alura

Conhecendo HTML, CSS e Javascript_

Construa suas primeiras aplicações em 10 dias.

3ª Edição

Edição

Vivian Matsui
Sabrina Barbosa

[2021]
Casa do Código
Rua Vergueiro, 3185 - 8º andar
04101-300 – Vila Mariana – São Paulo – SP – Brasil
casadocodigo.com.br

grupo alura

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código é a editora do Grupo Alura, grupo que nasceu da vontade de criar uma plataforma de ensino com o objetivo de incentivar a transformação pessoal e profissional através da tecnologia.

Juntas, as empresas do Grupo constroem uma verdadeira comunidade colaborativa de aprendizado em programação, negócios, design, marketing e muito mais, oferecendo inovação na evolução dos seus alunos e alunas através de uma verdadeira experiência de encantamento.

Venha conhecer os cursos da Alura e siga-nos em nossas redes sociais.

alura.com.br

@casadocodigo

@casadocodigo

ISBN

Impresso e PDF:

EPUB:

MOBI:

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

SOBRE ESTE LIVRO

A ideia de criar este livro veio durante a produção da 3ª edição da Imersão Dev da Alura que aconteceu entre os dias 13 e 24 de setembro de 2021. A Imersão foi pensada para pessoas sem nenhum contato anterior com programação, e sabemos que esta primeira experiência pode ser decisiva! Assim, ficamos felizes em acompanhar quem está começando agora da melhor forma possível, e sabemos que isso pode ser feito de várias maneiras: síncrona, assíncrona, por vídeo e também por texto.

O conteúdo a seguir está totalmente baseado nos 5 primeiros projetos desenvolvidos durante a Imersão Dev, mas conta com algumas explicações extras e diferentes. Ele foi pensado para servir de apoio ao conteúdo da Imersão, mas não é apenas uma transcrição dos vídeos e sim material extra em um formato mais adequado para o texto. Há várias formas de estudar e vários tipos de suporte para isso: vídeo, texto, áudio e inclusive papel. Parte do desafio do estudo de programação é justamente encontrar o que funciona melhor para você. Esperamos que este livro possa lhe ajudar de alguma forma a entrar com o pé certo nesse novo mundo.

Bons estudos!

SOBRE QUEM ESCREVEU

A produção dos capítulos foi dividida entre nosso time de conteúdo de programação:

Guilherme Lima Um dos condutores da Imersão Dev junto com a Rafaella, é desenvolvedor de software formado em Sistemas de Informação e possui experiência em programação usando diferentes tecnologias como Python, Javascript e Go.

Rafaella Ballerini Condutora da Imersão Dev com o Guilherme Lima, é instrutora front end da Alura e uma ex estudante de medicina apaixonada por tecnologia e que se jogou no mundo da programação.

Juliana Amoasei Desenvolvedora JavaScript com background multidisciplinar, sempre aprendendo para ensinar e vice-versa. Atualmente trabalho como instrutora na Escola de Programação da Alura e dou mentoria técnica a iniciantes na área de desenvolvimento web frontend e backend.

AGRADECIMENTOS

A todas as 80.000 pessoas que se inscreveram para participar da Imersão Dev, motivadas pela mudança de carreira, reforço nos estudos ou, também, por que não, pela curiosidade de aprender algo novo. A Imersão foi feita para vocês, assim como todo este livro e os materiais extras.

A todos os times da Alura! Quem aparece são as devs e os devs, mas todas as imersões são produzidas por muitas mãos: time de devs que pensa e produz o conteúdo, didática que garante que o conteúdo faça sentido para quem está vendo de fora, edição e produção de vídeo, marketing, que faz com que tanta gente entre em contato com o evento e possa aprender programação com a gente, e o Scuba Team que dá suporte nas plataformas. Nossos

agradecimentos!

COMO UTILIZAR ESTE LIVRO

Tudo que se refere a **trechos de código** neste livro utiliza uma das duas notações abaixo:

- trechos pequenos ou termos, como nomes de funções, palavras-chaves ou expressões aparecerão no texto `desta forma` . Por exemplo, quando mencionamos uma função chamada `executarCodigo()` ou uma palavra-chave como `.length` .
- trechos maiores de código, como blocos de função, objetos ou similares aparecerão no texto em um parágrafo separado, como por exemplo:

```
function exibeTexto() {
   console.log("Olá Mundo!")
}
```

Todo o código será disponibilizado via link para consulta ou download (mais sobre isso abaixo), **mas é importante que você leia para entender o que acontece em cada parte!**

Sinta-se a vontade para executar os programas no Codepen ou em algum editor de código de sua preferência.

Redes sociais

Paulo Silveira

- Instagram
- Twitter

- Linkedin

Rafaella Ballerini

- Instagram
- Linkedin
- Youtube

Guilherme Lima

- Instagram
- Youtube
- Linkedin

Juliana Amoasei

- Linkedin

Sumário

Versão: 26.1.7

Capítulo 1

VARIÁVEIS, OPERAÇÕES E MÉDIA

1.1 OBJETIVO DA AULA

Nesta primeira aula da **Imersão Dev**, vamos criar uma calculadora de média e aprender o que são variáveis e como realizar operações! O código da aula inicial para você acompanhar está aqui:

https://codepen.io/imersao-dev/pen/38cf7906dcec352d8dbd0f8c4c7c1b10

Não se esqueça de fazer o fork desse projeto para a sua conta, e de marcar a hashtag da #imersaodev e #alura.

Neste programa, focamos nas primeiras ferramentas principais de qualquer linguagem de programação: variáveis, operadores, arredondamento e console.

O legal desse projeto e os demais, é que todos podem ser personalizados da forma que você quiser, alterando as cores, a fonte, o texto e assim por diante, permitindo que você abuse da sua criatividade. Para isso, vamos utilizar uma ferramenta da internet, que é o CodePen.

Sendo assim, você não precisa se preocupar em baixar e instalar linguagens ou editores de texto. Você pode se perguntar se vai aprender a programar de fato ou se só veremos teoria de lógica da programação, e sim, você vai aprender a programar! Não somente isso, eu já quero te chamar de programador ou programadora desde esse primeiro projeto.

1.2 PASSO A PASSO

Antes de começarmos a escrever qualquer linha de código, temos que pensar no caminho que o programa deve percorrer e quais informações ele precisa para ser executado corretamente.

O que o programa deve fazer? Realizar o cálculo da média

Ok, mas como isso é feito?

A média de um aluno durante é definida pela soma das notas dos 4 bimestres e divida por 4. Por exemplo:

- nota do primeiro bimestre foi 9
- nota do segundo bimestre foi 5
- nota do terceiro bimestre foi 10
- nota do quarto bimestre 8

Vamos somar todas as notas:

9 + 5 + 10 + 8 = 32

Agora vamos dividir o valor da soma pela quantidade de notas e descobrir a média final:

32 / 4 = 8

A média final neste exemplo é 8.

Como o programa vai receber essas informações?

Agora que já sabemos quais dados o programa precisa para funcionar, como vamos guardar o valor da nota de cada bimestre, realizar a operação e exibir a nota final?

1.3 VARIÁVEIS

Para guardar as notas de cada bimestre na memória do computador, vamos criar variáveis. Na programação, uma variável é capaz de guardar um valor, uma palavra ou expressão. Mas como criamos uma variável?

Para criar uma variável no Javascript, precisamos de uma palavra especial chamada **var**. Por exemplo, para armazenar meu nome, observe o código abaixo:

```
var nome = "Guilherme"
```

- O **var** indica que queremos criar uma variável;
- a palavra **nome** é o nome da variável;
- o sinal de igual indica que vamos atribuir a variável nome um valor;
- a palavra "Guilherme" será o valor da variável `nome` .

É muito importante sempre lembrar de “guardar” todos os dados que queremos usar em nossos programas e, para isso, usamos variáveis. Variáveis são como pequenos espaços de memória onde guardamos dados, como por exemplo um número

ou um texto. Criamos com a palavra-chave `var` uma variável chamada `nome` que vai "guardar" a informação que o usuário vai passar para o programa, nesse caso, a palavra `Guilherme`.

> O nome da variável é muito importante! É através dele que o programa consegue identificar e localizar todos os dados que "guardamos" para serem utilizados pelo programa, e que podem ser muitos!

Alguns detalhes que sempre temos que lembrar sobre variáveis:

- Podemos dar o nome que quisermos para as variáveis, porém o JavaScript tem algumas regras: não comece com letra maiúscula nem com números, não use espaços, caracteres especiais e nem acentos. Escolha um nome que descreva o que a variável vai "guardar"! O padrão de nomes do JavaScript é `nomeDaVariavel`, usando maiúsculas para diferenciar as palavras (mas nunca no início da palavra) é chamado de CamelCase.

Certo, mas como posso visualizar o valor armazenado na variável `nome`?

1.4 CONSOLE

Durante o desenvolvimento, podemos testar o valor das variáveis com o `console.log`, para exibir uma mensagem ou informação no console do navegador.

```
console.log("Bem vindo " + nome)
```

Ao executar o programa, nossa saída será "Bem vindo Guilherme".

Sabendo disso, podemos criar as notas de cada bimestre, como mostra o código abaixo:

```
var notaDoPrimeiroBimestre = 9
var notaDoSegundoBimestre = 7
var notaDoTerceiroBimestre = 4
var notaDoQuartoBimestre = 2
```

1.5 REALIZANDO OPERAÇÕES MATEMÁTICAS

Agora que temos todas as notas armazenadas, podemos somar todas as notas e realizar a divisão. Mas precisamos guardar a nota final da média, ou seja, precisamos de uma variável.

```
var notaFinal = notaDoPrimeiroBimestre + notaDoSegundoBimestre +
notaDoTerceiroBimestre + notaDoQuartoBimestre / 4
console.log(notaFinal)
```

> O sinal de divisão é representado pela barra invertida `/` . Já a multiplicação é representada pelo asterisco `*` .

Vamos executar o programa e ver a saída no console? "20.5"??? Algo está errado.

Isso acontece por conta da ordem dos operadores da matemática.É realizado primeiro a divisão e posteriormente a soma, como ilustra o exemplo abaixo:

- 2 dividido por 4 é 0.5
- depois soma de 9 + 7 + 4 que é igual a 20
- 20 mais 0.5 é 20.5

Certo, mas podemos resolver esse problema? Queremos realizar primeiro a soma de todas as notas e posteriormente a divisão por 4. Para isso, podemos incluir o sinal de parênteses entre as notas de cada bimestre:

```
var notaFinal = (notaDoPrimeiroBimestre + notaDoSegundoBimestre +
 notaDoTerceiroBimestre + notaDoQuartoBimestre) / 4
```

Agora sim! Ao executar o programa a saída no console é "5.5" porém podemos melhorar ainda mais nosso programa arredondando o valor da nota final ou limitando a quantidade de números decimais.

- O JavaScript tem muitos trechos de código prontos para utilizarmos em nossos programas e o arredondamento é um deles: `variavel.toFixed(2)` . Estes códigos estão sempre escritos em inglês e devem ser usados exatamente como estão nos exemplos para funcionar — não coloque pontos nem espaços onde não tem, não troque maiúsculas por minúsculas e não esqueça dos parênteses nos lugares certos, pois esses pequenos detalhes fazem com que o JavaScript “entenda” o que está sendo escrito. Com prática e estudo você vai entender melhor para que servem e o que significam os pontos, parênteses e tudo mais.

Vamos deixar apenas uma casa decimal utilizando o `toFixed(1)` . Para fixarmos 2 casa, o código seria `toFixed(2)` e assim por diante:

```
var notaFixada = notaFinal.toFixed(1)
console.log(notaFixada)
```

1.6 CÓDIGO JAVASCRIPT COMPLETO DESTA AULA

```
var nome = "Guilherme"

var notaDoPrimeiroBimestre = 9
var notaDoSegundoBimestre = 7
var notaDoTerceiroBimestre = 4
var notaDoQuartoBimestre = 2

var notaFinal = (notaDoPrimeiroBimestre + notaDoSegundoBimestre +
 notaDoTerceiroBimestre + notaDoQuartoBimestre) / 4

var notaFixada = notaFinal.toFixed(1)

console.log("Bem vindo " + nome)
console.log(notaFixada)
```

Tudo certo? Então é hora de praticar!

1.7 RESUMO

Vamos repassar todos os passos do programa:

- Variáveis, manipulação dos valores armazenados e a memória do computador;
- Tipos de variáveis, como texto e inteiro;
- Fixando a quantidade de casas decimais com a função toFixed().

Clique neste link para acessar o código completo no Codepen.

Capítulo 2

CONVERSOR DE MOEDAS

2.1 OBJETIVO DA AULA

Nesta segunda aula da **Imersão Dev**, o objetivo é criar um programa que peça para o usuário fornecer um valor em real, e a partir desse valor o programa vai exibir na tela o equivalente em dolar. Neste programa, vamos aprender como o Javascript pode interagir com o HTML através de botões, valores decimais e conversão de tipo.

O código da aula inicial para você acompanhar está aqui:

https://codepen.io/imersao-dev/pen/zYNOZRX

Não se esqueça de fazer o fork desse projeto para a sua conta, e de marcar a hashtag da #imersaodev e #alura.

Neste programa, vamos aprender como o Javascript pode interagir com o HTML através de botões, valores decimais e conversão de tipo.

O legal desse projeto e os demais, é que todos podem ser personalizados da forma que você quiser, alterando as cores, a fonte, o texto e assim por diante, permitindo que você abuse da sua criatividade.

2.2 PASSO A PASSO

Antes de começarmos a escrever qualquer linha de código, temos que pensar no caminho que o programa deve percorrer e quais informações ele precisa para ser executado corretamente.

1. O que o programa deve fazer? Converter real para dolar Ok, mas como isso é feito? Se fôssemos converter 1 real para dolar "na mão", faríamos a seguinte conta: 1 * 5 = 5.0 (supondo que a conversão seja 5 para 1)

Onde `1` é o valor em `reais` que queremos converter e `5` é a taxa média do dólar no momento em que escrevemos este texto. Multiplicando um valor pelo outro, temos o resultado em reais: `5.0`.

1. Como o programa vai receber essas informações? Podemos concluir, a partir do que vimos no ponto 1, que os dados que o programa precisa para funcionar são:

2. o valor em dólar que queremos converter (será fornecido pelo usuário);

3. a taxa do dólar que será utilizada na conversão (que nós vamos fornecer ao programa no momento em que escrevemos o código);
4. exibir "resultado" da conversão. Após executar todo o código, o programa deve exibir ao usuário um número que representa o resultado da operação valor em dólar * taxa de câmbio.

Interação entre HTML e Javascript

Já que precisamos receber do usuário o valor em real que ele quer converter, a primeira coisa a fazer é "pegar" esse valor de alguma forma. Vamos utilizar o campo, também chamado de input presente na linha 10 do *HTML*:

```
<input type="number" id="valor" size="2" /><br>
```

Essa linha cria para nós aquela caixa branca onde o usuário vai digitar um valor. Além disso, na linha 11 do *HTML* temos logo abaixo um botão `converter` logo abaixo:

```
<button type="submit" onclick="Converter()">Converter</button>
```

E é justamente aqui que a mágica vai acontecer!

Como o programa vai receber o valor do HTML?

Ao clicar no botão, recebemos uma mensagem de erro no console que diz:

```
Uncaught ReferenceError: Converter is not defined
```

Numa tradução direta para o português, seria algo como: `Erro de referência: Converter não está definido` . Certo, mas o que isso quer dizer?

Isso significa que ao clicar no botão, vamos automaticamente chamar uma função do Javascript chamada `Converter` . Recebemos um erro, porque essa função ainda não existe. Vamos criar essa função e exibir uma mensagem de `clicou` no console sempre que o botão for pressionado?

```
function Converter() {
    console.log("clicou")
}
```

Sempre que clicamos no botão, podemos ver a mensagem `clicou` no console.

> Repare que uma função possui a palavra `function`, seguido do nome da função, nesse caso `Converter`. Ao lado do nome da função temos dois parênteses `()` e uma chaves `{`. Dentro das chaves, escrevemos todo o código que queremos executar e fechamos a chaves `}` ao final.

document.getElementById

Existe uma função no Javascript que podemos pegar o valor do HTML conhecido por `document.getElementById`.

Observe que o campo input no *HTML* possui um `id="valor"` e é justamente esse id que precisamos passar para esta função:

```
function Converter() {
  var valorElemento = document.getElementById("valor");
  console.log("valorElemento")
}
```

Ao executar o programa, nossa saída no console não é bem aquela que esperávamos:

```
<input type="number" id="valor" size="2">
```

Isso acontece pois a função retorna a referência do elemento através do seu ID. Porém, nós queremos o valor e não toda referência do campo. Para isso, podemos usar a palavra **value**, como mostra o código abaixo:

```
function Converter() {
  var valorElemento = document.getElementById("valor");
  var valor = valorElemento.value;
  console.log("valor")
}
```

Agora sim, temos o valor do campo armazenado na variável `valor` no Javascript.

Tipo texto e tipo número

Agora que já temos o valor em dólar, vamos esbarrar em um probleminha muito comum e característico da programação, que é a distinção entre textos e números. Por padrão, quando pegamos um valor com input, o JavaScript sempre interpreta essa informação como texto, não importa se o usuário escreve `2`. Essa é uma questão relacionada a tipos de dados em computação, que neste momento não vamos detalhar muito, mas você pode ler mais clicando neste link.

> Para saber se a variável armazena um tipo texto, também conhecido por string ou numérico, conhecido com Float para números decimais ou Int para números inteiros, você pode usar a função `typeof` e o nome da variável como ilustra o código abaixo.

```
console.log(typeof valor)
```

Outra dica: sempre que um número estiver entre aspas simples ou duplas, ele é do tipo texto!

Conversão de tipo

Já que é assim, se o usuário inserir o valor 1 para converter, o JavaScript pode ficar confuso, afinal de contas ele não sabe que ”10” é um número, e como é que ele vai fazer essa multiplicação?

Felizmente, como já dissemos, essa questão texto x número é comum. E o próprio JavaScript tem uma palavra-chave pronta para fazer essa conversão. Então, antes de qualquer outra coisa, devemos pegar o valor que está guardado na variável valorEmDolarTexto e transformá-lo de texto para número:

```
function Converter() {
  var valorElemento = document.getElementById("valor");
  var valor = valorElemento.value;
  var valorEmDolarNumerico = parseFloat(valor);
  console.log(valorEmDolarNumerico)
}
```

A palavra-chave `parseFloat()` é essa ferramenta pronta do JavaScript para converter o que está entre os parênteses de texto para número. No caso, o que vai ser convertido é o `valor` que o usuário vai inserir no programa através do input e que está guardado na variável `valor`.

Como sempre, precisamos guardar em uma variável esse valor convertido, para que o programa possa usá-lo. Já que agora o valor é um número, chamamos essa nova variável de `valorEmDolarNumerico`.

Já podemos fazer a conta? E a taxa do dólar? Há algumas formas de passarmos para o programa a informação da taxa do dólar. Para esta primeira versão do nosso conversor vamos dizer ao JavaScript exatamente qual é a taxa e utilizá-la na conta de multiplicação:

Realizando a conversão

Sem esquecer de "guardar" o valor da multiplicação em uma variável, afinal de contas precisamos dele para exibir ao usuário! Neste caso, a taxa utilizada é `5`.

```
function Converter() {
  var valorElemento = document.getElementById("valor");
  var valor = valorElemento.value;
  var valorEmDolarNumerico = parseFloat(valor);

  var valorEmReal = valorEmDolarNumerico * 5;
  console.log(valorEmReal);
}
```

Estamos guardando o resultado desta conta na variável valorEmReal.

Uma última coisa antes de exibir o resultado para o usuário! Quando fazemos contas que envolvem valores "quebrados" — ou seja, com um ou mais dígitos depois da vírgula — pode acontecer a tal da dízima periódica, ou o número após a vírgula ficar com muitas casas decimais. Então vamos usar uma outra palavra-chave que o JavaScript já deixou pronta para esses casos:

Exibindo o resultado na tela

Agora já está tudo pronto para exibir o resultado para o usuário.

Da mesma forma que pegamos o valor a ser convertido do *input* o HTML para o Javascrip, nós podemos manipular o valor interno de algum elemento da página para exibir o valor convertido, sem a necessidade de exibir uma mensagem no console, e sim na tela.

Na linha `12` do HTML temos o seguinte código:

```
<h2 id="valorConvertido"></h2>
```

Esse elemento foi criado para incluirmos o valor convertido. Para pegar esse elemento, vamos usar o `document.getElementById` indicando o ID do elemento e atribuir numa nova variável chamada `elementoValorConvertido` :

```
var elementoValorConvertido = document.getElementById("valorConve
rtido");
```

Agora, vamos criar uma mensagem que queremos exibir na tela e armazenar em uma nova variável chamada `valorConvertido` :

```
var valorConvertido = "O resultado em real é R$ " + valorEmReal;
```

Para finalizar, precisamos alterar o conteúdo interno do `elementoValorConvertido` . Para isso, existe um método chamado `innerHTML` , que numa tradução direta, seria algo como interior ou interno do elemento HTML e vamos atribuir o valor convertido:

```
elementoValorConvertido.innerHTML = valorConvertido;
```

Você pode ler mais sobre o innerHTML clicando neste link.

2.3 CÓDIGO JAVASCRIPT COMPLETO DESTA AULA

```
function Converter() {
  var valorElemento = document.getElementById("valor");
  var valor = valorElemento.value;
  var valorEmDolarNumerico = parseFloat(valor);

  var valorEmReal = valorEmDolarNumerico * 5;
  console.log(valorEmReal);
  var elementoValorConvertido = document.getElementById("valorCon
vertido");
  var valorConvertido = "O resultado em real é R$ " + valorEmReal
;
  elementoValorConvertido.innerHTML = valorConvertido;
}
```

Tudo certo? Então é hora de praticar!

2.4 RESUMO

Vamos repassar todos os passos do programa:

- Interagir com usuário para receber os preços a serem convertidos;
- Criar a lógica para saber como converter;
- Utilizar funções que interagem com a tela;
- Exibir o resultado na tela com innerHTML.

Clique neste link para acessar o código completo no Codepen.

CAPÍTULO 3

MENTALISTA

3.1 OBJETIVO DA AULA

Nesta aula da **Imersão Dev**, vamos criar uma jogo de adivinhação, onde o programa escolhe um número aleatório e nós vamos tentar acertar. O projeto inicial se encontra no link abaixo:

O código da aula inicial para você acompanhar está aqui:

https://codepen.io/imersao-dev/pen/vYgBwoj

Não se esqueça de fazer o fork desse projeto para a sua conta, e de marcar a hashtag da #imersaodev e #alura.

Neste programa, focamos em laços condicionais, operações booleanas, if e else.

3.2 PASSO A PASSO

Para iniciar esse programa, vamos criar uma variável chamada `numeroSecreto` e atribuir um número aleatório entre 0 e 10. Vamos utilizar uma função chamada `Math.random` . Vamos executar essa função no console e ver que sempre gera uma sequência de números diferentes.

```
> Math.random()
```

```
<- 0.20982489919137293

> Math.random()
<- 0.43822764751936005

> Math.random()
<- 0.2983069465358348

> Math.random()
<- 0.6899012983471291
```

Math.random

Segundo a documentação, a função Math.random retorna um número pseudo-aleatório no intervalo entre `0` e `1`, sendo que o valor 1 é **exclusivo**. Como queremos um número inteiro entre `0` e 10, vamos multiplicar a função `Math.random` por `11`:

```
> Math.random() * 11
<- 3.029855123323566

> Math.random() * 11
<- 4.808136849238727
```

Para finalizar, vamos utilizar a função `parseInt` que, basicamente, vai fazer a conversão para número inteiro, removendo os números depois do ponto.

```
var numeroSecreto = parseInt(Math.random() * 11);
```

> Para ver o número secreto, você pode usar o código `console.log(numeroSecreto)`. Podemos usar o console.log para realizar diversas verificações.

Pegando o valor do chute

Na linha 18 do código HTML, temos o seguinte input:

```
<input type="number" id="valor" />
```

Como vimos no capítulo anterior, podemos pegar o valor deste input com `document.getElementById` passando o ID `valor`. Porém quando faremos isso? Faremos isso assim que o botão chutar for clicado. Veja o código do botão no HTML na linha 19:

```
<button type="submit" onclick="Chutar()">Chutar</button>
```

Vamos criar uma função `Chutar` e atribuir a uma variável o valor do input:

```
function Chutar() {
  var chute = document.getElementById("valor").value;
}
```

parseInt

Por padrão, todos os campos digitados nos campos `input` são do tipo texto. Precisamos converter esse tipo para número no Javascript. Para isso, vamos utilizar a função `parseInt`, como mostra o código abaixo:

```
var chute = parseInt(document.getElementById("valor").value);
```

> No capítulo anterior, usamos o parseFloat para converter o valor de uma variável para decimal e nesta capítulo, usamos o parseInt para converter o valor de uma variável para o tipo inteiro.

Vamos ver o chute no console?

```
function Chutar() {
  var elementoResultado = document.getElementById("resultado");
  var chute = parseInt(document.getElementById("valor").value);
  console.log(chute);
}
```

Condicional if

Agora precisamos saber **se** a variável `chute` é igual ao `numeroSecreto`. No JavaScript e em muitas outras linguagens, essa condição **se** é escrita no idioma inglês **if** (que significa "se" em português).

```
if
```

Para criar a condição que verifica **se** a variável `chute` é igual ao `numeroSecreto`, vamos incluir o sinal de parênteses.

```
if ()
```

Dentro dos parênteses, vamos incluir nossa condição, e aqui tem um detalhe importante. Comparar **se** a variável `chute` **é igual** ao `numeroSecreto` não será feita com apenas um sinal de igual, e sim com dois (`==`)..

```
if (chute == numeroSecreto)
```

Caso isso seja verdadeiro, isto é, caso a variável `chute` seja **é igual** ao `numeroSecreto`, queremos fazer alguma coisa. Esse fazer alguma coisa será indicado pelos sinais de abrir e fechar chaves (`{ }`).

```
if (chute == numeroSecreto) {

}
```

Certo, e se o chute for igual ao número escolhido?

Isso acontece porque quando usamos apenas um sinal de igual, estamos atribuindo um valor a uma variável, e não é o que queremos fazer. Queremos comparar se o valor da variável operação é igual a um número, neste caso, 1.

> Dica: com um sinal de igual (`=`), estamos atribuindo um valor a uma variável (`var nome = "Maria"` , por exemplo). Com dois sinais de igual (`==`), estamos comparando dois valores.

```
if (chute == numeroSecreto) {
    elementoResultado.innerHTML = "Você acertou";
}
```

Exibindo o resultado na tela

Na linha 20 do código HTML, existe um `h2` com ID `resultado` , criado para mostrar o resultado na tela:

```
<h2 class="resultado" id="resultado"></h2>
```

Podemos alterar o conteúdo deste elemento com `innerHTML` , mas primeiro vamos pegar esse valor com Javascript. Dentro da nossa função `Chutar` , vamos incluir a seguinte linha:

```
var elementoResultado = document.getElementById("resultado");
```

Nesse momento, nossa função está assim:

```
function Chutar() {
  var elementoResultado = document.getElementById("resultado");
  var chute = parseInt(document.getElementById("valor").value);
  console.log(chute);
  if (chute == numeroSecreto) {
```

```
  }
}
```

Vamos alterar o interior do elemento resultado para exibir uma mensagem informando que a pessoa acertou o chute.

```
function Chutar() {
  var elementoResultado = document.getElementById("resultado");
  var chute = parseInt(document.getElementById("valor").value);
  console.log(chute);
  if (chute == numeroSecreto) {
    elementoResultado.innerHTML = "Você acertou";
  }
}
```

Vamos testar?

> Dica: você pode alterar o valor da variável `numeroSecreto` para 5 por exemplo, e chutar o número 5 para validar a condição onde o chute é igual ao número secreto e ver a mensagem "Você acertou" na tela.

Condicional else if

Vamos criar mais uma condição para verificar se o chutepossui um valor válido, ou seja, um número maior `10` **ou** menor que `0` usando o comando `else if` e a condição entre parênteses (não esqueça os bigodes, ou seja, de abrir e fechar as chaves). No Javascript, podemos usar o operador **ou** através do sinal `||` (barra barra).

```
else if (chute > 10 || chute < 0) {
    elementoResultado.innerHTML = "Você deve digitar um número de
 0 a 10";
```

```
}
```

Nossa função está assim:

```
function Chutar() {
  var elementoResultado = document.getElementById("resultado");
  var chute = parseInt(document.getElementById("valor").value);
  console.log(chute);
  if (chute == numeroSecreto) {
    elementoResultado.innerHTML = "Você acertou";
  } else if (chute > 10 || chute < 0) {
    elementoResultado.innerHTML = "Você deve digitar um número de
 0 a 10";
  }
}
```

Testando... Sempre que o chute for um valor inválido, informamos que o usuário deve digitar um número de 0 a 10.

Condicional else

E quando o chute for um número válido, porém errado? Precisamos informar o usuário. Para isso, vamos utilizar a condição `else` :

```
else {
    elementoResultado.innerHTML = "Errou";
}
```

Se o chute não for igual ao número secreto, e for um valor válido, a condição `else` será executada.

HTML: não conseguimos incluir no campo do chute uma palavra ou letra, pois no HTML especificamos que o input é do tipo `number`.

3.3 CÓDIGO JAVASCRIPT COMPLETO DESTA AULA

```
var numeroSecreto = parseInt(Math.random() * 11);

function Chutar() {
  var elementoResultado = document.getElementById("resultado");
  var chute = parseInt(document.getElementById("valor").value);
  console.log(chute);
  if (chute == numeroSecreto) {
    elementoResultado.innerHTML = "Você acertou";
  } else if (chute > 10 || chute < 0) {
    elementoResultado.innerHTML = "Você deve digitar um número de
 0 a 10";
  } else {
    elementoResultado.innerHTML = "Errou";
  }
}
```

Tudo certo? Então é hora de praticar!

3.4 RESUMO

Vamos repassar todos os passos do programa:

- Realizar o fork do projeto com template inicial;
- Fazer testes utilizando `console.log()`;
- Criar a lógica por trás do "chute" com `if`, `else if` e `else`;

- Utilizar a função `Math.random()` do JavaScript para gerar números aleatórios.

Clique neste link para acessar o código completo no Codepen.

Capítulo 4

ALURAFLIX

4.1 OBJETIVO DA AULA

Vamos criar um programa que permite você listar os alguns filmes em formato de imagem com a plataforma Aluraflix! **Nesse projeto, focamos em arrays**.

O código da aula inicial para você acompanhar está aqui:

https://codepen.io/imersao-dev/pen/15c30c8f7a2a723b9cbcc0943995be3f

Não se esqueça de fazer o fork desse projeto para a sua conta, e de marcar a hashtag da #imersaodev e #alura.

Nesta aula vamos aprender sobre array, um tipo de lista de elementos, assim como algumas ferramentas para alterarmos e trabalharmos com estas listas. Depois de criarmos uma lista, vamos exibir os pôsters de todos os nossos filmes preferidos na tela do navegador..

4.2 PASSO A PASSO

Vamos começar criando três variáveis para armazenar os primeiros valores, que no caso serão nomes de um filme:

```
var filme1 = “Yesterday”
var filme2 = “A chegada”
var filme3 = "Escola do Rock"
```

Agora, vamos imprimi-los no console da seguinte maneira:

```
console.log(filme1)
console.log(filme2)
console.log(filme3)
```

Mesmo que dê tudo certo, é possível perceber que quanto mais filmes adicionamos no nosso “catálogo” do Aluraflix, mais linhas de código temos que usar para armazenar e imprimir as variáveis. Sendo assim, podemos utilizar uma **variável especial, chamada array**. Ele nada mais é do que uma **variável que pode armazenar um conjunto de outras variáveis.** Vamos ver na prática como isso funciona, declarando o array e imprimindo no console:

```
var filmes = [“Yesterday”, “A chegada", "Escola do Rock]
console.log(filmes)
```

> Dica: existe outra forma de armazenar diferentes valores dentro de um array, pois nem sempre já temos todos eles definidos no momento em que os declaramos. Nestes casos, podemos declarar o array vazio e ir adicionando cada valor com a função `push` .

```
var filmes = []
filmes.push(“Yesterday”)
filmes.push(“A chegada”)
filmes.push(“Escola do Rock”)

console.log(filmes)
```

Mas e se quisermos apenas imprimir um valor específico desse

array? O que devemos fazer?

Para isso, podemos chamar esse valor pelo seu índice. **É importante destacar que o primeiro elemento sempre terá índice `0`**, o segundo índice terá `1`, o terceiro índice, `2`, e assim por diante. Eles servem como identificadores únicos de cada valor dentro do array, então você sempre saberá qual valor será impresso, da seguinte forma:

```
console.log(filmes[0]) // "Yesterday"
console.log(filmes[1]) // "A chegada"
console.log(filmes[2]) // "Escola do Rock"
```

Porém, ainda caímos no mesmo problema de antes. Se decidirmos adicionar 200 filmes, precisaremos de 200 linhas de `console.log()` para imprimir cada um deles. **Sendo assim, existe um laço de repetição (*loop*) que podemos utilizar, diferente do `while` visto na aula anterior, o `for`.** A sintaxe dele é a seguinte:

```
for (var i = 0; i < 3; i++) {
    //aqui serão escritos os comandos a serem executados dentro d
o laço de repetição
}
```

Não se assuste, vamos entender exatamente o que está acontecendo em cada parte desse código agora.

É possível perceber que existem três divisões dentro do parênteses do `for`, separados por ponto e vírgula. A primeira delas, `var i = 0`, está declarando uma nova variável `i` com o valor 0. **Essa será a variável utilizada como contadora do laço de repetição.**

Já na segunda divisão, temos `i < 3`. Essa é a condição para

que o programa entre no loop e execute os comandos dentro dele. Por fim, na última divisão temos `i++`, que é `i = i + 1` escrito de forma **mais compacta e simples**.

Então, agora podemos sintetizar melhor o que está acontecendo: o programa cria uma nova variável `i`, e atribui o valor `0` a ela. Ele verifica se a variável `i` é menor que 3. Caso seja, todos os comandos dentro das chaves serão executados. No final dessa execução, o valor `i` é aumentado por 1, de 0 passando para 1 e assim por diante, até ele deixar de ser menor que 3 e deixar de entrar no laço de repetição.

Agora que entendemos o `for`, podemos adicionar um comando dentro dele para vermos na prática.

```
for (var i = 0; i < 3; i++) {
    console.log(i)
}
```

Estamos imprimindo o valor de `i` cada vez que o programa entra no loop. Sendo assim, podemos perceber que ele vai diminuindo essa quantidade de vezes com o passar do tempo, a cada vez que entra no laço.

Vamos testar utilizar o `for` para imprimir os valores do array sem precisarmos escrever várias linhas de `console.log()`, assim:

```
for (var i = 0; i < 3; i++) {
    console.log(filmes[i])
}
```

Cada vez que entramos no loop, imprimimos o valor do array com o índice igual ao valor de `i`, percorrendo todos os elementos dentro dele.

Porém, ainda podemos melhorar o nosso código, pois dessa forma teremos sempre que saber a quantidade de elementos a serem colocados dentro da condição para compararmos com a variável `i`.

Vamos utilizar o `.length`, que vai retornar o número de elementos que temos dentro de um array, da seguinte forma:

```
for (var i = 0; i < filmes.length; i++) {
    console.log(filmes[i])
}
```

Tudo certo até aqui? Agora podemos apagar isso tudo que vimos e começar a implementar o projeto dessa aula!

A ideia agora é, em vez de colocarmos os nomes dos filmes para aparecerem na tela, mostrarmos as imagens de cada um deles.

Para isso, podemos buscar as capas dos filmes que queremos na internet e copiarmos o endereço dessas imagens. Isso pode ser feito clicando com o botão direito do mouse e selecionando a opção "Copiar endereço da imagem".

Podemos colar este endereço copiado dentro do nosso novo array:

```
var filmes = [“https://m.media-amazon.com/images/M/MV5BZDgzNzdmNj
EtMDAwMC00M2FiLTlkMTEtMDE0MDIyNTEwYmJlXkEyXkFqcGdeQXVyMjY3MjUzNDk
@._V1_UY268_CR12,0,182,268_AL_.jpg”]
```

Parece um pouco assustador, mas o que você deve entender olhando para esse link, é que ele tem no final um `.jpg`, indicando que trata-se realmente de uma imagem.

Podemos agora adicionar as imagens dos outros filmes dentro

do nosso array:

```
var listaFilmes = [“"https://upload.wikimedia.org/wikipedia/pt/7/
79/Yesterday_%282019%29_poster.jpg",
  "https://1.bp.blogspot.com/-ImZPRqLsluE/WFK156_6pNI/AAAAAAAAYBY
/0lEhNRF5wfQdLfr6hpT57_Jt2eBrE9H5wCLcB/s1600/arrival-kartoun-dese
rt.jpg",
  "https://br.web.img3.acsta.net/c_310_420/medias/nmedia/18/91/90
/98/20169244.jpg"”]
```

Já aprendemos a imprimir valores de um array com o `for`, então vamos fazer isso:

```
for (var i = 0; i < listaFilmes.length; i++){
    //comando para imprimir os filmes
}
```

Dessa vez, não vamos imprimir os valores dentro do nosso console com `console.log()`, e sim usar um comando para que as imagens sejam exibidas na tela, com `document.write()`. O nosso programa ficará assim:

```
for (var i = 0; i < listaFilmes.length; i++){
    document.write(listaFilmes[i])
}
```

Ok, acho que não ficou da forma que queríamos, certo? Isso porque queremos que as imagens apareçam na tela, e não os links escritos em forma de texto.

Para resolvermos isso, precisaremos de um pouquinho de HTML. Criaremos elementos na nossa página que indiquem que aquilo é uma imagem, e não mais um texto. Colocaremos uma nova tag `<img>` dentro do nosso `document.write()`:

```
for (var i = 0; i < listaFilmes.length; i++){
    document.write(“<img src=” + listaFilmes[i] + “>”)
}
```

O `src` dentro da nossa tag `img` é a propriedade que recebe o endereço da imagem, que no caso é o nosso valor dentro do array.

> Legal né? O interessante agora é você colocar vários outros filmes, com a intenção de criar realmente um catálogo com os que mais gosta, ou os que indica para alguém assistir!

Tudo certo? Então é hora de praticar!

4.3 RESUMO

Vamos repassar todos os passos do programa:

- Criar uma primeira array de filmes usando a sintaxe [];
- Utilizar o método filmes.push("Nome Do Filme") para inserir um novo elemento na lista (ou seja, um novo filme na array);
- Descobrir a quantidade de elementos em uma array com o método array.length;
- Selecionar elementos de uma array utilizando a sintaxe array[número], lembrando sempre que o primeiro índice começa com zero, ou seja, array[0] para o primeiro elemento;
- Utilizar a instrução for para iterar, ou seja, percorrer todos os elementos de uma array;
- Criar uma array com imagens de pôsters de alguns filmes que gostamos;
- Montar a lógica do programa que vai iterar esta array de

filmes e exibir cada um deles na tela, integrando o for do JavaScript com a tag `img` do HTML.

Clique neste link para acessar o código completo no Codepen.

Capítulo 5

ALURAFLIX, BOTÕES VALIDAÇÕES E FUNÇÕES

5.1 OBJETIVOS DA AULA

Criar um programa que armazene a imagem de um filme, caso a entrada seja uma imagem. Caso contrário, informar ao usuário que a imagem é inválida. Fazemos essa verificação pedindo ao JavaScript que localize no fim do link o trecho de texto `.jpg` (nesse caso, não serão aceitas imagens em .png). Criar um botão que execute uma função para validar o link da imagem do filme.

O código da aula inicial para você acompanhar está aqui:

https://codepen.io/imersao-dev/pen/15c30c8f7a2a723b9cbcc0943995be3f

Não se esqueça de fazer o fork desse projeto para a sua conta, e de marcar a hashtag da #imersaodev e #alura.

Neste programa, focaremos em aprender e praticar funções e como vincular o JavaScript a um botão no HTML para executar uma função no momento do clique.

5.2 PASSO A PASSO

Primeiro vamos pensar no que o programa deve fazer: uma listagem de filmes composta por imagens (de pôsteres, personagens, etc). Quem monta a lista é o próprio usuário, então precisamos prever **de que forma o usuário vai inserir os itens da lista no programa**. Para isso, é bem comum utilizarmos um campo no HTML e "unir" essa ponta (a parte que o usuário interage) com o JavaScript (onde é feito o processamento do programa).

A segunda parte é utilizar o JavaScript para modificar o HTML que exibe as informações da página. Mas — esse é um detalhe importante — essa etapa só deve acontecer depois que o usuário inserir uma informação no campo de input (no caso, a imagem de um filme). Se deixarmos esse código solto, ele não vai esperar o usuário inserir o dado necessário e vai executar com erro. É aí que entram as funções.

Função para adicionar filme na lista

Pensando sempre no fluxo do programa, não queremos que nada seja processado enquanto o usuário não clicar no botão para adicionar um novo filme - vamos considerar aqui que o usuário só vai clicar no botão depois de adicionar o endereço de uma imagem de filme no input.

Esse é um caso para uso de funções: existem muitas razões para organizarmos nosso código em funções, e um deles é justamente poder **controlar** quando um trecho de código é executado. No caso do nosso programa, se não colocarmos o código para adicionar um novo filme na lista dentro de uma função, o código seria executado logo no carregamento da página, e não daria nem

tempo do usuário preencher qualquer dado! O código seria executado com erro.

Então, a primeira coisa a fazer é criar uma função para que um certo trecho de código dentro dela só seja executado no clique do botão.

O código responsável por avisar o HTML que existe uma função para ser chamada no clique do botão é o atributo `onclick="adicionarFilme()"`:

```
<input type="text" id="filme" name="filme" placeholder="Insira en
dereço de imagem">
<button onClick="adicionarFilme()">Adicionar Filme</button>
```

Mas por enquanto essa função não existe. Vamos criá-la no arquivo `script.js`:

```
function adicionarFilme() {
 var campoFilmeFavorito = document.querySelector('#filme')
 var filmeFavorito = campoFilmeFavorito.value
 if (filmeFavorito.endsWith('.jpg')) {
     listarFilmesNaTela(filmeFavorito)
 } else {
     alert("Imagem inválida")
 }
 campoFilmeFavorito.value = ""
}
```

Esta função está executando os seguintes passos na lógica que imaginamos para o programa: 1 . Criando uma variável para encontrar o "local" no HTML (ou seja, no conteúdo exibido na tela) onde o usuário vai inserir o link para a imagem. Fazemos isso através do identificador `'#filme'` que está identificando o `input` no HTML (veja o código acima).

```
var campoFilmeFavorito = document.querySelector('#filme')
```

1. Usamos a palavra-chave `.value` para que o JavaScript consiga capturar o texto que foi inserido dentro do campo input.

```
var filmeFavorito = campoFilmeFavorito.value
```

1. Agora que já conseguimos capturar o link que o usuário inseriu no input, hora de validar se é um link de imagem válido, com final `.jpg` . Fazemos isso usando o condicional `if` e a ferramenta `.endsWith()` , que podemos traduzir literalmente para "termina com". Ou seja, se o texto (`string`) da variável que recebemos termina com o trecho `.jpg` .

```
 if (filmeFavorito.endsWith('.jpg')) {
     listarFilmesNaTela(filmeFavorito)
 } else {
     alert("Imagem inválida")
}
```

No código acima, passamos uma instrução para o caso de sucesso, ou seja, se o link de imagem for válido. Essa instrução é `listarFilmesNaTela(filmeFavorito)` . Se lermos o código novamente, vemos que `filmeFavorito` é a variável que "guarda" o link com a imagem do filme que o usuário inseriu na tela e pegamos lá do HTML. Mas o que essa variável está fazendo?

Aqui, vemos as funções novamente em ação. Não só podemos dizer exatamente quando um trecho de código vai ser executado, como podemos organizar melhor o programa, separando cada parte da lógica em uma função diferente, passando para cada uma delas as informações necessárias para funcionar. Essas informações são passadas entre parênteses, em forma de dados (números, strings, etc) ou variáveis — chamamos essas informações de

parâmetros ou **argumentos**.

```
listarFilmesNaTela(filmeFavorito)
```

Ainda não escrevemos a função `listarFilmesNaTela`, vamos fazer isso em seguida. Podemos executar funções dentro de outras funções!

1. Lembrando: temos que passar para o computador exatamente todos os passos do algoritmo. Então, para "limpar" o campo depois de clicar no botão, temos que avisar que o `.value` (que antes tinha o valor do link que o usuário inseriu) agora não tem mais valor de texto, ou o que chamamos também de "string vazia":

```
campoFilmeFavorito.value = ""
```

Função para exibir o filme na tela

De acordo com os passos do programa, depois de "pegarmos" o link da imagem, temos que inclui-la na tela sem apagar o que já está lá. Vamos escrever a função `listarFilmesNaTela`:

```
function listarFilmesNaTela(filme) {
 var listaFilmes = document.querySelector('#listaFilmes')
 var elementoFilme = "<img src=" + filme + ">"
 listaFilmes.innerHTML = listaFilmes.innerHTML + elementoFilme
}
```

1. De forma parecida com o que fizemos antes, vamos usar o identificador `id="listaFilmes"` para localizar a parte da tela (ou seja, do HTML) onde queremos adicionar a imagem do filme escolhido.

```
<div id="listaFilmes"></div>
```

No JavaScript, criamos uma variável para salvar as informações que existem na tela nesse momento:

```
var listaFilmes = document.querySelector('#listaFilmes')
```

1. Adicionando um filme novo à lista (ou incluindo o primeiro se a lista estiver vazia): aqui utilizamos o JavaScript para criar uma nova "tag" de imagem de HTML. Uma vez que o JavaScript inserir esta nova linha no HTML, este trecho de código será reconhecido e exibido como imagem.

```
var elementoFilme = "<img src=" + filme + ">"
```

Esta forma de escrever, misturando "strings" (ou seja, dados de texto) com variáveis, parece um pouco estranha. Porém é bastante comum utilizarmos formas similares a esta para atualizar e modificar elementos da tela (HTML) com JavaScript.

Se você se perguntou o que é a variável `filme` neste código, observe novamente a primeira linha da função `listarFilmesNaTela` : `filme` é o nome que damos ao **parâmetro** (ou **argumento**) da função. Ou seja, quando esta função for executada — ou "chamada", como costumamos dizer — a variável `filme` será substituída pela string que está sendo informada dentro do `if` da função anterior.

1. A última coisa a fazer é adicionar na tela a linha de HTML com a tag de imagem.

```
listaFilmes.innerHTML = listaFilmes.innerHTML + elementoFilme
```

Já havíamos criado antes a variável `listaFilmes` para salvar as tags de imagem que já existiam na tela. Agora estamos **reatribuindo** um valor a esta mesma variável: este novo valor é

composto do `innerHTML` (mais sobre esta palavra-chave abaixo) somado à nova "tag" de imagem que acabamos de criar e salvar na variável `elementoFilme`. Quando trabalhamos com JavaScript e usamos o operador de soma `+` com dados de texto, os dois textos vão ser unidos em um só.

Sobre o `innerHTML`

Quando utilizamos a palavra-chave `innerHTML`, normalmente estamos nos referindo a todo o conteúdo (*tags* e seus atributos) que é interno a determinado elemento HTML.

Ou, colocando de forma mais prática o uso que estamos fazendo no código do nosso programa, o `innerHTML` se refere a todo o conteúdo interno de determinado elemento. Por exemplo, o `innerHTML` do elemento identificado por `id="lista"`:

```
<div id="lista">

  <!-- inner HTML -->
  <p>item 1</p>
  <p>item 2</p>
  <p>item 3</p>
  <!-- fim do inner HTML -->

</div>
```

De acordo com o exemplo acima, vemos que `innerHTML` se refere a tudo que está entre entre as *tags* de abertura e fechamento do elemento `<div id="lista">`. Colocando de outra forma:

```
<div id="lista">innerHTML</div>
```

Caso queira saber mais, você pode consultar a documentação sobre `innerHTML` no MDN.

Tudo certo? Então é hora de praticar!

5.3 RESUMO

Vamos repassar todos os passos do programa:

- Sintaxe e criação de funções no JavaScript;
- Integrando funções criadas no JavaScript com o HTML que está sendo exibido na tela;
- Condicionando a execução (ou "chamada") de uma função a um clique em um botão na tela;
- Usando o JavaScript para acessar o que está sendo exibido na tela e pegar valores digitados pelo usuário com querySelector() e .value;
- Passar informações que as funções precisam para funcionar, através dos parâmetros;
- Utilizar o .endsWith() para verificar se um texto termina com determinados caracteres;
- Ver mais um exemplo de reatribuição de variável para "limpar" o texto do campo com "".

Clique neste link para acessar o código completo no Codepen.

Capítulo 6

TABELA DE CLASSIFICAÇÃO E OBJETOS NO JAVASCRIPT

6.1 OBJETIVO DA AULA

Criar um programa que mostre uma tabela informando as vitórias, empates, derrotas e a quantidade de pontos. Cada vitória soma 3 pontos e cada empate 1 ponto na tabela. Criar 3 botões chamados vitórias, empates e derrotas. Quando a vitória for clicada, adicionar 1 na quantidade de vitórias e calcular a quantidade de pontos com base nas regras acima. O mesmo com empate e derrota.

O código da aula inicial para você acompanhar está aqui:

https://codepen.io/imersao-dev/pen/c3ba80ae177fcded4c257015d4ce719c

Não se esqueça de fazer o fork desse projeto para a sua conta, e de marcar a hashtag da #imersaodev e #alura.

Neste programa, vamos aprender o que são objetos no Javascript

6.2 PASSO A PASSO

Quando criamos uma página com HTML e CSS, todo o conteúdo desta página não muda, a menos que troquemos seus valores no HTML. Isso pode ser ruim, pois, no nosso caso, quando uma vitória ou empate for alterado, não podemos esquecer de alterar também o valor dos pontos.

Sabendo disso, vamos realizar toda a manipulação dos dados desta tabela usando o Javascript. Quando uma vitória for computada, queremos adicionar o valor 1 ao número de vitória e recalcular os pontos do jogador ou jogadora. Mas como armazenar no Javascript o nome do jogador, o número de vitórias, empates, derrotas e os pontos?

Objetos no Javascript

Objetos são tipos de dados compostos: eles agregam vários valores em uma única unidade e nos permitem armazenar e recuperar esses valores por nome. Outra maneira de explicar isso é dizer que um objeto é uma coleção não ordenada de propriedades, cada uma delas com um nome e um valor. Os valores nomeados mantidos por um objeto podem ser valores como números e textos.

Ao observar os dados da tabela, observe que cada jogador possui um nome, a quantidade de vitórias, empates, derrotas e os pontos.

Nome	Vitórias	Empates	Derrotas	Pontos
Paulo	2	3	5	9
Rafa	3	2	1	11

Certo mais, como posso criar um objeto no Javascript que represente cada jogador?

Criando cada jogador como objeto no Javascript

Para criar cada jogador, vamos definir uma nova variável e atribuir o sinal de chaves `{ }`

```
var paulo = {}
```

O que queremos fazer agora é armazenar no objeto `paulo` seu nome, o número de vitórias, empates, derrotas e pontos. Para isso, vamos criar cada propriedade com seu `nome` , o sinal de dois pontos `:` e atribuir o valor de cada atributo separado por vírgula entre eles, como mostra o código abaixo:

```
var paulo = {
    nome: "Paulo",
    vitorias: 2,
    empates: 5,
    derrotas: 1,
    pontos: 0
}
```

> Dica: posso criar o objeto usando apenas uma linha também.

```
var paulo = {nome: "Paulo", vitorias: 2, empates: 5, derrotas: 1,
 pontos: 0}
```

Podemos criar a jogadora `rafa` , da mesma forma:

```
var rafa = {
    nome: "Rafa",
    vitorias: 3,
```

```
    empates: 5,
    derrotas: 2,
    pontos: 0
}
```

Em ambos jogadores, deixamos seus pontos zerados e vamos criar uma função que realiza o cálculo dos pontos. Mas como pegamos os valores de um objeto?

Atributos de um objeto

Para recuperar os valores de um objeto, podemos utilizar o `.` (ponto), seguido do nome do atributo que queremos ver. Vamos realizar alguns testes:

Para visualizar o número de vitórias da Rafa:

```
console.log(rafa.vitorias)
```

Para visualizar o número de empates ou o nome do Paulo:

```
console.log(paulo.empates)
console.log(paulo.nome)
```

Sendo assim, usamos o ponto para acessar os campos de um objeto.

Função para calcular pontos

Para alterar o valor de um campo em um objeto, podemos usar o sinal de igual `=` e atribuir um novo valor. Observe o exemplo a seguir:

```
console.log(paulo.vitorias)
paulo.empates += 1
console.log(paulo.vitorias)
```

Dica: podemos também atribuir 1 com o código `paulo.empates++`

Observe que o valor da vitória foi alterado. O que podemos fazer é criar uma função que realiza o cálculo dos pontos e atribui o valor na variável pontos. Para função conseguir identificar de quem são os pontos, vamos passar entre os parênteses o objeto jogador.

```
function calculaPontos(jogador) {}
```

Dentro da função, vamos criar uma variável chamada pontos e atribuir o número de vitórias do jogador multiplicado por 3 e somar com número de empates.

```
function calculaPontos(jogador) {
    var pontos = (jogador.vitorias * 3) + jogador.empates
}
```

Toda a vez que essa função é executada, queremos que ela devolva os pontos já calculados. Para isso, vamos incluir a palavra `return` seguido do nome da variável `pontos`

```
function calculaPontos(jogador) {
    var pontos = (jogador.vitorias * 3) + jogador.empates
    return pontos
}
```

Podemos calcular os pontos da Rafa e do Paulo, executando a função e atribuindo os pontos no campo `pontos` de cada objeto.

```
rafa.pontos = calculaPontos(rafa)
paulo.pontos = calculaPontos(paulo)
```

Maravilha! Isso ficou realmente incrível, mas não estamos vendo nada na tela. Isso que faremos a seguir.

Exibindo os jogadores na tela

Queremos exibir todos os jogadores na tela de uma só vez. Para isso vamos criar uma lista com todos nossos jogadores.

```
var jogadores = [rafa, paulo]
```

Além disso, vamos criar uma função que exibe todos os jogadores, passando nossa lista de jogadores.

```
function exibirJogadoresNaTela(jogadores) {}
```

Como queremos exibir os objetos que estão no Javascript na página HTML, podemos usar o `innerHTML` para isso. Vamos criar uma variável chamada `html` e juntar as informações como vitórias, empates, derrotas e pontos nela.

```
function exibirJogadoresNaTela(jogadores) {
    var html = ""
}
```

Podemos criar um `for` para exibir todos os jogadores de nossa lista.

```
function exibirJogadoresNaTela(jogadores) {
    var html = ""
    for (var i = 0; i < jogadores.length; i++) {}
}
```

Vamos varrer nossa lista icluíndo uma `tr` indicando uma nova linha e uma `td` para cada atributo do jogador.

```
function exibirJogadoresNaTela(jogadores) {
    var html = ""
    for (var i = 0; i < jogadores.length; i++) {
```

```
        html += "<tr><td>" + jogadores[i].nome + "</td>"
        html += "<td>" + jogadores[i].vitorias + "</td>"
        html += "<td>" + jogadores[i].empates + "</td>"
        html += "<td>" + jogadores[i].derrotas + "</td>"
        html += "<td>" + jogadores[i].pontos + "</td>"
    }
}
```

Para posicionar cada jogador de forma correta, criamos um elemtento no HTML com um ID chamado `tabelaJogador` , para exibir as informações de nossa tabela. Vamos recuperar ele elemento do HTML com `document.getElementById` e atribuir com `innerHTML` a variável `html` que criamos.

```
function exibirJogadoresNaTela(jogadores) {
    var html = ""
    for (var i = 0; i < jogadores.length; i++) {
        html += "<tr><td>" + jogadores[i].nome + "</td>"
        html += "<td>" + jogadores[i].vitorias + "</td>"
        html += "<td>" + jogadores[i].empates + "</td>"
        html += "<td>" + jogadores[i].derrotas + "</td>"
        html += "<td>" + jogadores[i].pontos + "</td>"
    }
    var tabelaJogadores = document.getElementById('tabelaJogadore
s')
    tabelaJogadores.innerHTML = html
}
```

Estamos vendo os elementos na tela, mas… Onde estão os botões?

Botões de cada objeto

Nossa função exibe os jogadores, mas não exibe os botões para adicionar vitórias, empates ou derrotas. Precisamos ajustar isso.

Vamos adicionar na função que exibe os jogadores na tela os botões e o concatenando o índice de cada jogador:

```
 html += "<td><button onClick='adicionarVitoria(" + i + ")'>Vitór
ia</button></td>"
 html += "<td><button onClick='adicionarEmpate(" + i + ")'>Empate
</button></td>"
 html += "<td><button onClick='adicionarDerrota(" + i + ")'>Derro
ta</button></td></tr>"
```

Veja como ficou nossa função completa:

```
function exibirJogadoresNaTela(jogadores) {
    var html = ""
    for (var i = 0; i < jogadores.length; i++) {
        html += "<tr><td>" + jogadores[i].nome + "</td>"
        html += "<td>" + jogadores[i].vitorias + "</td>"
        html += "<td>" + jogadores[i].empates + "</td>"
        html += "<td>" + jogadores[i].derrotas + "</td>"
        html += "<td>" + jogadores[i].pontos + "</td>"
        html += "<td><button onClick='adicionarVitoria(" + i + ")
'>Vitória</button></td>"
        html += "<td><button onClick='adicionarEmpate(" + i + ")'
>Empate</button></td>"
        html += "<td><button onClick='adicionarDerrota(" + i + ")
'>Derrota</button></td></tr>"
    }
    var tabelaJogadores = document.getElementById('tabelaJogadore
s')
    tabelaJogadores.innerHTML = html
}
```

Temos os botões aparecendo na linha de cada jogador. Porém, quando clicamos um erro informa que essas funções não foram definidas.

Função adicionar vitória

Vamos criar uma função chamada `adicionaVitoria` (mesmo nome do onclick do botão), variável chamada `jogador` e atriibuir da lista de jogadores no íindice `i` .

```
function adicionarVitoria(i) {
    var jogador = jogadores[i]
```

```
}
```

Agora que sabemos o jogador, vamos somar `1` na quantidade de vitória, adicionar o pontos no jogador e chamar a função para exibir os jogadores na tela.

```
function adicionarVitoria(i) {
    var jogador = jogadores[i]
    jogador.vitorias++
    jogador.pontos = calculaPontos(jogador)
    exibirJogadoresNaTela(jogadores)
}
```

Função adicionar empate

Para o botão de empate, vamos criar uma função chamada `adicionarEmpate` , somar 1 no empate do jogador com código `jogador.empate++` e assim como fizemos na função acima, atribuir os pontos calculados no jogador e chamar a função `exibirJogadoresNaTela` .

```
function adicionarEmpate(i) {
    var jogador = jogadores[i]
    jogador.empates++
    jogador.pontos = calculaPontos(jogador)
    exibirJogadoresNaTela(jogadores)
}
```

Função adicionar derrota

Está função será semelhante a outras, mas como não faremos nenhuma manipulação nos pontos do jogador, não precisamos atribuir nada nos pontos do jogador.

```
function adicionarDerrota(i) {
    var jogador = jogadores[i]
    jogador.derrotas++
    exibirJogadoresNaTela(jogadores)
```

```
}
```

Tudo certo? Então é hora de praticar!

6.3 RESUMO

Vamos repassar todos os passos do programa:

- Remover o código estático do HTML;
- Criar um objeto no Javascript para cada jogador;
- Criar uma função que receba um objeto como parâmetro para calcular os pontos;
- Exibir o objeto na página HTML;
- Criar uma função para adicionar vitória;
- Criar uma função para adicionar empate e outra para adicionar derrota;
- Recalcular os pontos quando vitória ou empate for adicionado.

Clique neste link para acessar o código completo no Codepen.

Capítulo 7

SUPER TRUNFO - LÓGICA DO JOGO

7.1 OBJETIVO DA AULA

Depois de toda essa prática, já podemos pensar em algum projeto um pouco mais extenso para aplicar tudo que vimos até agora e adicionar mais funcionalidades - as chamadas *features*. E esse projeto será um jogo clássico, o Super Trunfo!

O código da aula inicial para você acompanhar está aqui:

https://codepen.io/imersao-dev/pen/WNRNNOb

Não se esqueça de fazer o fork desse projeto para a sua conta, e de marcar a hashtag da #imersaodev e #alura.

Para criar este jogo, vamos focar em outra estrutura de dados muito utilizada, o `objeto` , além de mais funções e operadores.

7.2 PASSO A PASSO

Como das vezes anteriores, vamos fazer o exercício de colocar o processo em um *fluxo* com as seguintes informações:

- O que deve acontecer no programa em caso de sucesso;
- O que deve acontecer no programa em caso de falha;
- De que dados esse programa precisa;
- De onde virão esses dados.

Trabalhando com dados em objetos

Vimos anteriormente que é possível utilizar arrays `[]` para guardar conjuntos de dados. Isso funciona bem, por exemplo, para listas, quando os dados são do mesmo "tipo":

```
var arrayTelefones = [“11991231234”, “21992342342”, “31993453453”
];
```

Mas em alguns casos temos que passar grupos de dados um pouco mais complexos, por exemplo as informações de uma pessoa, por exemplo se for estudante de uma escola: estudantes têm nomes, datas de nascimento, e-mails de contato, em que classe estão, etc. Imaginando esses dados em uma tabela:

```
| Propriedade | Valor |
| --- | --- |
| nome | Nome Sobrenome |
| data de nascimento | 19/03/2008 |
| email | email@email.com |
|série | 5a B |
```

Não é o tipo de conjunto de dados que funcione muito bem em arrays, pois quando utilizamos arrays não é possível "classificar" cada um dos índices. Esse é um dos usos do *objeto*, que vamos ver aqui.

Para adaptarmos as informações de estudante acima para um *objeto*, é possível utilizar a seguinte sintaxe:

```
var estudante = {
```

```
 nome: "Nome Sobrenome",
 dataNascimento: "19/03/2008",
 email: "email@email.com",
 serie: "5a B"
}
```

Os objetos podem guardar dados de forma bastante complexa. Por exemplo, se temos mais de uma informação de contato para cada estudante, podemos salvá-las juntas, usando objetos dentro de objetos:

```
var estudante = {
 nome: "Nome Sobrenome",
 dataNascimento: "19/03/2008",
 contato: {
   email: "email@email.com",
   telefone: "11923452345"
 },
 serie: "5a B"
}
```

Certo, mas como podemos "pegar" as informações de dentro de um objeto para usar no código?

Fluxo do programa

Primeiro vamos relembrar bem resumidamente os passos do jogo Super Trunfo, pois já vimos essa parte no vídeo da aula:

1. Cada oponente tira a primeira carta de seu monte;
2. A pessoa escolher qual atributo da carta quer usar para "apostar" na comparação;
3. As cartas são comparadas e vence o turno a pessoa que tiver a carta com o valor maior no atributo que tiver sido escolhido.

Para nosso primeiro código, simplificamos algumas partes do

jogo, em comparação com a forma que jogamos ao vivo; nesta primeira versão, vamos jogar contra o computador.

> Começar de forma simples, testando a lógica com poucas "funcionalidades", é a forma mais comum de trabalho durante o desenvolvimento de um projeto. Sempre incrementamos nosso código aos poucos!

Podemos traduzir os passos acima de uma forma mais parecida com o fluxo de código:

1. Iniciamos com uma lista de cartas pré-determinadas;
2. O programa sorteia 2 cartas a partir dessa lista, uma para quem está jogando e outra que vai ser a primeira carta da "pilha" do computador;
3. O programa exibe na tela a carta que foi sorteada, para quem está jogando possa escolher o atributo que vai "apostar";
4. O usuário que está jogando faz sua escolha;
5. O programa automaticamente compara com a outra carta sorteada (a carta do computador) e anuncia quem venceu a rodada.

Hora de colocar esses passos todos em formato de código!

7.3 ESCREVENDO O CÓDIGO

Os objetos para cada carta

Cada carta do Super Trunfo segue um padrão de nome,

propriedade, etc, mas com seus próprios valores. Podemos pensar no exemplo abaixo:

```
var cartaSeiya = {
 nome: "Seiya de Pégaso",
 atributos: {
     ataque: 80,
     defesa: 60,
     magia: 90
 }
}

var cartaPokemon = {
 nome: "Bulbasauro",
 atributos: {
     ataque: 70,
     defesa: 65,
     magia: 85
 }
}

var cartaStarWars = {
 nome: "Lorde Darth Vader",
 atributos: {
     ataque: 88,
     defesa: 62,
     magia: 90
 }
}
```

Neste código, cada carta tem seu próprio objeto, salvo em sua própria variável. Já aprendemos anteriormente que uma array é um tipo de lista *ordenada* de elementos, então podemos usar estas duas estruturas de dados em conjunto, da seguinte forma:

```
// índice      0           1          2
var cartas = [ cartaSeiya, cartaPokemon, cartaStarWars ]
```

O programa vai ter que separar quais cartas serão sorteadas, então já vamos deixar as variáveis para a carta do computador e do usuário *declaradas, mas sem valor*:

```
var cartaMaquina
var cartaJogador
var cartas = [ cartaSeiya, cartaPokemon, cartaStarWars ]
```

Sorteando as cartas

Outra coisa que também já vimos foi como gerar números aleatoriamente. Então, agora que temos uma lista de cartas em uma array, é possível usar o JavaScript para determinar que cartas serão sorteadas!

```
var numeroCartaMaquina = parseInt(Math.random() * 3)
cartaMaquina = cartas[numeroCartaMaquina]
```

Com o código acima, os resultados possíveis são os abaixo. Revise o conteúdo anterior sobre `Math.random()` caso precise relembrar!

```
cartas[0] // cartaSeiya
cartas[1] // cartaPokemon
cartas[2] // cartaStarWars
```

Antes de continuar, temos que garantir que os números sorteados não sejam os mesmos para jogador e computador! Ou seja, **enquanto** (*while*) as duas cartas sorteadas forem as mesmas - o que pode acontecer bastante, pois nosso baralho por enquanto só tem três cartas - o programa deve sortear uma carta nova:

```
var numeroCartaMaquina = parseInt(Math.random() * 3)
cartaMaquina = cartas[numeroCartaMaquina]

var numeroCartaJogador = parseInt(Math.random() * 3)
while (numeroCartaJogador == numeroCartaMaquina) {
    numeroCartaJogador = parseInt(Math.random() * 3)
}
cartaJogador = cartas[numeroCartaJogador]
```

No código acima, o programa sorteia primeiro a carta do

computador, salva o número em `numeroCartaMaquina` e utiliza o recurso de array `array[numeroDoIndice]` para salvar o resultado em `cartaMaquina`.

Em seguida, o processo para sortear a carta do jogador deveria ser o mesmo, certo? Porém temos um passo a mais: uma *iteração* que verifica se o número sorteado (lembrando que, no caso do código acima, é um número entre 0 e 2) não é o mesmo, para que computador e jogador não acabem com o mesmo número de carta. Essa iteração está sendo feita com e ferramenta `while`; ou seja, *enquanto* o número sorteado para computador e jogador forem iguais, peça ao programa para sortear novamente *apenas para o jogador*.

Lembrando que, uma vez dentro do *loop*, o JavaScript não sai até que esteja resolvido. No caso, até que os números sorteados sejam diferentes. Só depois disso o programa realmente define qual é a carta sorteada para o jogador.

Finalizando a função do sorteio

Agora que já temos a lógica principal da primeira parte do programa (sortear as cartas), podemos finalizar esta função, que será chamada/executada quando o usuário clicar no botão `<button onclick="sortearCarta()" id="btnSortear">Sortear carta</button>`.

No trecho de HTML acima, o botão está executando a função `sortearCarta()`, então vamos escrevê-la com o código que já temos:

```
function sortearCarta() {
   var numeroCartaMaquina = parseInt(Math.random() * 3)
```

```
    cartaMaquina = cartas[numeroCartaMaquina]

    var numeroCartaJogador = parseInt(Math.random() * 3)
    while (numeroCartaJogador == numeroCartaMaquina) {
        numeroCartaJogador = parseInt(Math.random() * 3)
    }
    cartaJogador = cartas[numeroCartaJogador]
    console.log(cartaJogador)

    document.getElementById('btnSortear').disabled = true
    document.getElementById('btnJogar').disabled = false
    // exibirOpcoes()
}
```

O código dessa função é em grande parte o que já vimos acima. Além de organizado e "cercado" por uma função, veja algumas linhas que acrescentamos no final:

```
    document.getElementById('btnSortear').disabled = true
    document.getElementById('btnJogar').disabled = false
    // exibirOpcoes()
```

Para ajudar o usuário a fazer decisões na tela, é comum habilitarmos e desabilitarmos as botões, campos de texto e etc. A propriedade `.disabled` (ou "desabilitado" em inglês) pode fazer isso para nós; Depois de sorteadas as cartas, pedimos ao JavaScript que desabilite o botão de sorteio para não ser clicado novamente enquanto o jogo não se resolve - e habilitar o botão de jogo. Se deixarmos esses botões livres para serem clicados pelo usuário **fora do momento certo** todo o programa pode ficar *bugado*.

Você consegue imaginar algumas razões para esse controle que fazemos de quando e onde os usuários podem clicar e interagir com um programa na tela? Esta é uma boa hora para refletir sobre a importância dessas ferramentas!

Agora que a parte de sortear as cartas está definida, podemos passar para as próximas etapas do programa:

- Jogador escolhe um atributo;
- O programa faz a comparação com a carta sorteada para o computador.

Já podemos deixar a próxima função - estamos chamando aqui de `exibirOpcoes()` - chamada na última linha, porém "comentada" para não interferir no código por enquanto.

Percorrendo as opções/atributos

Quando programamos, sempre temos que lembrar que a quantidade de dados que temos que lidar pode ser grande, desde uma lista com 120 Pokémons até cadastros de clientes de um negócio. Nosso programa tem apenas três objetos, mas as pergunta são as mesmas para três cartas ou para 120 Pokémons:

1. Como "extrair" as informações que queremos de dentro de cada um dos objetos da lista, de forma automatizada?
2. Como exibir estas informações na tela?

Assim como aprendemos anteriormente o método `for` para

iterar uma array, o JavaScript tem outros métodos tanto para iterar arrays quanto objetos. Um deles é o chamado `for...in`, que serve explicitamente para iteração de *objetos*:

```
function exibirOpcoes() {
   for (var atributo in cartaJogador.atributos) {
       console.log(atributo)
   }
}

/* neste ponto, você já pode tirar o comentário da última linha d
a função anterior, para que exibirOpcoes() possa ser chamada/exec
utada.
```

A sintaxe é parecida com a do `for` clássico, porém a variável que normalmente chamamos apenas de `i` (lembrando que é somente uma convenção, você poderia dar o nome que quiser) agora chama `atributo`. E o que é exatamente `(var atributo in cartaJogador.atributos)` ou, traduzindo para o português, "variável `atributo` em `cartaJogador.atributos"?

Primeira coisa é entender de onde estão saindo os dados. No caso acima, `cartaJogador` é uma variável que está definida no início do código, em teoria salvando um objeto com os dados de uma carta do jogo… Você pode sempre recorrer ao `console.log(cartaJogador)` para confirmar!

E quando falamos de `cartaJogador.atributos`? Aqui começamos a lidar um pouco mais com os objetos e como acessar os dados dentro deles. Vamos rever como estão estruturados os objetos de cada carta, com um exemplo:

```
{
   nome: "Seiya de Pégaso",
   atributos: {
       ataque: 80,
       defesa: 60,
```

```
        magia: 90
    }
}
```

O objeto acima é uma das opções que podem ser sorteadas para a variável `cartaJogador` , de acordo com a lógica que vimos anteriormente. O JavaScript utiliza o que chamamos de *notação de ponto* para dizer ao código como acessar propriedades dentro de um objeto. Teste as seguintes opções em seu código:

```
console.log(cartaJogador.nome) //texto (string) com o nome na car
ta
console.log(cartaJogador.atributos) // objeto com os atributos da
 carta
```

Juntando tudo: O método `for...in` serve para iterar em objetos; no nosso código, o objeto em questão é o que está dentro de `cartaJogador.atributos` , ou seja, a lista de "poderes" de cada personagem. Pensando em **iteração**, ou seja, em pedir ao JavaScript para percorrer uma lista de informações, podemos concluir que essa iteração vai acontecer nesta parte do código:

```
    atributos: {
        ataque: 80,
        defesa: 60,
        magia: 90
    }
```

> Você vai perceber cada vez mais, durante seus estudos, que as estruturas de arrays e objetos muitas vezes são utilizadas em conjunto. O segredo é PRATICAR SEMPRE!

Exibindo as opções/atributos

Já descobrimos como o JavaScript pode acessar um objeto e "extrair" dados de dentro dele, então podemos exibir essas informações na tela para que o jogador escolha em qual "poder" da carta quer apostar.

```
function exibirOpcoes() {
   var opcoes = document.getElementById('opcoes')
   var opcoesTexto = ""
   for (var atributo in cartaJogador.atributos) {
       opcoesTexto += "<input type='radio' name='atributo' value=
'" + atributo + "'>" + atributo
   }
   opcoes.innerHTML = opcoesTexto
}
```

A função `exibirOpcoes()` completa está acima. As linhas que adicionamos estão relacionadas aos métodos do JavaScript usados para acessar os elementos corretos no HTML e adicionar mais elementos sem sobrescrever o que já está na tela:

```
   var opcoes = document.getElementById('opcoes')
   var opcoesTexto = ""
```

A variável `opcoes` acessa uma tag específica do HTML que esteja identificada por `id="opcoes"` . Em seguida, criamos a variável `opcoesTexto` com um valor de que chamamos de *string vazia* - um valor de texto, porém sem conteúdo.

Agora, dentro do iterador `for...in` podemos substituir o `console.log()` anterior, adicionando com o operador `+=` mais um valor de string na variável `opcoesTexto` . Como fizemos anteriormente, utilizamos strings de texto com a sintaxe do HTML *concatenadas* (ou seja, `"valor da string" + variavel`) com variáveis para que o JavaScript crie o HTML de forma automática. A tag de HTML utilizada aqui é a `<input type='radio' >` , vamos falar mais sobre isso um pouco mais abaixo.

A última linha adicionada é `opcoes.innerHTML = opcoesTexto` ; depois da iteração, o JavaScript já pode acessar `innerHTML` do elemento que salvamos na variável `opcoes` e inserir o novo conteúdo. Quando isso acontece, o HTML que está na tela vai ser atualizado.

As opções já estão na tela e o usuário pode selecionar em uma das "bolinhas" que acompanham as opções… Mas por enquanto nada vai acontecer, pois ainda não avisamos o JavaScript que ele tem que fazer qualquer coisa com essa informação… Ou seja, salvar a opção escolhida e utilizá-la no programa.

O usuário escolhe um atributo

O HTML reconhece vários tipos de interação que o usuário pode fazer na tela: preencher um texto, clicar em um botão, selecionar o que chamamos de *checkboxes*. E existe um tipo de interação específico para o caso deste jogo, o `radio` . Escolhemos esta tag pois limita o usuário a escolher somente uma opção de uma lista que já está pronta, então quem escolher não consegue nem adicionar opções nem clicar em mais de uma.

Agora que:

1. As carta já foram sorteadas e
2. O jogador já pode escolher um "poder" para comparar

É hora de jogar mesmo! O botão já deve estar liberado graças ao código que fizemos anteriormente (com o `.disabled = false`). Falta escrever a função que vai ser executada com o clique nesse botão, que chamamos de função `jogar()` . Não esqueça de conferir no HTML se ela está sendo executada em

`onclick="jogar()"`:

```
<form id="form">
<h2>Escolha o seu atributo</h2>
<div class="opcoes" id="opcoes"></div>
<button type="button" id="btnJogar" onclick="jogar()" disabled="f
alse">Jogar</button>
</form>
```

> A função `jogar()` vai ser responsável pela lógica de fazer as comparações entre as cartas e definir quem ganhou ou perdeu a rodada. Mas temos que voltar uma etapa, pois esta função depende de uma informação que não temos ainda: qual foi o poder escolhido pelo jogador na tela.

Vamos então já criar a função `jogar()`, já criando também a função que vai buscar as informações no HTML:

```
function jogar() {
   var atributoSelecionado = obtemAtributoSelecionado()

}
```

A linha acima executa a função `obtemAtributoSelecionado()` (que ainda vamos escrever!) e salva as *informações de retorno* dela na variável `atributoSelecionado`. Veja mais sobre retornos de função abaixo!

Hora de criar a função `obtemAtributoSelecionado()`:

```
function obtemAtributoSelecionado() {
   var radioAtributo = document.getElementsByName('atributo')
   for (var i = 0; i < radioAtributo.length; i++) {
       if (radioAtributo[i].checked) {
```

```
            return radioAtributo[i].value
        }
    }
}
```

Neste ponto do aprendizado começamos a rever conceitos que já aprendemos antes e como reaproveitá-los em diversas situações.

A variável `radioAtributo` procura no HTML todos os elementos com `name='atributo'` e salva em uma *lista*. Falando em listas, já pensamos que esta lista deve ser *iterável*, utilizando `for` por exemplo.

> Você consegue localizar no HTML e no código que já escrevemos onde `name='atributo'` está sendo utilizado?

O `for` nós já vimos em ação e você pode incluir alguns `console.log()` para relembrar o que está acontecendo em cada parte da iteração. Agora veja as linhas:

```
if (radioAtributo[i].checked) {
    return radioAtributo[i].value
}
```

O `for` está percorrendo a lista de ítens de "poderes" e verificando qual deles tem a propriedade `checked`, uma propriedade automática que o HTML adiciona toda vez que o usuário clica no botão de rádio correspondente na tela. Caso encontre, o JavaScript vai **retornar `.value` deste elemento**. Mas o que isso significa?

A palavra-chave `return` é muito importante e tem diversos

usos quando trabalhamos com funções. Neste primeiro momento, o importante é entender que é através dela que *o valor guardado em* `radioAtributo[i].value` *consegue ser acessado por outras funções, salvo em variáveis, etc*. Sem o `return` , o JavaScript pode fazer todas as operações dentro de uma função, mas outras partes do código que estão fora dela - ou seja, fora do bloco de código `{}` da função - não conseguem acessar a informação.

Por último, o que é `value` nessa parte específica do código? É onde está o valor (em texto mesmo, ou *string*) de cada um dos "poderes" da carta. Dê uma olhada no código que criamos para a função `exibirOpcoes()` e veja como o atributo HTML `value=` está sendo gerado com o nome de cada um dos "poderes".

Finalizando a lógica do jogo

Por enquanto a função `jogar()` somente executa a função `obtemAtributoSelecionado()` e, por causa do `return` , consegue salvar o `value` do "poder" selecionado pelo usuário na tela em uma variável.

```
function jogar() {
   var atributoSelecionado = obtemAtributoSelecionado()

}
```

Os passos finais desta primeira versão de jogo são:

1. Comparar os "poderes" nas cartas sorteadas;
2. Informar o resultado para o jogador

Os valores dos poderes são numéricos, então podemos concluir que uma comparação entre dois números inteiros pode ter três resultados: *valor A é menor que valor B, valor A é maior que valor B*

ou empate (ambos os valores são iguais).

Já praticamos anteriormente com as ferramentas para comparar valores e também como passar **condições** ao JavaScript: *se tal condição se cumprir, vá por tal caminho; caso contrário, siga por outro caminho*. Vamos reaproveitar tudo aqui:

```
function jogar() {
   var atributoSelecionado = obtemAtributoSelecionado()

   if (cartaJogador.atributos[atributoSelecionado] > cartaMaquina
.atributos[atributoSelecionado]) {
       alert('Venceu. A carta do computador é menor')
   } else if (cartaJogador.atributos[atributoSelecionado] < carta
Maquina.atributos[atributoSelecionado]) {
       alert('Perdeu. A carta do computador é maior')
   } else {
       alert('Empatou!')
   }
   console.log(cartaMaquina)
}
```

E como exatamente o JavaScript consegue obter o **valor** de cada atributo para comparar? Através da sintaxe `objeto[“nomeDaPropriedade”]` quando sabemos exatamente o nome da propriedade ou `objeto[variavel]` (sem aspas) quando não temos como passar o nome da propriedade ou será feita uma iteração.

Faça os seguintes testes no seu código, logo abaixo da variável `cartaSeiya` :

```
//usamos colchete e aspas quando sabemos exatamente o nome da pro
priedade e queremos acessar o valor correspondente
console.log(cartaPaulo.atributos["ataque"])

//usamos colchete e uma variável quando não sabemos exatamente ou
 não temos como fixar o nome da propriedade, por exemplo em caso
de iteração, para acessar o valor correspondente a cada proprieda
```

```
de
for (atributo in cartaPaulo.atributos) {
   console.log(cartaPaulo.atributos[atributo])
}
```

> Dica: agora é possível acessar os VALORES de cada propriedade (ou seja, cada "poder" das cartas) e aí sim fazer a comparação entre o valor do "poder" na carta sorteada para o jogador e para o computador. O fluxo do código que deve ser executado para cada resultado da comparação pode ser resolvido com `if/else if/else`.

Observe um detalhe na execução deste código: separamos o código em funções para que só sejam executados no momento certo e podemos ver um exemplo disso na função `obtemAtributoSelecionado()`; esta função está sendo chamada/executada a partir da execução da função `jogar()`. Como a própria função `jogar()` não é executada antes do jogador clicar no botão, não há o risco da função `obtemAtributoSelecionado()` ser executada sem os dados necessários, ou em um momento errado.

Esta primeira versão do Super Trunfo já tem bastante código, então vamos parar por aqui e respirar um pouco.

7.4 PARA SABER MAIS

Os temas que estamos vendo nestas aulas vão te acompanhar durante toda sua trajetória no desenvolvimento web. Abaixo seguem alguns temas interessantes para você já ir se

aprofundando:

- escopo
- a palavra-chave `return`
- manipulação de arrays (iteração, filtros, etc)
- manipulação de objetos (iteração, acesso a propriedades, acesso a valores)

Tudo certo? Então é hora de praticar!

7.5 RESUMO

Vamos repassar todos os passos do programa:

- Criar as cartas do jogo e definir seus atributos;
- Desenvolver uma função para sortear uma carta para o jogador e outra para a máquina;
- Exibindo os atributos das cartas na tela para o jogador;
- Obter o atribudo escolhido pelo jogador e comparar com a carta da máquina;
- Comparar o atributo de ambas as cartas e definir um vencedor.

Clique neste link para acessar o código completo no Codepen.

Capítulo 8

SUPER TRUNFO: MONTAGEM DAS CARTAS

8.1 OBJETIVO DA AULA

Além de resolver problemas de forma lógica, trabalhar com programação também envolve a **integração de ferramentas**. Para esta Imersão, utilizamos o JavaScript em conjunto com o HTML e CSS, que são a base para o que chamamos de **desenvolvimento web front end**.

O código da aula inicial para você acompanhar está aqui:

https://codepen.io/imersao-dev/pen/GREGPNb

Não se esqueça de fazer o fork desse projeto para a sua conta, e de marcar a hashtag da #imersaodev e #alura.

Nesta aula, vamos evoluir o que já criamos para o Super Trunfo, focando justamente na integração da lógica com a tela.

8.2 PASSO A PASSO

Como das vezes anteriores, vamos começar pelo fluxo do programa. O jogo já está fazendo o **mínimo necessário** para que o

fluxo lógico do Super Trunfo funcione, então podemos pensar em incrementar a experiência de jogo, adicionando alguns passos:

- após o sorteio das cartas, extrair de cada objeto a imagem da carta e o valor de cada "poder";
- exibir na tela estas informações, fazendo com que apareçam no local correto (ou seja, dentro da tag esperada).

Adicionando imagens para as cartas

Vamos pegar estas imagens da internet, como fizemos com os pôsteres do AluraFlix:

```
var cartaSeiya = {
   nome: "Seiya de Pégaso",
   imagem: "https://i.pinimg.com/originals/c2/1a/ac/c21aacd5d092b
f17cfff269091f04606.jpg",
   atributos: {
       ataque: 80,
       defesa: 60,
       magia: 90
   }
}

var cartaPokemon = {
   nome: "Bulbasauro",
   imagem: "http://4.bp.blogspot.com/-ZoCqleSAYNc/UQgfMdobjUI/AAA
AAAAACP0/s_iiWjmw2Ys/s1600/001Bulbasaur_Dream.png",
   atributos: {
       ataque: 70,
       defesa: 65,
       magia: 85
   }
}

var cartaStarWars = {
   nome: "Lorde Darth Vader",
   imagem: "https://images-na.ssl-images-amazon.com/images/I/51VJ
BqMZVAL._SX328_BO1,204,203,200_.jpg",
   atributos: {
```

```
        ataque: 88,
        defesa: 62,
        magia: 90
    }
}
```

Note que acrescentamos em cada objeto de carta um **atributo** a mais, o atributo `imagem`, cada um com um valor de `string` correspondente ao link da imagem.

> Você pode escolher outras imagens se quiser, mas não esqueça de conferir se o link termina com `.jpg` ou `.png`.

Exibindo as informações adicionais na tela

Por enquanto, o programa está exibindo na tela somente uma lista dos "poderes", que pegamos a partir da carta sorteada para o jogador — estávamos mostrando só a lista, sem valores! Hora de exibir também os valores, para que o jogador possa escolher melhor em qual poder quer apostar.

Para isso, precisamos:

- localizar um elemento HTML específico na tela, usando o `getElementById()` ou o `getElementsByName()`, de acordo com o caso;
- adicionar novas informações formatadas como tags de HTML, utilizando JavaScript;
- descobrir como incluir estilos CSS a estas novas informações que estão sendo adicionadas.

Vamos começar criando a função `exibeCartaJogador()` e

usando o `document.getElementById("carta-jogador")` para localizar o elemento com o identificador `id="carta-jogador"` no HTML e salvar este "endereço" em uma variável.

```
function exibeCartaJogador() {
   var divCartaJogador = document.getElementById("carta-jogador")

}
```

Para testar esse código, substitua a chamada da função `exibirOpcoes()` na úlfima linha da função `sortearCarta()` por esta que estamos começando a criar agora: `exibeCartaJogador()`.

Agora, quando o jogador clica no botão `sortear carta`, além da lógica do sorteio que já vimos anteriormente, o JavaScript também tem que "injetar" pedaços de código em HTML e em CSS para que os dados apareçam nos locais corretos da tela e com a aparência que esperamos.

Tanto o HTML quanto o CSS têm suas próprias sintaxes, que são diferentes do JavaScript. Porém, o JavaScript tem ferramentas que utilizam strings (caracteres entre aspas `" "`) combinados com variáveis para criar novos trechos de código em HTML e CSS, fazendo com que o navegador reconheça as instruções. Já fizemos um pouco disso nos projetos anteriores, toda vez que utilizamos a palavra-chave `innerHTML` para inserir tags HTML no formato de string.

O HTML inicial deste projeto já conta com imagem pronta para ser a base de todas as cartas, e o jogo já inicia com esta imagem aplicada na `<div>` de carta do jogador e de carta do computador:

```
<div>
   <div id="carta-jogador">
       <img src="https://www.alura.com.br/assets/img/imersoes/dev
-2021/card-super-trunfo-transparent-ajustado.png"
           style=" width: inherit; height: inherit; position: abs
olute;">
       <h3></h3>
   </div>
</div>
<div>
   <div id="carta-maquina" class="carta"><img
    src="https://www.alura.com.br/assets/img/imersoes/dev-2021/ca
rd-super-trunfo-transparent-ajustado.png"
           style=" width: inherit; height: inherit; position: abs
olute;"></div>
</div>
```

Já temos a base, mas como exibir as informações nela? Lembrando que, nessa hora, só devemos mostrar a carta sorteada para o jogador!

As informações são:

- nome da carta;
- imagem;
- poderes **e seus valores**.

Quando queremos adicionar estilo ao HTML, utilizamos as chamadas **propriedades de CSS**; por exemplo, trocar as cores dos textos, alinhar elementos, entre muitas outras coisas.

É possível atribuir estas propriedades a tags HTML de algumas formas. Vamos ver um exemplo baseado no código do primeiro projeto da Imersão:

No arquivo HTML atribuímos `class` a um elemento:

```
<h1 class="page-title">
 Conversor de moedas
```

```
</h1>
```

E no CSS utilizamos o nome de `class` para mudar propriedades; no exemplo abaixo, a cor do texto:

```
.page-title {
   color: #ffffff;
   margin: 0 0 5px;
 }
```

O exemplo acima é uma forma bastante utilizada. Mas também é possível passar estilos direto na tag HTML do elemento, dispensando o arquivo CSS. O exemplo acima ficaria, então, da seguinte forma:

```
<h1 class="page-title" style="color: #ffffff; margin: 0 0 5px;">
 Conversor de moedas
</h1>
```

O JavaScript consegue manipular tags HTML e adicionar `style` a uma tag, e é esta ferramenta que vamos utilizar para adicionar a imagem correspondente:

```
function exibeCartaJogador() {
   var divCartaJogador = document.getElementById("carta-jogador")
   divCartaJogador.style.backgroundImage = `url(${cartaJogador.im
agem})`
}
```

No código acima, utilizamos `divCartaJogador.style.backgroundImage = url(${cartaJogador.imagem}) ` para adicionar um atributo style ao elemento que já estava salvo na variável divCartaJogador . Além de style adicionamos também qual é a propriedade de estilo que queremos adicionar e seu valor, uma url (um caminho de um link) que estamos obtendo do objeto cartaJogador.imagem`.

Se precisar, relembre o processo de acessar dados de um objeto no material da aula anterior.

Quando o JavaScript processar esse trecho de código, o HTML gerado vai ser semelhante ao abaixo:

```
<div id="carta-jogador" style="background-image: url('https://[en
dereço da sua imagem].jpg');">
</div>
```

Ainda temos que exibir o nome da carta e os poderes com valores. Ou seja, **também temos que utilizar o JavaScript para gerar código HTML destes elementos**.

Podemos pegar o nome no mesmo objeto `cartaJogador` e criar uma tag `<p>` (tag genérica de texto) com esta informação:

```
function exibeCartaJogador() {
   var divCartaJogador = document.getElementById("carta-jogador")
   divCartaJogador.style.backgroundImage = `url(${cartaJogador.im
agem})`
   var nome = `<p class="carta-subtitle">${cartaJogador.nome}</p>
`
}
```

O nome ainda não está aparecendo na tela, pois ainda não atualizamos o valor de `divCartaJogador` com mais este trecho de código HTML.

```
function exibeCartaJogador() {
   var divCartaJogador = document.getElementById("carta-jogador")
   divCartaJogador.style.backgroundImage = `url(${cartaJogador.im
agem})`
   var nome = `<p class="carta-subtitle">${cartaJogador.nome}</p>
`
```

```
   divCartaJogador.innerHTML += nome
}
```

Para atualizar uma variável com um novo valor sem substituir totalmente o que já está salvo nela, podemos usar o operador `+=`, como já vimos nos exercícios com iteradores.

Imagem e nome da carta estão na tela, hora de lidar com a lista de poderes e os valores. Também temos que usar o JavaScript para criar tags HTML e inserir tudo no código; a diferença é que, nesse caso, **temos que fazer isso para cada um dos poderes que está dentro de um objeto**.

No estágio inicial deste projeto, vimos como percorrer objetos utilizando `for… in`. Podemos reutilizar este método para criar tags HTML de forma automatizada com cada poder e valor.

```
function exibeCartaJogador() {
   var divCartaJogador = document.getElementById("carta-jogador")
   divCartaJogador.style.backgroundImage = `url(${cartaJogador.im
agem})`
   var nome = `<p class="carta-subtitle">${cartaJogador.nome}</p>
`
   var opcoesTexto = ""

   for (var atributo in cartaJogador.atributos) {
       opcoesTexto += "<input type='radio' name='atributo' value=
'" + atributo + "'>" + atributo + " " + cartaJogador.atributos[at
ributo] + "<br>"
   }

   divCartaJogador.innerHTML += nome + opcoesTexto
}
```

Para cada um dos poderes, temos que criar uma `string` com a sintaxe do HTML e as informações que vamos acessar do objeto JavaScript referente à carta do jogador. Então, a primeira coisa a fazer é criar uma variável para salvar todas essas tags, começando

com um valor de “vazia”:

```
var opcoesTexto = ""
```

> A estrutura do `for… in` é a mesma que fizemos para a fase anterior deste projeto! Você pode consultar o material caso precise relembrar o que foi feito.

```
for (var atributo in cartaJogador.atributos) {
 // código aqui
}
```

O que será feito em cada **iteração** é que muda. Além disso, já vimos o `input type=”radio”` na fase anterior do projeto, e também como acessar os valores em um objeto, e não apenas as propriedades:

```
for (var atributo in cartaJogador.atributos) {
   opcoesTexto += "<input type='radio' name='atributo' value='" +
 atributo + "'>" + atributo + " " + cartaJogador.atributos[atribu
to] + "<br>"
}
```

Agora o JavaScript vai unir cada `atributo` (ou seja, cada item dentro do objeto `cartaJogador.atributos`) com strings que representam código HTML, salvando o resultado de cada iteração na variável `opcoesTexto` — repare que estamos usando o operador `+=` novamente para atualizar a variável ao invés de substituir seu valor.

Temos que atualizar também o valor que será enviado para a variável `divCartaJogador` : isso está sendo feito com a linha `divCartaJogador.innerHTML += nome + opcoesTexto` … Mas

se tentarmos atualizar a tela neste momento, perceberemos que os poderes não estão aparecendo na parte da tela onde deveriam.

Aqui vamos entrar um pouco mais em como HTML e CSS trabalham juntos para que todas as partes da tela estejam onde deveriam estar. Para este projeto, nosso time de front end já deixou algumas propriedades de CSS criadas no arquivo `style.css`, prontas para serem acessadas pelo HTML… Só precisamos criar o código para isso!

```
   divCartaJogador.innerHTML += "<div id='opcoes' class='carta-st
atus'>" + nome + opcoesTexto + "</div>"
```

A classe `class='carta-status'` vai cuidar do alinhamento, o que precisamos fazer é inserir uma tag para agrupar tudo. No caso, vamos criar uma tag de elemento do tipo `<div>`, adicionar as variáveis com o código HTML que acabamos de criar, sem esquecer de fechar a tag no final da linha, com `</div>`.

O resultado final desta função fica da seguinte forma:

```
function exibeCartaJogador() {
   var divCartaJogador = document.getElementById("carta-jogador")
   divCartaJogador.style.backgroundImage = `url(${cartaJogador.im
agem})`
   var nome = `<p class="carta-subtitle">${cartaJogador.nome}</p>
`
   var opcoesTexto = ""

   for (var atributo in cartaJogador.atributos) {
       opcoesTexto += "<input type='radio' name='atributo' value=
'" + atributo + "'>" + atributo + " " + cartaJogador.atributos[at
ributo] + "<br>"
   }

   divCartaJogador.innerHTML += "<div id='opcoes' class='carta-st
atus'>" + nome + opcoesTexto + "</div>"
}
```

HTML e CSS são competências à parte da lógica de programação em si! Durante a Imersão decidimos não focar muito em conceitos destas linguagens, pois seria muita coisa para ver ao mesmo tempo! Mas se você gostar, pode se aprofundar também nestes assuntos e revisitar este código mais tarde para ver como estas partes estão trabalhando juntas.

Exibir a carta do computador

Uma boa parte do código que fizemos na versão anterior pode ser aproveitada neste trecho.

A função `obtemAtributoSelecionado()` não vai ser modificada, pois a lógica de obter a lista de elementos `<input type='radio' name='atributo'>` para verificar qual está selecionado (`.checked`) é a mesma.

Na função `jogar()` — a função que usamos, entre outras coisas, para chamar/executar `obtemAtributoSelecionado()` —, vamos manter a lógica de verificação usada no `if` (valor maior/ valor menor/ empate), porém agora substituindo o `alert` por uma mensagem na tela:

```
function jogar() {
   var atributoSelecionado = obtemAtributoSelecionado()
   var htmlResultado = ""

   if (cartaJogador.atributos[atributoSelecionado] > cartaMaquina
.atributos[atributoSelecionado]) {
       htmlResultado = '<p class="resultado-final">Venceu</p>'
   } else if (cartaJogador.atributos[atributoSelecionado] < carta
```

```
Maquina.atributos[atributoSelecionado]) {
        htmlResultado = '<p class="resultado-final">Perdeu</p>'
    } else {
        htmlResultado = '<p class="resultado-final">Empatou</p>'
    }

    var divResultado = document.getElementById("resultado")
    divResultado.innerHTML = htmlResultado

    exibeCartaMaquina()
}
```

Começamos com a variável `htmlResultado` vazia, e para cada uma das **condições** do `if`, salvamos uma `string` com a tag HTML e a informação de acordo com a comparação (jogador perdeu, jogador ganhou, etc).

Os passos seguintes, como nas outras funções que criamos, é localizar o elemento `"resultado"` no HTML e inserir no `.innerHTML` deste elemento a string que o JavaScript salvou na variável `htmlResultado` de acordo com o caminho seguido pelo `if`.

Agora que já temos o resultado dessa rodada do jogo, podemos passar para o último passo, que é revelar a carta do computador. Já vamos deixar esta função sendo chamada na última linha, `exibeCartaMaquina()`.

Exibindo a carta do computador

Já temos a função `exibeCartaJogador()`, e se formos pensar na lógica do que precisamos fazer para exibir a carta do computador, podemos identificar coisas semelhantes:

- acessar objeto da carta sorteada;
- exibir nome e imagem a partir deste objeto;

- percorrer (iterar) o objeto `cartaMaquina.atributos` para acessar a lista de poderes;
- exibir estes atributos da forma correta na tela (nesse caso, não precisamos dos botões `type="radio"` para selecionarmos uma opção).

Assim, a função final fica da seguinte forma:

```
function exibeCartaMaquina() {
   var divCartaMaquina = document.getElementById("carta-maquina")
   divCartaMaquina.style.backgroundImage = `url(${cartaMaquina.im
agem})`
   var nome = `<p class="carta-subtitle">${cartaMaquina.nome}</p>
`
   var opcoesTexto = ""

   for (var atributo in cartaMaquina.atributos) {
       opcoesTexto += "<p type='text' name='atributo' value='" +
atributo + "'>" + atributo + " " + cartaMaquina.atributos[atribut
o] + "<br>"
   }

   divCartaMaquina.innerHTML += "<div id='opcoes' class='carta-st
atus --spacing'>" + nome + opcoesTexto + '</div>'
}
```

As **diferenças** entre as funções `exibeCartaJogador()` e `exibeCartaMaquina()` são:

- o momento em que são chamadas: exibimos na tela a carta do jogador quando o usuário clica no botão para sortear sua carta. Já a carta do computador só pode ser exibida **após** o usuário escolher um poder e clicar no botão `jogar`. Afinal de contas, não podemos deixar o usuário ver os poderes da carta do computador antes de fazer a escolha e confirmar clicando no botão!
- o HTML gerado no `for... in`: na carta do jogador, temos

que criar uma tag do tipo `<input type="radio">` para o usuário selecionar o poder que quer comparar. Para a carta do computador, podemos criar uma tag de texto normal do HTML (`<p>`) já que o jogador não vai fazer nenhuma escolha nesse caso, só acompanhar os resultados.

Você já ouviu falar em "reaproveitamento de código"? Quando utilizamos partes de código similares, como no caso das funções `exibeCartaJogador()` e `exibeCartaMaquina()` é normal tentarmos escrever de uma forma que tenha o melhor aproveitamento possível. Ou seja, tentamos escrever somente um trecho de código que sirva para várias situações; afinal de contas, o código para exibir cartas na tela deveria funcionar independente de ser a carta do jogador, do computador, ou qualquer outra situação. Você vai se deparar muito com esta questão durante seus estudos, e vai ter a oportunidade de estudar situações em que isso pode inclusive não ser vantajoso… **No código que acabamos de fazer, você consegue pensar em como reaproveitar a função de exibir cartas para o mesmo código funcionar nos dois casos, sem precisar ser reescrito?**

8.3 PARA SABER MAIS

Lembre-se que quase sempre existe mais de uma forma de fazer qualquer tarefa de programação. À medida em que você for aprendendo novos métodos, pode voltar neste projeto e praticar nele!

Abaixo, algumas questões que você pode usar para direcionar seus estudos:

- por que os trechos de código que estamos adicionando ao HTML direto pelo JavaScript já aparecem nos lugares certos?
- quais são as formas de se trabalhar com código CSS e quando cada caso é melhor, ou mais utilizado?

Tudo certo? Então é hora de praticar!

8.4 RESUMO

- Adicionando o campo imagem nos objetos com o caminho da imagem;
- Criar uma função que exibe a carta do jogador após o sorteio das cartas;
- Adicionar a moldura da carta;
- Escrever o resultado na tela após o duelo das cartas informando se o jogador venceu ou perdeu;
- Criar uma função que exibe a carta da máquina;
- Exibir os atributos e pontos da carta da máquina.

Clique neste link para acessar o código completo no Codepen.

Capítulo 9

FIGMA, HTML E CSS

9.1 OBJETIVO DA AULA

Nesta nona aula da Imersão Dev, desscobriremos como usar o Figma para um layout em código HTML e CSS para o nosso portfólio! Dessa vez, o código da aula será montado a partir do zero, então para isso você deve criar um novo pen em branco no codepen, clicando na sua imagem do perfil e selecionando `new pen`.

Hoje veremos muita coisa legal. Não somente com JavaScript, mas também com HTML e CSS, três tecnologias principais para o desenvolvimento front-end na web.

Vamos um portfólio com todos os projetos que fizemos ao longo da Imersão Dev, que foram sete.

9.2 FIGMA

Figma é a plataforma que a galera de design utiliza para desenhar como será a página que queremos desenvolver. Depois, o pessoal de front-end pega esse design já pronto, da equipe de designers, e começa a investigar tudo que tem na página que teremos que transformar em código.

Neste link, você encontra o projeto do Figma

9.3 HTML - ESTRUTURA DA PÁGINA

Para começar o nosso código, vou digitar o `<body>` , todo o conteúdo da nossa página estará dentro dele. Repare na sintaxe que estou usando, sempre colocando o sinal de menor e depois o sinal de maior. Para fechar a tag `<body>` , fazemos a mesma coisa mas com uma barra, `</body>` .

Nossa página será dividida em duas estruturas. A primeira é o `<header>` , que no nosso projeto vai ser esse espaço que tem a foto da Rafa, o nome dela e o texto descrevendo o trabalho dela. Além disso, outra tag importante é a `<main>` , que vai ter o conteúdo com os links dos projetos de cada aula.

```
<body>
  <header>

  </header>
    <main>

   </main>
</body>
```

Temos uma imagem, um texto com uma fonte maior e a descrição em fonte menor. Vou colocar essas propriedades dentro do nosso `<header>` . Primeiro colocaremos um `<img>` , que representa a imagem, note que antes de fechar a tag aparece algumas coisas aqui no autocompletar, que podemos usar como propriedades para essa imagem que queremos colocar. Esse src significa "source" e indica qual é o caminho da imagem que queremos utilizar.

9.4 INCLUINDO UMA IMAGEM

Vamos usar a imagem está no meu perfil de uma plataforma chamada GitHub, que é uma plataforma em que você consegue armazenar seus códigos, nela você consegue subir seus códigos e fazer diferentes versões dele. É uma plataforma que usaremos ao longo do tempo e é essencial para quem quer trabalhar na área de desenvolvimento.

Estamos na minha conta do GitHub. Uma coisa que eles fizeram e é muito legal é que basta colocar um .png no final da URL do meu perfil e já conseguimos acessar a minha foto.

Por exemplo:

- Este é o Github do Gui https://github.com/guilhermeonrails
- Para exibir a foto dele usamos o mesmo endereço e adicionamos o `.png`
- A url será `https://github.com/guilhermeonrails.png`

Estamos utilizando o GitHub agora apenas para pegar o caminho da foto, a foto de vocês se já tiverem uma conta no GitHub. Recomendo que vocês já criem uma conta lá para achar tranquilamente a sua foto simplesmente colocando .png no final da URL do seu perfil.

Por questão de segurança, quando colocamos o .png e pressionamos "Enter", a URL muda. Não é essa URL que pegaremos. Nós vamos pegar a URL do perfil mesmo, adicionando o .png no final. No caso do perfil da Rafaella, essa é a URL que pegaremos: https://github.com/rafaballerini.png. Vou colar a URL no nosso código e fechar a tag de imagem com barra e o sinal de maior (/>).

```
<body>
  <header>
      <img class="perfil-foto" src="https://github.com/rafaballer
ini.png" />

  </header>

   <main>

  </main>
</body>
```

9.5 CONTEÚDO DO HTML

O elemento `div` do HTML (ou Elemento de Divisão de Documento HTML) é usado para agrupar elementos para fins de estilo (usando os atributos class ou id). Vamos colocar uma `<div>` para dividir o nosso header para ficar fácil de conseguirmos manter. Dentro dessa `<div>` vou colocar aquele texto em uma tag `<h1>` .

Usamos muito o `<h1>` para título da página, para o que vai ser o texto principal da nossa página. Nesse caso, o título é o nosso

próprio nome no portfólio. por exemplo "Rafaella Ballerini".

Podemos incluir outro texto que aparece embaixo. Você pode usar um `<h2>` ou `<h3>` . O `<h1>` normalmente utilizamos apenas uma vez na página. Vamos inserir um `<h3>` com trecho "Instrutora e desenvolvedora front-end".

```
<body>
  <header>
      <img src="https://github.com/rafaballerini.png" />
        <div>
        <h1>Rafaella Ballerini</h1>
        <h3>Instrutora e desenvolvedora front-end #imersaodev</h3
>
        </div>
   </header>
//código omitido
```

Nós colocamos a imagem, o `<h1>` e `<h3>` , mas ainda não está muito bonito. Nós sabemos que podemos usar o CSS para deixar isso mais bonito.

9.6 CSS

No CSS, normalmente, utilizamos as próprias tags. Existem três tipos de seletores principais que podemos utilizar para estilizar o HTML, por exemplo, se utilizamos a tag `<div>` vai ser aplicado para todas as divs do nosso projeto; a classe vai ser para os elementos que tiverem a classe que escreveremos, pode ser uma classe "perfil", uma classe "título", e os elementos que que colocarmos essa classe terão isso aplicado; e o id sempre utilizamos para estilizar apenas um elemento específico. Quando queremos apenas aquele elemento, daquela forma, colocamos um id nele que vai ser único e não será aplicado em nenhum outro elemento.

Nesse caso, vamos utilizar muito mais o class e as tags para estilizar nossa página. Então, vamos começar a estilizar?

Primeiro começamos estilizando as coisas maiores, começando pelo fundo e depois vamos para os elementos. Vamos começar pela tag `<body>` , que é essa que vai pegar todos os elementos. No CSS você escreve:

```
body {

}
```

Vamos escrever todas as coisas que queremos estilizar, por exemplo, o fundo, a cor da fonte, etc. Vamos lá para o Figma pensar qual é o mais externo de todos os elementos da nossa página. A primeira coisa que podemos pegar é essa cor azul-escura de fundo, que vai ser o nosso background. E como conseguimos descobrir qual é a cor exata? Aliás, ela tem também um efeito gradiente. Para descobrir qual é a cor utilizaremos o próprio Figma.

Podemos clicar em cima da cor e lá na lateral direita tem uma aba vertical chamada "Inspect", de inspecionar, nela conseguimos ver alguma informação como a "colors", as cores, que nos informa que é um "Linear Gradient" (Gradiente Linear) e tem duas cores diferentes. Ele já nos informa quais cores estamos utilizando. Mas além disso, conseguimos ter o código CSS desse fundo. Nessa mesma aba, abaixo de "Colors", tem o "Code CSS" que passa a posição, largura, altura, e o nosso background que é exatamente o que queremos.

Preciso fazer uma observação: às vezes não é legal usar o posicionamento informado pelo Figma porque ele pode ter um

posicionamento dependente de elementos de uma forma diferente. É importante termos controle do posicionamento ao estilizarmos o CSS, então não recomendo utilizar essa parte de posicionamento, altura e largura do Figma. Até porque no CodePen tem um tamanho diferente.

Mas faremos isso aos poucos, vou mostrar como recomendamos que isso seja feito. O que vamos pegar agora é apenas o trecho do código CSS do background. Podemos copiar esse trecho, voltar para o CodePen e colar dentro do body do CSS.

```
body {
  background: linear-gradient(236.85deg, #041832 27.26%, #3468A7
96.03%);
}
```

Vamos salvar e rodar para ver o que vai acontecer. Agora nosso fundo está com a cor certa e com o gradiente linear que vimos no design do Figma.

> Quem quiser pode mudar a cor também. Mexer nesses números do código de cor e testar.

9.7 FONTES

Vamos voltar ao Figma e buscar a fonte certa. É a mesma para a página inteira, desde o nome até os nomes dos projetos. Na aba "Inspect" ao lado, temos a parte de tipografia chamada "Typography", e encontraremos a fonte "Roboto".

Para podermos trazer fontes para o nosso projeto, podemos

utilizar uma plataforma chamada Google Fonts, basta procurar no buscador. Acessando-a, veremos que existem muitas opções e podemos escrever texto para testar se gostamos, e escolher.

Fiquem à vontade para explorar outras fontes, mas por enquanto usaremos a Roboto que é bem conhecida. Encontrando-a na plataforma, clicaremos sobre esta e selecionaremos os pesos ou estilos que queremos, o que também já está no nosso Figma.

O nome está com o número "700" no campo "weight", e o texto de descrição da nossa página está com peso "400", então vamos usar esses dois valores. Quando acessamos os pesos da fonte Roboto no Google Fonts, teremos uma lista com textos de exemplos e seus respectivos valores de peso e estilo no canto superior esquerdo de cada item.

Escolheremos o "Regular 400" e clicaremos no botão de "select this style" nesta opção. Depois, encontraremos o "Bold 700" e selecionaremos este estilo também.

Existe a possibilidade de importarmos a fonte em nosso HTML ou no CSS, pois o Google Fonts oferece a opção de "link" e "import" na opção "Use on the web" na aba "Selected family".

Usaremos no CSS, já que estamos aprendendo como estilizar. Selecionaremos a opção "import", copiaremos o código a partir da tag `<style>` até o final e depois colocaremos no início do nosso código CSS.

É como se estivéssemos de fato importando algo para usarmos ao longo do nosso código, então é importante colocarmos no topo do projeto para usarmos quando quisermos.

```
@import url('https://fonts.googleapis.com/css2?family=Roboto:wght
@400;700&display=swap');

body {
    background: linear-gradient(236.85deg, #041832 27.26%, #3468A
7 96.03%);
}
```

De volta ao Google Fonts, também encontraremos a informação de como utilizá-la, pois apenas a importamos para o nosso projeto, mas não a estamos usando ainda.

Para isso, iremos na aba lateral de "Selected family" e encontraremos o campo de "CSS rules to specify families" com font-family: 'Roboto', sans-serif;, que é a forma como falamos ao CSS que queremos utilizar esta fonte.

Então copiaremos essa linha e a colocaremos dentro do body, porque de fato vamos usar esta fonte para todos os elementos da página.

Vamos salvar e rodar para ver se a fonte está diferente.

Assim já temos nosso fundo e nossa fonte correta.

9.8 ESTILIZANDO UMA DIV

Vamos estilizar o nosso cabeçalho, para isso, podemos primeiro criar uma nova classe para esse conjunto de coisas - imagem, título e subtítulo -, porque sempre estilizaremos do maior para o menor.

Por exemplo, poderíamos puxar a tag `<header>` que é a única que estamos utilizando, mas começaremos usar de fato as classes.

Primeiramente, dentro do `<header>` do HTML, criaremos uma class para este conjunto do cabeçalho. Após o sinal de igualdade e entre aspas, escreveremos container.

O "container" é um agregado de elementos que temos, então essa classe representará o nosso cabeçalho.

Para chamarmos uma tag no CSS, só precisaremos escrevê-la, mas para chamarmos uma class, colocamos um ponto antes do nome, obtendo .container neste caso. Isso para quando quisermos estilizar uma classe.

.

> Se tivéssemos escrito somente header, só usaríamos essas características para o `<header>` , mas como fizemos usando a classe, todos os elementos que tiverem a .container, e talvez não seja somente o `<header>` mesmo, também pegarão essas características.

Agora já temos o nosso body todo estilizado, ainda que não possamos ver muitas coisas na página ainda. Mas já temos estilizadas as coisas que usaremos nela inteira.

Vamos voltar ao Figma para sabermos o próximo passo. temos um fundo azul e um retângulo de fundo branco, o qual contém o cabeçalho do `<header>` e o `<main>` , que é o conteúdo principal.

Então precisaremos criar algo que englobe esses dois conjuntos de elementos. De volta ao nosso código, veremos o que podemos fazer para colocar este retângulo branco.

Temos o `<body>` com o `<header>` e o `<main>` separados no HTML, e precisamos englobá-lo em alguma divisão. Criaremos uma nova `<div>` que engloba desde antes do `<header>` até após o `</main>`.

Assim, poderemos estilizar essa `<div>` para termos o fundo branco do retângulo, pois se estilizarmos o `<body>`, será a parte mais externa do fundo azul. Portanto precisamos de uma divisão interna.

Em seguida, precisaremos criar uma nova classe para esta nova `<div>`. Podemos ter as tags para elementos no geral, mas as classes podem ser definidas para quando quisermos reutilizar a estilização ou definir estilizações diferentes, afinal estamos usando outras `<div>`.

Se estilizarmos a `<div>` da segunda linha do HTML, as linhas de `<h1>` e `<h3>` da `<div>` seguinte receberão essa mesma estilização, e não é o que queremos.

Portanto, criaremos uma nova class que terá o nome igual a "container" para o conjunto. Para estilizarmos uma classe, faremos de forma um pouco diferente em relação às tags.

```
<body>
    <div class="container">
        <header>
            <img src;"https://github.com/rafaballerini.png" />
            <div>
                <h1>Rafaella Ballerini</h1>
                <h3>Instrutora e desenvolvedora front-end #imersa
odev</h3>
            </div>
        </header>

        <main>
```

```
        </main>
    </div>
</body>
```

> Se colocássemos uma div e depois abríssemos chaves, funcionaria também, porém todas as outras divisões mudariam o estilo para este mesmo, e não é isso que precisamos, pois queremos estilizar somente as div que "carimbamos" com as características do container, com especial, título e outra classe que faça sentido.

Começaremos colocando a cor de fundo branca. Então vamos verificar qual é o valor desta cor para nosso background no "Rectangle 1" da aba "Inspect".

```
@import url('https://fonts.googleapis.com/css2?family=Roboto:wght
@400;700&display=swap');

body {
    background: linear-gradient(236.85deg, #041832 27.26%, #3468A
7 96.03%);
    font-family: 'Roboto', sans-serif;
}

.container {

}
```

Há várias cores brancas diferentes, por incrível que pareça. Poderíamos escrever background com dois pontos e white apenas, mas a equipe de design que fez este Figma já falou que este branco não é puro, é mais um cinza super claro, então temos que buscar a informação lá no Figma.

Então pegaremos a cor que já está no nosso "Inspect" e colaremos no nosso código. Ela não tem um gradiente como a do fundo, então é mais tranquilo.

Podemos tanto copiar do código CSS que já vem com a propriedade do background, quanto escrever a propriedade e colocar a cor.

Vamos incluir no `.container`, seguido do valor #ECF4FF da cor de fundo que pegamos do Figma.

Em seguida, vamos salvar e rodar para vermos como nossa página ficou.

```
@import url('https://fonts.googleapis.com/css2?family=Roboto:wght
@400;700&display=swap');

body {
    background: linear-gradient(236.85deg, #041832 27.26%, #3468A
7 96.03%);
    font-family: 'Roboto', sans-serif;
}

.container {
    background: #ECF4FF;

}
```

9.9 POSICIONAMENTO

Para aumentarmos essa área do fundo e diminuirmos o retângulo branco, podemos criar uma margem com a distância que queremos entre o elemento anterior e o atual.

Então aplicaremos a propriedade `margin:` com uns `64` pixels (no CSS, escrevemos apenas px) de espaço. Agora

começamos de fato a trabalhar com tentativa e erro do CSS, pois temos que ir testando se a distância colocada é satisfatória.

Tem algumas dessas informações no Figma, mas às vezes é interessante termos o controle das posições dos elementos em nosso código.

```
@import url('https://fonts.googleapis.com/css2?family=Roboto:wght
@400;700&display=swap');

body {
    background: linear-gradient(236.85deg, #041832 27.26%, #3468A
7 96.03%);
    font-family: 'Roboto', sans-serif;
}

.container {
    background: #ECF4FF;
    margin: 64px;
}
```

Parece que a margem ficou boa com esse espaço entre o fundo azul e o branco, mas podemos alterar se quisermos. A medida em pixels é o tamanho que usamos para distâncias no código.

9.10 SOMBRA E BORDA

Na aba "Inspect" do Figma, encontraremos no código CSS as propriedades `box-shadow:` e `border-radius:` . Para entendermos essas palavras, começaremos pelo border-radius:, que em uma tradução livre do inglês seria o "raio da borda".

O utilizamos para fazermos a borda arredondada do nosso elemento. Quanto maior for o valor deste raio, mais curvados serão os vértices deste elemento.

Para um exemplo, colocaremos o border-radius: igual a vinte pixels. Vamos salvar nosso arquivo e rodar para vermos a diferença.

```
@import url('https://fonts.googleapis.com/css2?family=Roboto:wght
@400;700&display=swap');

body {
    background: linear-gradient(236.85deg, #041832 27.26%, #3468A
7 96.03%);
    font-family: 'Roboto', sans-serif;
}

.container {
    background: #ECF4FF;
    margin: 64px;
    border-radius: 20px;
}
```

A "borda" da nossa divisão `.container` já ficaram arredondadas com vinte pixels.

Voltando ao CSS do Figma para entendermos a propriedade `box-shaddow:` , a qual faz a "sombra" por debaixo do nosso elemento.

Podemos simplesmente copiar e colar no nosso código porque já vem com algumas configurações de tamanho, cor e transparência para aplicarmos.

```
@import url('https://fonts.googleapis.com/css2?family=Roboto:wght
@400;700&display=swap');

body {
    background: linear-gradient(236.85deg, #041832 27.26%, #3468A
7 96.03%);
    font-family: 'Roboto', sans-serif;
}

.container {
```

```
    background: #ECF4FF;
    margin: 64px;
    border-radius: 20px;
    box-shadows: 6px 6px 6px #0E1D2F;
}
```

Feito isso, rodaremos para ver que há uma pequena sombra sob o elemento de fundo branco, a qual é mais visível quando o fundo azul está em tom mais claro, dando uma certa profundidade para o elemento.

9.11 COR DA FONTE

Agora precisaremos estilizar a cor da fonte. Às vezes a cor defaut é a preta mesmo, mas como dissemos sobre a cor branca, há vários tons de preto também.

Vamos voltar ao Figma para pegarmos o código da cor preta utilizada na página, e levá-lo para nosso .container mudar em todos os elementos que contém.

Escreveremos a propriedade `color:` que é a cor dos textos, seguido do código `#1C1C1C` da cor preta, diferente do `background:` que diz respeito à cor do fundo.

```
@import url('https://fonts.googleapis.com/css2?family=Roboto:wght
@400;700&display=swap');

body {
    background: linear-gradient(236.85deg, #041832 27.26%, #3468A
7 96.03%);
    font-family: 'Roboto', sans-serif;
}

.container {
    background: #ECF4FF;
    color: #1C1C1C;
    margin: 64px;
```

```
    border-radius: 20px;
    box-shadows: 6px 6px 6px #0E1D2F;
}
```

A mudança foi bem sutil, mas queremos que seja fiel ao design!

9.12 EDITANDO A FOTO

Precisamos fazer é alterar a posição da foto, que está muito grande e totalmente colada na borda do container, e queremos que tenha algum espaçamento e fique mais para dentro.

Para isso, usamos o `padding:` que atua de uma forma diferente do `margin:` que utilizamos para termos uma margem do elemento externo de fundo azul para o elemento interno de fundo branco que estamos estilizando.

Já o padding: faz diferente; dá uma margem do nosso container para os elementos que estão dentro dele. Colocaremos um valor de sessenta e quatro pixels para vermos como fica.

Vamos salvar e rodar para observar a página.

```
@import url('https://fonts.googleapis.com/css2?family=Roboto:wght
@400;700&display=swap');

body {
    background: linear-gradient(236.85deg, #041832 27.26%, #3468A
7 96.03%);
    font-family: 'Roboto', sans-serif;
}

.container {
```

```
    background: #ECF4FF;
    color: #1C1C1C;
    margin: 64px;
    border-radius: 20px;
    box-shadows: 6px 6px 6px #0E1D2F;
    padding: 64px;
}
```

Já está melhorando!

A primeira coisa que vamos estilizar logo depois da `<div>` é o nosso `<header>` no arquivo HTML. Também colocaremos uma class para ele, a qual poderá ser igual a "perfil" por exemplo.

```
<body>
    <div class="container">
        <header class= "perfil">
            <img src;"https://github.com/rafaballerini.png" />
            <div>
                <h1>Rafaella Ballerini</h1>
                <h3>Instrutora e desenvolvedora front-end #imersa
odev</h3>
            </div>
        </header>

        <main>

        </main>
    </div>
</body>
```

Rafaella: Para a estilizarmos, iremos ao arquivo CSS. e a chamaremos com o ponto, `.perfil` e abre as chaves.

Vamos estilizar do nosso `<header>` , mas se formos ao Figma, não encontraremos nenhum elemento entre o container, a imagem e o cabeçalho que consigamos identificar apenas visualmente.

Precisaremos estilizar o posicionamento dos elementos que irão estar dentro do cabeçalho. Para isso, criaremos uma nova

classe no nosso elemento-pai `<header>` que engloba a imagem e os títulos, e então estilizaremos os elementos-filhos internos.

Queremos posicioná-los da forma correta, e para isso utilizaremos o chamado `FlexBox`, que é uma forma de conseguirmos estilizar dinamicamente os elementos dentro de uma tag-pai, que neste caso é o cabeçalho `<header>`.

Por meio deste FlexBox conseguimos dizer se queremos os elementos na horizontal, alinhados verticalmente, centralizados e etc. É uma forma simples, mas há diversas outras de posicionar, como usando position: absolute ou grid:.

Mas o FlexBox é algo que facilita muito a vida, pois só dizemos se queremos centralizado, alinhado à direita e até se queremos um espaço entre eles por exemplo, e ele faz "sozinho".

> O Flexbox é uma "caixa flexível" e parece mágica. Podemos colocar muitas coisas nessa caixa para termos resultados bem diferentes e personalizados.

Neste caso utilizaremos bem pouco, então ficará bem simples mesmo. Não precisaremos ter um elementos numa posição de "x" e "y", pois o FlexBox já posiciona da forma que queremos, basta dizer. Para o usarmos, precisaremos colocar um `display: flex` em `.perfil` primeiro de tudo. Isso colocamos realmente no elemento-pai, que é o cabeçalho `<header>`.

Feito isso, estamos dizendo que vamos usar o FlexBox de fato. Em seguida, utilizaremos a propriedade `align-items:` que serve

para "alinhar os itens" com center para os alinharmos ao centro.

Salvaremos e rodaremos para vermos a diferença na página.

```
@import url('https://fonts.googleapis.com/css2?family=Roboto:wght
@400;700&display=swap');

body {
    background: linear-gradient(236.85deg, #041832 27.26%, #3468A
7 96.03%);
    font-family: 'Roboto', sans-serif;
}

.container {
    background: #ECF4FF;
    color: #1C1C1C;
    margin: 64px;
    border-radius: 20px;
    box-shadows: 6px 6px 6px #0E1D2F;
    padding: 64px;
}

.perfil {
    display: flex;
    align-items: center;
}
```

Com isso, todos os nossos elementos ficaram centralizados na página, e antes o nome e a descrição estavam embaixo da imagem.

Falamos para o cabeçalho que temos elementos internos que devem ficar centralizados na página, então ficarão ao centro. Mas notaremos que o nome "Rafaella Ballerini" está alinhado com a descrição embaixo.

Para entendermos, iremos ao arquivo HTML novamente. Os elementos-filhos do `<header>` não são a imagem, o título e o subtítulo, e sim a imagem e a `<div>`, a qual engloba outros dois elementos, os `<h1>` e `<h3>`.

Então temos na verdade dois elementos dentro do cabeçalho que estamos estilizando, então este segundo tem duas coisas em comum: o título e o subtítulo, e por isso estão juntos um acima do outro e não centralizados lado a lado.

Estamos fazendo isso justamente porque este é o formato que está no nosso Figma. Portanto é bom termos cuidado quando formos utilizar o FlexBox, prestando atenção aos elementos-filhos e ao pai, mas com o tempo fica mais fácil.

Já centralizamos a imagem e o texto da maneira como queremos, e a primeira coisa que faremos depois será deixar a borda da foto redonda, como vemos no Figma.

Como já vimos, existe a propriedade border-radius: que arredonda os limites de um elemento, até o formato circular. Na aba lateral de "Inspect" com a parte "CSS", encontraremos o valor.

Dentro da nossa tag img, criamos um class e, pode ser, por exemplo, "perfil-foto", isto é, `<img class="perfil-foto"` .

Na verdade, por vezes, o Figma tem um formato diferente de apresentar algumas informações. Ele está colocando a sombra como um filtro e nós podemos pegar as informações que queremos da nossa sombra, que são as que estão dentro dos parênteses e colocar a propriedade que já conhecemos no CSS, que é o box-shadow. Portanto, você pode copiar as informações que estão entre parênteses e colocar, no código, o `box-shadow` .

```
.perfil-foto {
  border-radius: 460px;
  max-height: 160px;
  box-shadow: 0px 4px 4px rgba(0, 0, 0, 0.25);
}
```

9.13 ESTILIZANDO OS TEXTOS

Depois da nossa foto, a próxima coisa que estilizaremos é, de fato, os escritos. O nosso nome e, também, o nosso subtítulo. Agora, vamos ao nosso HTML verificar como ele está. E, após o img, temos a div, que engloba essas duas coisas. Podemos, também, colocar uma classe para a div, a classe título, `<div class="titulo">`.

Seguindo, vamos ao CSS. Escreveremos, .titulo, para estilizarmos nossa classe. No Figma, notaremos que uma das únicas coisas que precisamos estilizar em conjunto, título e subtítulo, é puxá-los um pouco para a esquerda, porque estão um pouco grudados no nosso CodePen com a imagem. É interessante, portanto, puxarmos um pouco a margem da foto, para o nosso título ficar mais para a esquerda.

Voltando ao CodePen, é possível verificar como está essa parte. Está muito grudado! Sendo assim, colocaremos margem, que é a distância do elemento exterior para o que estamos estilizando. Neste caso, margin-left, porque é apenas a da esquerda. Não queremos colocar margem para cima, para baixo ou para a direita. A nossa margem terá 16px.

```
.titulo {
  margin-left: 16px;
}
```

Separou um pouco mais! Ficou melhor. Agora, estilizaremos o nosso h1 e o h3. Nós não utilizaremos o h1 em outro lugar, porque, normalmente, precisamos apenas de um, e h3 também não. Portanto, mais abaixo, nós usaremos outros tipos de tags. Podemos até usar a mesma tag para que vocês vejam usos distintos

do h1 e h3, mas, algo interessante é unir a classe do elemento pai, exterior.

Por exemplo, no nosso h1 e h3, o elemento de cima é o div, isto é, `<div class="titulo">` , que é o título. Nós podemos utilizar a classe desse elemento, dar um espaço e colocar a tag que queremos estilizar. Assim, deixaremos um pouco mais específico que apenas a tag h1 estará dentro da classe título.

```
 <header class= "perfil">
      <img class="perfil-foto" src="https://github.com/rafaballer
ini.png" />
      <div class="titulo">
        <h1>Rafaella Ballerini</h1>
        <h3>Instrutora e desenvolvedora front-end #imersaodev</h3
>
      </div>
</header>
```

No nosso CSS, faremos `.titulo h1 {` . Nós queremos estilizar a tag h1, que está necessariamente dentro de um elemento com classe título. Estamos especificando de uma forma um pouco diferente do que quando colocamos apenas uma classe para ela. É uma maneira bem interessante e amplamente utilizada.

Seguindo, vamos estilizar a nossa fonte, o nosso escrito. Primeiro, nós havíamos separado dois pesos diferentes de fonte, a 400 e 700. A 700 era, justamente, a do nosso título. Podemos até verificar no Figma. Portanto, a primeira coisa que faremos é o font-weight: 700;. Nós também consultaremos o tamanho da nossa fonte, porque, por padrão, existe um tamanho de fonte que o próprio CSS deixa na nossa página.

É interessante conferir qual foi o tamanho escolhido pela/o designer. Para isso, vamos ao Figma verificar o tamanho da fonte

também. O tamanho é 36px. Vamos colocar font-size:, que é "tamanho da fonte", igual a 36 px, font-size: 36px;, salvar e executar. O tamanho é 24px e o peso, 400px.

```
.titulo h3 {
  font-weight: 400;
  font-size: 24px;
}
```

Acho que, nesse, teremos alguma diferença, já que ele está bem maior. Vamos ver? O peso mudou bastante e o tamanho também. Ficou muito legal a parte do cabeçalho e temos ainda muitos detalhes pela frente, para deixar bem mais bonita parte abaixo. Vamos para a segunda parte!

9.14 CRIANDO A LISTA DOS PROJETOS

No HTML. existe a `ul` , que é uma lista não ordenada e a `ol` , que é uma lista ordenada. Neste caso, não há ordem, o 1 e 2 não são passos de uma receita que devemos seguir. Ou seja, ela não precisa estar ordenada, por isso, usaremos `ul` .

Vamos dar o título da nossa lista, `<ul>Projetos<ul>` , e, cada linha dessa nossa lista representará um projeto. Vamos colocar uma `<li></li>` , **sem esquecer de fechá-la**. Ao lado, colocaremos o view do projeto, para focarmos no HTML. A primeira coisa que faremos é criar uma tag âncora. ncora se refere às questões que estamos super acostumados a fazer, mas nunca paramos para pensar no que de fato está acontecendo.

Quando selecionamos determinada palavra, que abre uma outra tag, estamos selecionando uma "tag âncora", representada pela letra "a". Essa tag a, nós também adicionaremos e fecharemos,

`<li><a></a></li>` . Nós abrimos a li, abrimos a tag a e fechamos a li. Dentro, nós colocaremos texto. Repare que não estamos colocando o texto dentro a, mas, sim, entre eles. Nós colocaremos o nome do projeto que fizemos.

Vamos inserir dois projetos que desenvolvemos durante a imersão, por isso os colocarei. Um deles é o "Conversor de moeda", `<li><a>Conversor de moeda</a></li>` . Já é possível ver como aparecerá na tela: "Projetos", e, na linha abaixo, um marcador em formato de círculo e, à frente, "Conversor de moeda". Não consigo clicar nele, porque ainda não adicionamos ainda o link para onde queremos ir no projeto.

Dentro da nossa tag a, colocaremos outra propriedade para indicar o link da tag âncora, que colocaremos com href="" e, entre aspas, o endereço do projeto. Para isso, abriremos o CodePen e pegaremos o link para que, ao clicar, a pessoa seja direcionada para a página do CodePen. Nós selecionaremos, portanto, este endereço com "Ctrl + C", o colocaremos no href, salvamos e rodamos.

```
<main class="projetos">
      <ul>Projetos<ul>
        <li><a href="https://codepen.io/imersao-dev/pen/39044b884
fd63387c4b075671051ce1a">Conversor de moeda</a></li>

                </main>
            </div>
        </body>
```

Repare que a escrita "Conversor de moeda" aparece até de outra cor. Quando apertamos o link, somos direcionados para a aula do Conversor de Moeda. Vamos retornar, pois, não queremos ficar naquela aula agora. Podemos fazer isso para todas as outras aulas. A diferença desse código para a próxima aula, por exemplo,

será apenas o link e o nome.

Vamos testar com o projeto do Aluraflix, que ficou bonito. Basta copiar e colar o código dele, salvar e executar.

```
 <main class="projetos">
      <ul>Projetos<ul>
        <li><a href="https://codepen.io/guilhermeonrails/pen/poPZ
Gov?editors=0111">Conversor de moeda</a></li>
        <li><a href="https://codepen.io/imersao-dev/pen/XWpWrod"
> Aluraflix</a></li>
    </main>
  </div>
</body>
```

Ao selecionar o primeiro link com o botão direito, acessamos o projeto da aula 2, "Conversor de Moeda". Selecionando o segundo link, da mesma forma, acessamos a aula 5, "Aluraflix". Nós fizemos a estrutura, mas, esteticamente, está feio. Especialmente se comparado com a parte do nosso header, em que a imagem está estilizada, circular, nossos links não estão com uma boa apresentação.

9.15 EMOJIS

A primeira coisa que conseguimos verificar é: no nosso Figma, temos uma imagem para cada projeto que fizemos. Essas imagens não são exatamente imagens, mas, sim, emojis, que podemos utilizar no nosso sistema operacional. É como se estivéssemos copiando e colando o texto. Ele é lido, não é uma imagem, como a src, que precisávamos copiar e colar.

De fato, é algo que conseguimos copiar como texto e colar. Para isso, temos o site bag emoji, que conta com uma grande quantidade de emojis. Nós podemos usar, por exemplo, o emoji da

calculadora de notas. Vamos pesquisar por "number". Podemos passar o mouse por cima e copiar o próprio emoji. Algo interessante para quem usa o Windows é que, basta apertar a tecla "windows" e o ponto para algo mágico acontecer. Testem em casa.

> No Chrome, no "Menu Edit" também temos emojis, uma lista com vários. No “Edit”, encontramos "Emojis e símbolos", é possível escolhê-los e usá-los.

No Conversor de Moeda, coloquei um emoji de número, mas, no Aluraflix, o que vocês acham que eu deveria colocar? Sinta-se a vontade para incluir o emoji que preferir.

Essa foi a primeira coisa que colocamos para deixar um pouco melhor esteticamente. Vamos rodar e ver como ficou. Está ótimo, mas ainda padrão. Podemos estilizar, lembrando de sempre pensar primeiro nas coisas mais externas e depois nas mais internas. Após a nossa tag de header, o nosso cabeçalho, nós tínhamos, em seguida, a tag main, que tem o conteúdo principal da nossa página. Nós daremos uma classe para ela.

Para isso, voltaremos ao HTML e colocaremos a classe dentro da main, uma `classe="projetos"` , que será, de fato, o conteúdo dos projetos que temos.

```
<main class="projetos">
      <ul>Projetos<ul>
        <li><a
```

No CSS, nós estilizaremos o `.projetos` e temos muitas coisas a pensar pela frente. A primeira delas é, de fato, o nosso flexbox,

que estávamos usando para posicionar os elementos que estão dentro dessa tag, que, no caso, formarão a nossa lista. Então, nós usamos o display: flex; para indicarmos que queremos usar o flexbox.

Seguindo, nós utilizaremos uma propriedade do flexbox que é o flex-direction, a direção dos elementos que estarão dentro. Como eu já havia comentado, pode ser horizontal, vertical, enfim, existem várias propriedades e nós disponibilizamos o link de um artigo para que vocês confiram todos.

Nesse caso, nós utilizaremos uma direção vertical, de coluna, por isso, adicionaremos column, pois queremos que a nossa lista fique estilizada nesse sentido, o display: flex normalmente vem como horizontal, portanto, é importante ajustar o flex-direction como column.

```
.projetos {
  display: flex;
  flex-direction: column;
}
```

Além disso, vamos alinhar todos os nossos elementos no centro. Usaremos a propriedade do align-items: center;, para deixá-los mais centralizados. Agora, basta salvar e rodar. Vamos ver como ficará com essa estilização.

É preciso cuidado, porque, pensando sobre o design, eu sei que eles estudam muitas coisas, então, por trás existem diversos motivos pelo qual é centralizado, por exemplo. Motivos que nós não entendemos, mas que estão relacionados com experiência do usuário e que, realmente, fazem muito sentido. O ideal é sempre perguntar, "será que não seria legal mudar?" e entender o motivo de estar de determinada maneira.

A imagem, a questão de pegar um h3 e um h1 de uma determinada class, ficou muito interessante. Nós fizemos esse layout do zero.

Tudo certo? Então é hora de praticar!

9.16 RESUMO

- Aprendemos a mexer no Figma e transformar o design em código;
- Entendemos melhor como funciona HTML e CSS;
- Estruturamos o nosso portfólio com HTML, aprendendo todas as tags necessárias pra isso;
- Estilizamos o nosso portfólio com CSS, conhecendo os seletores, propriedades e valores necessários para isso.

Clique neste link para acessar o código completo no Codepen.

Capítulo 10
PORTFÓLIO

Estamos na nossa última aula da imersão! Eu sei que ainda vamos nos ver por aqui, mas queria já deixar os parabéns por você ter feito tudo durante essas duas semanas. Não é fácil aprender programação, e o fato de você ter chegado até aqui já é muita coisa.

10.1 OBJETIVO DA AULA

Vamos terminar o projeto que fizemos ontem, que foi nosso portfólio. Enquanto fazemos isso, ainda vamos incluir outras coisas bem legais.

A ideia dessa aula, vocês podem ver nessa tela do Figma. Ter um botão para alterar o tema para claro ou escuro (*dark mode*) e uma aba de projetos (que vamos evoluir ainda mais. Além disso, nessa parte do projeto, conseguiremos visualizar trechos do código e executar o projeto feito, como a calculadora, o mentalista e assim por diante.

10.2 ESTILIZANDO A LISTA DE PROJETOS

Portfólio é algo para ser bonito, não é? Precisa ser convidativo, além de ter links para seu e-mail, seu Linkedin, algumas coisas que

passei no desafio. Inclusive, se você quiser estilizar com outras cores, fique à vontade. No CSS, temos uma classe .projetos que engloba toda a parte azul do Figma, onde temos os nossos projetos.

Atualmente temos nela um `display: flex` , que posiciona os elementos filhos dentro dele (no caso a nossa lista), e o `flex-direction: column` , que define a direção como colunas.

```
.projetos {
            display: flex;
            flex-direction: column;
}
```

Além de posicionar os elementos, precisamos colocar nosso background. No Figma, vemos que a cor dessa parte é o azul (linear-gradient), o mesmo que tínhamos usado no .

```
.projetos {
  display: flex;
  flex-direction: column;
  background: linear-gradient(230.65deg, #499cfe 27.49%, #9cc8fc
83.19%);
}
```

Vamos salvar e rodar para verificarmos como está ficando. Com essa alteração, a seção "Projetos" ganhará um fundo com um gradiente azul. Uma coisa que está incomodando um pouco é que nossa foto de perfil está muito "grudada" nessa caixa, e seria interessante distanciarmos um pouco. Como vimos anteriormente, isso é possível adicionando uma margem, nesse caso margin-top para distanciarmos da parte de cima.

```
.projetos {
  display: flex;
  flex-direction: column;
  background: linear-gradient(230.65deg, #499cfe 27.49%, #9cc8fc
83.19%);
  margin-top: 32px;
```

```
}
```

Legal, bem melhor. Também podemos ajustar os elementos dentro de .projetos um pouco mais espaçados, algo que faremos com o `padding` .

```
.projetos {
  display: flex;
  flex-direction: column;
  background: linear-gradient(230.65deg, #499cfe 27.49%, #9cc8fc
83.19%);
  margin-top: 32px;
  padding: 32px;
}
```

Quando utilizamos apenas padding ou apenas margin, estamos adicionando essa propriedade a todos os cantos. Já quando usamos uma especificação, como margin-top, estamos aplicando a um canto específico (cima, nesse caso). Vamos analisar o projeto no Figma para entendermos o que mais podemos estilizar. Ainda temos um link de redirecionamento para os projetos, mais tarde faremos com que as telas do CodePen apareçam. Por enquanto vamos focar na caixa azul dos nossos projetos. Ainda falta incluirmos um border-radius, que deixa o elemento arredondado, e o box-shadow, que faz uma sombra na parte inferior. Vamos copiar esses elementos e incluir em nosso .projetos.

```
.projetos {
  display: flex;
  flex-direction: column;
  background: linear-gradient(230.65deg, #499cfe 27.49%, #9cc8fc
83.19%);
  margin-top: 32px;
  padding: 32px;
  box-shadow: 2px 2px 4px rgba(16, 16, 16, 0.42);
  border-radius: 20px;
}

.projetos-titulo {
```

```
  list-style: none;
  font-weight: 700;
  font-size: 36px;
}
```

Já temos as bordas arredondadas e a sombra, e agora precisamos estilizar os textos, tanto o título da lista quanto os elementos dela. Primeiramente, criaremos uma classe para o título da lista, que é a nossa

. No HTML, criaremos uma class="projetos-titulo"

```
<main class="projetos">
    <ul class="projetos-titulo">Projetos<ul>
```

No CSS, incluiremos um .projetos-titulo e abriremos chaves para começarmos a estilizar.

```
.projetos-titulo {

}
```

Primeiramente, removeremos o estilo dessa lista, o que inclui o ponto e a indentação. Quando queremos estilizar e personalizar um elemento, é interessante removermos a decoração padrão que vem do HTML e do CSS, algo que fazemos com list-style: none (de "nenhum").

```
.projetos-titulo {
  list-style: none;
}
```

Em seguida, alteraremos o peso da fonte (font-weight) para 700, seguindo o nosso projeto do Figma. Além disso, alteraremos o tamanho da fonte (font-size) para 36px.

```
.projetos-titulo {
  list-style: none;
  font-weight: 700;
```

```
  font-size: 36px;
}
```

Ficou mais ou menos, né? O título tá legal, mas os outros elementos estão estranhos. Os outros ainda precisamos arrumar. Agora adicionaremos uma classe para os itens da lista, que são nossos

- , chamada projetos-item. Vamos criar uma estrutura que será reutilizada em outras partes.

Assim como fizemos ali em cima, colocar o

- dentro de todos os projetos, por exemplo, como fizemos o

DENTRO DE TODAS AS DE PERFIL, MAS AMBOS OS JEITOS FUNCIONAM. AQUI VAMOS MOSTRAR COMO UTILIZAR UMA CLASSE PARA MAIS DE UM ELEMENTO.

```
<main class="projetos">
            <ul class="projetos-titulo">Projetos<ul>
                                    <li class="projetos-item"><a
href="https://codepen.io/guilhermeonrails/pen/poPZGov?editors=011
```

```
11">Conversor de Moeda</a></li>
                                    <li class="projetos-item"><a
href="https://codepen.io/guilhermeonrails/pen/yLbxpwm">Aluraflix<
/a></li>
                                    </li>
```

Vamos estilizar o nosso item da lista. No CSS, criaremos um .projetos-item e abriremos chaves. Para removermos os pontos antes de cada elemento da lista, usaremos o list-style-type: none.

```
.projetos-item {
            list-style-type: none;

}
```

Além disso, alteraremos o tamanho da fonte (font-size) e o peso (font-weight). Por fim, temos também uma propriedade line-height, que é a altura da linha, que é interessante adicionarmos. Dessa forma, teremos mais distanciamento entre cada linha da nossa lista.

```
.projetos-item {
  list-style-type: none;
  font-size: 24px;
  font-weight: 400;
  line-height: 48px;
}
```

Falta estilizarmos as `"âncoras"` dos nossos hiperlinks, que deixam um sublinhado no texto. Para alterarmos isso, usaremos uma classe .projetos-item a. Assim, toda tag que esteja dentro de um elemento projetos-item terá essa estilização.

Adicionaremos um `color: #1c1c1c` para alterarmos a cor do texto para a mesma dos outros elementos e removeremos a decoração do texto com text-decoration: none.

```
.projetos-item a {
  color: #1c1c1c;
```

```
  text-decoration: none;
}
```

Com isso, nosso projeto estará padronizado e poderemos clicar sobre o link do "Conversor de Moeda" da nossa página.

Imediatamente já identifica como um link, e direciona para a nossa página do Conversor de Moedas.

Rafaella: Exatamente, e poderíamos colocar links do LinkedIn, Facebook e outras páginas, basta mudar o href que colocamos o link do CodePen.

10.3 ABRINDO OS PROJETOS NO PORTFÓLIO

Ficou bem bacana este link sem o texto padrão que vem do HTML. Mas para nosso projeto principal, abriremos uma lista com caixinhas contendo os projetos que fizemos para visualizarmos.

Ou seja, queremos abrir apenas um pedaço do projeto que estamos pegando do nosso projeto no CodePen para exibir. Para isso, clicaremos com o botão direito sobre o retângulo que aparece com os projetos no Figma e abriremos o Conversor de Moedas em uma nova aba.

Neste projeto que fizemos, encontraremos um botão chamado "Embed" na barra inferior, ao lado de "Fork". Clicando em "Embed", abriremos uma caixa chamada "Embed This Pen" que apresenta algumas opções, como a de "Theme" para escolhermos o tema, que pode ser "Default", "Light", "Dark" ou "User Default".

Na aba de "HTML (Recommended)", encontraremos uma tag

com várias coisas, como o usuário, o tamanho que irá aparecer

e etc.

Copiaremos todo este conteúdo, que não é CSS nem JavaScript, e sim um parágrafo HTML. Em nosso projeto, abriremos a parte do HTML e tiraremos a tag-âncora .

Ao invés da tag , deixaremos a

ainda com o título de "Conversor de Moeda" para não o perdermos, pois também temos isso no Figma.

Tiraremos o href também e colocaremos a tag

nas duas linhas de

- .

```
//código omitido

    <main class="projetos">
        <ul class="projetos-titulo">Projetos"</ul>
            <li class="projetos-item"><p> Conversor de moeda</p><
/li>
            <li class"projetos-item"><p> Aluraflix</p></li>
        </main>
    </div>
</body>
```

De volta à caixa "Embed The Pen", copiaremos todo o conteúdo da tag

com a classe "codepen" e todas as propriedades, e depois colaremos embaixo da

que contém os títulos no nosso projeto.

Rafaella: Podemos pular uma linha se quisermos, e depois salvamos.

```
//código omitido

    <main class="projetos">
        <ul class="projetos-titulo">Projetos"</ul>
            <li class="projetos-item">
                <p> Conversor de moeda</p>
                <p class="codepen" data-height="300" data-theme-i
d="dark" data-default-tab="html,result" data-slug-hash="poPZGov"
data-user="guilhermeonrails" style="height: 300px; box-sizing: bo
rder-box; display: flex; align-items: center; justify-content: ce
nter; border2px solid; margin: 1em0; padding: 1em;">
                    <span> See the Pen <a href="https://codepen.i
o/guilhermeonrails/pen/poPZGov">
                        Conversor de moedas - ID3</a> by @guilima
dev (<a href="https://codepen.io/guilhermeonrails">@guilhermeonra
ils</a>)
                    on <a href="https:codepen.io">CodePen</a>.</s
pan>
                </p>
                <script async src="https://cpwbassets.codepen.io/
assets/embed/ei.js"></script>
            </li>

            <li class"projetos-item"><p> Aluraflix</p></li>
        </main>
    </div>
</body>
```

Na página, veremos que aparece o HTML, o CSS e o JavaScript do projeto Conversor de Moeda. Ao centro e acima deste espaço com o código, há o botão de "Result" para vermos a página do Conversor de fato.

Se colocarmos um valor "10" no campo de "Insira o valor" por exemplo, e depois clicarmos em "Converter", receberemos a mensagem de que "o resultado em real é R$50".

Ou seja, estamos rodando o nosso projeto dentro do portfólio direto pelo CodePen. Podemos escolher como queremos deixar o título, o alinhamento, o tamanho, é interessante explorar!

Podemos estilizar o quanto quisermos, mas por enquanto está bom.

Para vermos funcionando de verdade, abriremos o outro projeto chamado "Aluraflix" feito no quinto dia da Imersão em uma outra aba. Ao lado do botão "Fork", encontraremos o botão "Embed" novamente.

Clicando sobre este, teremos a caixa "Embed This Pen" também com a geração de um texto na tag

fornecido na aba "HTML (Recommended)". Copiaremos todo este código da tag

e colaremos em nosso projeto, da mesma forma que fizemos com o "Embed" do Conversor de Moeda.

Inclusive, esse botão "Embed" existe no YouTube e em outros lugares quando quisermos colocar um vídeo, um CodePen mesmo, uma postagem do Instagram ou Facebook e etc.

Usamos o termo "embedar" para fazer isso no dia a dia, ou seja, colocamos algo dentro de outra coisa.

```
//código omitido

    <main class="projetos">
      <ul class="projetos-titulo">Projetos<ul>
          <li class="projetos-item">
            <p> Conversor de moeda</p>
            <p class="codepen" data-height="300" data-theme-id="d
ark" data-default-tab="html,result" data-slug-hash="poPZGov" data
-user="guilhermeonrails" style="height: 300px; box-sizing: border
```

```
-box; display: flex; align-items: center; justify-content: center
; border: 2px solid; margin: 1em 0; padding: 1em;">
              <span>See the Pen <a href="https://codepen.io/guilh
ermeonrails/pen/poPZGov">
                  Conversor de moedas - ID3</a> by @guilimadev (<
a href="https://codepen.io/guilhermeonrails">@guilhermeonrails</a
>)
                on <a href="https://codepen.io">CodePen</a>.</spa
n>
            </p>
            <script async src="https://cpwebassets.codepen.io/ass
ets/embed/ei.js"></script>
          </li>
          <li class="projetos-item">
            <p> Aluraflix</p>
            <p class="codepen" data-height="300" data-theme-id="d
ark" data-default-tab="html,result" data-slug-hash="yLbxpwm" data
-user="guilhermeonrails" style="height: 300px; box-sizing: border
-box; display: flex; align-items: center; justify-content: center
; border: 2px solid; margin: 1em 0; padding: 1em;">
              <span>See the Pen <a href="https://codepen.io/guilh
ermeonrails/pen/yLbxpwm">
                  Aluraflix - dia 5</a> by @guilimadev (<a href="
https://codepen.io/guilhermeonrails">@guilhermeonrails</a>)
                on <a href="https://codepen.io">CodePen</a>.</spa
n>
            </p>
            <script async src="https://cpwebassets.codepen.io/ass
ets/embed/ei.js"></script>
          </li>
    </main>
  </div>
</body>
```

Pronto, agora temos os dois projetos sendo carregados na página, o Conversor de Moeda e o Aluraflix.

Não faremos para todas as aulas da imersão mas como o processo é bem tranquilo e repetitivo, fica o desafio para vocês!

10.4 DARK MODE

Caminhamos um pouco nesta parte do "Embed", e ficou bem legal. Mas ainda queremos saber como fazer um botão de alterar tema para clicarmos e mudarmos a cor de fundo da página para termos o tema "dark" por exemplo, parecido com o que há no GitHub.

Para fazermos este botão no nosso cabeçalho, voltaremos ao Figma para vermos como deve ser.

Ele ficará no canto superior direito do container com a legenda "Alterar o tema". De volta ao código, acessaremos a tag

e criaremos uma outra

após o título.

Portanto serão três elementos principais no nosso cabeçalho: a imagem, o título com o nome e o que fazemos, além do botão que altera o tema.

Essa divisão terá uma classe chamada "tema" que iremos estilizar. Dentro dela, colocaremos um botão de fato com a tag .

Depois, usaremos o onClick que vimos ao longo de todo o

curso. Sempre pegávamos o que já vinha no código feito por outra pessoa no HTML e CSS com toda a função.

Desta vez criaremos a função que utilizaremos depois para podermos alterar o tema, a qual pode ser chamada "mudaTema()".

Não faremos isso imediatamente porque vamos estilizar primeiro, mas já deixaremos a função dentro do onClick no JavaScript que usaremos.

Fecharemos essa tag e depois colocaremos o texto "Alterar tema" para a pessoa saber o que o botão faz.

Podemos salvar e rodar para vermos como está até agora.

```
<body>
  <div class="container">
    <header class="perfil">
        <img class="perfil-foto" src="https://github.com/rafaball
erini.png" />
        <div class="titulo">
          <h1>Rafaella Ballerini</h1>
          <h3>Instrutora e desenvolvedora front-end #imersaodev</
h3>
        </div>
      <div class="tema">
        <button onclick="mudaTema()">Alterar Tema</button>
      </div>
    </header>

//código omitido

  </div>
</body>
```

A primeira coisa que faremos é estilizar de fato. Vamos ao arquivo com o código CSS no CodePen, e o ideal seria seguirmos

uma linearidade dos elementos que vamos escrever para sabermos onde podemos achá-los quando entrarmos no HTML.

Então é interessante irmos até a parte do cabeçalho e alterar de lá mesmo. Temos o .perfil, o .perfil-foto, .titulo, .titulo h1 e o .título h3.

Antes de .projetos, colocaremos a estilização da nossa classe de .tema. Entre as chaves, colocaremos o button para estilizarmos diretamente de dentro da classe, a qual é a

que engloba esse botão.

Precisaremos usar algumas propriedades novas, pois se trata da estilização de um botão. Primeiro, vamos alinhá-lo de uma forma diferente.

Os itens que já alinhamos estão todos à esquerda, e queremos que só o botão fique sozinho à direita. Como estamos estilizando com Flexbox, existe também a propriedade align-self: para isso, sendo flex-end para deixarmos ao final no canto direito do conjunto.

Vamos salvar e executar para vermos onde ficou.

```
//código anterior omitido

.titulo h1 {
  font-weight: 700;
  font-size: 36px;
}

.titulo h3 {
  font-weight: 400;
  font-size: 24px;
}
```

```
.tema button {
    aligh-self: flex-end;
}

.projetos {
  display: flex;
  flex-direction: column;
  background: linear-gradient(230.65deg, #499cfe 27.49%, #9cc8fc
83.19%);
  margin-top: 32px;
  padding: 32px;
  box-shadow: 2px 2px 4px rgba(16, 16, 16, 0.42);
  border-radius: 20px;
}

//código posterior omitido
```

Ficou bem no canto direito mesmo.

10.5 AJUSTANDO A TIPOGRAFIA

O próximo passo é irmos ao Figma para vermos como o botão está no projeto. Clicando sobre este, iremos até as configurações de "Typography" na aba lateral e veremos qual é o tamanho da fonte usada. O tamanho é vinte e quatro pixels com peso de quatrocentos.

Então vamos estilizar isso. podemos copiar do Figma e colar no nosso arquivo CSS também. Portanto o font-size: fica 24px e o font-weight é 400.

Vamos rodar e ver o resultado na página.

```
//código anterior omitido

.titulo h1 {
  font-weight: 700;
  font-size: 36px;
}
```

```
.titulo h3 {
  font-weight: 400;
  font-size: 24px;
}

.tema button {
    align-self: flex-end;
    font-size: 24px;
    font-weight: 400;
}

.projetos {
  display: flex;
  flex-direction: column;
  background: linear-gradient(230.65deg, #499cfe 27.49%, #9cc8fc
83.19%);
  margin-top: 32px;
  padding: 32px;
  box-shadow: 2px 2px 4px rgba(16, 16, 16, 0.42);
  border-radius: 20px;
}

//código posterior omitido
```

10.6 BORDAS DO BOTÃO

Outra estilização que podemos fazer é darmos uma distância um pouco melhor do escrito para as bordas do botão, para que o texto fique um pouco mais centralizado.

Isso é o padding:, pois estamos alterando o elemento que queremos e estamos colocando o que está dentro dele mais para dentro.

Colocaremos esta propriedade no .tema button, e colocaremos 8px seguido de 16px. Estamos colocando duas informações nos dois números, em que a primeira diz respeito à distância que há entre o topo e o elemento interno, e entre a base e o elemento

interno.

Já a segunda informação é relativa às distâncias laterais entre os limites laterais do botão e o elemento interno do texto. A mesma coisa acontece para o margin:, então se quisermos dar uma margem acima e embaixo e outra de um lado e de outro, colocamos dois valores seguidos para cada um respectivamente.

Outra coisa é que o botão está com o fundo cinza, diferente do projeto no Figma, então precisamos encontrar a cor certa para usarmos. Ao selecionarmos o botão de "Alterar tema", teremos o valor na parte "Colors" para copiarmos.

Portanto o `background:` será da cor `#ECF4FF` .

O próximo passo é transformarmos as bordas do botão em arredondadas com o já conhecido border-radius:. De volta ao Figma, copiaremos o valor da propriedade de 100px e colaremos no nosso .tema button.

Outra coisa que faremos é passar a nossa borda, pois ela ainda está um pouco simples demais, e podemos deixá-la melhor definida. Podemos escolher sua espessura, seu tamanho e a cor que queremos.

```
//código anterior omitido

.titulo h1 {
  font-weight: 700;
  font-size: 36px;
}

.titulo h3 {
  font-weight: 400;
  font-size: 24px;
}
```

```
.tema button {
  align-self: flex-end;
  font-size: 24px;
  font-weight: 400;
  padding: 8px 16px;
  background: #ECF4FF;
    border-radius: 100px;
}

.projetos {
  display: flex;
  flex-direction: column;
  background: linear-gradient(230.65deg, #499cfe 27.49%, #9cc8fc
83.19%);
  margin-top: 32px;
  padding: 32px;
  box-shadow: 2px 2px 4px rgba(16, 16, 16, 0.42);
  border-radius: 20px;
}

//código posterior omitido
```

Porém, ainda não conseguimos colocar o botão no exato canto superior direito que precisamos, pois por enquanto está apenas alinhado à direita e centralizado horizontalmente com os demais elementos do cabeçalho.

Então vamos separar em mais uma divisão, e em uma teremos o nome e o título com o subtítulo, e em outra teremos o botão com suas configurações.

De volta ao nosso arquivo HTML, englobaremos a foto com o texto do cabeçalho dentro de uma tag

abaixo do

, cuja classe chamaremos de "perfil" com as informações do perfil.

Já o

será chamado de "cabeçalho", pois englobará os elementos de perfil e o botão de alterar o tema.

```
<body>
  <div class="container">
    <header class="cabecalho">
      <div class="perfil">
        <img class="perfil-foto" src="https://github.com/rafaball
erini.png" />
        <div class="titulo">
          <h1>Rafaella Ballerini</h1>
          <h3>Instrutora e desenvolvedora front-end #imersaodev</
h3>
        </div>
      </div>
      <div class="tema">
        <button onclick="mudaTema()">Alterar Tema</button>
      </div>
    </header>

//código omitido
    </div>
</body>
```

Perfeito. Vamos ver como o nosso CSS está. O perfil agora vai ser a nossa imagem junto com as nossas informações. Vamos precisar mudar a disposição dos elementos, podemos apagar o `align-items: center;` e acima desse perfil vamos estilizar o nosso cabeçalho.

Lembrando que nosso cabeçalho tem a tag perfil e a tag tema, que é o botão e o conjunto de foto com os títulos. Nesse `.cabecalho` do CSS colocaremos `display: flex` , para posicionar os elementos, e `justify-content: space-between` , para justificar os elementos. Queremos que tenha espaço entre os elementos, com o `space-between` colocaremos um espaço entre o perfil, com a foto e o título, e o nosso tema.

```
.cabecalho {
  display: flex;
```

```
  justify-content: space-between;
}
```

Já mudou. Comparado com o projeto do Figma está correto. Agora queremos fazer o botão funcionar. Como fazemos a parte do JavaScript para ele funcionar?

10.7 MUDANDO A COR

Por enquanto, ao clicar no botão só aparece no console que "mudaTema is not defined". A primeira coisa a fazer é criar essa função function mudaTema(). Vamos fazer alguma coisa dentro dessa função que vai manipular o CSS. Como fazemos isso? Vamos criar uma função no Javascript chamada `mudaTema` .

```
function mudaTema() {

}
```

Assim como temos aquele `document.getElementById()` , `document.getElementByName()` , entre outras coisas que usamos no JavaScript para manipular o HTML e o CSS, temos o `document.body()` . Esse body é a tag do HTML, que engloba todos os elementos que são visíveis na nossa página.

E é justamente aí que queremos atacar. Queremos pegar o nosso body e atribuir uma classe para ele. Ao atribuir essa classe, teremos no nosso CSS um CSS para essa classe. Vamos criar uma "classe dark", por exemplo. Vamos escrever no nosso CSS a estilização para essa classe, mas é o JavaScript que vai decidir quando essa classe entra e quando ela sai. Nesse caso, será no onClick.

No onClick do nosso botão de tema queremos alterar a classe

do nosso body. Então colocaremos document.body.classlist.toggle(), toggle é o comando que faz isso, traduzido para o português significa "alternância". Vamos alternar a classlist, a classe do nosso elemento body, e colocar dark como parâmetro para o nosso elemento.

```
function mudaTema() {
  document.body.classList.toggle("dark");
}
```

Porém, ainda não vai mudar nada em relação ao nosso tema. Porque ainda não fizemos a estilização da classe dark. Ao clicar no botão, ele vai mudar para a classe dark, mas ainda não temos a estilização desse estilo no CSS.

10.8 CSS DO DARK MODE

Vamos escrever esse código no CSS abaixo do código que já temos. Vamos inserir o `.dark` . Ao estilizar esse `.dark` , o que estamos estilizando de fato? A nossa tag body quando ela tiver a classe dark. O que normalmente estilizamos na tag body é o background, a fonte - a fonte vai ficar a mesma, não precisa trocar - mas o background vamos trocar. Vamos verificar no Figma qual é a cor do background do tema escuro.

Vou clicar na cor do dark mode e copiar o código CSS da cor e do linear-gradient do background para colar no nosso CSS.

```
.dark {
  background: linear-gradient(236.85deg, #375b86 27.26%, #6b87a9
96.03%);
}
```

Vamos salvar e rodar para ver se o botão está funcionando?

O fundo já está sendo alterado. Vamos, agora, estilizar o resto das coisas. Abrindo o Figma para ver o que mais precisamos alterar no nosso CSS. Precisamos trocar o nosso container, que antes era o espaço branco, e trocar a cor da fonte e a cor do botão. Já pode copiar o código CSS da cor do background do container e vamos voltar para o CodePen.

Agora temos que lembrar que cada elemento será estilizado de uma forma específica. Podemos colocar o .container da classe dark, então .dark .container. Vamos colocar o código da cor que você copiou do background. Além disso, também trocaremos a cor da fonte, vamos pegar lá no Figma qual é a cor da fonte no modo dark.

```
.dark {
  background: linear-gradient(236.85deg, #375b86 27.26%, #6b87a9
96.03%);
}

.dark .container {
  background: #333439;
  color: #f6f6f6;
}
```

Agora ficou legal. A fonte do texto de "Projetos" também ficou branca no modo dark, mas ficou estranho. O pessoal que fez o layout no Figma também percebeu que era melhor deixar essa fonte dos projetos e os nomes dos projetos em uma fonte escura.

Vamos ver no HTML qual era a tag dessa parte de projetos. Podemos pegar essa classe="projetos", porque queremos trocar a cor de todos os elementos. Então vamos escrever no CSS, .dark .projetos e colocar color escura igual a que está no Figma.

```
.dark .container {
  background: #333439;
```

```
  color: #f6f6f6;
}

.dark .projetos {
  color: #1c1c1c;
}
```

10.9 DARK MODE NO BOTÃO

Vamos salvar e rodar. Agora não está mais mudando a cor da fonte dos projetos. Outra coisa que devemos fazer é alterar o botão, porque no dark mode ele também fica escuro.

Podemos copiar o código da cor do botão no Figma.

Vamos criar o `.dark .tema button`, e dentro dele a cor do background. Além disso, temos que mudar a cor da fonte do botão. Além disso, podemos mudar também a borda do botão com o border de 2px, que é o tamanho da borda; solid, para ser uma borda sólida e a cor pode ser um pouco mais clara.

```
.dark .tema button {
  background: #1c1c1c;
  color: #ffffff;
  border: 2px solid #f7f7f7;
}
```

Vamos salvar e rodar. Clica em "Alterar Tema".

COLOCANDO O PORTFÓLIO NO AR

Ficou bem legal. Vamos colocar esse projeto no ar para que qualquer pessoa, com o link que vamos passar, possa visualizar esse layout.

Quando temos um projeto web, que é o caso do projeto que estamos desenvolvendo. Aqui está dentro do CodePen, que é uma ferramenta web e você só vai ver no CodePen. Às vezes queremos colocar isso no nosso site ou tem outros sistemas que permitem que você faça isso.

Então, quando queremos hospedar, existem vários serviços de cloud (na nuvem), tem ferramenta da Amazon, da Oracle, do Azure, do Google, tem esses grandes. Mas tem serviços menores de host de website. Tem algumas ferramentas para devs, como o GitHub, que até dá um espaço gratuito para fazermos isso. Não foi à toa que mostramos o GitHub na aula passada e estamos te provocando para usar o tal do GitHub porque, querendo ou não, você vai acabar usando GitHub na sua vida.

Agora o ideal é pegar esse site que fizemos e colocar no ar. Já está no ar no CodePen, mas vamos fazer de um jeito um pouco mais tradicional, mais profissional, vamos chamar assim.

E o que precisamos para conseguir vincular esse projeto que fizemos no CodePen lá no LinkedIn e ter um link para mostrar para várias outras pessoas?

10.10 CRIANDO UMA CONTA NO GITHUB

A primeira coisa a fazer é acessar o GitHub. Eu já estou logado. Mas se você não estiver, no canto superior direito vai ter um botão Sign up, clica nele e faz um cadastro para logar no GitHub.

10.11 CRIANDO UM REPOSITÓRIO

Para conseguirmos pegar o código no CodePen e passar para o GitHub, vamos criar uma pasta, também chamada de **repositório**, para manter todos os códigos do CodePen. Para criar essa pasta, vou clicar no ícone de mais, no canto superior direito, e clicar em "New Repository". Vamos ter que escolher um nome para esse repositório. Qual vai ser o nome?

Pode ser "certificard", é um bom nome para o nosso certificado.

Lembrando que não podemos ter dois repositórios com o mesmo nome, o GitHub faz essa verificação. Agora, não vou me preocupar com os outros campos, a descrição do projeto, se ele é público ou privado, readme, gitgnore.

Em seguida, aparecerá uma página com algumas linhas de código. É possível enviar os arquivos do CodePen para o GitHub só com linhas de código no terminal. Não é o que vamos fazer. Nós vamos fazer upload de arquivos existentes, clicando em "uploading an existing file". Vai aparecer um quadro para passarmos nossos códigos para o GitHub.

10.12 DOWNLOAD DO PROJETO DO CODEPEN

Vamos precisar fazer download do nosso código no CodePen. No canto inferior direito do CodePen existe um botão `"Export"`, ao clicar nele temos as opções de salvar como Gist do GitHub, não é o que queremos, e tem o "Export.zip" que é a opção que escolheremos. Quando eu clicar em `"Export.zip"`, ele vai preparar o ZIP e avisar que já podemos fazer o download dos

arquivos.

Ao abrir esse arquivo que baixamos, "certificardaula-10.zip". Ele vai extrair. Se você estiver usando Windows, pode usar o WinRar ou algo do tipo para extrair os arquivos do ZIP. Dentro dele tem uma série de arquivos, tem o "license.txt"; "README.markdown"; uma pasta "src" com index, script e style; e a pasta "dist", que é a pasta que vamos usar.

Vamos entrar na pasta "dist". Quando abrimos o arquivo `index.html` do **dist**, na linha 6, temos o seguinte código:

```
<link rel="stylesheet" href="./style.css">
```

Ou seja, ele já tem um link que "conversa" com o arquivo css. Se abrimos o source direto, ele não tem. Vamos abrir para verificarmos, e também ao link, como o VSCode. Esse link traz exatamente o código que estávamos vendo anteriormente, não é o que queremos. **Nós queremos o código do dist já com CSS e JavaScript funcionando**.

Então, vamos selecionar, não do src, mas do dist, os três arquivos e arrastar para o VSCode. Selecionaremos os três links do dist e arrastaremos para dentro do VSCode. Assim, ele pegará os três arquivos e mandará para o GitHub. Mais abaixo, há uma palavra que nos depararemos sempre como Devs que é `Commit` .

10.13 COMMIT

O Commit é a linha do tempo. No "Commit changes" eu "adicionei os arquivos do certificard". Isso nos permite perceber as alterações que estamos realizando no GitHub, temos o commit em uma foto - que nós colocamos - dos arquivos que fizemos e

precisamos alterar um script, o index.html, por isso faremos um novo commit, para ficar registrado.

Ao fazer o commit ele processa os arquivos e conseguiremos visualizar. Quando selecionamos o index.html, aparece o certificard com todo o código que desenvolvemos. O mesmo para o script.js e para o style.css. Algo interessante do GitHub é que ele mostra porcentagem de HTML que há no repositório de JavaScript.

10.14 GITHUB PAGES

Para que seja possível ter, de fato, um link que abre o nosso projeto, acessaremos "Settings" ou configurações. Se "scrollarmos" a página, encontraremos uma série informações, sendo, uma delas, a "GitHub Pages" que nós apertaremos. Na próxima página, selecionaremos o local onde desejamos inicializar o nosso projeto.

Porém, antes disso, precisamos saber que, no "certificard", implicitamente está dito: tudo que você está fazendo aqui, está na "Branch > main". Quando criarmos e ativarmos o GitHub Pages, precisamos indicar: o meu código principal está na "main". Portanto, retornando agora ao GitHub Pages e falamos: o GitHub Pages está desabilitado. Qual é a origem das páginas do projeto?

No primeiro campo, vamos selecionar "main" e salvar. Logo, ele apresentará uma mensagem avisando que o nosso site está pronto para ser publicado, "certificard". Vamos selecionar o link com o botão direito, acessá-lo e receberemos uma mensagem de erro, "404". Retornando ao "GitHub Pages" repare que estamos usando uma palavra chamada root. Outros frameworks

compreendem melhor o que é esse arquivo root.

O root conseguirá identificar por onde o projeto começa. No nosso caso, como só temos três, note que quando selecionamos o root, ele mostra: o root, que é para frameworks mais complexos. Nós escolheremos esta segunda opção, `docs` , apertaremos o botão "Save", ele apresenta "certificard". Agora, atualizaremos a nossa página e está tudo no ar.

Para isso acontecer, nós passamos por dois pontos importantes: A questão da "Branch", que é sobre o local em que estamos fazendo, e nós indicamos que é no "main", no principal; e **não utilizamos o root**, isto é, não estamos usando um framework JavaScript (em que uma parte do root aponta para a página índice). Optamos pelo docs e, a partir desse momento, já temos o nosso "certificard" funcionando corretamente.

CONSIDERAÇÕES FINAIS

Queria dar as boas-vindas na carreira de Dev e agradecer o mergulho na Imersão. Por mais uma semana, estaremos no Discord conversando com vocês, tirando dúvidas, explicando sobre a carreira, realizando encontros ao vivo, streamings, enfim, são muitas atividades. Sem contar que você já está nos seguindo no canal do YouTube da Alura e no Podcast do Hipsters e no Instagram do Guilherme.

Parabéns a você!! Agradeço a todas as pessoas que participaram. São muitas! Na edição, no time do Scuba, pessoas desenvolvendo o roteiro, fazendo imagens, editando, gravando, enfim, são muitas pessoas envolvidas mesmo! Não é a toa que a

Imersão alcançou esse tamanho. Última coisa, você pensou que a Imersão era de graça? Não, você precisa dar algo em troca!

Estou pedindo para que você faça uma avaliação do que você aprendeu, faça um vídeo ou um texto explicando algum dos projetos, o que você mais gostou, a parte de JavaScript ou de CSS e que você reproduza o conhecimento que adquiriu e mostre o seu portfolio, conte onde você quer chegar.

Falar e mostrar o seu código, facilitará o seu aprendizado, reforçará seu conhecimento e, quem sabe, até ajudará a expor seu portfolio para as empresas, para que você comece a encontrar as vagas de emprego. Sem promessa nenhuma, mas esse é um formato que também nos visibiliza. Você falará da Alura, da Imersão Dev e trará outras pessoas para que, no futuro, nas próximas edições da Imersão Dev, elas façam esse mesmo caminho que você trilhou.

Muito obrigado!! Espero você nessa carreira, nessa comunidade e no mercado de tecnologia.

Rafaella: Obrigada, pessoal!

Guilherme: Que a força esteja com você!!!