Writing classic “Breakout” in Java

Breakout was a game invented in 1976. The object of the game is simple: to bounce a ball in to a wall of bricks until all the bricks are destroyed. In this version…

In Java, we’re going to write this game with several classes, each that are responsible for a different part of the game. This is how modern software is decomposed, or “broken up”, in to small, manageable blocks.

# Breakout

The first class, Breakout, is the entry point for the game. If you’ve completed the Snake game, you can actually copy most of the Snake.java file to Breakout.java. This class creates a JFrame object, which is the window on the screen. The logic to run the game is in the runGame() method.

**package** elycc;

**import** java.awt.EventQueue;

**import** java.nio.file.Path;

**import** java.nio.file.Paths;

**import** javax.swing.JFrame;

/\*\*

\* Main game launcher.

\*/

**public** **class** Breakout {

/\*\*

\* Entry point for the game.

\* **@param** args command line parameters (unused).

\*/

**public** **static** **void** main(**final** String[] args) {

EventQueue.*invokeLater*(**new** Runnable() {

@Override

**public** **void** run() {

JFrame frame = **new** JFrame("Breakout");

frame.add(**new** Board());

frame.setResizable(**false**);

frame.pack();

frame.setLocationRelativeTo(**null**);

frame.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

frame.setVisible(**true**);

}

});

}

}

Note that the frame.add(new Board()) line won’t compile yet, because Java doesn’t have a definition for the Board class, so it can’t create an object that is an instance of the Board class.

Type in the listing for Board.java below. Before you start though, note the following:

1. To save you typing all the mouse handling methods at the bottom of the class, you can ask Eclipse to do it by right-clicking on the “MouseListener” (it will underline in red when you type it to let you know that these methods are required).
2. There are **two** drawText methods, they are very similar except that one requires you to specify a font and the other one doesn’t. This is an example of overloading, which is a very useful way of providing defaults (i.e. sensible values when the user doesn’t specify them).

**package** elycc;

**import** java.awt.\*;

**import** java.awt.event.\*;

**import** javax.swing.\*;

/\*\*

\* The main logic of the game.

\*/

**public** **class** Board1 **extends** JPanel **implements** ActionListener, MouseListener {

**int** screenWidth = 644;

**int** screenHeight = 480;

Font defaultFont = **new** Font("Courier", Font.***PLAIN***, 20);

GameState gameState;

Timer timer;

**public** **enum** GameState {

***NOTSTARTED***, ***PLAYING***, ***GAMEOVER***

}

/\*\*

\* Sets up everything that the game needs to run.

\*/

**public** Board1() {

setBackground(Color.***black***);

setFocusable(**true**);

setPreferredSize(**new** Dimension(screenWidth, screenHeight));

initGame();

}

/\*\*

\* Initialise the game by setting the board, setting up the ball, starting

\* the timer that moves the ball

\*/

**private** **void** initGame() {

// Set a timer to control the ball and the bat at roughly 60fps

timer = **new** Timer(16, **this**);

timer.start();

}

/\*\*

\* Called when Java decides that the window should be repainted.

\*/

@Override

**public** **void** paintComponent(Graphics g) {

**super**.paintComponent(g);

drawGraphics(g);

Toolkit.*getDefaultToolkit*().sync();

}

**void** drawText(Graphics g, **int** x, **int** y, String text, Color colour) {

drawText(g, x, y, text, defaultFont, colour);

}

**void** drawText(Graphics g, **int** x, **int** y, String text, Font font, Color colour) {

// Calculate the middle of the screen horizontally

**if** (x == -1) {

x = (getWidth() - getFontMetrics(font).stringWidth(text)) / 2;

}

// Calculate the middle of the screen vertically

**if** (y == -1) {

y = (getHeight() - getFontMetrics(font).getHeight()) / 2;

}

g.setColor(colour);

g.setFont(font);

g.drawString(text, x, y);

}

**void** drawGraphics(Graphics g) {

}

@Override

**public** **void** actionPerformed(**final** ActionEvent e) {

repaint();

}

@Override

**public** **void** mouseEntered(MouseEvent e) {

}

@Override

**public** **void** mouseExited(MouseEvent e) {

}

@Override

**public** **void** mouseClicked(MouseEvent e) {

}

@Override

**public** **void** mousePressed(MouseEvent e) {

}

@Override

**public** **void** mouseReleased(MouseEvent e) {

}

}

Run the code and the output should be very boring: just a windows on the screen, painted black, that you can close again.

Let’s add a bat to our breakout game that is controlled by the mouse. We’ll add two new methods:

1. To set the position of that bat based on where the mouse point is located.
2. To draw a bat on the screen.

We’ll need the game to remember where the bat is at all times, so create 2 new instance variables:

**int** batY = 400;

**int** batX;

(Remember that instance variables go at the top of the code, **inside** the class but **outside** of any methods.)

We’ll also remember the size of the bat, so create 2 new instance variables called batWidth and set it 100, and batHeight and set it to 5.

Java has methods built in to JFrame that reads the mouse position, so setting batX is pretty simple using a new method:

**void** moveBat() {

Point p = getMousePosition();

**if** (p != **null**) {

batX = (**int**) p.getX();

}

}

Drawing the bat is similarly straightforward, add another new method to do this:

**void** drawBat(Graphics g) {

g.setColor(Color.***WHITE***);

g.fillRect(batX, batY, batWidth, batHeight);

drawText(g, 100, 100, batX + ", " + batY, Color.***WHITE***);

}

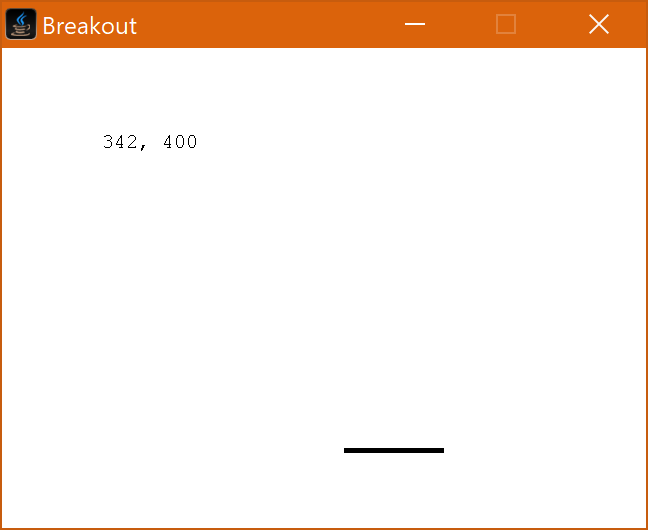
Note that we also add some helpful debugging information by writing the position of the bat in to the window, so we can see what’s going on.

Before this works, we need to make calls to these new methods at the right times.

In the first code listing of Board.java, we set up the Timer (called timer) which calls the actionPerformed method automatically around 60 times a second (1 second = 1000 milliseconds, so 1000 divided by 60 is approximately 16. So, by setting the timer to call actionPerformed every 16 milliseconds, we get roughly 60 frames per second.

So, to update the position of the bat 60 times a second, add a call to moveBat() at the top of the actionPerformedMethod. After moveBat is done, we tell Java to redraw all the graphics by calling repaint(). This is an internal method to JFrame which decides what to draw and eventually calls paintComponent in our Board.java class. This method then calls drawGraphics, which is the method that we’ll update.

Add a call to drawBat in the drawGraphics and run your code. You should see something like this:



Note how the bat follows the mouse, but **only** when the mouse cursor is inside the window.

## Adding a ball

Now we have a bat, we really need a ball to bounce around as well. As the ball will move of its own accord we’ll need four variables: 2 to store the ball’s X and Y co-ordinate; 1 to store how fast it’s moving horizontally and a final one to store how fast it’s moving vertically.

**int** ballX = 100;

**int** ballY = 100;

**int** ballDX = 2;

**int** ballDY = 4;

**int** ballSize = 5;

Similarly, to the bat, we’ll need two new methods:

The first method, moveBall will move the ball by adding ballDX to ballX and ballDY to ballY. Create this method first. The second method, drawBall will draw the ball on the screen at ballX, ballY of size ballSize (it’s a square ball). Use moveBat and drawBat to help you write these methods. (You may find it easiest to copy and paste them.)

Finally, add a call to moveBall to actionPerformed and a call to drawBall to drawGraphics.

## Collection Detection

Collision detection is the most important part of any game. As you saw when you ran your program, the ball just passed through the bat and continued off the screen.

Create a new method called handleCollisions() and add a call to it after the moveBall call in actionPerformed.

When you’ve done that, the method should look like this:

@Override

**public** **void** actionPerformed(**final** ActionEvent e) {

moveBat();

moveBall();

handleCollisions();

repaint();

}

The contents of handleCollisions is a bit tricky. Consider *when* the ball bounces of the bat:

1. IF the bottom of the ball is lower than the top of the bat, AND
2. The top of the ball is higher than the bottom of the bat, AND
3. The right-hand side of the ball is further right than the left-hand side of the bat, AND
4. The left-hand side of the ball is further left than the right-hand side of the bat.

The words in red indicate something that we know from the variables, such as:

1. The left-hand side of the bat is batX.
2. The right-hand side of the bat is batX + batWidth.
3. The top of the bat is batY.
4. The bottom of the bat is batY + batHeight.

You can work out how to calculate the top, left, right and bottom of the ball for yourself. Finally put it all together by adding an if statement to handleCollisions.

In Java, writing if statements requires brackets around the whole thing, i.e. if (…) and you can use && to represent an AND operator. For example:

if (something && somethingElse && somethingOther) {

}

Inside the curly brackets we’ll bounce the ball. The vertical direction of the ball is held in ballDY. To “bounce” the ball, reverse whatever its current value is. So, for example, if the ball is heading down at 2 pixels a frame then to make it go up change the vector to -2. So now the ball is going down at -2 pixels per frame, or more simply, going up at 2 pixels per frame.

Test your program and make sure that when the ball bounces ONLY when it hits bat.

We need to keep the ball enclosed at the top, left and right of the screen as well. Add another line in handleCollsions() to handle this:

// Check the ball has hit the top

**if** (ballY < 0) {

ballDY = -ballDY;

}

Finally, see if the ball has hit the left- or the right-hand side and reverse ballDX if so. You can use a logical OR in an if statement by using the || operator:

if (this || that) {

}

Run your code and now you have the basic workings of a game (you could even stop here and just count how many times you can bounce the ball before missing it).

## Adding the Wall of Bricks

The wall of bricks should span the entire window. The wall is going to be 14 bricks across and 5 bricks high. Each brick will be 16 pixels high. Finally, the wall will start 50 pixels from the top.

**int** bricksPerRow = 14;

**int** brickWidth = screenWidth / bricksPerRow;

**int** brickHeight = 16;

**int** brickYOffset = 50;

The bricks will be defined as a string array. Each element of the array is a single horizontal strip of bricks and each character in each string represents the colour of the brick that’s there, or a space if there’s a gap in the bricks.

String[] level = **new** String[] {

"RRRRRRRRRRRRRR",

"OOOOOOOOOOOOOO",

"YYYYYYYYYYYYYY",

"GGGGGGGGGGGGGG",

"BBBBBBBBBBBBBB" };

We’ll need a method to draw the entire wall that will need a method that draws a single brick.

**void** drawBrick(Graphics g, **int** x, **int** y, Color colour) {

g.setColor(colour);

g.fillRect(x \* brickWidth, y \* brickHeight + brickYOffset, brickWidth, brickHeight);

}

**void** drawWall(Graphics g) {

**for** (**int** y = 0; y < level.length; y++) {

**for** (**int** x = 0; x < bricksPerRow; x++) {

**char** brick = level[y].charAt(x);

Color colour;

**switch** (brick) {

**case** 'R':

colour = Color.***RED***;

**break**;

**case** 'O':

colour = Color.***ORANGE***;

**break**;

**case** 'Y':

colour = Color.***YELLOW***;

**break**;

**case** 'G':

colour = Color.***GREEN***;

**break**;

**case** 'B':

colour = Color.***BLUE***;

**break**;

**default**:

colour = **null**;

}

**if** (colour != **null**) {

drawBrick(g, x, y, colour);

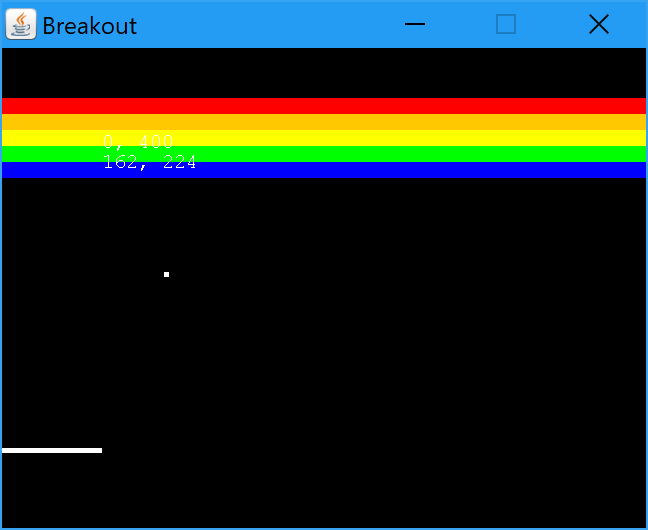
}

}

}

}

Finally, to render the wall add a call to drawWall(g) inside drawGraphics().



## Collision Detection

As the ball is square, we’ll check collisions with the ball at the four “corners” of the ball. Add the lines below to the bottom of the handleCollisions() method.

// Check if the ball has hit a brick

// Find which brick, or bricks, have been hit by the ball

handleBrickCollision(ballX, ballY);

handleBrickCollision(ballX + ballSize, ballY);

handleBrickCollision(ballX, ballY + ballSize);

handleBrickCollision(ballX, ballY + ballSize);

How the ball behaves when it hits a brick depends on which side of the ball hits the brick, but what does it mean to hit a brick?

It t urns out that this is extremely difficult to do in practice, but an approximation of the logic is shown below:

**void** handleBrickCollision(**int** x, **int** y) {

**int** brickX = x / brickWidth;

**int** brickY = (y - brickYOffset) / brickHeight;

**if** (brickY >= 0 && brickX >= 0 && brickY < level.length && brickX < bricksPerRow) {

**if** (level[brickY].charAt(brickX) != ' ') {

StringBuilder sb = **new** StringBuilder(level[brickY]);

sb.setCharAt(brickX, ' ');

level[brickY] = sb.toString();

ballDY = -ballDY;

}

}

}

Before testing your code, two more changes: initialise ballY to 300 (so it doesn’t start in the middle of the wall) initialise ballDY to -3 so that the ball starts by going up instead of down.

# Adding the Game Lifecycle

Make sure that GameState is declared as follows:

**public** **enum** GameState {

***NOTSTARTED***, ***PLAYING***, ***GAMEOVER***

}

Before the game starts, we’ll print a message saying “Click to start!” and wait for the user to click a mouse button before starting the game.

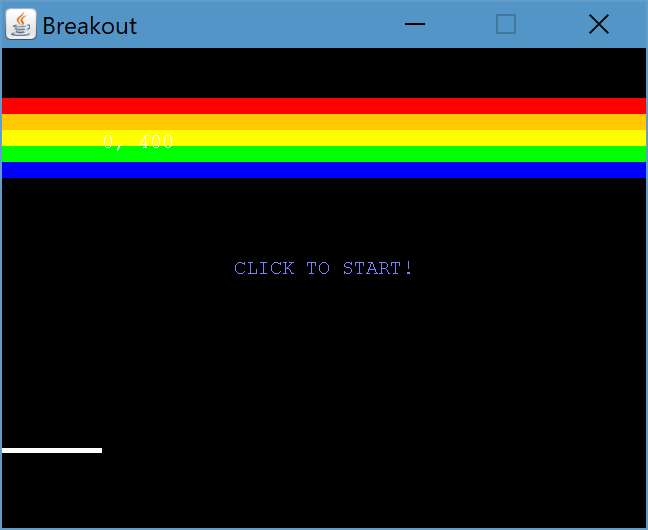
Make sure that gameState is initialised in initGame()

gameState = GameState.***NOTSTARTED***;

When the game isn’t running, the ball won’t be moving or being drawn, so modify drawGraphics to add a switch statement after drawWall and drawBat. If gameState is PLAYING then draw the ball (as it currently does), otherwise if gameState is NOTSTARTED then use drawText to write “CLICK TO START!” on the screen.

Of course, we need to stop the ball moving as well, so modify actionPerformed to only move the ball and handle collisions if gameState == GameState.PLAYING.

Run the program and you should see a message like this:



Finally, add some logic to the mouseClicked click handler method to change gameState.

**if** (gameState==GameState.***NOTSTARTED***) {

gameState = GameState.***PLAYING***;

}

Now that’s done, let’s end the game when the ball goes off the bottom of the screen. When the ball collides with the bottom of the screen, the game is over. So, add some new conditions to handleCollisions. You need to create a simple if statement that changes gameState to GameState.GAMEOVER if the ballY is greater than screenHeight.

We’ve already changed the code so that the ball is only moved when gameState is PLAYING, so all we need to do is add to the switch statement in drawGraphics to display “GAME OVER!”:

**case** ***GAMEOVER***:

drawText(g, -1, -1, "GAME OVER!", Color.***RED***);

**break**;

# Adding Some Skill

The game isn’t that skilful, alas. The ball always bounces the same way off of the bat. In the arcade Breakout game depending on where the ball hits the bat, the angle of the ball changes. If the ball hits the left third of the bat then the ball should go faster to the left, and the opposite on the right.

// If the ball hit the left third of the bat reduce ballDX to a minimum of -4

**if** (ballX < batX + batWidth/3) {

ballDX = ballDX - 1;

**if** (ballDX < -4) {

ballDX = -4;

}

}

// If the ball hit the left third of the bat reduce ballDX to a maximum of -4

**if** (ballX > batX + batWidth\*2/3) {

ballDX = ballDX + 1;

**if** (ballDX > 4) {

ballDX = 4;

}

}

Finally, to avoid the ball from going straight up, change ballDX to type float and initialise it to 2.5f. (f means float.)