Aula Prática 7 - Cores e Imagens Binárias

Trabalho prévio

Como auxílio aos exercícios seguintes, será útil ter disponíveis funções que permitam:

1. Sortear um número inteiro entre 0 e um valor máximo (parâmetro).
2. Limitar um número a uma dado intervalo [min, max].

   Exemplos:

   constrain(n = -3, min = 0, max = 255) → 0

   constrain(n = 30, min = 0, max = 255) → 30

   constrain(n = 270, min = 0, max = 255) → 255
3. Sortear um valor booleano, sendo a probabilidade de sair *true* igual à de sair *false*.
4. Criar uma matriz de booleanos com valores aleatórios, dados o número de linhas e colunas.
5. Criar uma matriz de booleanos que corresponda às casas de um tabuleiro de xadrez (8x8), sendo as casas brancas representadas por *true* e as casas pretas por *false*.

   Dica: uma forma de determinar se uma casa é branca consiste em verificar se a soma entre o índice da linha e o índice da coluna é um número par. Como exemplos, as casas (0, 0), (0, 2), (1, 1), (1, 3) são brancas.
6. Calcular a distância entre dois pontos.

$$d_{AB} = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$

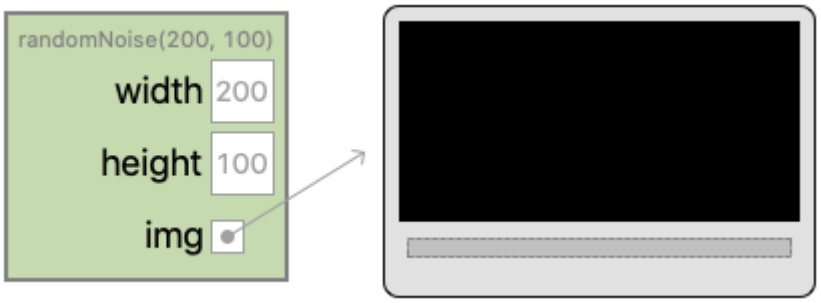
   Dica: a classe *Math*, tem funções para cálculo numérico, tais como:

   a. **sqrt(n)** para obter a raiz quadrada de um número
   b. **pow(base, exp)** para calcular potências

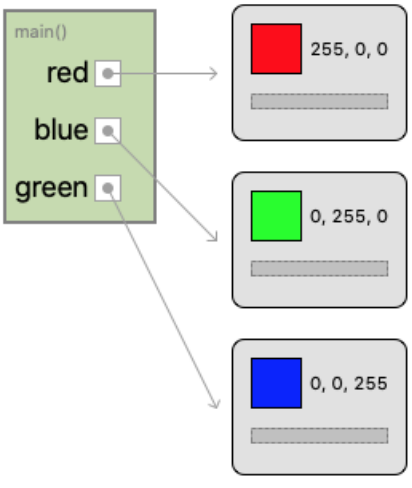
Visualizações no PandionJ

1. Para conseguir visualizar objetos específicos (cores, imagens, etc) no PandionJ, é necessário executar em modo *debug* (definindo necessariamente pelo menos um *breakpoint*).
2. Uma forma prática de realizar (1) sem ter que executar passo-a-passo, é definir um *breakpoint* no final de uma função/procedimento.

```
    static BinaryImage randomNoise(int width, int height) {
        BinaryImage img = new BinaryImage(width, height);

        return img;
    }
```



Nos casos onde não há retorno (*void*) podemos escrever uma última instrução de retorno vazio, com o objetivo de servir para o *breakpoint.* Contudo, não é normal escrever estas instruções em situações normais, pois não surtem qualquer efeito.




```
    static void main() {
        Color red = new Color(255, 0, 0);
        Color blue = new Color(0, 255, 0);
        Color green = new Color(0, 0, 255);
        return;
    }
```

Cores

Neste exercício o objetivo é desenvolver funções que criam cores RGB. As cores são representadas por objetos da classe Color (descarregar anexo). Os objetos podem ser criados e utilizados da seguinte forma:

```
Color c = new Color(255, 0, 0); // vermelho
int r = c.getR(); // 255, componente de vermelho
int g = c.getG();  // 0, componente de verde
int b = c.getB(); // 0, componente de azul
```

Um tom de cinzento corresponde a um objeto Color cujos valores de R, G, e B são iguais (valor referido como *luminosidade*). Por exemplo:

```
Color c = new Color(128, 128, 128); // cinzento intermédio
```

> Os objetos Color são imutáveis, isto é, após a sua criação não é possível alterar o valor de RGB.
>
> Desta forma, quaisquer transformações de cores terão que ser feitas criando novos objetos Color.

Desenvolva funções para:

a) Criar um tom de cinzento, dada a luminosidade.

```
static Color gray(int lum) {    }
```

b) Criar um tom de cinzento aleatório.

Dica: utilizar a função 1 do Trabalho Prévio.

```
static Color randomGray() {    }
```

c) Criar uma cor aleatória.

d) Criar um vetor de cores aleatórias dado um comprimento.

```
static Color[] randomColorArray(int length) {     }
```

e) Dada uma cor, obter a cor inversa (255 - R, 255 - G, 255 - B).

```
static Color inverted(Color color) {    }
```

f) Dada uma cor, criar uma cor mais clara/escura, dependendo de um acréscimo (delta) (positivo - mais clara; negativo - mais escura). Deve ter-se atenção para os valores se manterem válidos.

Dica: utilizar a função 2 do Trabalho Prévio.

```
static Color changeBrightness (Color color, int delta) { }
```

Imagens binárias

Os exercícios envolvem a manipulação de imagens binárias, caraterizadas pelo facto de cada píxel poder tomar apenas um de dois valores (visualmente teremos preto e branco). Utilize a classe BinaryImage fornecida nos anexos.

Um objeto BinaryImage pode ser criado da seguinte forma:

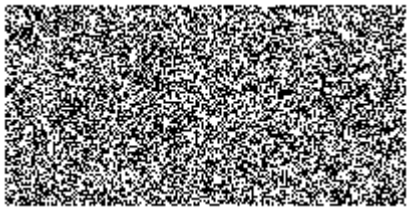
```
BinaryImage img = new BinaryImage(200, 100); // dimensão 200x100, todos os pixeis a preto
```

Exemplo de utilização das operações disponíveis:

```
int w = img.getWidth(); // 200, largura em pixeis
int h = img.getHeight(); // 100, altura em pixeis

// pinta o pixel do canto superior esquerdo de branco
img.setWhite(0, 0);

// pinta  o pixel do canto inferior direito de preto
img.setBlack(img.getWidth() - 1, img.getHeight() - 1);

// false, dado que o pixel (0, 0) não tem a cor preta
boolean b = img.isBlack(0, 0);

// true, dado que a coordenada (50, 50) é válida para uma imagem de 200x100
boolean v = img.validPosition(50, 50);
```

As seguintes alíneas consistem na descrição dos objetivos das funções/procedimentos a desenvolver.

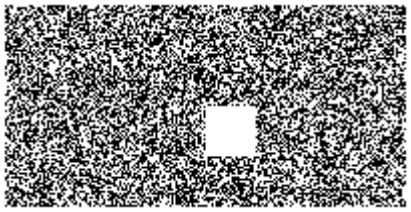
a) Criar uma imagem aleatória dada uma dimensão. A probabilidade de um píxel ser branco ou preto deverá ser equivalente. No exemplo abaixo, foi dada como dimensão 200 x 100.



Dica: utilizar a função 3 do Trabalho Prévio.

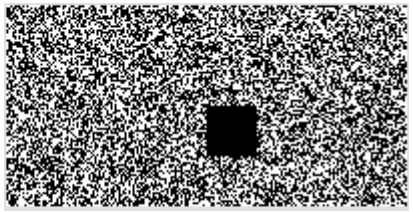
b) Preencher um quadrado branco fornecendo o ponto do canto superior esquerdo (x, y) e o comprimento de lado (side) do quadrado.

```
static void drawSquare(BinaryImage img, int x, int y, int side) { ... }
```



Neste exemplo, foi pintado um quadrado com canto no ponto (100, 50) e comprimento de lado 25.

c) Inverter uma imagem binária, de forma a que cada pixel branco passe a preto, e vice-versa.

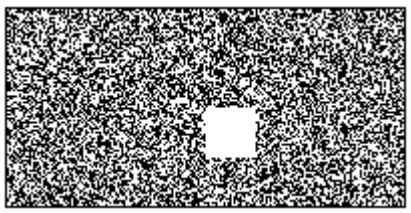


Desenvolva duas versões:

i) procedimento (que transforma a imagem)

ii) função (que cria uma imagem nova e deixa a existente inalterada).

d) Desenhar um contorno a preto nos píxeis limite de uma imagem.



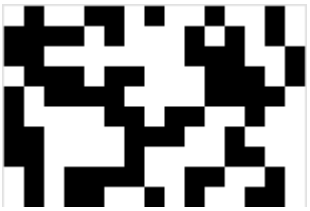
e) Criar uma imagem binária a partir de uma matriz de booleanos, tendo as mesmas dimensões da matriz, fazendo corresponder os valores *true* a branco e os *false* a preto. O resultado desta função, quando utilizada uma matriz aleatória, será equivalente ao da alínea (a).

```
static BinaryImage convert(boolean[][] data) {  }
```

Dica: utilizar a função 4 do Trabalho Prévio para gerar matrizes de teste.

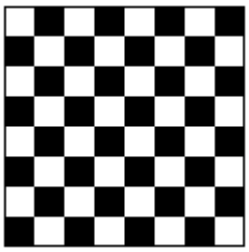
f) Torne a função anterior mais flexível, permitindo fornecer o tamanho de cada "pixel" na imagem (*n*). Ao passo que na função anterior cada valor booleano é mapeado para um único pixel, nesta será para um quadrado com dimensão variável (*n* x *n*).

```
BinaryImage img = convert(data, 10);
```



g) Criar uma imagem binária que forme um tabuleiro de xadrez, indicando o número de pixeis de cada posição. Note que há uma linha de um pixel em torno do tabuleiro.

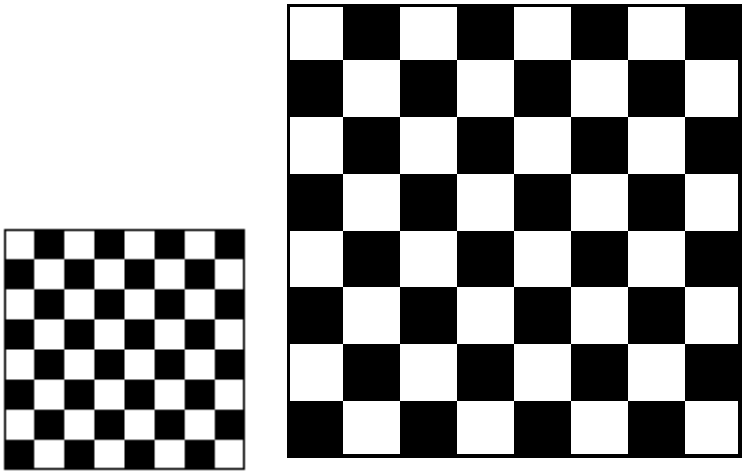
```
BinaryImage chess = createChessBoard(15);
```



Dica: utilizar a função 5 do Trabalho Prévio, em combinação com a função anterior.

h) Criar uma imagem escalada por um fator (p.e., fator 2 duplica o tamanho da imagem).

```
BinaryImage chessBig = scale(chess, 2);
```



Exercícios extra

a. Criar uma imagem binária através da sobreposição de duas imagens binárias, sendo que: se um pixel é branco em ambas as imagens, será branco na nova imagem; se um pixel é preto numa das imagens, será preto na nova imagem.

```
BinaryImage merge = merge(chess, chessBig);
```

b. Desenhar um círculo branco fornecendo o centro, sendo o raio igual a 5.

```
static void drawCircle(BinaryImage img, int x, int y) {  }
```

Dica: utilizar a função 6 do Trabalho Prévio.

c. Torne o procedimento anterior mais flexível, permitindo:

   i) raio do círculo variável

   ii) preencher a branco ou preto

d. Criar uma imagem binária com o seguinte aspecto.