Game API

Version 0.9, Draft

API GUIDE

COPYRIGHT

Trademarks and Service Marks

About this Document

This document describes the Game package, its features and how it helps to develop games. This document also provides an example to understand Game API.

Scope

This document is for novice users who have basic knowledge of Java ME and desire to know about Game API package and how to develop games using this.

Document History:

Date	Version	Comment
22/06/09	0.9	Draft

References:

1. Game API: http://developers.sun.com/mobility/midp/articles/gameapi/

2. Game Article: http://developers.sun.com/mobility/midp/articles/game/

Abbreviations:

Java ME	Java Micro Edition
API	Application Programming Interface
JVM	Java Virtual Machine
CLDC	Connection Limited Device Configuration
MIDP	Mobile Information Device Profile
JSR	Java Specification Request

Table of Contents

Table of Figures

Introduction

MIDP 2.0 has come up with a new Game package that was previously lacking MIDP 1.0. This new Game package simplifies game development. MIDP 1.0 provides limited support for game programming. To overcome these limitations, and enhance the games, Game Package has been introduced in MIDP 2.0.

Overview

Game package API *javax.microedition.lcdui.game* is compact and consists of only five classes. They are:

- Game Canvas
- Layer
- Layer Manager
- Sprite
- TiledLayer

GameCanvas

It is a subclass of Canvas. It also provides synchronized graphics flushing. It provides basic screen functionality and has the ability to query key status.

Layer

This class is an abstract class. It also represents visual elements in the game such as Sprite or TiledLayer. It forms the basis of Layer framework and provides basic attributes like location, size, and visibility.

LayerManager

This LayerManager class automates the rendering process. With the help of this class, developer can set a view window, which represents the users' view of the game and automatically renders the game layers to implement the desired view.

Sprite

This layer displays several graphics frames. It is a basic animated layer. It also provides various transformation and collision detection methods. Sprite is a subclass of Layer.

TiledLayer

TileLayer is a subclass of Layer. This layer is used to define large screen by dividing the screen in a grid of cells that can be filled with a small images (Tiles). Here you can fill the cells with animated tile. This package makes rich game development by simplifying the game logic, improving game performance. So creates smooth animations.

Developing Game using Game API

Here we can develop a simple game using Game API. We can develop a game screen where character moves horizontally i.e. along x-axis from one end to another end of the screen. So to avoid collision, character needs to jump across the obstacle. The complete code is given at the end of this document.

We will get to know the features of each class and how to use this class as we progress.

A. Starting with GameCanvas

Game development in JavaME revolves around:

- Code that tracks and updates game state.
- Painting, repainting the game screen based on the game state.
- Getting user inputs through keypressed, keyreleased events.
- Number of Threads.

All these stages are very important for the developer. With MIDP 1.0, the above 3 steps are executed by different threads. Example 1 shows how MIDP 1.0 game structure looks like.

```
public class MyCanvas extends Canvas implements Runnable {

 public void run() {
  while (true) {
   /*Game Thread
   *Update the game state.  Painting*/
   repaint();
   // Thread sleep delay
 }
}

 public void paint(Graphics g) {
  /*your game painting code*/
}
```

```
 protected void keyPressed(int keyCode) {
   /*System Calls response to keyPressed*/
    }

}
```

Example 1: MIDP 1.0 game structure

Problem with MIDP 1.0 game was that it spread across different methods. The results:

- No way of knowing when will system call paint().
- No way of knowing on system keypressed calls, paint() would render the screen based on the latest inputs.
- Animation might look jerky.
- Flickering issue.

GameCanvas resolves the screen painting and key events handling by executing game logic in single loop. It does so by providing:

- Double Buffering.
- Querying key states.

MIDP 2.0 game structure looks like the one shown in example 2.

```
public class MyGameCanvas extends GameCanvas implements Runnable {
  public void run() {
    Graphics g = getGraphics();
    while (true) {
      /*your game code updating state*/
      int keyState = getKeyStates();
      /*your key press code */
      /*your Painting code*/
      flushGraphics();
      /*Thread sleep
   }
  }
}
```

Example 2. MIDP 2.0 game structure

GameCanvas provides off-screen buffer in which all painting operations are done and then this buffer is rendered on the device screen. Graphics object is obtained using *getGraphics()*. Now any rendering on this graphics object is done on off-screen buffer. Rendering the off-screen buffer on the screen is done by using *flushGraphics()*.
Double buffering ensures smooth transitions of elements and resolves flickering issue.

GameCanvas also provides *getKeyStates()* for obtaining the current state of device keys. Developer can determine immediately which keys are pressed using *getKeyStates().*

The GameCanvas structure for out game would look like the one shown in example 3. Moving ahead, other classes would be introduced in the MyGameCanvas class.

```
import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.game.GameCanvas;
import java.io.IOException;

public class MyGameCanvas extends GameCanvas implements Runnable {
   /*character image*/
   private Image charImg;
   /*character coordinates*/
   private int charX;
   private int charY;
   /*character movement along x axis*/
   private final int dx = 1;
   /*screen Boundary*/
   private final int screenWidth = 160;
   private final int screenHeight = 160;
   /*define tile size*/
   Private final int tileSize = 16;
   /*screen refresh value*/
   private final int frameDelay = 30;

   /*MyGameMidlet*/
   private MyGameMidlet midlet;
   /*MyGameThread */
   private Thread myGameThread;

   public MyGameCanvas(MyGameMidlet midlet) {
      super(true);
      this.midlet = midlet;
      loadCharImg();
      setFullScreenMode(true);
      initialiseNewGame();
      myGameThread = new Thread(this);
```

```
    myGameThread.start();
}

private void loadCharImg()
{
    try {
        /* load character image from resource*/
        charImg = Image.createImage("/charImg.png");
    } catch (IOException ioex) {
        ioex.printStackTrace();
    }
}

public void initialiseNewGame() {
    charX = 0;
    charY = getHeight()-charImg.getHeight();
}

public void run() {
    /* get graphics object for this canvas */
    Graphics g = getGraphics();
    /*infinite loop*/
    while (true) {
        /*Check for user key presses*/
        getUserInput();
            /*render screen*/
        renderScreen(g);
            /*controls refresh rate*/
        try {
            Thread.sleep(frameDelay);
        } catch (Exception e) {
        }
    }
}

private void getUserInput() {
    /*get keys state*/
    int keyState = getKeyStates();
    /*calculate the position for x axis*/
    calculateCharMovement(keyState);
}

private void renderScreen(Graphics g) {
    /*clear the background*/
    g.setColor(0xffffff);
    g.fillRect(0, 0, screenWidth, screenHeight);
    /*draw character image*/
```

```
        g.drawImage(charImg,charX,charY, Graphics.TOP | Graphics.LEFT);
        /*paint off-screen buffer to screen*/
        flushGraphics();
    }

    private void calculateCharMovement(int keyState) {
        /*check which way to move and change direction*/
        if ((keyState & LEFT_PRESSED) != 0) {
            charX = Math.max(0,charX-dx);
        } else if ((keyState & RIGHT_PRESSED) != 0) {
            charX = Math.min(screenWidth - charImg.getWidth(), charX+dx);
        }
    }
}
```

Example 3: GameCanvas structure

B. Creating Backgrounds

GameCanvas section describes the game structure. Now we will add background to the game screen. Background can be added using TiledLayer class.

TileLayer is composed of grid of cells. This is a visual element. Here grids of cells can be filled with tile images. Source image can be divided into sets of equally sized tiles i.e. width and height. Tile is an image having equal width and height. For example: Figure 1 represents a single image having two tiles of 16 x 16 width and height.

Figure 1: Element Image

Fill the background using TileLayer in these tiles. Divide the screen into two parts, one representing the sky and other representing the earth. Upper part of the screen is our sky and lower part will be the earth part. TiledLayer class will take element image and divide it into two separate tile images used for rendering the screen background.

Figure 2: Sky Tile

Figure 3: Earth Tile

Figure 4: Collision Tile

In TileLayer, number is given to each tile starting with 1 from left. It means sky image will be assigned number 1 and earth image would be assigned number 2.

Now image file is created from the tile images. Define grid for background rendering. After defining the tile size, number of rows and columns in grid would be:

```
Number of columns = screenWidth / tile size.
Number of rows    = screenHeight / tile size.
```

i.e. Number of columns = (160/16) = 10 and Number of rows = (160/16) = 10.

The grid would look like:

```
// array that specifies which tile image goes where
int[] gridCells  = {
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, // earth
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, // earth
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, // earth
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, // earth
};
```

The constructor of TileLayer would be

```
TileLayer backgroundLayer = new TiledLayer(10, 10, elementImg, tileSize, tileSize);
```

Where 10 and 10 are the grid rows and columns, elementImg is the source image from where tiles would be created and each tile is of 16 x 16.

Use setCell method to fill the background.

```
public void setCell(int col, int row, int tileIndex)
```

Call *paint(Graphics g)* method to render the background.

Now add createBackgroundScreen() that would create TileLayer Object.

```
public void createBackgroundScreen()
{
   /* array that specifies which tile image goes where */
   int[] gridCells  = {
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
      2, 2, 2, 2, 2, 2, 2, 2, 2, 2, // earth
      2, 2, 2, 2, 2, 2, 2, 2, 2, 2, // earth
      2, 2, 2, 2, 2, 2, 2, 2, 2, 2, // earth
      2, 2, 2, 2, 2, 2, 2, 2, 2, 2, // earth
   };
   final int noRows = 10;
   final int noCols = 10;
   backgroundLayer =  new TiledLayer(noCols, noRows, elementImg,tileSize,tileSize);

   /* set the background with the images */
   for (int i = 0; i < gridCells .length; i++) {
      int column = i % 10;
      int row = (i - column)/10;
      backgroundLayer.setCell(column, row, gridCells [i]);
   }

   /* set the location of the background */
   backgroundLayer.setPosition(0, 0);
}
```

Similarly, we would create Collision TileLayer

```
   final int collisionIndex = 3;
   int[] collisionCells  = {
      collisionIndex,  /* fire  */
   };
   final int no_Collision_Rows = 1;
   final int no_Collision_Cols = 1;
   collisionLayer =  new TiledLayer(no_Collision_Cols, no_Collision_Rows,
                                                          elementImg,tileSize,tileSize);

   /* set the background with the images */
   collisionLayer.setCell(0, 0, collisionCells [0]);
  /*  set the location of the background */
   collisionLayer.setPosition(4*tileSize, screenHeight-(5*tileSize));
```

where 3 is the tile Index of elementImg for collision TileLayer.

To render the background we would add paint() of TileLayer object in renderScreen().

```
private void renderScreen(Graphics g) {
      /*clear the background*/
      g.setColor(0xffffff);
      g.fillRect(0, 0, screenWidth, screenHeight);

      /* Rendering background*/
      backgroundLayer.paint(g);

     /*Rendering Collision Layer*/
     collisionLayer.paint(g);

      /*draw character image*/
      /* g.drawImage(charImg,charX,charY, Graphics.TOP | Graphics.LEFT); */

      /*paint off-screen buffer to screen*/
      flushGraphics();
   }
```

We need collision layer object to be placed on the earth. For which we would use *setPosition(int x, int y)* method. By default it is rendered to left top coordinate system. The output screen looks as seen in figure 5.

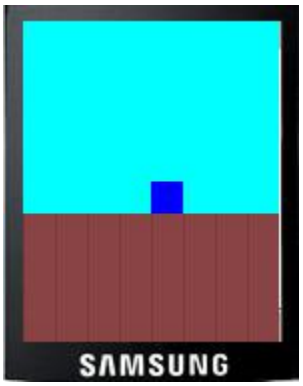

Figure 5 Screen output

C. Creating Animation Sprite

Game structure and background are created. Now we have to animate the character.

Animation is done by using Sprite class. Sprite concept is almost same as TileLayer. Sprite uses a sequence of source image frames for animation. Sprite requires one source

image from which frames are made. Like tiles, frames should also be of equal size. See the figure 6.

Using the character Image as source image, we can create character walk Sprite.

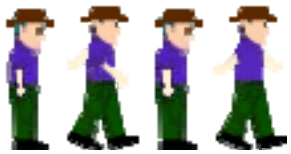

Figure 6: Character Image

In source image, each frame has a number, starting from 0 and counting up. The Sprite has a *frame sequence* that determines the order in which the frames will be shown. The default frame sequence for a new Sprite simply starts at 0 and counts up through the available frames.

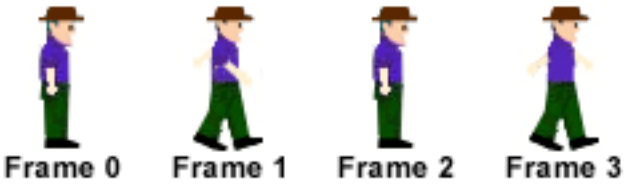

Figure 7: Different Frames

```
Sprite characterSprite = new Sprite(characterImg, characterWidth, characterHeight);
```

where characterWidth and characterHeight are same in each individual frame.

We will create a createCharacterSprite method for creating characterSprite Object.

```
public void createCharacterSprite() {
 final int characterWidth = 28;
final int characterHeight = 54;
 characterSprite = new Sprite(charImg, characterWidth, characterHeight);
characterSprite.setPosition(charX, chary);
}
```

These frames moves in circular path or sequential path. For example: frame 0 followed by 1, 2, 3 and again 0.

To move to the next or previous frame in the frame sequence, *nextFrame()* and *prevFrame()* methods should be used.

To specify a frame sequence that is different from the default, use the *setFrameSequence(int[] sequence)* which takes arbitrary sequence.

You can also jump to a particular point in the current frame sequence by calling *setFrame().*

To position the sprite in screen, *setPosition(int x, int y)* is used. To render Sprite use the *paint(Graphics g)* method.

Transformation using Sprite

In Sprite class, we can transform the source frames. This feature is essential, as our character would walk in right as well as left direction as required.

See figure 5. Image shows character movement in right direction, for making character to walk in left direction we would simply Flip the frames. This is done using Sprite *setTransform()* method.

To rotate the frames use mirror option or 90-degree option or use both as required. Constants in the Sprite class enumerate the possibilities. To rotate the character Sprite in left direction:

```
characterSprite.setTransform(Sprite.TRANS_MIRROR);
```

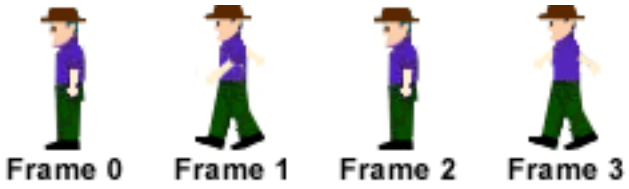

Figure 8: Mirrored Frames

While Transforming, Sprite's is positioned such that its *reference pixel* does not move. Sprite includes the concept of a *reference pixel*. The reference pixel is defined by specifying its location in the Sprite's untransformed frame using *defineReferencePixel(int x, int y)*

The reference pixel of a Sprite is located at 0, 0 in the Sprite's coordinate space, at its upper left corner by default. Location of the reference pixel is also transformed when a transformation is applied. The location of the Sprite is adjusted so that the reference pixel stays in the same place.

Checking Collision using Sprite

Now we have almost done with the game part we need to check collision part of the game. Sprite class provides two methods for collision:

1. **Collision with Sprite:**
   boolean collidesWith(Sprit s, boolean pixelLevel) checks for collision with the specified sprites.

2. **Collision with TileLayer:**
   boolean collidesWith(TiledLlayer s, boolean pixelLevel) checks for collision with TiledLayer. Method returns true it collides with a non empty tile in TiledLayer.

We are using TiledLayer collision (collisionLayer) detection. See the previous code example for reference.

Using LayerManager

Our Game example is completely over; we have defined game structure, defined TileLayers for background, collisionLayer for detecting collision, Sprite for character animation, transformation and collision detection.

What is the use of LayerManager then?

You know that TileLayer and Sprite are subclass of Layer. Game may have several TileLayers, sprites. Managing them might be complex that is where LayerManager comes into picture.

With the help of LayerManager, you can manage and organize the layers. This class provides methods to add *append(Layer layer)*, remove *remove(Layer layer)*, and insert *insert(Layer layer, int index)* layers from a game. LayerManager maintains an ordered list for appending, removing and inserting.

The layer at index 0 is painted on top followed by other layers in z-order i.e. layer 0 is *closest* to the user, and so on. It also provides a single *paint(Graphics g, int x, int y)* method to paint all Layers. That means all layers are painted by this single method instead of developer managing all individual layer paint(Graphics g) method.

So modify the code. Use LayerManager to manage backgroundLayer, collisionLayer and characterSprite.

The final code with all the changes is given below:

Class: MyGameCanvas.java

```
import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.game.GameCanvas;
import javax.microedition.lcdui.game.TiledLayer;
import javax.microedition.lcdui.game.LayerManager;
import javax.microedition.lcdui.game.Sprite;

import java.io.IOException;

public class MyGameCanvas extends GameCanvas implements Runnable {
   /*character, elements image*/
   private Image charImg,elementImg;
   /*character coordinates*/
   private int charX;
   private int charY;
   /*character movement along x axis*/
   private final int dx = 1;
   /*screen Boundary*/
   private final int screenWidth = 160;
   private final int screenHeight = 160;
   private final int tileSize = 16;
   /*Define Sprite Frame*/
   private final int charWidth = 28;
   private final int charHeight = 54;
   /*Define character walk Direction*/
   private int charDirection, prevDirection;
   /*define character state*/
   private int charState;
   private final int charWalk = 1;
   private final int charStop = 2;
   private final int charCollide = 3;
   private final int charJump = 4;
   /*character jump sequence*/
   private int charJumpReg_Y[] = {8,12,14,16,18,16,14,12,8};
   private int charJumpReg_X[] = {5,6,7,8,8,8,7,6,5};
   private int charJumpCounter;
   /*screen refresh value*/
   private final int frameDelay = 30;
   /*MyGameMidlet*/
   private MyGameMidlet midlet;
   /*MyGameThread */
   private Thread myGameThread;
   /*background TileLayer*/
   private TiledLayer backgroundLayer;
   /*collision TileLayer*/
```

```
private TiledLayer collisionLayer;
/*character Sprite*/
private Sprite characterSprite;
/*LayerManager */
private LayerManager layerManager;

public MyGameCanvas(MyGameMidlet midlet) {
    super(true);
    this.midlet = midlet;
    loadCharImg();
    createBackgroundScreen();
    createCharacterSprite();
    createLayerManager();
    initialiseNewGame();
    setFullScreenMode(true);
    myGameThread = new Thread(this);
    myGameThread.start();
}

private void loadCharImg()
{
    try {
        /* load character image from resource */
        charImg = Image.createImage("/charImg.png");
        elementImg = Image.createImage("/elements.png");
    } catch (IOException ioex) {
        /*System.out.println("loadCharImg "+ioex);*/
        ioex.printStackTrace();
    }
}

private void createBackgroundScreen()
{
    final int noRows = 10;
    final int noCols = 10;
    int[] gridCells  = {
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, // sky
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, // earth
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, // earth
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, // earth
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, // earth
    };
```

```
        backgroundLayer =  new
                            TiledLayer(noCols, noRows, elementImg,tileSize,tileSize);
        /* set the background with the images */
        for (int i = 0; i < gridCells .length; i++) {
           int column = i % gridCellsLength;
           int row = (i - column)/gridCellsLength;
           backgroundLayer.setCell(column, row, gridCells [i]);
        }
        /* set the location of the background*/
        backgroundLayer.setPosition(0, 0);

       final int fireIndex = 3;
       final int  no_Collision_Cols = 1;
       final int  no_Collision_Rows = 1;
        int[] collisionCells  = {
           fireIndex, /* collision Index*/
        };

        collisionLayer =  new TiledLayer(no_Collision_Cols, no_Collision_Rows,
elementImg,tileSize,tileSize);
        /* set the background with the images  */
        collisionLayer.setCell(0, 0, collisionCells [0]);
        /* set the location of the background */
        collisionLayer.setPosition(4*tileSize, screenHeight-(5*tileSize));
    }

    private void createCharacterSprite()
    {
        characterSprite = new Sprite(charImg,charWidth,charHeight);
        characterSprite.setFrame(0);
    }

    private void createLayerManager()
    {
        layerManager =  new LayerManager();
        layerManager.append(characterSprite);
        layerManager.append(collisionLayer);
        layerManager.append(backgroundLayer);
    }

    private void initialiseNewGame() {
        charX = 0;
        charY = screenHeight-charHeight-(4*tileSize);
        charJumpCounter = 0;
        charDirection = RIGHT_PRESSED;
        prevDirection = charDirection;
```

```
    charState = charStop;
    characterSprite.setTransform(Sprite.TRANS_NONE);
    characterSprite.setPosition(charX, charY);
}

public void run() {
    /* get graphics object for this canvas */
    Graphics g = getGraphics();
    /*infinite loop*/
    while (true) {
        /*Check for user key presses*/
        getUserInput();
        /*check for collision*/
        checkCharCollision();
        /*render Screen*/
        renderScreen(g);
        /*controls refresh rate*/
        try {
            Thread.sleep(frameDelay);
        } catch (Exception e) {
        }
    }
}

private void getUserInput() {
    /*get keys state*/
    int keyState = getKeyStates();
    /*calculate the position for x axis*/
    if(charState!=charCollide)
    calculateCharMovement(keyState);
}

private void calculateCharMovement(int keyState) {
    /*check which way to move and change direction*/
    if ((keyState & FIRE_PRESSED) !=0 ) {
        if(charState!=charJump) {
            charState = charJump;
            characterSprite.setFrame(0);
        }
    }else  if ((keyState & LEFT_PRESSED) != 0) {
            charX = Math.max(0,charX-dx);
            charDirection = keyState;
            charState = charWalk;

    } else if ((keyState & RIGHT_PRESSED) != 0) {
            charX = Math.min(screenWidth-charWidth, charX+dx);
            charDirection = keyState;
```

```
            charState = charWalk;
        }else {
            if(charState!=charJump) {
                charState = charStop;
                characterSprite.setFrame(0);
            }
        }

        if(charState == charWalk) {
            if(charDirection == RIGHT_PRESSED) {
                if(prevDirection == LEFT_PRESSED)
                    characterSprite.setTransform(Sprite.TRANS_NONE);
                characterSprite.nextFrame();
            } else if(charDirection == LEFT_PRESSED) {
                characterSprite.setTransform(Sprite.TRANS_MIRROR);
                characterSprite.nextFrame();
            }
            characterSprite.setPosition(charX, charY);
        }else
        if(charState == charJump)  {
            if(charJumpCounter<charJumpReg_Y.length) {
                if(charDirection == LEFT_PRESSED)
                    charX = Math.max(0,charX-charJumpReg_X[charJumpCounter]);
                else
                if(charDirection == RIGHT_PRESSED)
                    charX = Math.min(screenWidth-
charWidth,charX+charJumpReg_X[charJumpCounter]);

                characterSprite.setPosition(charX, (charY-charJumpReg_Y[charJumpCounter]));
                charJumpCounter++;
            }else {
                charJumpCounter = 0;
                charState = charStop;
                characterSprite.setFrame(0);
                characterSprite.setPosition(charX, charY);
            }
        }
        prevDirection = charDirection;
    }

    private void checkCharCollision()
    {
        /*Check for the collision of character and obstacle*/
        if(characterSprite.collidesWith(collisionLayer, true)) {
            charState = charCollide;
        }
    }
```

```
  private void renderScreen(Graphics g) {
    /*clear the background*/
    g.setColor(0xffffff);
    g.fillRect(0, 0, screenWidth, screenHeight);
    /*Layermanager paint */
    layerManager.paint(g,0,0);
    /*restart information*/
    if(charState == charCollide)
    {
      g.setColor(0xffff00);
      g.drawString("Collided.", screenWidth>>1, 20 + screenHeight>>1, 17);
      g.drawString("Press key 0 to restart.", screenWidth>>1, 40 + screenHeight>>1, 17);
    }
    /*paint off-screen buffer to screen*/
    flushGraphics();
  }

  protected void keyPressed(int keyCode)
  {
    if(keyCode == 48)
    {
      initialiseNewGame();
      return;
    }
  }
}
```