Processing – Introdução a Programação

Autores:

Alguém que fez estes slides e de modo humilde, deixou ou esqueceu de colocar o seu nome –

Logo, um autor desconhecido

Demais autores:

- Luiz Ernesto Merckle
- Vilson Vieira
- Claudio Cesar de Sá

Processing – Elementos da Linguagem

Os **comentários** são ignorados pelo computador, mas são importantes para as pessoas.
Através dos comentários podemos apontar notas e lembretes sobre a própria estrutura do código.
Todas as letras e símbolos que não são comentários são traduzidos pelo compilador.

Comentário (//)

```
// Duas barras são utilizadas para denotar um comentário
// não deve existir espaço entre as barras



/*

  Essa forma é usada para a escrita de
  comentários mais longos
*/
```

Processing – Elementos da Linguagem

As funções permitem o desenho de formas, seleção de cores, cálculos e a execução de outros tipos de ação.
O **nome** de uma função geralmente começa com letra minuscula e é seguido por parênteses. Os **parâmetros** são os elementos separados por vírgula que ficam dentro dos parênteses.
Algumas funções não possuem parâmetros enquanto outras podem conter muitos.

Funções

```
// A função size contem 2 parametros.
// O primeiro é a largura da janela de display e o segundo
// define a altura

size (200,200);



// Essa versão da função background contem apenas 1
// parametro. Ele determina a quantidade de cinza do fundo
// que pode variar de 0 (preto)  - 255 (branco)

background(102);
```

Processing – Elementos da Linguagem

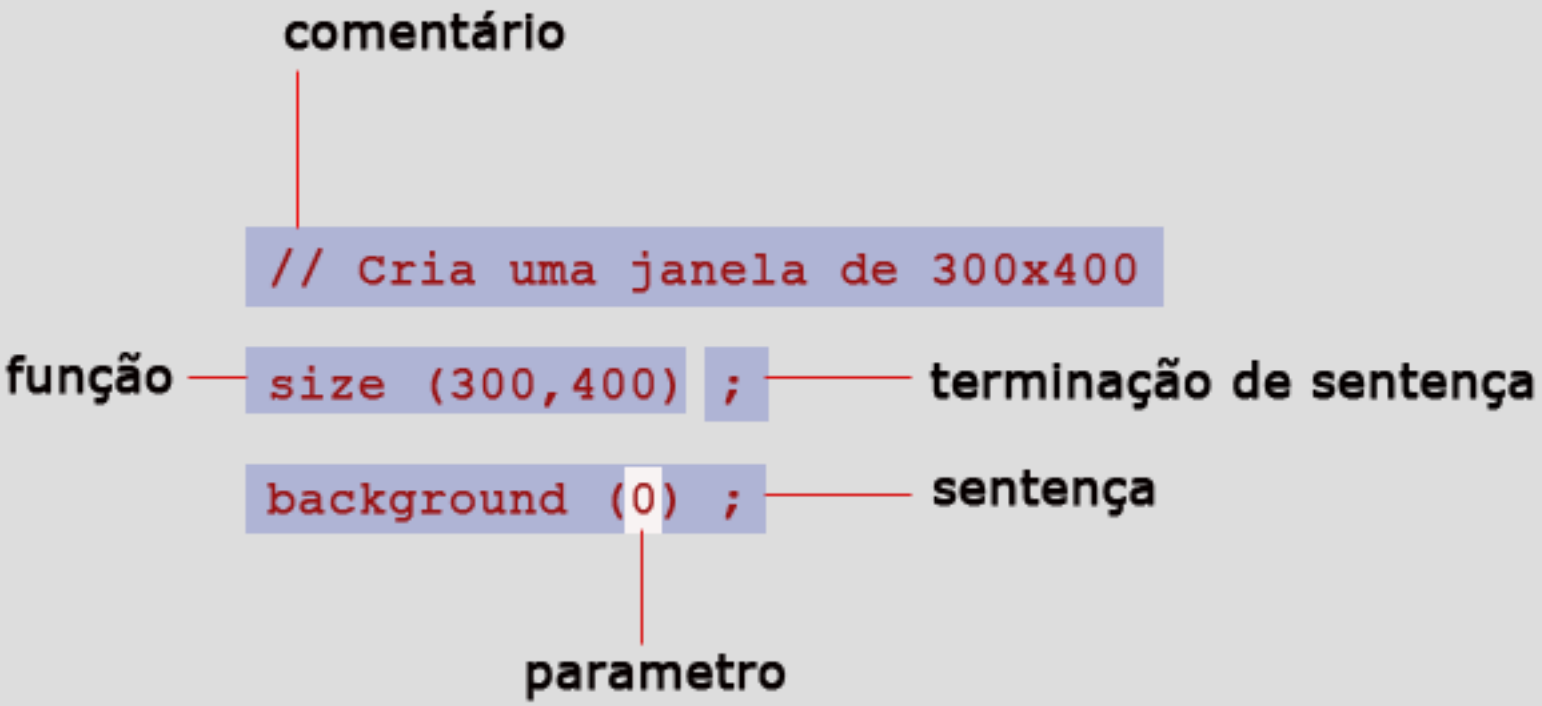
As expressões estão para o software assim como as frases estão para a linguagem humana.
Geralmente elas são combinações de **operadores** como +, * , e /.

Expressões

```
5

15.0+1.0

((3+2)*-10)+1

6>3

54<50
```

Processing – Elementos da Linguagem

O Processing diferencia caracteres maíusculos e minúsculos (***case sensitive***).Por exemplo , escrever "Size" pode produzir um erro. Na digitação, a quantidade de espaços entre os elementos não prejudica a execução correta do código.



Processing – Elementos da Linguagem

As funções **print()** e **printnl()** podem ser usadas para mostrarem dados durante a execução de um programa.
As funções não enviam páginas para uma impressora, mas escrevem texto para o **console**.
O console também pode ser usado para mostrar o estado ou valor de uma variável, confirmar um evento ou checar os dados enviados por um dispositivo externo.

Mensagens no Console – print() e println()

```
// Para imprimir um texto na tela
println (“Processing é bacana”);

//Enquanto println() salta para a próxima linha após a impressão
//o comando print() não salta

print(“10”);
println(“20”);
println(“30”);
```

Processing – Coordenadas e Primitivas

A tela do computador é composta de uma grade de milhões de **pixels**. Uma posição nesta grade corresponde a uma coordenada (**x**-horizontal e **y**-vertical. No Processing a origem destas coordenadas (0,0) está no canto superior esquerdo da tela. Logo, os valores das coordenadas aumentam de cima para baixo e da esquerda para direita. Podemos desenhar algumas figuras geométricas primitivas como pontos, linhas e planos utilizando essas coordenadas.

point (x,y)

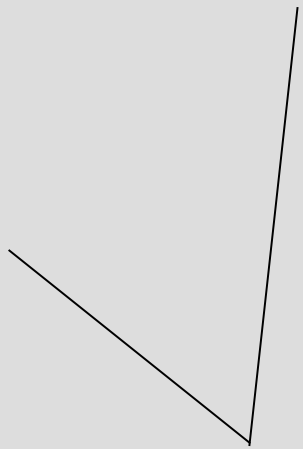
```
// um ponto corresponde a um pixel desenhado na coordenada x,y
Point (10,30);


// pontos com x,y identicos formam uma diagonal
point (20,20);
point (30,30);
point (40,40);
point (50,50);
point (60,60);


// pontos negativos não são visiveis
point (-10,20);
```

Processing – Coordenadas e Primitivas

Além de pontos podemos desenhar: linhas,triângulos,quadriláteros, retângulos,elipses e curvas bezier.

line (x1, y1, x2, y2)

```
// x1,y1 é o inicio  da linha e x2,y2 o fim
line (10,30,90,30)

// linhas verticais possuem  x1 e x2 identicos
line (40,10,40,90);
line (50,10,50,90);

// linhas horizontais possuem y1 e y2 identicos
line (10,30,90,30);
line (10,40,90,40);
```



Ex: desenhar duas linhas que compartilhem um ponto de conexão

Processing – Coordenadas e Primitivas

Além de pontos podemos desenhar: linhas,triângulos,quadriláteros, retângulos,elipses e curvas bezier.

triangle (x1, y1, x2, y2, x3, y3)

```
// os  3 pares de coordenadas definem os vertices do triangulo
triangle (60,10,25,60,75,65);
```

quad (x1, y1, x2, y2, x3, y3, x4, y4)

```
// os  4 pares de coordenadas definem os vertices do poligono
quad (20,20,20,70,60,90,60,40);
quad  (20,20,70,-20,110,0,60,40);
```

rect (x, y, largura, altura)

```
// os 2 primeiros parametros definem a coordenada do vertice superior esquerdo
// o terceiro e quarto parametro define a largura e altura do retangulo
rect (15,15,40,40);
rect  (55,55,25,25);
```

ellipse (x, y, largura, altura)

```
// os 2 primeiros parametros definem a localização do centro da elipse
// o terceiro e quarto parametro define a largura e altura da elipse
ellipse(35,0,120,120);
ellipse (38,62,6,6);
ellipse (40,100,70,70);
```

Processing – Coordenadas e Primitivas

A função **bezier** desenha uma curva orientada por uma série de pontos de controle e âncoras.
A curva é desenhada entre os pontos de âncora e os pontos de controle determinam sua forma.

bezier (x1, y1, cx1, cy1, cx2,cy2, x2, y2)

```
// a função contem 8 parametros que determinam 4 pontos
// a curva é desenhada entre o primeiro e quarto ponto
// os pontos de controle são determinados pelo segundo e terceiro parametro

bezier(32,20,80,5,80,75,30,75)
```

* ver exemplo na pasta Elementos

```
// podemos visualizar estes pontos
// basta desenhar circulos nas coordenadas dos pontos
// e linhas que ligam estes pontos

bezier (85,20,40,10,60,90,15,80);
line (85,20,40,10);
ellipse (40,10,4,4);
line (60,90,15,80);
ellipse (60,90,4,4);
```

Processing – Coordenadas e Primitivas

No código, a ordem das sentenças define qual figura aparece em primeiro plano.

Ordem de desenho

```
// se um retangulo é desenhado na primeira linha
// ele é mostrado antes de uma elipse que é desenhada na segunda linha
// revertendo a ordem colocamos o retangulo por cima

rect( 15,15,50,50);
ellipse (60,60,55,55);

ellipse (60,60,55,55);
rect (15,15,50,50);
```

Processing – Coordenadas e Primitivas

A função **fill** altera o valor de preenchimento das figuras e a **stroke** altera o valor de contorno.
Se nenhum valor de preenchimento é definido, o valor 255 (branco) é usado.
No caso do contorno 0 (preto) é o valor default.

fill (cinza, transparencia) - stroke(cinza)

```
// desenha um retangulo com preenchimento cinza
fill(153);
rect( 15,15,50,50);


// desenha um retangulo branco com contorno cinza
stroke(153);
rect( 15,15,50,50);

* ver exemplo na pasta Elementos
// desenha retangulos com niveis de transparencia
fill(0);
rect (0,40,100,20);
fill (255,51);
rect (0,20,33,60);
fill (255,127);
rect (33,20,33,60);
fill (255,204);
rect (66,20,33,60);
```

Obs: as funções **noFill()** e **noStroke()** interrompe o desenho de preenchimento e contorno automático (default).

Processing – Coordenadas e Primitivas

Os atributos de geometria também podem ser modificados.
As funções **smooth** e **noSmooth** habilita e desabilita a suavização (antialiasing). Uma vez usadas, toda as formas são afetadas.

smooth ( ) - noSmooth( )

```
smooth ();
ellipse (30,48,36,36);
noSmooth();
ellipse (70,48,36,36);
```

Processing – Coordenadas e Primitivas

Os atributos das linhas podem ser controlados por **strokeWeight** , **strokeCap** e **strokeJoin**.

* ver exemplo na pasta Elementos

strokeWeight (espessura)

```
smooth ();
line (20,20,80,20);
strokeWeight(6);
line (20,40,80,40);
strokeWeight(18);
line (20,70,80,70);
```

strokeCap ( )

```
// strokeCap requer  apenas um parametro que pode ser : ROUND, SQUARE ou PROJECT

smooth ();
strokeWeight(12);
strokeCap(ROUND);
line (20,30,80,30);
strokeCap(SQUARE);
line (20,50,80,50);
strokeCap(PROJECT);
line (20,70,80,70);
```

Processing – Coordenadas e Primitivas

- construir uma composição contendo uma elipse e uma linha

- modificar o código do exercicio anterior alterando o preenchimento, contorno e fundo (background).

Processing – Cores

Trabalhar com cores na tela do computador é diferente da aplicação de pigmentos sobre papel ou tela. Por exemplo, se adicionamos todas as cores juntas na tela do computador temos o branco como resultado. Na tela do computador misturamos as cores com adição de luz. No computador, a maneira mais comum de determinar uma cor é através do código RGB- ele determina a quantidade de vermelho, verde e azul.
A intensidade de cada elemento de cor é usualmente especificada com valores entre 255 e 0.

```
background (r,g,b)
fill (r,g,b)
fill (r,g,b,alpha)
stroke (r,g,b)
stroke (r,g,b,alpha)

// desenhando um quadrado com preenchimento verde

background (129,130,87);
noStroke();
fill(174,221,60);
rect (17,17,66,66);

// desenhando um quadrado com contorno verde sem preenchimento

background (129,130,87);
noFill();
strokeWeight(4);
stroke(174,221,60);
rect (19,19,62,62);
```

Processing – Cores

No processing, podemos selecionar o valor RGB de uma cor utilizando a ferramenta **Color Selector** que está no **Menu Tools**.O RGB do vermelho puro, por exemplo, é (255,0,0) ou #FF0000 (notação hexadecimal).

Além do RGB, podemos determinar a transparência (alpha) de uma cor utilizando o quarto parâmetro das funções.

```
* ver exemplo na pasta Elementos
// desenhando retangulos com alphas diferentes

background (116,193,206);
noStroke();
fill(129,130,87,102);
rect (20,20,30,60);
fill (129,130,87,204);
rect (50,20,30,60);
```

Processing – Cores

Usando a sobreposição de formas, a transparência pode ser usada para a criação de novas cores.

* ver exemplo na pasta Elementos

```
// desenhando circulos com alphas diferentes

background (0);
noStroke();
smooth();
fill(242,204,47,160); // amarelo
ellipse(47,36,64,64);
fill (174,221,60,160); // verde
ellipse(90,47,64,64);
fill(116,193,206,160);// azul
ellipse(57,79,64,64);

// experimente mudar a tonalidade do background
```

Processing – Lógica e Algoritmo

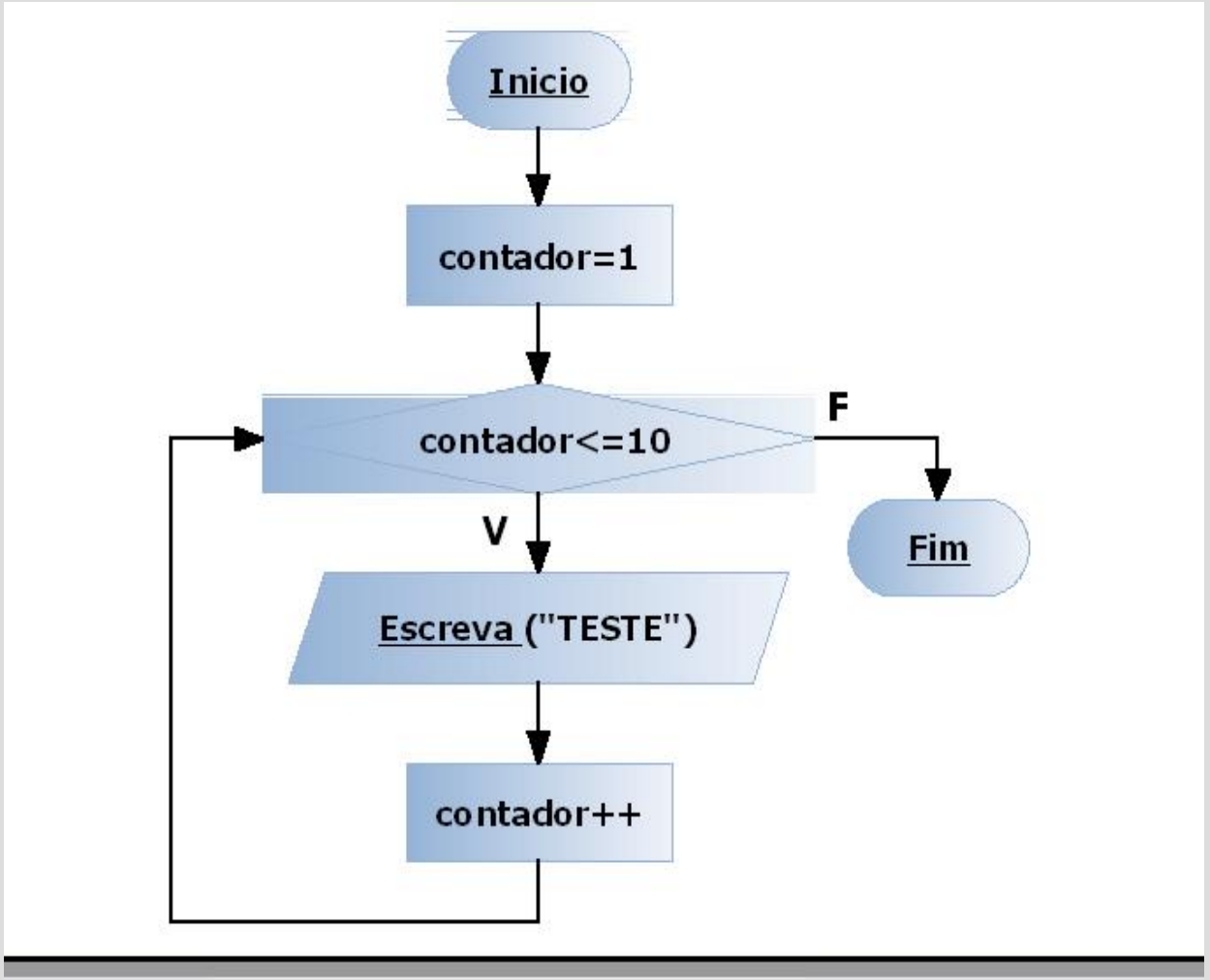
A elaboração de videogames, planilhas ou editores utilizam estruturas lógicas comuns como sequências lineares, decisões e repetições. Qualquer aplicação computacional estabelece um problema a ser resolvido e a divisão deste problema em unidades finitas faz parte do cotidiano de qualquer desenvolvedor. Antes de desenvolver qualquer software devemos saber representar os problemas envolvidos nesta atividade.
Um **algoritmo** é o conjunto de regras que fornece uma sequência de operações para resolver um problema especifico.

O fluxograma representa um algoritmo de maneira diagramática

Processing – Lógica e Algoritmo

A elaboração de videogames, planilhas ou editores utilizam estruturas lógicas comuns como sequencias lineares, decisões e repetições. Qualquer aplicação computacional estabelece um problema a ser resolvido e a divisão deste problema em unidades finitas faz parte do cotidiano de qualquer desenvolvedor. Antes de desenvolver qualquer software devemos saber representar os problemas envolvidos nesta atividade.
Um **algoritmo** é o conjunto de regras que fornece uma sequencia de operações para resolver um problema especifico.

De novo, o fluxograma representa um algoritmo de maneira diagramática



Processing – Lógica e Algoritmo

Os algoritmos podem ser representados no formato de diagrama, linguagem natural ou pseudocódigo.

```
Descrição do algoritmo em linguagem natural:
Iniciar o programa
O contador inicia com valor um.
Se o contador não chegar a dez eu imprimo a palavra teste e
incremento o contador.
Se o contador for maior que 10 eu encerro programa.

Descrição do programa em pseudocódigo:
INICIO
Contador=1

PASSO 1 {
SE (contador<=10) ENTÃO
    escrever "teste"
    contador=contador+1
    IR para PASSO 1
SENÃO
 IR para FINAL
}

FINAL
```

Processing – Estruturas

Além dos elementos geométricos, o Processing pode ainda manipular textos e imagens.
Antes de trabalhar com estes recursos vamos examinar um pouco as estruturas lógicas utilizadas na maioria das linguagens de programação.

- variáveis
- operações e funções matemáticas
- decisões e condicionais
- repetição

Processing – Variáveis

As memórias físicas dos computadores possuem um sistema de endereçamento para armazenagem e recuperação randomica de dados. Na programação, as variáveis correspondem a estes endereços e podem, por exemplo, salvar a posição de uma figura, do armazenamento de uma cor, ou de medir mudanças contínuas.
O Processing pode armazenar e modificar diferentes tipos de dados, incluindo numeros, letras, palavras, cores, Imagens, fontes e valores booleanos (verdadeiro, falso).

```
// as variáveis permitem a reutilização de uma informação durante a execução
// de um programa. Toda variável possui 2 partes, um nome e um valor.
// Adicionalmente, uma variável possui um tipo ou categoria de dado que ela
// pode armazenar.
// A variável deve  ser declarada antes de ser usada. Esta declaração define
O nome da variável e o tipo de valor que  pode armazenar.


int x; // Declara a variável x do tipo int
float y;
boolean b;

x=50; // atribue um valor para x;
Y=12.6;
b=true;


No Actionscript (Flash) a declaração de variáveis é um pouco diferente:

var x:int;
var b:boolean;
```

Processing – Variáveis

Após a declaração de uma variável não podemos alterar o tipo de dados.
Podemos declarar mais de uma variável em apenas uma linha de código, e os valores adicionados posteriormente.

```
float x, y, z;

x=-3.9;
y= 10.1;
z=124.23;

// podemos usar um atalho para declarar e associar um valor na mesma linha

int x=50;
boolean b=true;
```

Processing – Operações Matemáticas

As operações matemáticas são importantes , pois com elas podemos manipular resultados que podem controlar a posição e outras **propriedades** dos elementos visuais

```
= (atribuição)
+ (adição)
- (subtração)
* (multiplicação)
/ (divisão)
% (módulo)
( )
++ (incremento)
-  (decremento)
+= (atribuição aditiva)
-= (atribuição subtrativa)
*= (atribuição multiplicativa)
/= (atribuição divisão)
 - (negação)
```

Funções Aritmeticas

```
ceil (2.1) = 3  arredonda para cima

floor(2.1) = 2  arredonda para baixo

round(2.9)= 3   valor máximo

min(5,9)= 5     valor minimo na série

max(-4,-12,-9)= -4 valor máximo na série
```

Processing – Operações Matemáticas

Através do sinal = podemos atribuir valores ou expressões a uma determinada variável.

* ver exemplo na pasta ESTRUTURA

```
// no exemplo, definimos uma variável com o tipo inteiro (int)
// e o nome da variável é "cinza"
// inicialmente a variável recebe um valor (153) através do sinal = (atribuição)
// a variável é utilizada pela função fill() para o prenchimento do retangulo
// posteriormente a variável recebe o valor de uma expressão (cinza+102)
// isto é, recebe o valor da própria variável adicionada de 102


int cinza=153;
fill (cinza);
rect (10,10,55,55);
cinza=cinza+102;
fill(cinza);
rect (35,30,55,55);
```

Processing – Operações Matemáticas

Ainda podemos usar o sinal de atribuição (=) juntamente com sinais aritméticos (+,-,*,/) para criar atalhos capazes de incrementar ou decrementar valores numa variável.

```
int x=1
println (x); // imprime 1 no console
x++          // equivalente a x=x+1
println(x);  // imprime 2 no console
```

```
int y=1;
println(y);
y--
println(y);
```

```
int x=1;
println(x);
x+=5;
println(x); //imprime 6 no console
```

```
// ainda podemos inverter o sinal de um valor
int x=5;
x=-x;
println(x); //imprime -5 no console
```

Processing – Decisões

Na maioria das linguagens os programas são executados linearmente, linha a linha, de baixo para cima.
Apesar disso, podemos alterar a ordem de **fluxo** dos programas utilizando **estruturas decisórias** que podem desviar intencionalmente a ordem de execução linear. Estas estruturas tambem são utilizadas quando precisamos saltar certos pontos de processamento que necessitam que certas **condições** sejam previamente validadas. Portanto, para verificarmos se uma condição é verdadeira ou falsa devemos utilizar **expressões relacionais**.

```
>   maior que
<   menor que
>=  maior ou igual a
<=  menor ou igual a
==  equivalente a
!=  diferente (não equivalente a)
```

```
println (3>5); //imprime false

println (3>=5);//imprime false

println (3<=5);//imprime true

println (5==5);//imprime true

println (5<=5); // imprime true

println (5!=5);//imprime false

println (5==5);//imprime true
```

Processing – Decisões

Uma condicional permite ao programa fazer decisões sobre quais linhas de código deve ou não executar.
Eles executam ações apenas quando uma condição especifica é encontrada. As condicionais permitem que um Programa se comporte diferente dependendo do valor de suas variáveis. No Processing, assim como em outras linguagens utilizamos a estrutrura **if** para construir tais decisões.

Estrutura IF

A (condição) deve ser uma expressão que resulte em **true** (verdadeiro) ou **false**.
Quando a expressão resulta em true, o código dentro das chaves **{ }** é executado.
Se a expressão é false, o código (bloco de sentenças) é ignorado.

```
if (condição) {

    sentença
    sentença
    sentença
}
```

* ver exemplo na pasta ESTRUTURA

```
// altere o valor de x para mudar o resultado

int x= 150;
if (x>100) {
  ellipse(50,50,36,36);
}

if (x<100) {
  rect(35,35,30,30);
}

line(20,20,80,80);
```

Processing – Decisões

Ainda podemos utilizar a estrutura **IF...ELSE** para aumentar as possibilidades de execução no mesmo bloco de código.

Estrutura IF...ELSE

Se a condição for verdadeira o primeiro bloco de instruções é executado, senão (else) , se a condição for false, o segundo bloco é executado.

```
if (condição) {
   bloco1
} else {
   bloco2
}
```

* ver exemplo na pasta ESTRUTURA

```
// experimente alterar o valor de x

int x= 90;
if (x>100) {
   ellipse (50,50,36,36);
} else {
   rect(33,33,34,34);
}
line (20,20,80,80);
```

Processing – Decisões

Dentro de uma estrutura condicional podemos aninhar outras condicionais, estruturando uma sequência de testes posicionados em cascata.

```
// qual é o resultado deste código elipse, linha ou retangulo?

int x= 420;
if (x>100) {

   if (x<300) {
      ellipse (50,50,36,36);
   } else {
   line(50,0,50,100);
   }

} else {
 rect (33,33,34,34);
}
```

Processing – Decisões

Os **operadores lógicos** são usados para combinar duas ou mais expressões relacionais ou inversão de seus valores. Eles permitem a consideração de condições simultâneas.

Operadores Lógicos

```
&&   AND (todas as condições devem ser true)
||   OR  (uma das condições deve ser true)
!    NOT (o inverso da condição deve ser true)
```

Na tabela seguinte verificamos todas as combinações possíveis e seus resultados:

Expressão	Resultado
true && true	true
true && false	false
false && false	false
true || true	true
true || false	true
false || false	false
!true	false
!false	true

Processing – Decisões

* ver exemplo na pasta ESTRUTURA

```
// neste exemplo apenas a linha é desenhada

int a=10;
int b=20;
if ((a>5) && (b<30)) { // true &&  true
    line (20,50,80,50);
}

if ((a>15) && (b<30)) { // false && true
   ellipse (50,50,36,36);
}


// neste exemplo a linha é desenhada, pois a variável b=true

boolean b= true;
if (b==true) {   // se b é true desenhe a linha
    line (20,50,80,50);
}
if (!b== true) { // se b é false desenhe a elipse
   ellipse (50,50,36,36);
}
```

Processing – Repetição

As estruturas **iterativas** são utilizadas para compactar longas linhas de código repetitivo. Diminuindo a extensão do código tornamos os programas mais flexíveis em termos de manutenção e também ajuda na redução de erros. A estrutura **for** realiza cálculos e açoes repetitivas.

Estrutura FOR

```
for (inicio; teste; atualização) {
        Sentença
        Sentença
        Sentença
}
```

O parenteses associado a estrutura contem tres elementos: **inicio**, **teste** e **atualização**.
As sentenças dentro do bloco são executadas continuamente enquando o **teste** é validado como TRUE.
O elemento **inicio** determina o valor inicial da variável utilizada no teste.
O elemento **atualização** é usado para modificar a variável após cada iteração.

Sem o uso da estrutura **for** , se queremos desenhar 20 linhas horizontais paralelas devemos repetir o comando **line** 20 vezes, alterando uma por uma as coordenadas y (vertical) de cada linha. do código

```
line (20,0,80,0);
line (20,5,80,5);
line (20,10,80,10);
line (20,15,80,15);
          .
          .
          .
line (20,95,80,95);
```

Processing – Repetição

Estrutura FOR

Com a estrutura **for** as 20 linhas de código são reduzidas para apenas 3.

```
// a  variável i inicia com valor 0
// a condição  esperada é que i seja menor que 100
// na atualização, a variável i é incrementada de 5 (i+=5)
// em cada iteração o código desenha uma linha, variando as coordenadas y
// pois, em line (20,i,80,i), y1 e y2 equivalem a i
// o código é repetido 20 vezes, isto é, enquanto (i<100) seja verdadeiro


for (int i=0; i<100; i+=5) {
  line (20,i,80,i);
}
```

Processing – Repetição

A estrutura de repetição pode ser utilizada para o desenho de padrões pela simples modificação dos valores.

* ver exemplo na pasta ESTRUTURA – ex.01, ex.02 , ex.03

```
// neste exemplo varia a coordenada x
for (int x= -16; x<100; x+=10) {
    line (x,0, x+15, 50);
}

strokeWeight(4);
for (int x= -8; x<100; x+=10) {
    line (x, 50, x+15, 100);
}

// neste exemplo varia a largura e altura da elipse
noFill();
for (int d=150; d>0;d-=10){
    ellipse (50,50,d,d);
}

// neste exemplo variamos a coordenada x
// enquanto x for menor ou igual a 50 as coordenadas y1 e y2 equivale a 20 e 60
// senão, x continua variando mas coordenadas y1 e y2 mudam para 40,80

for (int x=20; x<=85; x+=5) {
     if (x<=50) {
        line (x,20,x,60);
     } else {
         line (x,40,x,80);
     }
}
```

Processing – Repetição

A estrutura de repetição pode ser utilizada para o desenho de padrões pela simples modificação dos valores.

* ver exemplo na pasta ESTRUTURA – ex.04, ex.05

```
// neste exemplo o código desenha uma série de linhas verticais, variando as coordenadas x1 e x2
// varia tambem a tonalidade de cinza (255-i) em cada iteração

for (int i=0; i<100; i+=2) {
     stroke (255-i);
     line (i, 0, i, 200);
}


// neste exemplo o código desenha uma série de retangulos
// o preenchimento de cada retangulo varia a cada iteração
// veja que o parametro alpha (129,130,87,i) assume o valor da variável i
// a cada iteração o valor de x tambem é incrementado (x+=20)
// pois os retangulos devem estar um ao lado do outro

background (116, 193, 206);
int x=0;
noStroke();
for (int i=51; i<=255; i+=51) {
     fill (129,130,87,i);
     rect (x,20,20,60);
     x+=20;
}
```

Processing – Repetição

Podemos usar a repetição para a variação de **matiz**, **saturação** e **brilho** das cores.
Além do RGB, no Processing podemos alterar as referências para uma cor utilizando a função **colorMode**.

* ver exemplo na pasta ESTRUTURA – ex.06, ex.07, ex.08

```
// No modo HSB,  H (hue) corresponde ao matiz , S (saturação ou pureza) e B (brilho)
// equivale a quantidade de luz.

// No exemplo, o código desenha uma série de linhas justapostas
// Variando o matiz (i*2.5 no parametro H) da função stroke

colorMode(HSB);
for (int i=0; i<100; i++) {
    stroke (i*2.5,255,255);
    line (i,0,i,100);
}

// No exemplo, o código desenha uma série de linhas justapostas
// Variando a saturação (i*2.5 no parametro S) da função stroke


colorMode(HSB);
for (int i=0; i<100; i++) {
    stroke (132,i*2.5, 204);
    line (i,0,i,100);
}

// No exemplo, o código desenha uma série de linhas justapostas
// Variando o brilho (i*2.5 no parametro B) da função stroke


colorMode(HSB);
for (int i=0; i<100; i++) {
    stroke (132, 108,i*2.5);
    line (i,0,i,100);
}
```

Processing – Repetição

Podemos usar a repetição para a variação de **matiz**, **saturação** e **brilho** das cores.
Além do RGB, no Processing podemos alterar as referencias para uma cor utilizando a função **colorMode**.

A função colorMode pode usar apenas um parametro – **colorMode (HSB)** ou **colorMode(RGB)**.
Quando utilizando HSB ainda é possível setar mais 3 parametros

ColorRange(HSB, valor1, valor2, valor3)

Os valores determinam o espectro(range) de cor, saturação e brilho possiveis de serem manipulados
- valor 1 equivale ao alcance maximo de matizes – 360 é o valor máximo
- valor 2 equivae ao alcance maximo de saturação – 100% é o valor máximo
- valor 3 equivale ao alcance máximo de brilho – 100% é o valor máximo

* ver exemplo na pasta ESTRUTURA – ex.09, ex.10

```
// exemplo de transição entre amarelo e azul
// escolhemos o valor 65 como matiz inicial para amarelo
// este valor é incrementado até chegar a 245 (65+180)que corresponde ao matiz do azul

colorMode (HSB, 360, 100, 100);
for (int i=0; i<100; i++) {
   float matiz=  65+(i*1.8);
   stroke (matiz, 70, 80);
   line (i,0,i,100);
}

// exemplo de transição entre azul e verde
// escolhemos o valor 200 como matiz inicial para o azul
// este valor é decrementado até chegar a 80 (200-120) que corresponde ao matiz do verde

colorMode (HSB, 360, 100, 100);
for (int i=0; i<100; i++) {
   float matiz=  200-(i*1.2);
   stroke (matiz, 70, 80);
   line (i,0,i,100);
}
```

Processing – Repetição

Utilizando repetição e colorMode (HSB,valor1,valor2,valor3)

Desenhar uma série de 10 quadrados com 20 pixels de lado com preenchimento e sem contorno
Os quadrados devem estar dispostos em fileira horizontal
Os quadrados devem estar separados por um intervalo de 5 pixels
Escolher qualquer matiz para preenchimento (fill) desde que haja variação crescente no brilho

```
Não esquecer o tamanho da janela ex: size (250,30);
Não esquecer o valor máximo do brilho e saturação é 100(%).
```

Processing – Repetição

Utilizando repetição e colorMode (HSB,valor1,valor2,valor3)

Desenhar uma série de 10 quadrados com 20 pixels de lado com preenchimento e sem contorno
Os quadrados devem estar dispostos em fileira horizontal
Os quadrados devem estar separados por um intervalo de 5 pixels
Escolher qualquer matiz para preenchimento (fill) desde que haja variação crescente no brilho

Duas soluções possíveis

```
size(250,30);
noStroke();
colorMode(HSB,360,100,100);
for (int i=0; i<250;i+=25) {
  fill(140,100,i*0.4); // brilho= i multiplicado por 0.4, pois i  varia de 0 a 250
     rect (i,5,20,20);
}
```

```
size(250,30);
noStroke();
int x=0;
colorMode(HSB,360,100,100);
for (int i=0; i<10;i++) {
  fill(140,100,i*10); // brilho= i multiplicado por 10, pois i varia de 0 a 10
     rect (x,5,20,20);
     x+=25;
}
```

Processing – Repetição

A estrutura FOR produz repetições em uma dimensão. Aninhando esta estrutura em outra cria-se uma iteração de duas dimensões.

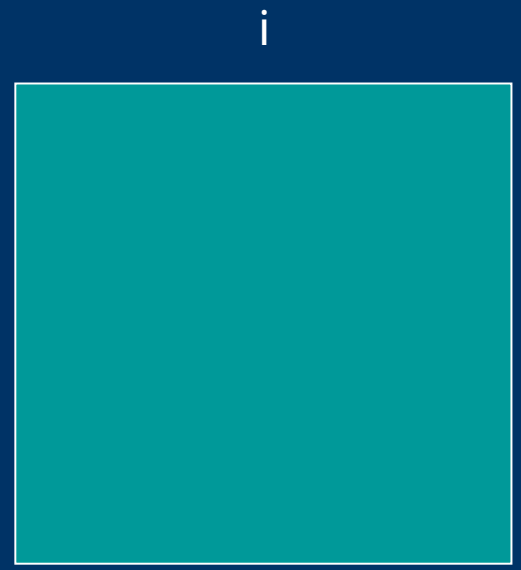
```
//No exemplo temos uma iteração de uma dimensão
//O código desenha uma coluna de 10 pontos
//Isso ocorre porque estamos variando a coordenada y enquanto x permanece constante (=10)

for (int y=10; y<100; y+=10) {
    point(10,y);
}

//No exemplo temos uma iteração de uma dimensão
//O código desenha uma linha de 10 pontos
//Isso ocorre porque estamos variando a coordenada x enquanto y permanece constante (=10)

for (int x=10; x<100; x+=10) {
    point(x,10);
}
```

* ver exemplo na pasta ESTRUTURA - ex.11

```
//No exemplo temos uma iteração de 2 duas dimensões
//O código desenha uma matriz de 10x10 pontos
//Isso ocorre porque dentro de  cada iteração correspondente a uma linha (variação de y)
//está aninhada uma iteração de 10 repetições que posiciona os pontos ao longo
//das colunas (variação do x)

for (int i=10; i<100; i+=10) {

  for (int j=10; j<100; j+=10) {
     point (i,j);
 }

}
```

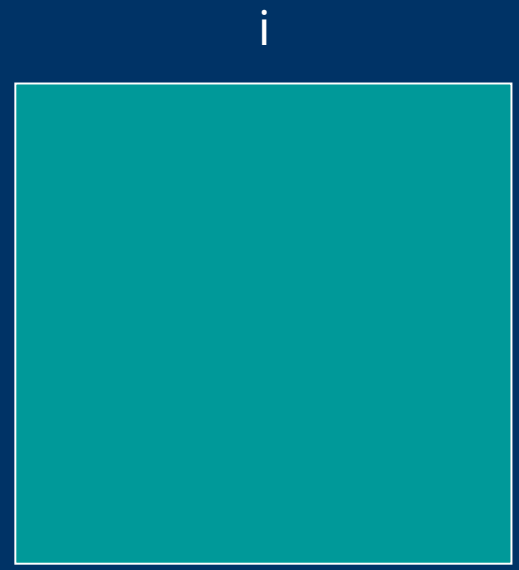
i vale 10

```
for(int i = 10; i <= 100; i += 10) {
  for(int j = 10; j <= 100; j += 10) {
    point(i, j);
  }
}
```

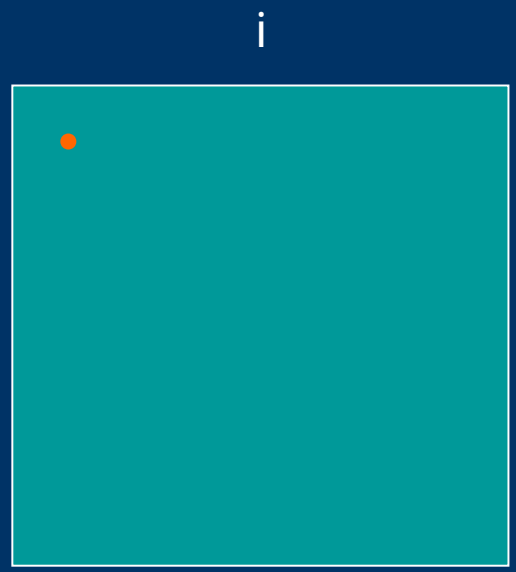


j vale 10

```
for(int i = 10; i <= 100; i += 10) {
  for(int j = 10; j <= 100; j += 10) {
    point(i, j);
  }
}
```



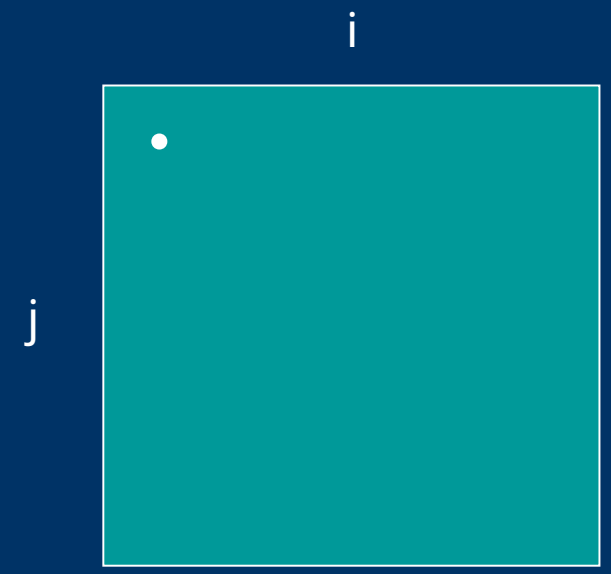
```
for(int i = 10; i <= 100; i += 10) {
  for(int j = 10; j <= 100; j += 10) {
    point(i, j);
  }
}
```



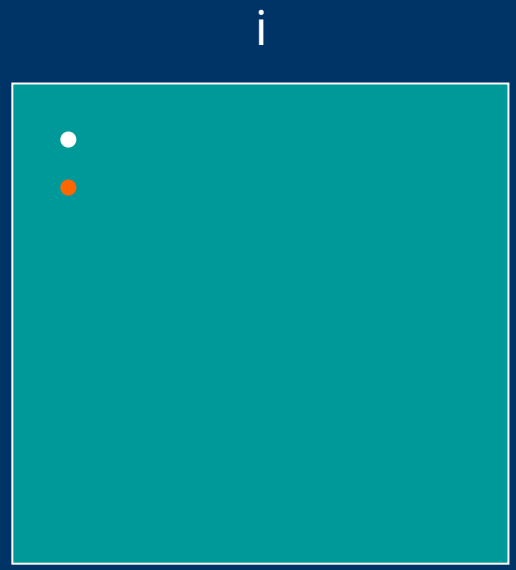
i=10, j=10. Desenha um ponto em (10,10)

```
for(int i = 10; i <= 100; i += 10) {
  for(int j = 10; j <= 100; j += 10) {
    point(i, j);
  }
}
```

Agora estamos na parte interior do laço,
Incrementando j com 10,
verificando se j é menor ou igual a 100.
Não é, então continuamos.



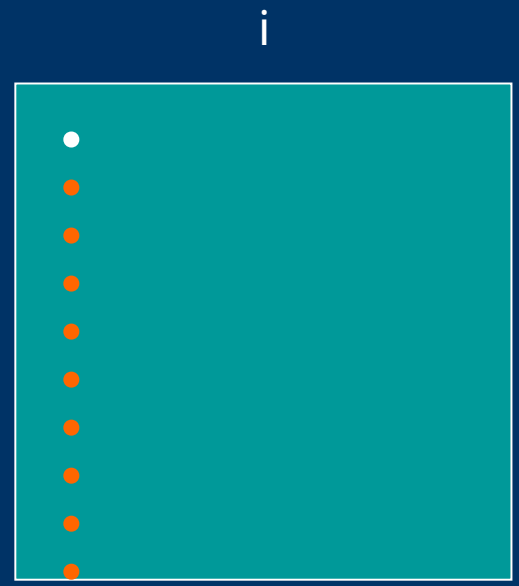
```
for(int i = 10; i <= 100; i += 10) {
  for(int j = 10; j <= 100; j += 10) {
    point(i, j);
  }
}
```



i continua igual a 10, mas j agora é igual a 20.
Desenhamos o novo ponto, e então repetimos o laço.

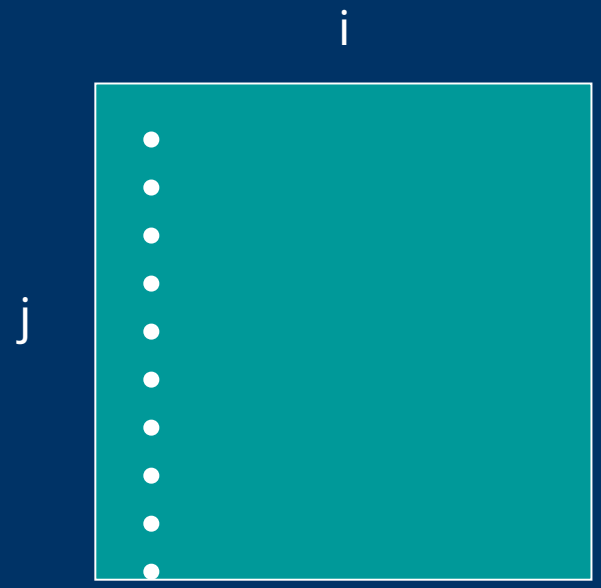
```
for(int i = 10; i <= 100; i += 10) {
  for(int j = 10; j <= 100; j += 10) {
    point(i, j);
  }
}
```

Então, continuamos a repetir este laço de j até que ele passe de 100. E então?



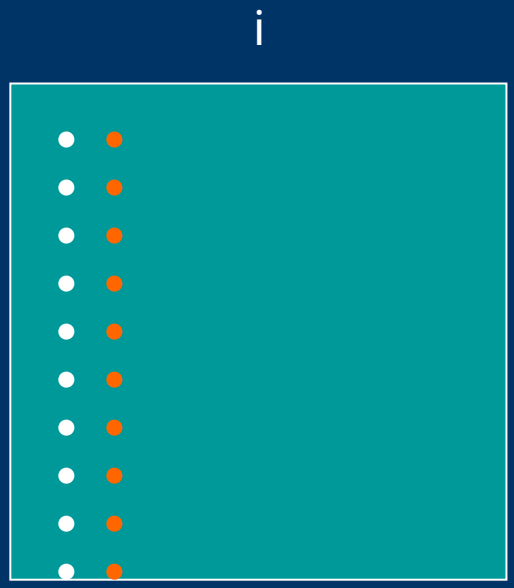
```
for(int i = 10; i <= 100; i += 10) {
  for(int j = 10; j <= 100; j += 10) {
    point(i, j);
  }
}
```

Bem, agora o programa saiu do laço do j,
parece que ainda esta dentro do laço do i,
e incrementa i com 10.



```
for(int i = 10; i <= 100; i += 10) {
  for(int j = 10; j <= 100; j += 10) {
    point(i, j);
  }
}
```

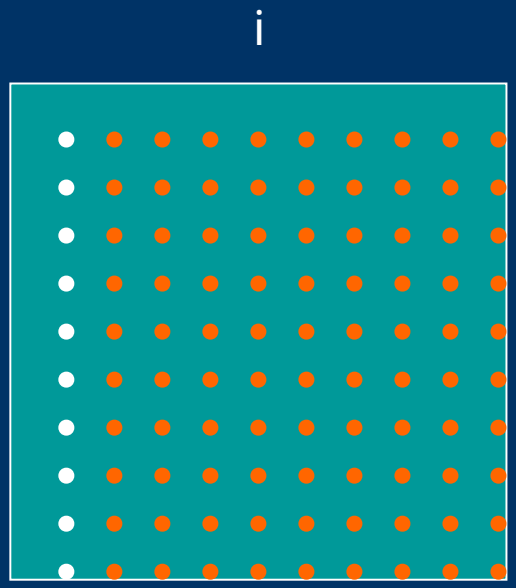
j



O programa atinge o laço de j novamente. Assim sendo, ele repete o desenho dos pontos tudo de novo. Só que desta vez o i vale 20, então a coluna de pontos está mais à direita.

```
for(int i = 10; i <= 100; i += 10) {
  for(int j = 10; j <= 100; j += 10) {
    point(i, j);
  }
}
```

j



O laço de i continua incrementando desta maneira, desenhando colunas de pontos, até passar de 100. O program sai então deste código determinado pelas chaves do laço do i.

Processing – Imagem

O Processing pode carregar imagens GIF, JPEG e PNG. Da mesma forma que números inteiros podem ser armazenados em variáveis do tipo int e true ou false no tipo boolean, as imagens podem ser armazenadas em variáveis do tipo **Pimage**. Antes de ser mostrada na tela, uma imagem deve ser previamente carregada com a função **loadImage()**. A função **image()** mostra a imagem armazenada na variável.

loadImage (nome da imagem)
Pimage variável

image (variável, x,y)
image (variável, x,y,largura, altura)

* ver exemplo na pasta ELEMENTOS

```
// o arquivo da imagem deve estar na pasta "data" do sketch
Pimage img;
img=loadImage("arch.jpg");
image (img,0,0);
```

```
// carregando uma imagem PNG com 8 bits alpha
PImage img;
img = loadImage("arch.png");
background(255);
image(img, 0, 0);
image(img, -20, 0);
```

Processing – Imagem

A função **tint()** pode colorizar uma imagem com tons de cinza ou RGB além de modificar a transparência (alpha).

tint (r, g, b)
tint (r, g, b,alpha)
tint (cinza)
tint (cinza,alpha)

noTint()

* ver exemplo na pasta ELEMENTOS

```
PImage img;                     // declara a variável img como tipo PImage
img = loadImage("arch.jpg");    //  carrega a imagem arch.jpg na variável img
tint(102);                      // tingir de cinza
image(img, 0, 0);               // mostra a imagem no ponto 0,0
noTint();                       // sem tinta
image(img, 50, 0);              // mostra a imagem no ponto 50,0


PImage img;
img = loadImage("arch.jpg");
background(255);
tint(255, 102);                 //alpha é setado para 102 sem mudança na tinta
image(img, 0, 0, 100, 100);
tint(255, 204, 0, 153);         //tinta é alterada para amarelo e alpha 153
image(img, 20, 20, 100, 100);   // apenas uma parte da imagem é tinta
```

Processing – Imagem

Uma vez armazenada numa variável a imagem pode ser exibida a qualquer instante.

* ver exemplo na pasta ELEMENTOS

```
img = loadImage("arch.jpg");
background(255);
tint(255, 51);
for (int i = 0; i < 10; i++) { // desenha a imagem 10 vezes movimentando para a direita
  image(img, i*10, 0);
}
```

Processing – Imagem

A imagem é definida como uma grade de pixels especificados com valores que definem uma cor. Esses valores podem ser lidos com a função **get()**.

get()

* ver exemplo na pasta ELEMENTOS

```
smooth();
strokeWeight(8);
line(0, 0, width, height);
line(0, height, width, 0);
noStroke();
ellipse(18, 50, 16, 16);
// a variável cruz recebe a imagem formada por todos os pixels da janela = get()
PImage cruz = get();
// a imagem contida em cruz é mostrada no ponto 42,30 com 40 pixels de largura e altura
image(cruz, 42, 30, 40, 40);
```

Processing – Imagem

A função **get()** sempre captura **todos** os pixels mostrados na janela de display.
Podemos usar a função **image()** para mostrar partes de uma imagem previamente carregada e capturada com **get()**.

* ver exemplo na pasta ELEMENTOS

```
PImage arvores;
arvores = loadImage("topanga.jpg");// carrega a imagem topanga.jpg na variável arvores
image(arvores, 0, 0);              // mostra no display a imagem contida na variável arvores
PImage crop = get();               // a variável crop contem a captura da janela
tint(0,120,0);                     // escolhe um tint verde
image(crop, 0, 50);                // a imagem contida em crop  é mostrada no ponto 0,50
```

Processing – Imagem

Quando usada com as coordenadas x,y, a função **get()** retorna valores que devem ser atribuídos a uma variável do tipo **color**. Esses valores podem ser usados para modificar a cor de outros pixels ou servir como parâmetro para as funções **fill()** e **stroke()**.

get (x,y)

```
PImage arvores;
arvores = loadImage("topanga.jpg");
noStroke();
image(arvores, 0, 0);
color c = get(20, 30); //a variável c (tipo color) armazena a cor do pixel situado em 20,30
fill(c);               // a cor de preenchimento é setada para a cor armazenada na variável c
rect(20, 30, 40, 40);  // um retangulo é desenhado e preenchido com a cor capturada
```

Processing – Imagem

Toda variável do tipo Pimage possui sua própria função get(). Desta forma, podemos capturar os pixels de uma imagem isoladamente, independente do que é mostrado na janela de display.

get (x,y,largura, altura)

```
PImage arvores;
arvores = loadImage("topanga.jpg");
stroke(255);
strokeWeight(12);
image(arvores, 0, 0);
line(0, 0, width, height);
line(0, height, width, 0);
// a variável arvoresCrop armazena os pixels capturados diretamente da imagem arvores
// os pixels são capturadas a partir do ponto 20,20
// e são armazenada numa area de 60x60 pixels
PImage arvoresCrop = arvores.get(20, 20, 60, 60);
// a imagem capturada é mostrada no ponto 20,20 da janela de display
// note que o desenho da cruz branca não foi capturada
// escolhemos uma cor azul para colorir a imagem recortada
tint (0,0,200);
image(arvoresCrop, 20, 20);
```

* ver exemplo na pasta ELEMENTOS

Processing – Texto

As variáveis do tipo **String** podem armazenar palavras e sentenças envolvidas com aspas (" ");

String variável

```
String m1="Avada ";
String m2="Kedrava";

String magia=m1+m2;

println(magia); // imprime a combinação m1+m2 = "Avada Kedrava"
```

* ver exemplo na pasta ELEMENTOS

Processing – Texto

Uma fonte de texto deve ser convertida para o o formato **VLW** para poder ser visualizada no Processing.
No menu **TOOLS** existe a ferrementa **Create Font** que converte qualquer fonte do sistema e salva na pasta **data** do sketch. Após a conversão, a fonte pode ser carregada com a função **loadFont()**. A fonte carregada é atribuida a uma variável do tipo **Pfont** e pode então ser utilizada desde que seja o parâmetro para a função **textFont()**.
A função **text()** é responsável pela exibição de qualquer caracter no display.

PFont `variável`

loadFont `(arquivo-da-fonte.vlw)`

textFont `(variável do tipo PFont)`

text `(dado, x, y)-- o parametro dado pode ser String,int ou float.`
text `(string, x, y, largura, altura)`

```
PFont ft;                          // declara  a variável ft do tipo PFont
ft = loadFont("Ziggurat-32.vlw"); // carrega a fonte Ziggurat na variável ft
textFont(ft);                      // determina a fonte armazenada em ft para a tipografia
fill(0);
text("LAX", 0, 40);                // escreve "LAX" coordenada (0,40)
text("AMS", 0, 70);                // escreve "AMS" coordenada (0,70)
text("FRA", 0, 100);               // escreve "FRA" coordenada (0,100)
```

* ver exemplo na pasta ELEMENTOS

Processing – Texto

A função fill() é usada tambem para o texto, determinando a tonalidade de cinza ou cor e transparência.

```
PFont font;
font = loadFont("Ziggurat-32.vlw");
textFont(font);
fill(0);
text(19, 0, 36); // escreve 19 na coordenada (0,36)
text(72, 0, 70); // escreve 72 na coordenada (0,70)
text('R', 62, 70); // escreve R na coordenada (62,70)
```

* ver exemplo na pasta ELEMENTOS EX.04

```
PFont font;
font = loadFont("Ziggurat-72.vlw");
textFont(font);
fill(0, 160); // preto com baixa opacidade
text("1", 0, 80);
text("2", 15, 80);
text("3", 30, 80);
text("4", 45, 80);
text("5", 60, 80);
```

Processing – Texto

Atributos do Texto

textSize (tamanho)

textLeading (distancia entre linhas)

textAlign (MODO) – modo pode ser LEFT, CENTER ou RIGHT

textWidth (string)-- retorna a largura do caracter ou string

```
// reduzindo uma fonte de 32 pixels
PFont font;
font = loadFont("Ziggurat-32.vlw");
textFont(font);
fill(0);
text("LNZ", 0, 40); // grande
textSize(18);
text("STN", 0, 75); // médio
textSize(12);
text("BOS", 0, 100); // pequeno
```

* ver exemplo na pasta ELEMENTOS – EX 05

Processing – Texto

```
// usando espacamento entre linhas
PFont font;
font = loadFont("Ziggurat-12.vlw");
textFont(font);
String lines = "L1 L2 L3";
textLeading(10);
fill(0);
text(lines, 5, 15, 30, 100);
textLeading(20);
text(lines, 36, 15, 30, 100);
textLeading(30);
text(lines, 68, 15, 30, 100);
```

```
// usando alinhamentos
PFont font;
font = loadFont("Ziggurat-12.vlw");
textFont(font);
line(50, 0, 50, 100);
fill(0);
textAlign(LEFT);
text("Left", 50, 20);
textAlign(RIGHT);
text("Right", 50, 40);
textAlign(CENTER);
text("Center", 50, 80);
```

* ver exemplo na pasta ELEMENTOS EX.06 EX.07

Processing – Randomicidade

A função **random()** é usada para criar valores imprevisíveis dentro de um espectro especificado por seus parâmetros. Quando um parâmetro é **passado,** a função retorna um valor que pode variar de zero (não incluso) até o valor do parâmetro. Quando dois parâmetros são utilizados a função retorna um valor aleatório situado entre os dois parâmetros. Os valores retornados são do tipo **float** e podem ser convertidos para valores inteiros através da função **int** (valor).

```
random (valor)
random (limite inferior, limite superior)

// desenha 5 linhas com cordenadas y1 e y2 randomicas
smooth();
strokeWeight(10);
stroke(0, 130);
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
line(0, random(100), 100, random(100));
```

Processing – Randomicidade

```
// repetição de 800 passos
// para cada passo a variável r pode valer de 1.0 a 10.0
// a espessura da linha varia randomicamente com r
// a variável offset pode assumir valores randomicos entre de 80.0 a 800.0

size (500,400);
background(0);
smooth();
stroke(255, 30);

for (int i = 0; i < 800; i++) {
  float r = random(10);
  //strokeWeight(r);
  float offset = r * 80.0;
  line(i, 100,0,i+offset);        // linhas partem do eixo horiz (y=100)

}
```

* ver exemplo na pasta ESTRUTURA

Processing – Transformação

O sistema de coordenadas (x,y) visto em exemplos anteriores pode ser transformado por funções de translação, rotação e escalonamento. Desta forma as figuras podem ser desenhadas na janela de display em posição, orientação e tamanhos diferentes.

A função **translate()** move a origem do canto superior esquerdo da tela para outra posição. Ela possui dois parâmetros correspondentes ao deslocamento nos eixos x e y.

translate (x,y)

```
// o mesmo retangulo é desenhado, mas apenas o segundo é afetado
// pela translação pois é desenhado depois

rect (0,5,70,30);
translate (10,30);
rect (0,5,70,30);


// um numero negativo é utilizado como parametro
// movendo as coordenadas na direção oposta

rect (0,5,70,30);
translate (10,-10);
rect (0,5,70,30);


// efeito aditivo
// o ultimo retangulo recebe o acumulo das 2 translações
rect(0, 5, 70, 30);
translate(10, 30); // desloca 10 pixels para direita e trinta para baixo
rect(0, 5, 70, 30);
translate(10, 30); // desloca tudo novamente
rect(0, 5, 70, 30); // 20 pixels para direita e sessenta pixels para baixo
```

* ver exemplos na pasta TRANSFORMAÇÃO

Processing – Transformação

A **ordem** na qual as funções de transformação são executadas pode mudar radicalmente o modo como as coordenadas são afetadas. A função **rotate()** rotaciona o sistema de coordenadas. Assim as formas podem ser desenhadas numa angulação determinada. Existe apenas um parâmetro que determina a quantidade de rotação (ângulo). A função de rotação assume que o parâmetro é especificado em **radianos**.
As figuras são rotacionadas em torno da **posição relativa à origem (0,0)**, e números positivos fazem a rotação no sentido horário. Assim como todas as transformações, o efeito da rotação tambem é **cumulativo**.

rotate (angulo em radianos)

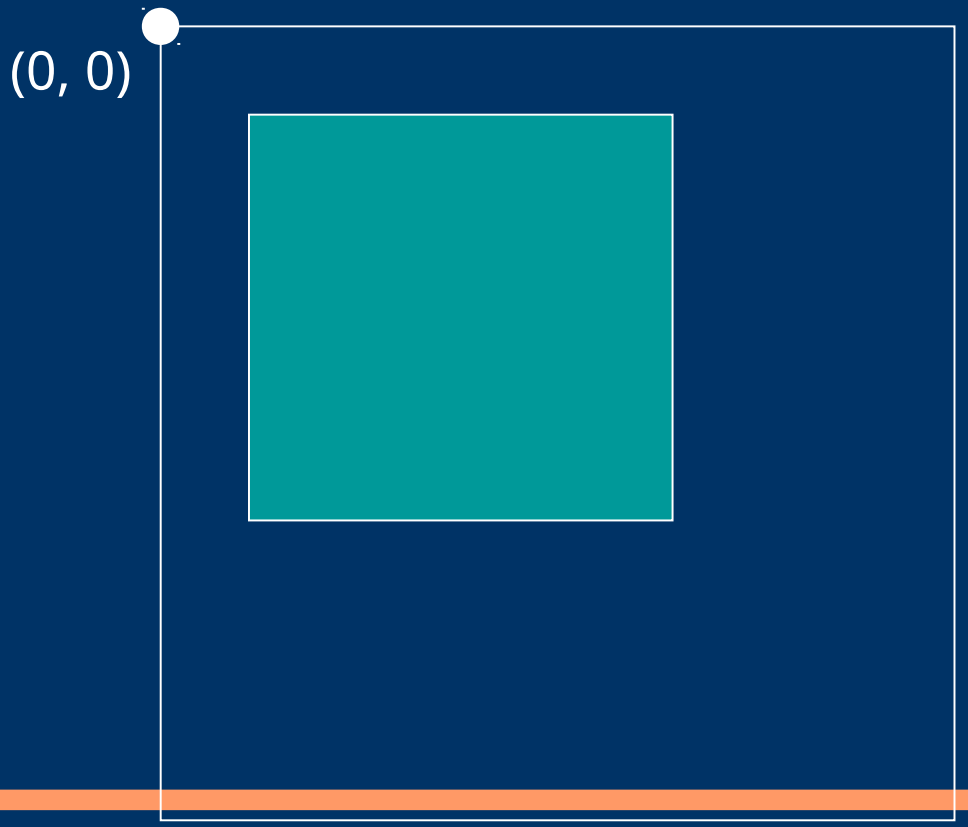
```
smooth();
rect (55,0,30,45);
rotate (PI/8);
rect (55,0,30,45);


// perceba o efeito aditivo da rotação
// o retangulo rotaciona em torno do ponto (0,0) no sentido antihorário
smooth();
rect(10, 60, 70, 20);
rotate(-PI/16);
rect(10, 60, 70, 20);
rotate(-PI/8);
rect(10, 60, 70, 20);
```

* ver exemplos na pasta TRANSFORMAÇÃO

Rotação

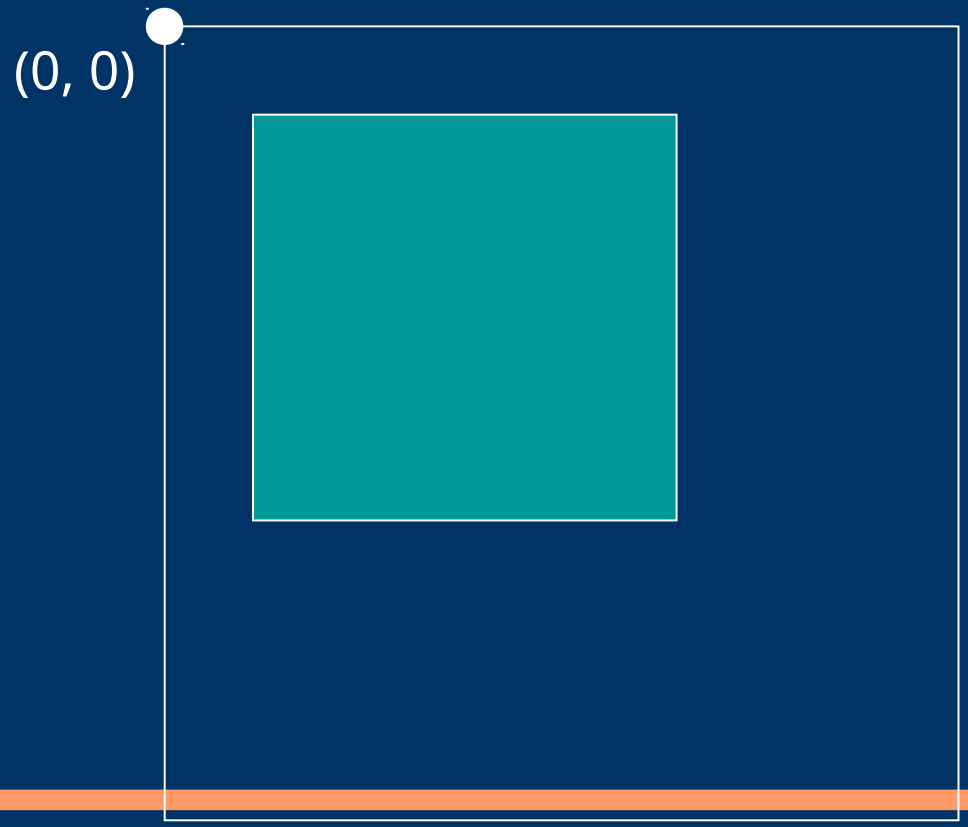
- Vamos ver um exemplo sem rotação

rect(10, 10, 50, 50);



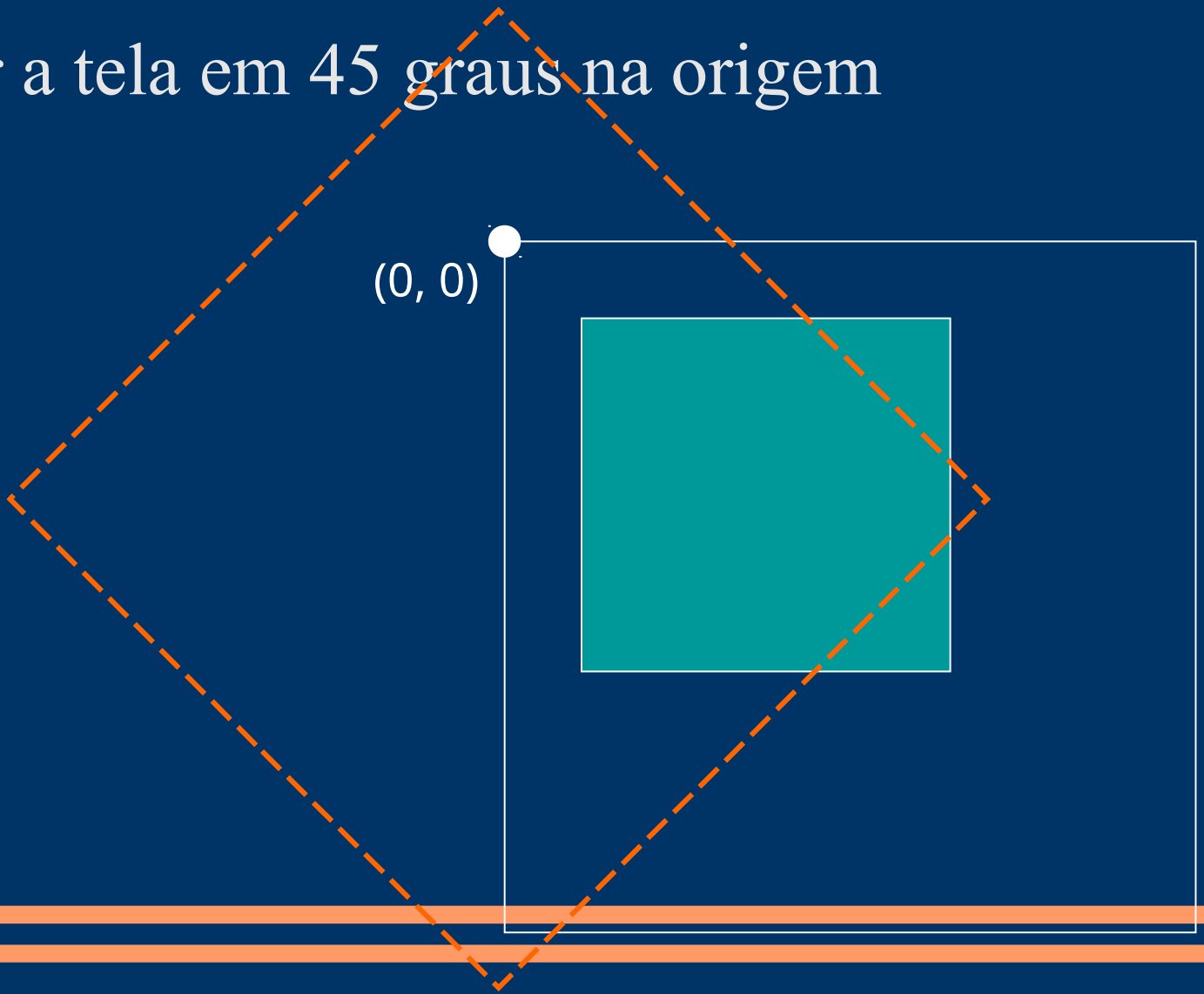
- Rotação

float angle = radians(45);



- Rotacionar a tela em 45 graus na origem

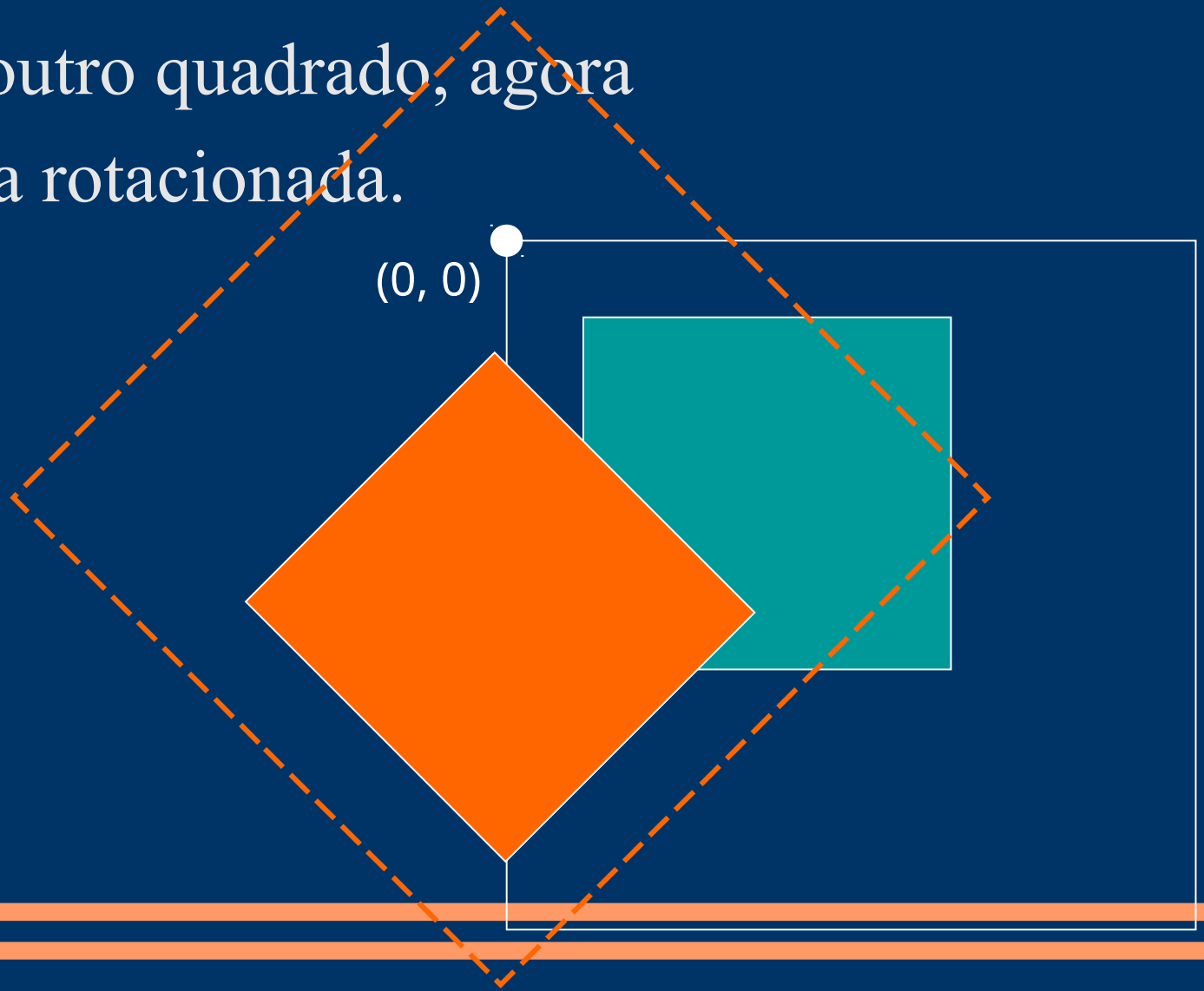
rotate(angle);



Rotação

- Desenhar outro quadrado, agora relativo a tela rotacionada.

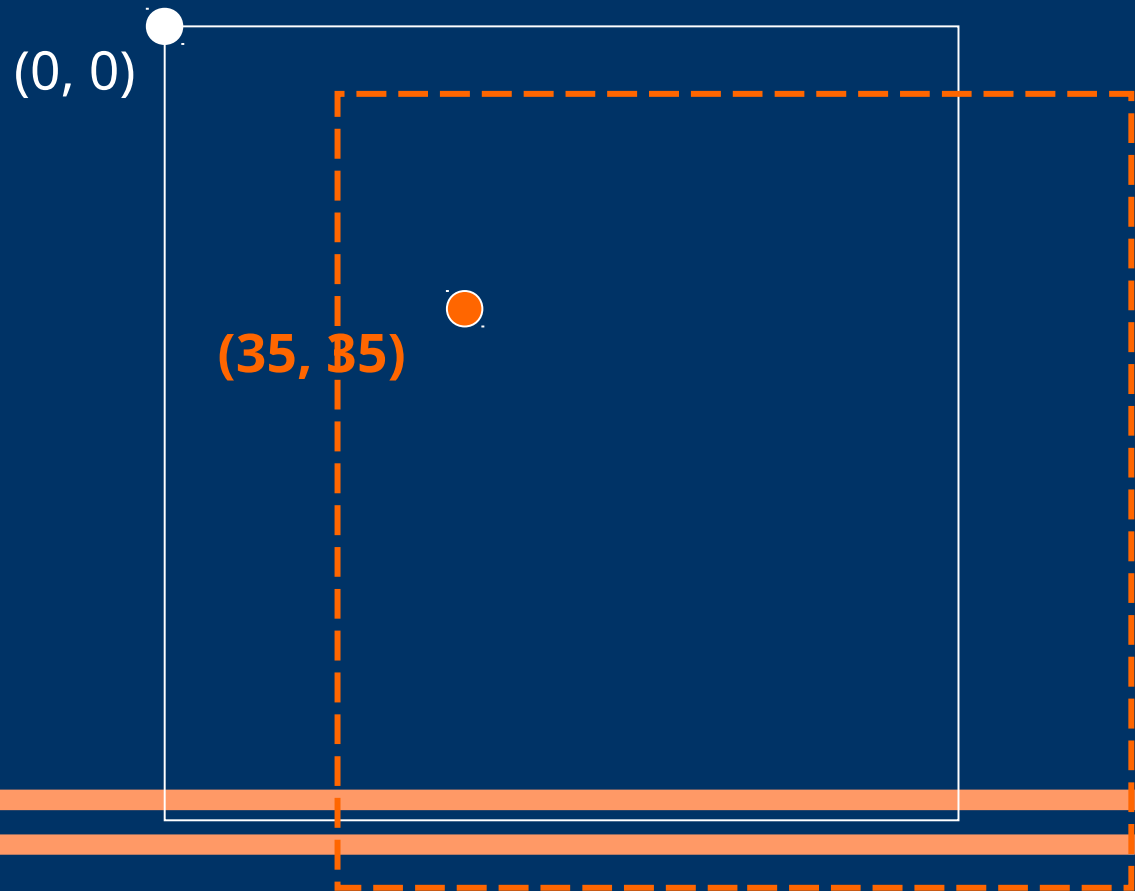
rect(10, 10, 50, 50);



Rotação

- Começar com o ponto de rotação

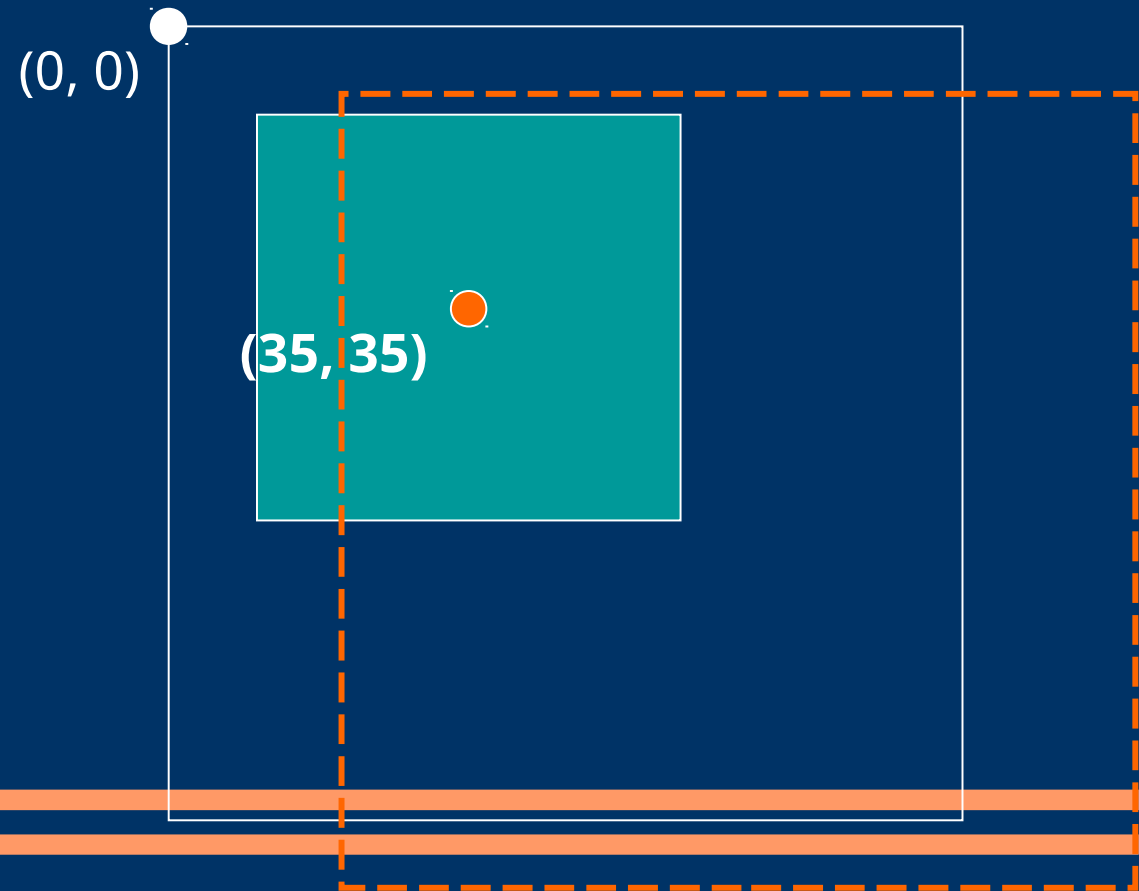
translate(35, 35);



Rotação

- Agora desenhar o quadrado com este ponto como seu centro.

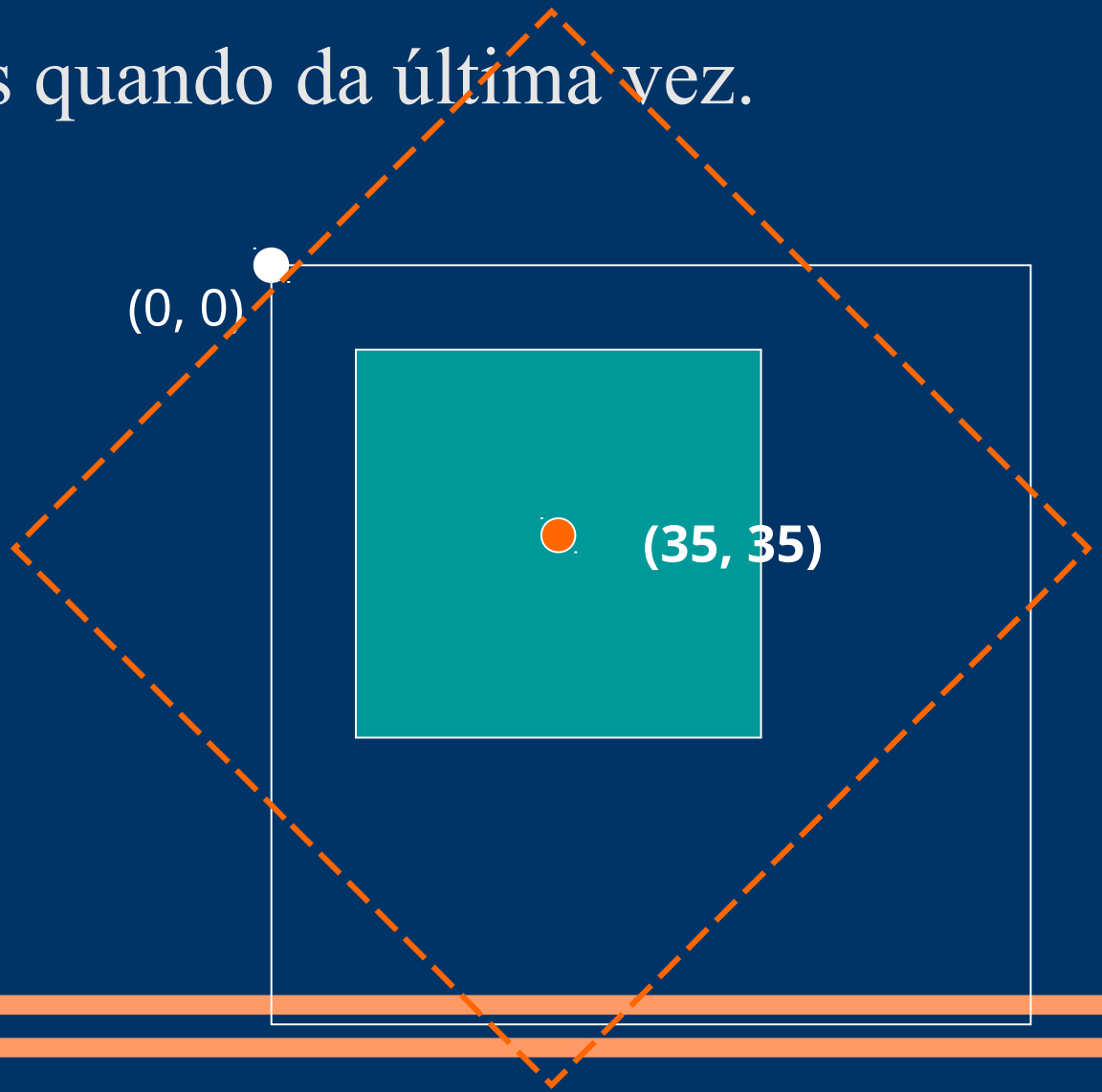
```
rect(-25, -25, 50, 50);
```



Rotação

- Então rotacionamos quando da última vez.

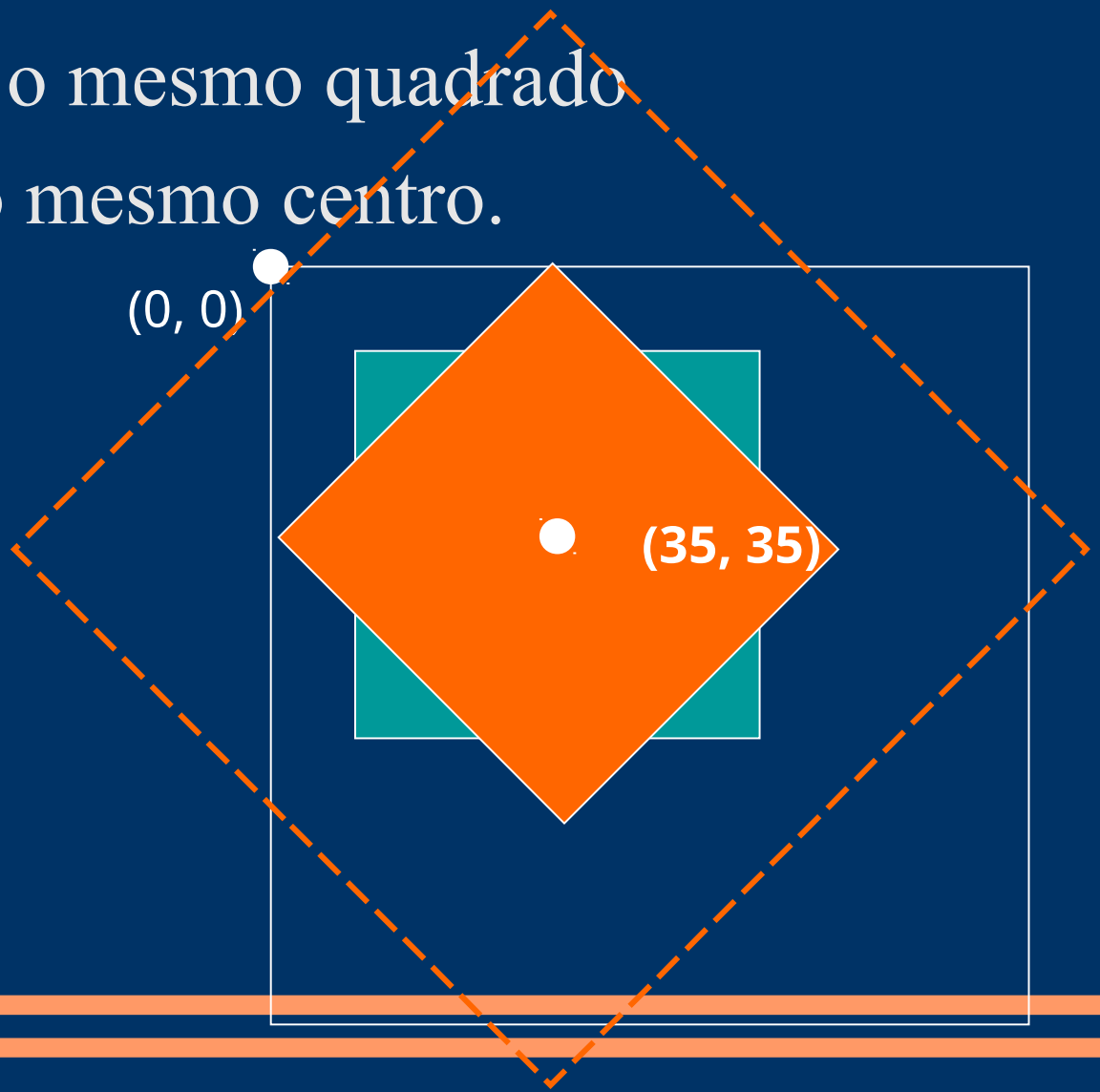
```
float angle = radians(45);
rotate(angle);
```



Rotação

- Agora desenhamos o mesmo quadrado como antes, ele terá o mesmo centro.

```
float angle = radians(45);
rotate(angle);
```



Processing – Transformação

A função **scale()** aumenta o sistema de coordenadas. Dessa maneira as figuras podem ser desenhadas maiores. A versão com um parâmetro aumenta a figura em todas as dimensões e a versão com dois parâmetros pode auamentar a figura nos eixos x e y separadamente. Os parâmetros da escala são definidos em termos de percentagem. Por exemplo, 2.0 corresponde a 200%, 1.5 para 150%.

scale (tamanho)
scale (tamanhox,tamanhoy)

```
smooth();
ellipse(32, 32, 30, 30);
scale(1.8);
ellipse(32, 32, 30, 30);


// a espessura do contorno da figura tambem é afetada pela escala
// para conservar o contorno proporcional dividimos o parametro de strokeWeight()
// pelo valor da escala.

float s = 1.8;
smooth();
ellipse(32, 32, 30, 30);
scale(s);
strokeWeight(1.0 / s);
ellipse(32, 32, 30, 30);

// efeito aditivo da escala
rect(10, 20, 70, 20);
scale(1.7);
rect(10, 20, 70, 20);
scale(1.7);
rect(10, 20, 70, 20);
```

* ver exemplos na pasta TRANSFORMAÇÃO

Processing – Transformação

As funções de transformação podem ser combinadas antes de uma figura ser desenhada.

Além disso, existe uma função que possiblita o armazenamento do sistema de coordenadas atual (**pushMatrix()**) e uma outra (**popMatrix( )**) para a posterior recuperação do estado original do sistema armazenado. A função pushMatrix() não pode ser usada sem popMatrix() e vice e versa.

pushMatrix ()

popMatrix ()

```
size (200,200);
Fill (0,30);

pushMatrix();          // a função pushMatrix inicia um sistema de coordenadas

translate (100,100); // translação do sistema para o centro da janela (100,100)
rotate (radians(45));// rotaciona o sistema em 45 graus
rect (0,0,100,100);  // desenha um retangulo no ponto 0,0 deste sistema
rectMode(CENTER);    // o proximo retangulo a ser desenhado tera seu ponto de registro centralizado
rect (0,0,100,100);  // o retangulo é desenhado no ponto 0,0 da coordenada

popMatrix();           // o sistema tranformado por translate e rotate é  retirado da pilha

rectMode (CORNER);   // o proximo retangulo tera seu ponto de registro no canto superir esquerdo
fill (125);
rect (0,0,100,100);  // o retangulo  é desenhado no ponto 0,0 do sistema de coordenadas recuperado
```

* ver exemplo na pasta TRANSFORMAÇÃO

Processing – Continuidade

Processing – Continuidade

As instruções dos programas vistos até agora são executadas apenas uma única vez.
Programas que executam animações ou que respondam às informações em tempo real devem rodar continuamente.
No Processing a função **draw()** é responsável pela execução contínua de blocos de código. Quando a função é finalizada, outro frame é desenhado na janela de display e o bloco de códigos é executado novamente desde a primeira linha. O padrão de execução é de **60 fps** (frames-por-segundo). A função **frameRate ()** pode contolar a velocidade de frames por segundo e **frameCount** retorna o número de frames mostrados desde o inicio.

draw ()

frameRate (fps)

frameCount é uma variável.

```
float y = 0.0;              // inicializa a variável y com o valor 0.0
void draw() {               // função draw () responsável pela repetição do código entre { }
  frameRate(30);            // seta o fps para 30
  line(0, y, 100, y);       // desenha uma linha horizontal variando valor de y1 e y2
  y += 0.5;                 // incrementa continuamente o valor da variável y
}


// mesmo do exemplo anterior
// mas antes de desenhar a linha redesenha também o fundo (background (204))

float y = 0.0;
void draw() {
  frameRate(30);
  background(204);
  Y += 0.5;
  line(0, y, 100, y);
}
```

Processing – Continuidade

Dentro da execução contínua do código, podemos incrementar variáveis e utilizar seus valores como parâmetros para outras funções. Estruturas decisórias (IF...ELSE) podem examinar e atualizar constantemente o estado destes elementos.

```
float y = 0.0;
void draw() {
  frameRate(30);
  background(y * 2.5); // dentro do bloco a tonalidade do background é modificada  = y*2.5
  y+=0.5;
  line(0, y, 100, y);
}
```

```
// segue a mesma estrutura do exemplo anterior
// após alguns segundos a linha atinge  o limite inferior da janela
// neste caso, a estrutura IF inicializa o valor da variável (y=0)
// desta forma a linha pode ser redesenhada no topo da janela novamente

float y = 0.0;
void draw() {
  frameRate(30);
  background(y * 2.5);
  y+=0.5;
  line(0, y, 100, y);
  if (y > 100) {
    y = 0;
  }
}
```

* ver exemplos na pasta ESTRUTURA

Processing – Continuidade

Algumas funções devem ser executadas apenas uma única vez. Instruções como determinar o tamanho da janela, por exemplo, podem ser executadas em apenas 1 frame pela função **setup().** As variáveis como aquelas que mudam de valor na iteração da função **draw()** devem ser declaradas fora de **setup()** ou **draw()**.

A função **noLoop()** interrompe a continuidade de **draw()** e pode ser usado como uma alternativa para o desenho de um frame apenas.

setup()

noLoop()

```
// dentro da função , a definição da janela, antialias e preenchimento são executados apenas
// uma vez
// note que a função noLoop() tambem é executada , impedindo que draw() repita continuamente
// o desenho da elipse
// neste caso, frameCount retornará o valor 1 (frame)


void setup() {
  size(100, 100);
  smooth();
  fill(0);
  noLoop();
}

void draw() {
  ellipse(50, 50, 66, 66);
  println(frameCount);
}
```

* ver exemplos na pasta ESTRUTURA

Processing – Continuidade

```
// utilizando texto

PFont font;

void setup() {
  size(100, 100);
  font = loadFont("Eureka-48.vlw");
  textFont(font);
  noStroke();
}

void draw() {
  fill(204, 24);
  rect(0, 0, width, height);
  fill(0);
  text("flicker", random(-100, 100), random(-20, 120));
}
```

* ver exemplos na pasta ESTRUTURA

Processing – Continuidade

Quando setup() e draw() são utilizados , torna-se necessário um planejamento sobre onde declarar variáveis. O local de declaração de uma variável determina seu **escopo** – **onde ela pode ser acessada pelo programa**. A regra para a determinação do **escopo**: variáveis declaradas dentro de um bloco só podem ser acessadas dentro do bloco. Variáveis declaradas no mesmo nível que setup() ou draw() podem ser acessada de qualquer local do programa. Variáveis declaradas dentro de setup() ou dentro de draw() só podem ser acessadas dentro das próprias funções.

Escopo das variáveis

```
int d = 51;                 // A variável "d" pode ser acessada de qualquer local

void setup() {
  size(100, 100);
  int val = d * 2;        // A variável "val" é local, só pode ser acessada dentro de setup
  fill(val);
}

void draw() {
  int y = 60;            // A variável "y" é local, só pode ser acessada dentro de draw
  line(0, y, d, y);
  y -= 25;
  line(0, y, d, y);
}


void draw() {
  int d = 80;
  if (d > 50) {
    int x = 10;             // Essa variável (x) só pode ser usada dentro do bloco IF
    line(x, 40, x+d, 40);
  }
  line(0, 50, d, 50);
  line(x, 60, x+d, 60);    // ERRO, x é  acessada fora do bloco
}
```

Processing – Continuidade

Utilizando setup() e draw() rotacionar continuamente um quadrado no intervalo de 5 graus.

Processing – Continuidade

Utilizando setup() e draw() rotacionar continuamente um quadrado no intervalo de 5 graus.

Solução

```
float angle = radians(45);
void setup() {
  size(100, 100);
  rectMode(CENTER);
  //frameRate(10);
  smooth();
}

void draw() {
  background(204);
  translate(50, 50);
  angle+=radians(5);
  rotate(angle);
 rect(0, 0, 40, 40);
}
```

Processing – Interação: Mouse

No Processing as variáveis **mouseX** e **mouseY** armazenam as coordenadas x,y do cursor relativas à origem no canto superior esquerdo da janela de display. A posição do mouse pode ser usada para controlar a localização de elementos visuais. Adicionar ou subtrair valores da posição do mouse gera relações constantes, equanto que multiplicar e dividir esses valores relações visuais mais dinâmicas.

mouseX
mouseY

```
//podemos adicionar, subtrair, multiplicar ou dividir os valores correspondentes
// a mouseX e mouseY para obter tipos diferentes de comportamentos dos objetos visuais
// retire as barras de comentário (//) para verificar cada comportamento

void setup() {
size(100, 100);
  smooth();
  noStroke();
}
void draw() {
  background(126);
  //ellipse(mouseX, mouseY, 33, 33); // segue a posição do mouse
  //ellipse(mouseX, 16, 33, 33);
  //ellipse(mouseX+20, 50, 33, 33);
  //ellipse(mouseX-20, 84, 33, 33);
  //ellipse(mouseX, 16, 33, 33);
  //ellipse(mouseX/2, 50, 33, 33);
  //ellipse(mouseX*2, 84, 33, 33);
}
```

* ver exemplos na pasta INTERAÇÃO

Processing – Interação: Mouse

Para inverter o valor do mouse, simplesmente subtraia o valor de **mouseX** da largura (**width**) da janela e subtraia o valor de **mouseY** da altura (**height**).

```
void setup() {
  size(100, 100);
  noStroke();
  smooth();
}

void draw() {
  float x = mouseX;
  float y = mouseY;
  float ix = width - mouseX;          // ix=inverte o valor de x
  float iy = mouseY - height;         // iy=inverte o valor de y
  background(126);
  fill(255, 150);
  ellipse(x, height/2, y, y);         // posicao, altura e largura variam proporcionalmente
  fill(0, 159);
  ellipse(ix, height/2, iy, iy);      // posição, altura e largura variam inversamente
```

* ver exemplos na pasta INTERAÇÃO

Processing – Interação: Mouse

As variáveis **pmouseX** e **pmouseY** guardam os valores do mouse do **frame anterior**. Se o mouse não se mover, os valores continuam o mesmo, mas se o mouse estiver se movendo rapidamente estes valores devem sofrer grandes variações.

pmouseX
pmouseY

```
// desenha uma linha entre a posição atual do mouse e a posição prévia
void setup() {
  size(100, 100);
  strokeWeight(8);
  smooth();
}

void draw() {
  background(204);
  line(mouseX, mouseY, pmouseX, pmouseY);
}
```

* ver exemplos na pasta INTERAÇÃO

Processing – Interação: Mouse

O estado de um botão e a posição do mouse juntos permitem a realização de várias ações. A variável **mousePressed** é verdadeira (true) se algum botão é pressionado e falso (false) se nenhum botão é pressionado. A variável **mouseButton** é LEFT, CENTER ou RIGHT dependendo do botão pressionado. A variável mousePressed É revertida para false logo que o botão é solto, mas a variável mouseButton retem o seu valor até que um botão diferente for pressionado.

mousePressed
mouseButton

```
// seta o preenchimento do quadrado para preto quando o botão  esquerdo for pressionado
// e para branco quando  o botão direito for pressionado

void setup() {
  size(100, 100);
}

void draw() {
  if (mouseButton == LEFT) {
    fill(0); // preto
  } else if (mouseButton == RIGHT) {
    fill(255); // branco
  } else {
    fill(126); // cinza
  }
  rect(25, 25, 50, 50);
}
```

* ver exemplo na pasta INTERAÇÃO

Processing – Interação: Mouse

```
// desenha linhas com diferentes valores de cinza quando o botão do mouse é pressionado ou não

void setup() {
  size(100, 100);
}

void draw() {
  if (mousePressed == true) {     // Se o mouse é pressionado
    stroke(255);                    // seta o contorno para branco
  } else {                        // senão
    stroke(0);                      // seta o controle para preto
  }
  line(mouseX, mouseY, pmouseX, pmouseY);
}
```

* ver exemplo na pasta INTERAÇÃO

Processing – Interação: Mouse

Com a estrutura FOR podemos criar desenhos mais complexos com apenas algumas linhas de código.

```
void setup() {
  size(100, 100);
}

void draw() {
  for (int i = -14; i <= 14; i += 2) {
    point(mouseX+i, mouseY);
  }
```

* ver exemplo na pasta INTERAÇÃO

```
void setup() {
  size(100, 100);
  noStroke();
  fill(255, 40);
  background(0);
}

void draw() {
  if (mousePressed == true) {// mudança de cor quando o mouse é pressionado
    fill(247,10,10, 26);
  } else {
    fill(216,94,94,26);
  }
  // 6 elipses são desenhadas
  //são posicionadas na mesma posição do mouse
  //mas variam na posição x, largura e altura
  for (int i = 0; i < 6; i++) {
    ellipse(mouseX + i*i, mouseY, i, i);
  }
}
```

Processing – Interação: Mouse

Imagens também podem ser usadas como ferramentas de desenho. Se a imagem é posicionada em relação ao cursor emcada frame, seus pixels podem ser usados para criar composições complexas.

```
// desenhando com imagem  transparente

PImage alphaImg;

void setup() {
  size(100, 100);
  // esta imagem é parcialmente transparente
  alphaImg = loadImage("alphaArch.png");
}

void draw() {
  int ix = mouseX - alphaImg.width/2; //a imagem é centralizada de acordo com mouse
  int iy = mouseY - alphaImg.height/2;
  image(alphaImg, ix, iy); //
}
```

* ver exemplo na pasta INTERAÇÃO

Processing – Movimento

```
int direction = 1;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  ellipseMode(RADIUS);
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  ellipse(33, y, radius, radius);
  y += speed * direction;
  if ((y > height-radius) || (y < radius)) {
    direction = -direction;
  }
}
```

Processing – Movimento

```
float y = 50.0;
float speed = 1.0;
float radius = 15.0;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
  ellipseMode(RADIUS);
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  pushMatrix();
  translate(0, y);

  // afetado pelo primeiro translate()
  ellipse(33, 0, radius, radius);
  translate(0, y);

  // afetado pelo primeiro e segundo translate()
  ellipse(66, 0, radius, radius);
  popMatrix();

  // nenhum  translate()
  ellipse(99, 50, radius, radius);
  y = y + speed;
  if (y > height + radius) {
    y = -radius;
  }
}
```

Processing – Movimento

```
float angle = 0.0;

void setup() {
  size(100, 100);
  smooth();
  noStroke();
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  fill(255);
  angle = angle + 0.02;
  translate(70, 40);
  rotate(angle);
  rect(-30, -30, 60, 60);
}
```