

## Noções de programação: exemplos com manipulação de imagens digitais

Prof. Guilherme Dutra Gonzaga Jaime

### Apresentação

Vamos estudar o conceito de códigos, instruções para computadores e noções básicas sobre a representação digital de imagens. Também desenvolveremos habilidades essenciais do pensamento computacional, como abstração, automação, reconhecimento de padrões, análise e avaliação, para ampliar o entendimento sobre programação.

### Propósito

## Objetivos

Módulo 1

### Manipulação de dados

Definir instruções para manipulação simples de dados.

Módulo 2

## Repetição for

Distinguir a estrutura da repetição for.

Módulo 3

## As expressões em código de computador

Reconhecer expressões usadas em códigos de computador.

Módulo 4

## A estrutura condicional if

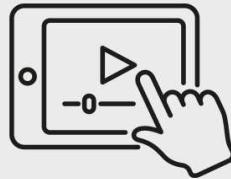
Distinguir a estrutura condicional if.



# Introdução

Apresentaremos neste vídeo os principais assuntos que serão abordados ao longo do conteúdo, com ênfase no conceito de programação dinâmica em páginas HTML e no uso da linguagem PHP para alcançar esse objetivo. Assistal!

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



## Material para download

Clique no botão abaixo para fazer o download do conteúdo completo em formato PDF.

[Download material](#)

```

//Buffer class, part 1
class Buffer {
public:
    void set(int x, int y, int z);
    void set(int x, int y, real z);
    void set(real z);
    int get(int x, int y) const;
    int get(int x, int y, int z) const;
    real get(int x, int y, real z) const;
    void plot(int x, int y, int z);
    void hline(int x1, int x2, int y, real z);
    void vline(int y1, int y2, int x, real z);
    void box(int x1, int y1, int x2, int y2, real z);
};

void Buffer::set(int x, int y, int z) {
    if (x < 0 || x > size_x - 1 || y < 0 || y > size_y - 1) {
        cout << "Error: set() - coordinates out of bounds" << endl;
        exit(1);
    }
    if (z < 0 || z > size_z - 1) {
        cout << "Error: set() - height out of bounds" << endl;
        exit(1);
    }
    int index = x * size_y * size_z + y * size_z + z;
    data[index] = z;
}

void Buffer::set(int x, int y, real z) {
    if (x < 0 || x > size_x - 1 || y < 0 || y > size_y - 1) {
        cout << "Error: set() - coordinates out of bounds" << endl;
        exit(1);
    }
    if (z < 0.0 || z > 1.0) {
        cout << "Error: set() - height out of bounds" << endl;
        exit(1);
    }
    int index = x * size_y * size_z + y * size_z + static_cast<int>(z * 255);
    data[index] = index;
}

void Buffer::set(real z) {
    if (z < 0.0 || z > 1.0) {
        cout << "Error: set() - height out of bounds" << endl;
        exit(1);
    }
    int index = static_cast<int>(z * 255);
    data[index] = index;
}

int Buffer::get(int x, int y) const {
    if (x < 0 || x > size_x - 1 || y < 0 || y > size_y - 1) {
        cout << "Error: get() - coordinates out of bounds" << endl;
        exit(1);
    }
    int index = x * size_y + y;
    return data[index];
}

int Buffer::get(int x, int y, int z) const {
    if (x < 0 || x > size_x - 1 || y < 0 || y > size_y - 1 || z < 0 || z > size_z - 1) {
        cout << "Error: get() - coordinates or height out of bounds" << endl;
        exit(1);
    }
    int index = x * size_y * size_z + y * size_z + z;
    return data[index];
}

real Buffer::get(int x, int y, real z) const {
    if (x < 0 || x > size_x - 1 || y < 0 || y > size_y - 1 || z < 0 || z > size_z - 1) {
        cout << "Error: get() - coordinates or height out of bounds" << endl;
        exit(1);
    }
    int index = x * size_y * size_z + y * size_z + static_cast<int>(z * 255);
    return data[index];
}

void Buffer::plot(int x, int y, int z) {
    if (x < 0 || x > size_x - 1 || y < 0 || y > size_y - 1 || z < 0 || z > size_z - 1) {
        cout << "Error: plot() - coordinates or height out of bounds" << endl;
        exit(1);
    }
    int index = x * size_y * size_z + y * size_z + z;
    data[index] = 255;
}

void Buffer::hline(int x1, int x2, int y, real z) {
    if (x1 < 0 || x1 > size_x - 1 || x2 < 0 || x2 > size_x - 1 || y < 0 || y > size_y - 1 || z < 0 || z > size_z - 1) {
        cout << "Error: hline() - coordinates or height out of bounds" << endl;
        exit(1);
    }
    for (int i = x1; i <= x2; i++) {
        data[i * size_y * size_z + y * size_z + static_cast<int>(z * 255)] = 255;
    }
}

void Buffer::vline(int y1, int y2, int x, real z) {
    if (y1 < 0 || y1 > size_y - 1 || y2 < 0 || y2 > size_y - 1 || x < 0 || x > size_x - 1 || z < 0 || z > size_z - 1) {
        cout << "Error: vline() - coordinates or height out of bounds" << endl;
        exit(1);
    }
    for (int i = y1; i <= y2; i++) {
        data[x * size_y * size_z + i * size_z + static_cast<int>(z * 255)] = 255;
    }
}

void Buffer::box(int x1, int y1, int x2, int y2, real z) {
    if (x1 < 0 || x1 > size_x - 1 || x2 < 0 || x2 > size_x - 1 || y1 < 0 || y1 > size_y - 1 || y2 < 0 || y2 > size_y - 1 || z < 0 || z > size_z - 1) {
        cout << "Error: box() - coordinates or height out of bounds" << endl;
        exit(1);
    }
    for (int i = x1; i <= x2; i++) {
        for (int j = y1; j <= y2; j++) {
            data[i * size_y * size_z + j * size_z + static_cast<int>(z * 255)] = 255;
        }
    }
}

```

## 1 - Manipulação de dados

Ao final deste módulo, você será capaz de definir instruções para manipulação simples de dados.

# Manipulação singular de dados

Refere-se à prática de escrever códigos simples para tratar dados de forma individual, sem a capacidade de manipular múltiplos dados eficientemente. Nesse contexto, vamos explorar instruções básicas para manipulação de dados, como carregar uma imagem, aplicar zoom e apresentar a imagem na tela.

Esse método envolve etapas simples: carregamos a imagem para a memória; em seguida, aplicamos um zoom para ampliar a imagem; depois, exibimos a imagem ampliada na tela. Essas instruções básicas são essenciais para entender os fundamentos da manipulação de dados em programação.

Neste vídeo, apresentaremos uma introdução à manipulação singular de dados no contexto da edição básica de imagens, incluindo instruções fundamentais de programação. Confira!

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Neste conteúdo, vamos aprender a escrever códigos simples de computador para manipulação singular de dados.

Por singular, entenderemos a ausência de condições de realizar uma manipulação de múltiplos dados de forma eficiente.

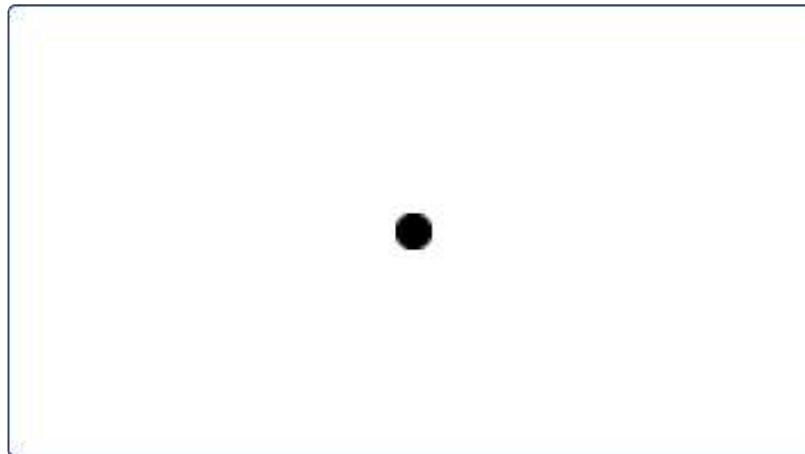
Vamos com calma! Um pixel de cada vez...

Para ilustrar melhor nossa discussão, usaremos imagens para aplicação de nossos códigos/instruções.



Como são instruções simples de manipulação de dados, você notará que, nos exemplos, manipularemos apenas um pixel de cada vez.

Para começar, observe a imagem a seguir, chamada circulo.bmp, que consiste em uma grade de 10x10 pixels.



Pequena imagem de 10 pixels de largura por 10 pixels de altura com fundo branco e círculo preto ao meio.

A imagem tem um fundo branco, com um círculo desenhado.

O **BMP** é um formato de arquivo usado para armazenar imagens digitais.

Não se preocupe, pois o tamanho da figura não é um erro. A imagem realmente está muito pequena.

## BMP

– Bitmap image file ou arquivo de imagem bitmap.

Nosso primeiro impulso é realizar uma ampliação ou zoom para vê-la melhor. Esse é o ímpeto que queríamos provocar em você quando a colocamos com esse tamanho.

### Como escrever um código que instrua o computador a ampliar a imagem circulo.bmp?

Para isso, usaremos três instruções que fazem parte do conjunto de funções da linguagem de programação JavaScript, veja!

Passo	Instrução	Significado em português
1	<pre>img = newSimpleImage("circulo.bmp");</pre>	Carrega a imagem circulo.bmp na memória e a

		armazena na variável que, neste exemplo, chamamos de img.
2	img.setZoom(20);	Estabelece ampliação de 20 vezes o tamanho original para a imagem armazenada na variável img.
3	print(img);	Apresenta a imagem na tela.

Tabela 1: ampliando e apresentando na tela a imagem circulo.bmp.

Guilherme Dutra Gonzaga Jaime.

Embora sejam simples, as instruções na tabela 1 nos permitiram instruir o computador a ampliar imagens digitais conforme nosso comando.

Agora que você aprendeu como ampliar a imagem circulo.bmp, vamos experimentar para atingir um nível de ampliação confortável.

## Atividade 1

Qual das sequências de instruções representa corretamente o processo de manipulação singular de dados para carregar uma imagem, aplicar zoom e exibi-la na tela?

A  
 img = new SimpleImage("circulo.bmp") /  
 img.setZoom(20) / print(img);

B

img = new SimpleImage("circulo.bmp") / print(img) /  
img.setZoom(20);

C print(img) / img = new SimpleImage("circulo.bmp") /  
img.setZoom(20);

D img.setZoom(20) / print(img) / img = new  
SimpleImage("circulo.bmp");

E img.setZoom(20) / img = new  
SimpleImage("circulo.bmp") / print(img);

Parabéns! A alternativa A está correta.

A sequência correta de instruções para manipulação singular de dados começa com o carregamento da imagem para a memória, seguido pela aplicação de zoom, e finaliza com a apresentação da imagem na tela. A alternativa A reflete exatamente esta ordem: a imagem é carregada na variável img com a instrução new SimpleImage("circulo.bmp"); em seguida, o zoom é aplicado com img.setZoom(20); finalmente, a imagem é exibida na tela com print(img). As outras opções apresentam uma sequência incorreta das instruções necessárias para esse processo.

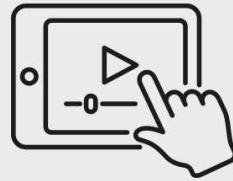
## Manipulando cada bit

Vamos utilizar duas novas instruções, as quais vão possibilitar alterar o zoom da imagem e modificar a cor do pixel. Compreender tais técnicas é fundamental para profissionais de TI, design gráfico e outras áreas que dependem de representações visuais precisas e de alta qualidade.

Neste vídeo, veremos como utilizar uma linguagem de programação para manipular imagens, ajustando as características de cada pixel.

Assista!

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Agora, vamos partir para um segundo cenário um pouco mais interessante.

Queremos estender o código usado na Prática 1 para sermos capazes de manipular individualmente cada pixel.

Para isso, tudo que precisamos fazer é ajustar os passos da tabela 1 e adicionar duas novas instruções logo antes da instrução `print`, veja a seguir. Essas instruções também fazem parte das funções padronizadas pela linguagem JavaScript.

Passo	Instrução	Significado em português
1	<code>img = new SimpleImage("circulo.bmp");</code>	Carrega a imagem circulo.bmp na memória e a armazena na variável chamada img.
2	<code>img.setZoom(20);</code>	20 vezes o tamanho original para a imagem armazenada na variável img. Um zoom menor do que 0 (zero) equivale a um afastamento. Por exemplo, um zoom de 0.5 equivale a um afastamento de 2x.
3	<code>pixel = img.getPixel(4,4)</code>	Obtém a referência ao pixel (4,4) da

		imagem armazenada na variável img e atribui essa referência à variável pixel.
4	pixel.setRed(255)	Instrui o computador a ajustar para 255 o nível de vermelho para o pixel em questão.
5	print(image);	Apresenta a imagem na tela.

Tabela 2: modificando um dos pixels de circulo.bmp para que ele fique vermelho.  
Guilherme Dutra Gonzaga Jaime.

Onde aparece a primeira instrução nova da tabela 2?

A primeira instrução nova que vemos na tabela 2 é a do passo 3, que instrui o computador a obter a referência ao pixel (4,4) da imagem armazenada na variável img e, então, a atribuir essa referência à variável pixel.

A partir de agora, qualquer operação que fizermos com a variável img será efetuada sobre o pixel (4,4).

Em seguida, no passo 4, a instrução pixel.setRed (255) ordena que o computador ajuste a saturação de vermelho do pixel para o nível 255, que é o maior valor possível.

## Considerações

Assim como a função setRed() ajusta o nível de vermelho para o pixel, temos duas outras funções análogas para manipular os níveis de azul e de verde de um pixel. A tabela 3 apresenta as três funções possíveis para essa manipulação, veja!

Passo	Instrução	Significado em português
1	<code>pixel.setRed(número)</code>	Ajusta o nível de vermelho do pixel conforme o número informado. O número deve ser entre 0 e 255.
2	<code>pixel.setGreen(número)</code>	Ajusta o nível de verde do pixel conforme o número informado. O número deve ser entre 0 e 255.
3	<code>pixel.setBlue(número)</code>	Ajusta o nível de azul do pixel conforme o número informado. O número deve ser entre 0 e 255.

Tabela 3: Três funções disponíveis para manipulações de cores de um pixel.  
Guilherme Dutra Gonzaga Jaime.

---

## Atividade 2

A manipulação de imagens e pixels é uma habilidade fundamental em programação para áreas como design gráfico, desenvolvimento de interfaces e outras aplicações visuais. Qual das seguintes opções corretamente descreve o processo para carregar uma imagem, aplicar zoom e mudar a cor de um pixel em JavaScript?

- Primeiro, use `pixel = img.loadPixel(4, 4);`, depois aplique  
**A** `img.zoomIn(2);` e mude a cor com  
`pixel.changeColor('red', 255);`.

- Primeiro, carregue a imagem com img =  
B loadImage("imagem.jpg");, depois aplique zoom com  
img.zoom(1.5); e mude a cor do pixel com pixel =  
img.getPixel(4, 4); pixel.setRed(255);
- Primeiro, use img = loadImage("imagem.jpg");, depois  
C aplique zoom com img.setZoom(3); e mude a cor com  
pixel = img.getPixel(2, 2); pixel.setColor('red', 255);
- Primeiro, carregue a imagem com img = new  
D Image("imagem.jpg");, aplique zoom com  
img.applyZoom(2); e mude a cor com pixel =  
img.accessPixel(4, 4); pixel.setColor('red', 255);
- Primeiro, carregue a imagem com img = new  
E SimpleImage("imagem.jpg");, depois aplique zoom com  
img.setZoom(2); e mude a cor do pixel com pixel =  
img.getPixel(4,4); pixel.setRed(255);

Parabéns! A alternativa E está correta.

A alternativa E descreve corretamente as etapas para carregar uma imagem, aplicar zoom e mudar a cor de um pixel em JavaScript. O comando img = new SimpleImage("imagem.jpg"); carrega a imagem na memória. O comando img.setZoom(2); aplica o zoom, ampliando a imagem. Finalmente, o comando pixel = img.getPixel(4, 4); pixel.setRed(255); acessa um pixel específico na posição (4, 4) e altera seu valor de cor vermelha para 255. Essa sequência de comandos demonstra a manipulação simples e eficaz de imagens e pixels, essencial para entender a manipulação singular de dados em programação.

## Manipulando imagens na prática

Após todo o conhecimento teórico adquirido, chegou a hora de colocar a mão na massa e aplicar esses conceitos.

Vamos praticar?

# Prática 1

À esquerda, temos o **Código-Fonte**; à direita, a **Saída** resultante do processamento das instruções fornecidas. Logo abaixo, o botão **Executar**, que é a forma como ordenamos ao computador que executa as instruções fornecidas.

## Código-Fonte

```
img = new SimpleImage("img/circulo.bmp");
img.setZoom(20);
print(img);
```

Rodar/Executar

## Saída

Após clicar no botão **Executar** e observar o resultado em **Saída**, compare a seguir a solução:

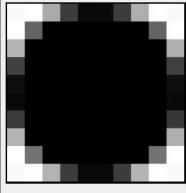
Mostrar solução

Código-Fonte

```
img = new SimpleImage("img/circulo.bmp");
img.setZoom(20);
print(img);
```

Rodar/Executar

Saída



A screenshot of a programming environment showing a code snippet and its output. The code uses the SimpleImage class to load an image named 'circulo.bmp', sets its zoom level to 20, and prints it. The resulting output is a small, solid black circle with a diameter of 20 pixels, centered on a white background.

Percebeu a diferença?

**Note que aparece um círculo de 20x20 pixels, maior do que o apresentado na primeira figura.**

Agora altere a instrução setZoom para realizar um zoom de 10 vezes em vez de 20. Clique em Executar e observe o resultado.

Mostrar solução

Código-Fonte

```
img = new SimpleImage("img/circulo.bmp");
img.setZoom(10);
print(img);
```

Rodar/Executar

Saída



A screenshot of a programming environment showing a code snippet and its output. The code is identical to the previous one, but the zoom level is set to 10 instead of 20. The resulting output is a very small, solid black circle with a diameter of 10 pixels, centered on a white background.

Note que a execução das instruções resulta em um círculo de 10x10 pixels, agora menor que o anterior.

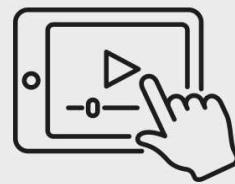
### Dica

Fique à vontade: pratique e experimente com qualquer valor que deseje para a instrução setZoom. Não tenha medo de errar!

## Prática 2

A seguir, assista a uma breve contextualização da Prática 1, seguida do nosso segundo exemplo prático.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Agora, vamos realizar alguns experimentos simples com o código da tabela 2, manipulando o código-fonte a seguir para observarmos o que acontece:

### Código-Fonte

```
img = new SimpleImage( "img/circulo.bmp" );
img.setZoom( 20 );
pixel = img.getPixel( 4, 4 );
pixel.setRed( 255 );
print(img);
```

Rodar/Executar

### Saída

Após clicar no botão **Executar** e observar o resultado em **Saída**, compare a solução:

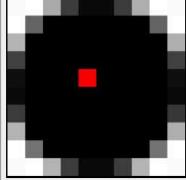
Mostrar solução

Código-Fonte

```
img = new SimpleImage( "img/circulo.bmp" );
img.setZoom( 20 );
pixel = img.getPixel( 4, 4 );
pixel.setRed( 255 );
print(img);
```

Rodar/Executar

Saída



Note que o pixel (4,4), que fica mais ou menos no meio do círculo, ficou vermelho.

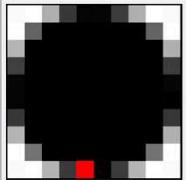
Agora, observe as soluções em alterações na mudança de alguns parâmetros:

Código-Fonte

```
img = new SimpleImage( "img/circulo.bmp" );
img.setZoom( 20 );
pixel = img.getPixel( 4, 9 );
pixel.setRed( 255 );
print(img);
```

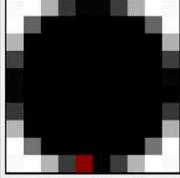
Rodar/Executar

Saída



### Alternando o parâmetro da função getPixel para (4, 9)

Observe que, como fornecemos a coordenada de outro pixel (linha 9 em vez de linha 4), o pixel vermelho ficou mais abaixo.

<b>Código-Fonte</b> <pre>img = new SimpleImage( "img/circulo.bmp" ); img.setZoom( 20 ); pixel = img.getPixel( 4, 9 ); pixel.setRed( 120 ); print(img);</pre>	<b>Saída</b> 
---	---

Colocando diferentes valores para o parâmetro da função `setRed`

Observe que, quanto menor o valor da saturação vermelha do pixel, mais escuro será o vermelho apresentado no pixel em questão.

## Atividade 3

Considerando os experimentos realizados na Prática 1, qual seria o código para que o resultado do zoom fosse um afastamento de 2 x em vez de uma aproximação de 2x?

A      `img = new SimpleImage( "img/circulo.bmp" );
 img.setZoom( 2 );
 print(img);`

B      `img = new SimpleImage( "img/circulo.bmp" );
 img.setZoom( -2 );
 print(img);`

C      `img = new SimpleImage( "img/circulo.bmp" );
 img.setZoom( -0.5 );
 print(img);`

D  

```
img = new SimpleImage( "img/circulo.bmp" );
img.setZoom( 0.5 );
print(img);
```

E  

```
img = new SimpleImage( "img/circulo.bmp" );
img.setZoom( -20 );
print(img);
```

Parabéns! A alternativa D está correta.

Conforme ilustrado na tabela 2, um afastamento de 2x equivale a um zoom de 0.5.

Passo	Instrução	Significado em português
1	<pre>img = new SimpleImage("circulo.bmp");</pre>	Carrega a imagem circulo.bmp na memória e a armazena na variável chamada img.
2	<pre>img.setZoom(20);</pre>	20 vezes o tamanho original para a imagem armazenada na variável img. Um zoom menor do que 0 (zero) equivale a um afastamento. Por exemplo, um zoom de 0.5 equivale a um afastamento de 2x.

		Obtém a referência ao pixel (4,4) da imagem armazenada na variável img e atribui essa referência à variável pixel.
3	pixel = img.getPixel(4,4)	
4	pixel.setRed(255)	Instrui o computador a ajustar para 255 o nível de vermelho para o pixel em questão.
5	print(image);	Apresenta a imagem na tela.

Tabela 2: modificando um dos pixels de circulo.bmp para que ele fique vermelho.  
Guilherme Dutra Gonzaga Jaime.

---

## Evoluindo a prática

Vamos praticar um pouco mais sobre manipulação simples de dados usando imagens, através das práticas 3 e 4 a seguir.

### Prática 3

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Vamos realizar alguns experimentos simples com o código da tabela 3, manipulando o código-fonte a seguir para observar o que acontece:

**Código-Fonte**

```
img = new SimpleImage( "img/circulo.bmp" );
img.setZoom( 20 );
pixel = img.getPixel( 4, 0 );
pixel.setRed( 255 );
print(img);
```

**Rodar/Executar****Saída**

Durante essa prática, observe com atenção a proposição em português feita no enunciado, em que o efeito desejado é descrito.

Depois, tente refletir:

**Qual é o código de computador que você deve escrever, ou seja, as instruções e os valores digitados no código-fonte para alcançar o efeito proposto?**

Na prática, você aprenderá a traduzir:



Do português



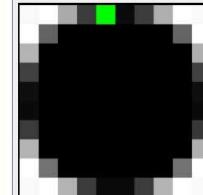
Para a linguagem do computador

O código fornecido ajusta o pixel (4,0) para vermelho. Ajuste o código para que o pixel (4,0) fique verde. Veja na solução:

Código-Fonte

```
img = new SimpleImage( "img/circulo.bmp" );
img.setZoom( 20 );
pixel = img.getPixel( 4, 0 );
pixel.setRed( 0 );
pixel.setGreen( 255 );
pixel.setBlue( 0 );
print(img);
```

Saída

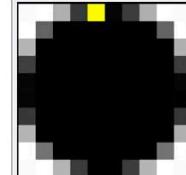


Agora, observe as soluções em alterações na mudança de alguns parâmetros:

Código-Fonte

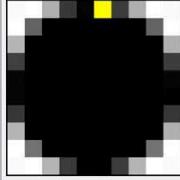
```
img = new SimpleImage( "img/circulo.bmp" );
img.setZoom( 20 );
pixel = img.getPixel( 4, 0 );
pixel.setRed( 255 );
pixel.setGreen( 255 );
pixel.setBlue( 0 );
print(img);
```

Saída



### Manipulação do pixel (4,0) para que fique amarelo

Aqui, o pixel (4,0) estava originalmente preto, ou seja, código RGB (0,0,0). Como sabemos que o amarelo é a combinação de verde e vermelho, a solução é bem simples: basta adicionar o nível máximo (255) dessas cores constituintes.

<b>Código-Fonte</b> <pre>img = new SimpleImage( "img/circulo.bmp" ); img.setZoom( 20 ); pixel = img.getPixel( 5, 0 ); pixel.setRed( 255 ); pixel.setGreen( 255 ); pixel.setBlue( 0 ); print(img);</pre>	<b>Saída</b> 
--	---

**Rodar/Executar**

### Manipulação do pixel (5,0) para que fique amarelo

A solução aqui é muito simples: basta alterar o código da solução anterior para que a instrução `getPixel( 4,0 )` passe a ser `getPixel( 5,0 )`.

## Prática 4

Suponha que, em vez de manipular apenas um pixel por vez, conforme fizemos neste módulo, desejássemos manipular todos os pixels de uma imagem, digamos de 10x10 pixels (exemplo: `circulo.bmp`). Se usássemos o código que aprendemos a escrever, teríamos de escrever 100 vezes a instrução `pixel = img.getPixel()`, em que, a cada vez, passaríamos os valores `x,y` para cada um dos pixels da imagem.

Isso traz vários problemas.

1. O primeiro deles é que o código ficaria difícil de ler/compreender.
2. O segundo é que, se for uma imagem maior, de 1980x1024 pixels, teríamos cerca de 2 milhões de pixels.

Portanto, a instrução `pixel = img.getPixel()` teria de ser escrita cerca de 2 milhões de vezes no código fonte, a cada vez fazendo referência para um pixel diferente.

Isso seria inviável, não é mesmo?

**Mostrar solução**

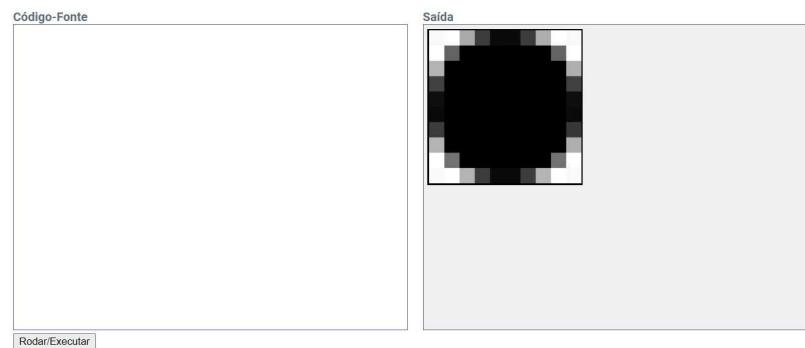
No mundo real, quase sempre desejamos realizar operações com uma quantidade muito grande de dados. Os engenheiros e

cientistas de computação pensaram nesta questão décadas atrás.

**Felizmente, há técnicas de programação/codificação que nos permitem manipular uma quantidade arbitrária de dados, sem ter de escrever muitos códigos de computador.**

## Atividade 4

Considere a imagem a seguir, chamada "circulo.bmp":



Qual é o código de computador necessário para manipular o pixel do canto superior esquerdo para que fique vermelho?

```
img = new SimpleImage( "img/circulo.bmp" );
img.setZoom( 20 );
```

A      

```
pixel = img.getPixel( 0, 0 );
pixel.setRed( 255 );
pixel.setGreen( 0 );
pixel.setBlue( 0 );
```

```
img = new SimpleImage( "img/circulo.bmp" );
img.setZoom( 20 );
```

B      

```
pixel = img.getPixel( 0, 0 );
pixel.setRed( 0 );
pixel.setGreen( 255 );
pixel.setBlue( 0 );
```

```
img = new SimpleImage( "img/circulo.bmp" );
img.setZoom( 20 );
```

C      pixel = img.getPixel( 0, 0 );
pixel.setRed( 0 );
pixel.setGreen( 0 );
pixel.setBlue( 255 );

```
img = new SimpleImage( "img/circulo.bmp" );
img.setZoom( 20 );
```

D      pixel = img.getPixel( 1, 1 );
pixel.setRed( 255 );
pixel.setGreen( 0 );
pixel.setBlue( 0 );

```
img = new SimpleImage( "img/circulo.bmp" );
img.setZoom( 20 );
```

E      pixel = img.getPixel( 1, 1 );
pixel.setRed( 0 );
pixel.setGreen( 255 );
pixel.setBlue( 255 );

Parabéns! A alternativa A está correta.

Conforme convenção de coordenadas de pixels, o pixel do canto superior esquerdo será sempre referenciado pela coordenada (0,0). Além disso, segundo o esquema RGB, a cor vermelha pura é representada pelo código RGB (255,0,0).

Veja a solução:

```
filename = "Input4K.txt"
flagCheckRicName = False

def checkRicName(filename, RICname):
    with open(filename) as f:
        for i, line in enumerate(f):
            if "<Name value>" in line:
                ricName = line
                flagCheckRicName = True
    print(filename, flagCheckRicName)
```

## 2 - Repetição for

Ao final deste módulo, você será capaz de distinguir a estrutura da repetição for.

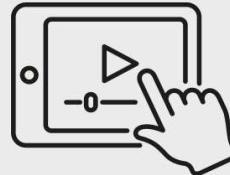
# Estrutura de repetição for

É essencial em linguagens de programação devido à sua eficiência na automatização de tarefas repetitivas. A estrutura de repetição for permite iterar sobre coleções de dados, processar elementos individualmente e realizar múltiplas operações de maneira sistemática.

O uso do for facilita o controle preciso sobre o número de iterações, tornando o código mais legível e menos propenso a erros. Essa estrutura é indispensável para operações como manipulação de arrays, geração de sequências numéricas e processamento de listas. Ela otimiza o desenvolvimento de algoritmos e a execução de tarefas complexas de forma organizada e eficiente.

Neste vídeo, vamos descobrir como usar a estrutura de repetição for para modificar as características de imagens. Confira!

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Estruturas de repetição são extremamente importantes, pois aumentam significativamente a capacidade de quem escreve códigos, permitindo a manipulação eficiente de múltiplos dados, ao contrário dos códigos que

Iidam apenas com dados individuais. Vamos explorar agora os princípios dessas estruturas.

Observe a imagem de um pássaro.



Imagen com 584 pixels de largura e 500 pixels de altura representando um pássaro.

Se multiplicarmos o número de pixels de largura pelo número de pixels de altura, teremos o total de pixel da imagem.

Nesse caso:

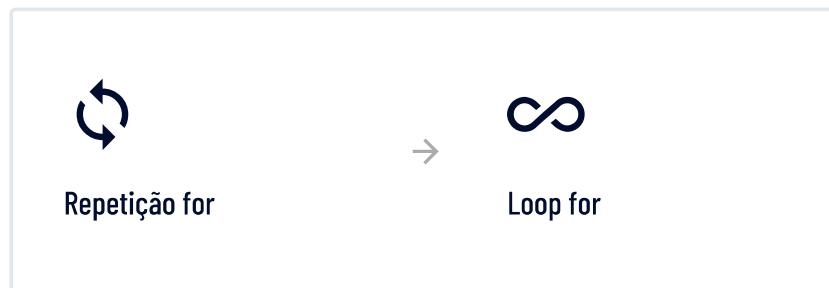
$$584 \times 500 = 292.000 \text{ pixels ou quase } 0.3 \text{ megapixels.}$$

Essa nem é uma imagem tão grande, dado que os telefones celulares atuais são capazes de tirar fotografias de mais de 10 milhões de pixels (10 megapixels). Mesmo assim, se o autor de um código de computador, sem usar estruturas de repetição, tentar escrever um código de computador para manipular cada um dos pixels da imagem, precisará repetir 292 mil vezes as instruções que apontam para determinado pixel (exemplo: `pixel = img.getPixel( x,y )`), cada uma delas seguidas com as instruções de manipulação desejada (exemplo: `pixel.setRed(255)`).

Claramente, isso não é uma maneira prática de realizar uma operação com grande quantidade de dados. As desvantagens vão desde um código de computador difícil de ler, por ser muito extenso, até a dificuldade de manutenção, atualização ou correção do código, caso haja necessidade futura.

O que buscamos é uma forma de escrever algumas poucas linhas de código que capturem as mudanças desejadas. Em seguida, podemos deixar o computador executar essas linhas repetidamente, uma vez para cada dado que precisamos manipular – nesse caso, uma vez para cada pixel da imagem.

Existem diferentes maneiras de realizar isso em códigos de computador, mas, por motivos de simplicidade, vamos estudar a estrutura de **repetição for**, também chamada de loop for.



Agora, analise a sintaxe da estrutura de repetição for apresentada a seguir.

```
for i in range(imagem.width):
    for j in range(imagem.height):
        r, g, b = pixels[i, j]
        # Aumentar o brilho adicionando um valor constante
        r = min(255, r + aumento)
        g = min(255, g + aumento)
        b = min(255, b + aumento)
        # Atualizar o pixel com os novos valores
        pixels[i, j] = (r, g, b)
```

Sintaxe da estrutura de repetição for destacada em negrito.

A linha de código `for i in range(image.width):`, traduzida de código de computador para português, significa: para cada posição horizontal da imagem armazenada na variável `image`, execute todas as instruções recuadas (em Python, usamos a indentação para definir o que pertence a esse bloco, e não chaves {} como em outras linguagens). Logo a seguir, temos `for j in range(image.height):`, que, para cada posição vertical da imagem, também executa as instruções abaixo.

Dentro dessa estrutura de repetição dupla, a primeira instrução `r, g, b = pixels[i, j]` lê as três cores (vermelho, verde e azul) do pixel localizado na posição `(i, j)` da imagem "pássaro.jpg". Depois, cada cor é aumentada de intensidade pela variável `aumento`, tomando cuidado para não ultrapassar o valor máximo permitido (255), usando a função `min`. Por exemplo, o novo valor de vermelho é calculado por `r = min(255, r + aumento)`, e o mesmo acontece para o verde e o azul.

Por fim, `pixels[i, j] = (r, g, b)` grava de volta no pixel  $(i, j)$  a nova cor ajustada.

Essas instruções são repetidas para cada um dos 292 mil pixels da imagem "pássaro.jpg", produzindo um resultado em que toda a imagem fica mais clara.

**Toda a sintaxe apresentada é requerida para que o computador entenda que se trata de uma estrutura for e quais são as instruções a serem repetidas.**

Na prática, o que o computador fará com o código apresentado é fixar o foco no primeiro pixel da imagem "pássaro.jpg", localizado na posição  $(0,0)$  — no canto superior esquerdo — e executar as instruções que leem os valores das cores vermelho, verde e azul. Em seguida, ele aumenta cada um desses valores em uma quantidade definida por aumento (sem ultrapassar o limite de 255) e reescreve o pixel com os novos valores. Esse primeiro pixel, portanto, ficará mais claro.

1

Em Python, não há o uso de chaves {} para indicar o começo e o fim do bloco de repetição. O que marca as instruções pertencentes ao for é o recuo (indentação) para a direita. Quando o computador chega ao final das instruções recuadas, ele retorna automaticamente para o início do laço for, passando para o próximo valor.

2

Assim, o computador se fixará no segundo pixel  $(1,0)$  da imagem e executará novamente as instruções: leitura das cores, ajuste dos valores e gravação do novo pixel. Esse processo se repetirá para o terceiro, quarto, quinto e todos os 292 mil pixels da imagem "pássaro.jpg", até que toda a imagem fique mais clara.

3

O recuo das instruções para a direita em Python não é apenas uma convenção visual: é **obrigatório**

Noções de programação: exemplos com manipulação de imagens digitais para que o interpretador comprehenda que as linhas fazem parte da estrutura de repetição. Sem o recuo correto, o código não funciona.

4

Mostrar que as linhas de dentro da estrutura de repetição são especiais em relação às demais é uma convenção muito comum, porque são executadas várias vezes, até que a condição estabelecida no início da estrutura for seja satisfeita.

Nesse exemplo, a condição para finalizar é que todas as instruções internas (leitura das cores, aumento dos valores e atualização do pixel) sejam executadas pelo computador para todos os pixels da imagem "pássaro.jpg".

A técnica de recuar a linha de código para deixar claro ao leitor que essas são instruções internas à estrutura de repetição é amplamente conhecida como indentação.

## Atividade 1

Estruturas de repetição são extremamente importantes, pois aumentam significativamente a capacidade de quem escreve códigos, permitindo a manipulação eficiente de múltiplos dados, ao contrário dos códigos que lidam apenas com dados individuais. Vamos considerar a imagem de um pássaro com 584 pixels de largura e 500 pixels de altura. Isso resulta em um total de 292.000 pixels. Qual das opções a seguir descreve corretamente a vantagem de usar a estrutura de repetição `for` para manipular cada pixel da imagem de um pássaro com 292.000 pixels?

- A A estrutura de repetição `for` elimina a necessidade de manipular pixels, tornando a imagem automaticamente perfeita.

- A estrutura de repetição for permite manipular todos os pixels da imagem com poucas linhas de código, tornando o processo mais eficiente e fácil de manter.
- C A estrutura de repetição for permite carregar a imagem na memória sem precisar usar variáveis adicionais.
- D A estrutura de repetição for permite aplicar zoom na imagem sem alterar a cor dos pixels.
- E A estrutura de repetição for é usada apenas para mudar a cor do primeiro pixel da imagem.

Parabéns! A alternativa B está correta.

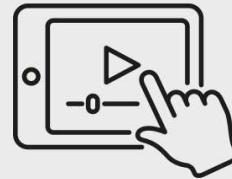
A estrutura de repetição for é essencial para a manipulação eficiente de grandes quantidades de dados, como pixels de uma imagem. Ao usar for, podemos escrever poucas linhas de código que o computador executa repetidamente para cada pixel, facilitando a leitura, a manutenção e a atualização do código. Isso contrasta com a abordagem impraticável de manipular cada pixel individualmente, que seria extremamente tediosa e propensa a erros.

## Praticando o uso do for na manipulação de imagens

Vamos praticar o uso da estrutura de repetição for na manipulação de imagens, através das quatro práticas a seguir.

### Prática 1

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



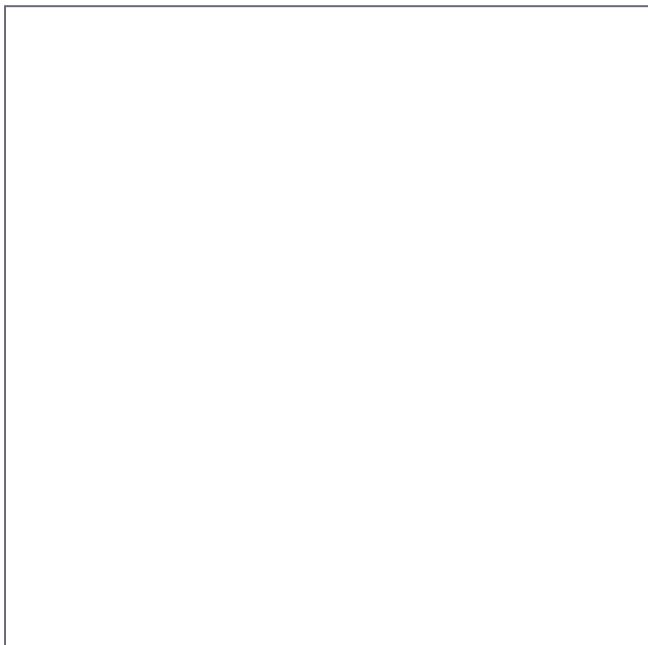
Vamos experimentar a estrutura *for*. Observe o código-fonte abaixo:

#### Código-Fonte

```
img = new SimpleImage("img/passaro.jpg");
for( pixel: img ){
    pixel.setRed(0);
    pixel.setGreen(0);
    pixel.setBlue(0);
}
print( img );
```

Rodar/Executar

#### Saída



Lembre-se de que, ao clicar em Executar, o computador realizará os seguintes passos, conforme instruções:

Instrução 1



Carregar a imagem “passaro.jpg” e armazená-la na variável img.

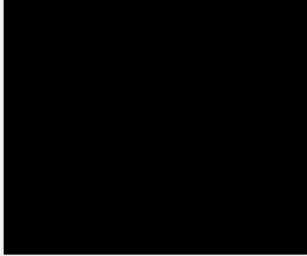
### Instrução 2

Para cada um dos 292 mil pixels da imagem “passaro.jpg”, executar as instruções indicadas:

- Ajustar intensidade de vermelho do pixel para 0 (zero);
- Ajustar a intensidade de verde do pixel para 0 (zero);
- Ajustar a intensidade de azul do pixel para 0 (zero);
- Imprimir/apresentar a imagem na tela.

Note que o computador as executará para cada um dos 292 mil pixels da imagem. Então, temos um total de 876 mil operações a serem realizadas pelo computador. Ao clicar em **Executar**, tente observar quanto tempo o computador leva para ajustar/executar as 876 mil instruções necessárias para colorir de preto cada um dos 292 mil pixels da imagem “passaro.jpg” e, depois, apresentar a imagem na tela.

A imagem preta deve aparecer na tela muito rapidamente:

<p>Código-Fonte</p> <pre>img = new SimpleImage("img/passaro.jpg"); for( pixel: img ){     pixel.setRed(0);     pixel.setGreen(0);     pixel.setBlue(0); } print( img );</pre>	<p>Saída</p> 
---	---

Ao executar a Prática 1, você obteve um retângulo preto, o que não é muito útil. Entretanto, não deixe de refletir sobre o quanto rapidamente o computador foi capaz de seguir suas instruções para manipular os quase 300 mil pixels da imagem.

## Prática 2

Vamos seguir assistindo a um vídeo com uma segunda prática.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



## Prática 3

Agora, vamos obter o canal alfa vermelho da imagem do pássaro apresentado na figura 2 com base neste código-fonte:

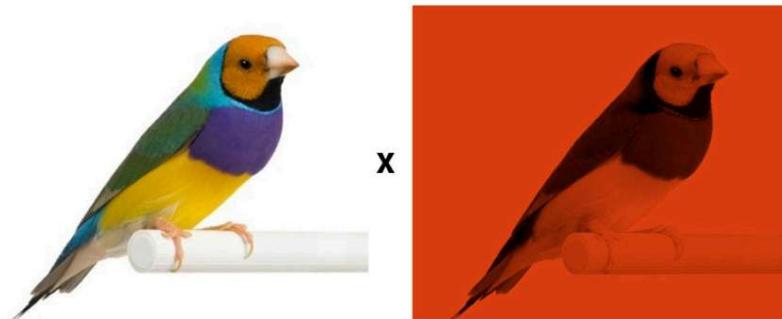
Execute o código-fonte e observe o canal alfa em vermelho da imagem apresentada:

<b>Código-Fonte</b> <pre>img = new SimpleImage("img/passaro.jpg"); for( pixel: img ) {     // pixel.setRed(0);     pixel.setGreen(0);     pixel.setBlue(0); } print( img );</pre>	<b>Saída</b> 
--	------------------

O que você notou?

Assim como na Prática 1, observe o quanto rapidamente o computador foi capaz de remover os componentes azul e verde da imagem em questão. Lembre-se: as instruções de dentro da estrutura *for* foram repetidas 876 mil vezes.

Compare com a imagem original da figura 2, replicada logo a seguir (por comodidade), com a imagem resultante do item “a”.



Observe, também, que o lindo colar azul do pássaro praticamente desapareceu na imagem canal alfa vermelho resultante. Isso significa que o colar é constituído principalmente dos componentes verde e azul.

Você compreendeu como o canal alfa em vermelho foi obtido?

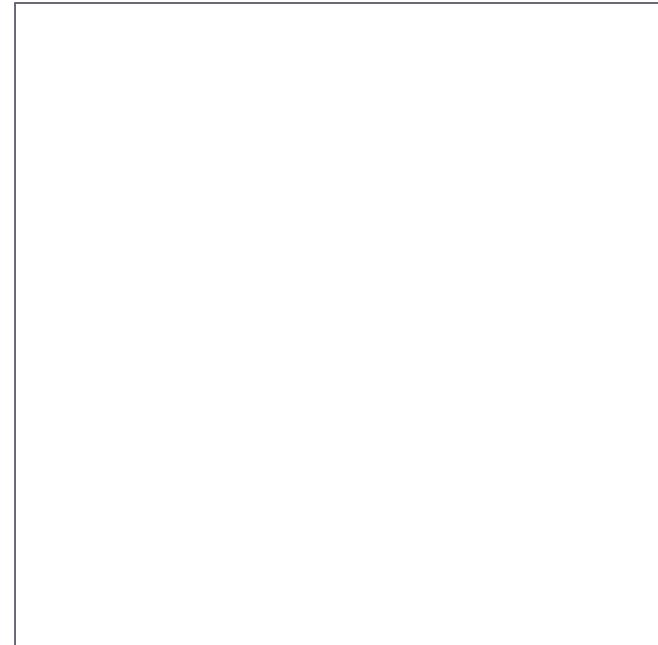
Para compreender melhor o código que foi executado, observe na caixa de código-fonte que a instrução `pixel.setRed()` está comentada, ou seja, marcada com “//” no início da linha:

#### Código-Fonte

```
img = new SimpleImage("img/passaro.jpg");
for( pixel: img ){
    // pixel.setRed(0);
    pixel.setGreen(0);
    pixel.setBlue(0);
}
print( img );
```

Rodar/Executar

#### Saída



Comentários de códigos são interessantes por duas razões: A primeira, é permitir que o autor do código-fonte documente seu código com frases que esclarecem o que ele estava pensando quando escreveu aquele trecho; A segunda, é fazer com que o computador ignore uma ou mais instruções.

**Você consegue alterar o código-fonte para obter os outros canais? Tente!**

Obtenha os canais verde e azul respectivamente.

<b>Código-Fonte</b> <pre>img = new SimpleImage("img/passaro.jpg"); for( pixel: img ){     pixel.setRed(0);     // pixel.setGreen(0);     pixel.setBlue(0); } print( img );</pre>	<b>Saída</b> 
---	--

Verde

<b>Código-Fonte</b> <pre>img = new SimpleImage("img/passaro.jpg"); for( pixel: img ){     pixel.setRed(0);     pixel.setGreen(0);     // pixel.setBlue(0); } print( img );</pre>	<b>Saída</b> 
---	---

Azul

## Prática 4

Afinal para que serve a estrutura de repetição *for*?

### Atenção!

A estrutura de repetição *for* é um recurso muito poderoso, que nos permite escrever algumas poucas linhas de código capazes de ordenar ao computador que processe/manipule uma enorme quantidade de dados.

Neste módulo, usamos exemplos de imagens para demonstrar o enorme potencial das estruturas de repetição. Este foi apenas um exemplo.

Esteja certo de que essa estrutura é amplamente usada em computação e faz parte do dia a dia de todos que, de alguma forma, escrevem

códigos para computador. Por fim, devemos mencionar um detalhe da linguagem usada aqui: JavaScript. Essa linguagem não possui uma estrutura de repetição *for* tão compacta e simples como a que estudamos.



Usamos uma versão com sintaxe propositalmente simplificada, pois nosso objetivo era compreender os princípios que motivam o uso de estruturas de repetição. Entendemos que o uso de uma estrutura simplificada ajudaria você a compreender melhor o que se passa.

Não queríamos gastar muito tempo fazendo-o compreender detalhes de sintaxe. Este comentário é importante, pois, se você tentar aproveitar os códigos de computador que escrevemos aqui, provavelmente terá problemas devido ao nosso uso simplificado da estrutura de repetição *for*.

Há outras linguagens de programação que possuem estrutura de repetição bem similar àquela que usamos aqui, mas não JavaScript.

## Atividade 2

Suponha a imagem a seguir, de 1.400 pixels de largura por 932 de altura:



Lembre-se de que o total de pixels de uma imagem é obtido pela multiplicação do número de pixels de altura pelo número de pixels de largura. Suponha que desejemos escrever um código de computador que retire os componentes de azul e de vermelho para cada pixel da imagem.

Para isso, as instruções seriam:

```
img.getPixel( 0,0 );
pixel.setBlue( 0 );
pixel.setRed( 0 );
```

Se escrevêssemos um código para cada um dos pixels da imagem, quantas linhas totais teríamos de escrever?

A 3.914.400

B 1.304.800

C 2.609.600

D 652.400

E 2.332

Parabéns! A alternativa A está correta.

O total de pixels da imagem é 1.400x932, ou seja, 1.304.800 pixels. Como são três instruções por pixel, o programador teria de escrever  $3 \times 1.304.800 = 3.914.400$  (3,9 milhões) de linhas de código. Obviamente, isso seria inviável. Então, é fundamental usar a estrutura de repetição `for`, que nos permite escrever a mesma solução com cerca de cinco linhas de código.



### 3 - As expressões em código de computador

Ao final deste módulo, você será capaz de reconhecer expressões usadas em códigos de computador.

## Expressões

Permitem manipular e processar dados de forma dinâmica, simplificando a resolução de problemas e a criação de soluções personalizadas em nossos programas. Essas expressões são fundamentais para desenvolver habilidades de programação e criar códigos mais eficientes em JavaScript, essenciais para uma programação eficaz.

Confira neste vídeo a importância da utilização de expressões simples de programação para manipulação de imagens.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Expressões são muito importantes, pois, em vez de usar apenas valores simples em instruções, podemos combinar diferentes valores para calcular parâmetros mais personalizados. Isso nos permite escrever códigos de computador que alcançam resultados mais realistas. Quando apresentarmos exemplos práticos de manipulação de imagens, você entenderá melhor o assunto.

Usaremos a linguagem de programação JavaScript, com algumas simplificações para facilitar a compreensão. Se analisarmos o código/instrução a seguir, veremos um tipo bem simples de passagem de valor (51):



Dizemos que 51 é o valor passado como parâmetro para a função print.

As linguagens de programação compreendem códigos que envolvem expressões.

### Exemplo

```
print(40+11);
```

Então, o termo 40+11, passado como parâmetro para a função print, é o que chamamos de expressão.

Basicamente, em vez de um número fixo conhecido antecipadamente, podemos usar um pouco de expressão aritmética. Assim, temos uma forma mais rica de fornecer parâmetros para funções.



Isso funciona da seguinte maneira: quando o computador executar a linha de código que contém a expressão, o primeiro passo será avaliá-la. Assim, ele lerá e resolverá a expressão para chegar ao valor resultante. Nesse exemplo, `40+11` é apenas uma soma. O computador calcula que `40+11` é igual a `51`. Uma vez que a expressão foi avaliada e o resultado foi `51`, então, o computador continua a execução do código, usando o resultado da expressão como parâmetro para a função. Em nosso exemplo o computador simplesmente imprimirá o valor `51` na tela.

De fato, podemos usar expressões em qualquer local do código de computador onde valores numéricos são admitidos. Portanto, é possível sempre embutir expressões aritméticas para que seja computado o valor que desejamos usar de fato. Isso nos possibilita resolver problemas mais realistas.

Antes de continuar, vamos conhecer três funções de manipulações de pixels que fazem parte da linguagem de programação JavaScript.

Passo	Instrução	Significado em português
1	<code>pixel.getRed()</code>	Obtém/Lê o valor atual para o componente RGB vermelho do pixel.
2	<code>pixel.getGreen()</code>	Obtém/Lê o valor atual para o componente RGB verde do pixel.
3	<code>pixel.getBlue()</code>	Obtém/Lê o valor atual para o componente RGB azul do pixel.

Tabela 4: Dado um pixel da imagem, como saber qual é o valor atual para os componentes RGB do pixel?

Guilherme Dutra Gonzaga Jaime.

Suponha um pixel em amarelo (exemplo: Código RGB `255,255,0`). Nesse caso, a função `pixel.getRed()` retornará o valor `255`, pois este é o valor atual do componente vermelho (`R`) do pixel.

De forma análoga, a função `pixel.getBlue()` retornaria `0` (zero), pois este é o valor atual do componente azul (`B`) do pixel em questão. Isso ocorre porque o amarelo puro não possui azul em sua composição.

Agora, imagine duplicar o valor atual do componente vermelho de um pixel. Se esse valor for `50`, ele será ajustado para `100`; se for `105`, será

ajustado para 210. Em resumo, o objetivo é fazer um ajuste relativo no valor, duplicando o valor atual, seja qual for.

Já conhecemos a função `pixel.getRed()` e a possibilidade do uso de expressões. Logo, a tarefa ficou mais fácil. Uma primeira solução seria usar este código:

JavaScript



Agora veja as informações sobre estas linhas:

## 1<sup>a</sup> linha

Chama `pixel.getRed()` e armazena o valor retornado na variável `ultimo`. Escolhemos esse nome para indicar que ele representa o último valor lido do componente vermelho do pixel.

## 2<sup>a</sup> linha

Usa a função `pixel.setRed()` para informar que o novo valor deste pixel será o dobro do último valor.

Confira o que acontece ao iniciar a execução da segunda linha:

1. O primeiro passo do computador será avaliar qual é o resultado da expressão `ultimo*2`. Se imaginarmos que o último valor para o pixel é 60, a expressão multiplicará esse valor por 2, o que resultará em 120.
2. Em seguida, o computador executará a função `pixel.setRed()` com o valor 120 como parâmetro, o que dobrará a intensidade de vermelho do pixel em questão.
3. Como resultado, as duas linhas de código efetivamente duplicam a intensidade de vermelho do pixel atual.

Agora que entendemos os princípios do uso de expressões em códigos de computadores, vamos analisar o mesmo exemplo de duplicação do

valor atual da intensidade de vermelho de um pixel, porém com uma solução muito mais comum no mundo real.

Veja a linha de código a seguir:

JavaScript



Na prática, desejamos mostrar que a solução em questão pode ser condensada em apenas uma linha de código.

Note que a variável `ultimo` não tinha um objetivo muito importante no código. Ela apenas armazenava temporariamente o último valor do componente vermelho do pixel para que este fosse usado na instrução seguinte.

Vamos supor que o pixel em questão esteja com os valores RGB (50,20,30). Então, o componente vermelho possui o valor 50. O computador fará o seguinte:

---

1

Executar a instrução  
`pixel.getRed()` para obter o valor  
atual do pixel, que é 50.

---

2

Avaliar a expressão  
`pixel.getRed()*2` e resolvê-la,  
multiplicando o valor atual do  
pixel por 2 e obtendo o resultado  
100.

### 3

Executar a instrução  
pixel.setRed() usando como  
parâmetro o resultado da  
operação aritmética do passo 2,  
ou seja, o valor duplicado do  
componente vermelho do pixel,  
que é 100.

Com isso, o novo valor RGB para o pixel em questão será RGB  
(100,20,30). Se fizermos o mesmo para todos os pixels de uma imagem,  
o usuário perceberá essa alteração como uma imagem com os tons de  
vermelho mais destacados.

Se, no mesmo exemplo, desejássemos reduzir pela metade a  
intensidade de vermelho do pixel, a linha de código ficaria assim:

JavaScript



Note que, em vez de usarmos o asterisco, que denota multiplicação,  
utilizamos o "/", que denota divisão.

Outra forma de escrever a mesma solução seria multiplicar o pixel atual  
por 0.5, o que é o mesmo que dividir por 2. Então, nesse caso, teríamos:

JavaScript



## Prática 1

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Vamos ajustar a tonalidade de cores de uma imagem por meio de instruções simples. Comecemos por esta:



Figura 4: Imagem de flores em amarelo com 587x330 pixels, totalizando quase 194 mil pixels.

Suponha que desejemos ajustar essa imagem para obtermos tonalidades mais para o laranja.

**Para alcançar o objetivo desta prática, basta reduzir um pouco a intensidade do componente verde (G) de cada um dos pixels da imagem.**

JavaScript



O que essa instrução faz é ajustar a intensidade de verde do pixel em questão para 70% do valor atual. Esse é o resultado aritmético que você obtém quando multiplica um valor por 0.7.

Note que o código-fonte fornecido não contém a instrução citada.

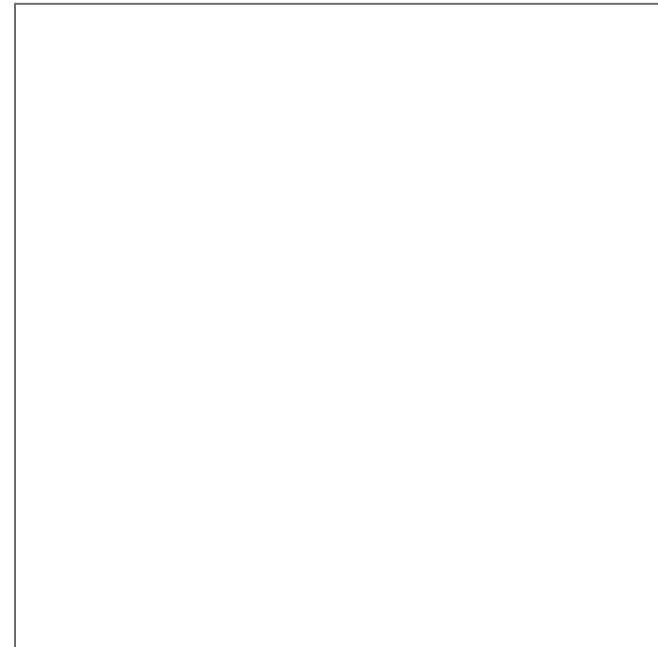
Apenas disponibilizamos o “esqueleto” do código contendo a solução:

#### Código-Fonte

```
img = new SimpleImage("img/flores.jpg");
for( pixel: img ){
    // Insira suas linhas de código abaixo
}
print( img );
```

Rodar/Executar

#### Saída



Se você clicar em **Executar**, o computador apenas apresentará a imagem original com as flores em amarelo. Isso ocorre porque não há instruções de ajuste de cores no código-fonte.

Você mesmo deve inserir a instrução que realiza a redução da tonalidade de verde do pixel. Insira a instrução dentro da estrutura de repetição for logo abaixo do comentário que indica a posição correta. Depois, clique em **Executar** para que o computador execute seu código. Observe o resultado.

Se tiver dúvidas, clique em **Solução**, mas não deixe de tentar implementar sua solução no código-fonte fornecido. Basta inserir uma linha de código!

Mostrar solução

Aqui, vemos lindas flores de cor laranja! Experimente reduzir ainda mais o componente de verde da figura. Observe que, quanto menor o componente de verde, mais as cores das flores se aproximam do vermelho.

Código-Fonte

```
img = new SimpleImage("img/flores.jpg");
for( pixel: img ){
    // Insira suas linhas de código abaixo
    pixel.setGreen( pixel.getGreen()*0.7 );
}
print( img );
```

Rodar/Executar

Saída



The screenshot shows a programming interface with a pink border. At the top, there's a button labeled 'Mostrar solução' with a dropdown arrow. Below it is a text area containing a Java-like code snippet for manipulating an image. To the right of the code is a preview window titled 'Saída' showing a photograph of yellow flowers. At the bottom left is a small button labeled 'Rodar/Executar'.

## Considerações

Reflita sobre a quantidade de operações realizada em um piscar de olhos pelo computador, dado o código-fonte que geramos.

Para cada um dos 194 mil pixels da imagem da figura 4, o computador fez as seguintes operações:

- Obteve a intensidade de verde do pixel;
- Multiplicou a intensidade de verde do pixel por 0.7;
- Ajustou a intensidade de verde do pixel para o resultado dos dois passos anteriores.

Então, em português, poderíamos descrever o objetivo deste exercício simplesmente como: tornar a imagem um pouco mais laranja.

Reflita sobre como esse objetivo foi traduzido do português para o código de computador que usamos ao realizarmos a operação. A capacidade de fazer essa tradução e escrever uma solução que computadores são capazes de executar rapidamente é uma habilidade chave para o chamado **pensamento computacional**.



Voltando à imagem da figura 4, suponha, agora, que desejamos convertê-la em uma imagem de escala de cinza.

Recorde, na tabela 5, que uma imagem em escala de cinza possui, para cada pixel, exatamente o mesmo valor para os componentes RGB (vermelho, verde e azul):

Esquema RGB	R - Vermelho	G - Verde	B - Azul
Branco	255	255	255
Azul	0	0	255
Vermelho	255	0	0
Verde	0	255	0
Amarelo	255	255	0
Magenta	255	0	255
Ciano	0	255	255
Preto	0	0	0
<b>Escala de cinza</b>	<b>(1,1,1) cinza muito escuro (quase preto)</b>		
	<b>(2,2,2)</b>		
	<b>(3,3,3)</b>		
	<b>(4,4,4)</b>		
	<b>(5,5,5)</b>		

	(6,6,6)
	.
	.
	.
	(252,252,252)
	(253,253,253)
	(254,254,254) cinza muito claro (quase branco)

Tabela 5: Escala de cinza do esquema de cores RGB.

Guilherme Dutra Gonzaga Jaime.

Use os controles deslizantes do visualizador RGB para comprovar este conceito:

Sem cor →→ Escuro →→ Mais claro →→ Saturação total

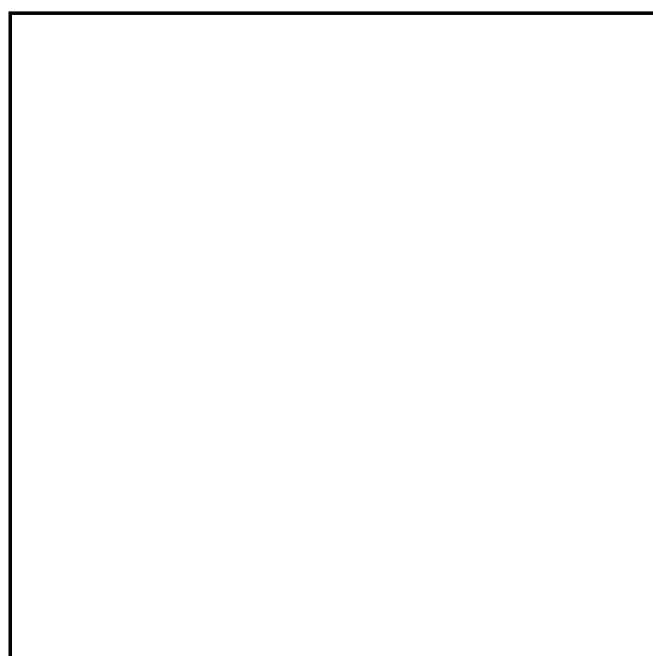
Vermelho (R - Red):

Verde (G - Green):

Azul (B - Blue):

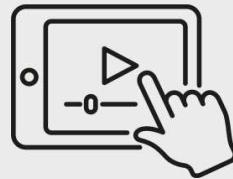
R:0 G:0 B:0

Mostrar Hexadecimal #000000



# Transformar a imagem para a escala de cinza

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Para alcançar o objetivo de converter a imagem de flores amarelas em uma imagem em escala de cinza, vamos calcular, para cada pixel, o valor médio dos componentes RGB. Depois, ajustaremos cada um dos três componentes para esse valor médio. Veja o trecho de código que calcula a média dos três componentes (R, G e B) para um dado pixel, depois, atribui esse valor médio aos três componentes de cor do pixel:

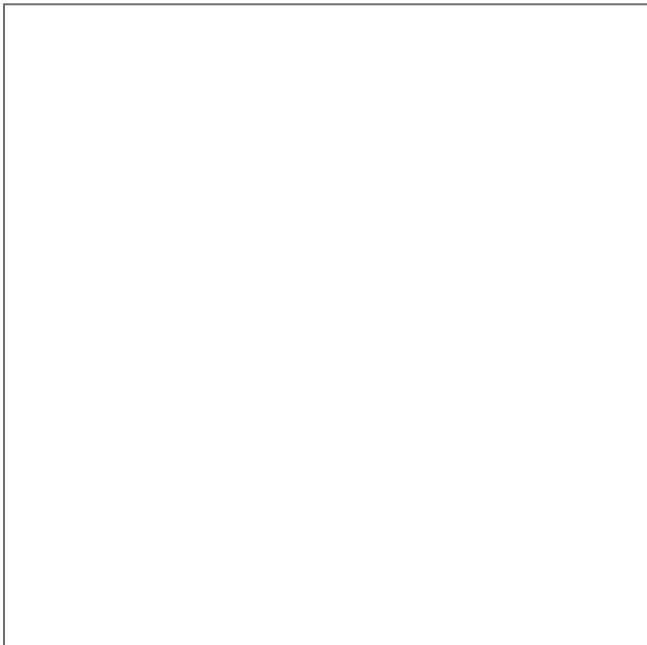
JavaScript



Copie e cole estas linhas de código no código-fonte de experimento de programação disponível no início desta prática:

**Código-Fonte**

```
img = new SimpleImage("img/flores.jpg");
for( pixel: img ){
    // Insira suas linhas de código abaixo
}
print( img );
```

**Rodar/Executar****Saída**

Como desejamos tornar todos os bits da imagem em escala de cinza, você deverá colar este trecho dentro da estrutura de repetição **for**. Depois, clique em **Executar**. Observe o computador executar suas instruções e transformar a imagem para a escala de cinza:

**Mostrar solução****Código-Fonte**

```
img = new SimpleImage("img/flores.jpg");
for( pixel: img ){
    // Insira suas linhas de código abaixo
    soma= pixel.getGreen() + pixel.getGreen() + pixel.getGreen();
    media= soma/3;
    pixel.setRed( media );
    pixel.setGreen( media );
    pixel.setBlue( media );
}
print( img );
```

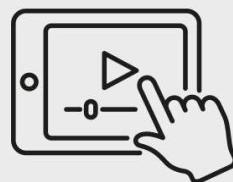
**Rodar/Executar****Saída**

Fique à vontade para experimentar com o código-fonte, obter diferentes resultados e refletir sobre o resultado obtido com as instruções que você testou. Por exemplo, você pode multiplicar a média por 1.1, 1.5 ou 2 e observar como a imagem vai ficando mais clara. Não tenha medo de errar!

## Prática 2

Assista ao vídeo com uma segunda prática.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



## Atividade 1

Sabendo que o uso de expressões permite calcular o valor de um parâmetro de maneira mais personalizada, qual seria a principal importância da manipulação de imagens usando pequenas expressões em JavaScript no desenvolvimento web?

- A Reduzir o tempo de carregamento das páginas web.
- B Permitir a criação de animações complexas sem a necessidade de bibliotecas externas.
- C Facilitar a manipulação dinâmica de imagens diretamente no navegador.
- D Eliminar a necessidade de utilizar imagens de alta resolução.

- E Melhorar a segurança das imagens carregadas no servidor.

Parabéns! A alternativa C está correta.

A principal importância da manipulação de imagens usando pequenas expressões em JavaScript é a capacidade de manipular dinamicamente as imagens diretamente no navegador. Isso permite aos desenvolvedores criarem aplicações interativas e responsivas sem a necessidade de realizar operações no lado do servidor, oferecendo uma experiência de usuário mais fluida e eficiente.

## Utilizando expressões na manipulação de imagens

Como você deve ter percebido, é muito importante entender que estamos o tempo todo falando de comunicação. Claro que, tratando-se da máquina, não podemos nos comunicar com ela da mesma forma que eu me comunico com você.

## Prática 3

Como você viu, o simples comando “torne a imagem um pouco mais laranja” não poderia ser realizado apenas com estas palavras. Foi necessário traduzir essa mensagem para código de computador, de forma que o comando fosse obedecido.

**Nesse sentido, apenas com a realização de tentativas, com a identificação de erros nos comandos e sua correção, é que podemos nos aprimorar e, pouco a pouco, dar embasamento ao nosso pensamento computacional.**

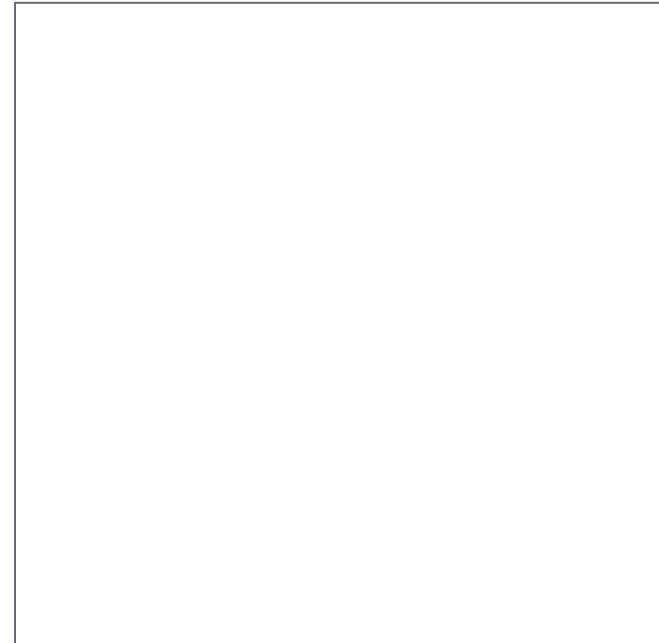
Você deverá usar as caixas de códigos para realizar as atividades do Verificando o aprendizado apresentadas a seguir. A primeira caixa de código será utilizada para a primeira atividade e a segunda caixa de código para a segunda atividade.

Clique em Executar para ordenar ao computador que execute as instruções escritas por você de acordo com o enunciado de cada atividade.

**Código-Fonte**

```
img = new SimpleImage("img/pag-39-2.jpg");
for( pixel: img ){
    // Insira suas linhas de código abaixo

}
print( img );
```

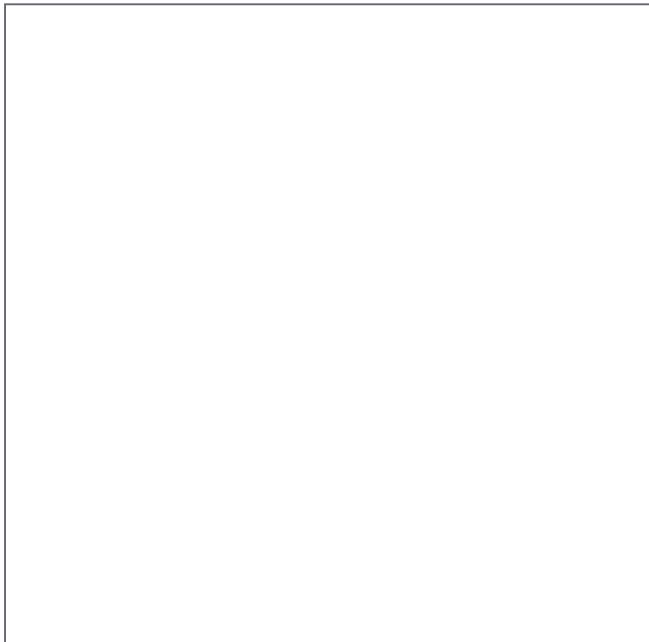
**Rodar/Executar****Saída****Código-Fonte**

```
img = new SimpleImage("img/pag-41.jpg");
for( pixel: img ){
    // Insira suas linhas de código abaixo

}
print( img );
```

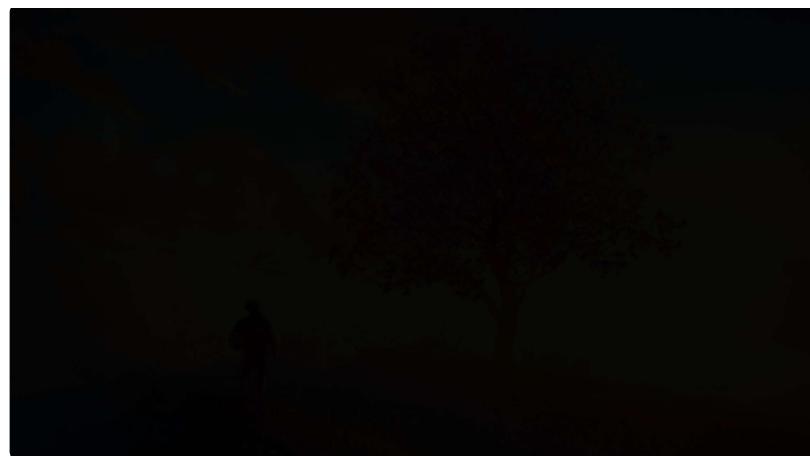
Rodar/Executar

Saída



## Atividade 2

A figura a seguir está muito escura:



Desejamos clareá-la, multiplicando os componentes de cores de cada pixel por 30. Clique aqui e escreva, no código-fonte, dentro da estrutura de repetição `for`, as linhas de código que instruirão o computador a realizar essa tarefa:

Quando você tiver realizado isso com sucesso, será possível observar o que há na imagem. Assinale a alternativa que descreve o que apareceu na imagem:

A Um submarino cercado de tubarões.

B Um homem observando uma árvore.

C Um raio caindo em uma árvore.

D Um castelo cercado de muralhas.

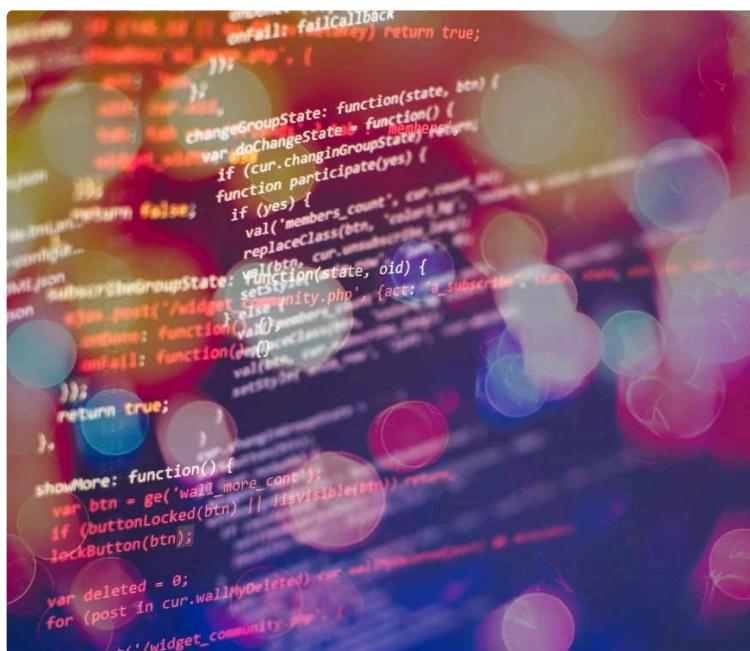
E Um homem em cima de um cavalo.

Parabéns! A alternativa B está correta.

O código-fonte a seguir descreve, em código de computador, o passo a passo necessário para clarear a imagem, conforme solicitado no enunciado, ou seja, para cada pixel da imagem, multiplicando os componentes de cores por 30:

```
img = new SimpleImage("img/muitoescura.png");
for( pixel: img ){
    // Insira suas linhas de código abaixo
    pixel.setRed(pixel.getRed()*30);
    pixel.setGreen(pixel.getGreen()*30);
    pixel.setBlue(pixel.getBlue()*30);
}
print( img );
```

Após o clareamento é possível identificar que a imagem mostra um homem observando uma árvore. Basicamente, estamos usando expressões para resolver a questão.



#### 4 - A estrutura condicional if

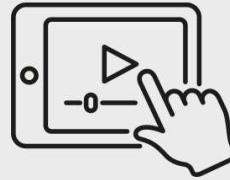
Ao final deste módulo, você será capaz de distinguir a estrutura condicional if.

## O que são estruturas condicionais?

A estrutura condicional if é fundamental em programação, permitindo que o código execute ações diferentes com base em condições específicas. Essa estrutura permite a tomada de decisões no código, possibilitando a implementação de lógica complexa e adaptativa. O domínio das condicionais if é importante para criar programas eficientes e dinâmicos, capazes de responder de forma adequada a diferentes situações e entradas de dados.

Neste vídeo, vamos explorar o conceito de estruturas condicionais, também chamadas de estruturas de decisão, e como elas são aplicadas

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Estruturas condicionais são as linhas de código de computador empregadas para expressar a ideia de lógica ou seletividade.

Usando esses tipos de instruções, somos capazes de preparar o computador para realizar um teste, cujo resultado será verdadeiro ou falso, aplicando-o para controlar se determinado pedaço de código será ou não executado. Todas as linguagens de programação possuem estruturas condicionais.



Vamos analisar aqui um tipo específico: a **declaração if** (do inglês, if-statement). Lembre-se de que if, em inglês, significa “se”.

Vamos nos basear em exemplos e práticas na linguagem Python, estando sempre voltados para a resolução de problemas interessantes relacionados à manipulação de imagens.

## Estrutura condicional

Veja a sintaxe de uma declaração if conforme a linguagem de programação Python:

Python



Repare na linha `if intensidade > limiar`. Em português, essa instrução significa:

**Se a intensidade média das cores (vermelho, verde e azul) do pixel for maior do que o valor definido pela variável `limiar`, então execute a instrução que colore o pixel de vermelho (255, 0, 0).**

Em outras palavras, as instruções das linhas 8 e 10 só serão executadas pelo computador dependendo do resultado desse teste. Pense nessa instrução como uma forma de selecionar os pixels da imagem "pássaro.jpg" que serão alterados para vermelho ou azul, conforme a intensidade medida. Veja!

1

Ainda na linha 7, se o valor da intensidade média do pixel em questão for, por exemplo, 100 e o `limiar` for 128, então o teste `if intensidade > limiar` retornará **falso**, e o computador irá para a linha 10, pintando o pixel de azul (0, 0, 255).

2

Se a intensidade do pixel for exatamente igual ao valor do `limiar` (por exemplo, 128), o teste `if intensidade > limiar` também será **falso**, e novamente o pixel será colorido de azul.

3

A execução da linha 8, que define o pixel como vermelho, só ocorrerá se a intensidade for **maior** que o limiar, por exemplo, 129, 200 ou 255.

4

Em Python, diferentemente do JavaScript, o teste dentro do if não precisa obrigatoriamente de parênteses, mas o uso deles é permitido para melhorar a clareza. O importante é que o computador entenda que se trata de uma decisão: se a condição for verdadeira, ele executa o que está recuado (indentado) logo abaixo do if; caso contrário, ele passa para a parte do else.

Na prática, frequentemente vemos uma declaração if ser usada dentro de uma estrutura de repetição, como o for duplo que percorre todos os pixels da imagem. Esse tipo de combinação permite que problemas bastante interessantes sejam resolvidos.

### Comentário

Na prática, frequentemente vemos uma declaração if ser usada dentro de uma estrutura de repetição, como a estrutura for. Esse tipo de combinação permite que problemas bastante interessantes sejam resolvidos.

A partir de agora experimentaremos a estrutura condicional if. Abordaremos várias práticas simples que envolvem manipulação de imagens digitais, começando por casos bem rudimentares e chegando a aplicações do mundo bem interessantes, ainda que simples.

Optamos por usar exemplos com imagens digitais, pois esta é uma forma muito simples, intuitiva e rápida para você observar os efeitos da execução de algumas linhas de código de computador. Isso ocorre porque estamos acostumados a observar imagens do mundo real desde que nascemos. Então, esse é um processo altamente intuitivo para todos.

**Para analisar e avaliar o efeito de determinadas linhas de código, basta observar a imagem e comparar o**

antes e o depois da execução do código.

## Atividade 1

A estrutura if é essencial em programação, pois permite que sejam escritos programas que consigam lidar de forma correta com entradas de dados diferentes. Qual das seguintes opções descreve corretamente como a estrutura condicional if pode ser utilizada para melhorar a lógica de um programa?

- A A estrutura if é usada para repetir blocos de código várias vezes até que uma condição seja satisfeita.
- B A estrutura if permite que o programa execute um bloco de código específico apenas se uma condição definida for verdadeira.
- C A estrutura if define um conjunto de regras estáticas que nunca mudam durante a execução do programa.
- D A estrutura if é usada apenas para verificar a igualdade entre duas variáveis.
- E A estrutura if é essencialmente um loop que continua sendo executado indefinidamente.

Parabéns! A alternativa B está correta.

A opção B é a correta porque descreve exatamente como a estrutura condicional if funciona em programação. A estrutura if avalia uma condição e, se essa condição for verdadeira, executa o bloco de código associado. Isso é fundamental para a lógica do programa, permitindo que ele se comporte de maneira diferente, com base em diferentes entradas ou estados. As outras opções são incorretas porque

confundem if com estruturas de repetição ou limitam sua funcionalidade a verificações de igualdade, o que não abrange todas as possibilidades do uso de if.

## Estruturas condicionais na prática

Agora chegou a hora de aplicarmos o uso de estruturas condicionais na prática, sempre no contexto de manipulação de imagens digitais.

### Prática 1

Considere a seguinte imagem:

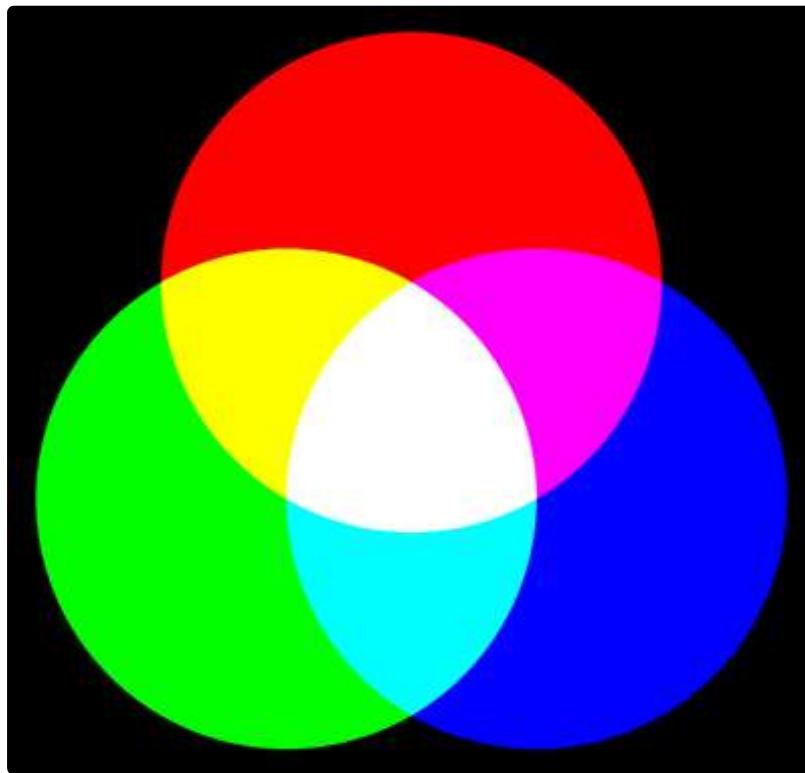


Figura 7: Imagem representativa do esquema de Cores RGB.

---

Imagine que desejamos escrever um código de computador que altere somente a região em vermelho, transformando-a em cinza.

Para entender como é possível escrever um código de computador a fim de alterar a região em vermelho, considere a tabela 7, que indica como cada cor é representada segundo o esquema RGB:

Esquema RGB	R - Vermelho	G - Verde	B - Azul
Branco	255	255	255
Azul	0	0	255
Vermelho	255	0	0
Verde	0	255	0
Amarelo	255	255	0
Magenta	255	0	255
Ciano	0	255	255
Preto	0	0	0
<b>Escala de cinza</b>	<b>(1,1,1) cinza muito escuro (quase preto)</b>		
	<b>(2,2,2)</b>		
	<b>(3,3,3)</b>		
	<b>(4,4,4)</b>		
	<b>(5,5,5)</b>		
	<b>(6,6,6)</b>		
	•		
	•		
	•		
	<b>(252,252,252)</b>		
	<b>(253,253,253)</b>		
	<b>(254,254,254) cinza muito claro (quase branco)</b>		

Tabela 7: Esquema RGB – Exemplos de cores comuns.

Note que o vermelho equivale ao código RGB(255,0,0). Uma primeira tentativa para transformar a região vermelha em cinza seria pensar nesse código RGB.

Então, podemos escrever um código que ordena ao computador que, para cada pixel da imagem, verifique se o componente vermelho é 255. Em caso positivo, ajuste a cor do pixel para cinza, ou seja, RGB(120,120,120).

## Vamos experimentar?

No código-fonte à esquerda estão listadas as linhas de código que representam, em código de computador, o passo a passo da tentativa do parágrafo anterior. Clique em Rodar/Executar e observe o resultado.

### Código-Fonte

```
img = new SimpleImage("img/RGB.png");
for( pixel: img ){

    if( pixel.getRed() == 255 ){
        pixel.setRed( 120 );
        pixel.setGreen( 120 );
        pixel.setBlue( 120 );
    }

}
print( img );
```

Rodar/Executar

### Saída

Veja a solução a seguir:

Mostrar solução

Código-Fonte

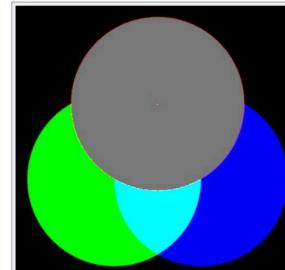
```
img = new SimpleImage("img/RGB.png");
for( pixel: img ){

    if( pixel.getRed() == 255 ){
        pixel.setRed( 120 );
        pixel.setGreen( 120 );
        pixel.setBlue( 120 );
    }

}
print( img );
```

Radar/Executar

Saída



Repare que não só a região vermelha foi transformada em cinza, mas também as regiões amarelo, branco e magenta. Colorimos mais do que a região proposta pelo exercício. Observe a tabela 7. Você notará que as cores amarelo, branco e magenta também possuem, assim como o vermelho, o valor 255 para o componente R.

Então, nossa instrução *if* ao computador precisa ser mais específica.

### Relembrando

O computador faz exatamente o que ordenamos. Na prática, precisamos checar três tarefas: para que apenas a região vermelha seja ajustada para cinza, devemos verificar se o Red é igual a 255, se o Green é igual a 0 (zero), e se o Blue é igual a 0 (zero).

Então, você precisará substituir no código-fonte toda a linha da instrução *if* pela instrução a seguir:

JavaScript



Você pode usar o recurso de copiar e colar para isso:

### Código-Fonte

```
img = new SimpleImage("img/RGB.png");
for( pixel: img ){

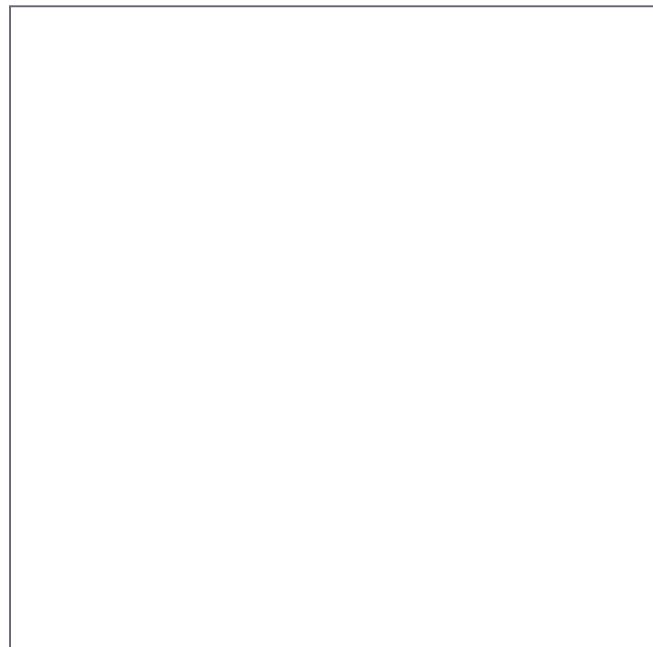
    if( pixel.getRed() == 255 ){
        pixel.setRed( 120 );
        pixel.setGreen( 120 );
        pixel.setBlue( 120 );
    }

}

print( img );
```

Rodar/Executar

Saída



Em seguida, clique em Rodar/Executar e observe que o objetivo desta prática será alcançado.

Mostrar solução

Código-Fonte

```
img = new SimpleImage("img/RGB.png");
for( pixel: img ){

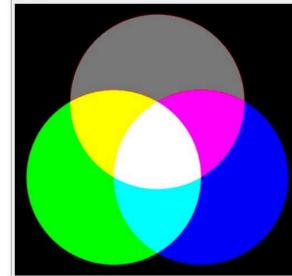
    if( pixel.getRed() == 255 && pixel.getGreen() == 0 && pixel.getBlue() == 0 ){
        pixel.setRed( 120 );
        pixel.setGreen( 120 );
        pixel.setBlue( 120 );
    }

}

print( img );
```

Rodar/Executar

Saída



Observe que, com essa ação, você programou para que a região somente vermelha fosse ajustada para cinza, mas, ao mesmo tempo, permitiu que as regiões de interseção entre as três cores se apresentassem sem alteração.

## Prática 2

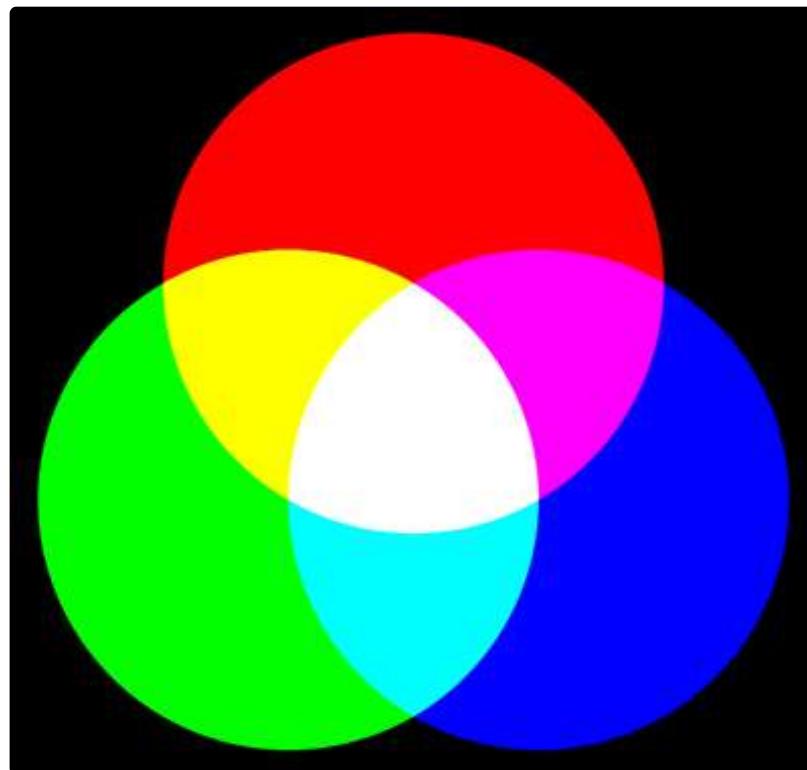
Preparamos um vídeo abordando outra prática.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



## Atividade 2

Na Prática 1, usamos como ponto de partida a imagem a seguir para realizar alguns experimentos:



Suponha que desejemos escrever uma linha de instrução *if* que selecione os pixels da região em azul para, então, torná-los amarelos.

Assinale a alternativa que contém o código correto para alcançar essa tarefa.

A  
if( pixel.getRed()==0 &&pixel.getGreen()==0  
&&pixel.getBlue()==255 ){  
pixel.setRed(255);  
pixel.setGreen(255);  
pixel.setBlue(0);  
}

B  
if( pixel.getRed()>0 &&pixel.getGreen()>0  
&&pixel.getBlue()==255 ){  
pixel.setRed(255);  
pixel.setGreen(255);  
pixel.setBlue(0);  
}

C  
if( pixel.getRed()==0 &&pixel.getGreen()==0  
&&pixel.getBlue()==255 ){  
pixel.setRed(0);  
pixel.setGreen(255);  
pixel.setBlue(255);  
}

D  
if( pixel.getRed()>0 &&pixel.getGreen()>0  
&&pixel.getBlue()==255 ){  
pixel.setRed(0);  
pixel.setGreen(255);  
pixel.setBlue(255);  
}

E  
if( pixel.getRed()>0 &&pixel.getGreen()>0  
&&pixel.getBlue()==0 ){  
pixel.setRed(0);  
pixel.setGreen(0);  
pixel.setBlue(0);  
}

Parabéns! A alternativa A está correta.

Conforme estudamos, precisamos usar a estrutura condicional *if* para realizar testes e, se o resultado for verdadeiro, executar algumas instruções.

No caso desta atividade, os pixels da região em azul são os pixels com código RGB(0,0,255). Então, para selecionar corretamente estes pixels, usamos a instrução *if*, conforme observamos a seguir:

```
if( pixel.getRed()==0 &&pixel.getGreen()==0 &&pixel.getBlue()==255 ){  
}
```

Depois, precisamos adicionar, dentro da estrutura *if*, instruções que ajustarão a cor dos pixels selecionados para amarelo. O código RGB do amarelo puro é (255,255,0). Então, o código completo ficaria conforme a alternativa A.

Veja a captura a seguir com o código-fonte e o resultado obtido:

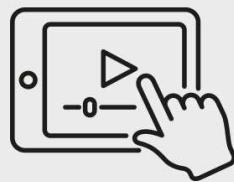
## Continuando a prática com estruturas condicionais

Que tal praticar o uso de estruturas condicionais na manipulação de imagens um pouco mais? As práticas a seguir solidificarão os conceitos aprendidos sobre estruturas condicionais.



Que tal praticar um pouco?  
Selecionando os pixels do meio-fio!

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Observe a seguinte imagem:



Figura 8: Uma calçada com o meio-fio pintado de amarelo.

Imagine que desejamos ajustar a imagem para que o meio-fio fique cinza em vez de amarelo.

Pela tabela 7 sabemos que o amarelo é uma cor composta pela combinação de vermelho e verde.

Vamos utilizar o código-fonte a seguir para experimentar com diferentes abordagens de construção para a instrução *if* até alcançarmos uma solução mais refinada que nos pareça suficientemente boa.

#### Código-Fonte

```
img = new SimpleImage("img/calçada.jpg");
for( pixel: img ){
    if( pixel.getRed() > 120 && pixel.getGreen() > 120 ){
        pixel.setRed( 0 );
        pixel.setGreen( 0 );
        pixel.setBlue( 0 );
    }
}
print( img );
```

Rodar/Executar

#### Saída

No código que já consta no código-fonte, o *if* está testando se os níveis de verde e de vermelho de cada pixel são maiores do que 120. Isso reflete a ideia de que um bom nível de verde e vermelho indica que o pixel **pode** ser amarelo.

**Se o teste do *if* for positivo, o pixel que atende à condição será para a cor preta: RGB (0,0,0). Isso será interessante, pois, ao analisarmos a imagem resultante, basta observarmos o que estiver em preto. Estes terão sido os pixels selecionados pela instrução *if*.**

Clique em **Executar** e observe o resultado:

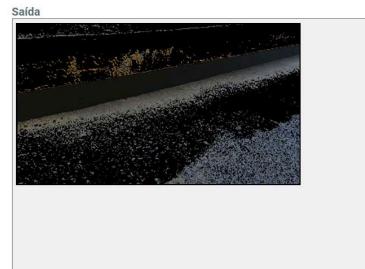
Mostrar solução

Código-Fonte

```
img = new SimpleImage("img/calçada.jpg");
for( pixel: img ){
    if( pixel.getRed() > 120 && pixel.getGreen() > 120 ){
        pixel.setRed( 0 );
        pixel.setGreen( 0 );
        pixel.setBlue( 0 );
    }
}
print( img );
```

Rodar/Executar

Saída

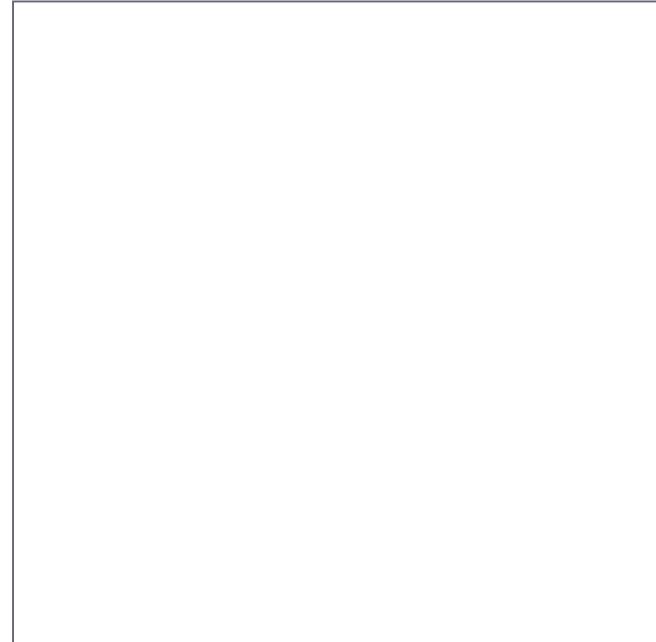


Repare que boa parte do meio-fio amarelo foi pintado de preto. Porém, analisando melhor, podemos detectar um problema: parte importante da calçada e do asfalto foram pintados de preto. Isso ocorre pois essas são áreas da imagem original em que os pixels são mais iluminados/claros.

Então, os três componentes, vermelho, verde e azul, devem estar acima de 120, gerando um tom cinza suficientemente claro. Instrua o computador a selecionar apenas pixels cujos componentes vermelho e verde estejam acima de 150.

**Código-Fonte**

```
img = new SimpleImage("img/calcada.jpg");
for( pixel: img ){
    if( pixel.getRed() > 120 && pixel.getGreen() > 120 ){
        pixel.setRed( 0 );
        pixel.setGreen( 0 );
        pixel.setBlue( 0 );
    }
}
print( img );
```

**Rodar/Executar****Saída**

Em seguida, clique em **Executar** e observe o resultado:

**Mostrar solução****Código-Fonte**

```
img = new SimpleImage("img/calcada.jpg");
for( pixel: img ){
    if( pixel.getRed() > 120 && pixel.getGreen() > 120 ){
        pixel.setRed( 0 );
        pixel.setGreen( 0 );
        pixel.setBlue( 0 );
    }
}
print( img );
```

**Rodar/Executar****Saída**

Observe que ainda falta nitidez nas cores, como se a tinta do meio-fio estivesse se espalhado também para a calçada. É necessário mais ajustes.

## Vamos praticar mais um pouco?

Ajuste a estrutura *if*, a fim de selecionar apenas pixels ainda mais claros e em tom de amarelo. Clique em **Experimentar** e observe o resultado:

Código-Fonte

```
img = new SimpleImage("img/calçada.jpg");
for pixel: img {
    if( pixel.getRed() > 150 && pixel.getGreen() > 150 ) {
        pixel.setRed( 0 );
        pixel.setGreen( 0 );
        pixel.setBlue( 0 );
    }
}
print( img );
```

Saída

## A seleção parece melhor, certo?

Temos menos quantidade pintada de preto na calçada. Porém outro problema tornou-se evidente: os pixels da lateral do meio-fio não foram selecionados pelo *if*. Isso ocorre porque seu tom de amarelo é mais escuro. Certamente, seu nível de luminosidade é menor do que 150. Portanto, esses pixels não passam no teste do *if*.

No último passo desta prática, tentaremos pensar em uma solução mais inteligente para a instrução *if*, de forma que consigamos, também, selecionar os pixels da lateral para realizar o ajuste de cores. Então, vamos construir uma estratégia melhor. Para começar, observe nosso “laboratório RGB” com controles deslizantes para os componentes vermelho, verde e azul:

Sem cor →→ Escuro →→ Mais claro →→ Saturação total

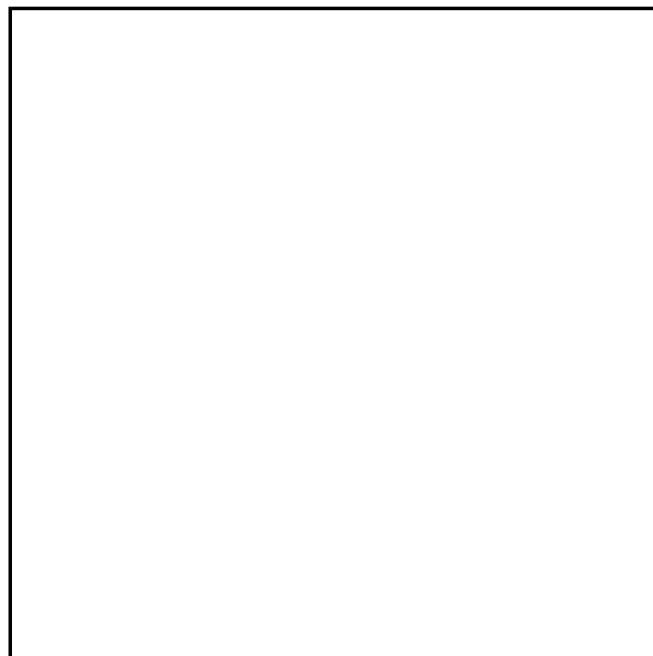
Vermelho (R - Red):

Verde (G - Green):

Azul (B - Blue):

R:0 G:0 B:0

Mostrar Hexadecimal #000000



Repare que você obteve um amarelo-escuro.

Agora, deslize o controle azul de 0(zero) para 84. Se tiver dificuldade em obter o valor exato, use as setas direcionais direita/esquerda no teclado.

Observe o resultado:



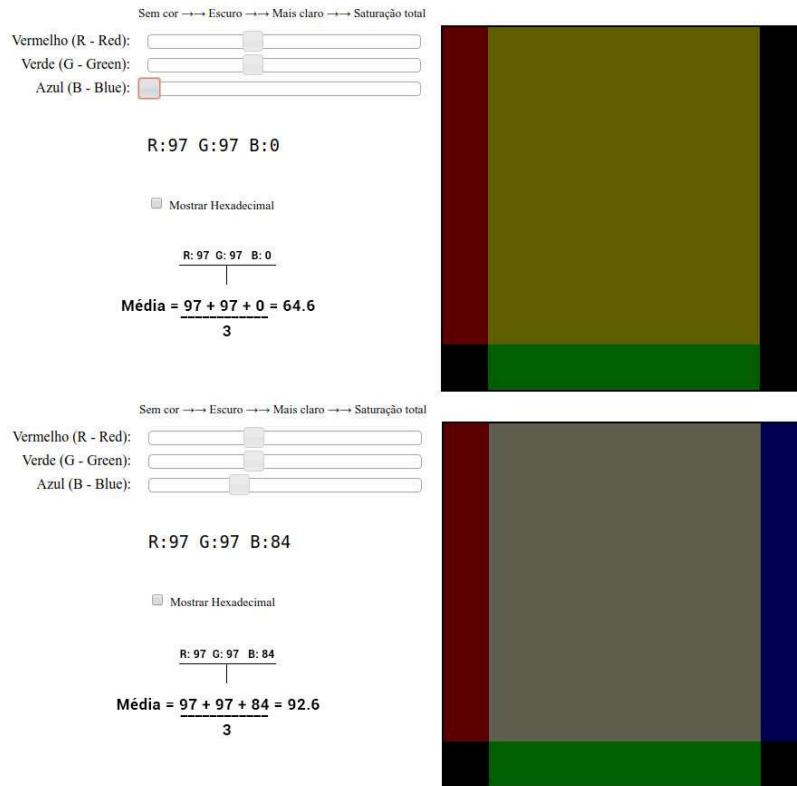
Observe o amarelo gradativamente aproximando-se do cinza.

## Considerações

Queremos tirar proveito do padrão da natureza que acabamos de observar.

Tons de amarelo são caracterizados por níveis equivalentes de vermelho e verde e um nível de azul muito mais baixo. Como, porém, podemos escrever uma estrutura if para instruir o computador a selecionar os pixels que atendam a esse padrão?

Felizmente, isso é muito simples! Observe as duas figuras a seguir:

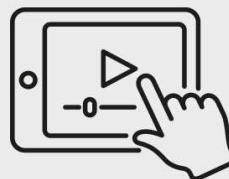


Na primeira figura, se calcularmos a média dos três componentes R(97), G(97) e B(0), obteremos o valor 64.6. A linha tracejada representa onde estaria esse valor nos controles deslizantes. Repare que os componentes vermelho e verde estão acima da média. O mesmo comportamento se repete na segunda figura.



**Que tal praticar um pouco? Alterando a cor do meio-fio para cinza!**

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Então, vamos usar essa observação para construir uma solução mais inteligente para a instrução *if* e observar se, desse modo, selecionamos os pixels em amarelo do meio-fio de forma mais satisfatória. Volte ao código-fonte desta prática e realize os ajustes destacados em sublinhado a seguir:

JavaScript



```
img = new SimpleImage("calcada.jpg");
```

**Código-Fonte**

```
img = new SimpleImage("img/calcada.jpg");
for( pixel: img ){
    if( pixel.getRed() > 120 && pixel.getGreen() > 120 ){
        pixel.setRed( 0 );
        pixel.setGreen( 0 );
        pixel.setBlue( 0 );
    }
}
print( img );
```

**Rodar/Executar****Saída**

Adicionaremos uma linha para o computador calcular a média automaticamente e, na estrutura *if*, testaremos se o valor do vermelho e do verde são maiores do que a média. Depois, clique em **Executar** e observe o resultado:

**Mostrar solução**

Código-Fonte

```
img = new SimpleImage("img/calçada.jpg");
for( pixel: img ){
    media( pixel.getRed() +pixel.getGreen() +pixel.getBlue() ) / 3;
    if( pixel.getRed() >media&&pixel.getGreen() >media ){
        pixel.setRed( 0 );
        pixel.setGreen( 0 );
        pixel.setBlue( 0 );
    }
}
print( img );
```

Rodar/Executar

Saída

Agora, confirmamos que a nossa estrutura *if* é capaz de selecionar, de forma bem satisfatória, os pixels que desejamos ajustar para a tonalidade cinza. Já pensamos em uma forma de escrever a instrução *if*, de modo que os pixels do meio-fio sejam corretamente selecionados para modificação de cor. O último passo para alcançarmos nosso objetivo é ajustar as instruções internas à estrutura *if*, em que, de fato, as cores dos pixels selecionados são ajustadas.

Durante os passos anteriores, deixamos as instruções alterando o valor dos componentes RGB para 0 (zero), para que pudéssemos observar o que ficou em preto e refletir se a nossa estrutura *if* estava correta.

**Agora, pense um pouco: queremos que o meio-fio, que era amarelo, apareça em escala de cinza.**

Pela tabela 7 já sabemos que a escala de cinza equivale a dizer que os três componentes R, G e B possuem o mesmo valor. Ora, nós já ordenamos que o computador calcule automaticamente a média dos três componentes do pixel.

Então, a variável chamada *media* contém uma estimativa de quanta luminosidade existe no pixel sendo tratado. Uma boa tentativa seria trocar o 0 (zero) das instruções de ajuste de cor internas ao *if* pela variável *media*. Realize esses ajustes no código-fonte, conforme destacado em sublinhado:

**Código-Fonte**

```
img = new SimpleImage("img/calcada.jpg");
for( pixel: img ){
    if( pixel.getRed() > 120 && pixel.getGreen() > 120 ){
        pixel.setRed( 0 );
        pixel.setGreen( 0 );
        pixel.setBlue( 0 );
    }
}
print( img );
```

**Rodar/Executar****Saída**

Observe o resultado:

Mostrar solução



Código-Fonte

```
img = new SimpleImage("img/calcada.jpg");
for( pixel: img ){
    media=( pixel.getRed() + pixel.getGreen() + pixel.getBlue() ) / 3;
    if( pixel.getRed() > media && pixel.getGreen() > media ){
        pixel.setRed( media );
        pixel.setGreen( media );
        pixel.setBlue( media );
    }
}
print( img );
```

Saída

Rodar/Executar

Isso realmente parece bem melhor! Conseguimos alcançar o nosso objetivo.

Escrevemos um código de computador que é capaz de instruí-lo a detectar automaticamente a região em amarelo do meio-fio, para, então, ajustar sua tonalidade de cor de amarelo para cinza.

## Atividade 3

Analise a instrução *if* a seguir:

```
if( pixel.getRed() ==162&&pixel. getGreen() ==162 &&pixel.getBlue() > 200 ){}
```

Assinale a alternativa que informa as características dos pixels que serão “selecionados” por este *if*. Se necessário, clique aqui e use os controles deslizantes do simulador RGB para chegar a sua resposta.

A Pixels em tonalidades de azul.

B Pixels em tonalidades de amarelo.

C Pixels em tonalidades de verde.

D Pixels em tonalidades de vermelho.

E Pixels em tonalidade laranja.

Parabéns! A alternativa A está correta.

O *if* em questão seleciona pixels que passem em três testes:

- Nível de vermelho igual a 162;
- Nível de verde igual a 162;
- Nível de azul maior do que 200.

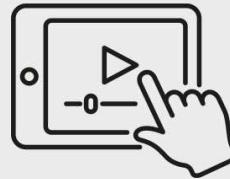
Conforme ilustrado pelo exemplo a seguir, obtido por meio do simulador RGB disponibilizado para esta atividade, o teste em questão equivale a escolher pixels em tonalidades de azul.

## Pensamento computacional

É uma habilidade essencial que será indispensável nas próximas décadas, similar à leitura, à escrita e à aritmética. Essa competência é indispensável para o mercado de trabalho, abrangendo diversas áreas, como educação, ciência, engenharia, medicina e direito.

O estudo do pensamento computacional envolve entender como os computadores representam conceitos do mundo real através de números, especificamente para imagens, utilizando a posição e cor de pixels. A abstração, que é a habilidade de focar os elementos essenciais de uma imagem, é a base desse pensamento, permitindo a criação de códigos para manipular e ajustar imagens automaticamente. Assista ao vídeo e confira!

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



O pensamento computacional já é visto em vários países como uma habilidade essencial para todos nas próximas décadas. Assim como a leitura, a escrita e a aritmética, essa competência será indispensável para o mercado de trabalho.

Professores e alunos do ensino fundamental, ensino médio e ensino superior, além de cientistas, engenheiros, historiadores, artistas, médicos, advogados, todos precisarão usá-la para desempenhar seu papel de forma competitiva. Esse estudo está baseado justamente no pensamento computacional.



De início, compreendemos que o computador representa qualquer conceito do mundo real por meio de números. No caso de imagens, os números indicam a posição do pixel e sua cor.

Para isso, precisamos desconsiderar qualquer outro detalhe da imagem que não seja fundamental para sua representação:

- Quantos pixels usaremos para representá-la.
- Qual é a cor de cada pixel.

Esse tipo de habilidade é denominada abstração e constitui um dos pilares primordiais do pensamento computacional.

Em seguida, escrevemos linhas de códigos (instruções) de computador para que ele fosse capaz de automaticamente manipular esses números e, assim, realizarmos ajustes nas imagens, conforme o objetivo proposto. Dessa forma, praticamos outra habilidade chave do chamado pensamento computacional, que é a **automação**, ou seja, escrever

soluções na forma de instruções que o computador é capaz de seguir automaticamente para chegar ao resultado desejado.

Durante o desenvolvimento das práticas, sempre paramos para observar e analisar os resultados obtidos nas imagens apresentadas. Refletimos sobre como poderíamos melhorar os resultados alcançados e realizamos ajustes em nosso código para alcançar resultados mais adequados. Isso constitui outra base do pensamento computacional denominada **análise/avaliação**.

Observe que, durante as práticas realizadas, o código-fonte que escrevemos sempre decompõe o problema proposto em partes mais simples, que seguiram este passo a passo padrão:

#### Passo 1

Carregar a imagem na memória para que fosse trabalhada – escrevemos uma linha de código que carrega a imagem.

#### Passo 2

Automaticamente processar, um a um, todos os pixels (centenas de milhares) de uma imagem – usamos a estrutura for para instruir o computador a repetir as instruções para cada pixel da imagem.

#### Passo 3

Selecionar em quais pixels desejamos realizar ajuste de cores – escrevemos a estrutura if para que o computador realizasse testes e, conforme o resultado (verdadeiro ou falso), executasse ou não as instruções de manipulação de cores do pixel.

#### Passo 4

Alterar a cor de um pixel – escrevemos linhas de código para alterar os componentes RGB (vermelho, verde e azul) para alterar a cor de um pixel, conforme nosso objetivo.

#### Passo 5

Imprimir o resultado na tela – escrevemos a função print para que o resultado das instruções (a imagem manipulada) fosse apresentado na tela.

A habilidade de decompor um problema em problemas bem menores e mais simples, que podem ser resolvidos isoladamente com uma ou poucas linhas de código de computador, é mais um dos pilares do pensamento computacional, denominado **decomposição**.

Além disso, durante a prática do ajuste de cor do meio-fio de amarelo para escala de cinza, foi preciso observar um padrão natural/orgânico de qualquer pixel de tonalidades amareladas. Eles possuem os componentes verde e vermelho maiores do que o componente azul.

**Aproveitamos essa observação para escrever um código que automaticamente selecionou pixels em vários tons de amarelo, permitindo ajustar suas cores. Essa habilidade é conhecida como detecção de padrões.**

A aprendizagem de todos esses pilares permitirá que você desenvolva cada vez mais saberes na área da computação.

## Atividade 4

O pensamento computacional é uma habilidade crucial e continuará sendo imprescindível no futuro. Em termos de importância, é similar à leitura, à escrita e aritmética, pois consiste em compreender como os computadores representam conceitos do mundo real através de números.

Especificamente em relação às imagens, essa representação numérica é feita utilizando a posição e a cor de seus pixels. Qual dos seguintes conceitos é um pilar fundamental do pensamento computacional que permite a criação de códigos para manipular e ajustar imagens automaticamente?

A Lógica formal

B Abstração

C Algoritmo

D Complexidade computacional

E Programação orientada a objetos

Parabéns! A alternativa B está correta.

A abstração é a habilidade de focar os elementos essenciais de um problema ou uma imagem, desconsiderando detalhes que não são fundamentais. No contexto do pensamento computacional, abstração permite representar conceitos do mundo real em termos de números que podem ser manipulados através de códigos. Esse conceito é importante para criar programas que ajustem imagens automaticamente, manipulando a posição e a cor dos pixels de maneira eficiente.

## O que você aprendeu neste conteúdo?

- Estruturas simples de programação para manipulações de imagens.
- Uso de estrutura de repetição para manipulação de imagens.
- Uso de estrutura condicional para manipulação de imagens.

## Explore +

Confira a indicação que separamos especialmente para você!

Pesquise o artigo **Proposta de atividades para o desenvolvimento do pensamento computacional no ensino fundamental**, de Daiane Andrade e outros autores. Campinas: Unicamp, 2013. p. 169-178.

## Referências

CARVALHO, A.; LORENA, A. **Introdução à computação: hardware, software e Dados**. Rio de Janeiro: LTC, 2017.

DALE, N.; LEWIS, J. **Ciência da Computação**. 4. ed. Rio de Janeiro: LTC, 2011.

FEDELI, R. D.; POLLONI, E. G. F.; PERES, F. E. **Introdução à Ciência da Computação**. 2. ed. São Paulo: Cengage, 2010.

FLANAGEN, D. **JavaScript: o guia definitivo**. 6. ed. Porto Alegre: Bookman, 2013.

GLENN, J. **Ciência da Computação: uma visão abrangente**. 11. ed. Porto Alegre: Bookman, 2013.

### Material para download

Clique no botão abaixo para fazer o download do conteúdo completo em formato PDF.

[Download material](#)

O que você achou do conteúdo?



Relatar problema