# Curso rapidíssimo de Processing para programadores C

**Pedro Guerreiro**
**Universidade do Algarve**
**Abril 2020**

## Programando em Processing

A linguagem Processing baseia-se na linguagem Java, a qual se baseia no C. Por isso, a sintaxe tem muitas semelhanças.

Em particular, as instruções `if-else`, `for` e `while` são como no C.

As variáveis declaram-se das mesma maneira. Mas, em vez de `double`, usamos `float`.

```
float tax = 0.23;
```

Existe um tipo `boolean`, para os valores lógicos `true` e `false`.

```
boolean isOdd(int x)
{
  return x % 2 != 0;
}
```

Para as cadeiras de caracteres, em vez de `char *` usamos o tipo `String`.

```
String s = "Programming in Processing is a lot of fun";
```

Os nomes das variáveis e das funções, quando são compostos, usam o chamado-*camelCase*. Por exemplo:

```
int smallNumber = 3;
int largeNumber = 20000;
int veryLargeNumber = 5000000;

boolean isVeryLarge(int x)
{
  return x >= veryLargeNumber;
}
```

O *camelCase* para o Processing, como para o Java, é uma questão de estilo convencional. Não é um regra da linguagem.

Os arrays usam-se da mesma maneira, mas definem-se assim, por exemplo:

```
int a1[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
int a2[] = new int[100];
```

Os arrays são colocados sempre na memória dinâmica, tal como se pode depreender da utilização daquele operador `new`.

Em vez de `structs` usamos `class`, e colocamos o construtor logo dentro da classe, assim, por exemplo:

```
class Square {
  float x; // horizontal coordinate of upper left corner
  float y; // vertical coordinate of upper left corner
  float side; // lenght of side
  Square (float x, float y, float side)
  {
    this.x = x;
    this.y = y;
    this.side = side;
  }
}
```

As variáveis de um tipo `class` são sempre criadas na memória dinâmica, também com aquele operador `new`, aplicado ao construtor, assim, por exemplo:

```
Square sq = new Square(100, 100, 300);
```

Tal como em C, um programa é uma coleção de funções, de variáveis globais e de declarações de novos tipos. Atenção: em C, quase não usamos variáveis globais, mas em Processing usaremos, disciplinadamente.

Pormenor: na declaração de uma função que não tem argumentos não se coloca `void` na posição da lista de argumentos (dentro dos parêntesis). Veremos exemplos disto daqui a pouco.

Não há função `main`. Em vez da função `main`, há a função `setup` e a função `draw`.

A função `setup` é chamada uma vez, automaticamente, quando o programa começa. Usamo-la para inicializar variáveis globais (ainda que algumas possam ser inicializadas logo na declaração).

A função `draw` é chamada repetidamente, 60 vezes por segundo, até o programa terminar. Usamo-la para desenhar coisas no ecrã. Se desenharmos sempre a mesma coisa, teremos uma imagem fixa. Se variarmos ligeiramente o desenho, entre cada duas chamadas, teremos uma animação.

A função `draw`, que não tem argumentos, trabalha sobre as variáveis globais (tal como a função `setup`.

Os programas correm numa janela. Para terminar o programa, fecha-se a janela.

Não há `scanf`. O input é feito por interação na janela.

Há `print` e `println`, que usamos para escrever na consola do Processing. Serve sobretudo para *debug*.

Ao programar em Processing, estamos sempre a consultar a referência.

## Primeiro programa: uma bola

Queremos um programa para desenhar um círculo vermelho inscrito numa janela quadrada com fundo preto.

Abra o Processing, para começar.

Precisamos de duas variáveis (na verdade, duas constantes), para representar as cores preto e vermelho:

```
color black = color(0, 0, 0);
color red = color(255, 0, 0);
```

Cada cor é representada por três números, cada entre 0 e 255, representado as componentes vermelho, verde e azul na cor. Vermelho, verde e azul, em inglês, *red*, *green*, *blue*, ou RGB.

Guardamos o ficheiro, com o nome `ball`, numa pasta `processing`, dentro da diretoria `LP_1920` (e não na pasta `Processing`, que aparece à primeira). Repare

que é criada uma pasta `ball`, na pasta `processing`, e dentro dela o ficheiro `ball.pde`.

Tipicamente, na função `setup`, começamos por fixar o tamanho da janela, com a função `size`;

```
void setup()
{
  size(480, 480);
}
```

Para desenhar um círculo, usamos a função `circle`, a qual tem três argumentos: a coordenada horizontal do centro, a coordenada vertical do centro, a medida do diâmetro.

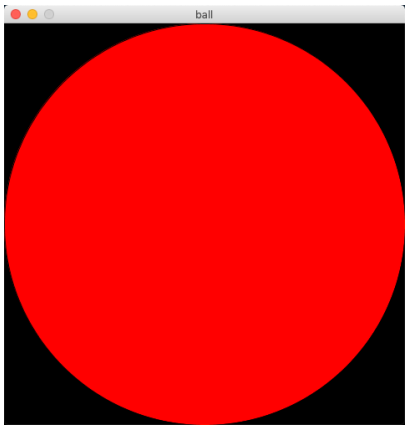
Quando uma figura fechada é desenhada, o seu interior sai com a cor que tiver sido fixada mais recentemente através da função `fill`. Antes de desenhar, limpamos o fundo, com a função `background`, "pintando-o" de uma cor pré-determinada, para apagar algum desenho que exista anteriormente. Logo, a função `draw` fica assim:

```
void draw()
{
  background(black);
  fill(red);
  circle(240, 240, 480);
}
```

O programa completo é a concatenação destas três partes: a declaração e inicialização das variáveis globais, a função `setup` e a função `draw`.

Guardamos na pasta `processing`, na nossa diretoria `LP_1920`.

E corremos a partir do ambiente de desenvolvimento, clicando no triângulo ou através do menu `Sketch->Run`.




## Estilo

Quanto ao estilo: usar aquelas constantes arbitrárias, 480, 240, espalhadas no programa, não fica bem. O melhor é fixar as dimensões da janela por meio de constantes e depois exprimir os cálculos em termos dessas constantes, ou equivalente. Observe, primeiro, a declaração das constantes:

```
final int windowWidth = 480;
final int windowHeight = windowWidth;
```

Por alguma razão técnica misteriosa, os argumentos da função `size` quando chamada na função `setup`, têm de ser números, não podem ser variáveis, nem sequer variáveis *finais*. Assim, das duas uma: ou deixamos como estava ou recorremos à função `settings`, que corre antes da função `setup`:

```
void settings()
{
  size(windowWidth, windowHeight);
}

void setup()
{
  //size(windowWidth, windowHeight); // not allowed here
}
```

Na função `circle`, usamos também as variáveis que fixámos de início:

```
void draw()
{
  background(black);
  fill(red);
```

```
  circle(windowWidth/2, windowHeight/2, windowWidth);
}
```

Guardamos esta versão num outro ficheiro.

Faça experiências com janelas de outros tamanhos, porventura com janelas retangulares com lados de comprimentos diferentes, mudando os valores das constantes e ajustando os argumentos na função `circle`.

## Animação

Experimentemos uma variante: fazer o círculo começar vazio e crescer até ao máximo, depois diminuir até desaparecer, depois aumentar até ao máximo e depois diminuir, e assim sucessivamente.

Precisamos de uma variável, ou de uma expressão, para usar na largura e na altura da elipse, em vez da constante 480, agora representada pela variável `windowWidth`. Da primeira vez que o círculo é desenhado, o diâmetro deve ser 0; da segunda, 1; depois 2, 3, ..., 479, 480, 479, 478, ..., 3, 2 ,1, 0, 1, 2, etc. Se tivermos uma variável inteira que conte o número de vezes que a função `draw` foi chamada, "basta" transformar esse valor no correspondente valor da sequência, assim, por exemplo:

```
int diameter(int x)
{
  int result = x % (windowWidth*2);
  if (result > windowWidth)
    result = windowWidth*2 - result;
  return result;
}
```

O número de vezes que a função `draw` foi chamada é registado na variável `frameCount`. Logo, a função `draw` fica assim:

```
void draw()
{
  background(black);
  fill(red);
  int d = diameter(frameCount);
  circle(windowWidth/2, windowHeight/2, d);
}
```

Guardamos noutro ficheiro e, experimentando, observamos o círculo a crescer e depois diminuir, alternadamente, como queríamos

A função `fill` preenche o interior da figura que for desenhada a seguir com a cor indicada no argumento. Por sua vez, o contorno da figura é desenhado com a cor que tiver sido fixada através da função `stroke`. Por exemplo, para desenhar os círculos vermelhos com contorno branco, faz-se assim:

```
final color black = color(0, 0, 0);
final color red = color (255, 0, 0);
final color white = color (255, 255, 255);

// ...

void draw()
{
  background(black);
  stroke(white);
  fill(red);
  int d = diameter(frameCount);
  circle(windowWidth/2, windowHeight/2, d);
}
```

Na verdade, por defeito o contorno é desenhado a preto, mas, como o fundo também era preto, não se notava.

Se não quisermos contorno nenhum, chamamos a função `noStroke`.

```
void draw()
{
  background(black);
  noStroke();
  fill(red);
  // ...
}
```

Se quisermos um contorno mais ou menos grosso, usamos a função `strokeWeight`:

```
void draw()
{
  background(black);
  stroke(white);
```

```
  strokeWeight(20);
  fill(red);
  // ...
}
```

## Classe *Circle*

Mais interessante será desenharmos vários círculos, em vez de apenas um. Para isso precisamos de um array de círculos e para ter um array de círculos precisamos antes de um tipo `Circle` que represente o conceito de círculo, adequadamente ao problema.

De facto, nos anteriores programas desenhámos círculos, mas não havia no programa nenhum objeto que representasse círculos.

Ora, na nossa aplicação, podemos considerar que um círculo é caracterizado pelas coordenadas do centro, pela medida do raio e pela cor. Logo:

```
class Circle
{
  float x; // x coordinate of center
  float y; // y coordinate of center
  float r; // radius
  color c; // color

  Circle (float x, float y, float r, color c)
  {
    this.x = x;
    this.y = y;
    this.r = r;
    this.c = c;
  }
}
```

Para desenhar um círculo destes, convém uma função. Por hipótese, será um círculo sem contorno:

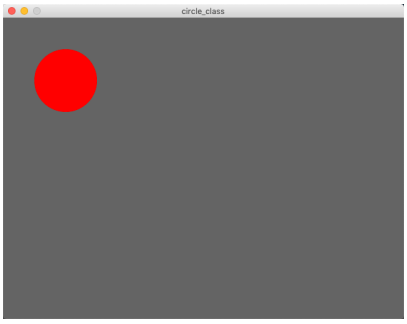
```
void circleDraw(Circle c)
{
  noStroke();
  fill(c.c);
  circle(c.x, c.y, 2*c.r);
}
```

Não confunda o argumento `c`, de tipo `Circle`, com o membro `c` de tipo `color`. A coincidência de nomes é fortuita.

Eis um programa completo, em esquema, para exercitar a classe e esta função `circleDraw`:

```
final color black = color(0, 0, 0);
final color red = color (255, 0, 0);
final color white = color (255, 255, 255);

final int horizontalSide = 640;
final int verticalSide = 480;

class Circle
{
  // ...
}

void circleDraw(Circle c)
{
  // ...
}

Circle c1 = new Circle(100, 100, 50, red);

void settings()
{
  size(horizontalSide, verticalSide);
}

void setup()
{
}

void draw()
{
  background(100);
  circleDraw(c1);
}
```

Ao correr o programa, a janela fica com o seguinte aspeto:




Neste caso, temos um *background* cinzento (e não preto), para se perceber que o círculo não tem contorno.

E note que usamos uma janela com largura 640 e altura 480, correspondente à razão 4:3. À falta de outros critérios, é boa ideia usar uma das razões habituais: 1:1, 3:2, 4:3 ou 16:9.

Em vez de usar aquela função `circleDraw`, é mais prático definir uma função *dentro* da classe, para o mesmo efeito. Observe:

```
class Circle {
  float x; // x coordinate of center
  float y; // y coordinate of center
  float r; // radius
  color c; // color

  Circle (float x, float y, float r, color c)
  {
    this.x = x;
    this.y = y;
    this.r = r;
    this.c = c;
  }

  void draw()
  {
    noStroke();
    fill(c);
    circle(x, y, 2*r);
  }
}
```

Com isto, a função `draw` ficaria assim:

```
void draw()
```

```
{
  background(100);
  c1.draw();
}
```

Não confunda as duas funções `draw`: uma é a "velha" função do Processing; a outra é uma função definida "dentro" da classe `Circle`. Estas funções definidas dentro das classes são chamadas *métodos*.

## Array de círculos

O próximo exercício é desenhar quatro círculos, um vermelho, um verde, um azul e um branco, cada um numa quarta parte da janela, todos crescendo e encolhendo ao mesmo tempo.

Usaremos uma janela rectangular, como no exemplo anterior:

```
final int horizontalSide = 640;
final int verticalSide = 480;
```

Assim, determinamos que o retângulo da janela é dividido em quatro retângulos de 320x240 e que o cada círculo fica centrado num desses retângulos, sendo a medida do raio igual a metade da altura desses retângulos, ou seja, 120.

Como temos uma nova cor a juntar às que usámos anteriormente, talvez seja boa altura para organizarmos a nossa coleção de cores. Contentamo-nos, por enquanto, com preto e branco, com as cores primárias, vermelho, verde e azul, e com as cores secundárias, amarelo, magenta e ciano:

```
final color black   = color(0, 0, 0);
final color white   = color(255, 255, 255);

// Primary colors
final color red     = color(255, 0, 0);
final color green   = color(0, 255, 0);
final color blue    = color(0, 0, 255);

// Secondary colors
final color yellow  = color(255, 255, 0);
final color magenta = color(255, 0, 255);
final color cyan    = color(0, 255, 255);
```

Vamos primeiro desenhar os círculos parados. Depois, logo os faremos crescer e encolher.

Eis os quatro círculos, inicializados um a um, da esquerda para a direita e de cima para baixo:

```
Circle c1 = new Circle(horizontalSide * 0.25, verticalSide *
0.25, verticalSide / 4, red);
Circle c2 = new Circle(horizontalSide * 0.75, verticalSide *
0.25, verticalSide / 4, green);
Circle c3 = new Circle(horizontalSide * 0.25, verticalSide *
0.75, verticalSide / 4, blue);
Circle c4 = new Circle(horizontalSide * 0.75, verticalSide *
0.75, verticalSide / 4, white);
```

Note que na janela o ponto de coordenadas (0, 0) é o canto superior esquerdo e e que a coordenada *y* cresce de cima para baixo.

Em vez de processar os círculos um a um, é mais prático metê-los num array e processá-los em conjunto. Podemos fazer assim:

```
Circle circles[] = {c1, c2, c3, c4};
```

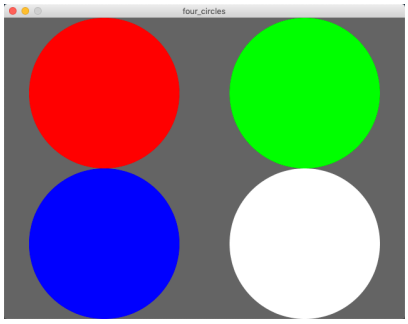
Desenhamo-los todos de uma vez, com um ciclo `for`:

```
void circlesDraw(Circle a[], int n)
{
  for (int i = 0; i < n; i++)
    a[i].draw();
}
```

E chamamos esta função `circlesDraw` na função `draw`:

```
void draw()
{
  background(100);
  circlesDraw(circles, 4);
}
```

Eis o desenho que obtemos ao correr o programa:




## Círculos crescentes

Queremos agora que os círculos cresçam e encolham. Temos de especificar até quanto devem crescer e qual a velocidade a que crescem e encolhem. Portanto, a classe dos círculos crescentes, `GrowingCircle`, será como a classe `Circle`, com mais dois membros, um para o raio máximo e outro que indique de quanto deve variar o raio entre cada duas chamadas da função `draw`:

```
class GrowingCircle
{
  float x;    // x coordinate of center
  float y;    // y coordinate of center
  float r;    // radius
  color c;    // color
  float dr;   // radius variantion per frame
  float rMax; // max radius.

  GrowingCircle (float x, float y, float r, color c, float dr)
  {
    this.x = x;
    this.y = y;
    this.r = r;
    this.c = c;
    this.dr = dr;
    this.rMax = r; // the max radius is the initial radius
  }

  void draw()
  {
    noStroke();
    fill(c);
    circle(x, y, 2*r);
  }
}
```

Note que, como o raio máximo é o raio inicial, o círculo, após ter sido inicializado com um dado raio, começa por encolher e só depois cresce, novamente até à dimensão inicial.

Esta classe repete muito da anterior, e o método `draw` até é igual ao outro. Mas repare que ainda faltará o método que há de fazer mudar o raio, antes de cada chamada da função `draw`.

Há uma maneira bem prática de evitar estas repetições, considerando que a nova classe `GrowingCircle` *estende* a classe `Circle`, e programando assim:

```
class GrowingCircle extends Circle
{
  float dr;   // radius variation per frame
  float rMax; // max radius

  GrowingCircle (float x, float y, float r, color c, float dr)
  {
    super(x, y, r, c);
    this.dr = dr;
    this.rMax = r;
  }
}
```

Note que como a o método `draw` seria igualzinho, não é preciso repetir. E repare na instrução `super(x, h, r, c)`, que indica que usamos o construtor da classe `Circle`, a qual constitui a *superclasse* da classe `Circle`, para inicializar os membros da superclasse.

Agora cada um dos nossos círculos é desta nova classe `GrowingCircle`.

```
GrowingCircle c1 = new GrowingCircle(horizontalSide * 0.25,
verticalSide * 0.25, verticalSide / 4, red, 1);
GrowingCircle c2 = new GrowingCircle(horizontalSide * 0.75,
verticalSide * 0.25, verticalSide / 4, green, 2);
GrowingCircle c3 = new GrowingCircle(horizontalSide * 0.25,
verticalSide * 0.75, verticalSide / 4, blue, 0.5);
GrowingCircle c4 = new GrowingCircle(horizontalSide * 0.75,
verticalSide * 0.75, verticalSide / 4, white, 5);
```

Para ficar mais engraçado, damos valores diferentes ao parâmetro `dr`.

O array será um array de círculos crescentes:

```
GrowingCircle circles[] = {c1, c2, c3, c4};
```

Resta agora programar o método `grow`.

Ora bem: ao crescer, o raio aumenta de `dr`, mas se ao aumentar ficar maior que o raio máximo, fica com o raio máximo, e `dr` muda de sinal. Inversamente, se ao encolher o raio ficar negativo, então fica com raio 0 e `dr` muda de sinal:

```
void grow()
{
  r += dr;
  if (r > rMax)
  {
    r = rMax;
    dr = -dr;
  }
  if (r < 0)
  {
    r = 0;
    dr = -dr;
  }
}
```

Resta preparar uma função para fazer crescer (ou encolher) todos os círculos:

```
void circlesGrow(GrowingCircle a[], int n)
{
  for (int i = 0; i < n; i++)
    a[i].grow();
}
```

Esta função será chamada antes da função `circlesDraw`, dentro da função `draw`. Ainda assim, para distinguir bem a ação de desenhar e a ação de calcular a figura que vai ser desenhada, colocamos sistematicamente as operações que calculam a nova figura dentro de uma função `update`, a qual, essa sim, é chamada pela função `draw` antes desta começar a desenhar. Fica assim:

```
void update()
{
  circlesGrow(circles, 4);
}

void draw()
{
```

```
  update();
  background(100);
  circlesDraw(circles, 4);
}
```

## Outras figuras

Nos programas com que brincámos, apenas desenhámos círculos, usando a função `circle`. Bem entendido, há funções para outras figuras geométricas: `point`, para desenhar pontos; `ellipse`, para desenhar elipses; `arc`, para desenhar arcos de elipse; `line`, para desenhar segmentos de reta; `triangle`, para desenhar triângulos ; `square`, para desenhar quadrados ; `rect`, para desenhar retângulos; e `quad` para desenhar quadriláteros.

É tudo, para já.