~~Lógica de Programação~~ Aprendendo a Programar

Versão 2.0 de 12/03/2019

AVISO DE RESPONSABILIDADE

As informações contidas neste material de treinamento são distribuídas “NO ESTADO EM QUE SE ENCONTRAM”, sem qualquer garantia, expressa ou implícita.

A cópia e uso deste material é livre inclusive para fins comerciais, desde que o material não sofra modificações e seja mantida a referência ao autor e a empresa Alfamídia.

03/2019

Rodrigo Losina
Alfamídia
www.alfamidia.com.br

1. Vamos Começar Programando

1.1 Vamos fazer um primeiro programa

Talvez você já tenha lido uma (ou muitas) apostilas de programação. Talvez mesmo já tenha feito cursos da – assim chamada – lógica de programação.

Imagino que você acredite que ainda não sabe programar, ou pior, que programar não é para você. (De outro modo, por que está lendo esta apostila?)

Vamos resolver isso fazendo um primeiro programa. Não deve levar mais de alguns minutos...

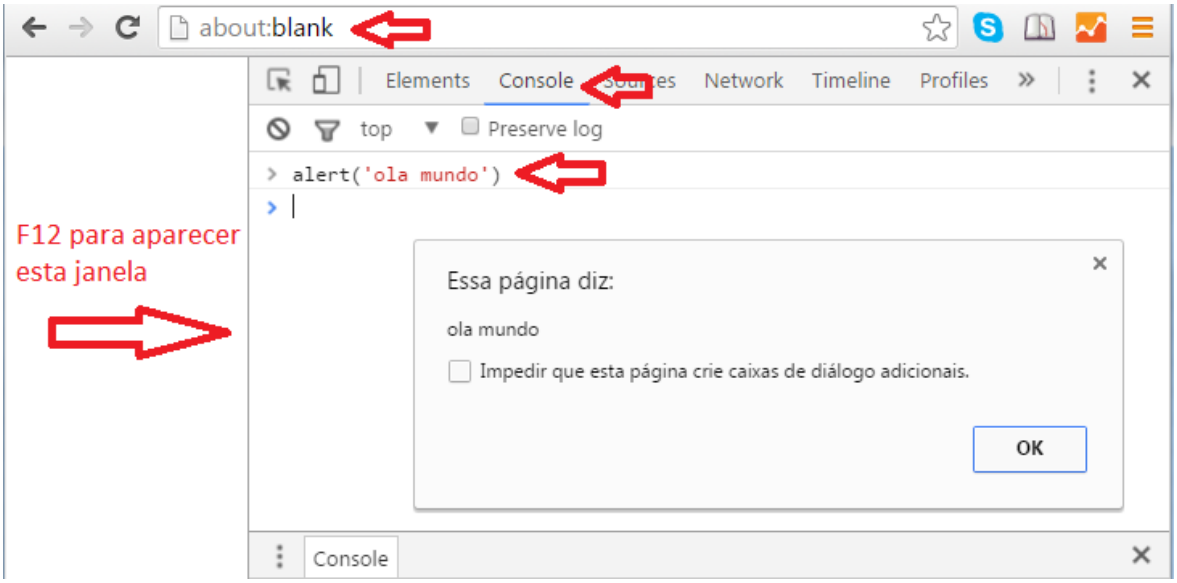
----------CRIANDO SEU PRIMEIRO PROGRAMA----------

1. Abra o Chrome (se você não sabe o que é o Chrome, eu admito, estamos com problemas).
2. Digite “about:blank” na entrada de endereços, como se estivesse digitando o endereço de um site. (cuidado para não colocar espaços depois do ‘:’).
3. Aperte a tecla F12
4. Vai abrir uma janela ocupando uma parte da área em que um site normalmente é exibido. Nela tem uma aba escrita “console”. Selecione esta aba.
5. Na área abaixo, escreva o seguinte: alert(“ola mundo”).
6. Tecle Enter.

Há duas possibilidades: funcionou; não funcionou.

Se apareceu uma janela com a mensagem “ola mundo”, então funcionou. Parabéns, você acabou de criar seu primeiro programa.

Caso contrário, alguma coisa deu errado. Você vai ter que descobrir o que aconteceu. Parabéns, você está lidando com seu primeiro bug. Nós vamos tratar de bugs mais adiante, então neste momento o que posso dizer é para olhar cada um dos passos e ver onde você pode ter errado. Para ajudar, observe a imagem a seguir:



1.2 Vamos fazer um segundo programa

Vamos agora parar e estudar um pouco de teoria, para entendermos exatamente o que é um programa, o que é lógica de programação, criarmos fluxogramas, pseudocódigos, etc?

Não, não vamos.

Se alguém tivesse ensinado a você a teoria do movimento antes de você dar o seu primeiro passo, você talvez ainda estivesse engatinhando, e dizendo que caminhar é muito difícil.

O que nós vamos fazer, é criar seu segundo programa.

----------CRIANDO SEU SEGUNDO PROGRAMA----------

Faça os mesmos passos 1 a 4 de antes (a partir de agora não vou mais repetir isso, ok?)

Digite: alert(“ola”);

Digite SHIFT ENTER (isso significa apertar a tecla SHIFT e AO MESMO TEMPO apertar a tecla ENTER).

Digite: alert(“mundo”);

Tecle ENTER

Se não deu certo agora, provavelmente você apertou ENTER sem o SHIFT, e o Chrome achou que seu programa tinha terminado no primeiro alert.

Se tudo deu certo, apareceu uma primeira janela de alerta, e depois uma segunda.

Vamos agora ver um pouquinho de teoria, mas muito pouco, muito pouco mesmo, e já vamos continuar programando.

1.3 Programas Seguem um Fluxo, Passo a Passo

O programa anterior nos apresentou um conceito que não é tão óbvio quanto pode parecer à primeira vista. Um programa executa um comando de cada vez, passo a passo. Em uma linha tínhamos um comando "alert('ola');", e na linha de baixo tínhamos o comando "alert('mundo')", e o programa executou a primeira linha, e depois executou a segunda.

Vamos chamar isso de um fluxo de execução, um 'caminho' que o programa segue, de executar uma linha após a outra. Em termos do que nos interessa nesta apostila, que são linguagens de programação imperativas, todo programa segue sempre um fluxo, executando as linhas de código em sequência. No futuro veremos como controlar este fluxo de execução.

Obs: nem tudo no mundo são linguagens de programação imperativas. Existem outros tipos de linguagens de programação e outras formas de criar programas, apenas não nos preocuparemos com elas nesta apostila.

1.4 Precisamos falar sobre variáveis

Vamos fazer nosso terceiro programa. Já somos programadores experientes, certo? Então não vou passar todo o passo a passo, apenas colocar o programa que quero que você escreva. Apenas nunca esqueça de dar SHIFT ENTER no console entre cada linha, para ir para a próxima:

```
-------------------------SEU TERCEIRO PROGRAMA------------------

var x;

x = 1;

alert(x);

x = 2;

alert(x);
```

```
x = x + 1;
alert(x);
```

Observe que, desta vez, o texto entre parenteses não está entra aspas. O resultado deverá ser três janelas, mostrando os números 1, 2 e 3.

Muito mais coisa está acontecendo aqui do que parece à primeira vista, e precisamos mesmo falar sobre isso, e especialmente sobre nossa variável chamada 'x'.

Declarando a variável: em muitas linguagens, é obrigatório declarar uma variável. Este não é o caso do javascript, e o programa vai funcionar exatamente igual se removermos a primeira linha. Na unidade sobre variáveis iremos mostrar quando e por que utilizar a declaração 'var'. Por ora, basta saber que é conveniente declarar todas as variáveis antes de utilizá-las.

O fluxo de execução: temos aqui nosso fluxo de execução novamente em ação. Cada comando foi executado em sequência.

A variável: temos uma variável chamada 'x', e temos o sinal de igualdade '='. Parece simples, mas é tudo mais complicado que parece, pois nem nossa variável é igual as variáveis que aprendemos nas equações do colégio, nem o sinal de igualdade é o mesmo que conhecíamos.

Se fosse diferente, se fosse igual a tudo que já aprendemos, o que dizer da expressão "x = x + 1". A primeira vez que a vi, lembro bem de pensar "como pode? 'x' não é igual a 'x + 1', 'x' é menor que 'x + 1'.

Ocorre que o sinal de '=' não é uma igualdade, é uma atribuição. Veremos mais adiante a representação de igualdade (que utiliza em algumas linguagens de programação o sinal '==').

"x = 2" não está dizendo que x é igual a dois, está dizendo que, a partir de agora, no fluxo de execução, x passa a receber o valor 2. "x = x + 1" significa que, a partir de agora no fluxo de execução, 'x' passa a receber o valor que 'x' tinha, acrescido de 1.

O que é uma variável: Então, agora começamos a entender o que é uma variável. Uma variável, para nós, é apenas uma forma de guardar uma informação. Imagine que você tem uma folha de papel, um lápis e uma borracha. Nesta folha você pode ter alguma informação, que você pode apagar a qualquer momento, para escrever outra no lugar.

Foi o que fizemos. Nessa folha de papel, que chamamos de 'x', primeiro escrevemos o número '1', depois utilizamos no nosso alert ('alert(x)'), depois apagamos e escrevemos o número '2' por cima. Fizemos nosso alert novamente.

Depois, vimos qual era o número no papel (era ‘2’), somamos com 1 (ficou ‘3’), e escrevemos novamente no papel.

Ficou fácil até aqui? Não se preocupe, já vamos começar a complicar.

1.5 Vamos fazer escolhas?

Vamos ver o que sabemos até agora, e o que mais nos faz falta, para poder criar um programa....

Sabemos trabalhar com variáveis para colocar e alterar números, e sabemos exibir uma mensagem na tela. Ainda é pouco, precisamos dominar pelo menos mais dois ou três conteúdos para poder criar programas que façam algo um pouco mais útil.

Vamos começar com escolhas. Nós queremos que o programa tome decisões. Por exemplo, podemos querer que ele exiba “ola mundo” em determinados momentos, mas exiba “tchau mundo” em outros.

Para isso, vamos olhar dois programas abaixo. Para adivinharem o que eles fazem, já dou uma dica: pensem em “if” como a palavra “se”, e “else” como a palavra “senão”.

```
----------AGORA DOIS PROGRAMAS DE UMA SÓ VEZ----------
var x;
x = 5;
if (x > 10) {
  alert(“ola mundo”);
} else {
  alert(“tchau mundo”);
}
----------AQUI VAI O SEGUNDO----------
var x;
x = 15;
if (x > 10) {
  alert(“ola mundo”);
} else {
  alert(“tchau mundo”);
}
```

Você adivinhou o que apareceria em cada programa? Se tudo deu certo, o primeiro exibiu “tchau mundo”, e o segundo exibiu “ola mundo”.

Por quê? Obviamente, porque no primeiro ‘x’ não era maior que 10, enquanto no segundo ele era.

E assim chegamos em nosso primeiro “**comando de controle de fluxo**”. Até marquei em negrito pela importância!

Por que se chamam comandos de fluxo? Por que eles alteram o fluxo de execução. Até agora havíamos visto os programas executarem uma linha de cada vez, mas agora vimos uma ‘bifurcação’ na nossa estrada, um desvio no nosso fluxo. Conforme a condição, entravamos em uma linha, ou na outra.

Nós não vamos perder tempo aqui em algumas questões que penso serem avançadas (e chatas) como ‘lógica booleana’ e outras palavras exóticas. Iremos vê-las mais adiante.

Mas vamos falar nas seções seguintes de duas coisas realmente importantes sobre o uso do comando “if”, que vocês precisam saber desde o primeiro momento, para poder utilizá-lo com segurança.

1.6 Blocos de Comando

Mexendo com o fluxo de execução, precisamos ser capazes de agrupar nossos comandos em blocos. Por quê? Vamos ver os dois comandos abaixo para entender isso.

```
-------------------------ESTE PROGRAMA ESTÁ CERTO-----------------------------
var x;
x = 5;
if (x > 10)
  alert(“ola mundo”);
alert(“e aqui terminamos”);
------------------ESTE PROGRAMA TAMBÉM ESTÁ CERTO----------------------
var x;
x = 5;
if (x > 10) {
  alert(“ola mundo”);
```

```
  alert(“e aqui terminamos”);
}
```

Os dois programas estão corretos, mas eles fazem coisas diferentes. As chaves definem um bloco, e precisamos definir blocos quando queremos fazer o fluxo de execução executar em um conjunto de comandos.

No primeiro caso, estamos dizendo para o primeiro ‘alert’ só ser executado em determinada condição. No segundo caso, estamos dizendo para todo o bloco entre as chaves, só ser executado se entrarmos na condição.

E chegamos em um bom momento para falarmos pela primeira vez em identação e boas práticas:

```
----------ESTE PROGRAMA ESTÁ CERTO MAS PARECE ERRADO----------
var x;
x = 5;
if (x > 10)
  alert(“ola mundo”);
  alert(“e aqui terminamos”);
```

Este programa está certo, na verdade, para o computador ele é exatamente igual ao primeiro que mostramos no início da seção. O problema é que para um ser humano, ele parece com o segundo.

Na maioria das linguagens, a identação, os espaços em brancos, a posição dos comandos na linha, não fazem diferença. Mas nós, humanos, automaticamente pensamos que coisas alinhadas são semelhantes.

Em resumo, sempre que entrar em um desvio de fluxo ou no início de um bloco, passe a escrever os comandos um pouco mais a direita. Quando sair do comando, volte. Assim, ao final do programa você estará exatamente na mesma coluna que quando iniciou. Isso vai ser fundamental quando você estiver fazendo programas maiores e mais complexos.

1.7 Nosso primeiro pega-ratão

Agora vamos ver a segunda coisa realmente importante do ‘if’. Se você realmente nunca leu nada sobre programação, você provavelmente vai errar em adivinhar o que faz este programa:

```
----------ESTE PROGRAMA VAI ENGANAR VOCÊ----------
var x;
x = 5;
if (x = 10)  {
  alert(“ola mundo”);
}
-----------------------------------------------------------------------------------------------
```

Ei, ei ei, é isso mesmo? O programa exibiu o alerta? Sim. Vamos acrescentar um novo alerta, após o ‘if’, para ajudar a desvendar este mistério:

```
----------SERÁ QUE FICOU ÓBVIO AGORA----------
var x;
x = 5;
if (x = 10)  {
  alert(“ola mundo”);
}
alert(x);
-----------------------------------------------------------------------------------------------
```

Não só o alerta foi exibido, como ‘x’ mudou para o valor ‘10’. Ficou claro agora? Nós repetimos o mesmo comando de atribuição dentro do ‘if’. Primeiro fizemos ‘x=5’, depois fizemos ‘x=10’.

Para fazermos comparações, o comando de igualdade (na maioria das linguagens), não é igual ao da matemática, não é ‘=’, mas sim ‘==’. Vamos então corrigir nosso programa:

```
----------AGORA SIM----------
var x;
x = 5;
```

```
if (x == 10)  {
  alert(“ola mundo”);
}
alert(x);
```

Pode parecer simplório falar de ‘==’ e de blocos, mas são duas das questões que mais atrapalham programadores iniciantes, e às vezes até profissionais, na hora de desenvolver um programa. Revisem estes dois itens para ter certeza que nunca vão esquecer de abrir e fechar as chaves, e nunca vão confundir comparação ‘==’ com atribuição ‘=’.

Obs: alguns programadores preferem inverter as comparações, escrevendo ‘if (5 == x)’, para garantir que não vão ter problemas mesmo que um dia esqueçam o ‘==’, pois neste caso o programa vai exibir um erro.

1.8 Vamos ouvir o usuário?

Calma, sem pânico. Este não é um curso avançado de programação, muito menos de psicologia. Ninguém vai obrigá-lo a falar com usuários ou outros seres humanos. Por enquanto queremos apenas que o usuário passe algum tipo de informação para nosso programa.

Vamos utilizar um novo comando, o comando prompt.

Obs: agora parece uma boa hora para mencionar que estamos há um tempinho programando em javascript, uma linguagem bastante poderosa e complexa. ‘alert’ e ‘prompt’ são específicos desta linguagem, mas não vamos nos preocupar muito com isso, pois praticamente qualquer linguagem vai ter algo parecido, ou seja, alguma forma de exibir informações e alguma forma de receber dados do usuário. As demais instruções que vimos são praticamente iguais em todas as linguagens imperativas.

```
----------VOCÊ DECIDE SE O ALERTA SERÁ EXIBIDO----------
var x;
x = prompt(“digite um número”);
if (x > 10)  {
  alert(“ola mundo”);
}
```

```
alert(x);
```

Quero crer que você já sabe o suficiente de programação para entender o que o ‘prompt’ fez, apenas executando o programa.

Senão, lá vai: o prompt recebeu a entrada do usuário e colocou na variável ‘x’. Se foi um número maior que 10, portanto, o programa exibiu ‘ola mundo’.

1.9 Agora estamos prontos para programar

Verdade?

Verdade.

Com o que vimos, já temos o suficiente para começar a praticar a programação. Na próxima unidade, começaremos a criar programas usando apenas o conteúdo visto nesta unidade.

Vamos então a algumas explicações de por que estamos ensinando programação assim. Veja, o grande problema de muitos cursos de programação é que tentam ensinar a teoria antes da prática. Imagine passar um semestre de educação física ouvindo sobre regras e técnicas de futebol, ou aulas inteiras ouvindo sobre como se anda de bicicleta.

Isso quer dizer que teorias não são importantes, ou que nunca vamos precisar saber o porquê das coisas? Não, longe disso. Eu sou 100% a favor de crianças estudarem gramática, por exemplo, só que faz muito mais sentido elas estudarem a gramática de uma lingua que elas já adquiram alguma prática de uso.

É o que faremos neste curso. Minha preocupação é que vocês pratiquem a programação, e desenvolvam a habilidade de criar programas que resolvam questões reais. Para tanto, vamos praticar e praticar, com exemplos cada vez mais complexos, para então, quando estiverem prontos e dominando as técnicas, vamos entender melhor os conceitos teóricos que embasam as linguagens de programação.

2. Enfrentando Nossos Primeiros Desafios

2.1 Programando com um Editor de Texto

Até agora estivemos utilizando o console do Chrome. Vamos agora utilizar também um editor de textos, para facilitar nosso trabalho. Você pode utilizar um editor específico para programação, como o "Sublime". Aqui, vamos falar apenas no uso do "bloco de notas", ou "notepad". Você provavelmente tem ele, ou algum equivalente em seu computador.

Dependendo do sistema operacional que você estiver utilizando, ele pode estar em diferentes locais. Em alguns sistemas, você pode digitar "bloco de notas" em alguma opção de pesquisa do sistema operacional, que ele vai aparecer.

Digite no editor o código abaixo.

```
----------MINHA PRIMEIRA PÁGINA COM JAVASCRIPT----------
<html>
<head>
<script>
alert("ola mundo");
</script>
</head>
</html>
---------------------------------------------------------------------------------------------
```

Salve o arquivo, e abra no Chrome. Deve aparecer o alerta.

A partir de agora, simplesmente copie os códigos mostrados entre as tags "<script>" e "</script>", e salve o arquivo com um novo nome. Com isso será mais fácil alterar e testar nossos códigos.

2.2 Método Exemplo-Desafio-Resposta

A partir desta unidade, vamos utilizar muito esta sequência. Vamos mostar um código de exemplo, e discutir questões que considerarmos novas nele. Nesta etapa, copiem o código para um editor, abram no Chrome, e estudem ele até terem certeza que entenderam todo o programa.

Na sequência daremos um desafio. Tentem criar um código que execute aquele desafio. Sempre será possível resolver o desafio utilizando apenas conteúdos previamente vistos.

Por fim, temos a resposta do desafio. Comparem com sua própria solução, vejam qual é a melhor. Se não conseguirem cumprir o desafio, analisem bem a resposta, tentem descobrir por que não conseguiram fazer sozinhos.

2.3 Vamos Fazer Operações Matemáticas

Observem o código abaixo. Tentem entender o que ele faz. Depois copiem para dentro de um editor de texto, salvem, e executem.

```
----------UMA CALCULADORA SIMPLÓRIA----------

var x;

var n1;

var n2;

x = prompt(“entre com uma letra para escolher a operação matemática: (s)ubtração, (m)ultiplicação ou (d)ivisão.”);

n1 = prompt(“entre com o primeiro número”);

n2 = prompt(“entre com o segundo número”);

alert(“O resultado é...”);

if (x == “s”) alert(n1 - n2);

if (x == “m”) alert(n1 * n2);

if (x == “d”) alert(n1 / n2);

-------------------------------------------------------------------------------------
```

Há muitos problemas com este código, a começar pelo fato que esta calculadora não tem a operação de soma, e vamos tratá-los, um por um, mais adiante. Mas tudo a seu tempo. O importante é que este código faz o que se propõe, ou seja, fazer uma operação matemática entre dois números e mostrar para o usuário.

Vamos agora estudar quais as novidades deste código, em relação ao que já vimos anteriormente.

A variável ‘x’ recebeu uma letra

Lembrem-se, uma variável, para nós, é uma forma de guardar informações. Não confundam com as variáveis que vocês estudaram em matemática. Até agora, havíamos visto apenas as variáveis armazenarem números, mas elas podem guardar muitas outras coisas.

No caso, a variável x recebe uma letra, que deverá ser ‘s’, ‘m’ ou ‘d’. Para comparar o valor de x com estas letras, temos que lembrar de colocá-las entre aspas, diferente das comparações com números.

Podemos fazer outras operações matemáticas além da soma

Vimos no exemplo, três outras operações matemáticas. O sinal de ‘-‘ obviamente é uma subtração. Estamos vendo que a multiplicação é feita utilizando o sinal ‘*’, enquanto que o sinal ‘/’ representa a divisão.

2.4 Nosso Primeiro Desafio

Neste primeiro desafio, faça um programa que receba como entrada do usuário um número inteiro entre 2 e 5. Depois, receba como entradas, 2, 3, 4 ou 5 números, conforme o primeiro número fornecido.

A seguir, exiba cada número, na ordem inversa da fornecida.

Por exemplo, se as entradas do usuário forem 3 (ou seja, é para recebermos 3 números), 1, 3 e 5, teremos como resposta três janelas que exibirão, em ordem, 5, 3 e 1.

Tente fazer o programa. Se você conhece outros comandos além dos mostrados até aqui, pode utilizá-los, mas nossa solução usa apenas o que já mostramos.

2.5 Resposta do Desafio

Não, não, não. Primeiro faça o desafio você. Abra o editor, escreva, teste, tenha certeza que entendeu bem e fez um programa que funciona corretamente. Só depois olhe a resposta do desafio.

2.6 Resposta Mesmo do Desafio

Ok, agora sim. Veja, a seguir, uma resposta do desafio. Obviamente, há como fazer soluções mais elegantes, em especial utilizando comandos que ainda não vimos, e tomando cuidados que ainda não ensinamos a tomar. Mas esta é uma solução aceitável dentro do que já foi ensinado:

----------NOSSO PRIMEIRO DESAFIO CONCLUÍDO----------

```
var x;
```

```
var n1, n2, n3, n4, n5;

x = prompt(“entre com um número de 2 a 5”);

n1 = prompt(“entre com o primeiro número”);

n2 = prompt(“entre com o segundo número”);

if (x > 2) n3 = prompt(“entre com o terceiro número”);

if (x > 3) n4 = prompt(“entre com o quarto número”);

if (x > 4) n5 = prompt(“entre com o quinto número”);

if (x > 4) alert(n5);

if (x > 3) alert(n4);

if (x > 2) alert(n3);

alert(n2);

alert(n1);
```

Você conseguiu realizar o desafio sozinho? Sim? Parabéns, você acabou de criar um programa sozinho. Você já pode se considerar um programador.

Não rolou? Bem, temos que trabalhar nisso, pois as coisas, depois, vão ficar mais difíceis. Não avance ainda para a próxima unidade. Você vai ter que trabalhar um pouco mais neste exercício.

Minha sugestão, estude o código acima, até ter certeza que entendeu ele bem, depois comece a modificá-lo. Aceite 6, 7 ou 8 números. Coloque uma primeira entrada pedindo para o usuário escolher se os números vão ser mostrados na ordem direta ou, como aqui, na ordem invertida. Ou seja, pratique até dominar bem este código, antes de avançar para nossa próxima unidade.

Obs:programadores adoram simplificar a escrita. Observem, no código acima, como declaramos várias variáveis junto. Quase toda linguagem de programação aceita este formato, em que fazemos várias declarações separadas por vírgulas.

3. Vamos Melhorar Nossos Códigos

3.1 Há Algo que Precisamos Comentar

Nesta unidade, nós vamos trabalhar no código de nossa calculadora, endereçando uma série de problemas do mesmo.

A primeira questão é que o código não está comentado. Comentários são textos que programadores colocam em um código para que eles mesmos e outros programadores possam ler. Eles são mais importantes quanto maior e mais complexo é um código, mas vamos utilizá-los mesmo nestes pequenos códigos da apostila para que você comece a desenvolver bons hábitos. No futuro, a medida que você adquirir naturalidade na programação, vai perceber quando é importante colocar comentários. Por hora, basta pensar que pode haver poucos comentários em um código, mas nunca há comentários demais.

```
----------UMA CALCULADORA SIMPLÓRIA----------

/* Neste código, x recebe um valor que vai ser utilizado para uma operação matemática. Dois valores são lidos e exibimos o resultado da operação */

    var x, n1, n2;

    x = prompt(“entre com uma letra para escolher a operação matemática: (s)ubtração, (m)ultiplicação ou (d)ivisão.”);

    n1 = prompt(“entre com o primeiro número”);

    n2 = prompt(“entre com o segundo número”);

    alert(“O resultado é...”);

    if (x == “s”) alert(n1 - n2); //exibimos a subtração

    if (x == “m”) alert(n1 * n2); // exibimos a multiplicação

    if (x == “d”) alert(n1 / n2); //exibimos a divisão

----------
```

Alguém poderia argumentar que os comentários acima são relativamente inúteis, e não estaria de todo errado, mas estamos utilizando-os apenas para ilustrar o uso de comentários.

Em qualquer linguagem, precisamos indicar de alguma forma que determinado texto não faz parte do programa, está ali apenas para ser lido por programadores.

Javascript adota dois formatos que são talvez os mais comuns e adotados pela maioria das linguagens:

"/*" e "*/" indicam que tudo que está entre estes dois comandos é um comentário.

"//" indica que, a partir deste comando, todo o restante da linha é um comentário.

É bom lembrar que outras linguagens podem adotar códigos diferentes. HTML, por exemplo, utiliza "<!—" e "-->" para seus comentários.

3.2 Não Somos Matemáticos, Podemos Usar Nomes de Variáveis que Seres Humanos Entendem

Na primeira linguagem de programação que utilizei, variáveis eram constituídas de até duas letras. Assim, além de 'x', 'y', 'z', podíamos ter variáveis chamadas de 'x1', 'AX', etc. Nada que um bloco do lado de seu computador com a lista do que cada variável significava não resolvesse, quando você não lembrasse mais o que era o "YK".

Felizmente, estes tempos já se perderam nos distantes primórdios da computação. Basicamente, você pode e deve utilizar nomes de variáveis tão claros e descritivos quanto possíveis. Eu diria que isso é até mais importante que comentar o código, e fundamental quando outras pessoas vão editar seu programa.

Vamos ver o código agora com este ajuste:

----------UMA CALCULADORA SIMPLÓRIA----------

//observe que aqui também juntamos a declaração var com a atribuição, outra forma comum de reduzirmos o código.

var operacaoMatematica = prompt("entre com uma letra para escolher a operação matemática: (s)ubtração, (m)ultiplicação ou (d)ivisão.");

var n1 = prompt("entre com o primeiro número");

var n2 = prompt("entre com o segundo número");

alert("O resultado é...");

if (operacaoMatematica == "s") alert(n1 - n2); //exibimos a subtração

if (operacaoMatematica == "m") alert(n1 * n2); // exibimos a multiplicação

if (operacaoMatematica == "d") alert(n1 / n2); //exibimos a divisão

---

Observe que não alterei os nomes de variável ‘n1’ e ‘n2’. Poderia utilizar, por exemplo ‘numero1’, mas penso que não agregaria mais significado nem deixaria o código particularmente mais claro. Porém, enquanto ‘x’ não têm nenhum significado no exemplo, ‘operacaoMatematica’ já nos dá uma boa ideia de qual a função da variável.

Aqui vale também comentarmos outra ‘melhor prática’, que é o uso de um único padrão para o nome de variáveis. Javascript e suas bibliotecas adotam, tipicamente, o formato denominado ‘camelCase’, no qual a primeira palavra que dá origem a variável é em minúscula, e as demais tem a primeira letra em maiúscula.

Existem outras convenções alternativas, que não trataremos aqui. O importante, como este é um assunto controverso, é que todos de uma equipe adotem uma mesma convenção. Neste curso, por exemplo, utilizaremos em todos os exemplos o ‘camelCase’ como padrão para nomes de variáveis.

3.3 Precisamos de Tantos Alertas?

Temos usado algumas estruturas sem pensar muito a respeito, e não há problema nenhum nisso, mas é importante, aos poucos, começar a entendê-las um pouco melhor.

Por exemplo, quando temos, em um código, o comando “alert(5)”, estamos mandando o programa exibir o número 5. Da mesma forma, quando fazemos “alert(x)”, estamos dizendo para ser exibido o conteúdo de “x”, seja este conteúdo um número ou outra informação. Por último, quando dizemos “alert(“ola mundo”)”, estamos dizendo para o programa exibir o texto “ola mundo”. Vamos chamar este texto de uma “string”. Uma string nada mais é que um conjunto de caracteres.

Existem várias coisas que podemos fazer com strings. Em algumas linguagens, estas operações podem ser feitas de formas um pouco diferentes, mas as operações em si são praticamente iguais. Agora, vamos tratar de apenas uma, que é a concatenação.

Concatenar duas strings significa juntá-las. Em javascript, podemos utilizar o sinal ‘+’. Quando temos dois números, sabemos que o ‘+’ significa uma soma. Pois bem, quando temos duas strings, como “ola” e “mundo”, “ola” + “mundo” significa concatenar, ou seja, juntar, as duas strings, resultando em “olamundo”.

E o que acontece se temos uma string e um número, como, por exemplo, “ola” + 5? Neste caso, o resultado será uma string, a string “ola5”. Mais adiante, quando entendermos bem como um programa trata números e strings, isso vai ficar mais claro, mas pense apenas que concatenar um número e uma string resulta em uma nova string.

Sabendo disso, observe uma nova versão de nosso programa:

----------UMA CALCULADORA SIMPLÓRIA----------

var operacaoMatematica = prompt(“entre com uma letra para escolher a operação matemática: (s)ubtração, (m)ultiplicação ou (d)ivisão.”);

var n1 = prompt(“entre com o primeiro número”);

var n2 = prompt(“entre com o segundo número”);

// colocaremos na variável resultado o valor da operação definida.

if (operacaoMatematica == “s”) resultado = n1 - n2;

if (operacaoMatematica == “m”) resultado = n1 * n2;

if (operacaoMatematica == “d”) resultado = n1 / n2;

alert(“O resultado é “ + resultado);

Observem que “resultado” é uma variável que contém o número resultante das operações. Dentro do “alert” estamos concatenando a string com este número que, como mencionamos, resulta em uma nova string, que é então exibida na janela.

3.4 Novo Pega-Ratão: Soma ou Concatenação?

Agora você vai descobrir por que nossa primeira calculadora não somava. Era para evitar um problema que agora vamos enfrentar. Veja o código abaixo. Tente adivinhar qual o resultado que vai aparecer no alert.

---

var primeiroNumero = “1”;

var segundoNumero = “2”;

alert(primeiroNumero + segundoNumero);

Surpreso com o resultado que apareceu? Vamos ver o que fizemos. Atribuímos a primeiroNumero uma string (strings são sequências de caracteres entre aspas), e fizemos o mesmo com segundoNumero, depois enviamos um alerta da concatenação entre primeiroNumero e segundoNumero, resultando na string composta por dois caracteres “12”.

Vamos ver o que acontece quando não colocamos as aspas:

```
var primeiroNumero = 1;

var segundoNumero = 2;

alert(primeiroNumero + segundoNumero);
```

Agora sim temos o resultado esperado. Então, agora sabemos que uma variável tanto pode conter o número 2, quanto a string “2”.

Isso costuma ser verdade nas diferentes linguagens imperativas. A diferença entre elas é que, em muitas, uma variável só aceita um tipo de dados, e você precisa no início declarar este tipo. Por exemplo “int primeiroNumero;” é um comando em algumas linguagens diferentes da que estamos trabalhando, que diz que primeiroNumero só vai aceitar numeros. Se você tentar colocar uma string, vai aparecer um erro.

Como esta é uma questão complexa, que envolve a diferença entre linguagens, voltaremos a tratar dela mais adiante.

Mas vamos ver o que acontece se tentamos subtrair duas strings que contém digitos. Quando tentamos somar, sabemos que é feita uma concatenação, mas o que acontece na subtração?

```
var primeiroNumero = “2”;

var segundoNumero = “1”;

alert(primeiroNumero - segundoNumero);
```

Talvez você esperasse um erro, e é o que acontece em muitas linguagens. Javascript, porém, optou por tentar fazer o que o programador aparentemente queria. Já que ele não tem como subtrair duas strings, ele assume que na verdade as strings queriam representar números, transformou elas automaticamente em números, e fez a operação.

Ou seja, você não precisou se preocupar. Isso tem um lado bom e um lado ruim. Por um lado, você não precisa se preocupar com o tipo da informação sempre que o javacript sabe o que fazer. Por outro lado, justamente quando o javascript faz algo diferente do que você esperava, ele pode gerar um erro que vai ser difícil de resolver. Esse é um dos motivos que muitos programadores preferem aquelas linguagens em que você diz exatamente as coisas, por exemplo, diz exatamente que “primeiroNumero” só vai receber inteiros.

Vamos, então, ver um último código para chegarmos a raiz de nosso problema de somas na calculadora, que talvez você já tenha advinhado:

```
var primeiroNumero = prompt("entre com o primeiro número");

var segundoNumero = prompt("entre com o segundo número");

alert(primeiroNumero + segundoNumero);
```

Como talvez imaginem, o alerta está concatenando os dois números. Isso se deve ao fato que primeiroNumero e segundoNumero recebem strings, mesmo se você digitar apenas números na entrada. Você pode experimentar digitando textos ou textos e números, para tornar mais claro. Este código acima concatena duas strings, independente delas terem letras ou números.

Vamos ver agora o código da nossa calculadora operando também com somas. Para tanto, faremos o que se chama de conversão de tipos. Se o usuário entrar com a string "12", queremos colocar em uma variável o número 12.

----------UMA CALCULADORA SIMPLÓRIA----------

```
var operacaoMatematica = prompt("entre com uma letra para escolher a operação matemática: a(dição), (s)ubtração, (m)ultiplicação ou (d)ivisão.");

var entradaString1 = prompt("entre com o primeiro número");

//convertemos entradaString1 (uma string) no número n1

var n1 = parseInt(entradaString1);

var entradaString2 = prompt("entre com o segundo número");

//convertemos entradaString2 (uma string) no número n2

var n2 = parseInt(entradaString2);

if (operacaoMatematica  == "a") resultado = n1 + n2;

if (operacaoMatematica  == "s") resultado = n1 - n2;

if (operacaoMatematica == "m") resultado = n1 * n2;

if (operacaoMatematica == "d") resultado = n1 / n2;

alert("O resultado é " + resultado);
```

Nós continuaremos evoluindo nossa calculadora em aulas futuras, a medida que aprendermos novos conceitos e novas boas práticas, mas agora ela está armazenando números em n1 e n2, ao invés de strings. Para tanto utilizamos o comando “n1 = parseInt(entradaString1);”. No futuro entenderemos que “parseInt”, e também “prompt” e “alert” são todos funções que fazem ações específicas, e ensinaremos você a criar suas próprias funções. Por enquanto, basta saber que você pode atribuir a uma variável um número que resulta da string que você escreve no comando “parseInt”.

4. Criando um Pequeno Programa de Cálculo

Nesta unidade vamos colocar um desafio que pode ser resolvido com os conteúdos já apresentados.

4.1 Cálculo de IMC

O cálculo do IMC, ou índice de massa corpórea, é dado pela seguinte fórmula:

IMC = PESO / (ALTURA)²

No nosso desafio, o usuário entrará com as informações de peso e altura. O programa, entretanto, não retornará apenas o IMC, mas também uma avaliação com base no IMC gerado, seguindo a tabela a seguir:

Resultado	Situação
Abaixo de 17	Muito abaixo do *peso*
Entre 17 e 18,49	Abaixo do *peso*
Entre 18,5 e 24,99	*Peso* normal
Entre 25 e 29,99	Acima do *peso*
Entre 30 e 34,99	*Obesidade* I
Entre 35 e 39,99	*Obesidade* II (severa)
Acima de 40	*Obesidade* III (mórbida)

Ou seja, se o usuário entrar com um peso e uma altura que gerem, no cálculo, um número de IMC maior que 25 mas menor que 30, o sistema exibirá um aviso de "Acima do peso".

4.2 Dicas para Solução do Desafio

Se você está com dificuldades de completar o desafio, observe as seguintes dicas:

1 – Utilizando o comando ‘prompt’, você armazenará em variáveis (por exemplo, peso e altura) o peso e altura fornecidos pelo usuário.

2 – Neste momento, considere que o usuário pode entrar com números em fração, e vai utilizar o “.” como forma de entrar com os números fracionários. Mais adiante vamos ver como converter do formato brasileiro (por exemplo 5,10) para o americano (5.10). Neste momento, aceite os dados sem validar.

3 – Por enquanto, evite utilizar funções como parseInt, pois os números serão fracionários, e não inteiros. Como estamos multiplicando e dividindo, o javascript vai converter automaticamente as entradas.

4 – Utilize uma nova variável, por exemplo “imc”, para receber o resultado da operação entre “peso” e “altura”.

5 – Utilize uma sequência de ‘if” e “else” para testar cada condição do IMC.

4.3 Solução do Desafio

```
--------------------------------Cálculo do IMC ------------------------------

var peso = prompt("entre com seu peso");

var altura = prompt("entre com sua altura");

var imc = peso / (altura * altura);

alert("Seu IMC ficou em: " + imc );

if (imc < 17)

  alert("muito abaixo do peso");

else {

  if (imc < 18.5)

    alert("abaixo do peso");

  else {

    if (imc < 25)

      alert("peso normal");

    else {

      if (imc < 30)
```

```
        alert("acima do peso");
      else {
        if (imc < 35)
          alert("obesidade I");
        else {
          if (imc < 40)
            alert("obesidade II");
          else
            alert("obesidade III");
        }
      }
    }
  }
}
```

Algumas questões são importantes de tratarmos aqui. Primeiro, já temos um programa de alguma complexidade, então se você conseguiu criar uma solução que funcione (parecida ou não com esta), está no caminho certo.

Se você não conseguiu, é possível que algumas questões que apareceram neste exemplo tenham atrapalhado você. Vamos comentar sobre elas:

Números fracionários: nós não vamos entrar em detalhes de como as linguagens de programação tratam números fracionários, pelo menos neste momento. Saiba apenas que é algo que você terá que entender pelo menos em um grau superficial em algum momento.

Mas é importante dizermos aqui que, como na maioria das linguagens, o Javascript entende números fracionários na notação em que a fração é indicada por um ponto (‘.’) e não por uma vírgula (‘,’). Nada nos impediria de alterar a string para aceitar também as vírgulas, mas envolve um tipo de recurso que não vimos ainda.

Outra questão dos números fracionários é que não podemos simplesmente utilizar "parseInt", pois esta função retorna um inteiro, ou seja, perderíamos a parte do número depois da vírgula.

Sequências de "if" e "else": para tratarmos as diferentes condições, utilizamos comandos de "if" dentro de outros comandos. No caso, utilizamos o "if" dentro do bloco executado a partir de um "else". Poderíamos ter também um "if" dentro de outro "if".

Este tipo de construção é muito comum, e pode ser confuso para alguém que está começando na área. Observe, por exemplo, que testamos apenas se o IMC é menor que determinado número, nunca se é maior. Isso é possível justamente pelo encadeamento de "if". Por exemplo, no comando "if (imc < 35)" não precisamos testar se o IMC é maior ou igual a 30, já que ele só chegará no comando se for. Se for menor que 30, ele entrará em um dos "if" anteriores, e nunca entrará no else em que temos o "if (imc < 35)".

4.4 Em Dúvida de Como Encadear "IF"s? Pratique

Você precisa saber colocar comandos dentro de um "if", inclusive outros comandos "if".

Talvez isso pareça mais natural a você, e você tenha conseguido criar uma solução para o problema do IMC, ou pelo menos tenha entendido claramente a solução apresentada.

Mas é também possível que, para você, isso ainda não esteja muito claro. Sem problema, só que você, neste caso, terá que praticar, até estas construções de "if" e "else" sejam mais naturais. Não quero entrar em outros comandos de fluxo antes de você ter plena naturalidade com o "if".

Assim sendo, segue uma sugestão de prática. Faça o mesmo código anterior, porém utilizando apenas o operador ">" (ou o ">=", que significa basicamente maior ou igual). Você terá que inverter a sequência de "if", testando primeiro se o IMC é maior ou igual a 40, e colocando os demais testes dentro do else.

Conseguindo inverter a sequência de "if" e trocar o operador de "<" para ">=", gerando o mesmo resultado, penso que você estará aptor a seguir para a próxima unidade.

5. Um Pouquinho de “Lógica Booleana”, mas não Vamos Chamar desse Nome Difícil

5.1 Usando “E” e “OU”

Não existe nada, absolutamente nada, por mais simples, que um profissional de informática não consiga transformar em algo extremamente complexo e assustador.

Creio que um dos melhores exemplos é a “lógica booleana”. Com as “tabelas verdades”, e complexas operações lógicas que você nunca vai usar na prática, é possível assustar qualquer iniciante na área até o ponto dele desistir da profissão.

Vamos, então, ensinar aqui, 90% do que você vai usar daquilo que chamam de “lógica booleana”. Basicamente, vamos falar sobre situações do tipo “OU” e situações do tipo “E”.

O “e” ou “AND” em inglês, é a mais simples de entendermos, pois é exatamente igual ao que utilizamos no dia a dia. Basicamente diz que, para algo ser verdade, as duas condições apresentadas têm que ser verdade.

A única coisa complicada aqui, e é só nos acostumarmos, é que em Javascript, assim como boa parte das linguagens, o comando “AND” ou “E” é escrito assim: “&&”.

Vejam o código abaixo, e tentem adivinhar o que vai ser mostrado:

```
--------------------------------Exemplo de IF com AND----------------------------

var x = 10;

var y = 5;

if ( (x > 6) && (y > 6) )

    alert(“x e y são maiores que 6”);

if ( (x > 3) && (y > 3) )

    alert(“x e y são maiores que 3”);

-------------------------------------------------------------------------------------
```

Se você entendeu, mesmo antes de testar, que o código acima exibirá apenas o alerta de que “x e y são maiores que 3”, parabéns, você já entendeu 50% de tudo que precisa entender sobre lógica booleana neste momento.

Vamos agora aos outros 50% que quero que você entenda neste momento, o comando “OR” ou “OU”. Este comando só tem uma dificuldade adicional em relação ao comando “AND”. O que utilizamos na linguagem coloquial com a expressão “ou” geralmente dá a idéia de exclusividade, ou seja, a situação somente é verdadeira se um dos item for verdadeiro e o outro for falso.

Tudo que precisamos lembrar, para o comando “OR”, é que o resultado será verdadeira caso qualquer uma (OU AS DUAS) condições for verdadeira.

Por fim, importante saber que o comando OR em javascript (e muitas outras linguagens) é representado por “||”. Trata-se da tecla PIPE, que geralmente está ao lado da letra “z” no teclado, sendo necessário clicar em SHIFT, mas isso pode variar de teclado para teclado.

Vamos ver um código com OR, então:

```
--------------------------------Exemplo de IF com AND----------------------------

var x = 10;

var y = 5;

if ( (x > 6) || (y > 6) )

    alert(“x ou y, ou dos dois, são maiores que 6”);

if ( (x > 3) || (y > 3) )

    alert(“x ou y, ou os dois, são maiores que 3”);

-------------------------------------------------------------------------------------
```

Você entendeu que ambos os alertas serão exibidos? Parabéns, você sabe o que precisa saber, por enquanto, sobre lógica booleana.

Obs: se você já teve alguma aula, ou até uma prova, de lógica booleana, talvez esteja se perguntando se estou falando sério. É isso mesmo? Lógica booleana é só isso? Em termos de 90% do que você vai precisar, sim, é só isso. E os outros 10%? Veremos mais adiante, mas, assim, lembre-se sempre que toda vez que você faz uma operação que é difícil para você entender, será difícil para outros programadores entenderem também. O segredo de um bom programador não é saber fazer códigos complexos, é saber fazer códigos simples.

5.2 Vamos Refazer Nosso Teste do IMC

Agora, com o uso de “AND”, podemos reescrever nosso código de cálculo de IMC. É questionável se o programa resultante será melhor que o apresentado originalmente, mas nosso objetivo aqui é didático.

```
--------------------------------Cálculo do IMC ------------------------------

var peso = prompt("entre com seu peso");

var altura = prompt("entre com sua altura");

var imc = peso / (altura * altura);

alert("Seu IMC ficou em: " + imc );

if (imc < 17)   alert("muito abaixo do peso");

if ( (imc >= 17) && (imc < 18.5) ) alert("abaixo do peso");

if ( (imc >= 18.5) && (imc < 25) ) alert("peso normal");

if ( (imc >= 25) && (imc < 30) ) alert("acima do peso");

if ( (imc >= 30) && (imc < 35) ) alert("obesidade I");

if ( (imc >= 35) && (imc < 40) ) alert("obesidade II");

if (imc >= 40) alert("obesidade III");

------------------------------------------------------------------------------
```

Vale observar que o programa acima é ligeiramente menos eficiente que o anterior, porém também pesa o fato de ser também mais fácil de entender. Como não tratamos ainda de aspectos de otimização de código e eficiência, não vamos nos preocupar com isso por enquanto.

6. O Comando Switch

Uma situação muito comum, em programação, é quando temos que optar entre um conjunto de ações convorme os valores de uma variável. Nós já aprendemos a trabalhar com isso utilizando uma sequencia de 'if'.

Por exemplo, observe o exemplo de nossa calculadora, em particular o código marcado em negrito:

----------UMA CALCULADORA SIMPLÓRIA----------

var operacaoMatematica = prompt("entre com uma letra para escolher a operação matemática: a(dição), (s)ubtração, (m)ultiplicação ou (d)ivisão.");

var entradaString1 = prompt("entre com o primeiro número");

//convertemos entradaString1 (uma string) no número n1

var n1 = parseInt(entradaString1);

var entradaString2 = prompt("entre com o segundo número");

//convertemos entradaString2 (uma string) no número n2

var n2 = parseInt(entradaString2);

if (operacaoMatematica == "a") resultado = n1 + n2;

if (operacaoMatematica == "s") resultado = n1 - n2;

if (operacaoMatematica == "m") resultado = n1 * n2;

if (operacaoMatematica == "d") resultado = n1 / n2;

alert("O resultado é " + resultado);

Observe que assumimos que "operacaoMatematica" possa ter 4 valores, com ações diferentes em cada um deles.

O código acima é inclusive ligeiramente ineficiente. Por exemplo, ele testa se "operacaoMatematica" é ou não igual a "s", mesmo na hipótese de operacaoMatematica ser igual a "a" e já termos realizado a soma.

Sem termos esta perda de eficiência, normalmente teríamos esta parte do código escrito da forma abaixo:

```
-------------------------------------------------------------

if (operacaoMatematica  == “a”) resultado = n1 + n2;

else if (operacaoMatematica  == “s”) resultado = n1 - n2;

else if (operacaoMatematica == “m”) resultado = n1 * n2;

else if (operacaoMatematica == “d”) resultado = n1 / n2;

else alert(“operacao inválida”);

-------------------------------------------------------------
```

Neste formato inclusive melhoramos ligeiramente o código também por tratar o cenário em que o usuário entrou com um comando diferente dos quatro especificados.

Ocorre que, como é muito comum este tipo de situação, em que temos diversas ações correspondendo a valores diferentes de uma variável, existe um comando específico para este tipo de situação, o comando “switch”.

Observe o código a seguir, equivalente ao código anterior:

```
-----------------------------------------------------------------

switch(operacaoMatematica) {

   case “a”: resultado = n1 + n2;  break;

   case “s”: resultado = n1 - n2;  break;

   case “m”: resultado = n1 * n2;  break;

   case “d”: resultado = n1 / n2;  break;

   default: alert(“operacao inválida”);

}

-----------------------------------------------------------------
```

Embora seja possível substituir por um conjunto de “if” e “else”, quando o programa está exatamente definindo um entre um conjunto de possíveis ações, o comando switch é mais claro para outros programadores, que já automaticamente entendem que se trata de uma seleção entre várias opções, sem ter que analisar a sequência de “if” e “else”.

7. Controlar o Fluxo é um Desafio

7.1 Uma Soma Infinita

Nosso desafio dessa unidade é começar a controlar melhor o fluxo do programa. Para isso, iremos introduzir um novo comando de controle de fluxo, que resolve um problema que os recursos que vimos até agora não conseguem tratar.

Nosso programa é um somador infinito. Ele vai somar números sem parar, sempre mostrando o resultado e pedindo o próximo número. Quando for escrito “fim” na entrada, em vez de um número, o programa termina.

Vamos a ele:

```
-------------------------UM SOMADOR SIMPLÓRIO--------------------------

    var novaEntradaString = “0”;

    var resultadoSomas = 0;

    var n1;

    while (novaEntradaString != “fim”) {

        novaEntradaString = prompt(“entre com um número para somar, ou digite
  fim para encerrar”);

          n1 = parseInt(novaEntradaString);

          resultadoSomas = resultadoSomas + n1;

          alert(“O resultado até o momento é: “ + resultadoSomas);

    }

-------------------------------------------------------------------------------------
```

Como no exemplo anterior, há problemas neste código que trataremos mais adiante, a medida que vamos aprendendo mais sobre programação. O que importa é que ele gera o resultado que queremos. Digitando diversos números, eles vão sendo somados, até digitarmos “fim”.

Entretanto, este é um programa que não conseguiríamos fazer sem um novo comando que nos permita não apenas desviar o fluxo de execução, mas principalmente, voltar atrás e repetir um comando. Só que antes vamos tratar de uma outra coisa, primeiro....

7.2 Mas que Raios é Este "!="

Vamos vê-lo muito, mais adiante. Por hora, basta saber que "!" significa "NÃO". Assim, "!=" basicamente significa "não igual". Algumas linguagens utilizam "<>", que significa em essência a mesma coisa.

Se você está em dúvida, apenas observe que os dois códigos abaixo fazem a mesma coisa. Em dúvida ainda? Digite-os e pratique com eles. Este é um conceito que você não pode ter dúvidas antes de seguir adiante no curso:

----------DOIS CÓDIGOS QUE FAZEM A MESMA COISA----------

```
var testeNumero = prompt("entre com um número");

if (testeNumero == 10) alert("o número é 10");

else alert("o número não é 10");
```

```
var testeNumero = prompt("entre com um número");

if (testeNumero != 10) alert("o número não é 10");

else alert("o número é 10");
```

7.3 O Comando "While"

Agora, vamos olhar nosso segundo comando de controle de fluxo de nosso curso: o "while".

"While", que significa "enquanto" em inglês, basicamente repete uma instrução ou um bloco de instruções (marcado entre chaves, como já vimos no "if").

E ele repete quantas vezes? Ele vai repetir enquanto (daí seu nome) a condição entre parenteses for verdadeira.

Percebem as semelhanças e diferenças do comando "if"? Para ambos, se o que está entre parenteses é verdadeiro ou falso, eles vão direcionar o fluxo de alguma forma. A diferença é que, no caso do "while", o que ele faz é entrar dentro do bloco repetidas vezes, até a condição entre parenteses ser falsa.

Isso significa que o código dentro das chaves pode não ser executado nenhuma vez (por exemplo se a condição já for de início falsa), até infinitas vezes.

Então, inicialmente "novaEntradaString" é igual a "0", logo é diferente de "fim". Ele entra no código, e ao final testa novamente, para ver se vai repetir mais uma vez ou não. Se "novaEntradaString" novamente for diferente de "fim", ele novamente entra no bloco.

Dizemos que, quando ele está entrando neste bloco de comandos do "while", ele está "entrando no loop", e que quando ele finalmente sai do bloco, ele está "saindo do loop".

7.4 O Desafio de uma Calculadora Infinita

A esta altura, você sabe como fazer uma calculadora com 4 operações. Sabe também como fazer um "loop infinito", do qual o programa sai apenas quando o usuário digitar "fim".

Nosso desafio, agora, é juntar as duas coisas. Tente fazer uma calculadora que vá fazendo operações uma sobre a outra, mas ao invés de termos apenas somas, como no exemplo acima, queremos as quatro operações, como vimos em unidades anteriores.

Assim, você primeiro entra com um número, depois define a operação e o segundo número. Depois uma nova operação e o próximo número, sempre aplicando sobre o resultado anterior, até receber o comando "fim".

7.5 Algumas Dicas

Ok, você já fez a calculadora? Pode ir direto para o subitem seguinte.

Nem tentou, bom, primeiro tente, antes de continuar.

Se você já tentou, mas está empacado, bom, neste, e somente neste caso, esta unidade é para você.

Vamos tentar pensar como um programador e em como ele resolve um problema...

Primeiro, nós já temos uma parte da solução, certo? Já temos como fazer um loop infinito que apenas soma. Dê uma olha lá no início da unidade e tenha certeza que você está entendendo bem o que é feito lá.

Agora, compare com o exemplo anterior que fazia várias operações. Observe que neste novo exemplo, temos na verdade um único "prompt" e uma única variável que

recebe dados. Ela recebe nosso número e acumula ainda a função de receber o comando “fim”.

No exemplo anterior, porém, tínhamos dois prompts e duas variáveis. Uma para receber a operação e uma para receber o número. É aqui que temos a chave da solução do problema.

Substitua um comando “prompt” do exemplo acima, por dois comandos, como no primeiro exemplo da calculadora. Um dos comandos define a operação, e o outro o número. Coloque, após, o mesmo código de soma, divisão, etc, com base no “if” que já vimos antes.

Ainda sem progressos? Tente realmente um pouco mais, sozinho. É importante que você tente até conseguir realizar este código, para só então seguir para o próximo passo, que é analisar a solução que apresentamos.

Lembre-se sempre, programar não é apenas conhecer os comandos e códigos, mas principalmente desenvolver a habilidade de resolver problemas. Eu realmente quero que você tenha segurança nestes exemplos antes de começarmos desafios mais complexos.

7.6 Resultado do Desafio Calculadora Infinita

Observem o código proposto a seguir. Conforme sua habilidade e naturalidade com programação, você já pode estar até desenvolvendo códigos mais elegantes. Este exemplo a seguir tem como foco ser simples e com base no que ensinamos até agora:

```
-------------------------UM SOMADOR SIMPLÓRIO--------------------------

var operacaoMatematica = "";

var entradaInicial = prompt("entre com o primeiro número");

var resultado = parseInt(entradaInicial);

var novaEntradaString, n1;

while (operacaoMatematica != "fim") {

  operacaoMatematica = prompt("entre com uma letra para escolher a operação matemática: a(dição), (s)ubtração, (m)ultiplicação ou (d)ivisão, ou (fim) para encerrar.");

  novaEntradaString = prompt("entre com o próximo número");

  n1 = parseInt(novaEntradaString);

  if (operacaoMatematica  == "a") resultado = resultado + n1;

  if (operacaoMatematica  == "s") resultado = resultado - n1;

  if (operacaoMatematica == "m") resultado = resultado * n1;
```

```
    if (operacaoMatematica == "d") resultado = resultado / n1;
    alert("O resultado até o momento é: " + resultado);
  }
```

Se você não conseguiu gerar uma solução equivalente, estude-a. Execute o código e tente entender o que cada linha de código executa.

Na próxima unidade, trataremos de melhorar estes códigos desta unidade, mas nas unidades seguintes já começaremos a trabalhar com desafios mais complexos, então realmente é fundamental você se sentir seguro com os desafios até o momento.

Se não conseguiu realizar sozinho este desafio, estude-o e tente modificar o programa, alterar os nomes de variáveis, colocar outras condições, por exemplo ter um comando que define diretamente o valor de “resultado” para poder reiniciar os cálculos.

Enfim, se ainda em dúvida, pratique.

8. Exercitando nossa Capacidade de Programação

A seguir propomos uma série de desafios que estão no mesmo nível de complexidade dos exemplos que já trabalhamos. É importante que você faça os exercícios e tenha segurança na programação.

A medida que mais conteúdos forem apresentados, teremos desafios bem mais complexos para implementar.

8.1 Vamos Encontrar o Maior e Menor Número

Neste desafio, o usuário vai entrando com números, em sequência, e cada vez que ele entrar com um novo número, o programa deve mostrar o maior e menor número de toda a sequência até então. Ao digitar “fim”, o programa encerra.

8.2 Cálculo do Fatorial

Entrando com um número, o sistema mostrar o fatorial deste número. Exemplo, ao entrar com o número 4, o sistema irá mostrar o resultado de 4 x 3 x 2 = 24.

8.3 Cálculo do Fatorial em um Loop

Neste exemplo, faça o mesmo código anterior, porém repita o processo até o usuário digitar “fim”. Ou seja, se o usuário digitar 3, o sistema retorna 6 (fatorial de 3), depois aguarda nova entrada do usuário e exibe seu fatorial, até que a entrada do usuário seja “fim”.

8.4 Resposta dos Exercícios

Confira e, caso não tenha completado algum, confira a resposta dos exercícios propostos. Se não tiver conseguido realizar nenhum dos exercícios, confira a resposta do primeiro, estude, e então tente fazer os dois restantes.

```
-------------------------MAIOR E MENOR NÚMERO--------------------------

var menorNumero = 0;

var maiorNumero = 0;

var entrada = prompt("entre com um número, ou ‘fim’ para encerrar");

alert("o maior número até agora é " + entrada + " e o menor número é " + entrada );

menorNumero = parseInt(entrada);

maiorNumero = menorNumero;
```

```
while (entrada != "fim") {
  entrada = prompt("entre com um número, ou ‘fim’ para encerrar");
   var temp = parseInt(entrada);
   if (temp > maiorNumero)
            maiorNumero = temp;
  if (temp < menorNumero)
            menorNumero = temp;
  alert("o maior número até agora é " + maiorNumero + " e o menor número é " + menorNumero);
}
```

----------FATORIAL ----------

```
    var entrada = parseInt(prompt("entre com o número"));
    var resultado = entrada;

    while (entrada > 1) {
      entrada = entrada - 1;
      resultado = resultado * entrada;
    }
    alert("O fatorial é " + resultado);
```

Se você não conseguiu resolver os exercícios, tente ao menos fazer o terceiro exercício tomando por base a resposta acima e os exemplos que já montamos de loops.

----------FATORIAL EM LOOP----------

```
    var entrada = prompt("entre com o número");
    var resutado;
```

```
while (entrada != "fim") {
  resultado = entrada;
  while (entrada > 1) {
    entrada = entrada - 1;
    resultado = resultado * entrada;
  }
  alert("O fatorial é " + resultado);
  entrada = prompt("entre com o número");
}
```

Observe que fizemos um “while” dentro de outro “while”. Este tipo de construção é muito comum.

Entretanto, se você construir um código com 3, 4, 5 ou mais condições de loop umas dentro das outras, seu código ficará muito confuso. Para evitar isso, podemos isolar os códigos dentro de funções, recurso que veremos na unidade seguinte.

9. Criando Funções

9.1 Uma Função que Calcula o IMC

Observe o código a seguir:

```
--------------------------------Função para IMC -----------------------------
function calculaIMC(peso, altura)
{
  var imc = peso / (altura * altura);
  return imc;
}


var peso = prompt("entre com seu peso");
var altura = prompt("entre com sua altura");
var imc = calculaIMC(peso,altura);
alert("Seu IMC ficou em: " + imc );
if (imc < 17)   alert("muito abaixo do peso");
if ( (imc >= 17) && (imc < 18.5) ) alert("abaixo do peso");
if ( (imc >= 18.5) && (imc < 25) ) alert("peso normal");
if ( (imc >= 25) && (imc < 30) ) alert("acima do peso");
if ( (imc >= 30) && (imc < 35) ) alert("obesidade I");
if ( (imc >= 35) && (imc < 40) ) alert("obesidade II");
if (imc >= 40) alert("obesidade III");
--------------------------------------------------------------------------
```

Nós já vimos este código antes, só que agora ele está escrito de uma forma ligeiramente diferente. O que nós fizemos foi isolar a parte que faz o cálculo do IMC em uma código separado, que vamos chamar de função.

Vamos agora discutir como se cria uma função, como ela trabalha, quais as vantagens e desvantagens de seu uso.

9.2 Parâmetros e Retornos de uma Função

Uma função, em linguagens de programação imperativas (que é o que trataremos nesta apostila), pode receber parâmetros e pode gerar um retorno. Por exemplo, no cálculo do IMC, temos dois parâmetros, o peso e a altura. Esta função tem um retorno que é o valor do IMC calculado.

Nós já conhecemos outras funções, que fazem parte da linguagem javascript. O "alert", por exemplo, recebe uma string, que será exibida em uma janela. O "prompt", além de exibir o parâmetro informado, também retorna uma string com o valor fornecido pelo usuário.

Observe como especificamos os parâmetros e o retorno. Os parâmetros são definidos entre parênteses, separados por vírgula, enquanto para o retorno, utilizamos o comando "return":

```
function calculaIMC(peso, altura)

{

  var imc = peso / (altura * altura);

  return imc;

}
```

9.3 Utilizamos Funções para Não Repetir Códigos

Uma das razões mais óbvias para utilizarmos funções é evitarmos a duplicação de código. Isso é mais fácil de exemplificar em códigos com uma complexidade maior do que os códigos que vamos ver nesta apostila, mas imagine que o cálculo do IMC fosse mais complexo, envolvendo diversas linhas de código. Imagine também que seu programa precisasse calcular o IMC em diversos momentos diferentes. Ao invés de repetir o cálculo em inúmeras partes do código, fazemos o código do cálculo uma única vez, na função IMC.

9.4 Utilizamos Funções para Facilitar a Manutenção e Tornar o Código Legível

Funções permitem agrupar códigos, facilitando sua localização e alteração. Por exemplo, vamos supor que você queira agora mudar a fórmula de cálculo do IMC. Mesmo que seja utilizada uma única vez em seu programa, é muito mais fácil localizar a função IMC e fazer as alterações nela, que tentar localizar o cálculo em meio ao código. Em códigos muito grandes, esta se torna uma questão relevante.

9.5 Funções Podem Ser Utilizadas Facilmente em Outros Programas

Já que é possível criar funções que executam determinadas tarefas, não poderíamos então criar uma série de funções muito úteis e disponibilizá-las para que qualquer um as utilize em seus códigos?

Se você teve esta ideia, parabéns, você acabou de reinventar as bibliotecas de funções. Pode-se criar, por exemplo, uma biblioteca para trabalhar com objetos 3D, ou uma biblioteca das funções utilizadas normalmente em sites, ou para construção de jogos, etc. Pesquisando na internet, você encontrará bibliotecas de funções, que são basicamente conjuntos de funções prontas que realizam estas e inúmeras outras funcionalidades.

9.6 Desafio: Vamos Criar uma Primeira Função

Vamos fazer um desafio muito simples, apenas para que você pratique a criação de funções:

Crie uma função “entradaInteiro”, que funcione exatamente como a função “prompt”, exceto que, ao invés de receber uma string, a função receberá um número inteiro.

9.7 Solução

Observe o código abaixo.

```
-------------------------CRIANDO UMA FUNÇÃO------------------

function entradaInteiro( mensagem )

{

  var retornoString = prompt(mensagem);

  var retornoInteiro = parseInt(retornoString);
```

```
  return retornoInteiro;

}

-------------------------------------------------------------------------------------
```

Espero que tenha conseguido resolver este primeiro desafio, mas mesmo que tenha tido dificuldade, o código em si deve ser relativamente óbvio de entender.

9.8 Funções podem ser utilizadas diretamente em Cálculos e Parâmetros

Quando queremos utilizar o retorno de uma função em um único local, não precisamos necessariamente atribuir este retorno a uma variável, para então fazer uso da variável. Podemos utilizar diretamente a função, simplificando e reduzindo o código, e não criando uma variável que efetivamente não tem nenhuma necessidade de existir.

Observe as duas evoluções do código anterior, e perceba que realizam exatamente o mesmo resultado do código original.

```
-------------------------ESTE É UM EXEMPLO TÍPICO------------------

function entradaInteiro( mensagem )

{

  var retornoString = prompt(mensagem);

  return parseInt(retornoString);

}

-------------------------------------------------------------------------------------

---ESTE É MAIS SOFISTICADO, MAS TAMBÉM MUITO COMUM----

function entradaInteiro( mensagem )

{

  return parseInt(prompt(mensagem));

}

-------------------------------------------------------------------------------------
```

Isso significa que, se tivermos uma longa lista de comandos em sequência, um como entrada do outro, podemos reduzir diversas linhas a uma única chamada de funções como parâmetros de outras funções, fazendo todo um código virar uma única linha enorme, da mesma forma que este parágrafo está rapidamente se tornando uma imensa frase, gramaticalmente correta, ou pelo menos eu assim espero, mas a cada momento mais confusa e complexa de entender, a tal ponto que, se eu estivesse lendo-o em voz alta, estaria começando a perder o fôleogo e talvez mesmo estivesse quase a esquecer que o ponto original do presente texto é a discussão sobre juntar inúmeros comandos em uma única função de uma linha, e, o mais importante, possivelmente por analogia mostrando ao leitor que, no momento que perdemos clareza na leitura, estamos ultrapassando o limite do razoável na construção do código da mesma forma que, penso, este parágrafo há muito ultrapassou o que seria um tamanho consistente para uma frase.

Resumindo, qual o limite para quanto código devemos juntar em uma linha, reduzindo o uso de variáveis? O bom senso.

9.9 Funções e Escopo de Variáveis

Em muitos códigos javascript você encontra a declaração de ‘var’ nas variáveis utilizadas, da seguinte forma:

```
var x;

x = 1;
```

Ou simplesmente:

```
var x = 1;
```

Porém em javascript e algumas outras linguagens, o mesmo código, a maioria das vezes, vai executar o mesmo resultado se você simplesmente não declarar as variáveis.

A declaração de variáveis, em algumas linguagens, é utilizada para definir o tipo de uma variável, ou seja, o tipo de dados que ela vai aceitar. Por exemplo, uma variável em diversas linguagens é declarada como “int x;” para dizer que ela só aceita números inteiros, e vai acontecer um erro se você tentar atribuir a ela uma string.

Tipicamente, nestas linguagens, você só pode utilizar uma variável depois de declará-la. Isso evita um erro comum em linguagens como javascript, em que, sem querer, você comete um erro de digitação ao escrever a variável, por exemplo trocando uma letra minúscula por maiúscula, e não percebe que o programa agora faz algo diferente do que você esperava.

A linguagem que estamos utilizando para ensinar programação, javascript, não tem 'tipos de variáveis' (isso tem um nome, se chama linguagem fracamente tipada, em oposição a linguagens fortemente tipadas). Entretanto, existe um outro motivo que pode tornar necessário declarar uma variável: a definição de escopo.

Para entendermos, vamos olhar e comparar os dois códigos a seguir.

```
var i = 1; //vamos utilizar i como o número de vezes que nosso código é utilizado.


function calculoComplexoTresNumeros( a, b, c)
{
   i = a + b; //vamos somar a e b e colocar em uma varíavel temporária
   i = i / 2;
   i = i + c;
   return(i);
}


//nosso programa vai fazer um cálculo complexo com os números 1, 2 e 3
alert("Esta é a execução de número " + i);
alert("Nosso resultado é " + calculoComplexoTresNumeros(1,2,3) );
alert("Terminamos a execução de número " +i);
i++;
alert("Vamos começar a execução de número " + i);
```

Existem vários motivos para o código acima ser muito ruim, e espero que a esta altura vocês não estejam mais escrevendo códigos assim. Em especial, a variável que define o número da execução deveria se chamar algum nome que o programador facilmente entendesse, como “numeroExecucao”.

Mas o verdadeiro problema aparece quando o código é executado e não faz o que era esperado.

Por quê? Porque o valor da variável ‘i’ foi alterado dentro da função. “i” é o que chamamos de uma variável global, uma variável que foi criada fora de qualquer função, e, no entanto, seu conteúdo foi alterado dentro da função “calculoComplexoTresNumeros” com o comando “i = a + b”. Quem programou a função, porém, e pode nem ter sido a mesma pessoa que criou a variável “i”, estava apenas utilizando uma variável temporária qualquer.

Este problema estará resolvido se utilizarmos o comando “var” dentro da função. Ele faz com que uma nova variável (que pode ter o mesmo nome de uma variável existente, não importa) exista apenas dentro da função, e deixe de existir quando saímos da função. O mesmo código (ainda horrível) estará correto na forma a seguir:

```
var i = 1; //vamos utilizar i como o número de vezes que nosso código é utilizado.

function calculoComplexoTresNumeros( a, b, c)

{

   var i = a + b; //vamos somar a e b e colocar em uma varíavel temporária, que só vai existir até sairmos da função.

   i = i / 2;

   i = i + c;

   return(i);

}

//nosso programa vai fazer um cálculo complexo com os números 1, 2 e 3

alert(“Esta é a execução de número “ + i);

alert(“Nosso resultado é “ + calculoComplexoTresNumeros(1,2,3) );
```

```
alert(“Terminamos a execução de número “ +i);

i++;

alert(“Vamos começar a execução de número “ + i);

---------------------------------------------------------------------------------
```

Chamamos a isso de escopo. O “i” de fora da função, é uma variável de escopo global, ou simplesmente variável global, e o “i” de dentro da função é uma variável de escopo local, ou simplesmente variável local.

Questões de escopo e uso de variáveis são tão importantes que retomaremos na próxima unidade, que será exclusivamente dedicada ao uso de variáveis.

10. Algumas Informações sobre Variáveis e Usos Delas

Nós já vimos algumas informações sobre variáveis ao longo do curso, mas aqui vamos reunir um conjunto de informações relevantes para facilitar sua compreensão.

10.1 Vamos Facilitar Mudar o Valor de Uma Variável

É muito comum alterarmos o valor de uma variável somando ou subtraindo um número a ela. Isso é tão comum que temos alguns comandos específicos para fazer isso. A seguir mostramos o comando e seu equivalente em operações com o operador '='. Estes comandos são comuns a praticamente todas as linguagens de programação comerciais utilizadas hoje em dia.

- i++ é igual a i = i + 1

- i-- é igual a i = i – 1

- i += 5 é igual a i = i + 5

- i -= 5 é igual a i = i – 5

Ou seja, estes operadores '++', '--', '+=' e '-=' são bastante simples de entender. Eles basicamente adicionam '1' ou um número (ou subtraem) de uma variável.

A partir de agora, sempre que fizer sentido, iremos utilizar estes operadores, pois é o que você irá encontrar também em programas que vier a ter acesso.

10.2 Algumas Vezes um Contador é Apenas um Contador

Um uso muito comum de variáveis, e já fizemos em exemplos anteriores, é como contadores. Por exemplo, se tivermos que repetir um loop por 10 vezes, normalmente utilizaremos um contador que vai variar, por exemplo de 1 a 10 (mais usualmente, utilizaremos de 0 a 9, e vamos mais adiante explicar por quê).

Você não precisa dar a estes contadores um nome semanticamente significativo, a menos que faça sentido. Muitas vezes o contador está ali apenas para contar o número de repetições do loop.

Neste caso, é de praxe utilizarmos a variável 'i', como no exemplo abaixo:

----------ESTE CÓDIGO VAI REPETIR 10 VEZES----------

var i = 0;

```
while (i < 10) {
    alert(i);
}
```

Muitas vezes você precisará de mais de um contador, por exemplo com um ‘while’ dentro de outro ‘while’. Neste caso, é também comum utilizarmos a sequência ‘i’, ‘j’ e ‘k’, embora programadores possam ter suas preferências específicas.

10.3 Linguagens Fortemente Tipadas e Fracamente Tipadas

Vocês já estão trabalhando com tipos de variáveis faz algum tempo. Inclusive já falamos sobre o problema que ocorre se você tenta somar dois números que são não verdade strings do número, e da necessidade de utilizar a função ‘parseInt’.

Este tipo de problema não existe desta forma em linguagens fortemente tipadas. Nestas linguagens, antes de utilizar uma variável, você precisa declará-la e dizer seu tipo. Uma variável do tipo ‘inteiro’ só vai aceitar receber números. Se você tentar colocar uma string, um erro será gerado.

Em linguagens fracamente tipadas, uma variável não tem um tipo previamente definido. Por exemplo – e já vimos isso anteriormente – eu posso atribuir um número para uma variável, e depois uma string, e depois um número fracionário. Isso não quer dizer que não existam tipos diferentes de dados, apenas que a variável pode variar seu tipo ao longo da execução.

Em geral, é mais fácil escrever programas em linguagens fortemente tipadas, pois o custo de ter que primeiro declarar e definir o tipo da variável é irrisório. As vantagens de não enfrentar um erro como os que já vimos, por outro lado, são bem relevantes.

10.4 Rever Alguns Tipos que Já Usamos: Tipo Inteiro

Uma variável do tipo inteiro está armazenando um número, positivo ou negativo, não fracionário. É possivelmente o tipo mais fácil de entendermos, pois estamos acostumados a trabalhar com variáveis desta forma.

Se você estiver programando em uma linguagem fortemente tipada, terá que cuidar para ter certeza que está utilizando a variável apenas como contador ou com cálculos que gerem apenas números inteiros. Se você fizer uma operação que gere um resultado com vírgulas, por exemplo, poderá gerar um erro ou, muito pior, em certas circunstâncias terá um número apenas da parte inteira do resultado, com a parte em fração sendo descartada.

---------LINGUAGEM FORTEMENTE TIPADA INVENTADA------

INTEIRO x;

x = 5 / 2;

//OPA, ou vai gerar erro, ou x vai receber 2. Fato é que x não receberá 2,5 por não ser um valor inteiro

---

10.5 NaN? O que Diabos é Isso?

Vamos retomar um de nossos primeiros programas, e mostrar um problema no mesmo:

----------UM SOMADOR SIMPLÓRIO----------

```
var novaEntradaString = "0";

var resultadoSomas = 0;

var n1;

while (novaEntradaString != "fim") {

    novaEntradaString = prompt("entre com um número para somar, ou digite fim para encerrar");

        n1 = parseInt(novaEntradaString);

        resultadoSomas = resultadoSomas + n1;

        alert("O resultado até o momento é: " + resultadoSomas);

}
```

Já conhecemos este código. Só que, desta vez, ao executá-lo, depois de algumas somas, você vai digitar "FIM" ao invés de "fim".

É relativamente fácil de entender por que o programa não encerrou, certo? "FIM" é diferente de "fim" (e vamos tratra de como endereçar este tipo de questão no futuro). O problema é o número que apareceu depois: NaN.

A questão é o que é retornado da função “parseInt(‘FIM’)”? Esta função não pode retornar uma string, pois o retorno dela é um número. Mas qual o número que vai ser retornado ao tentarmos transformar em inteiro a palavra “FIM”?

Existem algumas possibilidades. Algumas linguagens podem retornar de imediato um erro, ou apenas retornar 0. Javascript optou por retornar um valor especial do tipo número, o NaN, que significa: não é um número (not a number).

A partir deste momento, a calculadora não faz mais nenhuma operação, pois ao somarmos um número qualquer com ‘NaN’ temos como resultado ‘NaN’, que é atribuído a ‘resultadoSomas’.

Ao final desta unidade vamos mostrar como tratar esta situação utilizando a função ‘isNaN’. Se quiser, você já pode se testar implementando a solução por conta própria.

10.6 Rever Alguns Tipos que Já Usamos: Tipos Fracionários

Eu chamei de “tipos” e não “tipo”, pois aqui temos uma questão mais complexa, que envolve representação de números. Nós não vamos nos aprofundar neste tema aqui, pela sua complexidade, mas é algo que você terá que ter um conhecimento mais profundo se for trabalhar com programas que efetuem cálculos matemáticos com uma necessidade de precisão.

Vamos apenas explicar o problema que você deve cuidar ao lidar com números reais, e para isso vamos começar perguntando: quanto é 10 dividido por 3? Escreva este número todo na forma de uma fração decimal:

Conseguiu? Não, né. Em algum momento você teve que parar de escrever e simplesmente colocar ‘...’: “3,33333333....”.

Bom, e como o computador vai representar este número? A solução típica é ele fazer algo equivalente, basicamente armazenando os números até onde ele consegue. Não é exatamente como fizemos acima, pois o computador não utiliza base 10 (números formados por dígitos de 0 a 9), mas sim binário, e geralmente utiliza um modelo chamado “ponto flutuante”, mas o resultado prático é semelhante: vai haver alguma margem de erro, dependendo do número que estamos trabalhando.

O que quero dizer com tudo isso? Duas coisas que você precisa saber:

1 – Existem diferentes formas de armazenas um número real, com maior ou menor precisão

Em algumas linguagens você terá tipos diferentes, que ocupam mais ou menos memoria, e tem maior ou menor precisão. Por exemplo, em ‘C’, você tem tipos

como “float”, que armazena um número em 32 bits, e “double”, que armazena o mesmo número em 64 bits.

Para o Javascript, em particular, inteiros e reais são todos tratados como “number”.

2 – Erros de precisão podem se acumular

Por favor, se um dia você trabalhar no desenvolvimento de um míssel, ou no controle da turbina de um avião, não desenvolva o software com base exclusivamente nesta apostila.

Ocorre que o que pode parecer uma variação insignificante pode se acumular rapidamente ao longo do programa. Isso significa que, em determinado momento, você espera receber o número ‘2’, e, no lugar recebe o número ‘1,99999999999’, apenas porque realizou algumas divisões e multiplicações e não se deu conta que estava perdendo um pouco de precisão no meio dos cálculos.

Você não entendeu ou não acreditou em mim? Então veja o problema abaixo e descubra que programas também podem errar:

----------VAMOS FAZER O COMPUTADOR ERRAR----------

```
var x = 10;

var i = 0;

while (i < 100) { //dividimos x por 3, por 100 vezes

  x = x / 3;

  i++;

}

i = 0;

while (i < 100) { //multiplicamos x por 3, por 100 vezes

  x = x * 3;

  i++;

}

alert(x);
```

O resultado deveria ser 10, mas provavelmente você viu um alerta com um resultado diferente. Nada diferente do que costumávamos fazer quando criança, quando dividíamos e multiplicávamos um número em nossas calculadoras, até a calculadora mostrar um número errado.

Ok, nada diferente do que **EU** costumava fazer.

Ok, ok, eu não tinha vida, mas voltando....

Tipos com fração são perigosos, tome cuidado com eles, especialmente se estiver utilizando divisões e multiplicações.

10.7 Voltando ao Tipo Inteiro

Vamos dar apenas mais uma palavrinha sobre o tipo inteiro para dizer: também em tipos inteiros temos um limite para a capacidade do computador armazenar o número.

Em geral, não costumamos esbarrar neste limite, mas existem situações que podem gerar problema, especialmente se estiver trabalhando com números muito grandes e/ou alguma restrição na capacidade de armazenamento.

Por exemplo, em linguagem 'C', o tipo "int" é especificado na linguagem como tendo que aceitar números entre "-32.767" e "32.767", o que não são valores tão altos assim. Hoje em dia, tipicamente as implementações de 'C' trabalham com escalas bem maiores para o tipo 'int', mas é importante saber que existem limites. Se você for trabalhar intensivamente com números, é bom saber como eles estão sendo armazenados no computador.

10.8 Rever Alguns Tipos que Já Usamos: Tipo string

Nós já vimos que uma variável também pode receber valores do tipo 'string'. Uma string nada mais é que um conjunto de caracteres, que podem inclusive ser números.

Nem todas as linguagens tem o tipo 'string' de forma nativa (ou seja, na linguagem em si, sem utilizar bibliotecas e frameworks, que são como extensões da linguagem – mal comparando) . Em 'C', por exemplo, este tipo não existe.

Vamos falar aqui de string em javascript, outras linguagens podem ter pequenas diferenças.

Para definirmos uma string podemos utilizar aspas simples ou aspas duplas, e podemos ter dentro da string aspas também, desde que sejam diferentes para o computador entender que não estamos fechando a string. Acompanhe as diferentes strings atribuídas abaixo. Coloque alertas para ver o resultado, se desejar:

```
---------------------------------------------------------------------------

x = “a”;

x = ‘alfa’;

x = “alfamídia”;

x = “16”;

x = “string com ‘ola mundo’”;

x = ‘outra string com “ola mundo”’;

---------------------------------------------------------------------------
```

Além de concatenar strings, que já vimos anteriormente (operador ‘+’), há uma série de outras operações que podemos realizar. Podemos substituir letras dentro da string, cortar a string, acessar posição a posição da string, etc. Algumas das funções de tratamento de strings em javascript serão vistas em uma unidade futura. Em outras linguagens, em geral existem funções equivalentes, talvez apenas com alguma pequena diferença no nome da função ou na ordem dos parâmetros.

10.9 Um Tipo Novo, o Boolean, e Seu Uso

O tipo boolean, ou bool, tem apenas dois valores, verdadeiro ou falso (ou eventualmente, em alguma linguagem, três: verdadeiro, falso e indefinido. Não vamos nos preocupar com isso ainda).

Vamos mostrar seu uso e depois discutir a respeito:

```
-----SOMADOR INFINITO UTILIZANDO UMA VARIÁVEL BOOLEAN----

var testeCondicao = true;

var entrada;

var resultadoSomas = 0;

while (testeCondicao) {

  entrada = prompt("entre com um número");

  if (entrada == "fim") {

    testeCondicao = false;
```

```
  } else {
    resultadoSomas += parseInt(entrada);
    if (isNaN(resultadoSomas)) {
      testeCondicao = false;
    }
    else
      alert(resultadoSomas);
  }
}
```

Escreva o código do somador infinito acima e observe seu resultado. Exceto pelo uso da função "isNaN" e da variável "testeCondicao", ele não é muito diferente do código de soma infinita que já vimos antes.

Entretanto, o loop while continua até a variável "testeCondicao" não ser mais verdadeira. Inicialmente ela recebe o valor "true" (verdadeiro), mas em duas situações diferentes ela pode receber o valor "false", encerrando o loop.

A rigor, poderíamos programar sem precisar utilizar o tipo booleano. Poderíamos, por exemplo, definir "testeCondicao" com o valor "1", e fazer o loop prosseguir enquanto "testeCondicao" não for "0". Desta forma, teríamos o mesmo resultado, trabalhando com "1" e "0" no lugar de "true" e "false". No entanto, é bom nos acostumarmos a criar códigos compreensíveis para programadores, e utilizar o tipo boolean para situações em que estamos usando uma variável apenas para um teste de verdadeiro ou falso tornará o código mais legível.

Obs: talvez você tenha percebido que, a rigor, neste cenário, não precisamos de dois testes. Já que o sistema irá encerrar em qualquer entrada que não seja um número, não é necessário testar a string "fim" que fará o mesmo que qualquer outra string. O objetivo do teste no código acima é didático.

10.10 Que Tipo de Variável que Eu Sou?

Como em javascript uma variável pode ser de diferentes tipos, e inclusive mudar de tipo sem percebermos (por exemplo já vimos que um operador '+' pode resultar em uma soma de dois inteiros ou uma concatenação de duas strings), é conveniente sermos capazes de descobrir que tipo é uma variável.

Nestes casos, o javascript (assim como tipicamente outras linguagens em que o tipo das variáveis pode variar) possui uma forma de identificarmos dinamicamente o tipo de uma variável. Observe os códigos abaixo, para compreender o uso de “typeof”:

```
var x = "ola";

alert(typeof x);

x = 5;

alert(typeof x);

x = false;

alert(typeof x);

x = "5";

alert(typeof x);

if (typeof x == "number")   alert("é um número");

if (typeof x == "string")      alert("é uma string");
```

Observe que, mesmo uma string que pode ser convertida para um número, como “5”, ainda é uma string. Tanto que, como já vimos, se usarmos o operador “+” em duas strings de valor “5”, teremos como retorno “55”.

11. A Estrada Até Aqui

Enquanto ouvimos “Carry on Wayward Son”, vamos revisando o que aprendemos até agora, e trabalhando com um exemplo de código para praticarmos:

Um programa (em linguagens imperativas) executa de forma sequencial, seguindo por um fluxo que você pode determinar através de comandos como “if” e “while”, e outros que ainda veremos.

Através de variáveis, podemos guardar e recuperar informações, que mudam ao longo do programa. No caso de linguagens com tipos dinâmicos, o próprio tipo da informação armazenado nas variáveis pode mudar durante a execução.

Além disso, podemos utilizar funções que fazem parte da linguagem que estivermos programando, bem como funções que forem incluídas de bibliotecas de funções, e criarmos as nossas próprias.

A partir de agora, além de estruturas importantes, como objetos, e alguns comandos essenciais que ainda não vimos, veremos muito mais funções javascript, porém, até por não ser esta uma apostila da linguagem Javascript, mas sim de programação em si, vamos apenas utilizar e descrever as novas funções que formos usando, sem nos determos nas mesmas.

Veja o exemplo a seguir, que utiliza apenas estruturas e códigos que já vimos e algumas novas funções. Execute-o e observe o resultado:

Vamos exibir no console os 100 primeiros números primo:

```
----------NUMEROS PRIMOS----------

function validaPrimo(numero)

{

  var contador = 2;

 while (contador < (numero+1)/2) {

    if (numero % contador == 0)

      return false;

   contador++;

 }
```

```
  return true;
}
var contadorPrimos = 0;
var numeroAtual = 2;
while (contadorPrimos < 100) {
    if (validaPrimo(numeroAtual)) {
        console.log(numeroAtual);
        contadorPrimos++;
    }
    numeroAtual++;
}
```

O código acima tem um único comando que não vimos ainda: “console.log”. Trata-se de uma função do objeto console, e funções de objetos é um dos assuntos que iremos tratar nesta unidade.

O importante é que o código acima, embora mais complexo que os que já vimos, não tem – exceto por este detalhe – nenhuma novidade.

Veja se consegue entende-lo. E aqui o comando ‘console.log’ será muito útil. Você pode, por exemplo, inserir chamadas a ‘console.log’ em qualquer parte do código para entender melhor o funcionamento do programa.

Observe o programa a seguir, praticamente igual ao anterior, porém com várias informações colocadas no console:

```
----------NUMEROS PRIMOS----------
function validaPrimo(numero)
{
  console.log("Entrando na função validaPrimo para testar se " + numero + " é primo ");
  var contador = 2;
  while (contador < (numero+1)/2) {
```

```
        console.log("Testando se " + numero + " é divisível por " + contador);
        if (numero % contador == 0) {
            console.log("Descobrimos que " + numero + " é divisivel por " + contador + " e portanto não é primo ");
            return false;
         }
         contador++;
        }
        console.log("Fizemos todos os testes. " + numero + " é primo ");
        return true;
    }
    var contadorPrimos = 0;
    var numeroAtual = 2;
    while (contadorPrimos < 10) {
        console.log("------------------------------");
        console.log("vamos validar se " + numeroAtual + " é primo chamando a função validaPrimo");
        if (validaPrimo(numeroAtual)) {
            console.log(numeroAtual);
            contadorPrimos++;
            console.log("Este é nosso número primo de número  " + contadorPrimos);
        }
        numeroAtual++;
    }
    ----------------------------------------------------------------------------------------------
```

Números primos, vale recordar, são números inteiros divisíveis apenas por 1 e por eles mesmos.

Executando o programa acima, possivelmente você vai entender melhor o algoritmo, se não havia compreendido ainda. Na prática, programadores utilizam funções como “console.log” apenas quando estão em um teste para resolver um bug ou entender um código que outra pessoa fez.

A partir de agora, sempre que um programa for apresentado e você tiver dúvidas, inclua chamadas a console.log e observe o resultado exibido para entender melhor o fluxo de execução do programa.

Obs: existem mecanismos mais sofisticados para acompanhamento e teste de programas, mas há situações em que gerar um log é a forma mais prática ou a única viável para determinada situação

12. Objetos em Javascript

Orientação a objetos é algo muito, muito importante. E você precisa aprender a programar orientado a objetos. Precisa mesmo.

Só que não é o objetivo desta apostila ☹

Mas, mesmo assim, vamos ver um pouquinho de objetos em Javascript, por que eles são a forma como trabalhamos com tipos de dados estrturados na linguagem. Tipos de dados estruturados são, sim, objetivo desta apostila.

12.1 Você Não É Apenas um Número

Muitas vezes, o tipo de informação que você está trabalhando poderia ser melhor expressado se você pudesse colocar mais informações que apenas um inteiro ou uma string em uma variável.

Vamos falar de alguns exemplos em que esta questão fica mais clara:

Coordenadas cartesianas: Você quer representar pontos no espaço através de coordenadas x, y e z. Embora você possa criar, por exemplo, três variáveis para cada ponto, por exemplo “ponto1x”, ”ponto1y”, seria muito mais conveniente poder colocar todas as coordenadas em uma única variável: ponto1.x, ponto1.y, ponto1.z. Neste formato, só existe uma variável, “ponto1”, mas ela aceita três números, vinculados as coordenadas x, y e z.

Dados de pessoas em um cadastro: Imagine que você tenha um cadastro de clientes. Ao invés de ter que criar funções passando sempre todos os dados de cada cliente de forma separada, nome, idade, cpf, etc., seria muito mais conveniente colocar tudo em uma variável, como “cliente.nome”, “cliente.idade”, “cliente.cpf”. Neste caso, a única variável que você está trabalhando é “cliente”, porém ela armazena diversas informações.

12.2 Criando Variáveis Estruturadas

Em diversas linguagens você irá encontrar formas de construir uma estrutura de informações que são vinculadas a uma variável. Vamos apresentar aqui a forma como é feito em Javascript, com alguns exemplos

---

```
var ponto3d = {x:0, y: 10, z: 10};

ponto3d.x += 30;
```

```
console.log(ponto3d);
```

```
var pessoa = {nome: “José da Silva”, idade: 30, cpf: “999.999.999-99”}

alert(pessoa.nome);

pessoa.idade += 1;

alert(“A idade agora é “ + pessoa.idade);

alert(typeof pessoa);
```

Como pode ser observado no segundo, dos exemplos, o tipo destas variáveis, em Javascript, é “object”.

12.3 Podemos Vincular Funções a um Objeto

Observe o exemplo a seguir:

```
var pessoa = {
    nome: "João",
    sobrenome: "da silva",
    nomeCompleto: function() {
        return this.nome + " " + this.sobrenome;
    }
}

alert(pessoa.nomeCompleto());
```

Não vamos nos aprofundar muito neste exemplo, para não tornar esta uma apostila de orientação a objetos, ou de linguagem javascript.

Para este momento, basta observarmos que, além de valores diretamente definidos, como no caso de “nome” e “sobrenome”, podemos também ter uma função, como “nomeCompleto”.

Nós já utilizamos este recurso anteriormente, ao chamarmos “console.log”. O que fizemos, na prática, foi chamar a função “log” de “console”.

Esta não é a única forma de vincular uma função a uma variável, em Javascript, e o exemplo está aqui apenas para você começar a entender que uma função também pode ser vinculada a uma variável. Ao se aprofundar em Javascript, você descobrira que pode enviar uma função como parâmetro de outra função, e utilizar uma série de recursos mais sofisticados.

13. Arrays e o Comando FOR

13.1 Arrays, Não Podemos Viver sem Eles

Vamos começar a falar de arrays com um desafio. Estão lembrados do desafio de inverter 5 números, de uma de nossas primeiras unidades? Se não estão, aqui vai ele novamente:

```
----------INVERSÃO DE 5 NUMEROS----------

    var n1, n2, n3, n4, n5;

    var totalNumeros = prompt("entre com um número de 2 a 5");

    n1 = prompt("entre com o primeiro número");

    n2 = prompt("entre com o segundo número");

    if (totalNumeros > 2) n3 = prompt("entre com o terceiro número");

    if (totalNumeros > 3) n4 = prompt("entre com o quarto número");

    if (totalNumeros > 4) n5 = prompt("entre com o quinto número");

    if (totalNumeros > 4) alert(n5);

    if (totalNumeros > 3) alert(n4);

    if (totalNumeros > 2) alert(n3);

    alert(n2);

    alert(n1);

----------
```

Pois bem, vamos agora sofisticar este desafio colocando a seguinte questão: e se não houvesse limite para o número de entradas que vamos receber para inverter?

Na verdade, o exemplo acima não é a forma mais adequada de implementarmos um programa que inverte números, era apenas uma forma razoável com os comandos que conhecemos até agora.

Vamos pensar um pouco sobre este problema. Sem sabermos que teremos exatamente 5 números, o que precisaremos é alguma forma de armazenar 'n' inteiros, que podem ser 5, como podem ser mais ou menos. Nós queremos, por exemplo, ter

alguma forma de guardar os vários inteiros em algum tipo de ‘n1,n2,n3,n...,até nTOTAL’.

Linguagens de programação costumam resolver isso com um tipo de dado que ainda não havíamos visto, o array. Vamos apresentar uma solução mais adequada e sem limites para o problema da inversão, mas primeiro trataremos de um novo comando.

13.2 While Não É Tudo Aquilo

Até agora, o comando “while” deve ser um dos mais importantes comandos que você aprendeu, por um motivo simples: ele é o único comando de loop que mostramos até agora. Ou seja, sem o “while”, você só consegue criar programas que executam uma vez e terminam (na verdade, existe uma outra forma de fazer loops apenas com o que já aprendemos, que é utilizando funções recursivas, mas a complexidade de desenvolver códigos recursivos me fez decidir não incluir tal assunto nessa apostila).

Isso está prestes a mudar com a apresentação do comando de loop favorito de 9 entre 10 programadores, o “for”.

O motivo do “for” ser tão popular, apesar de ele ser um pouco mais difícil de dominar seu uso que o “while” é que ele faz em uma só linha e de forma bastante padronizada e organizada, as três funções mais comuns de um loop.

Vamos pensar, por um momento, que queremos executar uma sequência de ações por um determinado número de vezes, digamos 10 vezes. Observe como um comando típico com while funcionaria:

```
---------------------------------------------------------------------------------

var i = 0;

while (i < 10) {

    console.log(i); /// aqui teríamos todo o código do loop

    i++;

}

---------------------------------------------------------------------------------
```

Não há nada de errado com este código. O problema é justamente que ele é extremamente comum, e qualquer programador experiente já fez comandos parecidos milhares de vezes.

A grande questão é que temos basicamente as mesmas três ações que sempre se repetem. A inicialização da variável de contole (“var i = 0”), o código que testa a

variável de controle (“i < 10”) e o código que incrementa a variável (“i++”). Embora pareça muito fácil ver este código e identificar os elementos, imagine um código extremamente complexo, com outras váriaveis sendo declaradas, “while” dentro de “while”, e um longo código dentro do loop, até chegarmos ao comando de incremento (“i++”).

Programadores experientes sabem que em um comando deste tipo existe sempre o risco de esquecer de incrementar a variável, ou confundir uma variável de controle com alguma outra variável dentro do código, etc.

Esta é a principal razão da popularidade do comando “for”.

Veja, a seguir, como o mesmo código assim se apresenta com o comando:

```
for (var i = 0; i < 10; i++) {
    console.log(i); /// aqui teríamos todo o código do loop
}
```

Basicamente é a mesma coisa, apenas escrito de uma forma mais conveniente, em que toda a parte de controle do loop está na mesma linha, tornando tudo mais legível e rápido de escrever, e menos sujeito a erro.

O comando “for”, então, tem três partes. A inicialização das variáveis de controle, o teste e o incremento.

Obs: você pode utilizar o comando for de outras formas, por exemplo, colocando códigos de dentro do loop junto com o comando “i<10”, ou inicializando outras variáveis além das de controle. Existe, porém, um universo de diferenças entre o que se pode fazer, e o que se deve fazer. Não faça, utilize o comando “for” apenas na forma normal de seu uso, colocando como parâmetro as variáveis que controlam o loop.

A partir de agora, sempre que tivermos um loop que faça sentido expressar por um comando “for”, faremos uso do mesmo. Ele será particularmente útil para navegarmos por arrays.

13.3 Vamos Criar nosso Primeiro Array

Vamos criar um array de inteiros, já preenchido com dados, e vamos imprimir no console, do primeiro ao último elemento, entre vírgulas, e depois de forma invertida.

```
var meuPrimeiroArray = [1,2,3,4,5,6,7,8,9,10];

var ordemCerta = "";

var ordemErrada = "";

for ( var i = 0; i < 10; i++) {

    ordemCerta += meuPrimeiroArray[i] + ",";

}

for ( var i = 9; i >= 0; i--) {

    ordemErrada += meuPrimeiroArray[i] + ",";

}

console.log(ordemCerta);

console.log(ordemErrada);
```

Apesar de pequeno, este código apresenta várias questões interessantes, e vamos estuda-lo com atenção:

Conversão automática de número em string: talvez você nem tenha percebido, mas tivemos aqui uma conversão automática de 'number' em 'string'. Assim como, com os operadores '-', '*' e '/' vimos situações em que uma string era convertida automaticamente para número, aqui tivemos o oposto. Como um dos elementos da operação, a virgula, era uma string, o código converteu automaticamente o número inteiro também para string, para efetuar uma concatenação.

Obs:é discutível se esta prática é recomendada. Uma alternativa seria utilizar o comando toString, que veremos na sequência, forçando a conversão do inteiro em string.

Criando e atribuindo valores a um array: observe o comando utilizado na primeira linha. Ele ao mesmo tempo criou o array de 10 posições "meuPrimeiroArray" e atribui valores inteiros a cada posição do array.

As Posições do Array Iniciam em 0: a maioria das linguagens trabalha com o conceito de que a primeira posição de um array é definida como 0. Assim, no exemplo acima, colocamos o valor '1' na primeira posição do array, e para recuperar este valor, temos que utilizar o código 'meuPrimeiroArray[0]'.

13.4 Podemos Também Incluir Novos Elemntos em um Array

Aqui, antes mesmo de começarmos, uma observação é importante: javascript é uma linguagem que automatiza uma série de questões envolvendo o uso da memoria do computador. Em javascript, é muito simples incluir novos elementos em um array, sem que precisemos nos preocupar em reservar um espaço na memoria do computador para os mesmos. Em outras linguagens, como C, isso é mais complexo. É importante você entender exatamente como utilizar "arrays" e quais os recursos dos mesmo na linguagem em que você estiver programando.

Dito isso, observe como incluir novos elementos em um array em Javascript:

```
--------------------------------------------------------------------------

var meuPrimeiroArray = [1,2,3]; // array com 3 elementos, nas posições 0, 1 e 2

meuPrimeiroArray[3] = 4; // posição 3 recebeu um valor. Array tem 4 posições

meuPrimeiroArray[4] = 5; // posição 3 recebeu um valor. Array tem 5 posições

var ordemCerta = "";

for ( var i = 0; i < 5; i++) {

    ordemCerta += meuPrimeiroArray[i] + ",";

}

--------------------------------------------------------------------------
```

Mas como fazemos, se quisermos que nosso array comece vazio? Nada mais simples:

```
--------------------------------------------------------------------------

var meuPrimeiroArray = []; // array inicia vazio

meuPrimeiroArray[0] = 1; // posição 0 recebeu um valor. Array tem 1 posição

meuPrimeiroArray[1] = 2; // posição 1 recebeu um valor. Array tem 2 posições

var ordemCerta = "";

for ( var i = 0; i < 2; i++) {

    ordemCerta += meuPrimeiroArray[i] + ",";

}
```

Talvez já tenha ficado óbvio para você que existe uma questão que está nos atrapalhando nestes programas, que é sempre temos que mudar o teste no “for”. E mesmo isso só é possível porque sabemos a quantidade de elementos no array, mas e se dependesse de uma entrada do usário? Vamos resolver também esta questão:

```
var meuPrimeiroArray = []; // array inicia vazio

meuPrimeiroArray[0] = 1; // posição 0 recebeu um valor. Array tem 1 posição

meuPrimeiroArray[1] = 2; // posição 1 recebeu um valor. Array tem 2 posições

var ordemCerta = "";

for ( var i = 0; i < meuPrimeiroArray.length; i++) {

    ordemCerta += meuPrimeiroArray[i] + ",";

}
```

13.5 Agora é com Você

Como primeiro desafio com arrays, faça o mesmo programa que mostramos no inicio, com um array de 10 elementos e a exibição em ordem normal e inversa, já utilizando o tamanho (‘length’) do array. Entretanto, corrija o bug que faz com que após o último número apareça uma vírgula. Ou seja, queremos retornos como “1,2,3,4,5,6,7,8,9,10” e não “1,2,3,4,5,6,7,8,9,10,”.

13.6 Vamos Inverter a Entrada Sem Limite

Faça um algoritmo que receba quantos números o usuário inserir, até ele colocar “fim”. Neste momento, mostre um alerta com todos os números colocados, separados por vírgula e na ordem inversa.

13.7 Crie uma função que retorne o array de entrada invertido

Bastante semelhante ao exercício anterior, mas queremos que você faça a inversão do array em uma função, e retorna um novo array com o resultado.

13.8 Resposta dos Desafios

```
--------------------------------------------------------------------------

var meuPrimeiroArray = [1,2,3,4,5,6,7,8,9,10];

var ordemCerta = "";

var ordemErrada = "";

ordemCerta = meuPrimeiroArray[0];

for ( var i = 1; i < meuPrimeiroArray.length; i++) {

    ordemCerta += "," + meuPrimeiroArray[i];

}

ordemErrada = meuPrimeiroArray[meuPrimeiroArray.length-1];

for ( var i = meuPrimeiroArray.length-2; i >= 0; i--) {

    ordemErrada += "," + meuPrimeiroArray[i];

}

console.log(ordemCerta);

console.log(ordemErrada);

--------------------------------------------------------------------------
```

Esta é uma das soluções mais eficientes para a questão de evitar a virgula no final. Da mesma forma pode-se fazer o loop até o penúltimo item, e adicionar o último após o loop.

Uma terceira alternativa é fazer um if dentro do loop para testar se já estamos na última iteração, para, neste caso, não incluir a vírgula. Esta alternativa é mais cara computacionalmente.

Seguindo para o segundo desafio:

```
--------------------------------------------------------------------------

var entrada = "";
```

```
var meuArray = [];

while (entrada != "fim") {

	entrada = prompt("entre com valor, ou 'fim' para encerrar");

	if (entrada != "fim") {

		meuArray[meuArray.length] = entrada;

	}

}

ordemErrada = meuArray[meuArray.length-1];

for ( var i = meuArray.length-2; i >= 0; i--) {

	ordemErrada += "," + meuArray[i];

}

alert(ordemErrada);
```

Observe que o comando “meuArray[meuArray.length] = entrada” basicamente é uma inserção de um novo elemento no final do array. O Javascript, assim como várias outras linguagens, tem um comando que permite inserir elementos no final de um array de forma mais direta: “meuArray.push(entrada);”. Experimente substituir um comando pelo outro para confirmar que o resultado permanece igual.

E, por fim, isolando nossa inversão de array em uma função e criando um novo array invertido:

```
function inverteArray(arrayEntrada)

{

	arraySaida = [];

	for ( var i = meuArray.length-1; i >= 0; i--) {

	arraySaida.push(meuArray[i]);

	}
```

```
        return arraySaida;
}
var entrada = "";
var meuArray = [];
while (entrada != "fim") {
        entrada = prompt("entre com valor, ou 'fim' para encerrar");
        if (entrada != "fim") {
                meuArray.push(entrada);
        }
}
var arrayInvertido = inverteArray(meuArray);
console.log(arrayInvertido);
```

---

14. E a Partir de Agora?

O que mostramos nesta apostila foi efetivamente uma introdução a programação. Há sem dúvida pelo menos duas grandes questões que não endereçamos aqui, e que são essenciais para um desenvolvimento profissional, mas também são continuações naturais do seu estudo:

14.1 Desenvolvimento Orientado a Objetos

Nós poderíamos apresentar, seguindo o mesmo padrão adotado aqui, exemplos gradativamente mais sofisticados de programação orientada a objetos. Há um pequeno empecilho de Javascript implementar Orientação a Objetos de uma forma diferente da maioria das linguagens imperativas.

De todo modo, possivelmente faria mais sentido criarmos uma apostila em separado para programação orientada a objetos. Algo para avaliarmos com base no sucesso deste material.

14.2 Funções e Recursos Específicos de Javascript

A partir de agora, para avançar para programas mais sofisticados e úteis em Javascript, é inevitável um uso mais amplo tanto das funções da linguagem quanto de recursos específicos que o Javascript tem e que o diferenciam de outras linguagens.

Entretanto, estas questões contrastam com o objetivo de ensinar conceitos de programação que possam ser facilmente aplicados em qualquer linguagem imperativa. Aqui realmente se faz necessário aprofundar tais questões em uma apostila específica de Javascript.

De todo modo, se tiveres interesse em dar andamento aos estudos na linguagem, não deixe de visitar o site da Alfamídia, onde disponibilizamos alguns cursos gratuitos de programação Javascript, que já assume o conhecimento justamente apresentado neste material.

14.3 Mas Algo Mais Importante Ainda Falta

Praticar.

Mais que qualquer recurso ou função que pudesse ser mostrado nesta apostila, o que mais ainda faz falta para você se tornar um programador, é praticar.

Não se preocupe em apenas criar programas por conta, ou realizar desafios. Sinta-se livre também para estudar códigos disponíveis na Internet. Entenda como cada

código foi construído, tente ver programas que usem ideias, construções, que você não tinha ainda visto ou pensado.

A medida que você praticar mais e mais, comecará a ficar mais prático criar seus próprios algoritmos.

Como mensagem final deste material, recomendo apenas o seguinte:

Pense em programar como falar em uma linguagem, como inglês ou português. Pratique de todas as formas que puder. Você não precisa conseguir criar programas em um primeiro momento, e se não conseguir não se preocupe com isso. Estude programas existentes, altere-os, teste, tente entende-los. Pense que você está se exercitando, e assim desenvolvendo sua capacidade de programação.