2020 CI401
Introduction to programming

Week 2.01
Breakout lab notes

10th February 2021

Roger Evans

Module leader

Introduction to Breakout

Starter project - Breakout

- Breakout is a classic computer arcade game
- Here's a video to remind you:
  - https://www.youtube.com/watch?v=AMUv8KvVt08
- In this week's lab exercise, you are given a version of Breakout that does almost nothing.
- The lab exercise is to turn it into a functional game (with help from tutors if you need it)
- The solution to this lab is the starting point for your independent project work, if you choose to do Breakout.

Demonstrating Breakout

- The solution system –a simple, working, Breakout game, with a row of bricks which disappear correctly when the ball bounces off them

- The lab exercise system – without any bricks! But it does have a ball, a working bat and a score function

What's actually happening in the code of this game?

- On the screen, there's a score, some bricks, the ball and the bat
- The screen appears to change – the ball moves, the score updates, bricks disappear, the bat moves when you tell it to
- But as we saw with our JavaFX animation, nothing is really moving – what's happening is the image on the screen is being changed very quickly (50 times a second)
- What our program is doing is working out what should appear on the screen at each point, and then displaying it

How does the program decide what to display?

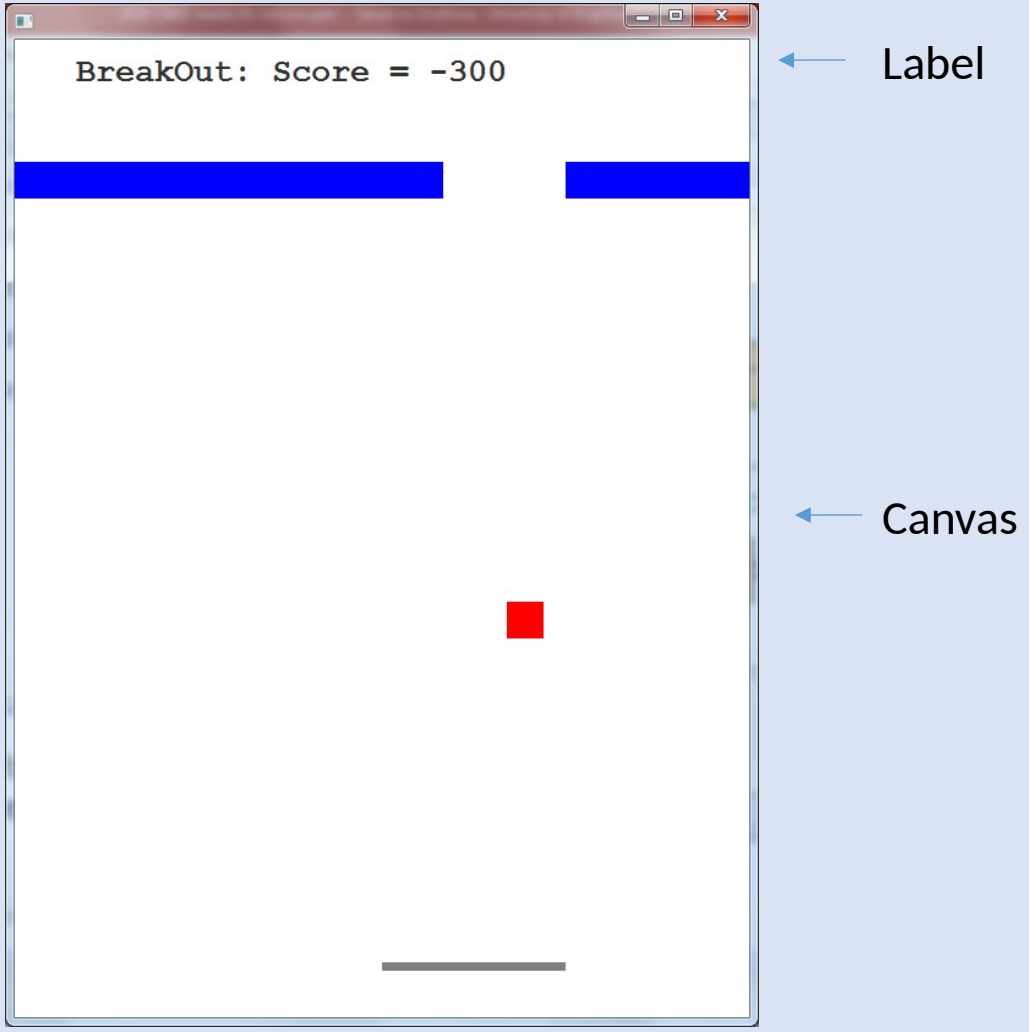
- The ball is ‘moving’. So for each image (50 times a second), the program needs to work out where the ball has moved to (so it needs to know where it is, what direction it is going in, and how fast it is supposed to be ‘moving’)
- The bat also ‘moves’, but only when a key is pressed. So the program is watching for keys to be pressed, and changing the position of the bat when that happens
- Whenever the ball moves the program also checks whether it has hit anything:
  - The side or top – the ball needs to change direction to bounce off
  - The bottom – the ball bounces off and there is a score penalty
  - A brick – the ball bounces off, the brick disappears, and the score increases
  - The bat – the ball bounces off

How is the program organised?

- The program follows a very common pattern for writing GUI programs called model-view-controller (or MVC)
- We will see MVC a few more times this semester. Today we just need to get an idea of the basics
- In MVC the main structure of the program is divided into three classes
  - The Model class – where all the calculations about what is happening in the game are stored
  - The View class – which manages what you actually see on the screen
  - The Controller class – which decides what happens when the user presses keys etc.

The View class

- We start with the View, because it is the most ‘visible’
- This is a JavaFX window like the ones we have seen before
- In fact it only has two visual components:
  - A Label, showing the score
  - A Canvas, on which it ‘paints’ the current game state (repainting the whole thing 50 times a second)
- In addition it has an event handler, which detects keyboard presses



The Model class

- This is where the 'game logic' is
- It has a loop, running 50 times a second, when it moves the ball and checks whether it has hit anything
- It also responds to commands to move the bat (and a few other things)
- It uses a set of 'Top Trumps' cards (the GameObj class)for all the elements of the game – bat, ball, bricks
- All the mode is doing is changing numbers on the cards

boolean visible	true
int topX	0
int topY	0
int width	0
int height	0
Color colour	
int dirX	1
int dir Y	1

← GameObj Class

boolean visible	true
int topX	200
int topY	350
int width	30
int height	30
lor colour	red
dirX	1
dir Y	1

↑ Ball

boolean visible	true
int topX	200
int topY	650
int width	150
int height	10
Color colour	gray
int dirX	1
int dir Y	0

← Bat

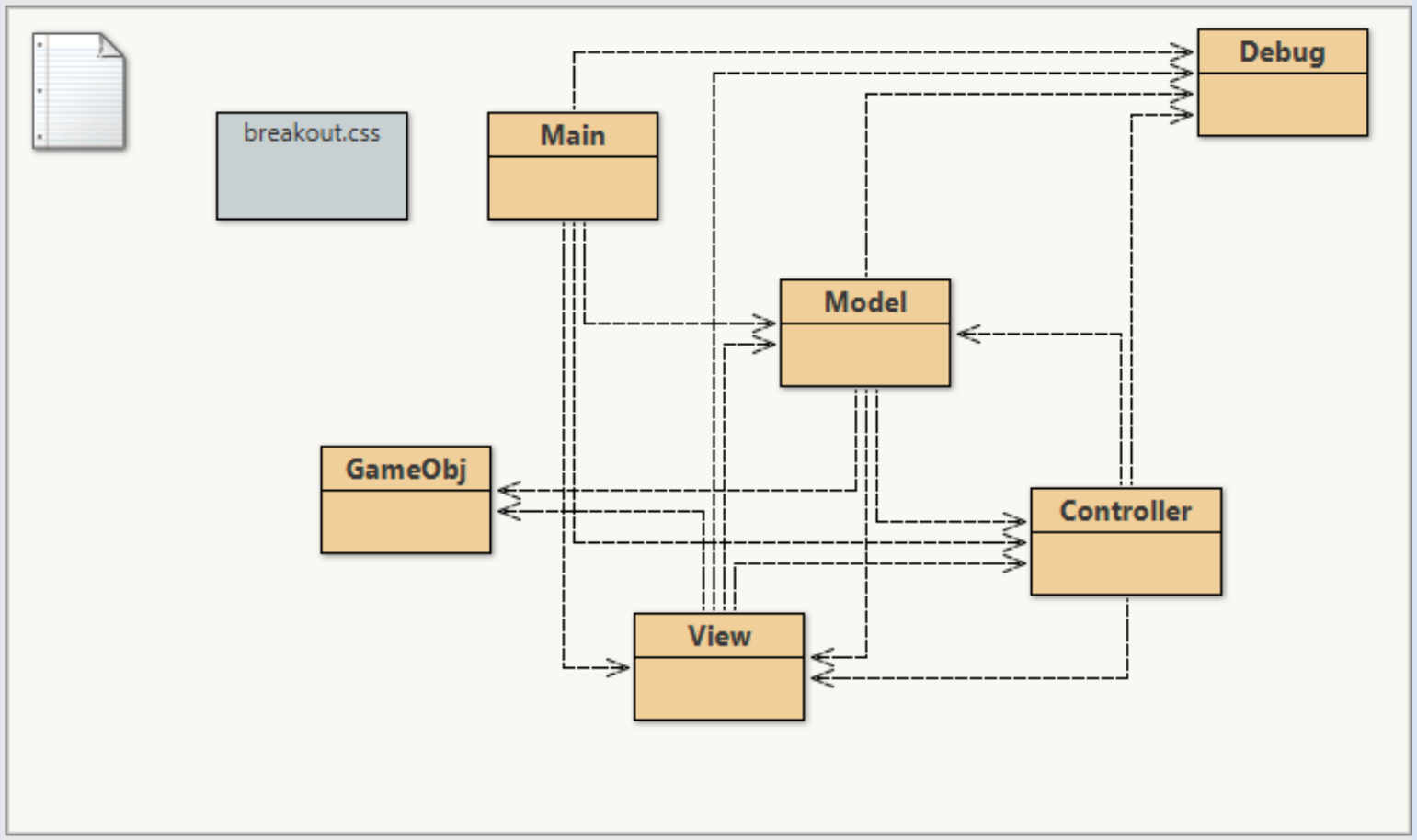
The Controller class

- The simplest of the three classes
- This provides the event handler function that the View uses
- Whenever the user presses a key, the controller decides what the game should do and tells the the Model to do it
- For example
  - When the user presses ‘<‘, it tells the Model to move the bat left
  - When the user presses ‘>’, it tells the model to move the bat right
- You can change the way keys map into commands without changing the View or the Model
  - Make a left-handed version (using , say ‘Z’ and ‘X’ instead of ‘<‘, ‘>’)
  - Add functions for a two player version (on one keyboard)

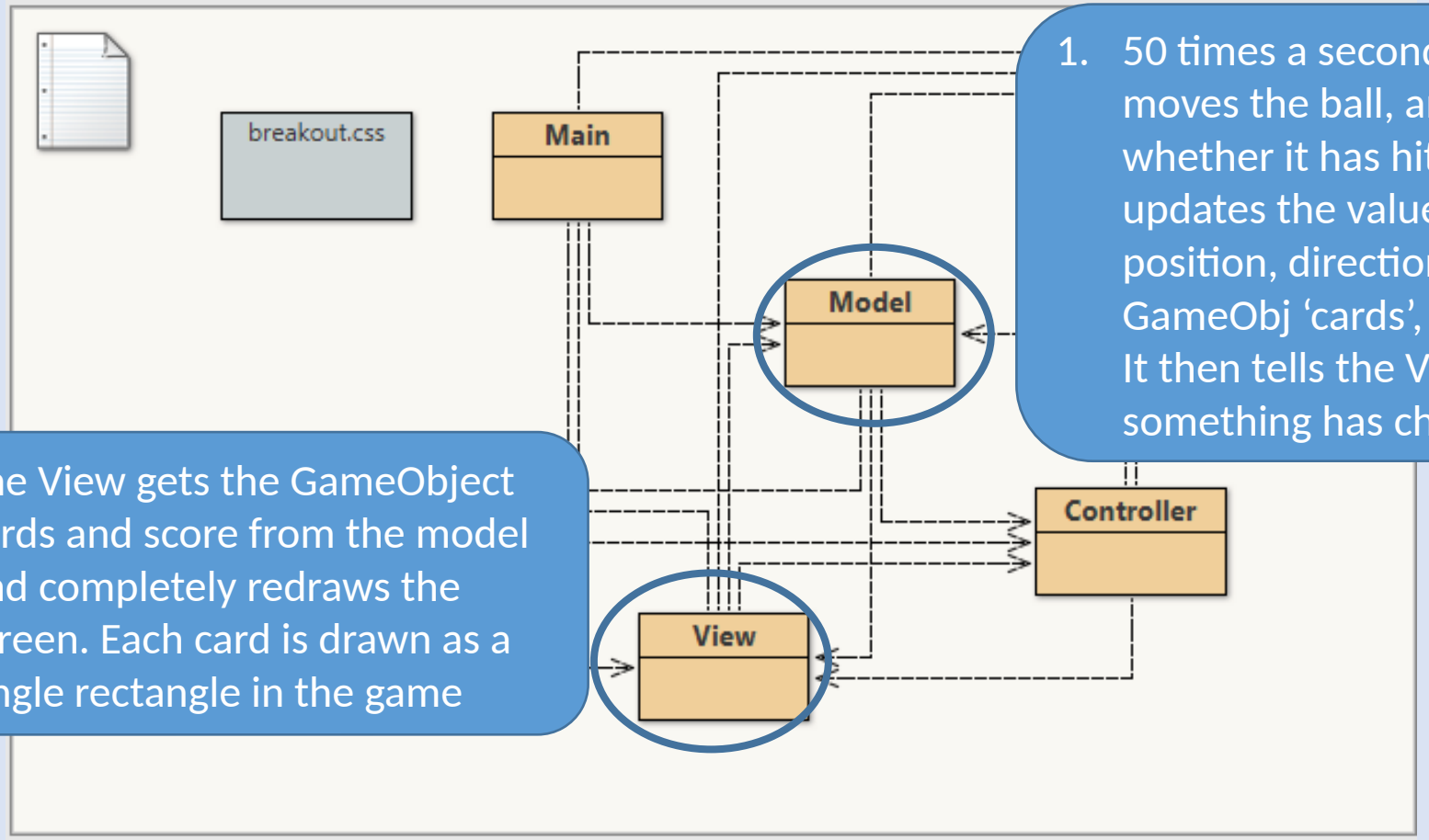
Other classes

- The Main class – starts the program, creates the Model, View and Controller objects and ‘joins them together’
- The GameObj class – the main data class for the things in the game (this is the ‘Top Trumps deck’ the model uses)
- The Debug class – prints out messages about what is going on to help you debug the program

Breakout classes



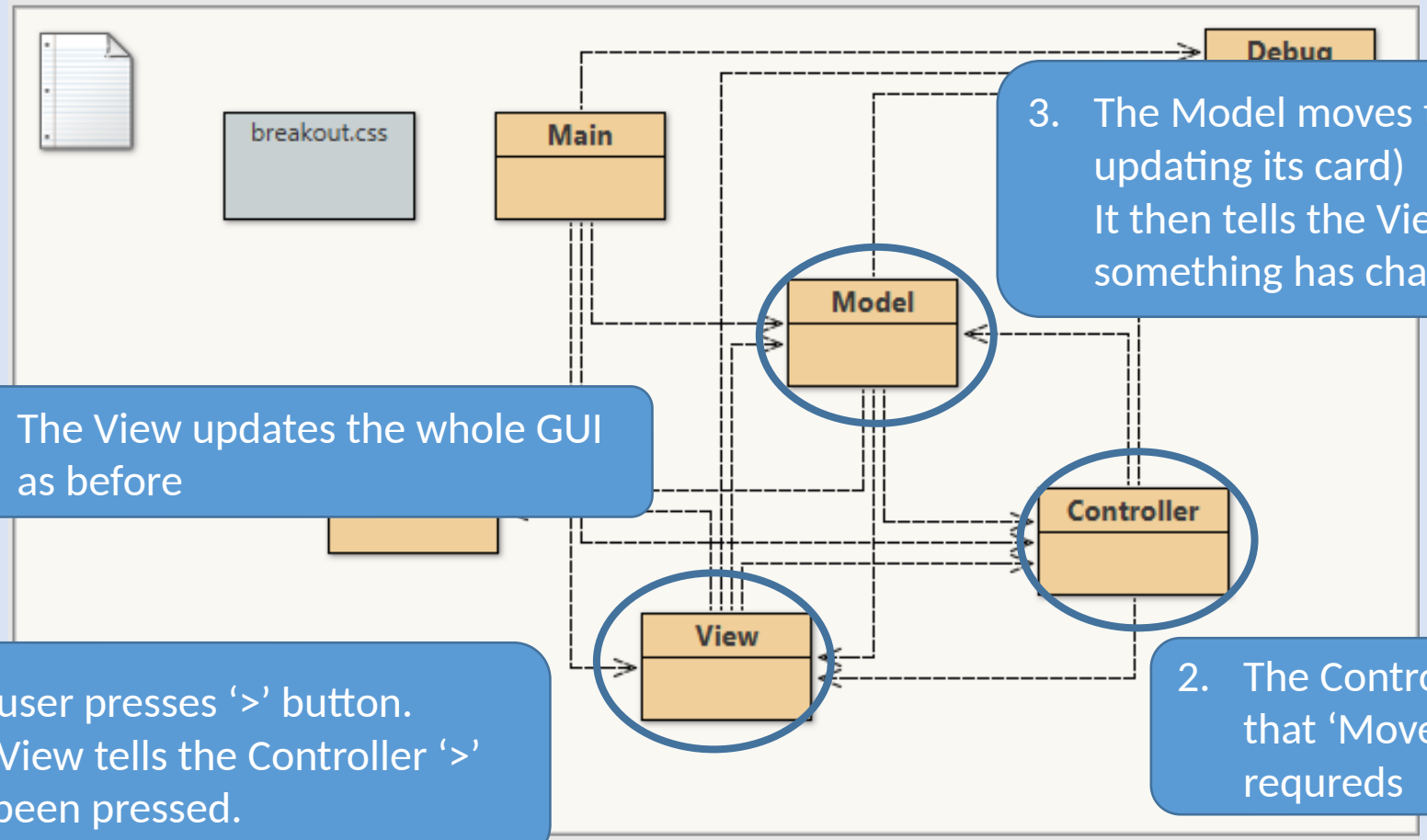
How the components talk to each other – main loop



1. 50 times a second, the model moves the ball, and checks whether it has hit anything. It updates the values (visible, position, direction etc) on the GameObj 'cards', and the score. It then tells the View that something has changed

2. The View gets the GameObject cards and score from the model and completely redraws the screen. Each card is drawn as a single rectangle in the game

How the components talk to each other – key presses



1. The user presses '>' button.
   The View tells the Controller '>' has been pressed.

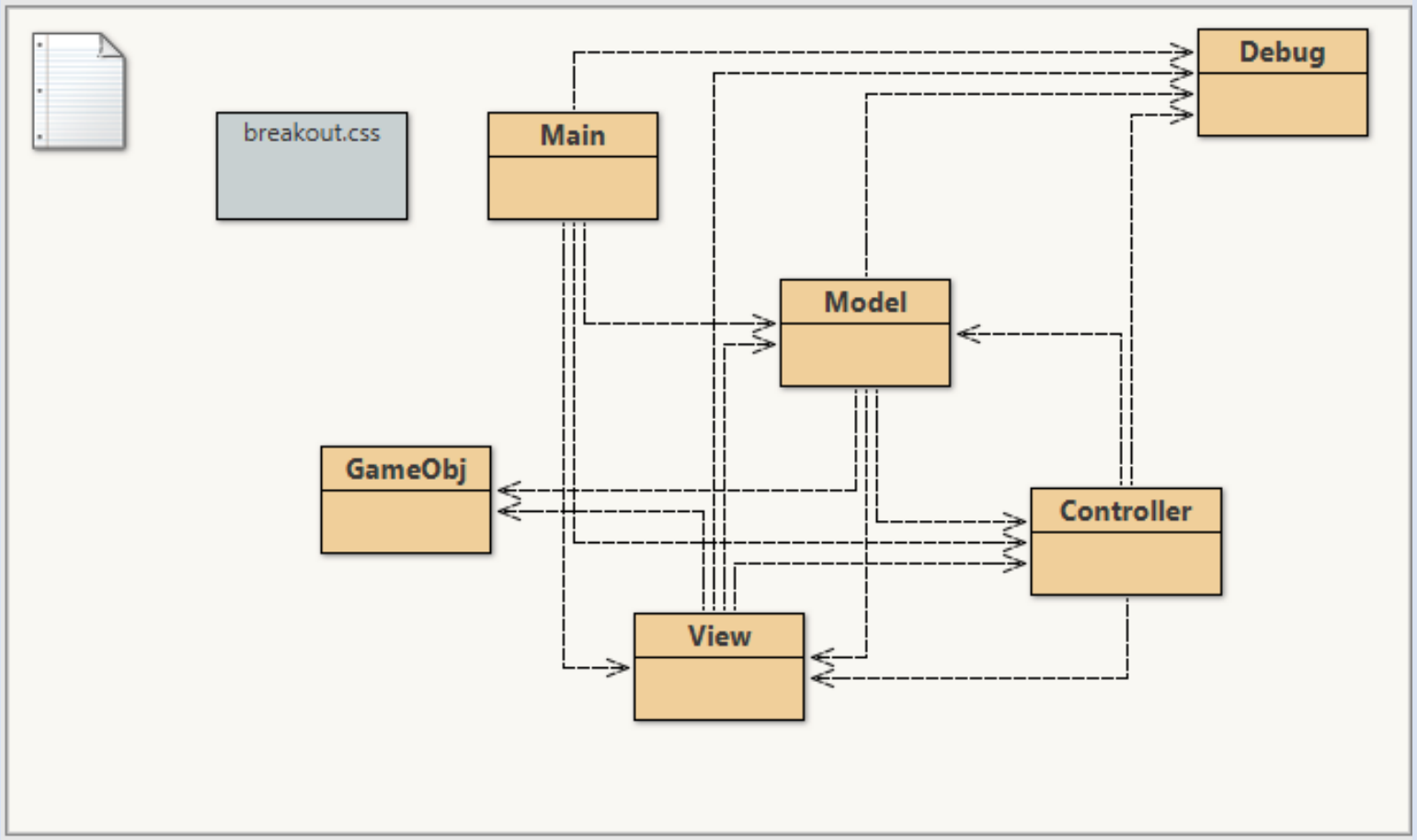
2. The Controller tells the Model that 'Move bat left' has been requreds

3. The Model moves the bat (by updating its card)
   It then tells the View that something has changed

4. The View updates the whole GUI as before

The Breakout code

Breakout classes



The Main class

Main class

- Create Model, View and Controller objects
- Join them together
- Start the view (user interface)
- Set the model running (in a loop)

```
21      public void start(Stage window)
22      {
23          int H = 800;          // Height of game window (in pixels)
24          int W = 600;          // Width  of game window (in pixels)
25
26          // set up debugging and print initial debugging message
27          Debug.set(true);    // change this to 'false' to stop breakout printing messages
28          Debug.trace("Main::start: Breakout starting");
29
30          // Create the Model, View and Controller objects
31          Model model = new Model(W,H);
32          View  view  = new View(W,H);
33          Controller controller  = new Controller();
34
35          // Link them together so they can talk to each other
36          // Each one has instance variables for the other two
37          model.view = view;
38          model.controller = controller;
39
40          controller.model = model;
41          controller.view = view;
42
43          view.model = model;
44          view.controller = controller;
45
46          // start up the game interface (the View object, passing it the window
47          // object that JavaFX passed to this method, and then tell the model to
48          // start the game
49          view.start(window);
50          model.startGame();
51
52          // application is now running - print a debug message to say so
53          Debug.trace("Main::start: Breakout running");
54      }
55  }
```

The View class

View class start method

- Sets up the main JavaFX interface
- The GUI uses a Pane layout manager
- The pane contains a Canvas object on which we will draw boxes (for bat, ball and bricks)
- There is also a Label object for the score
- Lastly we add an event handler which runs whenever a key is pressed

```
47  public void start(Stage window)
48  {
49      // breakout is basically one big drawing canvas, and all the objects are
50      // drawn on it as rectangles, except for the text at the top - this
51      // is a label which sits 'in front of' the canvas.
52
53      // Note that it is important to create control objects (Pane, Label,Canvas etc)
54      // here not in the constructor (or as initialisations to instance variables),
55      // to make sure everything is initialised in the right order
56      pane = new Pane();        // a simple layout pane
57      pane.setId("Breakout");  // Id to use in CSS file to style the pane if needed
58
59      // canvas object - we set the width and height here (from the constructor),
60      // and the pane and window set themselves up to be big enough
61      canvas = new Canvas(width,height);
62      pane.getChildren().add(canvas);     // add the canvas to the pane
63
64      // infoText box for the score - a label which we position in front of
65      // the canvas (by adding it to the pane after the canvas)
66      infoText = new Label("BreakOut: Score = " + score);
67      infoText.setTranslateX(50);  // these commands setthe position of the text box
68      infoText.setTranslateY(10);  // (measuring from the top left corner)
69      pane.getChildren().add(infoText);  // add label to the pane
70
71      // Make a new JavaFX Scene, containing the complete GUI
72      Scene scene = new Scene(pane);
73      scene.getStylesheets().add("breakout.css"); // tell the app to use our css file
74
75      // Add an event handler for key presses. By using 'this' (which means 'this
76      // view object itself') we tell JavaFX to call the 'handle' method (below)
77      // whenever a key is pressed
78      scene.setOnKeyPressed(this);
79
80      // put the scene in the window and display it
81      window.setScene(scene);
82      window.show();
83  }
```

View class handler and drawPicture

- The handler method gets called whenever a key is pressed
- It just calls a method in the controller passing it the keypress event
- drawPicture gets called to update the screen when the model changes
- It clears the screen (paints it white), and draws the bat and the ball
- You need to make it draw the bricks as well (NB: they are just GameObjs like the bat and ball)

```
84
85      // Event handler for key presses - it just passes the event to the controller
86      public void handle(KeyEvent event)
87      {
88          // send the event to the controller
89          controller.userKeyInteraction( event );
90      }
91
92      // drawing the game image
93      public void drawPicture()
94      {
95          // the game loop is running 'in the background' so we have
96          // add the following line to make sure it doesn't change
97          // the model in the middle of us updating the image
98          synchronized ( model )
99          {
100             // get the 'paint brush' to pdraw on the canvas
101             GraphicsContext gc = canvas.getGraphicsContext2D();
102
103             // clear the whole canvas to white
104             gc.setFill( Color.WHITE );
105             gc.fillRect( 0, 0, width, height );
106
107             // draw the bat and ball
108             displayGameObj( gc, ball );   // Display the Ball
109             displayGameObj( gc, bat  );   // Display the Bat
110
111             // *[2]*****************************************************[2]*
112             // * Display the bricks that make up the game                 *
113             // * Fill in code to display bricks from the brick array      *
114             // * Remember only a visible brick is to be displayed         *
115             // ************************************************************
116
117
118
119
120             // update the score
121             infoText.setText("BreakOut: Score = " + score);
122         }
123     }
```

View class displayGameObject and update

- displayGameObj paints a game object on the screen.
- The object is just a rectangle, and it knows where it is, how big it is and what colour it is
- update is called by the Model whenever it changes something (50 times a second!)
- It fetches all the game information from the model, and then redraws the picture on the screen

```
125     // Display a game object - it is just a rectangle on the canvas
126     public void displayGameObj( GraphicsContext gc, GameObj go )
127     {
128         gc.setFill( go.colour );
129         gc.fillRect( go.topX, go.topY, go.width, go.height );
130     }
131
132     // This is how the Model talks to the View
133     // This method gets called BY THE MODEL, whenever the model changes
134     // It has to do whatever is required to update the GUI to show the new game position
135     public void update()
136     {
137         // Get from the model the ball, bat, bricks & score
138         ball    = model.getBall();              // Ball
139         bricks  = model.getBricks();            // Bricks
140         bat     = model.getBat();               // Bat
141         score   = model.getScore();             // Score
142         //Debug.trace("Update");
143         drawPicture();                      // Re draw game
144     }
145 }
```

The Controller class

Controller class

- The Controller has one method which is called by the view when a key is pressed
- It gets the code (the particular key that was pressed) from the event, and then uses a switch statement to tell the model what to do – move the bat, speed up, finish etc.

```
// This is how the View talks to the Controller
// AND how the Controller talks to the Model
// This method is called by the View to respond to key presses in the GUI
// The controller's job is to decide what to do. In this case it converts
// the keypresses into commands which are run in the model
public void userKeyInteraction(KeyEvent event )
{
  // print a debugging message to show a key has been pressed
  Debug.trace("Controller::userKeyInteraction: keyCode = " + event.getCode() );

  // KeyEvent objects have a method getCode which tells us which key has been pressed.
  // KeyEvent also provides variables LEFT, RIGHT, F, N, S (etc) which are the codes
  // for individual keys. So you can add keys here just by using ther name (which you
  // can find out by googling 'JavaFX KeyCode')
  switch ( event.getCode() )
  {
    case LEFT:                    // Left Arrow
      model.moveBat( -1);         // move bat left
      break;
    case RIGHT:                   // Right arrow
      model.moveBat( +1 );        // Move bat right
      break;
    case F :
      // Very fast ball movement
      model.setFast(true);
      break;
    case N :
      // Normal speed ball movement
      model.setFast(false);
      break;
    case S :
      // stop the game
      model.setGameState("finished");
      break;
  }
}
```

The Model class

Model class initaliseGame

- The Model class manages the game objects – the ball, the bat and the bricks
- Each object knows where it is, how big it is and what colour it is
- initialiseGame creates these objects in their starting positions
- The bat and ball are created here, and an array of GameObjs for the bricks
- You need to add some actual bricks to the array.
- Each brick is a game object and you need to set its position, size and colour

```
75  // Start the animation thread
76  public void startGame()
77  {
78      initialiseGame();                           // set the initial game state
79      Thread t = new Thread( this::runGame );     // create a thread running the runGame met
80      t.setDaemon(true);                          // Tell system this thread can die when it
81      t.start();                                  // Start the thread running
82  }
83
84  // Initialise the game - reset the score and create the game objects
85  public void initialiseGame()
86  {
87      score = 0;
88      ball   = new GameObj(width/2, height/2, BALL_SIZE, BALL_SIZE, Color.RED );
89      bat    = new GameObj(width/2, height - BRICK_HEIGHT*3/2, BRICK_WIDTH*3,
90          BRICK_HEIGHT/4, Color.GRAY);
91      bricks = new GameObj[0];
92      // *[1]*****************************************************[1]*
93      // * Fill in code to make the bricks array                     *
94      // ************************************************************
95
96
97
98  }
```

Model class animation loop

- The game ‘works’ by running a loop which changes the position of the ball 50 times a second and checks whether the ball has hit anything
- Each time the model changes, it updates the view, so the display changes
- The loop runs in a separate Thread, (like a second program), which updates the game, tells the view, and then sleeps for 20 milliseconds before doing it again

```
101  // The main animation loop
102  public void runGame()
103  {
104      try
105      {
106          Debug.trace("Model::runGame: Game starting");
107          // set game true - game will stop if it is set to "finished"
108          setGameState("running");
109          while (!getGameState().equals("finished"))
110          {
111              updateGame();                          // update the game state
112              modelChanged();                        // Model changed - refresh screen
113              Thread.sleep( getFast() ? 10 : 20 ); // wait a few milliseconds
114          }
115          Debug.trace("Model::runGame: Game finished");
116      } catch (Exception e)
117      {
118          Debug.error("Model::runAsSeparateThread error: " + e.getMessage() );
119      }
120  }
121
```

Model class updateGame

- updateGame changes the state of the game each time round the loop
- It moves the ball (the ball knows which way it is going), checks whether it has hit the sides of the screen (and changes its direction if so, so that it 'bounces' off)
- Then it checks whether it has hit the bat, and again changes direction if so.
- You need to add some code to check whether it has hit any of the bricks (loop through the array). If so, make the brick invisible, and make the ball bounce off it.

```
122    // updating the game - this happens about 50 times a second to give the impression of move
123    public synchronized void updateGame()
124    {
125        // move the ball one step (the ball knows which direction it is moving in)
126        ball.moveX(BALL_MOVE);
127        ball.moveY(BALL_MOVE);
128        // get the current ball possition (top left corner)
129        int x = ball.topX;
130        int y = ball.topY;
131        // Deal with possible edge of board hit
132        if (x >= width - B - BALL_SIZE)  ball.changeDirectionX();
133        if (x <= 0 + B)  ball.changeDirectionX();
134        if (y >= height - B - BALL_SIZE)  // Bottom
135        {
136            ball.changeDirectionY();
137            addToScore( HIT_BOTTOM );     // score penalty for hitting the bottom of the scree
138        }
139        if (y <= 0 + M)  ball.changeDirectionY();
140
141       // check whether ball has hit a (visible) brick
142        boolean hit = false;
143
144        // *[3]*********************************************************[3]*
145        // * Fill in code to check if a visible brick has been hit      *
146        // * The ball has no effect on an invisible brick               *
147        // * If a brick has been hit, change its 'visible' setting to   *
148        // * false so that it will 'disappear'                          *
149        // *************************************************************
150
151
152
153
154        if (hit) {
155            ball.changeDirectionY();
156        }
157
158        // check whether ball has hit the bat
159        if ( ball.hitBy(bat) ) {
160            ball.changeDirectionY();
161        }
162    }
```

Model class modelChanged

- modelChanged gets called whenever the model changes.
- It calls update in the view object, which will then request the current state of the model in order to update the GUI.
- The other methods here are used by the controller to change the model when keys are pressed.

```
164  // This is how the Model talks to the View
165  // Whenever the Model changes, this method calls the update method in
166  // the View. It needs to run in the JavaFX event thread, and Platform.runLater
167  // is a utility that makes sure this happens even if called from the
168  // runGame thread
169  public synchronized void modelChanged()
170  {
171      Platform.runLater(view::update);
172  }
173
174
175  // Methods for accessing and updating values
176  // these are all synchronized so that the can be called by the main thread
177  // or the animation thread safely
178
179  // Change game state - set to "running" or "finished"
180  public synchronized void setGameState(String value)
181  {
182      gameState = value;
183  }
184
185  // Return game running state
186  public synchronized String getGameState()
187  {
188      return gameState;
189  }
190
191  // Change game speed - false is normal speed, true is fast
192  public synchronized void setFast(Boolean value)
193  {
194      fast = value;
195  }
196
```

The GameObj class

GameObject class

GameObject is mainly a data class – storing information about a single object (which is always a rectangle)

It has a position, size and colour, and also a direction of movement (only the bat uses this)

```
10 public class GameObj
11 {
12     // state variables for a game object
13     public boolean visible  = true;     // Can be seen on the screen (change to false when the
14     public int topX   = 0;              // Position - top left corner X
15     public int topY   = 0;              // position - top left corner Y
16     public int width  = 0;              // Width of object
17     public int height = 0;              // Height of object
18     public Color colour;                // Colour of object
19     public int   dirX   = 1;            // Direction X (1, 0 or -1)
20     public int   dirY   = 1;            // Direction Y (1, 0 or -1)
21
22
23     public GameObj( int x, int y, int w, int h, Color c )
24     {
25         topX   = x;
26         topY = y;
27         width  = w;
28         height = h;
29         colour = c;
30     }
31
32     // move in x axis
33     public void moveX( int units )
34     {
35         topX += units * dirX;
36     }
37
38     // move in y axis
39     public void moveY( int units )
40     {
```

GameObject class hitBy

- GameObject has methods to move the object (in the current direction) and also to change direction (in the X or Y axis)
- Its most important method is hitBy, which returns true if this object has been hit by (ie overlaps with) the object provided as argument
- This is how the game knows when the ball hits a brick or the bat

```
38  // move in y axis
39  public void moveY( int units )
40  {
41      topY += units * dirY;
42  }
43
44  // change direction of movement in x axis (-1, 0 or +1)
45  public void changeDirectionX()
46  {
47      dirX = -dirX;
48  }
49
50  // change direction of movement in y axis (-1, 0 or +1)
51  public void changeDirectionY()
52  {
53      dirY = -dirY;
54  }
55
56  // Detect collision between this object and the argument object
57  // It's easiest to work out if they do NOT overlap, and then
58  // return the opposite
59  public boolean hitBy( GameObj obj )
60  {
61      boolean separate =
62          topX >= obj.topX+obj.width      ||    // '||' means 'or'
63          topX+width <= obj.topX          ||
64          topY >= obj.topY+obj.height     ||
65          topY+height <= obj.topY ;
66
67      // use ! to return the opposite result - hitBy is 'not separate')
68      return(! separate);
69
70  }
71
```

Lab exercises
Week 2.01

Breakout lab exercises

- Download the Breakout game as a BlueJ (or Eclipse) project
- Run it (as a JavaFX application)
  - you will see a simple game screen with a moving ball, a score
  - Also you can use the < and > to move the bat at the bottom of the screen
  - But there are are no bricks!
- The lab exercise for this week is to get a general idea of how the Breakout program works, and add code to it for a single row of the bricks
- The Seminar slides give some screenshots to help you with this

Breakout – adding bricks

- There are three places in the program where you need to add code:
  - Model class
    - add bricks to the model (the shell has no bricks!) - code point [1]
  - View class
    - display the bricks on screen - code point [2]
  - Model class
    - if the ball hits a brick, the brick disappears – code point [3]

- There are big comments in the code to show you where this code should go