Game programming for beginners

Games in Java

Until very recently, professional games have been developed in C or C++. This has changed and now there are great games developed completely in java. The game industry for mobiles is growing and java is the language to program in Android. Android must be by now the most used operative system for smart phones. In the other hand, games like Minecraft, have millions of users and are developed by only one developer, without the support of a big company.

I hope, with this series of tutorials, I can motivate you to step into the world of java programming and in particular into game programming.

The game: Mini Tennis

In this series of tutorials we will developed from scratch a version of one of the most famous games.

This game is not meant to be the next most sold game but just a platform from which to teach and maybe inspire someone to be the next most succesful developer in the world ;)

Index

- Our first graphic: JFrame, JPanel, paint method
- Animation of a moving object
- Sprites
- Events. Keyboard input
- Adding the sprite "racquet"
- Collision detection
- Adding sound to our game
- Creating a Sound class for our game
- Adding punctuation and speed increase
- Creating executable jar archive

Our first graphic: JFrame, JPanel, paint method

To paint something we first need a surface where to paint on. This surface or canvas where we are going to paint our first example is a JPanel object. In the same way a canvas needs a frame to hold it, our JPanel will be framed in a window made by the JFrame class.

JFrame: The window

The following code creates a window "Mini Tennis" of 300 pixels by 300 pixels. The window won´t be visible until we call setVisible(true). If we don´t include the last line "frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)", when we close the window the program won´t finish and will continue running.

```
import javax.swing.*;

public class Game {
        public static void main(String[] args) {
                JFrame frame = new JFrame("Mini Tennis");
                frame.setSize(300, 300);
                frame.setVisible(true);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
}
```

If we run it we will obtain:

With these few instructions we will obtain a window which can be maximize, minimize, change it´s size with the mouse, etc. When we create a JFrame object we start an engine which manages the user interface. This engine communicates with the operative system both to paint in the screen as to receive information from the keyboard and from the mouse. We will call this engine "AWT Engine" or "Swing Engine" because it is made by these two libraries. In the first java versions only AWT existed and then Swing was added. This engine uses several threads.

What is a thread in java?

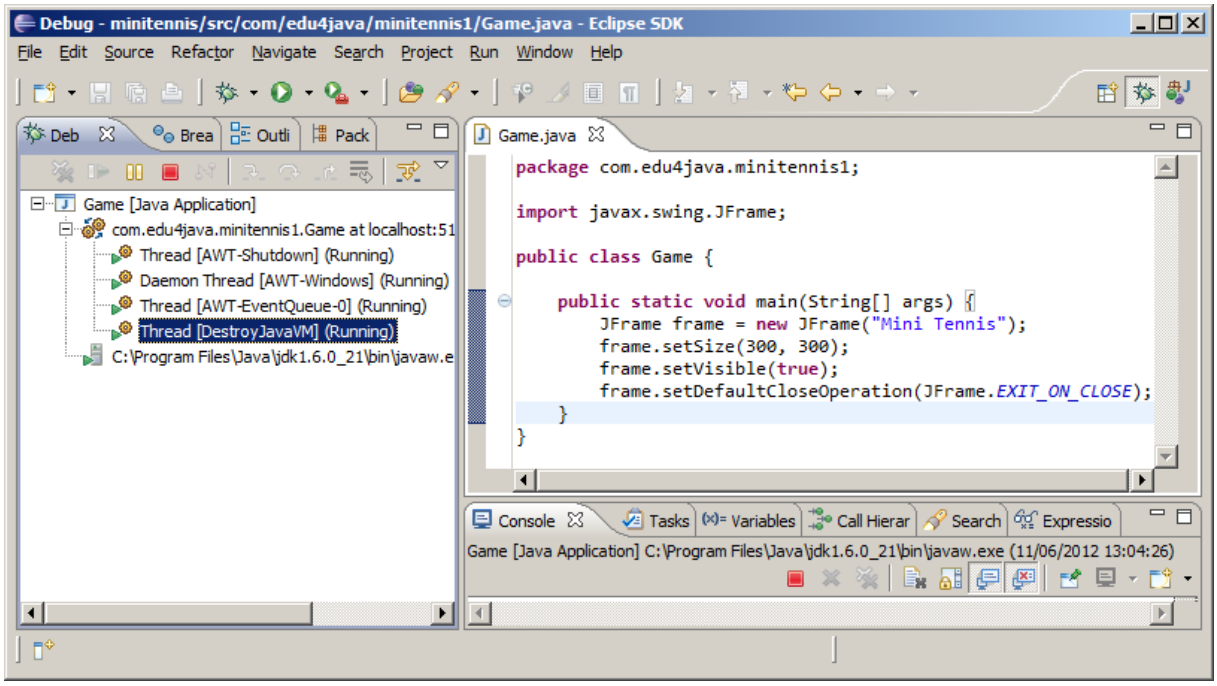
A program is executed by just one processor, line by line. Threads allow a program to start several executions at the same time. This is as if, there were several processors running at the same time their own sequence of instructions.

Even though threads and concurrence are very powerful tools, there can be problems when two threads enter the same variables. It is interesting to think that two threads can be running the same code at the same time.

We can think that a thread is like a cook preparing a dish reading a recipe. Two concurrent threads would be like two cooks working in the same kitchen, preparing one dish with the same recipe o with differents recipes. The problems come when both try to use the same frying pan at the same time.

AWT Engine and Thread AWT-EventQueue

The AWT Engine starts several threads which can be seen if we start the aplication with debug and we go to the debug perspective. Each thread is as if it was an independent program running at the same time as the other threads. Further on we will see more about threads, meanwhile I am only interested that you remember the third thread we see in the debug view called "Thread [AWT-EventQueue-0]" this thread is the one in charge of painting the screen and receiving the mouse and keyboard events.



JPanel: The canvas

To be able to paint we want to know WHERE and where is an JPanel object which will be included in the window. We extend the JPanel class to be able to overwrite the paint method which is the method called by the AWT Engine to paint what appears in the screen.

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class CanvasDemo extends JPanel {
        public void paint(Graphics g) {
                Graphics2D g2d = (Graphics2D) g;
                g2d.setColor(Color.RED);
```

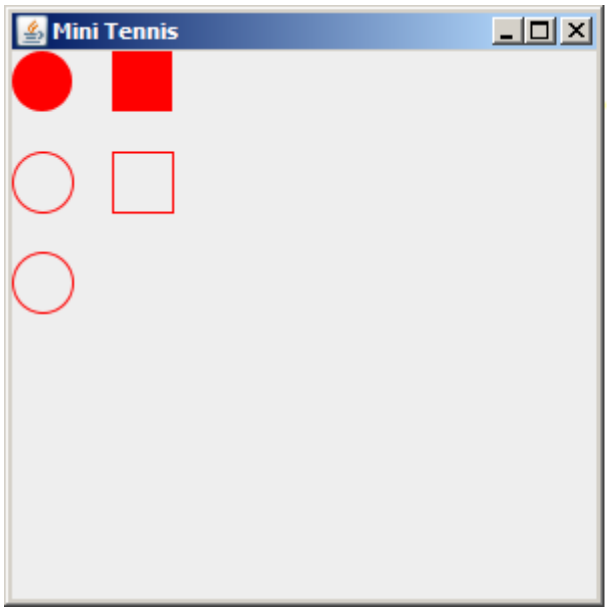
```
                g2d.fillOval(0, 0, 30, 30);
                g2d.drawOval(0, 50, 30, 30);
                g2d.fillRect(50, 0, 30, 30);
                g2d.drawRect(50, 50, 30, 30);

                g2d.draw(new Ellipse2D.Double(0, 100, 30, 30));
        }

        public static void main(String[] args) {
                JFrame frame = new JFrame("CanvasDemo");
                frame.add(new CanvasDemo());
                frame.setSize(300, 300);
                frame.setVisible(true);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
}
```

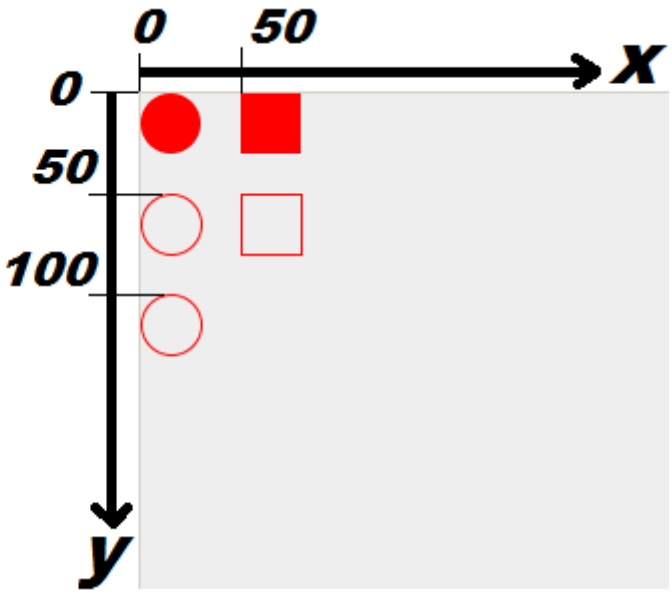
The paint method receives by parameter a Graphics2D object which extends from Graphics. Graphics is an old class used by AWT which has been replaced with Graphics2D which has more and better functionality. The parameter is still a Graphics type due to compatibility but we will use Graphics2D, so we need to create a variable g2d "Graphics2D g2d = (Graphics2D) g;". Once we have g2d we can use all the Graphics2D methods to draw.

The first thing we do is choose the colour we use to draw: "g2d.setColor(Color.RED);". After, we draw circles and squares.

Positioning in the canvas. Coordinate "x" and "y"

To draw something inside the canvas we should indicate in which position we are going to start painting. For this, each of the points in the canvas has an associated position (x,y) being (0,0) the point of the top-left corner.





The first red circle is painted with "**g2d.fillOval(0, 0, 30, 30)**": the first two parameters are the position (x,y) and after comes the width and the height. As a result we have a circle with 30 pixels of diameter in the position(0,0).

The empty circle is drawn with "**g2d.drawOval(0, 50, 30, 30)**": which draws a circle in the position x=0 (left margin) and y=50 (50 pixels below the top margin) with a height of 30 pixels and a width of 30 pixels.

Rectangles are drawn with "**g2d.fillRect(50, 0, 30, 30)**" and "**g2d.drawRect(50, 50, 30, 30)**" in a similar way to the circles.

Lastly "**g2d.draw(new Ellipse2D.Double(0, 100, 30, 30))**" draws the last circle using an Ellipse2D.Double object.

There are a lot of methods in Graphics2D. Some of them will be seen in the following tutorials.

When does the AWT engine call the paint method?

The AWT engine calls the paint method every time the operative system reports that the canvas has to be painted. When the window is created for the first time paint is called. The paint method is also called if we minimize and after we maximize the window and if we change the size of the window with the mouse.

Animation of a moving object

In this tutorial we are going to see how to move a circle around our canvas. We get this animation by painting the circle in a position and then erasing it and drawing it in a near by position. What we get is a moving circle.

The position of the circle

As we said before, every time we paint something we have to define its position (x,y). To make the circle move, we have to modify the position (x,y) each time and paint the circle in the new position.

In our example, we keep the current position of our circle in two properties called "x" and "y". We also create a method called moveBall() which will increase in 1 both "x" and "y", each time we call it. In the paint method we draw a circle with a diameter of 30 pixels in the position (x,y) given by the properties before described; "g2d.fillOval(x, y, 30, 30);".

Game loop

At the end of the main method we start an infinite cicle "while (true)" where we repeatedly call moveBall() to change the position of the circle and then we call repaint(), which forces de AWT engine to call the paint method to paint again the canvas.

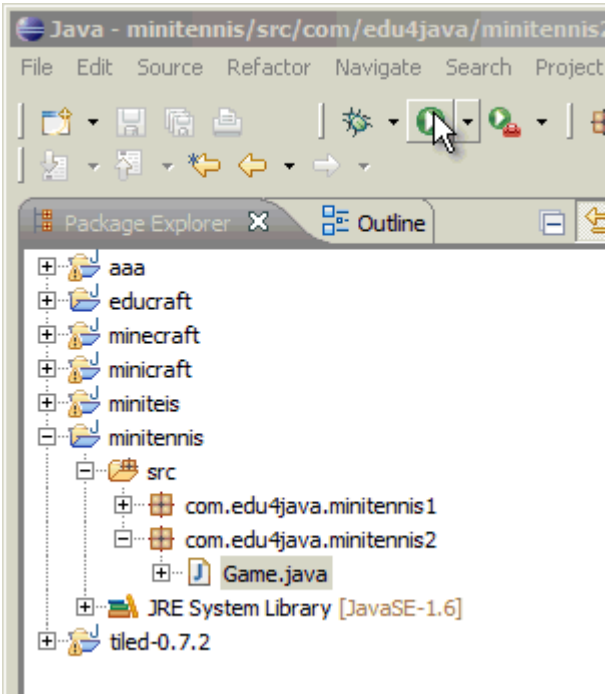
This cycle is known as "Game loop" and carries out two operations:

1. **Update:** update of the physics of our world. In our case the update is given by the moveBall() method, which increases the "x" and "y" in 1.
2. **Render:** painting of the current state of our world including the changes made before. In our example, it is carried out by the call to the method repaint() and the following call to the "paint" method carried out by the AWT engine, and more specifically by the "event queue thread".

```
import java.awt.*;
import javax.swing.*;

public class Game extends JPanel {
        int x = 0;
        int y = 0;

        private void moveBall() {
                x = x + 1;
                y = y + 1;
        }

        public void paint(Graphics g) {
                super.paint(g);
                Graphics2D g2d = (Graphics2D) g;
                g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                                RenderingHints.VALUE_ANTIALIAS_ON);
                g2d.fillOval(x, y, 30, 30);
        }

        public static void main(String[] args) throws InterruptedException {
                JFrame frame = new JFrame("Mini Tennis");
                Game game = new Game();
                frame.add(game);
                frame.setSize(300, 400);
                frame.setVisible(true);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                while (true) {
```

```
                    game.moveBall();
                    game.repaint();
                    Thread.sleep(10);
                }
            }
}
```

When we run this code we obtain:

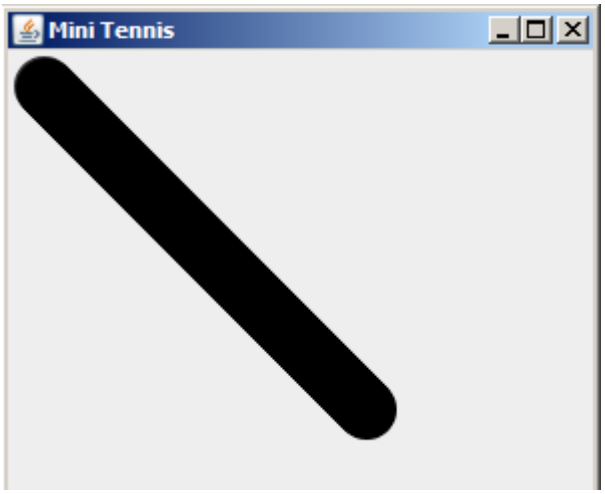


Analyzing our paint method

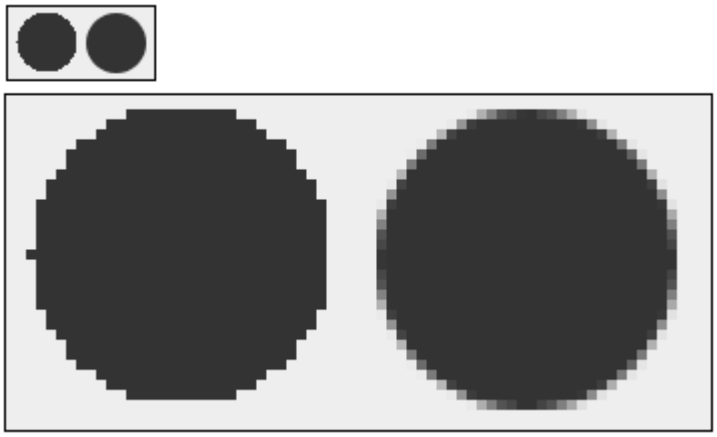
As we said in the last tutorial, this method is run each time the operative system tells the AWT engine that it is necessary to paint the canvas. If we run the repaint() method of a JPanel object, what we are doing is telling the AWT engine to execute the paint method as soon as possible. Calling repaint(), the canvas is painted again and we can see the changes in the position of the circle.

```
        public void paint(Graphics g) {
                super.paint(g);
                Graphics2D g2d = (Graphics2D) g;
                g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                                RenderingHints.VALUE_ANTIALIAS_ON);
                g2d.fillOval(x, y, 30, 30);
        }
```

The call to *"super.paint(g)"*, cleans the screen and if we comment this line we can see the following effect:



The instruction; *"g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON)"* makes the borders of the figures smoother, as you can see in the following graphic. The circle on the left is without applying ANTIALIAS and the one on the right; applying ANTIALIAS.



Sprites

All the objects moving in the screen have their own characteristics such as the position (x,y), speed and direction, etc. All of these characteristics can be isolated in an object which we are going to call "Sprite".

Speed and direction

In the last tutorial we got the ball (circle) to move. It moved downwards and to the right, one pixel every round of the Game Loop. When it got to the border of the screen the ball continued, vanishing from the canvas. Now, we are going to make the ball bounce back once it touches de borders of the canvas, changing its direction.

```
import java.awt.*;
import javax.swing.*;

public class Game extends JPanel {
        int x = 0;
        int y = 0;
        int xa = 1;
        int ya = 1;

        private void moveBall() {
                if (x + xa < 0)
                        xa = 1;
                if (x + xa > getWidth() - 30)
                        xa = -1;
                if (y + ya < 0)
                        ya = 1;
                if (y + ya > getHeight() - 30)
                        ya = -1;

                x = x + xa;
                y = y + ya;
        }

        public void paint(Graphics g) {
                super.paint(g);
                Graphics2D g2d = (Graphics2D) g;
                g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                                RenderingHints.VALUE_ANTIALIAS_ON);
                g.fillOval(x, y, 30, 30);

        }

        public static void main(String[] args) throws InterruptedException {
                JFrame frame = new JFrame("Mini Tennis");
                Game game = new Game();
                frame.add(game);
                frame.setSize(300, 400);
                frame.setVisible(true);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                while (true) {
                        game.moveBall();
                        game.repaint();
                        Thread.sleep(10);
                }
        }
}
```

In this code we can see that there are two more properties; "xa" and "ya", which represents the speed in which the ball is moving. If xa=1, the ball moves to the right, one pixel every round of the Game Loop, if xa=-1, the ball moves to the left. In the same way ya=1 moves the ball down and ya=-1 moves the ball up. This is done with the lines, **"x = x + xa"** and **"y = y + ya"** of the moveBall() method.

Before running the previous instructions, we verify that the ball doesn´t go out of the borders of the canvas. For example, when the ball gets to the right border, or when (x + xa > getWidth() - 30), what we`ll do, is to change the direction of the movement on the "x" axis or what is the same we assign -1 to "xa"; **"xa = -1"**.

```
        private void moveBall() {
                if (x + xa < 0)
                        xa = 1;
                if (x + xa > getWidth() - 30)
                        xa = -1;
                if (y + ya < 0)
                        ya = 1;
                if (y + ya > getHeight() - 30)
                        ya = -1;

                x = x + xa;
                y = y + ya;
        }
```

Each if sentence limits a border of the canvas.

Creation of the "Ball" Sprite

The idea is to create a class called Ball which isolates everything that has to do with the ball. In the following code we can see how we extract all the code from the class Game, which has to do with the ball, and we add it to our new class Ball.

```
import java.awt.*;
import javax.swing.*;

public class Game extends JPanel {
        Ball ball = new Ball(this);

        private void move() {
                ball.move();
        }

        public void paint(Graphics g) {
                super.paint(g);
                Graphics2D g2d = (Graphics2D) g;
                g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                                RenderingHints.VALUE_ANTIALIAS_ON);
                ball.paint(g2d);
        }

        public static void main(String[] args) throws InterruptedException {
                JFrame frame = new JFrame("Mini Tennis");
                Game game = new Game();
                frame.add(game);
                frame.setSize(300, 400);
                frame.setVisible(true);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                while (true) {
```

```
                        game.move();
                        game.repaint();
                        Thread.sleep(10);
                }
        }
}
```

The Ball sprite needs the reference to the Game object to obtain the borders of the canvas and in this way know when to change the direction. In the move() method the Ball class calls the methods **game.getWidth()** and **game.getHeight().**

```
import java.awt.*;

public class Ball {
        int x = 0;
        int y = 0;
        int xa = 1;
        int ya = 1;
        private Game game;

        public Ball(Game game) {
                this.game = game;
        }

        void move() {
                if (x + xa < 0)
                        xa = 1;
                if (x + xa > game.getWidth() - 30)
                        xa = -1;
                if (y + ya < 0)
                        ya = 1;
                if (y + ya > game.getHeight() - 30)
                        ya = -1;

                x = x + xa;
                y = y + ya;
        }

        public void paint(Graphics2D g) {
                g.fillOval(x, y, 30, 30);
        }
}
```

If we execute Game, we will obtain the same result as if we execute the previous version. The convenience of putting the code of the Ball into a Sprite type class becomes more clear when we include the racquet with a new Sprite, in the next tutorial.

Events. Keyboard input

In this tutorial we will see how the events work and particularly how to obtain, from a java program, the information about keyboard events. We will also explain the concept and the use of the anonymous classes, which are the most common way of managing events in java. We will leave aside the game for a moment and we will explain the capture of events in a simple example.

Keyboard reading example

To read from the keyboard it is necessary to register an object which be in charge of "listening if a key is pressed". This object is known as "Listener" and it will have methods that will be called when someone presses a key. In our example the Listener is registered in the JPanel (or KeyboardExample) using the addKeyListener(KeyListener listener) method.

```
import java.awt.event.*;
import javax.swing.*;

public class KeyboardExample extends JPanel {
        public KeyboardExample() {
                KeyListener listener = new MyKeyListener();
                addKeyListener(listener);
                setFocusable(true);
        }

        public static void main(String[] args) {
                JFrame frame = new JFrame("Mini Tennis");
                KeyboardExample keyboardExample = new KeyboardExample();
                frame.add(keyboardExample);
                frame.setSize(200, 200);
                frame.setVisible(true);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }

        public class MyKeyListener implements KeyListener {
                public void keyTyped(KeyEvent e) {
                }

                public void keyPressed(KeyEvent e) {

        System.out.println("keyPressed="+KeyEvent.getKeyText(e.getKeyCode()));
                }

                public void keyReleased(KeyEvent e) {

        System.out.println("keyReleased="+KeyEvent.getKeyText(e.getKeyCode()));
                }
        }
}
```

In the constructor of the KeyboardExample class, we create the listener and we register it. So that the JPanel object receives the keyboard notifications it is necessary to include the instruction setFocusable(true), which allows KeyboardExample to receive the focus.

```
        public KeyboardExample() {
                KeyListener listener = new MyKeyListener();
                addKeyListener(listener);
                setFocusable(true);
        }
```

The MyKeyListener class is the one I use to create the Listener object. This Listener will write on the console, the name of the method and the key which are affected by the event.

```
public class MyKeyListener implements KeyListener {
        public void keyTyped(KeyEvent e) {
        }

        public void keyPressed(KeyEvent e) {

System.out.println("keyPressed="+KeyEvent.getKeyText(e.getKeyCode()));
        }

        public void keyReleased(KeyEvent e) {

System.out.println("keyReleased="+KeyEvent.getKeyText(e.getKeyCode()));
        }
}
```

Once it is registered, when KeyboardExample (our JPanel) has the focus and someone presses a key, KeyboardExample will report it to the listener registered. The Listener of our example implements the KeyListener interface which has the keyTyped(), keyPressed() and keyReleased() methods. The keyPressed method will be called each time the key is pressed (and several times if the key is maintained pressed).

The keyTyped(), keyPressed() and keyReleased() methods receive a KeyEvent object as a parameter, which contains information on which key has been pressed or released. Using e.getKeyCode() we can obtain the key and if we pass a key code to KeyEvent.getKeyText(...), we can obtain the text which is associated to the key.

In the next tutorial we will continue to develop our game.

Adding the sprite "racquet"

In this tutorial we will add a racquet using a Sprite called Racquet. The racquet will move to the left or to the right when we press the cursor keys, so our program has to read from the keyboard.

New Sprite "Racquet"

The first thing we have to do is add a new property called racquet in the class "Game", where we keep the racquet sprite. In the move() method we add a call to racquet.move() and in the paint() method a call to racquet.paint(). Until now, everything is similar to the sprite "Ball", but we have to do something else because the position of the racquet responds to the keyboard.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Game extends JPanel {
        Ball ball = new Ball(this);
        Racquet racquet = new Racquet(this);

        public Game() {
                addKeyListener(new KeyListener() {
                        public void keyTyped(KeyEvent e) {
                        }

                        public void keyReleased(KeyEvent e) {
                                racquet.keyReleased(e);
                        }

                        public void keyPressed(KeyEvent e) {
                                racquet.keyPressed(e);
                        }
                });
                setFocusable(true);
        }

        private void move() {
                ball.move();
                racquet.move();
        }

        public void paint(Graphics g) {
                super.paint(g);
                Graphics2D g2d = (Graphics2D) g;
                g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                                RenderingHints.VALUE_ANTIALIAS_ON);
                ball.paint(g2d);
                racquet.paint(g2d);
        }

        public static void main(String[] args) throws InterruptedException {
                JFrame frame = new JFrame("Mini Tennis");
                Game game = new Game();
                frame.add(game);
                frame.setSize(300, 400);
                frame.setVisible(true);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                while (true) {
                        game.move();
                        game.repaint();
```

```
                        Thread.sleep(10);
                }
        }
}
```

In the constructor of the class "Game" we can see how we register a listener to capture the events of the keyboard. In the keyPressed() method of the listener, we inform the racquet that a key has been pressed by calling racquet.keyPressed(e). We do the same for keyReleased(). With this the sprite "racquet", will know when a key has been pressed. Let´s look now at the classes "Ball" and "Racquet".

```
import java.awt.*;

public class Ball {
        int x = 0;
        int y = 0;
        int xa = 1;
        int ya = 1;
        private Game game;

        public Ball(Game game) {
                this.game= game;
        }

        void move() {
                if (x + xa < 0)
                        xa = 1;
                if (x + xa > game.getWidth() - 30)
                        xa = -1;
                if (y + ya < 0)
                        ya = 1;
                if (y + ya > game.getHeight() - 30)
                        ya = -1;

                x = x + xa;
                y = y + ya;
        }

        public void paint(Graphics2D g) {
                g.fillOval(x, y, 30, 30);
        }
}
```

The "Ball" class hasn´t got any changes. Let´s compare it with the class "Racquet":

```
import java.awt.*;
import java.awt.event.*;

public class Racquet {
        int x = 0;
        int xa = 0;
        private Game game;

        public Racquet(Game game) {
                this.game= game;
        }

        public void move() {
                if (x + xa > 0 && x + xa < game.getWidth()-60)
                        x = x + xa;
```

```
        }

        public void paint(Graphics2D g) {
                g.fillRect(x, 330, 60, 10);
        }

        public void keyReleased(KeyEvent e) {
                xa = 0;
        }

        public void keyPressed(KeyEvent e) {
                if (e.getKeyCode() == KeyEvent.VK_LEFT)
                        xa = -1;
                if (e.getKeyCode() == KeyEvent.VK_RIGHT)
                        xa = 1;
        }
}
```

Unlike "Ball", "Racquet" hasn´t got any properties for the position "y" or for the speed "ya". This is because the racquet doesn´t change its vertical position; it will only move left or right, never up or down. In the pait method, the g.fillRect(x, 330, 60, 10)instruction defines a rectangle of 60 by 10 pixels in the position (x,y)=(x,330). As we can see "x" can change but "y" is fixed in 330 pixels from the top border of the canvas.

The move() method is similar to the one in "Ball" as it increases in "xa" the position "x" and controls that the sprite doesn´t go out of the borders.

```
        public void move() {
                if (x + xa > 0 && x + xa < game.getWidth()-60)
                        x = x + xa;
        }
```

In the beginning the value of "x" is zero, which indicates that the racquet will be in the left border of the canvas. "xa" is also initialize to zero, which makes the racquet look static in the beginning, because x = x + xa won´t change "x" while "xa" is zero.

When someone presses a key, the keyPressed method of "Racquet" will be called and this will set "xa" to 1, if the key pressed is the right direction (KeyEvent.VK_RIGHT), what will move the racquet to the right the next time the move method is called (remember x = x + xa). In the same way if we press the key KeyEvent.VK_LEFT it will move to the left.

```
        public void keyPressed(KeyEvent e) {
                if (e.getKeyCode() == KeyEvent.VK_LEFT)
                        xa = -1;
                if (e.getKeyCode() == KeyEvent.VK_RIGHT)
                        xa = 1;
        }
```

When a key is released, the method keyReleased is called and "xa" changes its value to zero, which makes the racquet stop.

```
        public void keyReleased(KeyEvent e) {
                xa = 0;
        }
```

If we run the example we can see how the ball moves bouncing against the borders and the racquet moving when we press the direction keys. When the ball collides with the racquet, it goes through as if it didn´t exist. In the next tutorial we will see how to make the ball bounce on the racquet.

Collision detection

In this tutorial we will learn how to detect when a sprite collides with another. In our game we will make the ball bounce on the racquet. We will also make the game finish if the ball gets to the lower border of the canvas, showing a popup window with the classic message "Game Over".

Game Over

Below we can see our class Game, which is exactly the same as the previous one, with the only difference that this one has a gameOver() method;

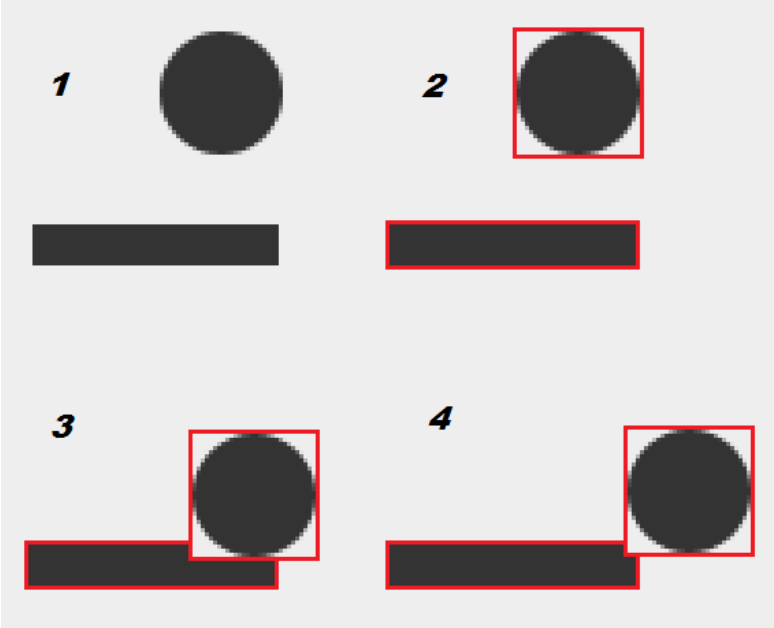
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Game extends JPanel {
        Ball ball = new Ball(this);
        Racquet racquet = new Racquet(this);

        public Game() {
                addKeyListener(new KeyListener() {
                        public void keyTyped(KeyEvent e) {
                        }

                        public void keyReleased(KeyEvent e) {
                                racquet.keyReleased(e);
                        }

                        public void keyPressed(KeyEvent e) {
                                racquet.keyPressed(e);
                        }
                });
                setFocusable(true);
        }

        private void move() {
                ball.move();
                racquet.move();
        }

        public void paint(Graphics g) {
                super.paint(g);
                Graphics2D g2d = (Graphics2D) g;
                g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                                RenderingHints.VALUE_ANTIALIAS_ON);
                ball.paint(g2d);
                racquet.paint(g2d);
        }

        public void gameOver() {
                JOptionPane.showMessageDialog(this, "Game Over", "Game Over",
JOptionPane.YES_NO_OPTION);
                System.exit(ABORT);
        }

        public static void main(String[] args) throws InterruptedException {
                JFrame frame = new JFrame("Mini Tennis");
                Game game = new Game();
                frame.add(game);
                frame.setSize(300, 400);
                frame.setVisible(true);
```

```
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                while (true) {
                        game.move();
                        game.repaint();
                        Thread.sleep(10);
                }
        }
}
```

The gameOver() method launches a popup using JOptionPane.showMessageDialog with the message "Game Over" and an "Accept" button. After the popup, System.exit(ABORT) makes the program finish. The gameOver() method is public, because it will be called from the sprite "Ball" when it detects that it has got to the lower border of the canvas.

Sprite collision

To detect the collision between the ball and the racquet we will use rectangles. In the case of the ball we will use a square around the ball as you can see in the figure 2.



The class java.awt.Rectangle has an intersects method(Rectangle r) which returns true when two rectangles occupy the same space, like in the case of the figure 3 or 4. This method is not quite exact, because as you can see in the figure 4, the ball doesn´t touch the racquet, but for our example it it more than enough.

Below we can see the Racquet class, with the only difference that we have added a getBounds() method, which returns a rectangle type of object, indicating the position of the racquet. This method will be used by the sprite "Ball", to know the position of the racquet and in this way to detect the collision.

```
import java.awt.*;
import java.awt.event.*;

public class Racquet {
        private static final int Y = 330;
        private static final int WIDTH = 60;
        private static final int HEIGHT = 10;
```

```
        int x = 0;
        int xa = 0;
        private Game game;

        public Racquet(Game game) {
                this.game = game;
        }

        public void move() {
                if (x + xa > 0 && x + xa < game.getWidth() - WIDTH)
                        x = x + xa;
        }

        public void paint(Graphics2D g) {
                g.fillRect(x, Y, WIDTH, HEIGHT);
        }

        public void keyReleased(KeyEvent e) {
                xa = 0;
        }

        public void keyPressed(KeyEvent e) {
                if (e.getKeyCode() == KeyEvent.VK_LEFT)
                        xa = -1;
                if (e.getKeyCode() == KeyEvent.VK_RIGHT)
                        xa = 1;
        }

        public Rectangle getBounds() {
                return new Rectangle(x, Y, WIDTH, HEIGHT);
        }

        public int getTopY() {
                return Y;
        }
}
```

Another small change is the inclusion of constants:

```
        private static final int Y = 330;
        private static final int WIDTH = 60;
        private static final int HEIGH = 20;
```

As we said before, the value of the "y" position, was fixed to 330. This value is used both in the paint method as in getBounds. When we create a constant, the good thing is that if we want to change the value, we only have to change it in one place. In this way we avoid the possible error of changing it in one place and not changing it in another.

The way of defining a constant is declaring a "static final" property and writing it in upper case. The compiler allows us to use lower case, but the standar says we use upper case for the constants.

Lastly, the Ball class:

```
import java.awt.*;

public class Ball {
        private static final int DIAMETER = 30;
        int x = 0;
```

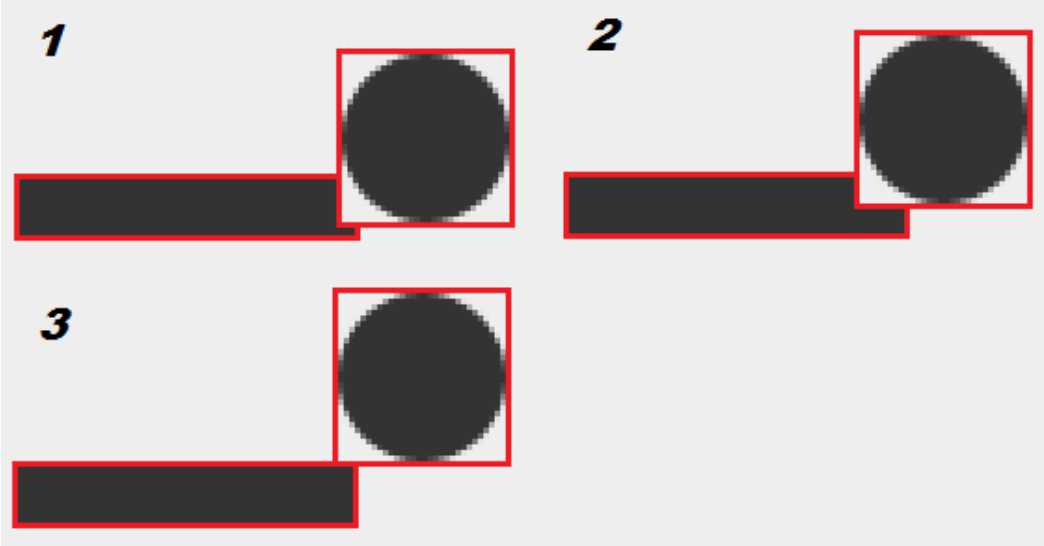
```
        int y = 0;
        int xa = 1;
        int ya = 1;
        private Game game;

        public Ball(Game game) {
                this.game= game;
        }

        void move() {
                if (x + xa < 0)
                        xa = 1;
                if (x + xa > game.getWidth() - DIAMETER)
                        xa = -1;
                if (y + ya < 0)
                        ya = 1;
                if (y + ya > game.getHeight() - DIAMETER)
                        game.gameOver();
                if (collision()){
                        ya = -1;
                        y = game.racquet.getTopY() - DIAMETER;
                }
                x = x + xa;
                y = y + ya;
        }

        private boolean collision() {
                return game.racquet.getBounds().intersects(getBounds());
        }

        public void paint(Graphics2D g) {
                g.fillOval(x, y, DIAMETER, DIAMETER);
        }

        public Rectangle getBounds() {
                return new Rectangle(x, y, DIAMETER, DIAMETER);
        }
}
```

In a similar way, we have included the getBounds() method and the DIAMETER constant to the class "Racquet".

More interesting is the inclusion of the new method called collision() which returns true, if the rectangle occupied by the racquet "game.racquet.getBounds()" intersects with the rectangle occupied by the ball "getBounds()".

```
        private boolean collision() {
                return game.racquet.getBounds().intersects(getBounds());
        }
```

If the collision takes place, we will change the direction and the position of the ball. If the collision occurs on the side (figure 1), the ball could be several pixels below the upper side of the racquet. In the following game loop, even if the ball moved upwards (figure 2), it could still be in collision with the racquet.



To avoid this, we will place the ball on top of the racquet (figure 3) using:

```
y = game.racquet.getTopY() - DIAMETER;
```

The "Racquet" getTopY() method gives us the position in the "y" axis of the upper part of the racquet, and if we discount the DIAMETER, we obtain the exact position where to put the ball so that it is on top of the ball.

Lastly, it is the move() method of the "Ball" class which uses the new methods collision() and gameOver() of the "Game" class. The bounce when it gets to the lower border is replaced by a call to game.gameOver().

```
if (y + ya > game.getHeight() - DIAMETER)
        game.gameOver();
```

And including a new conditional using the collision() method, we get the ball to bounce upwards if it collides with the racquet:

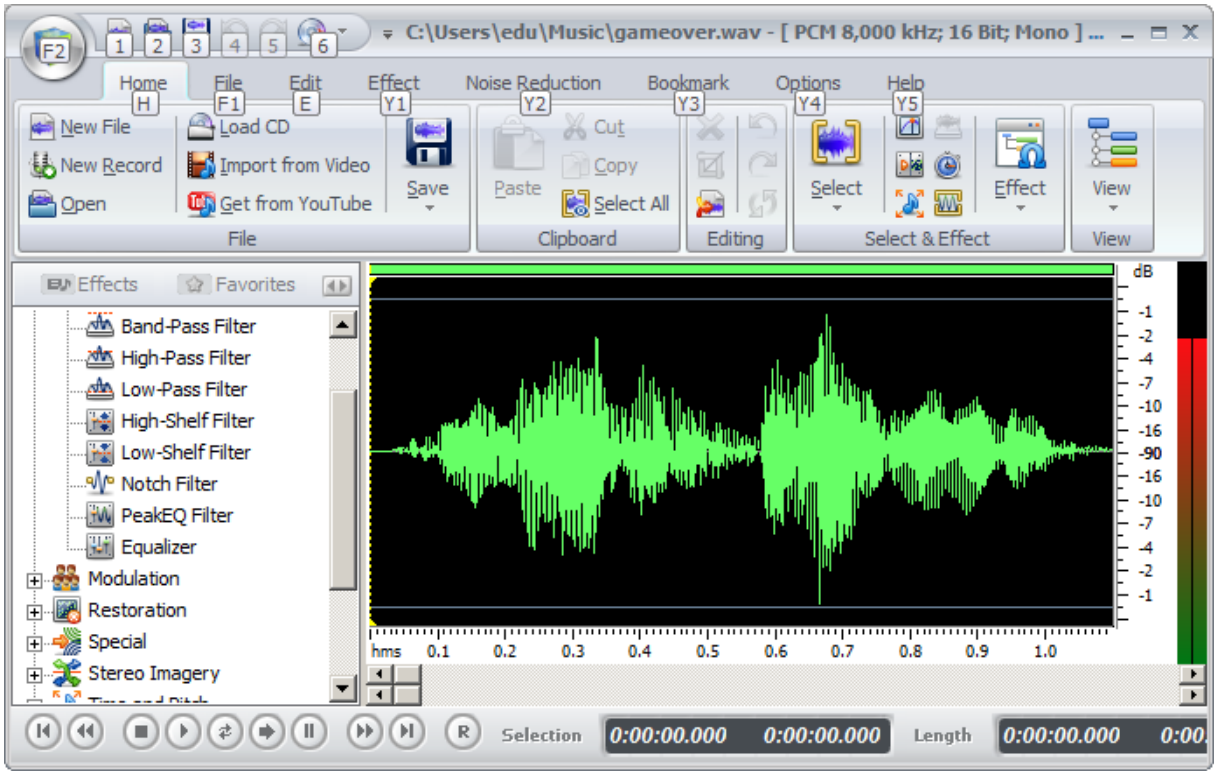
```
if (collision())
        ya = -1;
```

Adding sound to our game

A game with no sound is not complete. In this tutorial we will add background music, the noise of the bouncing of the ball and a "Game Over" with funny voice at the end of the game. To avoid copyright problems we are going to create the sounds ourselves.

Creating sounds

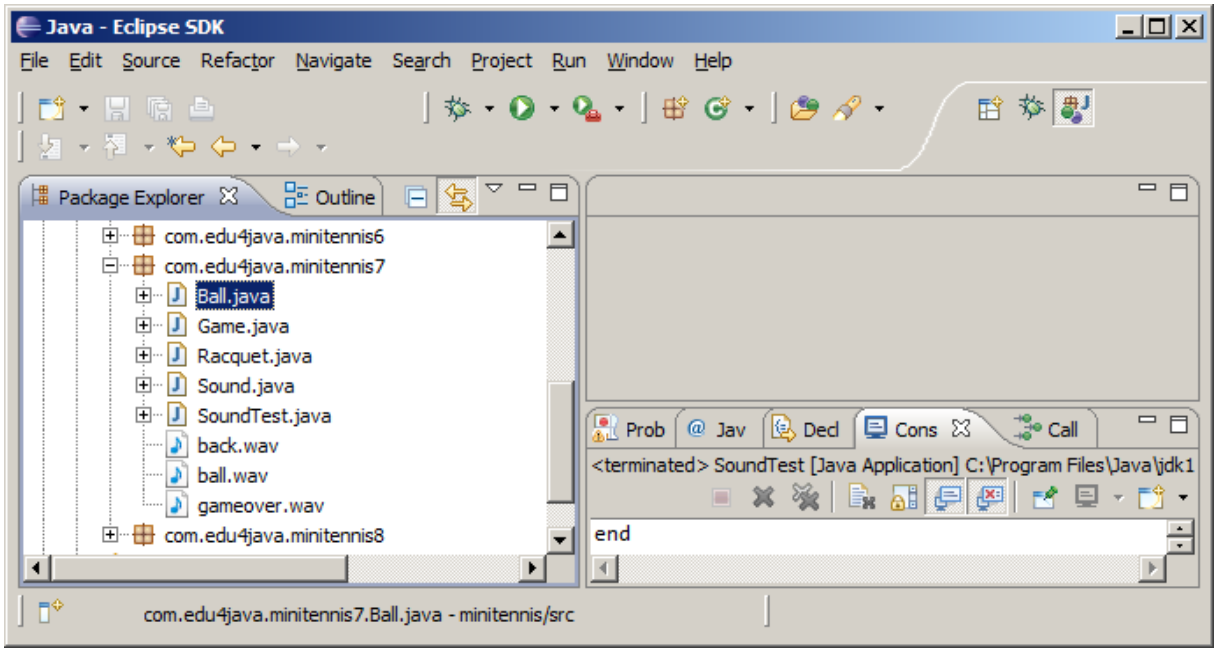
To create the sounds I looked up in Google to find a "free audio editor" and I found http://free-audio-editor.com/. I have to say that the free version of this product is powerful and easy to use.



With this editor I have created the archives: back.wav, gameover.wav, ball.wav. In the youtube video you can see how I did it and you can create them yourselves. You can also download those three and use them, which I now declare them copyright free. What you have to do is copy these archives to the default package.

Downloading sounds

If you don't want to create any sounds, you can also look online for sounds, for example: http://www.freesound.org



Play sounds using AudioClip

To play these sound archives we will use the AudioClip class. We will create AudioClip objects, using the static method: Applet.newAudioClip(URL url) of the Applet class. This method needs un URL object which indicates where is the audio archive we are wanting to load and play. The following instruction creates a new URL object, using a location in Internet:

```
URL url = new URL("http://www.edu4java.com/es/game/sound/back.wav");
```

The next instruction uses a directory inside the local archive system:

```
URL url = new
URL("file:/C:/Users/Eli/workspace/minitennis/src/com/edu4java/minitennis7/back.
wav");
```

We will look for our archive using the classpath. This is the system which uses java to load the classes or more specifically the *.class archives which define the classes of the program. To obtain an URL from the classpath we use the getResource(String name) method of the Class class, where "name" is the name of the archive we want to obtain.

Below we can see two ways of how to obtain the URL of the "back.wav" archive, which is located in the same package as the SoundTest class or what is the same, in the same directory as the SoundTest.class archive.

```
URL url = SoundTest.class.getResource("back.wav");
```

```
URL url = new SoundTest().getClass().getResource("back.wav");
```

Both "SoundTest.class" and "new SoundTest().getClass()" gives us a class object which has the getResource method we want to use.

I have created the SoundTest class to show you how does AudioClip work but it isn´t necessary for our game. Below we can see the SoundTest source code complete:

```
import java.applet.*;
import java.net.URL;

public class SoundTest {
        public static void main(String[] args) throws Exception {
                URL url = SoundTest.class.getResource("back.wav");
                AudioClip clip = Applet.newAudioClip(url);
                AudioClip clip2 = Applet.newAudioClip(url);
                clip.play();
                Thread.sleep(1000);
                clip2.loop();
                Thread.sleep(20000);
                clip2.stop();

                System.out.println("end");
        }
}
```

This is the way to obtain the back.wav archive from the classpath. The classpath is the directories and archives *.jar collection from where our program can read the classes (*.class archives).

One advantage of this metodology is that we only have to indicate the position of the archive regarding the class which uses it. In our case as it is in the same package we only have to write "back.wav". Another advantage is that the sound archives can be included in the *.jar archive. We´ll see more about *.jar archives later on. Once we have the URL object we can create AudioClip objects using Applet.newAudioClip(url).

```
                AudioClip clip = Applet.newAudioClip(url);
                AudioClip clip2 = Applet.newAudioClip(url);
```

The AudioClip object has a play() method which starts an independent thread which plays the audio of the archive just once. To play the audio more than once we can use the loop() method of AudioClip which will play the sound repeatedly until the stop() method is called over the same AudioClip object.

Two audioClips can play at the same time. In the example I create two audioClips with the same audio: clip and clip2. I play "clip" with play, I wait a second Thread.sleep(1000) and play clip2 with loop. The result is a mixture of the two audios. Lastly, after 20 seconds Thread.sleep(20000) I call clip2.stop() and I stop the repetition of clip2.

Creating a Sound class for our game

To keep the audioclips of our game we create a Sound class, which we´ll have a constant with an audioclip for each of the sounds we use. These constants are public so that any object which have access to them, can play them. For example, in the Ball class we can play the sound of the bouncing of the ball using Sound.BALL.play() at the moment we know the ball changes its direction.

```
import java.applet.*;

public class Sound {
	public static final AudioClip BALL =
Applet.newAudioClip(Sound.class.getResource("ball.wav"));
	public static final AudioClip GAMEOVER =
Applet.newAudioClip(Sound.class.getResource("gameover.wav"));
	public static final AudioClip BACK =
Applet.newAudioClip(Sound.class.getResource("back.wav"));
}
```

The audioclips objects will be created when the Sound class loads, the first time someone uses the Soung class. From this moment on, they will be re-used. Let´s now look at the modifications in the Game class:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Game extends JPanel {
	Ball ball = new Ball(this);
	Racquet racquet = new Racquet(this);

	public Game() {
		addKeyListener(new KeyListener() {
			public void keyTyped(KeyEvent e) {
			}

			public void keyReleased(KeyEvent e) {
				racquet.keyReleased(e);
			}

			public void keyPressed(KeyEvent e) {
				racquet.keyPressed(e);
			}
		});
		setFocusable(true);
		Sound.BACK.loop();
	}

	private void move() {
		ball.move();
		racquet.move();
	}

	public void paint(Graphics g) {
		super.paint(g);
		Graphics2D g2d = (Graphics2D) g;
		g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
				RenderingHints.VALUE_ANTIALIAS_ON);
		ball.paint(g2d);
		racquet.paint(g2d);
	}
```

```
        public void gameOver() {
                Sound.BACK.stop();
                Sound.GAMEOVER.play();
                JOptionPane.showMessageDialog(this, "Game Over", "Game Over",
JOptionPane.YES_NO_OPTION);
                System.exit(ABORT);
        }

        public static void main(String[] args) throws InterruptedException {
                JFrame frame = new JFrame("Mini Tennis");
                Game game = new Game();
                frame.add(game);
                frame.setSize(300, 400);
                frame.setVisible(true);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                while (true) {
                        game.move();
                        game.repaint();
                        Thread.sleep(10);
                }
        }
}
```

In the last line of the Game class constructor, we add Sound.BACK.loop(), which will initiate the playing of our background music and will play repeatedly till it gets to the gameOver() method, where we stop the background music with Sound.BACK.stop(). After Sound.BACK.stop() and before the popup, we inform that the game is over playing "Game Over" Sound.GAMEOVER.play().

In the Ball class, we change the move() method so that it plays Sound.BALL when the ball bounces.

```
import java.awt.*;

public class Ball {
        private static final int DIAMETER = 30;

        int x = 0;
        int y = 0;
        int xa = 1;
        int ya = 1;
        private Game game;

        public Ball(Game game) {
                this.game = game;
        }

        void move() {
                boolean changeDirection = true;
                if (x + xa < 0)
                        xa = 1;
                else if (x + xa > game.getWidth() - DIAMETER)
                        xa = -1;
                else if (y + ya < 0)
                        ya = 1;
                else if (y + ya > game.getHeight() - DIAMETER)
                        game.gameOver();
                else if (collision()){
                        ya = -1;
                        y = game.racquet.getTopY() - DIAMETER;
```

```
                } else
                        changeDirection = false;

                if (changeDirection)
                        Sound.BALL.play();
                x = x + xa;
                y = y + ya;
        }

        private boolean collision() {
                return game.racquet.getBounds().intersects(getBounds());
        }

        public void paint(Graphics2D g) {
                g.fillOval(x, y, DIAMETER, DIAMETER);
        }

        public Rectangle getBounds() {
                return new Rectangle(x, y, DIAMETER, DIAMETER);
        }
}
```

What I did in move() is add a changeDirection variable, which I initialize to true. Adding an "else" to every "if" and writing a changeDirection = false which only will runs if none of the conditions in the "if" are true, we will know if the ball has bounced. If the ball has bounced, changeDirection will be true and Sound.BALL.play() will be executed.

Adding punctuation and speed increase

Every game needs a measurement of success. In our case we will include in the top left corner of the screen the punctuation, which will be the number of times we are able to hit the ball with the racquet. On the other hand, the game should be a bit more complicated each time, so that the player doesn´t get bored.

The moving objects of the game are the ball and the racquet. Changing the speed of these two objects, we will modify the speed of the game. We are going to include a property called "speed" in the Game class to keep the speed of the game. The property "speed" will be 1 initially, and it will increase each time we hit the ball with the racquet.

For the puctuation we need another property to increase each time we hit the ball. Instead of creating a new property, I am going to re-use "speed". The only inconvenience is that the punctuations start in 0 and not in 1 as "speed". The solution I thought of was, add a getScore() method which returns the value of speed, minus 1.

```
        private int getScore() {
                return speed - 1;
        }
```

Let´s see what are the modifications made to the Game class:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Game extends JPanel {

        Ball ball = new Ball(this);
        Racquet racquet = new Racquet(this);
        int speed = 1;

        private int getScore() {
                return speed - 1;
        }

        public Game() {
                addKeyListener(new KeyListener() {
                        public void keyTyped(KeyEvent e) {
                        }

                        public void keyReleased(KeyEvent e) {
                                racquet.keyReleased(e);
                        }

                        public void keyPressed(KeyEvent e) {
                                racquet.keyPressed(e);
                        }
                });
                setFocusable(true);
                Sound.BACK.loop();
        }

        private void move() {
                ball.move();
                racquet.move();
        }
```

```
        public void paint(Graphics g) {
                super.paint(g);
                Graphics2D g2d = (Graphics2D) g;
                g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                                RenderingHints.VALUE_ANTIALIAS_ON);
                ball.paint(g2d);
                racquet.paint(g2d);

                g2d.setColor(Color.GRAY);
                g2d.setFont(new Font("Verdana", Font.BOLD, 30));
                g2d.drawString(String.valueOf(getScore()), 10, 30);
        }

        public void gameOver() {
                Sound.BACK.stop();
                Sound.GAMEOVER.play();
                JOptionPane.showMessageDialog(this, "your score is: " +
getScore(),
                                "Game Over", JOptionPane.YES_NO_OPTION);
                System.exit(ABORT);
        }

        public static void main(String[] args) throws InterruptedException {
                JFrame frame = new JFrame("Mini Tennis");
                Game game = new Game();
                frame.add(game);
                frame.setSize(300, 400);
                frame.setVisible(true);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                while (true) {
                        game.move();
                        game.repaint();
                        Thread.sleep(10);
                }
        }
}
```

To paint the punctuation in the top left corner, we add the following code at the end of the paint method:

```
                g2d.setColor(Color.GRAY);
                g2d.setFont(new Font("Verdana", Font.BOLD, 30));
                g2d.drawString(String.valueOf(getScore()), 10, 30);
```

In the first line we choose the color; grey, in the second line the type of letter; Verdana, bold type of 30 pixeles and finally the position (x,y) = (10,30), where we paint the punctuation.

In the gameOver() method, we modify the second parameter to show the punctuation achieved:

```
                JOptionPane.showMessageDialog(this, "your score is: " +
getScore(),
                                "Game Over", JOptionPane.YES_NO_OPTION);
```

The move() method of the Ball class has been modified to take into account the new property "game.speed". When the ball changed direction, the properties of speed "xa" and "ya" were changed to 1

or -1. Now, taking into account speed, these properties, change to game.speed or -game.speed. We have also added in the conditional if(collision()), that "speed" increases "game.speed++".

```
import java.awt.*;

public class Ball {
	private static final int DIAMETER = 30;

	int x = 0;
	int y = 0;
	int xa = 1;
	int ya = 1;
	private Game game;

	public Ball(Game game) {
		this.game = game;
	}

	void move() {
		boolean changeDirection = true;
		if (x + xa < 0)
			xa = game.speed;
		else if (x + xa > game.getWidth() - DIAMETER)
			xa = -game.speed;
		else if (y + ya < 0)
			ya = game.speed;
		else if (y + ya > game.getHeight() - DIAMETER)
			game.gameOver();
		else if (collision()){
			ya = -game.speed;
			y = game.racquet.getTopY() - DIAMETER;
			game.speed++;
		} else
			changeDirection = false;

		if (changeDirection)
			Sound.BALL.play();
		x = x + xa;
		y = y + ya;
	}

	private boolean collision() {
		return game.racquet.getBounds().intersects(getBounds());
	}

	public void paint(Graphics2D g) {
		g.fillOval(x, y, DIAMETER, DIAMETER);
	}

	public Rectangle getBounds() {
		return new Rectangle(x, y, DIAMETER, DIAMETER);
	}
}
```

Below we can see the class Racquet:

```
import java.awt.*;
import java.awt.event.*;

public class Racquet {
```

```
        private static final int Y = 330;
        private static final int WITH = 60;
        private static final int HEIGHT = 10;
        int x = 0;
        int xa = 0;
        private Game game;

        public Racquet(Game game) {
                this.game = game;
        }

        public void move() {
                if (x + xa > 0 && x + xa < game.getWidth() - WITH)
                        x = x + xa;
        }

        public void paint(Graphics2D g) {
                g.fillRect(x, Y, WITH, HEIGHT);
        }

        public void keyReleased(KeyEvent e) {
                xa = 0;
        }

        public void keyPressed(KeyEvent e) {
                if (e.getKeyCode() == KeyEvent.VK_LEFT)
                        xa = -game.speed;
                if (e.getKeyCode() == KeyEvent.VK_RIGHT)
                        xa = game.speed;
        }

        public Rectangle getBounds() {
                return new Rectangle(x, Y, WITH, HEIGHT);
        }

        public int getTopY() {
                return Y - HEIGHT;
        }
}
```

Here, the modification is similar to Ball. In the keyPressed(KeyEvent e) method, the modification of "speed" xa changes from -1 and 1 to -game.speed and game.speed.

Note: "Java Beans" standard says that the access to the property "game.speed" should be done using a method in this way "game.getSpeed()". The direct access to a property is not considered correct in business java. Strangely enough, in the area of the games development it is very common and it is justified because of its eficiency. This is very important in mobile programming due to the shortage of resources.

Creating an executable jar file.

In this tutorial we will see how to create an executable file for a java application, in particular for our game.

JAR file

A jar file is a compressed file with the zip algorithm of compression, which can contain:

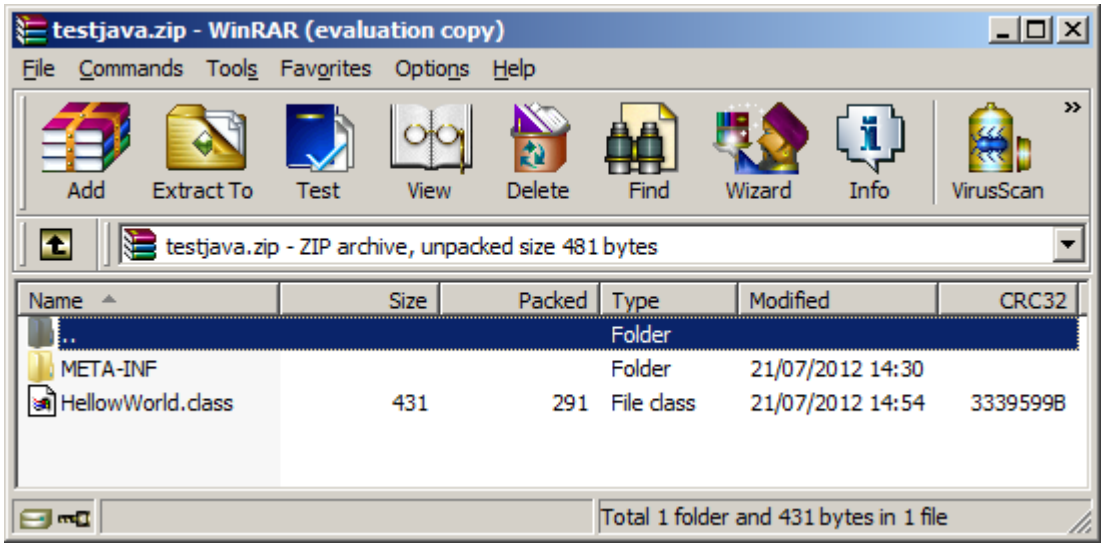
1. The *.class files which are generated from the compilation of the *.java files, which make up our application.
2. The resources files needed by our application (For example the sound file *.wav)
3. Optionally, we can include the source code files *.java
4. Optionally, we can have a configuration file "META-INF/MANIFEST.MF".

Creating an executable JAR file

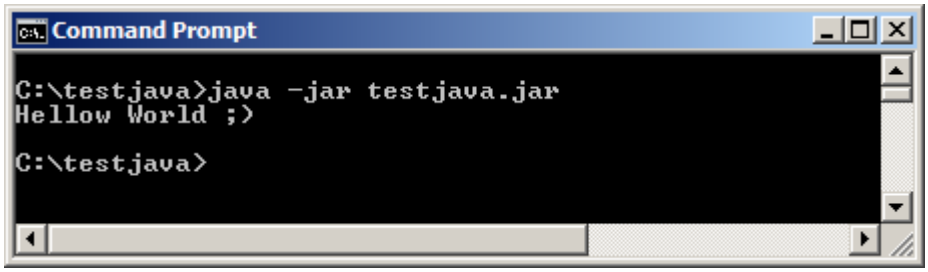
For the jar file to be executable, we have to include in the MANIFEST.MF file, a line indicating the class which contains the static method main() which will be used to start the aplication. In our last example it would be like this:

```
Main-Class: HelloWorld
```

It is important to highlight, that at the end of the line, we have to add a carriage return so that it works. I invite you to create a testjava.zip file containing a HelloWorld.class file, the META-INF directory and inside it a MANIFEST.MF file with the Main-Class line: HelloWorld. For this you can use the Winzip or WinRAR programs, which you can download free (look for it in Google).



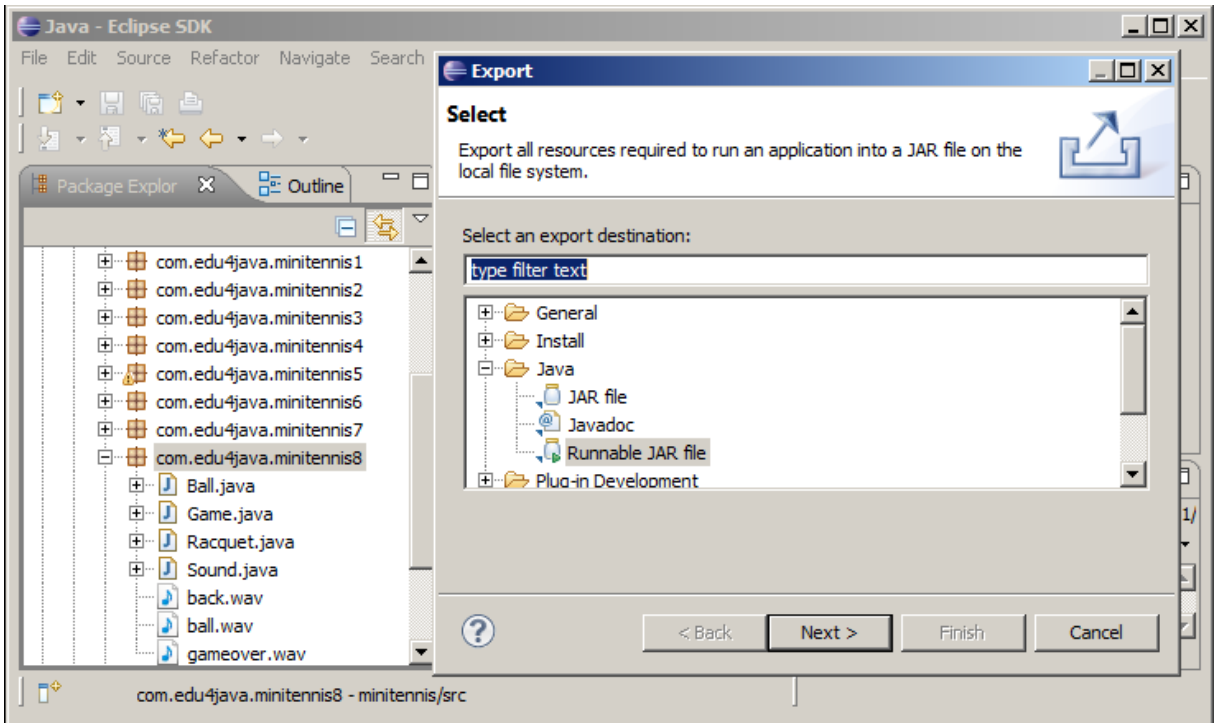
Once you have created the testjava.zip file, we change its name for testjava.jar and we execute it from the command line:



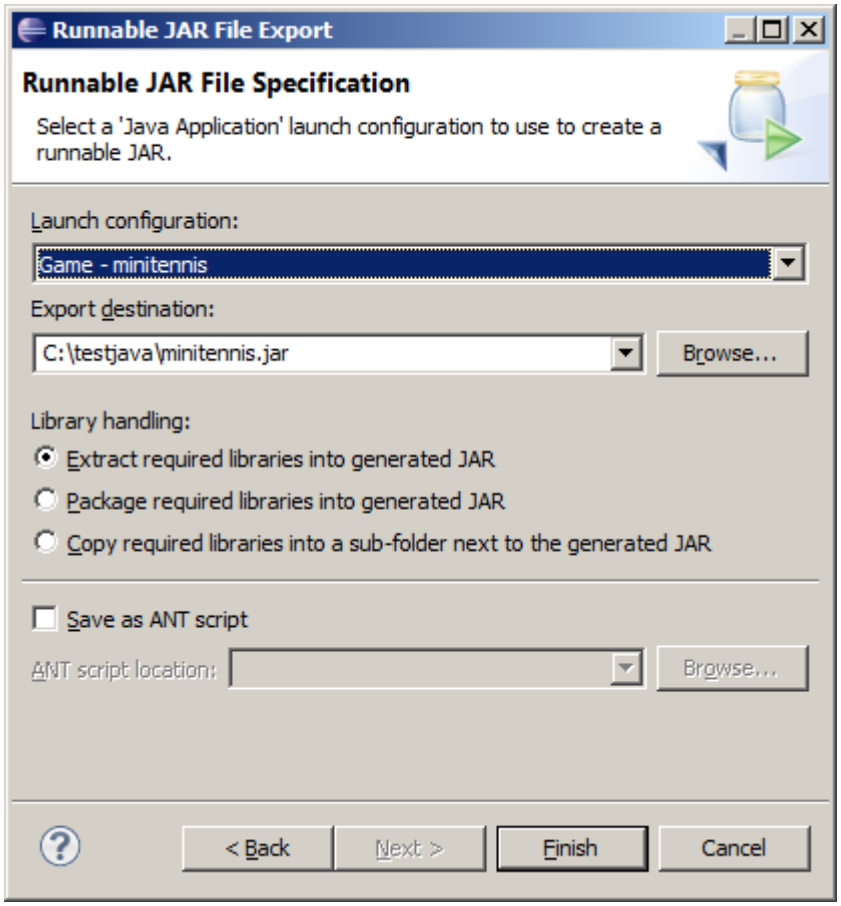
We can also execute it with a double click on the JAR file.

How to create an executable JAR file from eclipse.

To create an executable JAR we go to File-Export and select Runnable JAR file



As we can see below, in "Launch configuration" we select the one we use to execute the final version of our application and in "Export destination" we indicate where we want to save our JAR and with what name:

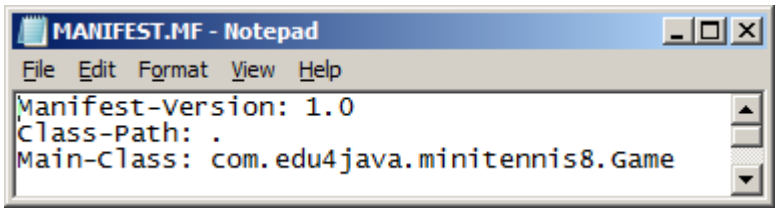


If java is correctly installed over Windows, a doble click on minitennis.jar would be enough to execute our application.

Looking at minitennis.jar

If we decompress our minitennis.jar file, we will find the *.class files which make up our game. This files are inside the directory tree with the names of the java packages which contain the classes.

Inside META-INF/MANIFEST.MF, we can see in the last line, how the game must start with the main() method of the Game class, which is in the com.edu4java.minitennis8 package.



Eclipse does a great job, compiling, executing and creating JAR files, but is good to understand that eclipse uses the java installation in a similar way as in our HelloWorld example.