

Utilização da biblioteca

CanvasLib

Pedro Pereira

Outubro de 2020

Introdução

A biblioteca *CanvasLib* destina-se a dar suporte pedagógico na unidade curricular de Programação da Licenciatura em Engenharia Informática e de Computadores do ISEL.

Esta biblioteca permite desenvolver em *Kotlin* software com interface gráfica usando um conjunto reduzido de operações de desenho (segmentos, círculos, retângulos, arcos e texto), de execução temporizada e de recolha de estímulos do utilizador através do rato e do teclado.

O código produzido pode ser executado na JVM (*Java Virtual Machine*) ou no *browser* no contexto de uma página.

Os restantes capítulos deste documento descrevem as operações disponíveis na biblioteca e a forma de utilizar a *CanvasLib* para usar no contexto de projetos desenvolvidos no ambiente integrado *IntelliJ* para executar na JVM.

Projeto *IntelliJ*

O *IntelliJ* é um IDE (*Integrated Development Environment*) para desenvolver código em *Kotlin*, Java e outras linguagens. Esta descrição assume que é utilizada a versão 2020.2.3 do *IntelliJ IDEA*, ou versão posterior.

Para usar a *CanvasLib* deve ser criado um projeto seguindo os passos:

1. Criar um projeto selecionando a opção “New Project” na janela de *Welcome*, ou escolhendo no menu da janela principal as opções “File > New > Project”.
2. Na janela *New Project* selecionar o tipo de projeto como “Kotlin”.

Opção 1 : Caso tenha hipótese de escolher “Project template”, faça as seguintes seleções:

- a. em “Project template:” a opção “Console Application” na secção “JVM”
- b. em “Build System:” a opção “Gradle Kotlin”
- c. em “Project JDK:” a versão do JDK mais recente que esteja instalada, por exemplo: 13
- d. em “Location:” escolher a pasta raiz onde ficará o projeto
- e. em “Name:” editar o nome do projeto, que é recomendável que seja igual ao último nome do caminho indicado em “Location:”
- f. prima o botão “Next”, que passa para uma segunda janela onde deve continuar a selecionar:
- g. em “Template:” a opção “None”
- h. em “Test Framework:” a opção “None”
- i. em “Target JVM version:” a versão mais atual da JVM disponível, por exemplo: 13
- j. prima o botão “Finish”.

Opção 2 : Caso não tenha hipótese de escolher “Project template”, faça as seguintes seleções:

- a. seleccione o tipo de projeto como “Gradle” (e não “Kotlin”)
- b. em “Project SDK:” a versão do JDK mais recente que esteja instalada, por exemplo: 13
- c. ativar esta opção “Kotlin DSL build script”
- d. em “Additional Libraries and Frameworks:” desativar “Java” e ativar “Kotlin/JVM”
- e. prima o botão “Next”, que passa para uma segunda janela onde deve continuar a selecionar:
- f. em “Location:” escolher a pasta raiz onde ficará o projeto
- g. em “Name:” editar o nome do projeto, que é recomendável que seja igual ao último nome
- h. prima o botão “Finish”.

Depois de terminado todo o processo da criação do projeto, deve criar uma pasta “libs” e copiar para esta os ficheiros jar de CanvasLib, seguindo os passos:

1. Criar a pasta “libs” na raiz do projeto:
 - a. seleccionar a raiz do projeto (o nome do projeto na vista “Project”)
 - b. escolher no menu as opções “File” > “New” > “Directory”
 - c. editar o nome para “libs” e terminar com <enter>
2. Copiar os ficheiros “CanvasLib-jvm-1.0.0.jar” e “CanvasLib-jvm-1.0.0-sources.jar” para a pasta “libs”. Os ficheiros estão disponíveis nos “Recursos” da página de Programação no moodle. Para tal deve:
 - a. usar o browser para fazer download dos ficheiros
 - b. fazer a cópia dos dois ficheiros para a pasta “libs”.

O ficheiro “build.gradle.kts” do projeto, que configura o processo de *build*, deve ser editado manualmente para adicionar a pasta “libs” como um repositório adicional e indicar a dependência de *CanvasLib*, ficando o ficheiro “build.gradle.kts” com o seguinte aspeto:

```
...
repositories {
    ...
    flatDir { dirs("libs") } // Adicionar esta linha
}

dependencies { // Adicionar esta secção “dependencies” caso não exista
    ...
    implementation("pt.isel:CanvasLib-jvm:1.0.0") // Adicionar esta linha
}
...
```

Depois de editar o ficheiro “build.gradle.kts” é necessário indicar ao IDE que o processo de *build* foi modificado, seleccionando o botão que apareceu no canto superior direito na zona de edição ou usando a combinação de teclas (Ctrl + Shift + O).

Criação do Canvas

Na biblioteca *CanvasLib* está definido o tipo **Canvas**, que corresponde à área de desenho.

Cada objeto do tipo **Canvas** corresponde a uma área de desenho que aparecerá na janela (no caso da JVM) ou na página (no caso do browser). Esta biblioteca permite a existência de vários objetos do tipo **Canvas**, mas os exemplos apresentados neste documento apenas usam um objeto.

As funções e os tipos de dados da biblioteca *CanvasLib* estão definidos no *package* `pt.isel.canvas`. Por este motivo, é recomendado iniciar os módulos que usam a *CanvasLib* com: `import pt.isel.canvas.*`

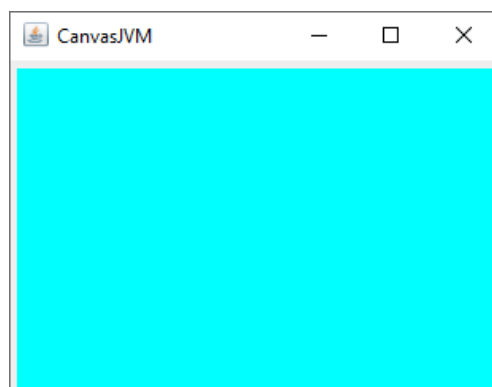
A função `main`, chamada no início do programa, deve chamar as funções `onStart` e `onFinish` da biblioteca para definir o que deve ser executado nos momentos de carregamento e de fecho da janela/página.

A criação de objetos Canvas deve ser realizada no código `onStart`. Se não for criado nenhum Canvas a janela nem chega a ser aberta.

Uma versão minimalista da função `main`, que apenas cria um Canvas é a seguinte:

```
import pt.isel.canvas.*

fun main() {
    print("Begin ")
    onStart {
        val arena = Canvas(300, 200, CYAN)
        print("Start ")
    }
    onFinish {
        print("Finish ")
    }
    print("End ")
}
```



Sendo aberta a janela com aspeto apresentado e sendo escrito no output **Begin End Start**, sendo escrito **Finish** e terminando o programa depois do utilizador fechar a janela com o botão X do canto superior direito.

Como ficou provado nesta execução, o programa não termina quando a função `main` acaba a execução, mas termina quando é fechada a janela.

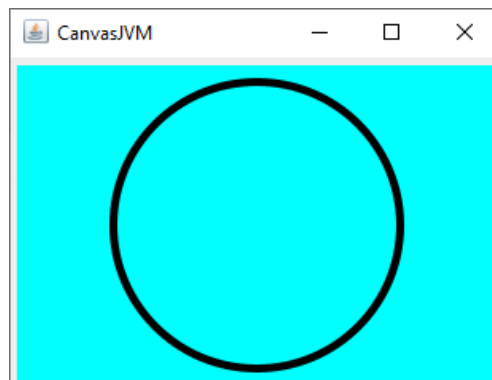
Desenhar no Canvas

As dimensões da área de desenho do Canvas, assim como a cor de fundo são especificadas na criação do objeto. Se não for indicada a cor de fundo, esta será branca.

class `Canvas(val width: Int, val height: Int, val background: Int = WHITE)`

As propriedades `width`, `height` e `background` do objeto criado podem ser consultadas.

```
fun main() {
    onStart {
        val arena = Canvas(300, 200, CYAN)
        val radius = arena.height/2 - 10
        arena.drawCircle(
            arena.width/2, arena.height/2, radius, BLACK, 5
        )
    }
    onFinish { }
}
```

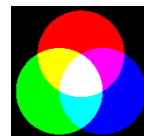


O modelo de cores

Existem funções no Canvas para desenhar segmentos, retângulos, círculos, arcos e texto. Estas funções recebem um parâmetro do tipo `Int`, que pode ser omitido, tendo o valor por omissão `BLACK`.

Na *CanvasLib* estão definidos os valores das cores `BLACK`, `WHITE`, `RED`, `GREEN`, `BLUE`, `YELLOW`, `CYAN` e `MAGENTA`, mas podem ser indicadas outras cores.

Todas as cores podem ser definidas com a mistura das 3 cores principais (vermelho, verde e azul). Designa-se por RGB à forma de codificar uma cor indicando um valor de 0 a 255 para cada uma das três componentes (*Red*, *Green*, *Blue*). A combinação `RGB(255,255,0)` é amarelo porque é mistura de vermelho e verde. Branco é `RGB(255,255,255)` porque é mistura das três cores. Usando valores intermédios formam-se outras cores, por exemplo, o violeta é `RGB(238,130,238)`.



No valor do tipo `Int` que indica a cor, os 3 bytes de menor peso do indicam os valores de cada componente RGB. A representação em hexadecimal (base 16) é a mais cómoda para especificar uma cor, porque cada dois dígitos representam um byte. Por exemplo, o valor `0xFFFF00` é (255,255,0) que representa o amarelo.

Área de desenho

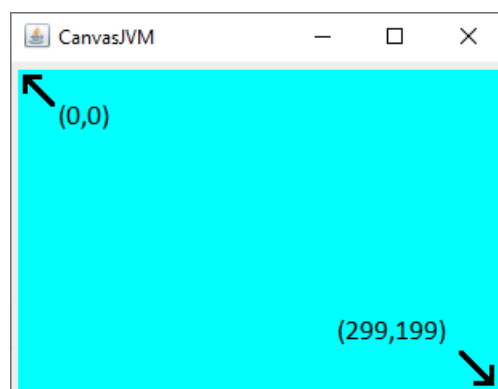
Todas as funções para desenhar recebem parâmetros com os valores de x e de y de um ou mais pontos envolvidos no desenho. Estas coordenadas têm como referencial o canto superior esquerdo.

O canto superior esquerdo da área de desenho do Canvas está na posição (x=0, y=0) e o canto inferior direito é (x=width-1, y=height-1).

```
val arena = Canvas(300, 200, CYAN)
...
```

Para apagar toda a área de desenho é chamada a função `erase`.

```
fun erase()
```

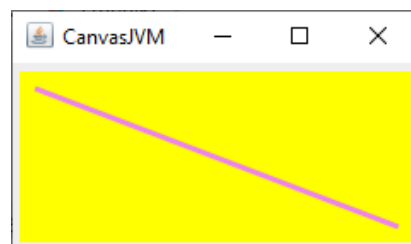


Desenhar Segmentos

Para desenhar um segmento de reta é chamada a função `drawLine` passando como argumentos os valores de x e de y dos extremos e opcionalmente a cor da linha e a grossura da linha. Por omissão, a grossura da linha é 3.

```
fun drawLine(xFrom :Int, yFrom :Int, xTo :Int, yTo :Int,
             color :Int = BLACK, thickness :Int = 3)
```

```
val arena = Canvas(230, 100, YELLOW)
arena.drawLine(10, 10, arena.width-10, arena.height-10,
0xEE82EE)
```



Desenhar Retângulos

Para desenhar retângulos é chamada a função `drawRect` passando como argumentos os valores de x e y do canto superior esquerdo, a largura e a altura e opcionalmente a cor da linha ou do interior e a grossura da linha. Se for indicada a grossura da linha é desenhada a periferia do retângulo, caso contrário é desenhado o retângulo a cheio.

```
fun drawRect(x: Int, y: Int, width: Int, height: Int,
             color: Int =BLACK, thickness: Int =0)
```

```
val arena = Canvas(230, 100, YELLOW)
arena.drawRect(10,10, 80,80, BLUE)
arena.drawRect(100,10, 120,80, RED,5)
```



Desenhar Circunferências e Círculos

Para desenhar círculos ou circunferências é chamada a função `drawCircle` passando como argumentos os valores de `x` e `y` do centro, o raio e opcionalmente a cor da linha da circunferência ou do interior do círculo e a grossura da linha. Se for indicada a grossura da linha é desenhada uma circunferência, caso contrário é desenhado um círculo a cheio.

```
fun drawCircle(xCenter: Int, yCenter: Int, radius: Int,
               color: Int = BLACK, thickness: Int = 0)
```

```
val arena = Canvas(230, 100, YELLOW)
arena.drawCircle(50, arena.height/2, radius = 40, GREEN)
arena.drawCircle(arena.width-50, arena.height/2, 40, thickness = 10)
```

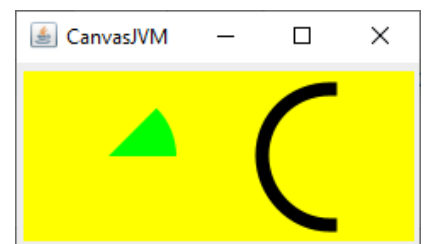


Desenhar Arcos de circunferências e Fatias de círculos

Para desenhar arcos ou fatias de círculos é chamada a função `drawArc` passando como argumentos os valores de `x` e `y` do centro de rotação, o raio, o ângulo de partida e o ângulo de chegada em graus no sentido direto e opcionalmente a cor da linha ou do interior e a grossura da linha. Se for indicada a grossura da linha é desenhado o arco de circunferência, caso contrário é desenhada a fatia do círculo a cheio.

```
fun drawArc(xCenter: Int, yCenter: Int, radius: Int,
            startAng: Int, endAng: Int = 360,
            color: Int = BLACK, thickness: Int = 0)
```

```
val arena = Canvas(230, 100, YELLOW)
arena.drawArc(50, arena.height/2, radius = 40, 0, 45, GREEN)
arena.drawArc(arena.width-50, arena.height/2, 40, 90, 270, BLACK, 8)
```

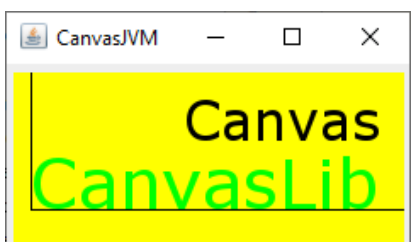


Desenhar Texto

Para desenhar texto é chamada a função `drawText` passando como argumentos os valores de `x` e `y` do canto inferior esquerdo da caixa de texto que serve de base, o texto a escrever, a cor do texto e, opcionalmente, a dimensão da fonte a usar. Por omissão é reutilizada a última dimensão de fonte usada ou 32 se for a primeira vez.

```
fun drawText(x: Int, y: Int, txt: String,
             color: Int = BLACK, fontSize: Int? = null)
```

```
val arena = Canvas(230, 100, YELLOW)
arena.drawText(100, 40, "Canvas")
arena.drawText(10, 80, "CanvasLib", GREEN, 40)
arena.drawLine(10, 80, arena.width, 80, BLACK, 1)
arena.drawLine(10, 80, 10, 0, BLACK, 1)
```

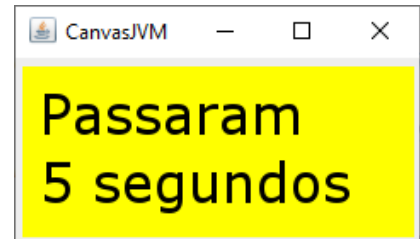


Temporização

Para executar um troço de código passado um determinado tempo é chamada a função `onTime`, passando como argumento o tempo em milissegundos e o código a executar.

```
fun onTime(delay :Int, handler :()->Unit)
```

```
onStart {  
    val arena = Canvas(230, 100, YELLOW)  
    arena.onTime(5000) {  
        arena.drawText(10,40,"Passaram")  
        arena.drawText(10,80,"5 segundos")  
    }  
}
```



A figura mostra a janela depois de terem passado 5 segundos.

Para executar periodicamente um troço de código é chamada a função `onTimeProgress`, passando como argumento o período em milissegundos e o código a executar. O código a executar pode usar o parâmetro recebido que tem o tempo real decorrido desde o momento em que a função foi chamada. Caso seja útil pode ser usado o objeto retornado, do tipo `TimerCtrl`, que tem a função `stop` para permitir parar a execução periódica.

```
fun onTimeProgress(period :Int, handler :(Long)->Unit) :TimerCtrl
```

```
onStart {  
    val arena = Canvas(230, 100, YELLOW)  
    arena.onTimeProgress(500) { tm :Long ->  
        val dx = (tm/50).toInt()  
        arena.drawCircle(20+dx, 50, 20,RED,5)  
    }  
}
```



Este exemplo desenha uma nova circunferência a cada meio segundo.

A figura mostra a janela depois de terem passado 4,5 segundos.

Recolha de Informação do rato

Para executar um troço de código em cada *click* do rato é chamada a função `onMouseDown`. O código a executar pode usar o parâmetro recebido que tem um valor do tipo agregado `MouseEvent` com informação do rato quando ocorreu o *click* no *canvas*.

```
data class MouseEvent(val x :Int, val y :Int, val down :Boolean)
```

As propriedades `x` e `y` têm a coordenada do rato. A propriedade `down` está a `true` se o botão está premido.

```
fun onMouseDown(handler :(MouseEvent)->Unit)
```

```
onStart {  
    val arena = Canvas(230, 100, YELLOW)  
    arena.onMouseDown { me :MouseEvent ->  
        arena.drawCircle(me.x, me.y, 20, GREEN)  
    }  
}
```



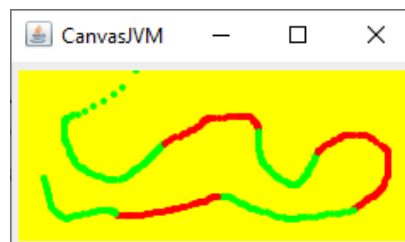
Este exemplo desenha um círculo por cada *click* do rato, com centro na posição do *click*.

A figura mostra a janela depois de 5 *clicks* em posições diferentes.

Para executar um trecho de código em cada movimento do rato, na área do *canvas*, é chamada a função `onMouseMove`. O código a executar pode usar o parâmetro recebido, do tipo `MouseEvent`, com as coordenadas correntes do rato e o estado do botão do rato.

```
fun onMouseMove(handler : (MouseEvent) -> Unit)
```

```
onStart {  
    val arena = Canvas(230, 100, YELLOW)  
    arena.onMouseMove { me ->  
        val color = if (me.down) RED else GREEN  
        arena.drawCircle(me.x, me.y, 2, color)  
    }  
}
```



Este exemplo desenha um pequeno círculo a cada movimento do rato na posição corrente do rato, sendo vermelho se o botão do rato estiver premido e verde caso não esteja.

Recolha de Informação do teclado

Para executar um trecho de código cada vez que uma tecla do teclado é premida é chamada a função `onKeyPressed`. O código a executar pode usar o parâmetro recebido que tem um valor do tipo agregado `KeyEvent` com informação da tecla que foi premida.

```
data class KeyEvent(val char :Char, val code :Int, val text :String)
```

A propriedade `char` é o símbolo da tecla (caso exista). A propriedade `code` tem o código da tecla. A propriedade `text` tem uma descrição textual da tecla.

```
fun onKeyPressed(handler : (KeyEvent) -> Unit)
```

```
onStart {  
    val arena = Canvas(230, 100, YELLOW)  
    var x = 10  
    arena.onKeyPressed { k ->  
        arena.drawText(x, 40, k.text)  
        x += 20  
    }  
}
```



A figura mostra a janela depois de premidas as teclas **q**, **w**, **e**, **r**, **t** e **y**.

Fechar o Canvas

Para fechar a área de desenho de do *canvas* é chamada a função `close`, ficando inutilizado o objeto. Caso este seja o último objeto da janela/página, esta será fechada, sendo executado o código associado em `onFinish`.

```
fun close()
```

```
onStart {  
    val arena = Canvas(230, 100, YELLOW)  
    arena.onMouseDown { arena.close() }  
}
```

Este exemplo fecha o *canvas* quando ocorrer um *click* do rato na área do *canvas*.

Reprodução de sons

Para reproduzir um som é chamada a função global `playSound`, passando como argumento o nome do ficheiro com o áudio a reproduzir disponível nos recursos (pasta **resources** do projeto *IntelliJ*) ou na pasta raiz do projeto. Caso o nome indicado não tenha extensão é assumido que o ficheiro tem a extensão **wav**.

`fun playSound(sound: String)`

```
onStart {  
    val arena = Canvas(230, 100, YELLOW)  
    arena.onMouseDown { playSound("click") } //play click.wav file  
}
```

Neste exemplo é reproduzido o som do ficheiro "click.wav" por cada *click* do rato. O ficheiro está na pasta raiz ou na pasta *resources* do projeto.

Para evitar o tempo de espera envolvido no carregamento demorado do ficheiro de áudio na primeira vez que determinado som é reproduzido, os sons podem ser carregados antes de serem reproduzidos. Para tal, pode ser chamada a função `loadSounds`, passando como argumentos os nomes de todos os sons que se pretendem carregar.

`fun loadSounds(vararg names: String)`

```
onStart {  
    loadSounds("click", "ping", "pong")  
    ...  
}
```

Este exemplo faz o pré-carregamento de 3 ficheiros de áudio antes de serem reproduzidos.