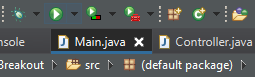
Breakout Evaluation Document

Introduction

In this document I will explain the additions I have made to the breakout game coursework. I will also be detailing the challenges I faced and solutions I implemented, along with an explanation and presentation of the tests I have completed to ensure my program is running as intended. As this is an overview of an entire program, I will be mainly explaining the code at a high level so that it is easier to understand and summarise.

How to use the program

The program itself is very simple in operation. Using Eclipse, navigate to the main class and hit the run button to run the program.

Once run the game begins instantly, you can choose to either use the right and left arrows or the mouse to control the bat and hit the ball against the bricks above, whilst attempting to prevent the ball from hitting the bottom of the screen. If the ball hits the bottom of the screen 100 points are deducted from the score. Each time a brick is hit, you gain 100 points. The coloured bricks have 3 lives thanks to the PersistentGameObj class which I will explain further later in the document.

The coloured bricks have three levels. They start off green, then go yellow when hit once, then yellow when hit twice and then red on their last life before being destroyed and playing a breaking sound, each hit previous plays a cracking sound to show that they have been damaged.

There are two modes to the game, Fast and Normal. Normal mode is selected by default when launching the game. Fast mode may be selected by pressing the F key, to return to the default setting either restart the game or press the N key. Fast mode makes the game update at a higher rate and so increases the ball’s speed, making the game more difficult.

There is currently no end screen for the game, however you can press S to stop the game.

This program is an example of one level for the game, however If I were to continue the game myself, I would add more levels to beat to make the game more interesting to play.

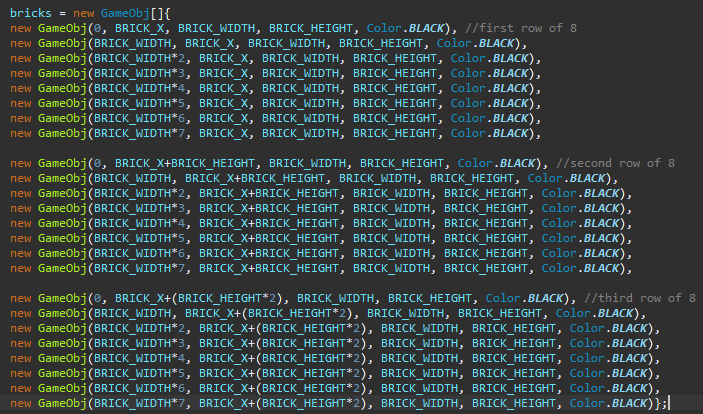
Features I added

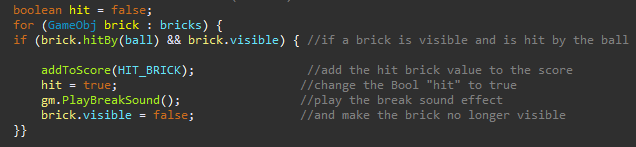
In the following section I will detail the features I added to the game, and how they work.

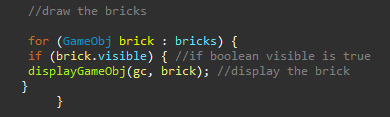
The Black Bricks (GameObj)

This is the code in the Model class that creates the standard black bricks. As you can see, instead of using numbers for the brick’s x and y location I use the brick variables declared at the start of the class. Doing it this way means that I can easily change the size and location of the bricks later instead of changing each object individually.



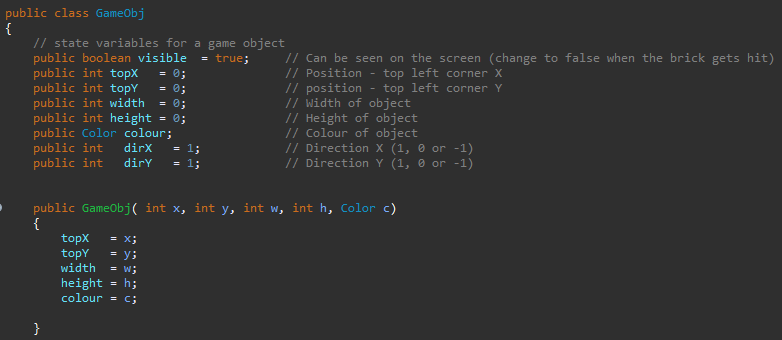


The following code also in the Model class shows how I made the bricks disappear when hit by the ball. The gm.PlayBreakSound(); method will be explained later in the document.

The code below is needed in the View class to make sure that only visible bricks are displayed. When I first wrote the code in model I was confused as to why my bricks were not disappearing. It turned out that I needed the if statement to check if the brick’s visible Boolean was true before displaying.

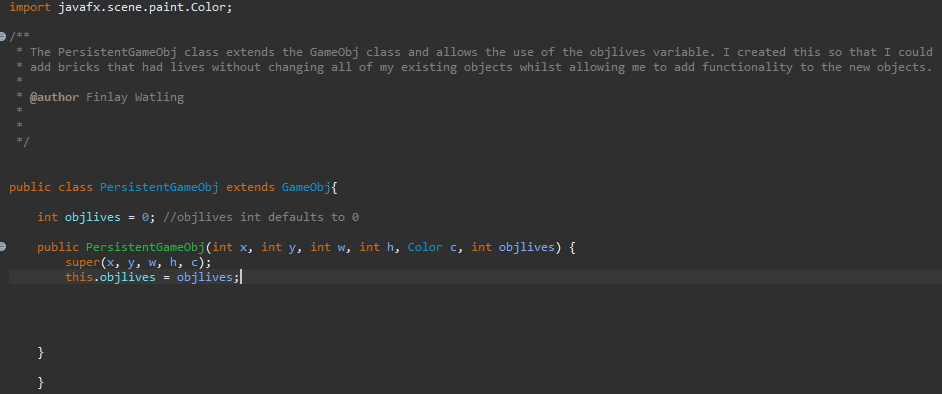
The Coloured Bricks (PersistentGameObj)

The Screenshot below shows the GameObj class. To make the game more interesting I wanted to add bricks that needed to be hit more than once.

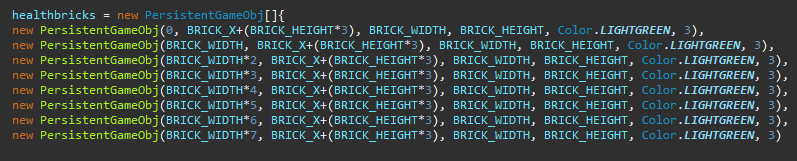


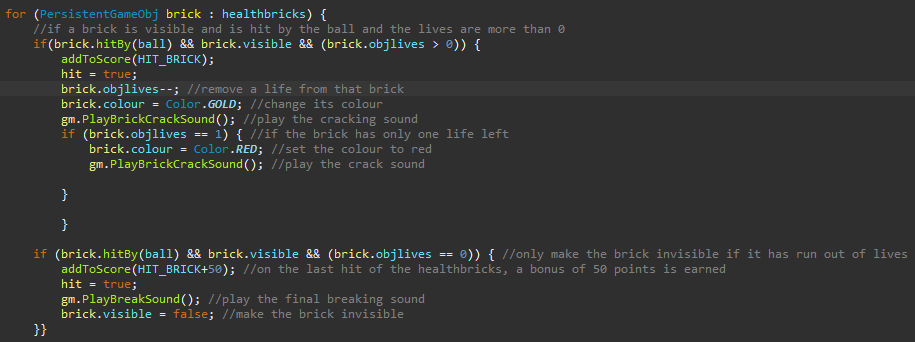
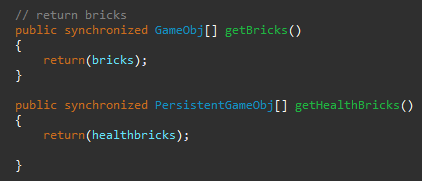
To do this I made a new class that extends the GameObj class, I did this to make it easier to implement the changes, if I just added the objlives variable to GameObj I would need to add the new parameter to all the black bricks. This way not only makes it easier to understand and demonstrates modular code but also shows that I know how to extend classes.

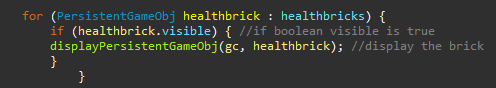
In the following screenshot you can see the new PersistentGameObj class that extends GameObj. I declare the objlives integer and set it to default to 0. I then use the super keyword to refer to the parent class and access the parent’s variables and methods.

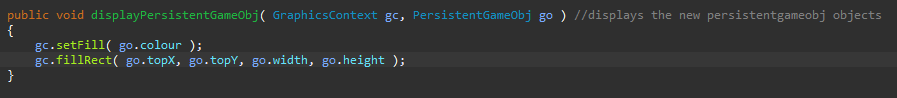


The following code shows the creation of the healthbricks array that stores PersistentGameObj objects.

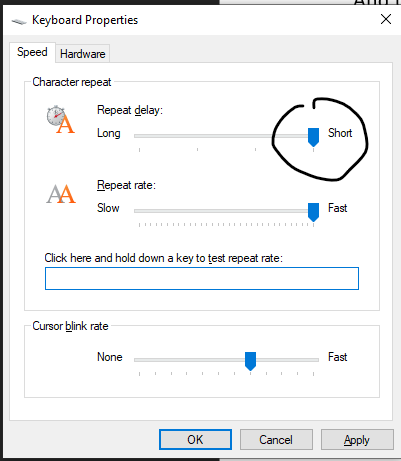


The following code in the Model class shows how I implemented the objlives variable with the new PersistentGameObj bricks.

And the changes in the View class to display the new bricks.



Implementing mouse movement

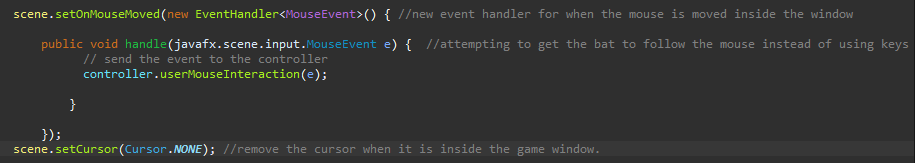
While playing the first iterations of the game I felt that the controls were a bit clunky due to the lag between pressing the right and left arrows to move the bat and the screen updating. After trying to fix this delay for a few hours, I realised through a post on superuser.com that it is a feature built into the windows operating system. Although this delay may not bother all users, I found it particularly bothersome and so sought to find a better solution.

https://superuser.com/questions/1164303/windows-how-do-i-disable-the-keyboard-delay

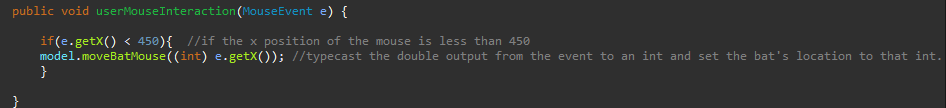
Mouse movement is not subject to such delays, and so if I could implement mouse movement to my program then that would fix my issues.

I searched the internet and found that there is a built-in method to handle mouse events. https://docs.oracle.com/javase/8/javafx/api/javafx/scene/input/MouseEvent.html

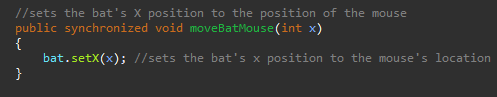
The following screenshots show the implementation of Mouse movement in my code.

The following code is in the View class. The scene.setCursor method makes it so that the cursor is invisible when inside the window. I added this because I found the cursor to be distracting when playing using mouse movement.

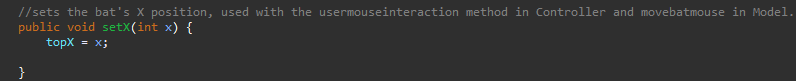
The following code is in the Controller class. I use the if statement to prevent the bat from going off the screen. This screenshot also demonstrates typecasting. As the e.getX function outputs a double I needed to convert it to an int when passing it to the MoveBatMouse method.



The following code is in the Model class.



The following code is in the GameObj class.

I was extremely happy with how the implementation of mouse movement turned out as prior to this coursework I had not used event handlers at all.

Preventing the bat from going through the borders of the game window

When testing the program, I realised that the bat could go past the left and right borders of the window and sometimes whilst playing I would lose the bat and not be able to find it again. To combat this, I wrote the following code in the Model class to prevent the bat from passing through the borders.

Creating the GameMedia class to add sound to my program

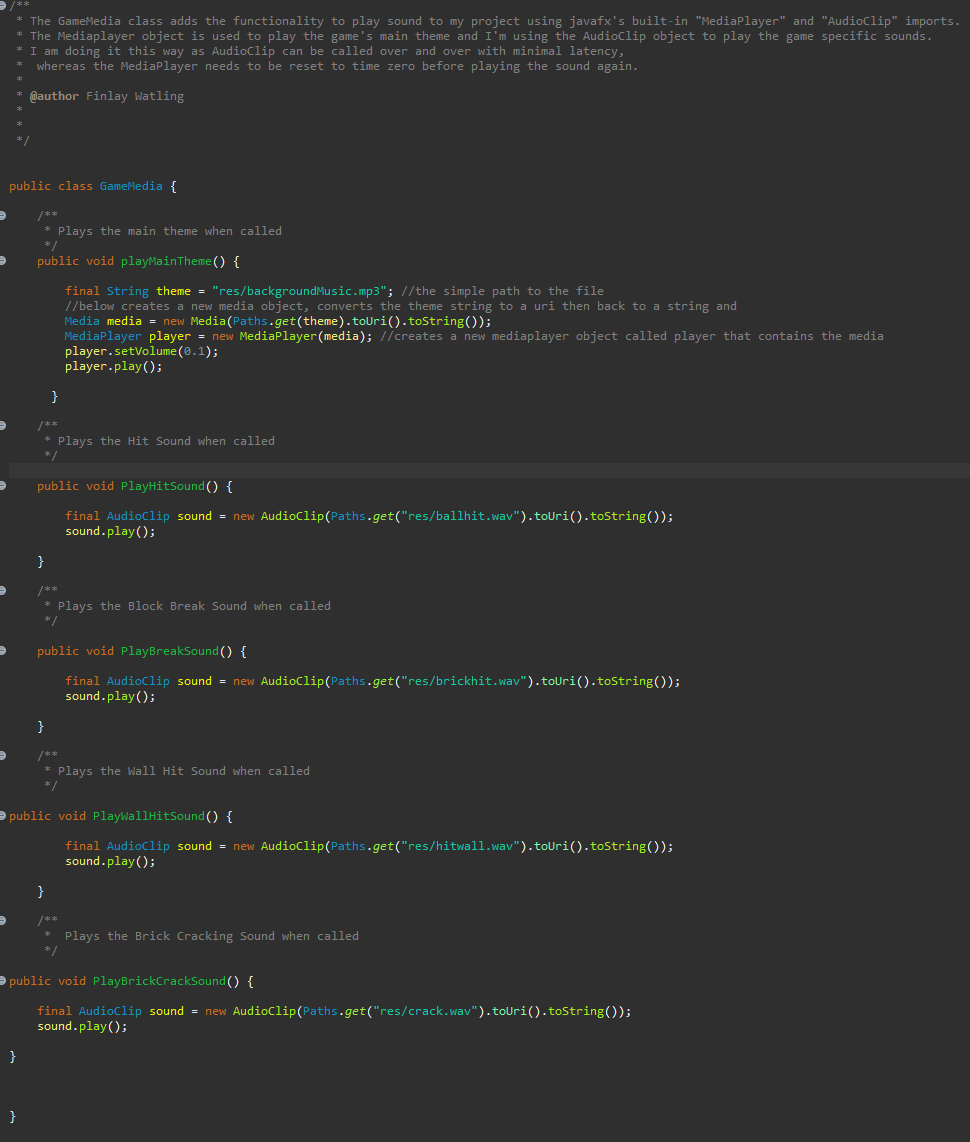
I wanted to add sound to my game to make it more interesting and so I created the GameMedia class. There is a description of the class in the Javadoc comments but I will explain its workings at a high level in the following section.

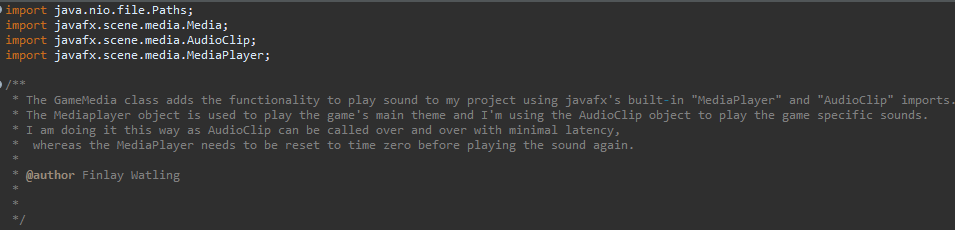
When researching how to implement sound using javafx libraries I saw that the MediaPlayer import was used a lot. After successfully using it to play the main theme tune for my game I attempted to use it for some of the other sounds in the game such as the ball hitting the bat etc. However, I could not figure out how to get the sound to play every time the ball was hit. This was due to a limitation in MediaPlayer. Each time the media player plays a sound it needs to be reset to time 0 before playing it again.

https://docs.oracle.com/javafx/2/api/javafx/scene/media/MediaPlayer.html

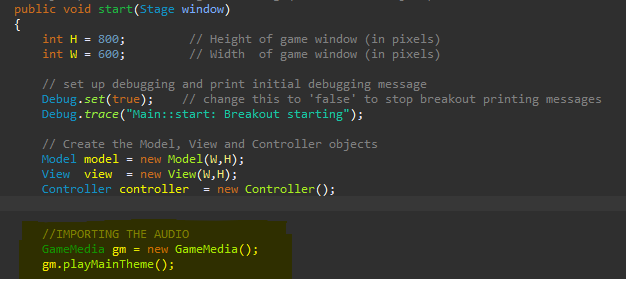
The aforementioned method is not practical, nor is it an efficient use of memory, so I looked for another import that could do what I needed. I came across the AudioClip class which was perfect. It not only plays sounds over and over when called with minimal latency, but can overlap clips too, meaning if the ball bounces two times before the first audio clip has finished, it will just get called again and they can be played simultaneously.

<https://docs.oracle.com/javase/8/javafx/api/javafx/scene/media/AudioClip.html>

The following screenshots show my implementation of the GameMedia class.



The following screenshot is from the Main class.



The following screenshots are from the Model class.

