
JS
javascript
na pratica

ÍNDICE

Módulo 1: Programação

Aula 1: Variáveis (Páginas 4 a 6)
Aula 2: Operadores Aritméticos (Páginas 7 a 9)
Aula 3: Operadores Condicionais (Páginas 10 e 11)
Aula 4: IF e ELSE (Páginas 12 e 13)
Aula 5: Lógica Booleana (Páginas 14 e 15)
Aula 6: Funções (Páginas 16 e 17)
Aula 7: Objetos (Páginas 18 a 20)
Aula 8: Métodos (Páginas 21 e 22)
Aula 9: While (Página 23)
Aula 10: Continue e break (Página 24)
Aula 11: FOR (Página 25)
Aula 12: Arrays (Páginas 26 e 27)
Aula 13: Operações com Arrays (Páginas 28 e 29)
Aula 14: Arrays Bidimensionais (Páginas 30 e 31)

Módulo 2: Snake

Páginas 32 a 36

Módulo 3: Flappy Bird

Páginas 37 a 40

Módulo 4: Codi Memory

Páginas 41 a 46

Módulo 5: Dino

Páginas 47 a 52

INTRODUÇÃO AO CURSO

JavaScript é uma linguagem de programação muito usada no desenvolvimento de aplicativos para a Web. O JavaScript não apareceu derrepente, tudo começou com a criação, pela Netscape, de uma linguagem de criação de Scripts Server-side, esta linguagem foi implantada nos servidores de WEB da Netscape.

Quando a Microsoft viu que o sistema Server-side tinha futuro, criou uma linguagem, JScript, e implantou o sistema nos servidores de WEB ISS da empresa. A Netscape, indignada pela cópia de seu sistema pela Microsoft, entrou em um novo projeto. Deixando o sistema Server-side de lado, a empresa passou a desenvolver um sistema "client-side", que roda no navegador do usuário. Este novo sistema que estava surgindo permitia aos usuários processarem os scripts diretamente, ao invés de usar o servidor. Este sistema permitiu uma grande melhoria na velocidade de processamento dos dados, e colocou a Netscape no topo novamente.

A Microsoft, que estava perdendo mercado para a Netscape, copiou mais uma vez o sistema da concorrente, implantando o Client-side em seu navegador, o Internet Explorer. Este navegador veio junto com o Windows 95. O JavaScript teve vários nomes. O primeiro foi Mocha, o segundo foi LiveScript, e quando a Netscape passou a ter suporte á tecnologia Java em seu navegador, mudou o nome para JavaScript, como um jogo de marketing, para popularizar o script.

Uma confusão comum, que ocorre ate mesmo entre o pessoal da área de informática, é achar que o JavaScript tem relação com o Java. O JavaScript não foi baseado nem é ligado ao Java em seu método de criação.

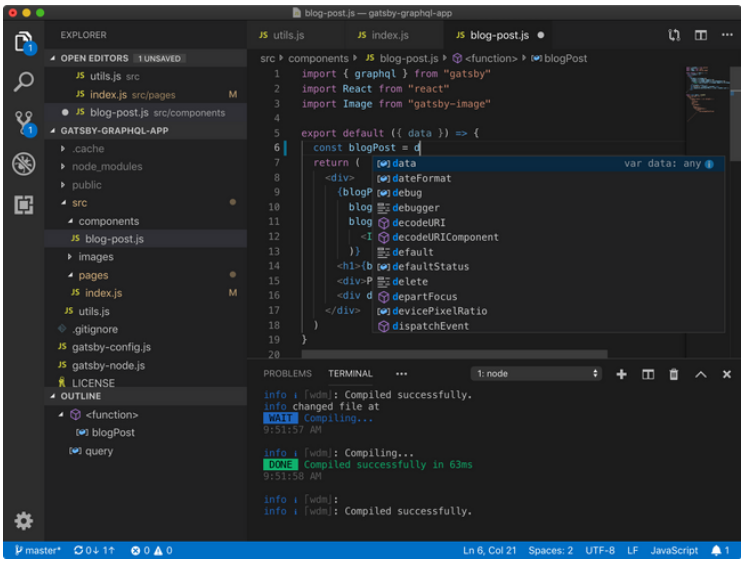
Tanto o JScript (da Microsoft) quando o JavaScript (Netscape) só podiam ser usados nos navegadores das empresas que os criaram. A Netscape encaminhou o JavaScript para a empresa ECMA, para que fosse feita uma padronização da linguagem, permitindo assim seu funcionamento na maioria dos navegadores.

A linguagem padronizada passou a se chamar ECMAscript, e tem este nome até hoje. Embora o nome tenha mudado, quase ninguém usa o nome ECMAscript, o nome Javascript é bem mais conhecido e usado no mundo todo, o Javascript é hoje a linguagem mais popular do mundo.
O grande diferencial do JavaScript é que este permite o desenvolvimento dos códigos dentro do código HTML. O programador esta fazendo o site,e basta colocar o código " <script>" e iniciar a programação em Javascript, permitindo também códigos em HTML dentro do código de JavaScript. Para finalizar a programação em JavaScript, basta digitar "</script>". Para programar em JavaScript é necessário saber o básico de HTML e ter um editor de textos disponível.

E é exatamente isso que precisamos fazer agora, baixar um programa para editar os nossos códigos, existem 2 opções:

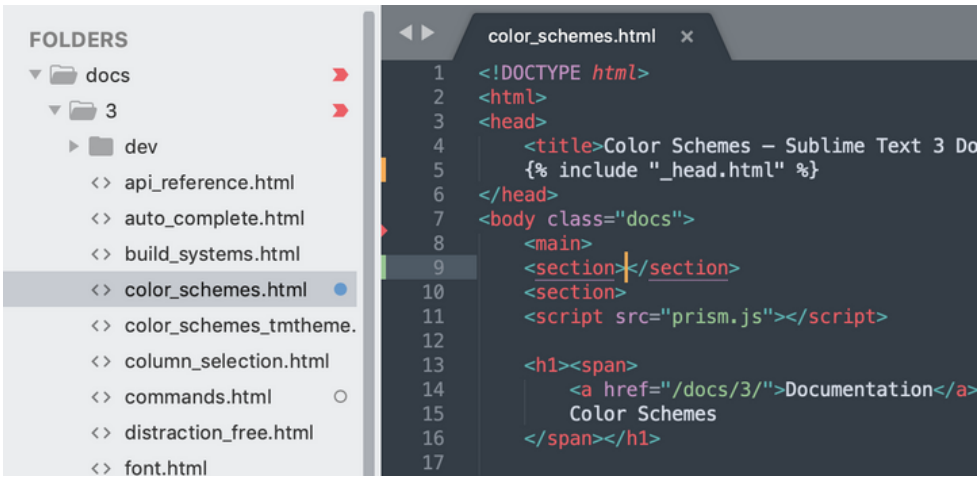
VISUAL STUDIO CODE

IDE que suporta diversas linguagens, dá um amplo suporte a erros e possui ferramentas profissionais.



SUBLIME TEXT

Editor simples de textos, é como um bloco de notas colorido.



Eu deixo nas suas mãos a escolha de qual programa baixar, ambos tem a instalação super simples, basta baixar e usar.
Aqui no curso eu vou utilizar o Sublime Text e recomendo que você faça essa escolha, pois se você fizer algo errado o Sublime não avisa onde é você terá que pesquisar, ler o código e descobrir você mesmo onde errou, o que faz com que você aprende mais e preste mais atenção ao código, mas a escolha é sua.

Para encerrar essa primeira aula, vamos exibir uma mensagem na tela. O clássico "Hello world" dá sorte a programadores iniciantes.

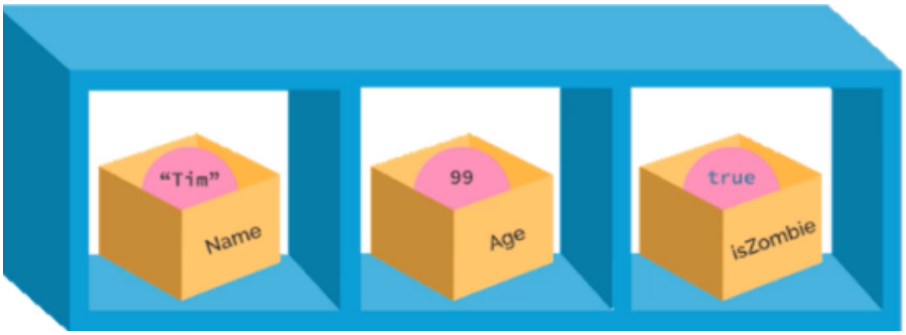
```
<script>
    alert('Hello world')
</script>
```

Tudo certo, agora vamos aprender cada aspecto dessa linguagem.

AULA 1: VARIÁVEIS

Variáveis são uma construção comum encontrada na maioria das linguagens de programação.
Elas permitem que você armazene e recupere dados da memória do seu computador. Para usar isso, você fornece às variáveis um nome e atribui a elas um valor. Para recuperar o valor salvo ou alterá-lo, basta saber o nome da variável. A maneira como você atribui um valor a uma variável difere com base no tipo de dados que deseja atribuir.
Então a gente vê aqui na tela a representação das variáveis como uma caixa de armazena dados, repare que diferentes tipos de dados estão sendo armazenados.

O primeiro é um texto, o segundo um número e o terceiro um valor booleano.



As variáveis em linguagens de programação geralmente têm tipos de dados internos . Em JavaScript, nós teremos quatro tipos principais de dados:

TIPOS DE DADOS

Number: Representa qualquer número, incluindo inteiros, decimais e números positivos e negativos.
Boolean: Representa um valor lógico que é true ou false.
String: Representa dados textuais; que é qualquer valor entre aspas.
Undefined: Criamos um nome da variável sem atribuir um valor a ela

É importante observar que as variáveis nas linguagens de programação podem ser estáticas ou dinamicas, JavaScript é uma linguagem dinâmica. Isso significa que as variáveis não estão diretamente associadas a nenhum tipo de dados específico, portanto, podem ser atribuídos (e reatribuídos) valores de todos os tipos.
Para maior clareza do código, é recomendável não alterar o tipo de dados de uma variável, a menos que haja um motivo muito específico para fazer isso.

REGRAS DE NOMENCLATURA DE VARIÁVEIS

Existem algumas regras em JavaScript em relação à nomenclatura de variáveis, essas regras basicamente evitam o caos , dado que cada programador tem suas manias então é bom estabelecer um teto.

- Os nomes devem começar com uma letra ou o sinal de sublinhado (_)
- Os nomes podem conter letras, números (não no começo) e o sinal de sublinhado (_)
- Os nomes não podem ser palavras-chave reservadas para JavaScript ; por exemplo, let

DECLARAÇÃO DE VARIÁVEL E ATRIBUIÇÃO

Quando você nomeia uma variável pela primeira vez, ela é chamada de declaração de variável e quando você atribui um valor à variável, é chamada de atribuição de variável. A declaração e a atribuição podem acontecer juntas ou separadamente. Para declarar uma variável, usaremos a palavra-chave **let**, mas também podemos usar **var**. Além disso, observe que terminamos cada instrução JavaScript com um ponto e vírgula (;). O uso do ponto-e-vírgula não é obrigatório em JavaScript, mas é recomendado.
Para adicionar comentários ao código JavaScript, usamos // e para exibir informações no console, usamos a função **console.log()**.

Agora vamos criar algumas variáveis e mostrar seu valor no arquivo aula1.html.

```
<script>
  // declaração de variável
  let player;                    // tipo indefinido
  // atribuição de valor a variável
  player = 'Mario';              // player é uma String
  // Mostrando o valor no console
  console.log(player);

  // Variável númerica
  let pontos = 0;
  console.log(pontos);

  // Variável booleana
  let tavivo = true;
  console.log(tavivo);

  // Outras coisas
  let valor = 50.5;
  console.log(valor);
  // declaração ruim
  aleatoria = 'Alguma coisa';
  console.log(aleatoria);
  // indefinição
  let algo;
  console.log(algo);
</script>
```

Agora um desafio pra você:
Exercício 1: Crie uma variável, guarde dela o seu nome e mostre o seu valor na tela.

AULA 2: OPERADORES ARITMÉTICOS

Operadores ariméticos são aqueles que você aprendeu com a tia lá do ensino fundamental, somado a um operador novo, vejamos:

Adição (+):
Exemplo 10+4 = 14

Subtração (-):
Exemplo 10-4 = 6

Multiplicação (*):
Exemplo: 10*4 = 40

Divisão (/):
Exemplo 40/10 = 4

Resto da divisão (%): Resto que sobra de uma divisão não exata
Exemplo: 10%2 = 0 (divisão exata)
Exemplo 2: 10%3 = 1 (divisão não exata)

Agora vamos brincar um pouco com cada uma dessas operações:

ADIÇÃO

Crie o arquivo aula2.html

```
<script>

let a = 10 + 4;
console.log(a);

let b = 10;
let c = a + b;
console.log(c);

//c = c + 1;
c += 1;
console.log(c);

</script>
```

Acompanhe a saída de dados no console.
(Inspecionar elemento > Console).

SUBTRAÇÃO

```
<script>

let b = 20
let x = 10 - 4;
console.log(x);

let y = b + x;
console.log(y);

y = y - 1;
console.log(y);

</script>
```

MULTIPLICAÇÃO

Neste exemplo eu quero fazer uma questão de vestibular, mas é bem fácil.
Exemplo: Numa padaria o preço do biscoito é 2 reais, Maria quer comprar 10 unidades de biscoito, quanto ela vai pagar? Construa o problema e mostre a resposta em JavaScript.

```
<script>

let punitario = 2;
let unidades = 10;
let total = punitario * unidades;
console.log(total);

</script>
```

DIVISÃO

Exemplo: Numa escola, 53% dos 100 alunos são do sexo masculino. Qual o total de alunos do sexo masculino nessa escola? Construa o problema e mostre a resposta em JavaScript.

```
<script>

let n = 100;
let pct = 53
let total = (53/100)*100;
console.log(total);

</script>
```

RESTO DA DIVISÃO

Mostre o resto da divisão de 10 por 3:

```
<script>

let total = 10 % 3;
console.log(t);

</script>
```

Agora, um exercício pra você fazer antes da próxima aula.

Exercício 2: Um fazendeiro mediu sua terra, de formato retangular, para cercá-la inteiramente com uma cerca de madeira. Quantos metros de cerca ele deverá fazer para sua fazenda que possui 1600 metros de largura por 2789 metros de altura?.

AULA 3: OPERADORES CONDICIONAIS

Antes de conversarmos sobre as condicionais, vamos avançar mais um passo na execução de Javascript em nossa máquina, em vez de utilizarmos um único arquivo .html com a tag do Javascript dentro dela, vamos agora usar 2 arquivos por aula, um arquivo .js (Javascript) e um arquivo. html (HTML), e pra não virar uma bagunça eu aconselho que você crie uma pasta para cada aula no seu computador.
Lembre-se de que usamos o atributo **<script></script>** para identificar a referência JavaScript externa, o arquivo HTML servirá apenas para chamar o outro (.js), futuramente isso fará mais sentido, pois existem alguns recursos especiais em JavaScript que são carregados no arquivo .html para serem usados no arquivo .js.

Crie o arquivo "aula4.html":

```
<script src="script.js"></script>
```

Crie agora o arquivo "script.js" e salve na mesma pasta do "aula4.html".

OPERADORES CONDICIONAIS

Um operador de condicional compara seus operandos e retorna um valor booleano (verdadeiro ou falso) com base no resultado da comparação.

São eles:
== Operador de igualdade
!= Operador de desigualdade (diferente)
> Maior que
>= Maior ou igual
< Menor que
<= Menor ou igual

Agora vamos brincar um pouco em JavaScript:

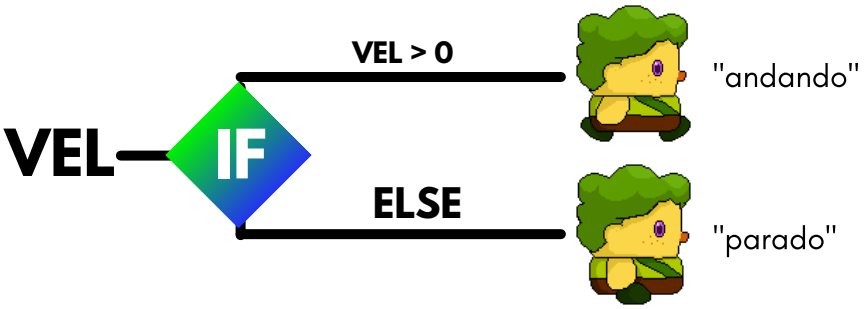
```
let a = 100
let b = 200
let c = a<b

alert(c)
```

```
let a = 100
let b = 200

let c = a<b
console.log(c)

let d = a == b
console.log(d)

let e = a != b
console.log(e)
```

Agora é com voce.
Exercício 3: Crie as variáveis a(valor=1), b(valor=2) e exiba no console o resultado de:
a) a>=b
b) a<=b
c) a!=b
d) a==b.

AULA 4: IF E ELSE

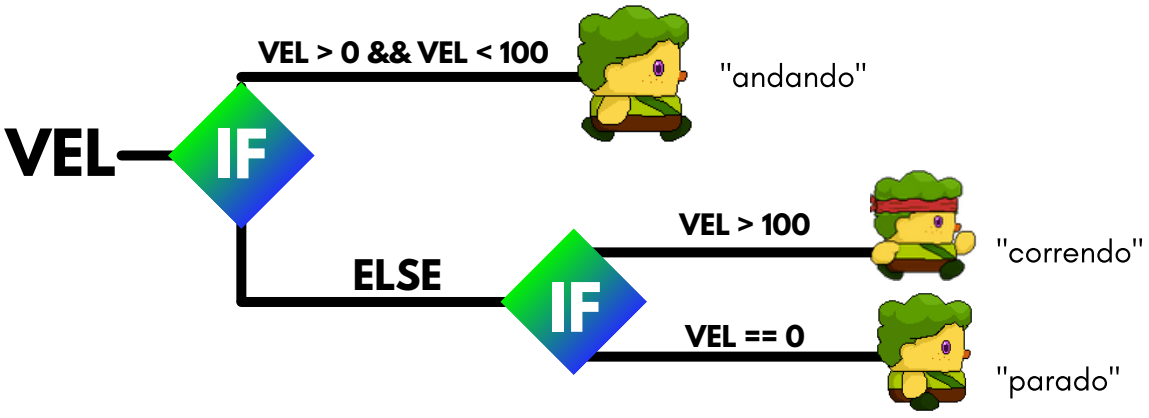
Essa condicional têm três possíveis caminhos: **if**(se), **else**(senão) e **else if**(senão se), este último é um caminho alternativo com uma segunda condição, vamos explorar todos eles.
A declaração IF pergunta "se" uma determinada condição foi cumprida, se foi cumprida o programa toma uma determinada ação.



Na imagem acima, estamos analisando a variável "vel", que controla a velocidade do Player, se vel é maior que zero, o player está andando, caso contrário ele estará parado (não há velocidade negativa aqui).
Agora, vamos criar isso em Javascript.

```
let vel
vel = 100

if(vel>0){
    alert("andando")
}
else{
    alert("parado")}
```

Repare que não tem problema esquecer o ";".
Agora vamos adicionar mais uma possibilidade:
Se a velocidade do player é maior que zero e menor que 100, ele está andando
Senão, se a velocidade do player é maior que 100, ele está correndo
Senão, se a velocidade do player é igual a 0, ele está parado.
Repare que ao explicar a situação, falamos "senão, se" o que indicar que teremos mais de uma condição depois do primeiro if, o nosso exemplo fica:



Atualizamos então o nosso código para:

```
let vel
vel = 0

if(vel>0 && vel<100){
    alert("andando")
}
else if(vel > 100){
    alert("correndo")
}
else if(vel == 0){
    alert("parado")
    }
```

Agora chegou a sua vez.

Exercício 4: Construa uma sistema para mostrar a nota de um aluno, são 3 situações possíveis:

1) nota >= 60 : "aprovado"

2) nota < 60 e maior que 40 > "pode fazer prova de recuperação"

3) nota < 40 "reprovado".

AULA 5: LÓGICA BOOLEANA

A lógica booleana pode nos ajudar a determinar os resultados em nossos jogos com base nos dados que temos e operações lógicas que fazemos com eles. Essa aula vai mesclar conceitos das duas aulas anteriores e consolidá-los.

! (NÃO)

Esse ´o símbolo da negação, podemos utilizá-lo para obter o inverso de algum dado, isso vale apenas para valores booleanos (true ou false).
Exemplo: Mostrar se um personagem é mágico ou não mágico:

```
let magico = !true

if(magico){
    alert("Magico!")
}
else{
    alert("Nao magico")
}
```

&& (E)

Na hora de criar uma condição, quando queremos que duas coisas aconteçam ao mesmo tempo usamos o &&.
Exemplo: Mostrar que um player que tem mais de 2 horas jogando e 1000 pontos deverá receber um bônus:

```
let pontos = 1000
let horasjogo = 3

if(pontos == 1000 && horasjogo > 2){
    alert("Toma bonus")
}
```

|| (OU)

Se você tem duas condições e quer que apenas uma delas aconteça, o "ou" é a conjunção mais indicada.
Exemplo: Se o player tem menos de 2 vidas ou menos de 1000 pontos, daremos a ele um bônus para mantê-lo no jogo:

```
let pontos = 1000
let vidas = 3

if(pontos < 1000 || vidas < 2){
    alert("Toma bonus")
}
```

Agora é com você:

Exercício 5: Crie um sistema que dará ao player diferentes tipos de chaves, o programa deverá emitir um alerta com qual chave o player ganhará:

Se ele tem 3 vidas e mais de 1000 pontos = chave azul

Se ele tem menos de 3 vidas ou menos de 1000 pontos = chave verde

Se ele tem 1000 pontos e não é magico = chave laranja

AULA 6: FUNÇÕES

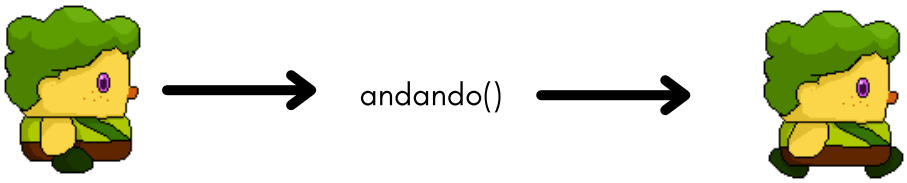
As funções são um bloco de construção fundamental de muitas linguagens de programação, incluindo JavaScript. Uma função em JavaScript é como uma receita no sentido de conter um conjunto de instruções, mas não as executa até que seja chamada. As funções permitem uma organização de código aprimorada e menos repetição, pois são blocos de código encapsulados que podem ser chamados quantas vezes forem necessárias. As funções podem ser escritas como uma declaração de função ou expressão de função .

PARÂMETROS E SAÍDAS DA FUNÇÃO

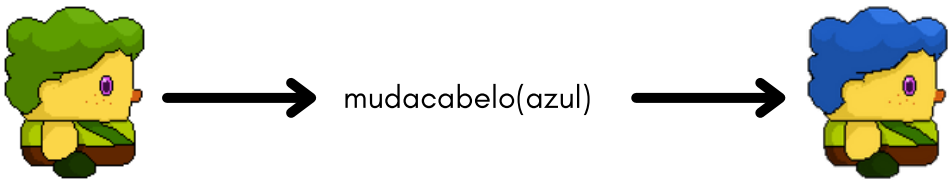
As funções aceitam opcionalmente uma ou mais entradas (chamadas de parâmetros) e, opcionalmente, retornam uma saída. Eles costumam ser mais poderosos quando retornam uma saída com base na entrada.
Assim como na matemática, sua principal utilidade é receber um dado e transformar em outro.

Exemplo de função matemática: f(x) = 2x
Se a função recebe a entrada 2, retornará 4 pois:
f(2) = 2*2 = 4

Na programação, podemos ter por exemplo uma função que faça o nosso player andar:



A função recebe a velocidade do player em 0 e retorna a velocidade > 0, fazendo com que o player possa andar.
E se tivéssemos uma função para mudar a cor do cabelo do player?.



Repare que nesta função, especificamos a cor ao chamar a função, e o resultado dessa função é o player com a nova cor de cabelo.

DECLARAÇÃO DE FUNÇÃO

Para declarar (criar) uma função em Javascript, usamos a seguinte nomenclatura:

```
function nomedafuncao(parametro1,parametro2) {
  corpo da função
  return valorderetorno;
}
```

Agora vamos criar algumas funções:

Exemplo 1: Uma função que converte minutos em horas:

```
// criação da função
function htm(horas) {
  let resultado = horas * 60;
  alert(resultado);
  return result;
}

// chamada da função
let a = htm(10);
```

Exemplo 2: Uma função que retorna o estado atual do player:

```
// criação da função
function estado(vel) {
  if (vel == 0){
    alert("parado")
  }
  else if(vel>0 && vel<100){
    alert("andando")
  }
  else if(vel > 100){
    alert("correndo")
  }
}

// chamada da função
let vel = 0
estado(vel)
```

Exemplo 3: Uma função que recebe o tempo de jogo do player e seu número de vidas, e verifica se ele tem direito a algum bônus:

```
// criação da função
function recebebonus(tempo,vidas) {
  if(tempo > 1000 && vidas < 2){
    alert("Recebe bonus")
  }
}

// chamada da função
recebebonus(2000,1)
```

Agora é com você:

Exercício 6: Implemente uma função que vai controlar a loja de um jogo, ela recebe o dinheiro que o player possui (número), e o item que o player quer comprar (string), se o player puder fazer a compra, a quantidade daquele item recebe +1, caso contrário o programa deverá retornar "moedas não suficientes para efetuar a compra", são 3 itens disponíveis: (A função deverá ser executada no console e não no código).

- Espada (100 moedas), Escudo (200 moedas), Adaga (300 moedas)

AULA 7: OBJETOS

Objetos são estruturas de dados que permitem armazenar várias propriedades e entidades mais complexas em uma única variável. Pense em um objeto como uma coleção de propriedades que possuem um nome e um valor.

DECLARAÇÃO DE OBJETO

Imagine que você está criando um jogo e precisa armazenar informações sobre o seu jogador. Você pode considerar todos os detalhes e armazená-los como variáveis separadas.

- Nome
- Pontuação
- Vidas

No entanto, o JavaScript fornece o Tipo de dados **object** como a melhor maneira de armazenar esse tipo de dados relacionados. Podemos representar o jogador declarando um objeto . Para criar um objeto, basta usar as chaves ({...}) e adicionar um ponto-e-vírgula no final.

```
let player = {};

console.log(player);
```

Para criar propriedades, simplesmente vá dentro das chaves e adicione um nome de propriedade seguido por dois pontos, espaço e valor da propriedade.

```
let player = {
    vidas: 3
};

console.log(player);
```

Para adicionar mais propriedades, basta usar uma vírgula para separá-las. É prática comum colocar cada par de nome/valor de propriedade em sua própria linha. Vamos inserir mais informações do jogador que identificamos acima e observar a saída impressa no console.

```
let player = {
    vidas: 3 ,
    nome: 'Codi',
    pontos: 99,
    vivo: true,
    cordaroupa: 'green',
    tamanho: 'M'
};

console.log(player);
```

Observe que existem várias propriedades que queremos manter associadas à roupa do jogador.O JavaScript nos permite agrupar informações associadas dentro de um objeto usando um objeto aninhado .

OBJETOS ANINHADOS

Quando você tem objetos que contêm outros objetos como propriedades, são chamados de objetos aninhados . A roupa do jogador é um caso de uso perfeito para esta opção. Vamos refatorar o objeto do jogador para incluir a roupa como um objeto aninhado com propriedades e valores adicionais.

```
let player = {
    vidas: 3,
    nome: 'Codi',
    pontos: 99,
    vivo: true,
    roupa: {
        cor: 'green',
        tamanho: 'M',
        custo: 100,
        nova: true
    }
};

console.log(player);
```

Na opção Console no navegador você consegue ver "roupa" como um novo objeto:

```
▼{vidas: 3, nome: "Codi", pontos: 99, vivo: true, roupa: {…}}
   nome: "Codi"
   pontos: 99
  ▶roupa: {cor: "green", tamanho: "M", custo: 100, nova: true}
   vidas: 3
   vivo: true
```

ACESSANDO PROPRIEDADES DO OBJETO

Para acessar as propriedades depois de declarar o objeto, você pode simplesmente digitar o nome do objeto, ponto e o nome da propriedade. Podemos demonstrar isso registrando no console qualquer propriedade do objeto jogador .

```
console.log(player['nome']);
console.log(player.nome);

console.log(player.roupa.cor);
console.log(player['roupa']['cor']);
```

MODIFICAR PROPRIEDADES DO OBJETO

Quando se trata de modificar as propriedades de um objeto, seguimos a mesma abordagem que fizemos para acessar as propriedades.
Novamente, você pode usar a sintaxe de ponto ou colchete, é recomendável usar sempre a de ponto, conforme a imagem:

```
player.vidas = 4;
console.log(player['nome']);
console.log(player.nome);

player.roupa.cor = 'azul';
console.log(player.roupa.cor);
console.log(player['roupa']['cor']);
```

ADICIONANDO PROPRIEDADES DO OBJETO

Quando se trata de adicionar propriedades de objeto, seguimos a mesma abordagem que fizemos para modificar propriedades.
Crie um nova propriedade para mostrar se o player tem ou não uma armadura:

```
player.armadura = true
console.log(player);
```

EXCLUINDO PROPRIEDADES DO OBJETO

Para fins de integralidade, abordaremos como excluir uma propriedade, mas isso não é feito com frequência e sempre deve ser feito com muito cuidado. Quando se trata de excluir propriedades do objeto, seguimos a mesma abordagem que fizemos para modificar as propriedades.
Exclua a propriedade que colocamos acima:

```
delete player.armadura
console.log(player);
```

Agora é com você:
Exercício 7: Sua missão é criar um game do tipo RPG, para isso você precisa configurar o player com os seguintes atributos: nome, vidas, pontos, emagico (booleano), espada (booleano) e magia (número).
Este player vai ter um objeto chamado mapa dentro de si, com os atributos: nome, cidades, cidadesvisitadas, objetosencontrados.
Em seguida, crie uma nova propriedade para o player chamada temcalice (booleano), e delete-a.

AULA 8: MÉTODOS

Um método é uma função que vive dentro de um objeto da mesma forma que outras propriedades estão. Em vez de manter um valor como String, Number, Boolean ou Objeto aninhado, teremos aqui uma Função como valor. Como os métodos vivem dentro de um objeto, devemos usar uma palavra-chave de contexto JavaScript especial (**this**) dentro da função para acessar outras propriedades do objeto.
Nesta aula, usaremos o exemplo de criação de um jogo de Pet Virtual para demonstrar os conceitos de declaração, execução e contexto de Método. Nosso jogo exigirá certas informações (Atributos) a serem armazenados e certas ações (Métodos) para modificar essas informações. Podemos usar um objeto para armazenar os atributos e os métodos.

DECLARAÇÃO DE MÉTODO

Vamos começar nosso exemplo declarando um objeto player com duas propriedades como vimos antes de saúde e felicidade. A seguir, adicionaremos o primeiro de nossos três métodos necessários, play. É importante notar que o Método play é definido como uma propriedade do jogador com uma Função atribuída como seu valor, mas a própria Função é apenas definida e não executado aqui.

```
let player = {
  saude: 100,
  felicidade: 50,
  play: function() {
    this.felicidade += 10;
  }
};
```

EXECUTANDO

O objetivo do Método play é aumentar a propriedade diversão, indicando que nosso jogador gosta de jogar. Se tentarmos acessar a play() diretamente, obteremos um erro no console informando que a play() não está definida.
Como os métodos vivem dentro de um objeto, devemos usar uma palavra-chave de contexto JavaScript especial (this) dentro da função para acessar outras propriedades do objeto. A palavra-chave this se refere ao objeto (player) que possui o método e nos dá acesso a todas as suas propriedades, incluindo diversão, que agora podemos modificar dentro do método de jogo.
Os métodos são executados da mesma forma que as funções, com uma diferença importante. O nome do objeto deve ser usado para executar a função dentro dele. O exemplo abaixo faz com que o código seja executado exibindo a seguinte saída no console.

```
// Antes da execução do método
console.log(player);

// Execução do método
player.play();

// Após a execução do método
console.log(player);
```

Resultado obtido no console:

```
▶ {saude: 100, felicidade: 50, play: f}
▶ {saude: 100, felicidade: 60, play: f}
```

AMPLIANDO

Agora o nosso segundo método visa alimentar o nosso pet virtual, dando a ele uma laranja ou uma uva, quando isso acontece o player ganha +10 de saúde no caso da laranja e +20 de saúde no caso da uva:

Altere o objeto:

```
let player = {
  saude: 100,
  felicidade: 50,
  play: function() {
    this.felicidade += 10;
  },
  eat: function(comida){
    if(comida == 'laranja'){
        this.saude += 10;
    }
    else if(comida == 'uva'){
        this.saude += 20;
    }
  }
};
```

Executando o método no código:

```
// Antes da execução do método
console.log(player);

// Execução do método
player.eat('laranja');
player.eat('uva');

// Após a execução do método
console.log(player);
```

Resultado no console:

```
▶ {saude: 100, felicidade: 50, play: f, eat: f}
▶ {saude: 130, felicidade: 50, play: f, eat: f}
```

Agora é com você:

Exercício 8: Crie duas propriedades para o objeto acima: moedas (inicial = 20) e roupa (inicial = 'nao').

Crie um método **compraroupa()** para comprar uma roupa padrão que custa 20 moedas, se tiver o suficiente em moedas, você compra e a propriedade roupa ficará com o valor 'sim'. Caso contrário uma mensagem de erro deverá aparecer na tela.

AULA 9: LOOPS: WHILE

Os Loops While nos permitem executar um determinado trecho do código várias vezes. Configuramos um bloco While com uma instrução condicional que permite que o código dentro do loop seja executado repetidamente até que a condição não seja mais verdadeira. Embora os Loops possam ser poderosos na programação, deve-se tomar muito cuidado no gerenciamento da condição. Com isso, queremos dizer que a condição deve ter a capacidade de se tornar falso em algum ponto, caso contrário corremos o risco de criar um loop infinito que pode fazer com que seu aplicativo, e possivelmente seu servidor web travem.

Exemplo 1: Algoritmo que conta de 0 a 10:

```
// Inicializando a variável de soma
let total = 0;

// Inicializando a variável de limite
let j = 1;

/* Enquanto j é menos que 10, faça: */
while(j <= 10) {
  // Total recebe ele mesmo e +1
    total = total + 1;
  // Exibindo a soma passo a passo:
    console.log('Iteration Sub-Total:', total);
  // incremento da variável indicado um ciclo completo
    j++;
}

console.log('Total:', total);
```

Exemplo 2: Player perde 1 vida enquanto estiver colidindo com o inimigo, se chegar a 0 o loop deverá ser interrompido imediatamente:

```
let vidas = 100;
let colidindo = true;

/* Enquanto colindo = true, faça: */
while(colidindo) {
    vidas -= 1;
    console.log(vidas);
    if(vidas <=0){
        break;
    }
}
```

Repare que o comando break pode ser usado para interromper o loop imediamente, vamos falar mais sobre ele na próxima aula.
Agora é com você.
Exercício 9: Crie um loop para somar todos os números de 0 a 100.

AULA 10: LOOPS: BREAK E CONTINUE

Na última aula, aprendemos sobre os Loops While. Nesta, aprendemos sobre duas palavras-chave em JavaScript (Break e Continue) que podem ser usadas para controlar o fluxo de código dentro de um loop. Essas duas palavras-chave geralmente são usadas dentro de um loop e junto com Instruções Condicionais de Lógica Booleana .

BREAK

Comando que permite terminar a execução de um loop se uma determinada condição for atendida. Isso substitui a condição principal do loop While.
Ele é usado, sobretudo para interromper algo que "saiu do controle".

Exemplo: Nosso Player pode armazenar até 10 itens, e mergulhamos em um mar de itens em determinado momento do jogo, não poderemos armazenar todos eles, então o que fazer?.

```
let itens = 0;
let colidindoitens = true;

/* Enquanto colindo = true, faça: */
while(colidindoitens) {
    itens += 1;
    console.log(itens);
    if(itens >= 10){
        break;
    }
}
```

CONTINUE

Permite que você retorne à condição do loop while em vez de continuar a execução dentro do loop se uma determinada condição for atendida. É como se você cortasse o loop no meio.
Exemplo: Suponha que em determinado momento do jogo você colidiu com uma poção "castigo", e sempre que você colidir com uma moeda não poderá ganhar aqueles pontos, como podemos representar isso em JavaScript?.

```
let castigo = true
let colidiu = true
let moedas = 0

if(colidiu){
    if(castigo == true){
        continue;
    }
    moedas++;
}
```

Agora é com você:
Exercício 10: Um determinado computador antigo consegue armazenar números até o 9999, sua missão é criar um loop infinito que soma números de 1 em 1, e criar uma condição que interrompa o ciclo caso o número chegue a 9999.

AULA 11: LOOPS: FOR

Nas últimas duas lições, aprendemos sobre Loops While e duas palavras-chave JavaScript para controlar o fluxo dentro do loop. Nesta aula veremos um loop alternativo, o For. Configuramos e usamos esse loop de maneira diferente, mas é importante observar que ainda é um loop e ainda podemos usar as mesmas palavras-chave JavaScript (break e continue) para controlar o fluxo dentro dele.

DECLARAÇÃO

```
for(let i = 0; i < 10; i++) {
    console.log('Iteração:', i);
}
```

O For é um pouco mais detalhado que o While, repare que é declarada uma variável "I" com valor zero, e enquanto i é menor que 10, a variável i é acrescida de de 1 (i++ -> i = i + 1). O loop acima conta de 0 a 10, execute no console e veja cada iteração.

Exemplo: Vamos simular o crescimento populacional de uma cidade em um período de dez anos. Sabemos que a população atual é de 100 habitantes e que a taxa de crescimento esperada é de 5% ao ano . Qual será a população em 10 anos?

```
let populacao = 100;

// O limite é de 10 anos, i = 10
for(let i = 0; i < 10; i++) {
  // Aumentando 5% a cada iteração
    population *= 1.05;
}

// Mostrando o resultado final
console.log('Populacao apos 10 anos:', population);
```

Agora é com você:
Exercício 11: Crie um algoritmo que imprima a tabela de conversão de graus Celsius-Fahrenheit para o intervalo desejado pelo usuário.
O algoritmo deve solicitar ao usuário o limite superior, o limite inferior do intervalo.
Fórmula de conversão: C = 5 (F - 32) / 9.

AULA 12: ARRAYS

Arrays são objetos semelhantes a listas, comuns à maioria das linguagens de programação, usados para representar coleções ordenadas. Eles são coleções porque podem representar uma lista de qualquer tamanho contendo qualquer tipo de dados . Eles são considerados ordenados porque mantêm um índice numérico sequencial (representado como um inteiro) de zero ao comprimento do array - 1 .

estado
size = 4

DECLARAÇÃO

Para criar um array vazio, definimos uma variável recebendo colchetes vazios.
Para criar um array com quaisquer elementos, coloque o array recebendo os elementos com aspas simples e separados por vírgula dentro de colchetes.

Exemplo: Criaremos um array que represente os 4 estados de um player, conforme a imagem acima:

```
let estado = ['ganhou', 'andando', 'correndo', 'parado'];
console.log('Array:', estado);
console.log('Tamanho: ', estado.length);
```

E se quiséssemos acessar o primeiro e o último elementos do array?

```
console.log('Primeiro Elemento:', estado[0]);
console.log('Último Elemento: ', estado[estado.length-1]);
```

De modo geral, para acessar o elemento do array basta especificar a posição na notação nomedoarray[posição-1], se ele for o 3º elemento o número da posição é 2, dado que o array começa a posição zero.

ATRIBUIÇÃO

Agora que já aprendemos a recuperar um dado em uma posição específica no código, vamos usar esse acesso a uma posição específica para alterar o seu conteúdo.
Altere o último elemento no array de 'parado' para 'andando':

```
estado[estado.length-1] = "andando";
console.log('Último Elemento: ', estado[estado.length-1]);
```

ADICIONANDO ELEMENTOS

Arrays de JavaScript têm muitos métodos embutidos. Para adicionar um elemento ao final de um array, podemos usar o método **push**.

```
estado.push('parado');
console.log(estado);
```

O nosso array ficará assim:

```
▼Array(5) i
    0: "ganhou"
    1: "andando"
    2: "correndo"
    3: "andando"
    4: "parado"
    length: 5
```

DELETANDO UM ELEMENTO

Arrays de JavaScript têm muitos métodos embutidos. Para remover um elemento do final de um array, podemos usar o método **pop** . Este método não tem parâmetros.

```
estado.push('parado');
estado.pop();
console.log(estado);
```

Observe que agora temos 4 elementos no array porque removemos o último.

```
▼(4) ["ganhou", "andando", "correndo", "andando"]
    0: "ganhou"
    1: "andando"
    2: "correndo"
    3: "andando"
    length: 4
```

Agora é com você.

Exercício 12: Crie um array chamado mapa, que armazenará as vilas de um jogo RPG.

a) Adicione nele as seguintes vilas: Fire, State, PeopleFear, JF.

b) Adicione mais duas ao final da lista: Jungle, Orquidea.

c) Remova a vila Orquidea.

d) Altere o nome da vila 'Fire' para 'CaveMens'.

AULA 13: OPERAÇÕES COM ARRAYS

Um dos aspectos mais poderosos do array é a capacidade de operar facilmente sobre ele e realizar ações, cálculos e muito mais para cada um dos seus elementos.
Há muitas maneiras de operar em Arrays e aqui abordaremos três: **While**, **For** e um método integrado de Array chamado **forEach**. É importante ter em mente que o primeiro elemento de um Array está no índice 0 e o último elemento está no índice tamanho - 1.

Exemplo: Um Professor armazena a nota de seus alunos em um array, durante a correção da prova de 5 questões que valem um total de 100 pontos, ele teve que anular uma questão, e agora todas as notas deverão ser acrescidas de 20 pontos:
Construímos então o problema usando o loop **While**:

```
let notas = [40, 20, 50,43,23,56,22,34,43,44];

console.log('Notas antes da correção:', notas);
// usando um loop para percorrer todo o array
let i = 0;
while(i < notas.length) {
  notas[i] += 20;
  i++;
}
console.log('Notas após a correção:', notas);
```

A saída no Console será:

```
Notas antes da correção:
▶ (10) [40, 20, 50, 43, 23, 56, 22, 34, 43, 44]
Notas após a correção:
▶ (10) [60, 40, 70, 63, 43, 76, 42, 54, 63, 64]
```

Outra maneira de fazer isso é iterar através de um loop **For** e aumentar um elemento de pontuação em cada iteração do loop até que todas as pontuações tenham sido aumentadas. Esta é simplesmente uma abordagem diferente da anterior.

```
let notas = [40, 20, 50,43,23,56,22,34,43,44];

console.log('Notas antes da correção:', notas);
// usando um loop para percorrer todo o array
for(i = 0; i < notas.length; i++){
  notas[i] += 20;
}
console.log('Notas após a correção:', notas);
```

O método embutido JavaScript Array **forEach** executa uma função fornecida uma vez para cada elemento Array. A função fornecida é conhecida como retorno de chamada e pode ter três parâmetros:

```
let notas = [40, 20, 50,43,23,56,22,34,43,44];

console.log('Notas antes da correção:', notas);
// usando um loop para percorrer todo o array
notas.forEach(function (entry, index, scores) {
    notas[index] += 20;
});
console.log('Notas após a correção:', notas);
```

Agora é com você.

Exercício 13: Suponha que em um jogo de RPG, o seu inventário é um array com 4 posições, uma determinada poção é armazenada no inventário como o número 1, sua missão é criar um programa que percorra o seu array inventário em busca do número 1, se encontrar o número 1 deverá mostrar na tela: "Poção encontrada!".

AULA 14: ARRAYS BIDIMENSIONAIS

Arrays bidimensionais são Arrays dados em duplas. Por exemplo:

let estado = [['parado','pulo']['esquerda','direita']]

Isso permite uma maneira intuitiva de armazenar 2D, 3D ou até valores dimensionais superiores. No caso mais simples, por exemplo, um array 2D, você pode pensar nele como uma tabela com linhas, colunas e uma célula que se cruza contendo informações como em uma planilha.

DECLARAÇÃO

Considerando o Array bidimensional mostrado acima, podemos visualizar 2 linhas e 2 colunas. Agora vamos codificá-lo em Javascript:

```
let estado = [['parado','pulo'], ['esquerda','direita']];
console.log(estado);
```

Resultado no console:

```
▼(2) [Array(2), Array(2)]
  ▶0: (2) ["parado", "pulo"]
  ▶1: (2) ["esquerda", "direita"]
    length: 2
```

ACESSANDO ELEMENTOS

Para ajudar a visualizar como acessar os elementos do Array, você pode abrir os Arrays dentro do Array principal para ver cada índice e seu valor.

```
▼(2) [Array(2), Array(2)]
  ▼0: Array(2)
      0: "parado"
      1: "pulo"
      length: 2
    ▶__proto__: Array(0)
  ▼1: Array(2)
      0: "esquerda"
      1: "direita"
      length: 2
```

Então, se quisermos acessar as posições os conteúdos: "pulo" e "direita", fazemos: estado[numero da linha][numero da coluna]:

```
let estado = [['parado','pulo'], ['esquerda','direita']];
console.log(estado);

// Acessando a posição "pulo"
console.log(estado[0][1]);

// Acessando a posição "direita"
console.log(estado[1][1]);
```

Sua tarefa é criar o game "Campo Minado":
a) Crie uma matriz bidimensional 4x4 que armazenará o tabuleiro do jogo, espaço em branco será representado pelo 0 e bomba será o 1, distribua 0 e 1 pelas posições da matriz.
b) Utilize variáveis para armazenar algumas posições específicas da matriz, a sua escolha
Exemplo: jogada1 = campo[0][0].
c) Armazene todas as jogadas em um array chamado jogo.
d) Utilize um loop a sua escolha para percorrer todas as posições do array jogo, se pelo menos uma das posições tiver uma bomba, a mensagem "você perdeu" deverá ser exibida, se todo o vetor for percorrido e nenhuma bomba for encontrada, a mensagem "você venceu" deverá ser exibida como alerta.

SNAKE

INTRODUÇÃO

Crie uma pasta e nela teremos 2 arquivos, um .html de nome "Index" e um .js de nome "script".
No arquivo "index.html" vamos configurar o tamanho da tela e chamar pelo arquivo "script.js":

```
<canvas id="canvas" width="400" height="400"></canvas>

<script src="script.js"></script>
```

Agora vamos criar a função base de funcionamento da tela do jogo, assim que a janela atual for carregada, a tela deverá ser criada e posteriormente dar início ao jogo:

```
window.onload = function(){}
```

Tudo que vamos criar, vai estar dentro dos colchetes desta função.
Comece recuperando o tamanho da tela que criamos no canvas na outra janela.

```
// Recupera o tamanho da tela
canvas = document.getElementById("canvas");
ctx = canvas.getContext("2d");
```

Agora vamos criar as variáveis que precisaremos durante o desenvolvimento deste jogo:

```
// variáveis
snake = [];
positionX = 10;
positionY = 10;
foodX = 15;
foodY = 15;
velX = 0;
velY = 0;
grid = 20;
tam = 3;
```

A cobra será uma array com vários quadrados juntos, um após o outro.
As variáveis **positionX** e **positionY** controlam a posição da cobra a partir da cabeça.
foodX e **foodY** controlam a posição da comida, vamos gerar várias daqui a pouco.
velX e **velY** controlam a velocidade da cobra.
grid controla o tamanho da grade, composta de 20 quadrados.
tam é o tamanho inicial da cobra.
Na próxima aula vamos criar a função jogo e dar início a construção da mecânica desse jogo.

MECÂNICA DO JOGO

Crie a nova função chamada jogo, logo após o colchetes da função onload que criamos na aula anterior, essa função vai controlar a mecânica do jogo e deverá ser chamada a partir da função onload.

```
function jogo(){}
```

O jogo é contínuo, ele acontece a cada segundo, então nós não podemos chamar a função jogo apenas uma vez, teremos que fazer isso de uma maneira contínua, e para isso vamos usar a função **setInterval()** que chama uma outra função em um espaço de tempo em milisegundos, coloque a chamada da função dentro da onload:

```
// Chamada a função jogo a cada x milisegundos
setInterval(jogo, 100);
```

Já dentro da função jogo(), vamos colorir a tela do jogo com a função fillStyle():

```
// Configuração da tela
ctx.fillStyle = "#2980B9";
// deslocamento x, deslocamento y, largura, altura
ctx.fillRect(0,0, canvas.width, canvas.height);
```

Agora vamos criar a nossa cobra com a ajuda do loop for:

```
// Configuração da cobra
ctx.fillStyle = "#00f102";
for(var i=0; i < snake.length; i ++){
    ctx.fillRect(snake[i].x * grid, snake[i].y * grid, grid - 1 , grid - 1);
}
```

O contexto é mutiplicado pelo grid nas duas primeiras casas apenas para centralizar na tela.
Para atualizar a posição da cobra, usaremos a função **push()** antes da função acima:

```
// Posicionando a cobra
snake.push({x: positionX, y: positionY});
```

ENTRADA DE DADOS

Na anterior, criamos a tela do jogo e a cobra que será o player, mas como iremos controlá-la? No Javascript temos a função keydown, que monitora quais teclas estão sendo pressionada no teclado.
E para organizar o nosso raciocínio, vamos usar o comando **switch**, que é como se fosse um comand if else mais organizado:

```
// Controles
document.addEventListener("keydown", function(e){
    switch(e.keyCode){
        // direita = 39
        case 39:
            velX = 1;
            velY =0;
            break;
            // esquerda = 37
        case 37:
            velX = -1;
            velY = 0;
            break;
            // cima = 38
        case 38:
            velY = -1;
            velX = 0;
            break;
            // down = 40
        case 40:
            velY = 1;
            velX = 0
            break;
    }
});
```

Existem sites que te mostram o keycode de cada tecla.
E para que a cobra se movimente a partir dessa atualização de varíaveis, adicione o trecho a seguir dentro de função jogo() e acima da criação da cobra:

```
// deslocamento
positionX += velX;
positionY +=velY;
```

Execute e veja que está funcionando, mas a cobra tem tamanho infinito, precisamos limitar o seu tamanho de acordo com o valor da variável tam.
Adicione o trecho embaixo da criação da cobra.

```
// Apagando
while(snake.length > tam){
    snake.shift();
}
```

O shift() tira o primeiro valor de dentro de um array. Se a cobra tiver um tamanho maior que o permitido, esse ou esses quadrados serão cortados.
Repare que quando a cobra chega na borda, ela desaparece, vamos fazer com que as bordas sejam espelhadas:

```
// Espelhando
if(positionX < 0){
    positionX = grid;
}
if(positionX > grid){
    positionX = 0;
}
if(positionY < 0){
    positionY = grid;
}
if(positionY > grid){
    positionY = 0;
}
```

COMIDA

Começamos criando um novo objeto amarelo que será a comida:

```
// Configurando a comida
ctx.fillStyle = "#F1C40F";
ctx.fillRect(foodX* grid, foodY * grid, grid - 1, grid - 1);
```

Agora a condição para comer a comida:

```
// Comendo
if(positionX == foodX && positionY == foodY){
    tam++;
    foodX = Math.floor(Math.random() * grid);
    foodY = Math.floor(Math.random() * grid);
}
```

Execute e veja se está tudo certo.
Agora a condição para reiniciar o jogo, quando a cobra toca no próprio corpo.

```
// Configuração da cobra
ctx.fillStyle = "#00f102";
for(var i=0; i < snake.length; i ++){
    ctx.fillRect(snake[i].x * grid, snake[i].y * grid, grid - 1 , grid - 1);
    if(snake[i].x == positionX && snake[i].y == positionY){
        tam = 5;
    }
}
```

Tudo certo, boa partida!.

FLAPPY BIRD

INTRODUÇÃO

Começaremos baixando o material da aula e no nosso arquivo index.html, onde vamos organizar a tela do jogo, colocando um título do navegador e um título interno, que vai aparecer acima da tela do jogo, nossa tela terá o tamanho 288x512 pixels, e vamos conectar aqui o nosso arquivo script.js :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Flappy Bird</title>
  </head>
  <body>
   <h3>Flappy Bird by AG</h3>

   <canvas id="canvas" width="288" height="512"></canvas>

   <script src="script.js"></script>
  </body>
</html>
```

Agora, no arquivo script.js, começamos referenciando o canvas, para obter o tamanho da tela e configurando o aspecto 2d da tela:

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
```

Agora, vamos usar a função Image() para carregar as imagens do jogo, não iremos utilizar css neste game, meu objetivo com isso é mostrar que você também pode carregar tudo no Javascript, futuramente neste mesmo curso vamos carregar imagens com css.

```
// Carregando imagens
var bird = new Image();
bird.src = "images/bird.png";
var bg = new Image();
bg.src = "images/bg.png";
var chao = new Image();
chao.src = "images/chao.png";
var canocima = new Image();
canocima.src = "images/canocima.png";
var canobaixo = new Image();
canobaixo.src = "images/canobaixo.png";
```

Agora vamos declarar as variáveis que vamos usar durante o game:

eec = variável que controla o espaço entre os canos, em pixels

constant = variável que vai ajudar em uma soma futura

bx = posição x do pássaro

by = posição y do pássaro

gravity = gravidade que puxará o pássaro para baixo.

score = acúmulo do pontos.

```
var eec = 100;
var constant;
var bX = 33;
var bY = 200;
var gravity = 1.4;
var score = 0;
```

E agora vamos carregar os sons:

```
var fly = new Audio();
fly.src = "sounds/fly.mp3";
var scor = new Audio();
scor.src = "sounds/score.mp3";
```

CENÁRIO

Os nossos canos serão armazenados no array de nome "cano", vamos declará-lo e colocar sua posição horizontal inicial ao fim da tela, os canos sempre virão do fim da tela a direita se movimentando para a esquerda.

```
var cano = [];

cano[0] = {
    x : canvas.width,
    y : 0
};
```

Agora vamos a criar a função jogo(), que controlará toda a lógica do game.
Ela terá ao seu final, a função **requestAnimationFrame()**, que chama a função jogo recursivamente, como uma animação.
E após a declaração da função, adicionamos uma chamada, para que ela seja ativada pela primeira vez, e a partir daí seja chamada infinitas vezes pela requestAnimationframe():

```
function jogo(){
    requestAnimationFrame(jogo);
}

jogo();
```

Tudo que será adicionado nessa função, será acima do trecho requestAnimationframe(), começamos posicionando o fundo do jogo com a **drawImage()**:

```
ctx.drawImage(bg,0,0);
// drawImage(imagem,X,Y)
```

Agora vamos configurar o chão do jogo e o pássaro, já incluindo nele a gravidade que o puxará para baixo:

```
// Desenhando o chão
ctx.drawImage(chao,0,canvas.height - chao.height);
// Desenhando o Pássaro
ctx.drawImage(bird,bX,bY);
// Ação da gravidade
bY += gravity;
```

Agora vamos configurar o voo do pássaro, começamos criando uma função para monitorar a entrada de dados do teclado, se uma tecla for pressionada, essa função chamará a função voa() que fará o pássaro se deslocar 26 pixels para cima:

```
// Captura de tecla
document.addEventListener("keydown",voa);

// Voando
function voa(){
    bY -= 26;
    fly.play();
}
```

Execute e veja o cenário construído e o pássaro voando, a seguir vamos criar os canos.

CONFIGURANDO OS CANOS

Na criação dos canos, vamos usar um for que vai ler o tamanho do array cano, mas esse array ficará aumentando de tamanho infinitamente.
A variável "constant", vai pegar a posição do cano de cima e somar a ela o valor da variável eec (espaço entre os canos) e a partir daí, poderemos obter uma posição vertical para o cano de baixo.
Em seguida, desenhamos o cano de cima, e o cano de baixo, repare que sua posição y não é igual ao cano de cima, ela é acrescida de constant.

```
for(var i = 0; i < cano.length; i++){
    // Espaço entre os canos
    constant = canocima.height+eec;
    // Configuração do cano de cima
    ctx.drawImage(canocima,cano[i].x,cano[i].y);
    // Configuração do cano de baixo
    ctx.drawImage(canobaixo,cano[i].x,cano[i].y+constant);
    // Movimentação do cano
    cano[i].x--;
    // Quando o cano alcança x=125, um novo é criado
    if( cano[i].x == 125 ){
        cano.push({
            x : canvas.width,
            y : Math.floor(Math.random()*canocima.height)-canocima.height
        });
    }
```

MECÂNICA

Agora vamos concluir a criação da mecânica do jogo, existem 2 coisas a serem trabalhadas aqui: o fim do jogo e a marcação de pontos, começamos criando a condição que vai detectar se o pássaro colidiu um cano ou com o chão:

```
// Passáro entre as duas bordas do cano
if( bX + bird.width >= cano[i].x && bX <= cano[i].x + canocima.width
    // Pássaro colidiu com a parte de cima ou com a parte de baixo
    && (bY <= cano[i].y + canocima.height || bY+bird.height >= cano[i].y+constant)
    // Passaro caiu no chão?
    || bY + bird.height >=  canvas.height - chao.height){
    location.reload(); // reinicia o jogo
}
```

Se o cano conseguiu chegar a posição x = 5, quer dizer que ele já passou pelo pássaro e o pássaro não colidiu com ele, logo o player marcará um ponto:

```
if(cano[i].x == 5){
    score++;
    scor.play();
}
```

Em seguida, construímos o placar ao fim da função (acima da requestAnimationframe()):

```
ctx.fillStyle = "#000";
ctx.font = "20px Verdana";
ctx.fillText("Placar : "+score,10,canvas.height-20);
```

Tudo pronto, boa partida!.

CODI MEMORY

INTRODUÇÃO

Neste game, vamos usar um arquivo .html, um arquivo .js e um arquivo .css que é a chamada folha de estilo, que vamos usar para configurar a tela do jogo de maneira separada.
Começamos com o arquivo .html para conecta os outros dois:

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
<head>
  <meta charset="UTF-8">
  <title>Codi Memory</title>
  <link rel="stylesheet" href="style.css"></link>
  <script src="script.js" charset="utf-8"></script>
</head>
<body>

  <h3>Placar: <span id="result"></span></h3>

  <div class="grid">
  </div>

</body>
</html>
```

Agora a folha de estilo "style.css":

```
.grid {
  display: flex;
  flex-wrap: wrap;
  width: 400px;
  height: 300px;
}
```

Em seguida, começamos a programar o jogo em "script.js", neste momento você deve baixar a pasta com as imagens e colocar junto aos arquivos da aula na mesma pasta em que eles estão.
Todo o game será programado dentro da função a seguir, que basicamente avisa quando o HTML foi completamente carregado:

```
document.addEventListener('DOMContentLoaded', () => {})
```

Em seguida, vamos criar um Array que vai guardar o nome e a imagem de cada um dos nossos cards:

Temos que carregar cada um dos cards 2 vezes, pois queremos formar pares.

```
// Carregamento dos cards
const cardArray = [
  {
    name: 'ganhou',
    img: 'images/ganhou.png'
  },
  {
    name: 'ganhou',
    img: 'images/ganhou.png'
  },
  {
    name: 'direita',
    img: 'images/direita.png'
  },
  {
    name: 'direita',
    img: 'images/direita.png'
  },
  {
    name: 'tras',
    img: 'images/tras.png'
  },
  {
    name: 'tras',
    img: 'images/tras.png'
  },
  {
    name: 'correndo',
    img: 'images/correndo.png'
  },
  {
    name: 'correndo',
    img: 'images/correndo.png'
  },
  {
    name: 'pulo',
    img: 'images/pulo.png'
  },
  {
    name: 'pulo',
    img: 'images/pulo.png'
  },
  {
    name: 'esquerda',
    img: 'images/esquerda.png'
  },
  {
    name: 'esquerda',
    img: 'images/esquerda.png'
  }
]
```

Criamos agora uma referência a classe grid que criamos no arquivo .css para formatar a tela do jogo:

```
const grid = document.querySelector('.grid')
```

Em seguida, criaremos a função que montará a tela com os cards virados.

```
// Criando a tela
function createBoard() {
  for (let i = 0; i < cardArray.length; i++) {
    var card = document.createElement('img')
    card.setAttribute('src', 'images/card.png')
    card.setAttribute('data-id', i)
    card.addEventListener('click', flipCard)
    grid.appendChild(card)
  }
}
```

O comando for varre todo o cardArray, distribuindo os cards pela tela com a imagem "card", mas cada um com o seu data-id, a função **addEventListener()** monitora a entrada de dados do mouse, se houver um click ele chama a função flipcard() que vai virar os cards. Em seguida, cada card é colocado como filho de grid, coloque uma chamada dessa função antes do fim do código.

```
createBoard()
```

VIRANDO CARDS

Acima da função createboard(), declare uma constante **resultDisplay** que vai se encarregar de anexar o placar do jogo e mantê-lo atualizado acima do board.
O array **cardsChosen**, que armazena os cards escolhidos a cada jogada
O array **cardChosenId** que armazena os cards escolhidos a cada jogada
e o array **pares**, esse eu coloquei bem diferente para se destacar dos outros, pois é ele que controla a mecânica base do jogo, armazenando os cards que fizeram par.

```
const resultDisplay = document.querySelector('#result')
var cardsChosen = []
var cardsChosenId = []
var pares = []
```

Agora vamos criar a função para virar os cards:

```
// Virando os cards
function flipCard() {
  var cardId = this.getAttribute('data-id')
  cardsChosen.push(cardArray[cardId].name)
  cardsChosenId.push(cardId)
  this.setAttribute('src', cardArray[cardId].img)
  if (cardsChosen.length ===2) {
    setTimeout(checkForMatch, 500)
  }
}
```

Sempre que clicamos em um card, disparamos essa função que começa declarando uma variável **cardID** que recebe o ID daquele card clicado.
O array **cardsChosen** é acrescido do nome do card clicado.
O array **cardsChosenID** é acrescido do ID do card clicado.
Em seguida, o card receberá a imagem de acordo com seu id.
Se clicamos em 2 cards, vamos chamar a função para verificar se formamos ou não um par.

CONFERINDO PARES

Acima da função flipcard(), vamos criar a função para verificar se a escolha de dois cards forma um par.

```
// Conferindo pares
function checkForMatch() {
  var cards = document.querySelectorAll('img')
  const optionOneId = cardsChosenId[0]
  const optionTwoId = cardsChosenId[1]
```

Se clicarmos duas vezes no mesmo card, vamos exibir uma mensagem de erro na tela e virar o card novamente:

```
if(optionOneId == optionTwoId) {
  cards[optionOneId].setAttribute('src', 'images/card.png')
  cards[optionTwoId].setAttribute('src', 'images/card.png')
  alert('Você clicou na mesma imagem')
}
```

Se escolhermos dois cards diferentes, e o ID do primeiro card escolhido for igual ao ID do segundo card escolhido, conseguimos formar um par e neste caso os cards ficarão com a imagem branca e não serão mais clicáveis, e ao fim serão adicionados ao array pares. Caso contrário, clicamos em cards que não formam um par e eles serão virados novamente.

```
else if (cardsChosen[0] === cardsChosen[1]) {
  alert('Você conseguiu um par!')
  cards[optionOneId].setAttribute('src', 'images/white.png')
  cards[optionTwoId].setAttribute('src', 'images/white.png')
  cards[optionOneId].removeEventListener('click', flipCard)
  cards[optionTwoId].removeEventListener('click', flipCard)
  pares.push(cardsChosen)
} else {
  cards[optionOneId].setAttribute('src', 'images/card.png')
  cards[optionTwoId].setAttribute('src', 'images/card.png')
  alert('Ops! Jogue Novamente :)')
}
```

Logo depois, vamos resetar os arrays dessa jogada e exibir o quantidade de pares formados na tela:

```
cardsChosen = []
cardsChosenId = []
resultDisplay.textContent = pares.length
```

Se o array pares, tiver o tamanho referente a metade dos cards carregados, quer dizer que todos os pares foram registrados, e você venceu o jogo.

```
  if  (pares.length === cardArray.length/2) {
    resultDisplay.textContent = 'Parabéns! Você encontrou todos os pares!'
  }
}
```

Também colocaremos uma opção para melhorar a aleatoriedade do jogo:

```
cardArray.sort(() => 0.5 - Math.random())
```

Tudo pronto, boa partida!.

DINO

INTRODUÇÃO

Assim como no jogo anterior, vamos precisar usar um arquivo .html e um arquivo .css junto com o nosso arquivo em Javascript.
Começamos criando uma arquivo "index.html", em sua primeira parte, vamos colocar o título do jogo e deixar indicada a conexão dos próximos arquivos.

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
<head>
  <meta charset="UTF-8">
  <title>Jogo dino</title>
  <link rel="stylesheet" href="style.css"></link>
  <script src="script.js" charset="utf-8"></script>
</head>
```

Na segunda parte do arquivo html, vamos encaixar as peças do jogo que vamos configurar no .css, são elas: desert, dino e grid, cada uma mais externo ao outro, e também colocaremos um alert que vai mostrar mensagens ao player:

```
<body>

  <div id="desert">
    <h2 id="alert"></h2>
    <div class="grid">
      <div class="dino"></div>
    </div>
  </div>

</body>
</html>
```

Agora vamos criar o arquivo "style.css":
Com o uso de ".", podemos criar classes, elas representam a configuração de objetos do jogo que podem ser gerados pelo código e não serão únicos, e essas classes serão chamadas nos outros arquivos.
Começamos com a classe dino, que coloca a textura no dinossauro e configura as suas dimensões: (absolute: maneira mais manual de posicionar um elemento na tela)

```
.dino {
  position: absolute;
  width: 60px;
  height: 60px;
  background-image: url(dino.png);
  bottom: 0px;
}
```

Agora vamos criar a classe obstacle, que representará os cactus:

```
.obstacle {
  position: absolute;
  width: 60px;
  height: 60px;
  background-image: url(cactus.png);
  bottom: 0px;
}
```

E agora vamos criar um ID chamado desert.
Os ids são uma forma de identificar um elemento, e devem ser ÚNICOS para cada elemento. É como se fossem impressões digitais de nossos dedos ou RGs. Através deles, pode-se atribuir formatação a um elemento em especial.
Neste caso, o deserto do game será único.

```
#desert {
  position: absolute;
  bottom: 0px;
  background-image: url('bg.png');
  background-repeat: repeat-x;
  animation: slideright 600s infinite linear;
  -webkit-animation: slideright 600s infinite linear;
  width: 100%;
  height: 200px;
}
```

Agora no arquivo script.js, vamos configurar para que o Javascript seja executado assim que o HTML for carregado:

```
document.addEventListener('DOMContentLoaded', () => {})
```

Na sequência, dentro desta função vamos referenciar os ids e classes do jogo, salvando em variáveis, para facilitar seu encaixe no jogo.

```
const dino = document.querySelector('.dino')
const grid = document.querySelector('.grid')
const body = document.querySelector('body')
const alert = document.getElementById('alert')
```

Tambem vamos declarar algumas variáveis que serão úteis mais a frente:

```
  let jumping = false
  let gravity = 0.9
  let gameo = false
  let dinopy = 0
```

Agora, vamos finalizar o arquivo style.css com os comandos responsáveis por animar o chão do jogo:

```
@keyframes slideright {
  from {
      background-position: 70000%;
  }
  to {
      background-position: 0%;
  }
}

@-webkit-keyframes slideright {
  from {
      background-position: 70000%;
  }
  to {
      background-position: 0%;
  }
}
```

PULANDO

Agora vamos criar as funções que vão configurar o pulo do dinossauro, começamos com a função jumpcontrol(), que vai tratar o pulo do dinossauro, se apertarmos a tecla espaço (código 32) e o player não estiver pulando, a variável jumping recebe o valor true e a função jump() é chamada: (O site site keycode.info te fornece esses dados)

```
function jumpcontrol(e) {
  if (e.keyCode === 32) {
    if (!jumping) {
      jumping = true
      jump()
    }
  }
}
```

Essa função será chamada quando pressionarmos uma tecla, pode ser qualquer tecla pois ela será tratada dentro da função, também vamos declarar e colocar o valor zero na variável position:

```
document.addEventListener('keyup', jumpcontrol)
```

Agora vamos criar a função jump(), que fará o player pular:
Começamos declarando um contador, e a variável timerId, que recebe o valor da função setInterval(function, time), que chama uma determinada função em um tempo, a função nós vamos criar aqui mesmo e o tempo será de 20ms.
O método **setInterval ()** continuará chamando a função até que **clearInterval()** seja chamado ou a janela seja fechada.
setInterval(function() {}, 20), esses 20ms são como o FPS do jogo, quanto menor mais rápido esse aspecto do jogo vai, se usarmos 40 o pulo vira mais lento, abra as chaves.
O pulo da física, é separado em dois momentos, o momento da subida e o da descida,, começamos com subida.
Vamos somar +30px ao player, adicionamos 1 ao contador de rodadas, suavizamos o movimentos com a gravity e movemos a textura, traduzindo números para a unidade 'px'.
Se o contador estiver em 15 rodadas, quer dizer que o player não poderá mais subir na tela, e começará a cair, neste caso vamos parar o **setInterval()** de subida e vamos abrir um novo, o de descida, começamos decrementando - 5 da variável position e 1 da count, a cada 20ms ele perde 5 pixels, multiplicamos isso a gravidade para dar suavidade aos movimentos, e movimentamos a textura de acordo com a nossa escolha "absolute" lá no css.
Nós estamos decrementando o count, se ele chega a 0, vamos parar de chamar a função e colocar o valor "false" na variável jumping, indicado que o pulo terminou.

```
function jump() {
  let count = 0
  let timerId = setInterval(function () {
    // caindo
    if (count === 15) {
      clearInterval(timerId)
      let downTimerId = setInterval(function () {
        if (count === 0) {
          clearInterval(downTimerId)
          jumping = false
        }
        dinopy -= 5
        count--
        dinopy = dinopy * gravity
        dino.style.bottom = dinopy + 'px'
      },20)

    }
    // subida
    dinopy +=30
    count++
    dinopy = dinopy * gravity
    dino.style.bottom = dinopy + 'px'
  },20)
}
```

OBSTÁCULOS

Chegou a hora de gerarmos os obstáculos que o player terá que desviar, e vamos amarrar aqui toda a lógica do jogo:

Começamos declarando a função geraobst().
A variável **randomTime** receberá um tempo aleatório entre 0 e 4000 milisegundos.
A variável **obstaclepx** receberá a posição inicial horizontal em x = 1000.
A constante **obstacle**, faz uma referência a div que declaramos no arquivo .html, será ali que os obstáculos serão inseridos.

```
function gerarobst() {
  let randomTime = Math.random() * 4000
  let obstaclepx = 1000
  const obstacle = document.createElement('div')
```

Se o fim do jogo ainda não aconteceu, vamos adicionar um novo obstáculo, primeiro referenciado a classe obstacle que fizemos no arquivo .css, e depois adicionando esse clone a grid, ao jogo.
Em seguida, atualizando a unidade do deslocamento para 'px'.

```
if (!gameo) obstacle.classList.add('obstacle')
grid.appendChild(obstacle)
obstacle.style.left = obstaclepx + 'px'
```

Agora vamos tratar o movimento desses obstáculos, com um setInterval(function() {}, 20), inicialmente a função apenas deslocará para a esquerda os obstáculos de 10 em 10 pixels.
Mas, se a posição do obstáculo estiver entre 0 e 60, quer dizer que ele está na área do dinossauro, e se a posição (dinopy) do dinossauro for menor que 60, que dizer que o dinossauro não está pulando, neste caso haverá o "game over".

```
let timerId = setInterval(function() {
  console.log(dinopy)
  if (obstaclepx > 0 && obstaclepx < 60 && dinopy < 60) {
    clearInterval(timerId)
    alert.innerHTML = 'fim de jogo'
    gameo = true
    // Remover todas as cópias
    body.removeChild(body.firstChild)
    while (grid.firstChild) {
      grid.removeChild(grid.lastChild)
    }
  }
  obstaclepx -=10
  obstacle.style.left = obstaclepx + 'px'
},20)
```

Se o jogo não terminou, a função gerarobst() seja chamada a cada x milisegundos aleatórios, ao final da declaração da função chamaremos ela pela primeira vez.

```
    if (!gameo) setTimeout(gerarobst, randomTime)
}

gerarobst()

})
```

Tudo certo, boa partida!.