; console.log("Valor de i no do while é " + i); } while (i <= 3);`. Isso significa que, mesmo que a condição `i <= 3` seja falsa logo de início, o código dentro do laço será executado uma vez antes da verificação.

Essas estruturas são poderosas ferramentas para controlar o fluxo de execução em seus programas. Com o `for`, podemos facilmente iterar um número fixo de vezes, enquanto o `while` e o `do while` são mais flexíveis para situações onde a condição de término depende de eventos dinâmicos ou entradas do usuário. Na prática, o uso correto dessas instruções permite a criação de programas mais eficientes e adaptáveis a diferentes cenários de execução.

Trabalhando com rótulos

Na quarta e última parte da nossa aula sobre "Instruções", vamos explorar o conceito de rótulos em JavaScript e como eles podem ser utilizados em conjunto com loops e o comando `continue`. Embora o uso de rótulos não seja comum em aplicações modernas, é importante entender seu funcionamento e como eles podem ser aplicados para controlar o fluxo de execução do código.

Um rótulo em JavaScript é definido por um nome seguido de dois pontos. Ele permite que você identifique um bloco específico de código, normalmente usado em conjunto com loops. Por exemplo, quando temos loops aninhados (um loop

dentro de outro), os rótulos podem ser utilizados para pular de um loop interno para uma iteração específica do loop externo, dependendo de uma condição. 

Vamos ver um exemplo prático. Considere um loop for em que a variável *i* é inicializada em 0, com a condição de que *i* seja menor que 3 e seja incrementada a cada iteração. Dentro desse loop, há um segundo loop for com uma variável *j* que também é incrementada enquanto *j* < 3. Aqui, podemos usar o comando continue seguido do nome do rótulo para pular para a próxima iteração do loop externo quando *j* atinge um valor específico, como 1. Isso significa que, ao invés de continuar executando o código dentro do loop interno, o controle de execução salta para a próxima iteração do loop externo.

Quando rodamos esse código, o navegador mostrará que, ao atingir a condição definida no continue, o loop interno é interrompido, e o controle retorna ao loop externo, iniciando uma nova iteração. Por exemplo, se alterarmos a condição para que o continue seja acionado quando *j* for igual a 2, o loop interno executará uma vez a mais antes de pular para a próxima iteração.

Embora existam outras formas mais comuns de controlar o fluxo de loops em JavaScript, como a combinação de break e continue sem rótulos, é útil conhecer essa ferramenta para entender como o controle de execução pode ser refinado em cenários específicos.

Com isso, concluímos nossa aula sobre instruções em JavaScript, cobrindo desde as instruções básicas até as mais avançadas, como as que envolvem rótulos. Praticar o uso dessas instruções é fundamental para dominar a lógica de programação e criar soluções eficientes e funcionais.

Conteúdo Bônus

Para auxiliar nos estudos sobre operadores, recomendo assistir ao vídeo "LÓGICA DE PROGRAMAÇÃO - LAÇO DE REPETIÇÃO "PARA" (LAÇO FOR)", que está



Referência Bibliográfica

ASCENCIO, A. F. G.; CAMPOS, E. A. V. de. Fundamentos da programação de computadores. 3.ed. Pearson: 2012.

BRAGA, P. H. Teste de software. Pearson: 2016.

GALLOTTI, G. M. A. (Org.). Arquitetura de software. Pearson: 2017.

GALLOTTI, G. M. A. Qualidade de software. Pearson: 2015.

MEDEIROS, E. Desenvolvendo software com UML 2.0 definitivo. Pearson: 2004.

PFLEEGER, S. L. Engenharia de software: teoria e prática. 2.ed. Pearson: 2003.

SOMMERVILLE, I. Engenharia de software. 10.ed. Pearson: 2019.

Ir para exercício