THE *Ball* CLASS

Create a **Bouncing Ball** program that uses a **Ball** class that moves and bounces a ball within a **JPanel**. Use the following UML diagram to create the **Ball** class:

Ball
- xPos, yPos, ballWidth, ballHeight: int - imgBall: ImageIcon - c: Color - useImage: boolean
Ball () Ball(xPos: int, yPos: int, imgBall: ImageIcon) Ball(xPos: int, yPos: int, ballWidth: int, ballHeight: int, c: Color) + draw(g2: Graphics2D): void + getBallHeight(): int + getBallWidth(): int + getImage(): Image + getX(): int + getY(): int + move(): void + move(xDir: int, yDir: int): void + setBallWidth(width: int): void + setBallHeight(height: int): void + setColor(c: Color): void + setImageIcon(imgBall: ImageIcon): void + setX(x: int): void + setY(y: int): void

The **Ball** class will be responsible for knowing its own location (i.e. x- and y-position), its width and height, its colour, the image file in case the user wants to use an image of a ball, and a boolean which will help us determine whether we're using an image or not. We are going to implement this responsibility by providing the **Ball** class with variables that will store each of these values.

We are going to declare the variables as follows:

```
public class Ball {

   private int xPos, yPos, ballWidth, ballHeight;
   private Color c;
   private ImageIcon imgBall;
   private boolean useImage;
}
```

IMPLEMENTING CONSTRUCTORS

We are going to create a **no-arg** constructor that does not take any parameters and initializes our class fields or variables with default values. It's going to look something like this:

```
public Ball() {

   this.xPos = 0;
   this.yPos = 0;
   this.ballWidth = 50;
   this.ballHeight = 50;
   this.c = Color.RED;
   this.useImage = false;
   this.imgBall = null;
}
```

In the above example, the position of the ball is going to start at (0, 0) and it's going to be 50 pixels wide by 50 pixels high. Since we are using a **no-arg** constructor for the **Ball** class and no parameters are being passed to the constructor, we need to assign default values for the **Ball** object that will be created. When initializing the values of class fields, you should refer to them using the word **this** in order to distinguish them from other variables that are passed to constructors and methods within the class.

OVERLOADING CONSTRUCTORS

One way to overload the constructor and provide more flexibility to the **Ball** class is to provide any or all of these values. For example, we can provide an implementation of the constructor that takes five parameters:

```
public Ball(int xPos, int yPos, int ballWidth, int ballHeight,
   Color c) {

   this.xPos = xPos;
   this.yPos = yPos;
   this.ballWidth = ballWidth;
   this.ballHeight = ballHeight;
   this.c = c;
   this.useImage = false;
   this.imgBall = null;
}
```

When you initialize a **Ball** object using this constructor, you will explicitly pass the x-position, the y-position, the width, the height and the colour of the new **Ball**. So now we can create a **Ball** object in our program as follows:

```
Ball b = new Ball(0, 20, 30, 30, Color.BLUE);
```

Let's create a third constructor that allows the user to create a **Ball** using the image of a ball instead of a filled ellipse.

```
public Ball(int xPos, int yPos, ImageIcon imgBall) {
```

```
        this.xPos = xPos;
        this.yPos = yPos;
        this.imgBall = imgBall;
        this.ballWidth = imgBall.getIconWidth();
        this.ballHeight = imgBall.getIconHeight();
        this.useImage = true;
        this.c = null;
    }
```

In the above example, we are initializing the x- and y-position of the ball according to the first two parameters that are passed to the constructor. The variable **imgBall** is going to equal the **ImageIcon** that is passed to the constructor, while the width and height variables are going to equal the width and height of the image. Since we're using an image in this version of the constructor, we don't need a colour which is why we assign it the value **null**.

IMPLEMENTING METHODS

Now that we have our class variables and constructors in place, it's time to create methods for our **Ball** class. We'll start with **accessor** methods, which are used to return the values of our instance variables.

```
    public int getX() {

        return xPos;
    }

    public int getY() {

        return yPos;
    }

    public int getBallWidth() {

        return ballWidth;
    }

    public int getBallHeight() {

        return ballHeight;
    }

    public Image getImage() {

        return imgBall.getImage();
    }
```

Now we should create **mutator** methods, which are used to change the values of our class fields. Since the ball's x-position and y-position will be changing all the time, we need methods that are going to change those values.

```
    public void setBallWidth(int width) {
```

```
      this.ballWidth = width;
   }

   public void setBallHeight(int height) {

      this.ballHeight = height;
   }

   public void setColor(Color col) {

      this.c = col;
   }

   public void setImageIcon(ImageIcon imgBall) {

      this.imgBall = imgBall;
   }

   public void setX(int x) {

      this.xPos = x;
   }

   public void setY(int y) {

      this.xPos = y;
   }
```

We are also going to need a **draw()** method which draws the **Ball** by calling the **fill()** method on the **Graphics2D** object that is passed as a parameter.

```
   public void draw(Graphics2D g2) {

      if (useImage == false)
      {
         g2.setColor(c);
         g2.fill(new Ellipse2D.Double(xPos, yPos, ballWidth,
            ballHeight));
      }
      else
      {
         g2.drawImage(getImage(), x, y, null);
      }
   }
```

In our **drawImage()** method, we are either going to draw an ellipse or we are going to draw an image of a ball depending on which constructor is used to create a **Ball** object.

OVERLOADING METHODS

In order to get the ball to move, we are going to need to create a method called **move()** that will change the x-position and the y-position of the ball. The first **move()** method will move the ball 1 pixel along the x-axis and 2 pixels along the y-axis by default.

```
public void move() {

   this.xPos += 1;
   this.yPos += 2;
}
```

Let's create a second **move()** method that allows one to specify how many pixels along the x- and y-axis to move the ball.

```
public void move(int xDir, int yDir) {

   this.xPos += xDir;
   this.yPos += yDir;
}
```

USING THE *BALL* CLASS IN A PROGRAM

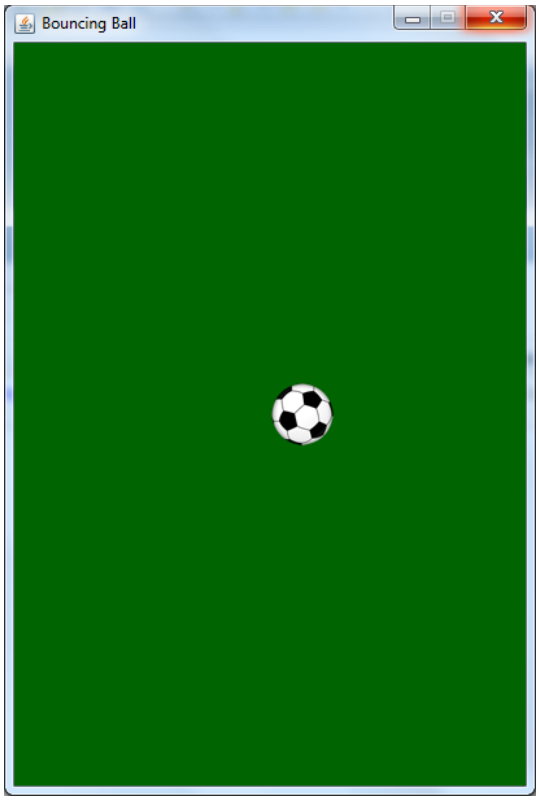
Now that we have factored all of the responsibility for the **Ball** out of the program itself, we can use the **Ball** class within the program as you might use any other object.

We will start by declaring an instance of the **Ball** as a member of our program:

```
public class BouncingBall extends JPanel implements ActionListener {

   // Declare global variables
   private Ball ball;
   private int xDir, yDir;
   private Timer t;

   public static void main(String[] args) {

      new BouncingBall();
   }

   public BouncingBall() {

      // Declare and initialize Ball object
      ball = new Ball(0, 0, new ImageIcon("soccer_ball.png"));

      // Initialize the two variables that will determine how many
      pixels the ball will move horizontally and vertically
      xDir = 1;
      yDir = 3;
```

```
        // Initialize Timer object and set interval to 50 milliseconds
        t = new Timer(50, this);

        // Set properties of JPanel object
        setLayout(null);
        setBackground(new Color(0, 100, 0));

        // Declare, initialize and set properties of JFrame
        JFrame frame = new JFrame();
        frame.setContentPane(this);
        frame.setSize(400, 600);
        frame.setTitle("Bouncing Ball");
        frame.setResizable(false);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);

        // Start the timer
        t.start();
    }

    public void actionPerformed(ActionEvent e) {

            // Move the ball by resetting the x- and y-position of the
            ball
            ball.move(xDir, yDir);
            repaint();
    }

    public void paintComponent(Graphics g) {

        // Draw contents onto the panel
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        ball.draw(g2);
    }
}
```

When the program launches, the ball will start moving horizontally to the right and vertically to the bottom of the screen.

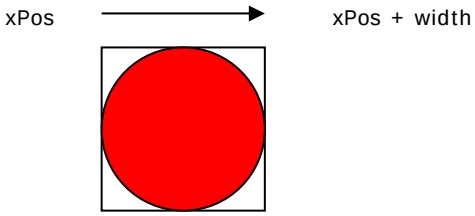


The problem, of course, is that the ball is going to eventually disappear from the screen. To solve this problem, we will need to check if the ball has banged into any of the walls (top, bottom, left or right). In order to do this, we will need to write the code inside the **actionPerformed()** method as follows:

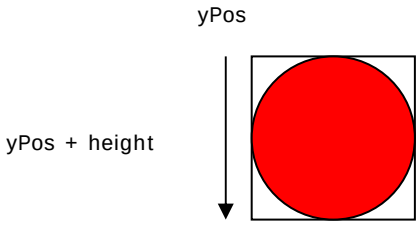
```
public void actionPerformed(ActionEvent e) {

   ball.move(xDir, yDir);

   if (ball.getX() <= 0 || ball.getX() + ball.getWidth() >=
      getWidth())
   {
      xDir = -xDir;
   }
   else if (ball.getY() <= 0 || ball.getY() + ball.getHeight() >=
      getHeight())
   {
      yDir = -yDir;
   }

   repaint();

}
```

Now when you run the program, the ball will check if it hits any of the four walls and reverse its position accordingly.

When checking if the ball reaches the right wall, we need to get the x-position of the ball and add the width because we want to check if the right side of the ball hits the wall. If we didn't add the width, we would be checking if the left side of the ball hits the wall.



When checking if the ball reaches the bottom wall, we need to do the same thing we did when checking if the ball hits the right wall with the only difference being that we need to use the y-position of the ball and the height of the panel.



ENCAPSULATING RESPONSIBILITY

Our program currently is responsible for moving the ball and then telling the ball to draw itself. Wouldn't it make more sense to make the **Ball** responsible for moving and drawing? Shouldn't we, in other words, make the **Ball** responsible for its own behaviour? There is no reason for a programmer using the **Ball** class to create a program to know how to move or draw the ball. This process of containing responsibilities in a class is called **encapsulation**.

We can encapsulate the entire process of moving and drawing the ball by creating a **move()** method that changes the x- and y-position of the ball and checks if it hits any of the four walls. We can make it move and draw by doing the following:

```
public void move(Graphics2D g2) {

   xPos += xDir;
   yPos += yDir;

   if (xPos <= 0 || xPos + width >= panelWidth)
   {
      xDir = -xDir;
   }
   else if (yPos <= 0 || yPos + height >= panelHeight)
   {
      yDir = -yDir;
   }

   draw(g2);
}
```

Now all we need the **draw()** method to do is draw the ball itself:

```
public void draw(Graphics2D g2) {

   if (useImage == false)
   {
      g2.setColor(c);
      g2.fill(new Ellipse2D.Double(xPos, yPos, width, height));
   }
   else
   {
      g2.drawImage(getImage(), x, y, null);
   }
}
```

In the program itself, we would simply tell the ball to move within the **paintComponent()** method and call the **repaint()** method in the **actionPerformed**() method:

```
public void paintComponent(Graphics g) {

   // Draw contents onto the panel
   super.paintComponent(g);
   Graphics2D g2 = (Graphics2D) g;
   ball.move(g2);
}

public void actionPerformed(ActionEvent e) {

   repaint();
}
```

Now a programmer can create a program using the **Ball** class and simply tell the **Ball** to move without really knowing how it is drawn, how it knows if it hits a wall, etc.