Guia

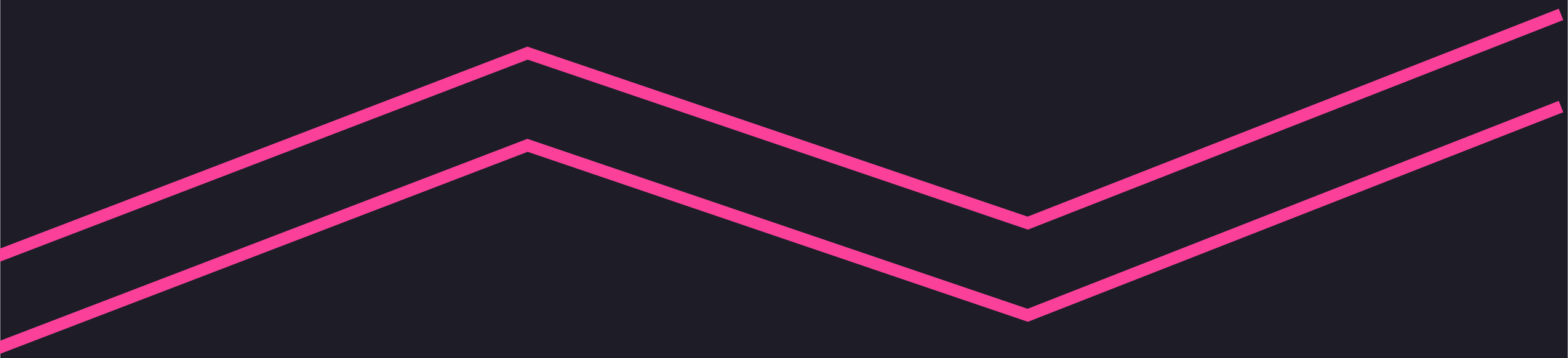
DRIVEN

JavaScript do Zero

Índice

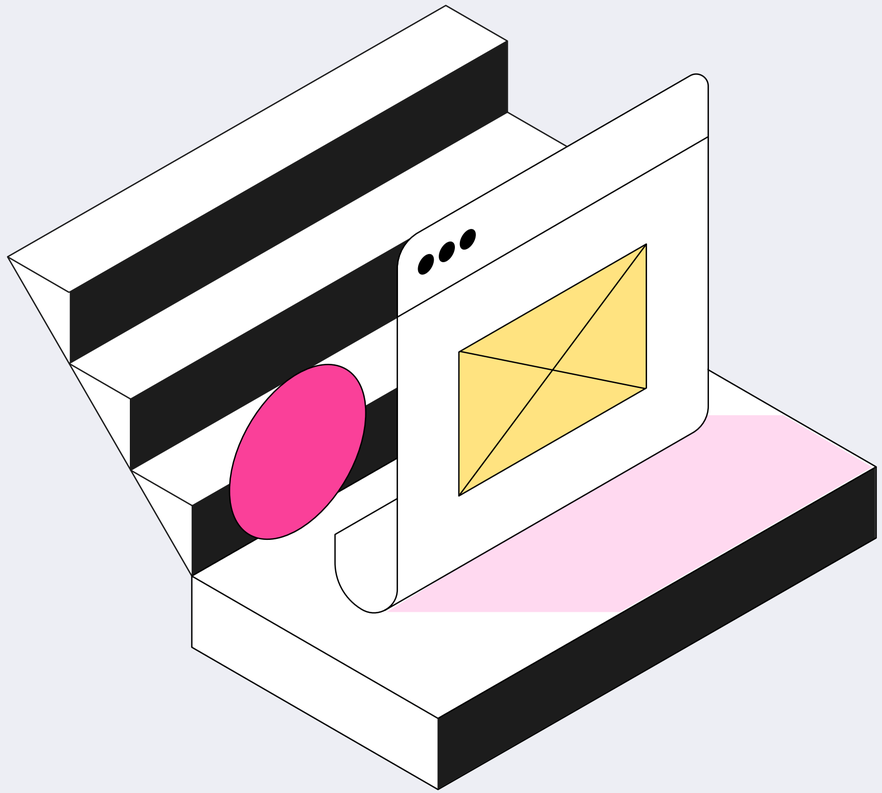
Introdução ao JavaScript 3
Meu 1º JavaScript 8
Calculando com JavaScript 13
Manipulando texto 18
Condicionais 22
Operadores Lógicos 31
Loops 38
Arrays 45

Capítulo 1



Introdução ao JavaScript

Quando construímos um site ou software, geralmente temos dois tipos de trabalho: construir a **interface*** da aplicação (o visual) e o seu **comportamento**.



Para fazer nossa aplicação se comportar como queremos, precisamos aprender a dar instruções para o computador criar **sequências de passos** informando exatamente o que queremos que aconteça em cada funcionalidade do nosso programa. O nome que damos a essa sequência de passos é **algoritmo**.

Exemplo:

Imagina que você queira fazer um programa que ajuda seu professor a calcular a média de um aluno. Supondo que a média é dada por 3 notas com os pesos 3, 3 e 4, você terá uma sequência de passos mais ou menos assim:

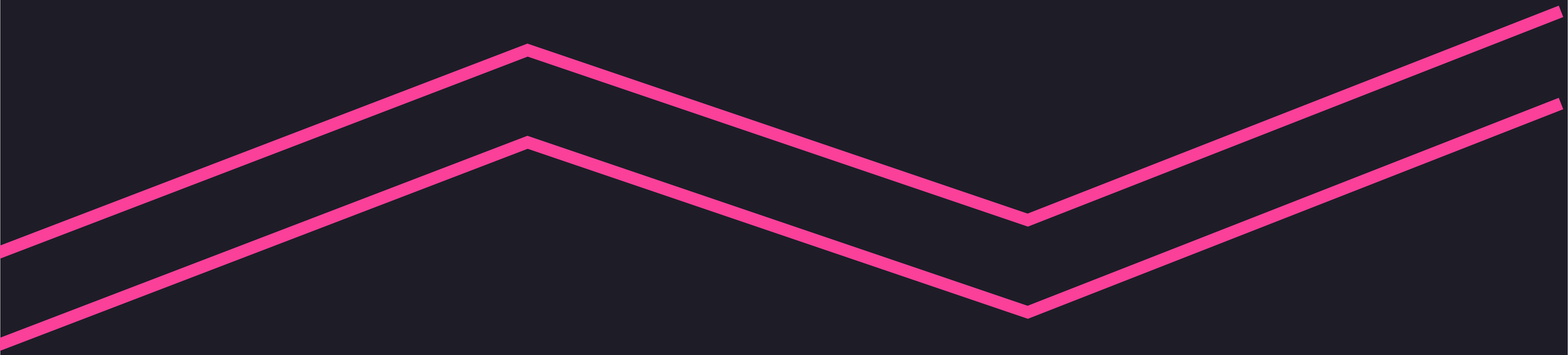
```
Algoritmo CalculadoraDeMedia (recebe nota1,
nota2 e nota3)

   - multiplica nota1 por peso 3
   - multiplica nota2 por peso 3
   - multiplica nota3 por peso 4
   - a média é a soma de todas as notas em
seguida dividida por 10
   - responde essa média
```

Depois de criar essa lógica, bastaria agora executar esse algoritmo enviando para ele as notas do aluno:

```
 Executar Algoritmo CalculadoraDeMedia (recebe
notas 5, 5 e 10) => responderá 7
```

Para criar esses algoritmos, usaremos uma das linguagens mais utilizadas atualmente no mundo, o **JavaScript**.

Capítulo 2



Meu 1º JavaScript

Nosso próximo passo é aprender a **criar algoritmos** utilizando JavaScript. Para isso, primeiro precisamos saber onde o escrevemos e executamos. Existem várias ferramentas que nos permitem escrever JavaScript, uma delas é o **REPL**, um editor online muito prático.

Para simplificar, antes de fazer nossa calculadora de médias, vamos implementar um algoritmo simples que soma apenas dois números:

```
Algoritmo Somar (recebe num1 e num2)
 - soma num1 com num2
 - retorna essa soma
```

Para representar o algoritmo acima em JavaScript, podemos escrever:

```
function somar(num1, num2) {
  let soma = num1 + num2;
  return soma;
}
```

Observe alguns conceitos no algoritmo anterior:

• `function somar(...) { ... }` é a estrutura básica de criação de funções em JavaScript. Uma função é a forma mais simples de representarmos um algoritmo. Dentro dela é que vamos colocar nossa sequência de passos;

• Dentro dos parênteses `(num1, num2)` especificamos os **parâmetros** do nosso algoritmo, que são as entradas, os dados que ele precisa pra funcionar;

• A operação `num1 + num2` é o que está de fato fazendo a soma. Repare que **somar em JS** é bem simples, é só usar o sinal de + mesmo;

• Em seguida, em `let soma = ...;` estamos colocando o resultado dessa soma em uma **variável** chamada `soma`. Variáveis são caixinhas em que podemos colocar valores dentro, nesse caso estamos guardando na caixinha `soma` o resultado da operação `num1 + num2`;

• Por fim, em return soma; estamos retornando pra quem tiver chamado essa nossa função o resultado final da operação, que estava guardado na variável soma;

• Repare também que ao final de cada passo do meu algoritmo, tem um **ponto-e-vírgula**. Ele diz pro JS que terminamos aquele passo.

Agora que criamos o algoritmo, o próximo passo é executá-lo. Queremos portanto fazer em JavaScript algo como:

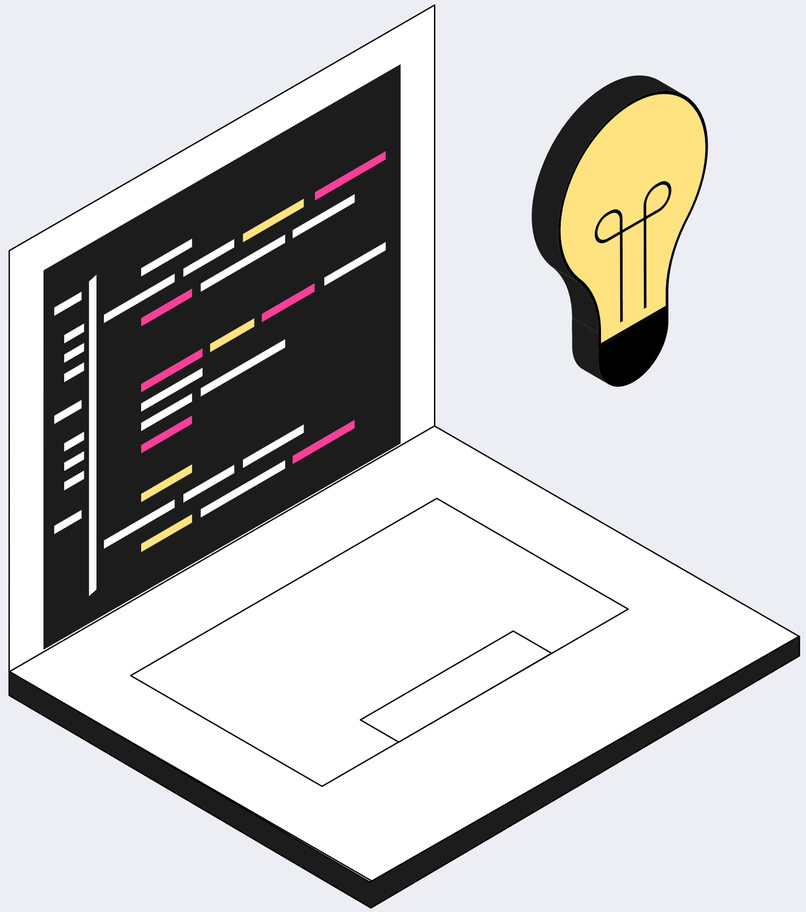
```
Execute o algoritmo somar com os valores 2 e 3
```

Pra fazer isso em JavaScript, basta:

```
somar(2, 3);
```

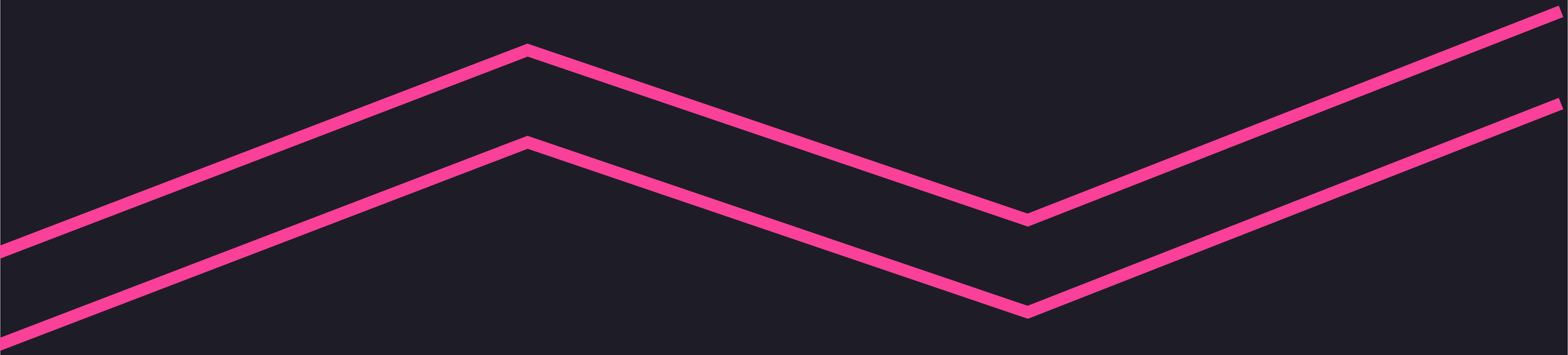
Exercício:

Abaixo estão alguns exercícios para você praticar o que aprendemos até aqui.



1) Crie uma função com nome. A função deve estar vazia, sem nenhum parâmetro nem código dentro.

Capítulo 3



Calculando com JavaScript

Agora que já sabemos criar nossas funções em JavaScript, vamos explorar algumas outras operações matemáticas além da soma.

Primeiramente, realizamos as 4 operações básicas da matemática dessa forma:

```
let soma = 10 + 2;              // Resultado: 12
let subtracao = 10 - 2;         // Resultado: 8
let multiplicacao = 10 * 2;     // Resultado: 20
let divisao = 10 / 2;           // Resultado: 5
```

Repare que utilizamos o `// ...` para inserir comentários em JavaScript. Comentários são trechos de código que não serão executados e servem só para adicionar alguma anotação útil ao código

Você pode fazer várias operações de uma vez, por exemplo:

```
let media = (10 + 2) / 2;       // Resultado: 6
```

Observe que utilizamos parênteses para mudar a precedência de operações (a soma acontecer antes da divisão). Precisamos disso, porque o JavaScript respeita a ordem de precedência da matemática. Por padrão, multiplicação e divisão acontecem antes de soma e subtração.

```
let media = 10 + 2 / 2; // Sem os parênteses daria 11
```

Para representar números decimais em JavaScript, utilizamos o separador ponto:

```
let decimal = 2.5;
```

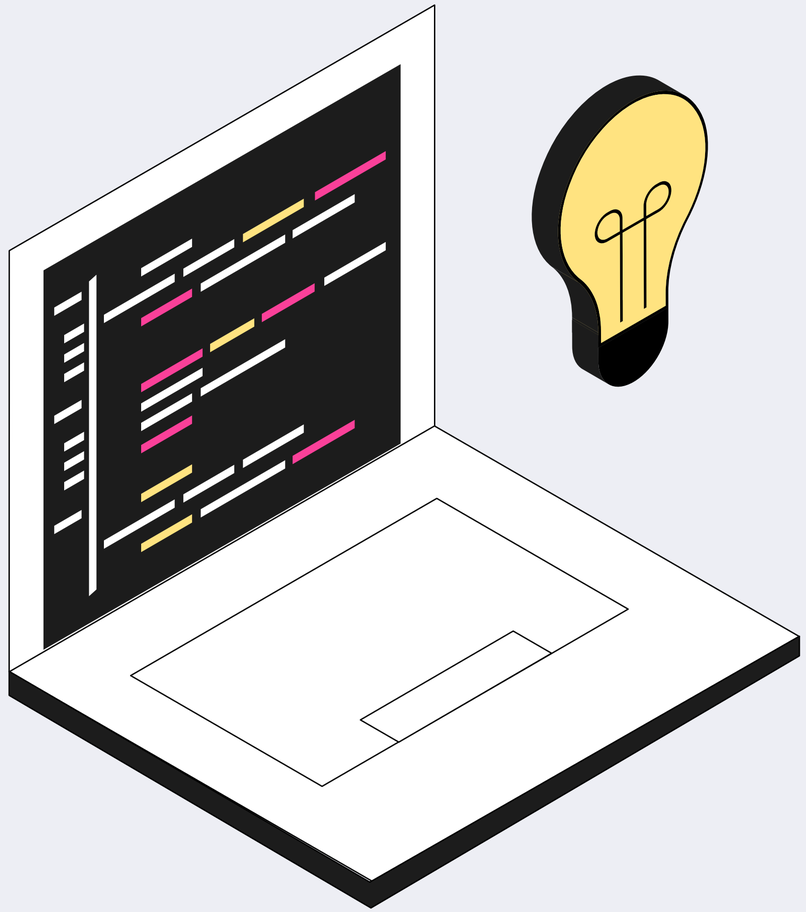
Uma divisão em JavaScript pode resultar em valores decimais, por exemplo:

```
let resto = 10 % 3;          // Resultado: 1, pois o resto de 10 dividido por 3 é 1.
```

Dica: isso é muito útil pra saber se um número é par ou ímpar. Basta olhar o resto da divisão dele por 2. Se for 0, é par. Se for 1 é ímpar.

Exercício:

Abaixo estão alguns exercícios para você praticar o que aprendemos até aqui.



1) Implemente a função, que calcula a média simples entre 2 números.

2) Implemente a função, que retorna o resto da divisão entre 2 números.

Capítulo 4

Manipulando texto

Até agora manipulamos números com JavaScript, mas muitas vezes precisamos também manipular textos, que na computação chamamos de **strings**.

Primeiramente, para criar um texto em JavaScript, podemos fazer:

```
let texto = "Esse é meu primeiro texto";
return texto;
```

Repare que o texto obrigatoriamente deve ser envolvido por aspas.

Para juntar duas strings podemos utilizar o operador de soma +:

```
let introducao = “Meu nome é”;
let nome = “Pedro”;
let mensagem = introducao + nome;   //
Resultado: “Meu nome é Pedro”;
```

Essa operação é chamada de **concatenação** (juntar strings).

Podemos juntar strings com números, da mesma forma:

```
let media = 7;
let mensagem = “Sua média foi:” + media;   //
Resultado: “Sua média foi: 7”
```

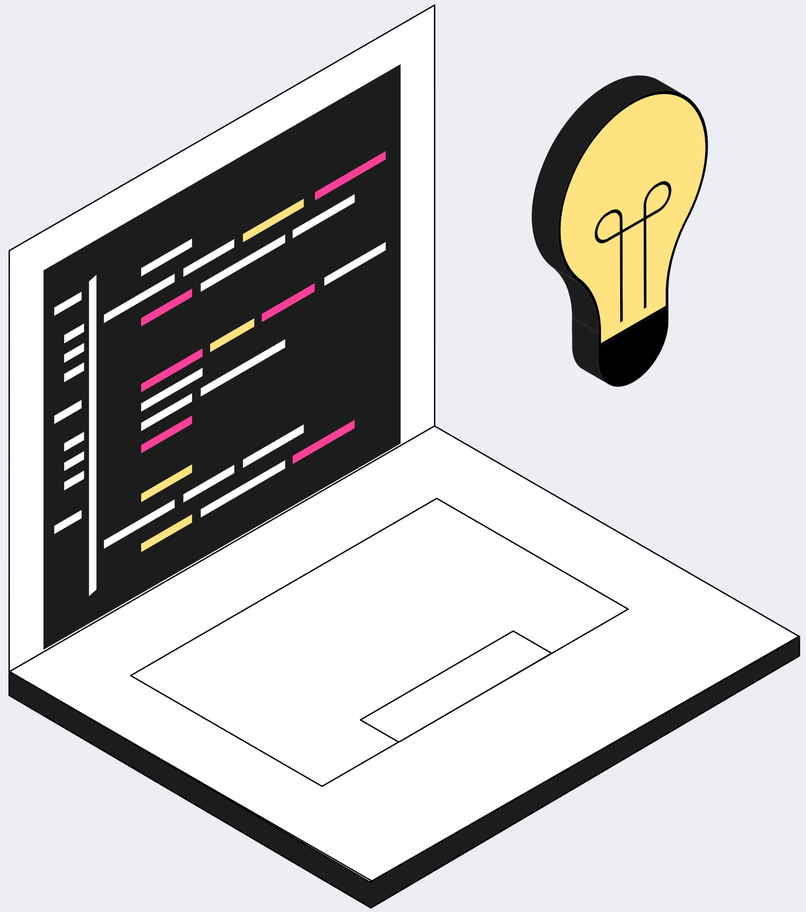
Exercício:

Abaixo estão alguns exercícios para você praticar o que aprendemos até aqui.

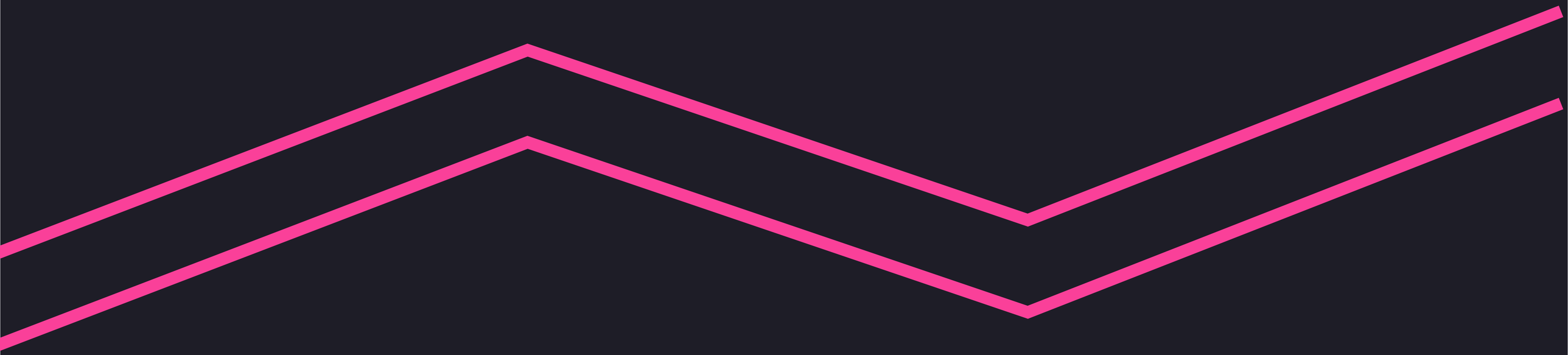
1) Implemente a função, que retorna a string “Hello World”.

2) Implemente a função, que retorna a concatenação dos dois parâmetros que ela recebe.

3) Implemente a função, que soma dois números e retorna o texto A soma deu: x, sendo x o resultado da soma.



Capítulo 5



Condicionais

Quando precisamos desenvolver uma aplicação, muitas vezes o comportamento das funcionalidades muda dependendo de alguma condição.

Por exemplo: se um site é proibido para menores de idade, devemos ou não permitir o acesso da pessoa, conforme sua idade.

O algoritmo que queremos então é algo como:

```
se alguma_condicao
 fazer tal coisa
senão
 fazer outra coisa
```

Para implementar esse algoritmo em JavaScript é bem parecido, basta usar a sintaxe do `if/else`. Por exemplo, se quisermos responder se alguém é maior ou menor de idade, poderíamos fazer:

```
function verificar(idade) {
  if(idade < 18) {
    return “É menor de idade”;
  } else {
    return “É maior de idade”;
  }
}
```

O `if` (“se” em inglês) serve para, dada alguma condição, executar ou não o código dentro dele:

```
if(...) {
  // esse código só será executado caso a condição seja VERDADEIRA
}
```

Já o `else` (“senão em inglês”), pode ser usado logo após o `if`, para executar algum outro código caso a condição dele não seja verdadeira.

```
if(...) {
  ...
} else {
  // esse código só será executado caso a condição do if seja FALSA
}
```

Nem todo `if` precisa ter um `else`. Se você não quiser executar nada no caso contrário, basta utilizar somente o if, sem o `else { ... }`.

Se quiser checar várias condições de uma vez, você pode encadear mais ifs dessa forma:

```
function verificarMedia(media) {
  if(media >= 7) {
    return "Aprovado";
  } else if(media >= 5) {
    return "Prova final";
  } else {
    return "Reprovado";
  }
}
```

Repare que a construção do meio emenda um `else` e um `if`. Isso significa que ele só vai verificar a condição do meio (maior ou igual a 5) se a primeira tiver sido falsa (a nota não foi maior ou igual a 7).

Falando agora das condições que você pode verificar dentro de um `if`, as mais comuns são:

```
media > 7          // Maior que 7
media >= 7         // Maior ou igual a 7
media < 7          // Menor que 7
media <= 7         // Menor ou igual a 7
media == 7         // Igual a 7 (ATENÇÃO, SÃO DOIS
IGUAIS!)
media != 7         // Diferente de 7
```

Quando fazemos comparações em geral, isso nos dá um valor que pode ser **verdadeiro** ou **falso**. Por exemplo, se a nota for 8, a comparação `nota > 7` nos dará verdadeiro e o código de dentro do if será executado**.** Porém se a nota for 5, a comparação `nota > 7` dará **falso** e o código do if não será executado.

Como na computação precisamos o tempo todo fazer comparações e tomar decisões, esses valores de verdadeiro ou falso tem até um nome especial: são chamados de **booleanos**. E você pode representá-los em JavaScript usando as palavras `true` (verdadeiro) ou `false` (falso).

Por exemplo, se passarmos `true` para o if, ele sempre executará (ou se passar `false`, nunca executará):

```
if(true) {
   // sempre executará, pois true é sempre
verdadeiro
}

if(false) {
   // nunca executará, pois false é sempre falso
}
```

Você também pode armazenar esses valores em variáveis, por exemplo:

```
let minhaVariavel = true;

if(minhaVariavel) { ... }
```

Atenção: não faz muito sentido armazenar um booleano dessa forma, afinal se você já sabe que ele vai ser `true`, não tem nem porque colocar um if.

Mas passa fazer sentido quando armazenamos o resultado de uma comparação em uma variável, por exemplo:

```
function verificarAprovacao(nota1, nota2, nota3)
{
   let media = (nota1 + nota2 + nota3) / 3;

   let passou = (media > 7);    // Vai armazenar
true ou false na variável "passou", dependendo
do resultado da comparação
   if(passou) {
     return "Aprovado";
   } else {
      return "Reprovado";
   }
}
```

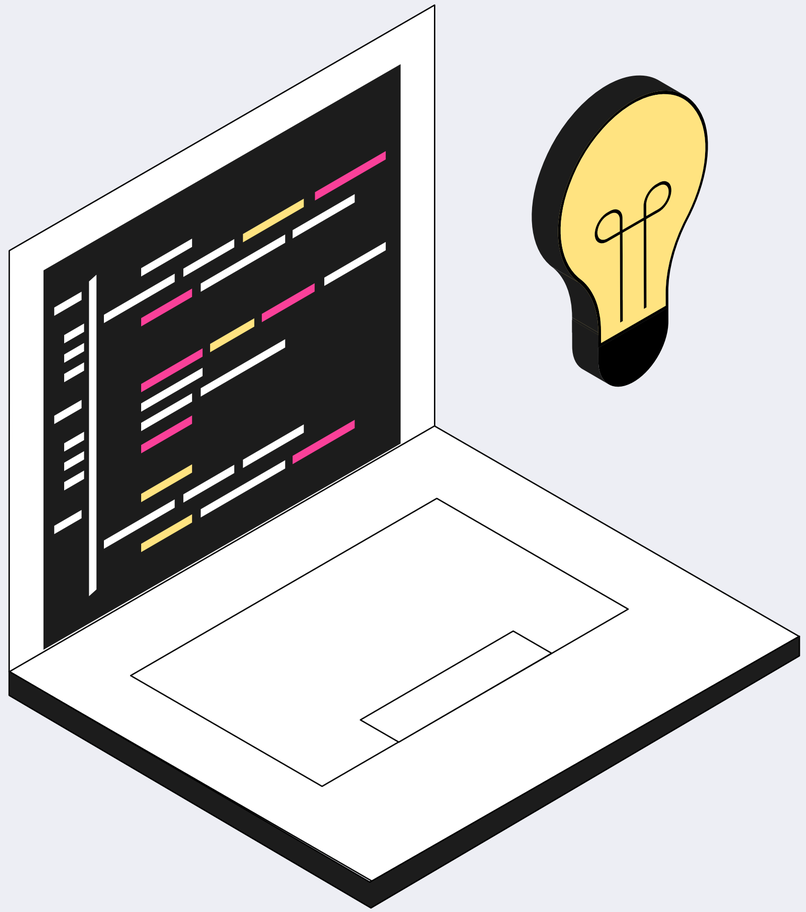
Exercício:

Abaixo estão alguns exercícios para você praticar o que aprendemos até aqui.

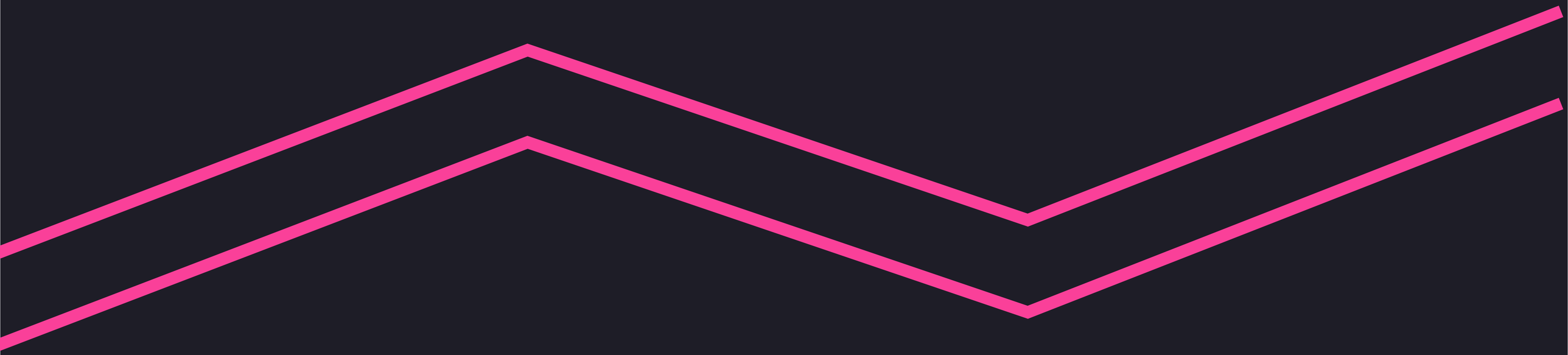
1) Implemente a função, que retorna “Maior de idade caso a idade passada seja maior que 17 ou “Menor de idade caso contrário.

2) Implemente a função, que retorna “Aprovado” caso a nota passada seja, “Prova Final caso seja maior ou igual a 5 porém menor que 7 ou “Reprovado” caso contrário.

3) Implemente a função, que recebe 3 notas e retorna os booleanos ou, indicando se a média simples entre as 3 notas (somar e dividir por 3) é maior ou igual a 7.



Capítulo 6



Operadores Lógicos

Vimos no último capítulo que dentro do `if` podemos fazer **comparações** para o código ser executado. Por exemplo, uma comparação pode ser `nota >= 7`.

Quando trabalhamos com condicionais e booleanos, uma situação muito comum é querer levar mais de uma comparação em consideração ao mesmo tempo. Por exemplo, se a regra para alguém ser aprovado for: **ficar com pelo menos 7 de média e ter menos de 10 faltas**. Teríamos um algoritmo mais ou menos assim:

```
Algoritmo aprovacao(recebe nota1, nota2, nota3 e
faltas)
  - calcular media usando as notas
  - se a media for pelo menos 7 E TAMBÉM as
faltas forem menores que 10
       - responder aprovado
  - senao (caso qualquer uma das comparações
acima forem falsas)
      - responder reprovado
```

Fazer cada uma das comparação já sabemos. A primeira é media `>= 7` e a segunda é `faltas < 10`. Mas como fazer aquele “e também” ali do meio? É simples, podemos usar o operador do JavaScript: `&&` (dois “e-comerciais” juntos), chamamos esse operador de AND.

```
function aprovacao(nota1, nota2, nota3, faltas)
{
   let media = (nota1 + nota2 + nota3) / 3;

   if(media >= 7 && faltas < 10) {
      return “Aprovado”;
   } else {
      return “Reprovado”;
   }
}
```

Podemos usar a técnica de armazenar o resultado das comparações em variáveis e deixar nosso código mais legível:

```
function aprovacao(nota1, nota2, nota3, faltas)
{
  let media = (nota1 + nota2 + nota3) / 3;

  let aprovadoPorMedia = (media >= 7);        //
Vai armazenar true ou false
  let aprovadoPorPresenca = (faltas < 10);    //
Vai armazenar true ou false

  if(aprovadoPorMedia && aprovadoPorPresenca) {
// Um pouco mais fácil de ler a regra :)
    return "Aprovado";
  } else {
    return "Reprovado";
  }
}
```

Repare que o operador `&&` nos dá verdadeiro somente se ambas as condições forem verdadeiras. Se uma delas ou ambas forem falsas, ele dará falso.

Existe também um outro operador para um caso muito comum: quando você está interessado(a) em que **pelo menos uma condição seja verdadeira**.

Por exemplo, imagina que o aluno pode ser classificado pra uma faculdade em duas situações: caso tire pelo menos 8 no vestibular da faculdade ou pelo menos de 700 no ENEM:

```
function classificado(notaVestibular, notaEnem) {

  let classificadoVestibular = (notaVestibular >=
8);
  let classificadoEnem = (notaEnem >= 700);

  if(classificadoVestibular || classificadoEnem) {
    return "Classificado";
  } else {
    return "Desclassificado";
  }
}
```

Esse operador `||` (duas barras verticais coladas), chamado de OR, retorna verdadeiro quando pelo menos 1 das comparações forem verdadeiras. Ele só dará falso se ambas derem falso, ou seja, se o aluno não passou nem pelo vestibular nem pelo ENEM.

Por fim, um último operador booleano muito útil é quando precisamos inverter um booleano, ou seja, transformar `true` em `false` ou vice-versa. O operador que faz isso em JavaScript é a exclamação `!`, chamado de NOT.

```
function aprovado(nota1, nota2, faltas) {
  let media = (nota1 + nota2) / 2;

  let aprovadoPresenca = (faltas < 10);

  if(!aprovadoPresenca) {                    //
Leia como: "se NÃO aprovado por presença". Ou
seja, se aprovadoPresenca for false, vira true e
entra no if e vice-versa
    return "Reprovado por presença";
  }

  ...
}
```

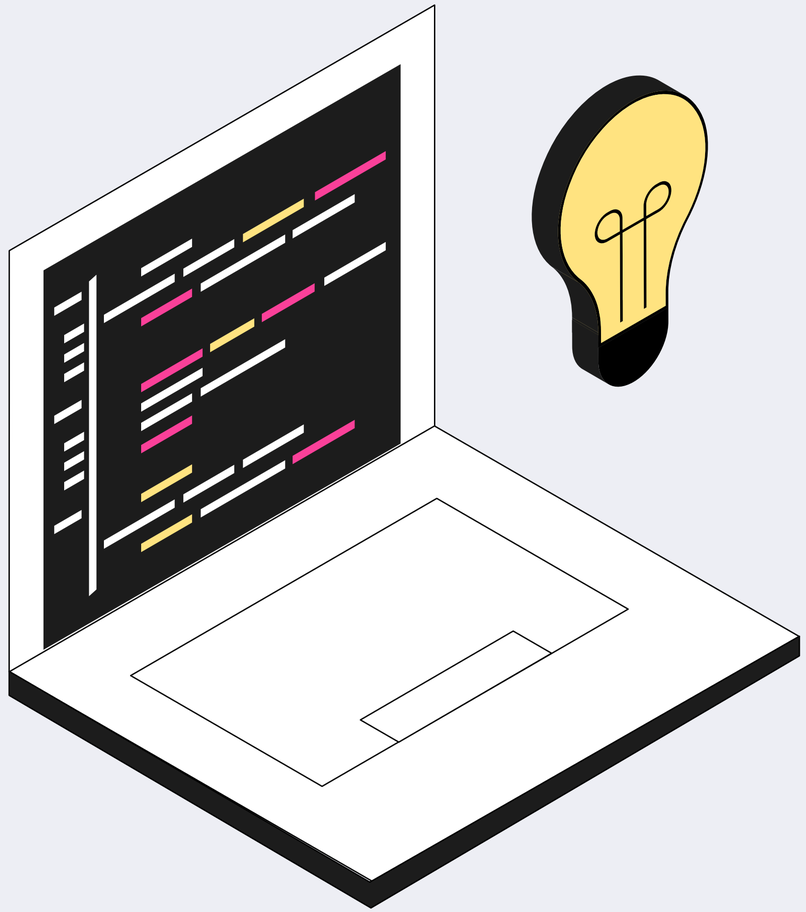
Exercício:

Abaixo estão alguns exercícios para você praticar o que aprendemos até aqui.

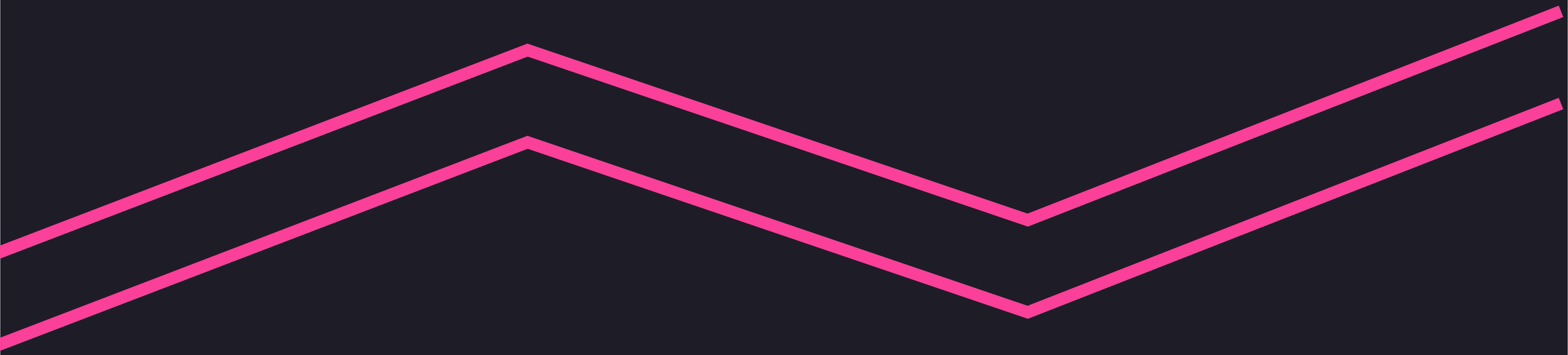
1) Implemente a função, que recebe um preço e um booleano indicando se já está com desconto ou não. Se o preço for maior que 100 e não estiver com desconto, a função deve retornar “Quero pechinchar”. Caso contrário, deve retornar “Negócio fechado”.

2) Implemente a função, que recebe uma nota e um número de faltas e retorna “Aprovado” caso a média seja maior ou igual a 7 e o número de faltas menor que 10, ou “Reprovado” caso contrário.

3) Implemente a função, que recebe 3 números e retorna Têm negativo caso pelo menos 1 deles seja menor que 0. Caso contrário, ela deve retornar Tudo positivo.



Capítulo 7



Loops

Quando implementamos algoritmos, em geral queremos economizar nosso tempo de trabalho manual. E isso é particularmente útil quando alguma coisa precisa ser feita várias vezes. Por exemplo, digamos que você queira repetir alguma ação 100 vezes (calcular a média de 100 alunos de uma vez).

Quando trazemos isso para programação, podemos usar um recurso presente na maioria das linguagens chamado loop. Um loop é uma forma de você dizer pro computador: enquanto uma determinada condição for verdadeira, repita esse código de novo e de novo e de novo.

Assim como em outras linguagens, em JavaScript é possível criar loops de várias formas diferentes. Veremos duas delas aqui. A primeira é usando o recurso `while`:

```
while(alguma_condicao) {
  // esse código será repetido sem parar, até
que a condição acima dê falso
}
```

Atenção: enquanto a condição for verdadeira, o código ficará repetindo. Isso significa que se a condição for sempre verdadeira e nunca mudar, você terá criado um loop infinito e irá travar seu programa.

```
while(true) {
   // isso é um loop infinito, não faça isso hahah
}
```

Ou seja, se quisermos controlar a quantidade de vezes que o loop acontece, precisamos que a condição de alguma forma passe a dar `false` depois da quantidade de vezes que nos interessa. Um jeito de fazer isso é usar alguma variável para armazenar quantas vezes o loop rodou. Por exemplo:

```
let contador = 0;

while(contador < 100) {
   // algum código que quero repetir 100 vezes
   contador = contador + 1;
}
```

Repare que agora, a cada loop, somamos 1 ao contador. Assim, quando ele chegar a 100, a condição vai passar a dar falso, e o loop será encerrado.

Repare também que agora estamos alterando a variável contador. Quando queremos alterar uma variável, não deve-se colocar o `let` na frente. Ele deve ser usado somente no momento que criamos a variável.

Uma observação, sempre que você quiser aumentar um número em 1 unidade, tem um jeito mais prático que é usar o operador `++`:

```
contador = contador + 1
// é o mesmo que:
contador++;
```

Por exemplo, podemos usar loops para gerar uma string com a frase “Não devo contar mentiras” 100 vezes:

```
function umbridgeGenerator() {
  let contador = 0;
  let frase = “”;

  while(contador < 100) {
    frase = frase + “Não devo contar mentiras”;
    contador++;
  }

  return frase;  // retornará “Não devo contar
mentirasNão devo contar mentirasNão devo contar
mentiras...” (100 vezes)
}
```

Essa construção do `while` (de criar uma variável, uma condição e um incremento) é tão comum, que existe uma outra forma de criar um loop usando menos linhas. Observe os trechos ao lado:

```
let contador = 0;

while(contador < 100) {
  // ...
  contador++;
}
```

Em vez de `while`, você pode fazer a mesma coisa acima usando a segunda forma de se criar loops em JavaScript: a construção `for`. Dessa forma:

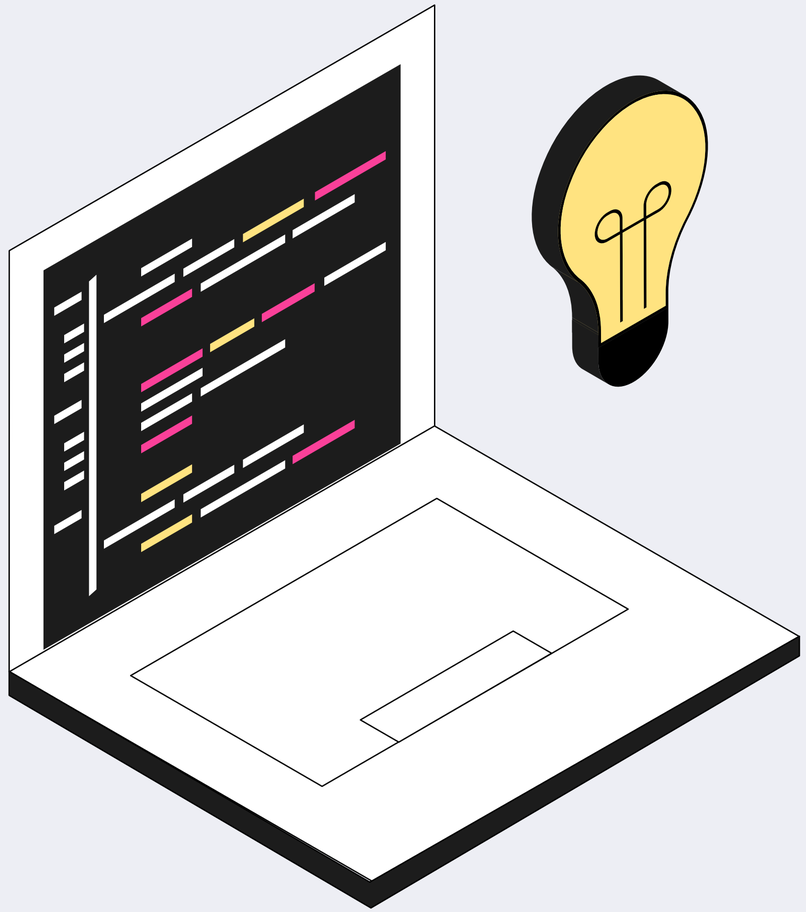
```
for(let contador = 0; contador < 100;
contador++) {
  // esse código será executado 100 vezes
}
```

Exercício:

Abaixo estão alguns exercícios para você praticar o que aprendemos até aqui.

1) Utilizando loops, implemente a função que recebe um texto e um número de repetições. Ela deve retornar uma string com o texto repetido n vezes, sendo n o número de repetições.

2) Utilizando loops, implemente a função que recebe um texto e um número de repetições. Ela deve retornar uma string com o texto repetido n vezes, sendo n o número de repetições.



Capítulo 8

Arrays

Até então, temos trabalhado com variáveis que guardam 1 valor cada uma, por exemplo:

```
let nota1 = 5;
let nota2 = 7;
let nota3 = 10;
```

Porém, um caso muito comum na computação é quando precisamos armazenar **listas** de valores. Por exemplo, se quisermos guardar os nomes de todos os alunos, teríamos que fazer algo como:

```
let aluno1 = "Ana";
let aluno2 = "Bia";
let aluno3 = "Carlos";
...
```

Isso fica pouco prático e dificulta manipularmos essas informações. Para resolver isso, em muitas linguagens de programação existe o conceito de listas de valores, que chamamos de arrays. Em JavaScript podemos criar **arrays** usando colchetes `[]`:

```
let alunos = [“Ana”, “Bia”, “Carlos”];
```

O mais interessante dessa construção é que podemos acessar os valores de um array usando a posição deles (começando em 0). No exemplo acima, Ana está na posição `0`, Bia na posição `1` e Carlos na posição `2`. Por exemplo, podemos retornar a Bia fazendo:

```
   // posições  0       1       2
let alunos = [“Ana”, “Bia”, “Carlos”];

return alunos[1]; // Retornará “Bia”
```

Podemos também obter o tamanho de uma array, usando a propriedade `.length`:

```
let alunos = [“Ana”, “Bia”, “Carlos”];

return alunos.length; // Retornará 3 pois a array tem 3 itens
```

O mais interessante de usarmos arrays é que como podemos acessar os valores pelas posições numéricas, é um prato cheio pra usarmos loops.

Lembrando que nos loops temos um “contador” que começa como 0 e vai até o número que quisermos (até o final da array, que tal?). Então se precisamos fazer alguma coisa para cada elemento de uma array, poderíamos fazer:

```
let alunos = [“Ana”, “Bia”, “Carlos”];

for(let contador = 0; contador < alunos.length; contador++) {
   // esse código repetirá 3 vezes, o número de itens na array
}
```

Como temos essa variável contador dentro do loop, podemos usar isso para acessar as posições da array. Por exemplo, para concatenar todos os nomes:

```
let alunos = [“Ana”, “Bia”, “Carlos”];
let nomes = “”;

for(let contador = 0; contador < alunos.length;
contador++) {
  nomes = nomes + alunos[contador];
}
```

Uma forma mais comum de nomear essa variável `contador` é usar a letra `i` (de índice), ficando mais curta e legível, mas é só um nome como outro qualquer:

```
let alunos = [“Ana”, “Bia”, “Carlos”];
let nomes = “”;

for(let contador = 0; contador < alunos.length;
contador++) {
  nomes = nomes + alunos[contador];
}
```

Dando um outro exemplo, se quiséssemos fazer uma função que soma todos os valores de um array poderíamos fazer:

```
function somarTudo(lista) {
  let soma = 0;

  for(let i = 0; i < lista.length; i++) {
    soma = soma + lista[i]
  }

  return soma;
}
```

No código acima, se passássemos uma lista `[10, 20, 30]` por exemplo, ela retornaria a soma: `60`.

Exercício:

Abaixo estão alguns exercícios para você praticar o que aprendemos até aqui.

1) Implemente a função que recebe 3 nomes e retorna esses nomes em uma array.

2) Implemente função que recebe uma array e retorna a soma de todos os seus números multiplicados por 2.

3) Implemente a função que recebe uma array de números positivos diferentes entre si e retorna o índice do maior número encontrado.

Exemplo: se a array for [10,50,30], o maior número é o 50, então a função deve retornar o índice 1.

Lembre-se que os índices das arrays começam em 0.

4) Implemente a função que vai retornar qual o dia da semana vai ser a partir de um dia passado como string e de uma quantidade de dias a ser avançado. Para isso, a função deve receber uma string e um número como parâmetros e retornar uma string.

Obs: os dias devem ser retornados no seguinte formato

“Segunda-feira, Terca-feira, Quarta-feira, Quinta-feira, Sexta-feira, Sabado, Domingo”

Se for passado “Segunda-feira” e 5: a função deve retornar “Sabado”, pois avançar 5 dias a partir da segunda-feira cai no sábado.
Se for passado “Segunda-feira” e 8, a função deve retornar “Terca-feira”, pois avançar 8 dias a partir da segunda-feira cai na terça-feira da semana seguinte.

5) Implemente a função que recebe 3 números como parâmetros. Os dois primeiros delimitam um intervalo. A função deve retornar um array contendo os menores números pares dentro do intervalo. A quantidade de elementos nesse array deve ser igual ao 3o parâmetro passado

Obs:

O 1o parâmetro sempre será menor que o 2o parâmetro

No intervalo passado sempre haverá números pares suficientes para a quantidade passada

A função deve retornar os menores números pares possíveis dentro do intervalo

Exemplo: se for passado os valores “2”, “10”, “3”, a função deve retornar o array [4,6,8]

Exemplo: se for passado os valores “2”, “10”, “2”, a função deve retornar o array [4,6]