University of Reading

INTRODUCTION TO FUNCTIONS

This document supports the 'Introduction to functions' video.

PLEASE NOTE: The word 'collusion' in this document should be replaced with 'collision'. We have chosen not to make this change in the document as we are unable to update the code at this point. This whole block of code (notice the curly brackets) is the 'setupBeginning' function.

```
//This is run before a new game (also after an old game)
@Override
public void setupBeginning() {
	//Initialise speeds
	//mCanvasWidth and mCanvasHeight are declared and managed elsewhere
	mBallSpeedX = mCanvasWidth / 3;
	mBallSpeedY = mCanvasHeight / 3;

	//Place the ball in the middle of the screen.
	/*mBall.Width() and mBall.getHeigh() gives us the height and width of the image of
the ball*/
	mBallX = mCanvasWidth / 2;
	mBallY = mCanvasHeight / 2;

	//Place Paddle in the middle of the screen
	mPaddleX = mCanvasWidth / 2;

	//Place SmileyBall in the top middle of the screen
	mSmileyBallX = mCanvasWidth / 2;
	mSmileyBallY = mSmileyBall.getHeight()/2;

	//Place all SadBalls forming a pyramid underneath the SmileyBall
	mSadBallX[0] = mCanvasWidth / 3;
	mSadBallY[0] = mCanvasHeight / 3;

	mSadBallX[1] = mCanvasWidth - mCanvasWidth / 3;
	mSadBallY[1] = mCanvasHeight / 3;

	mSadBallX[2] = mCanvasWidth / 2;
	mSadBallY[2] = mCanvasHeight / 5;

	//Get the minimum distance between a small ball and a bigball
	//We leave out the square root to limit the calculations of the program
	//Remember to do that when testing the distance as well
	mMinDistanceBetweenRedBallAndBigBall = (mPaddle.getWidth() / 2 + mBall.getWidth()
/ 2) * (mPaddle.getWidth() / 2 + mBall.getWidth() / 2);

	}
```

Looking up where a function is called in Eclipse

Select (highlight) the function and right-click on it. From the menu select 'References' then you will be shown another menu. Select 'Workspace' (Figure 1).

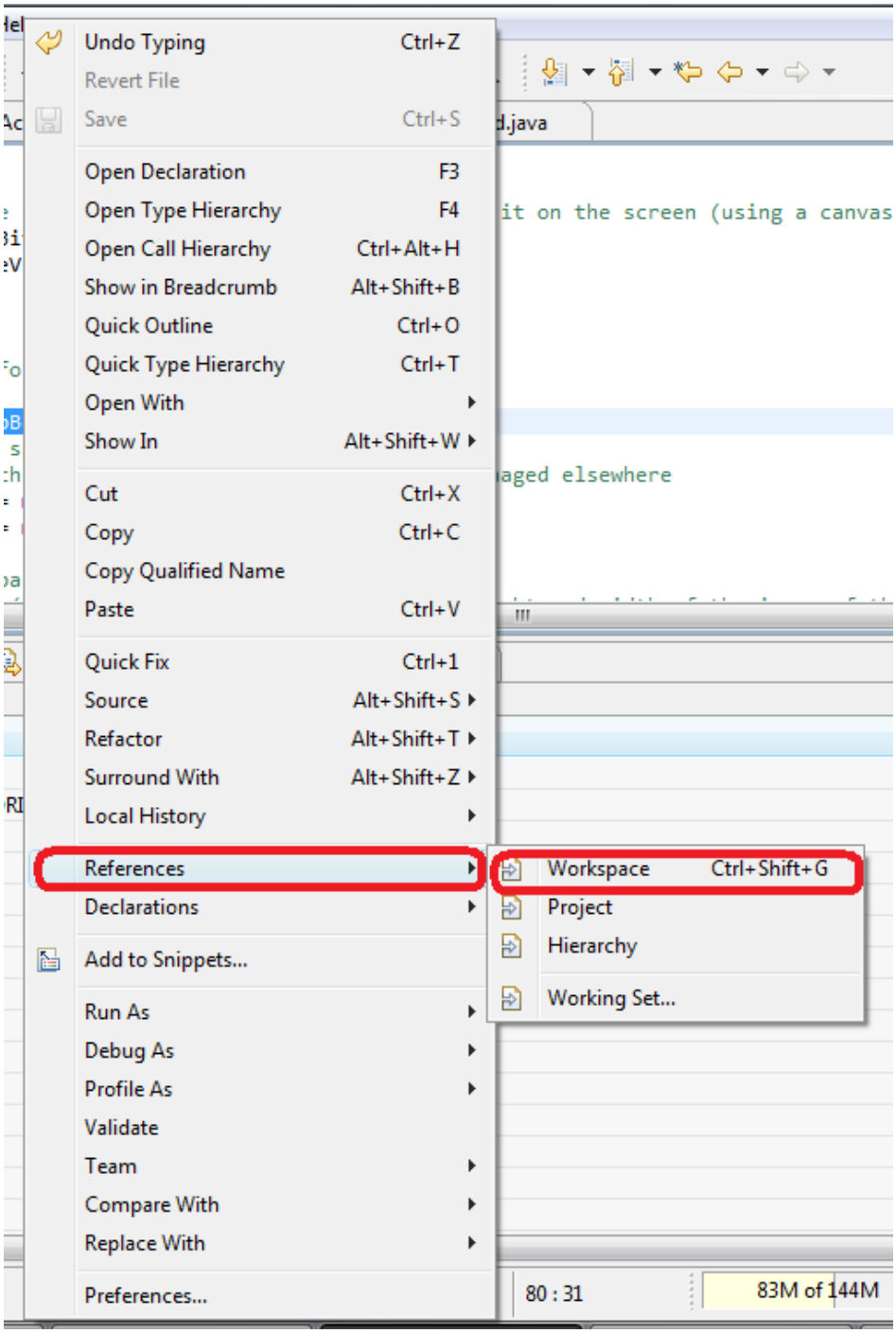

Figure 1: Looking up where a Function is called from

This will open up a 'Search' pane showing where the function is called from (Figure 2). Notice that 'setupBeginning' is called from the 'doStart' function.

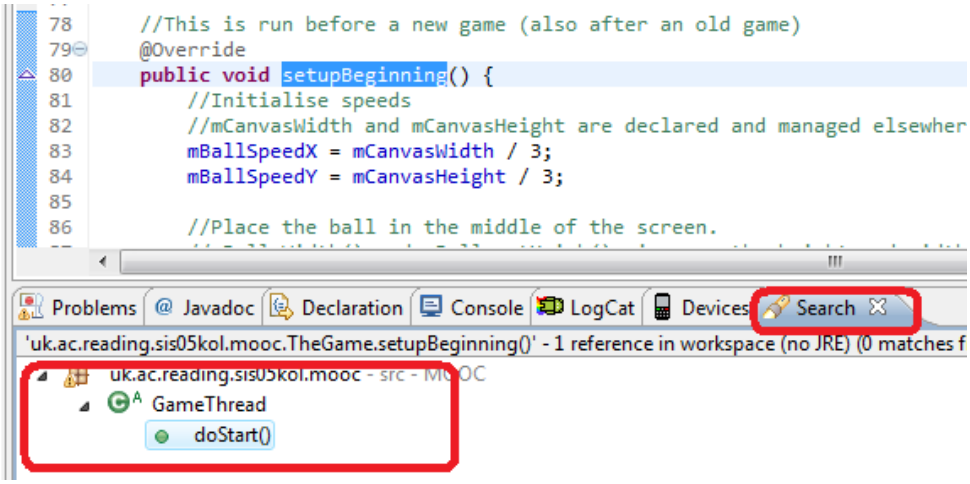

Figure 2: setupBeginning() called from doStart()

Now double-click on 'doStart()' shown in Figure 2.

You will be taken to the 'doStart' function.

Figure 3: doStart()

Calling functions

When the 'doStart()' function is called from somewhere in our program, the block of code shown in Figure 3 (including the 'setupBeginning' function) gets executed.

When line 89 in Figure 3 gets executed, the 'setupBeginning' function gets called.
Now, the statements in the 'setupBeginning' function (listed in Page 1) start to execute, one by one.

When all statements in the 'setupBeginning' function finish execution, the lines following the call to setupBeginning (in doStart) start to execute.

At first this jump from one piece of code to another can be confusing. However, this is a very useful and powerful tool in programming.

This flow of execution is illustrated in

Figure 4.

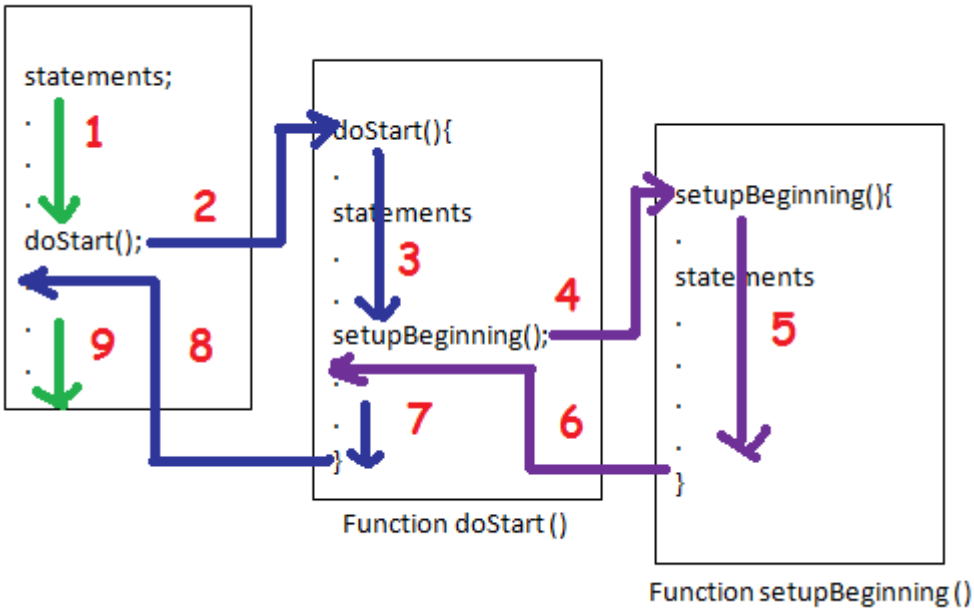

Figure 4: Call to functions

1: Statements in the program execute until the line where the function 'doStart' is called to begin execution.

2: Now the 'doStart' function gets called.

3: Statements in the 'doStart' function are executed sequentially until the call for 'setupBeginning'.

4: Now the 'setupBeginning' function gets called.

5: Statements in the 'setupBeginning' function are executed sequentially until the end.

6: At the end of the execution of the 'setupBeginning' function, it is the turn of the line following the call to 'setupBeginning' to execute (we call this as the 'setupBeginning' function returns control to the calling function – in this case 'doStart').

7: The rest of the statements following 'setupBeginning' in 'doStart' are executed.

8: At the end of the 'doStart' function, the control is returned to the caller.

9: The rest of the statements following the 'doStart' function are executed.

In the video, this execution is shown by using a breakpoint, and "stepping into" the code.

Using the debugger to view the control flow of function calls

In order to see this in action, the debugger can be used.

In the GameThread.java file, inside the doStart() function, place a breakpoint at the calling point of setupBeginning()

Note: To setup a breakpoint double-click on the left side margin of the IDE's editor.

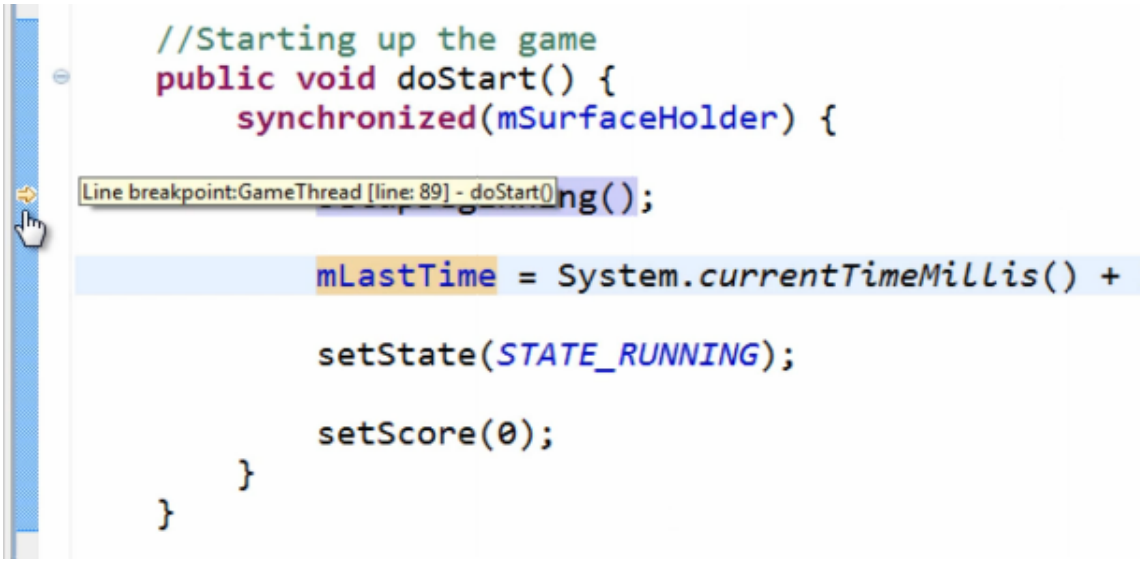

Figure 5: Setting up breakpoint at call to setupBeginning()

Now start the program in the debug mode (Figure 6).

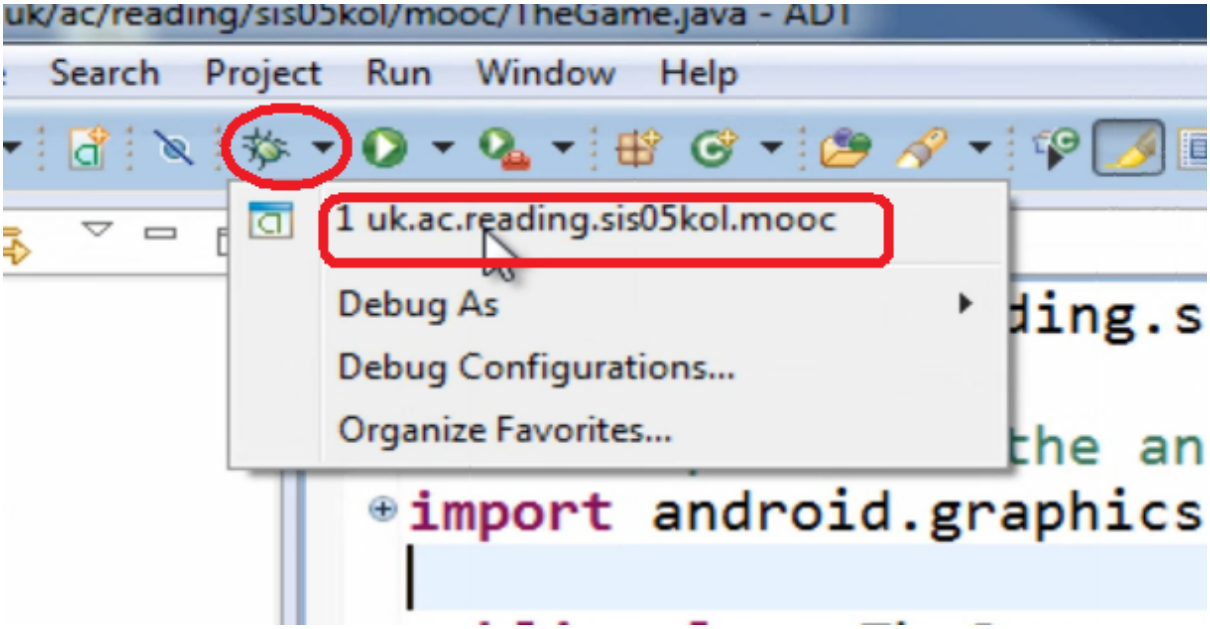

Figure 6: Running program in debug mode

Clicking on the emulator screen will take you to the debug mode. If you get 'Confirm Perspective Switch' message (Figure 7) displayed, select 'Yes' to continue.

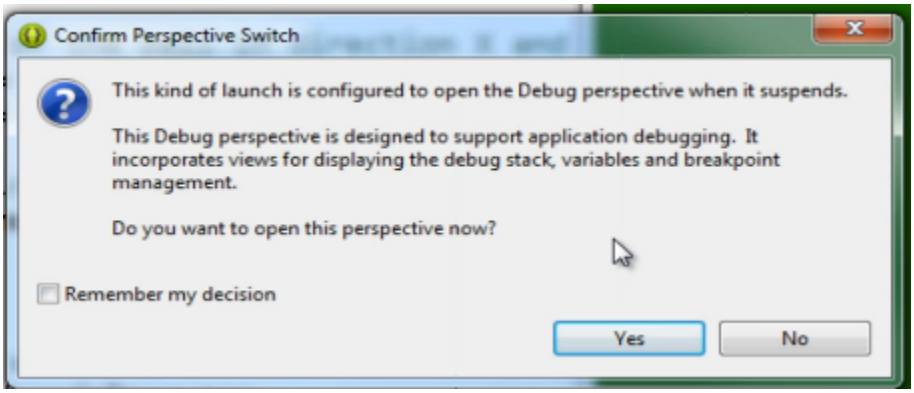

Figure 7: confirm perspective switch dialog

Now you should be taken to the debug perspective which will look similar to this:

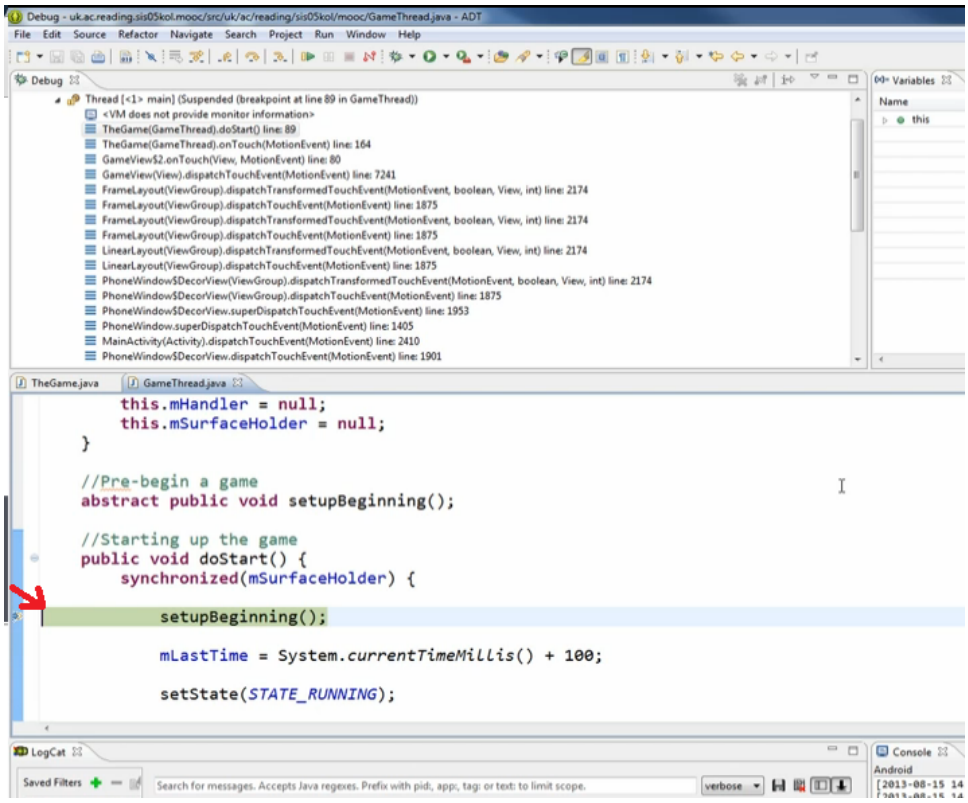

Figure 8: Execution Stopped at setupBeginning()

The execution has stopped just before calling the function setupBeginning().

In order to go into the function to see what happens inside, you will have to select the 'Step Into' option (F5) in the debugging mode (Figure 9).

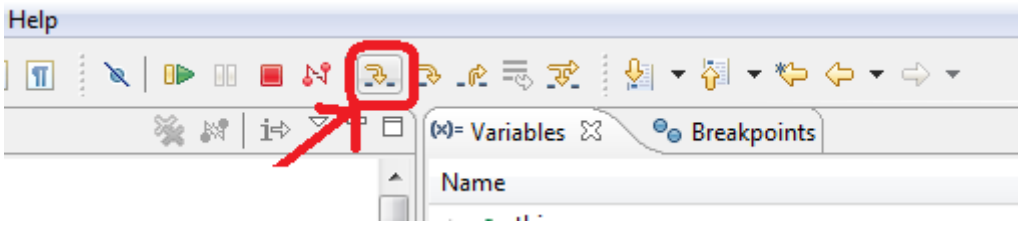

Figure 9: step into option in debugging

Step into will take you to the function, setupBeginning().

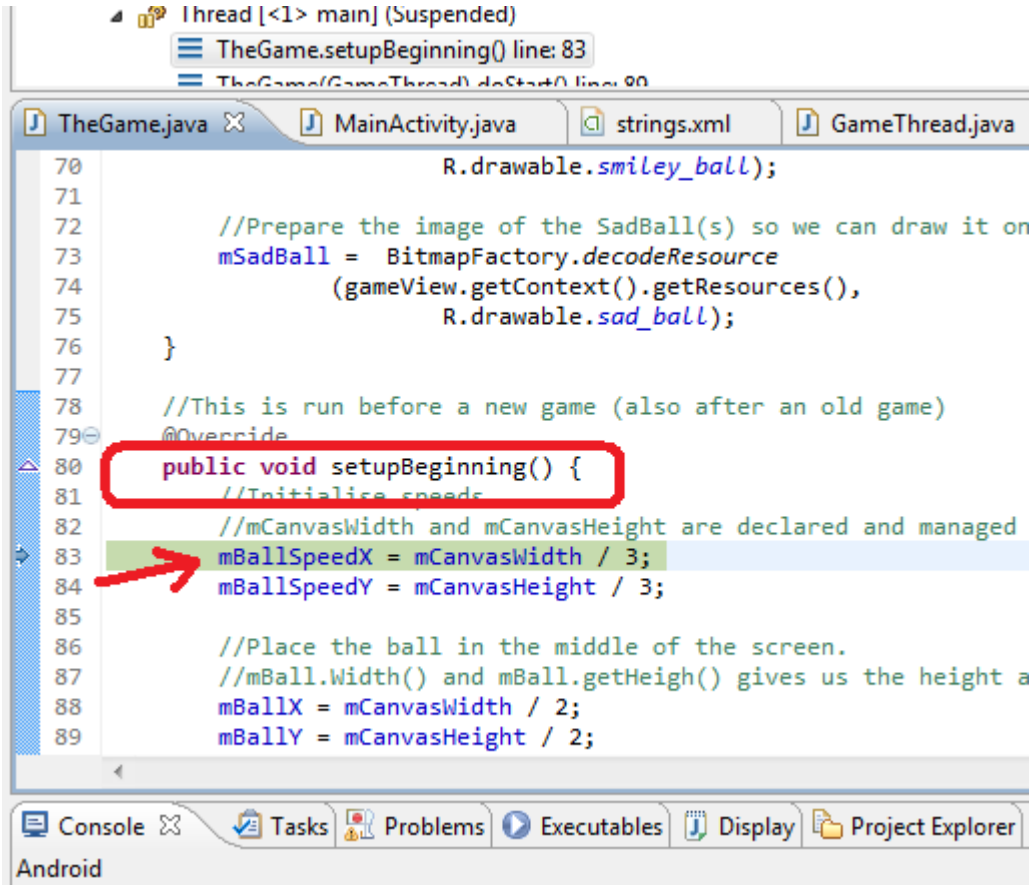

Figure 10: setupBeginning() in debug mode

Next the 'step over' tool (F6) can be used to execute the statements, line by line.

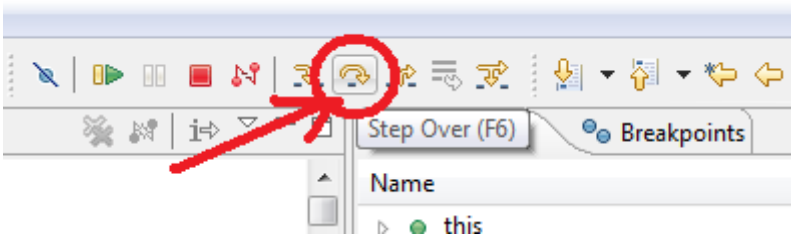

Figure 11: Step Over Option in Debugging

At the end of the setupBeginning() function, notice how it goes back to the doStart(), and continues the execution of statements following the call to the setupBeginning() .

```
78      //This is run before a new game (also after an old game)
79      @Override
80      public void setupBeginning() {
81          //Initialise speeds
82          //mCanvasWidth and mCanvasHeight are declared and managed elsewhere
83          mBallSpeedX = mCanvasWidth / 3;
84          mBallSpeedY = mCanvasHeight / 3;
85
86          //Place the ball in the middle of the screen.
87          //mBall.Width() and mBall.getHeigh() gives us the height and width of the ima
88          mBallX = mCanvasWidth / 2;
89          mBallY = mCanvasHeight / 2;
90
91          //Place Paddle in the middle of the screen
92          mPaddleX = mCanvasWidth / 2;
93
94          //Place SmileyBall in the top middle of the screen
95          mSmileyBallX = mCanvasWidth / 2;
96          mSmileyBallY = mSmileyBall.getHeight()/2;
97
98          //Place all SadBalls forming a pyramid underneath the SmileyBall
99          mSadBallX[0] = mCanvasWidth / 3;
100         mSadBallY[0] = mCanvasHeight / 3;
101
102         mSadBallX[1] = mCanvasWidth - mCanvasWidth / 3;
103         mSadBallY[1] = mCanvasHeight / 3;
104
105         mSadBallX[2] = mCanvasWidth / 2;
106         mSadBallY[2] = mCanvasHeight / 5;
107
108         //Get the minimum distance between a small ball and a bigball
109         //We leave out the square root to limit the calculations of the program
110         //Remember to do that when testing the distance as well
111         mMinDistanceBetweenRedBallAndBigBall = (mPaddle.getWidth() / 2 + mBall.getWid
112     }
```

MOOC/src/uk/ac/readin

Figure 12: End of setupBeginning() Function

TheGame.java | MainActivity.java | strings.xml | GameThread.java

```
81
82      //Pre-begin a game
83      abstract public void setupBeginning();
84
85      //Starting up the game
86      public void doStart() {
87          synchronized(mSurfaceHolder) {
88
89              setupBeginning();
90
91              mLastTime = System.currentTimeMillis() + 100;
92
93              setState(STATE_RUNNING);
94
95              setScore(0);
96          }
97      }
```

Figure 13: Returning control to doStart()

Here, using the debugging option, we have shown how the functions are called, and what happens when code in a function finishes execution. You can revisit Figure 4 to check your understanding of the sequence of execution when a function is called.

Now we will look at creating functions.

Creating Functions

Functions are a set of instructions bundled together to achieve a specific outcome. They are a good alternative to having blocks of code repeating throughout program.

Where in the game code could benefit from using a function?

An example can be found in updateGame() function in TheGame.java file.

```
protected void updateGame(float secondsElapsed)
{
       float distanceBetweenBallAndPaddle;
       //If the ball moves down on the screen perform potential paddle collision
       if(mBallSpeedY > 0)
       {
          /*Get actual distance (without square root - remember?) between the mBall
and the ball being checked*/
              distanceBetweenBallAndPaddle = (mPaddleX - mBallX) * (mPaddleX -
mBallX) + (mCanvasHeight - mBallY) *(mCanvasHeight - mBallY);

          //Check if the actual distance is lower than the allowed => collision
          if(mMinDistanceBetweenRedBallAndBigBall >= distanceBetweenBallAndPaddle)
          {

             /*Get the present velocity (this should also be the velocity going away
after the collision)*/
              float velocityOfBall = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);

                    //Change the direction of the ball
                    mBallSpeedX = mBallX - mPaddleX;
                    mBallSpeedY = mBallY - mCanvasHeight;

                    //Get the velocity after the collision
                    float newVelocity = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);

                    /*using the fraction between the original velocity and present
velocity to calculate the needed*/
                    /*speeds in X and Y to get the original velocity but with the new
angle*/
                    mBallSpeedX = mBallSpeedX * velocityOfBall / newVelocity;
                    mBallSpeedY = mBallSpeedY * velocityOfBall / newVelocity;

                }
             }

             //Move the ball's X and Y using the speed (pixel/sec)
             mBallX = mBallX + secondsElapsed * mBallSpeedX;
             mBallY = mBallY + secondsElapsed * mBallSpeedY;

             //Check if ball hits either the left side or right side of screen
             //But only do if ball is moving towards that side of screen
             //If it does that => change the direction of ball in the X direction
```

```
            if((mBallX <= mBall.getWidth() / 2 && mBallSpeedX < 0) || (mBallX >=
mCanvasWidth - mBall.getWidth() / 2 && mBallSpeedX > 0) )
            {
                  mBallSpeedX = -mBallSpeedX;
            }

            distanceBetweenBallAndPaddle = (mSmileyBallX - mBallX) * (mSmileyBallX
- mBallX) + (mSmileyBallY - mBallY) *(mSmileyBallY - mBallY);

            //Check if actual distance is lower than the allowed => collision
            if(mMinDistanceBetweenRedBallAndBigBall >=
distanceBetweenBallAndPaddle)
            {

                  /*Get the present velocity (this should also be the velocity
going away after the collision)*/
                  float velocityOfBall = (float) Math.sqrt(mBallSpeedX*mBallSpeedX
+ mBallSpeedY*mBallSpeedY);
                  //Change the direction of the ball
                  mBallSpeedX = mBallX - mSmileyBallX;
                  mBallSpeedY = mBallY - mSmileyBallY;
                  //Get the velocity after the collision
                  float newVelocity = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);
                  /*using the fraction between the original velocity and present
velocity to calculate the needed*/
                  /*speeds in X and Y to get the original velocity but with the new
angle*/
                  mBallSpeedX = mBallSpeedX * velocityOfBall / newVelocity;
                  mBallSpeedY = mBallSpeedY * velocityOfBall / newVelocity;

                  //Increase score
                  updateScore(1);
            }

            //Loop through all SadBalls
            for(int i = 0; i < mSadBallX.length; i++)
            {
                  /*Perform collisions (if necessary) between SadBall in position i
and the red ball*/
                  /*Get actual distance (without square root - remember?) between
the mBall and the ball being checked*/
                  distanceBetweenBallAndPaddle = (mSadBallX[i] - mBallX) *
(mSadBallX[i] - mBallX) + (mSadBallY[i] - mBallY) *(mSadBallY[i] - mBallY);

                  /*Check if the actual distance is lower than the allowed =>
collision*/
                  if(mMinDistanceBetweenRedBallAndBigBall >=
distanceBetweenBallAndPaddle)
                  {
                        /*Get the present velocity (this should also be the
velocity going away after the collision)*/
                        float velocityOfBall = (float)
Math.sqrt(mBallSpeedX*mBallSpeedX + mBallSpeedY*mBallSpeedY);
                        //Change the direction of the ball
                        mBallSpeedX = mBallX - mSadBallX[i];
                        mBallSpeedY = mBallY - mSadBallY[i];
                        //Get the velocity after the collision
                        float newVelocity = (float)
Math.sqrt(mBallSpeedX*mBallSpeedX + mBallSpeedY*mBallSpeedY);
```

```
                        /*using the fraction between the original velocity and
present velocity to calculate the needed*/
                        /*speeds in X and Y to get the original velocity but with
the new angle*/
                        mBallSpeedX = mBallSpeedX * velocityOfBall / newVelocity;
                        mBallSpeedY = mBallSpeedY * velocityOfBall / newVelocity;
                    }
                }
            /*If the ball goes out of the top of the screen and moves towards the
top of the screen => */
            //change the direction of the ball in the Y direction
            if(mBallY <= mBall.getWidth() / 2 && mBallSpeedY < 0)
            {
                    mBallSpeedY = -mBallSpeedY;
            }
            //If the ball goes out of the bottom of the screen => lose the game
            if(mBallY >= mCanvasHeight)
            {
                    setState(GameThread.STATE_LOSE);
            }
      }
```

This, as you can see is a function with many statements. However, there are particular segments of code that are similar. We have highlighted them here for clarity – Block 1 and Block 2.

```
protected void updateGame(float secondsElapsed)
{
      float distanceBetweenBallAndPaddle;
      //If the ball moves down on the screen perform potential paddle collision
      if(mBallSpeedY > 0)
      {
//BLOCK 1 BEGINS
         /*Get actual distance (without square root - remember?) between the mBall
and the ball being checked*/
            distanceBetweenBallAndPaddle = (mPaddleX - mBallX) * (mPaddleX -
mBallX) + (mCanvasHeight - mBallY) *(mCanvasHeight - mBallY);

         //Check if the actual distance is lower than the allowed => collision
         if(mMinDistanceBetweenRedBallAndBigBall >= distanceBetweenBallAndPaddle)
         {
            /*Get the present velocity (this should also be the velocity going away
after the collision)*/
            float velocityOfBall = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);

                  //Change the direction of the ball
                  mBallSpeedX = mBallX - mPaddleX;
                  mBallSpeedY = mBallY - mCanvasHeight;

                  //Get the velocity after the collision
                  float newVelocity = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);

                  /*using the fraction between the original velocity and present
velocity to calculate the needed*/
                  /*speeds in X and Y to get the original velocity but with the new
angle*/
                  mBallSpeedX = mBallSpeedX * velocityOfBall / newVelocity;
                  mBallSpeedY = mBallSpeedY * velocityOfBall / newVelocity;

            }
          }//BLOCK 1 ENDS
```

```
            //Move the ball's X and Y using the speed (pixel/sec)
            mBallX = mBallX + secondsElapsed * mBallSpeedX;
            mBallY = mBallY + secondsElapsed * mBallSpeedY;

            //Check if ball hits either the left side or right side of screen
            //But only do if ball is moving towards that side of screen
            //If it does that => change the direction of ball in the X direction
            if((mBallX <= mBall.getWidth() / 2 && mBallSpeedX < 0) || (mBallX >=
mCanvasWidth - mBall.getWidth() / 2 && mBallSpeedX > 0) )
            {
                  mBallSpeedX = -mBallSpeedX;
            }

            distanceBetweenBallAndPaddle = (mSmileyBallX - mBallX) * (mSmileyBallX
- mBallX) + (mSmileyBallY - mBallY) *(mSmileyBallY - mBallY);

            //Check if actual distance is lower than the allowed => collision
            if(mMinDistanceBetweenRedBallAndBigBall >=
distanceBetweenBallAndPaddle)
            {

                  /*Get the present velocity (this should also be the velocity
going away after the collision)*/
                  float velocityOfBall = (float) Math.sqrt(mBallSpeedX*mBallSpeedX
+ mBallSpeedY*mBallSpeedY);
                  //Change the direction of the ball
                  mBallSpeedX = mBallX - mSmileyBallX;
                  mBallSpeedY = mBallY - mSmileyBallY;
                  //Get the velocity after the collision
                  float newVelocity = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);
                  /*using the fraction between the original velocity and present
velocity to calculate the needed*/
                  /*speeds in X and Y to get the original velocity but with the new
angle*/
                  mBallSpeedX = mBallSpeedX * velocityOfBall / newVelocity;
                  mBallSpeedY = mBallSpeedY * velocityOfBall / newVelocity;

                  //Increase score
                  updateScore(1);
            }

            //Loop through all SadBalls
            for(int i = 0; i < mSadBallX.length; i++)
            {
                  /*Perform collisions (if necessary) between SadBall in position i
and the red ball*/
                  /*Get actual distance (without square root - remember?) between
the mBall and the ball being checked*/

//BLOCK 2 BEGINS
                  distanceBetweenBallAndPaddle = (mSadBallX[i] - mBallX) *
(mSadBallX[i] - mBallX) + (mSadBallY[i] - mBallY) *(mSadBallY[i] - mBallY);

                  /*Check if the actual distance is lower than the allowed =>
collision*/
                  if(mMinDistanceBetweenRedBallAndBigBall >=
distanceBetweenBallAndPaddle)
                  {
```

```
                        /*Get the present velocity (this should also be the
velocity going away after the collision)*/
                        float velocityOfBall = (float)
Math.sqrt(mBallSpeedX*mBallSpeedX + mBallSpeedY*mBallSpeedY);
                        //Change the direction of the ball
                        mBallSpeedX = mBallX - mSadBallX[i];
                        mBallSpeedY = mBallY - mSadBallY[i];
                        //Get the velocity after the collision
                        float newVelocity = (float)
Math.sqrt(mBallSpeedX*mBallSpeedX + mBallSpeedY*mBallSpeedY);

                        /*using the fraction between the original velocity and
present velocity to calculate the needed*/
// CONTINUED ON NEXT PAGE
                        /*speeds in X and Y to get the original velocity but with
the new angle*/
                        mBallSpeedX = mBallSpeedX * velocityOfBall / newVelocity;
                        mBallSpeedY = mBallSpeedY * velocityOfBall / newVelocity;
                    }
// BLOCK 2 ENDS
                }

            /*If the ball goes out of the top of the screen and moves towards the
      top of the screen => */
            //change the direction of the ball in the Y direction
            if(mBallY <= mBall.getWidth() / 2 && mBallSpeedY < 0)
            {
                mBallSpeedY = -mBallSpeedY;
            }

            //If the ball goes out of the bottom of the screen => lose the game
            if(mBallY >= mCanvasHeight)
            {
                setState(GameThread.STATE_LOSE);
            }

}
```

Only these two blocks need to be investigated, at the moment, to identify the similarities.

Investigating the two code blocks show that they are performing the same task but based on different inputs. In code block 1 mPaddleX and mCanvasHeight are used while in code block2 the corresponding values are mSadBallX[i] and mSadBallY[i].

This is a good example of where a function can be used to replace a repeating block of code.

Code Block 1

```
distanceBetweenBallAndPaddle = (mPaddleX - mBallX) * (mPaddleX - mBallX) +
(mCanvasHeight - mBallY) *(mCanvasHeight - mBallY);

        //Check if the actual distance is lower than the allowed => collision
        if(mMinDistanceBetweenRedBallAndBigBall >= distanceBetweenBallAndPaddle)
        {
            /*Get the present velocity (this should also be the velocity going away
after the collision)*/
            float velocityOfBall = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);
                    //Change the direction of the ball
```

```
                    mBallSpeedX = mBallX - mPaddleX;
                    mBallSpeedY = mBallY - mCanvasHeight;
                    //Get the velocity after the collision
                    float newVelocity = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);
                    /*using the fraction between the original velocity and present
velocity to calculate the needed*/
                    /*speeds in X and Y to get the original velocity but with the new
angle*/
                    mBallSpeedX = mBallSpeedX * velocityOfBall / newVelocity;
                    mBallSpeedY = mBallSpeedY * velocityOfBall / newVelocity;
                }
```

Code Block 2

```
distanceBetweenBallAndPaddle = (mSadBallX[i] - mBallX) * (mSadBallX[i] - mBallX) +
(mSadBallY[i] - mBallY) *(mSadBallY[i] - mBallY);

            /*check if the actual distance is lower than the allowed => collision*/
            if(mMinDistanceBetweenRedBallAndBigBall >=
distanceBetweenBallAndPaddle)
            {
                    /*Get the present velocity (this should also be the velocity
going away after the collision)*/
                    float velocityOfBall = (float) Math.sqrt(mBallSpeedX*mBallSpeedX
+ mBallSpeedY*mBallSpeedY);
                    //Change the direction of the ball
                    mBallSpeedX = mBallX - mSadBallX[i];
                    mBallSpeedY = mBallY - mSadBallY[i];
                    //Get the velocity after the collision
                    float newVelocity = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);
                    /*using the fraction between the original velocity and present
            velocity to calculate the needed*/
                    /*speeds in X and Y to get the original velocity but with the new
angle*/
                    mBallSpeedX = mBallSpeedX * velocityOfBall / newVelocity;
                    mBallSpeedY = mBallSpeedY * velocityOfBall / newVelocity;
            }
```

Now have a look at this function: updateBallCollusion. Did you notice that this new function is doing the same thing done by code block1 and code block2? What is the difference? This function takes two arguments (values passed into a function are called parameters or arguments) of type float: x and y. (Refer to the Functions Tutorial in Activity 6.3 for more details.)

```
//Collusion control between mBall and another big ball
private void updateBallCollusion(float x, float y)
{
      /*Get actual distance (without square root - remember?) between the mBall and
the ball being checked*/
      float distanceBetweenBallAndPaddle = (x - mBallX) * (x - mBallX) + (y -
mBallY) *(y - mBallY);
      //Check if the actual distance is lower than the allowed => collision
      if(mMinDistanceBetweenRedBallAndBigBall >= distanceBetweenBallAndPaddle)
      {
            /*Get the present velocity (this should also be the velocity going away
      after the collision)*/
```

```
            float velocityOfBall = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);
            //Change the direction of the ball
            mBallSpeedX = mBallX - x;
            mBallSpeedY = mBallY - y;
            //Get the velocity after the collision
            float newVelocity = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);
            /*using the fraction between the original velocity and present velocity
      to calculate the needed*/
            /*speeds in X and Y to get the original velocity but with the new
angle*/
            mBallSpeedX = mBallSpeedX * velocityOfBall / newVelocity;
            mBallSpeedY = mBallSpeedY * velocityOfBall / newVelocity;
      }
}
```

If you carefully look at Code Block 1 and this new Function, you will see that, instead of mPaddleX and mCanvasHeight, this function is using x and y. Similarly, comparing with Code Block 2 will show you that instead of mSadBallX[i] and mSadBallY[i] this function is using x and y.

Thus we can replace the whole of Code Block1 with a call to function 'updateBallCollusion' with parameters 'mPaddleX' and 'mCanvasHeight'

updateBallCollusion (mPaddleX, mCanvasHeight);

This function call will call the 'updateBallCollusion' function, replacing the use of 'x' with 'mPaddleX' and 'y' with 'mCanvasHeight'.

Can you write a function call to replace the whole of Code Block 2?

Compare your answer with the answer given in the next page.

updateBallCollusion (mSadBallX[i], mSadBallY[i]);

This function call will call the 'updateBallCollusion' function, replacing the use of 'x' with 'mSadBallX[i]' and 'y' with 'mSadBallY[i]'.

Once the changes are made, the **updateGame** function will look like this:

```
protected void updateGame(float secondsElapsed)
{
	float distanceBetweenBallAndPaddle;
	//If the ball moves down on the screen perform potential paddle collision
	if(mBallSpeedY > 0)
	{
		updateBallCollusion (mPaddleX, mCanvasHeight);
	}

	//Move the ball's X and Y using the speed (pixel/sec)
	mBallX = mBallX + secondsElapsed * mBallSpeedX;
	mBallY = mBallY + secondsElapsed * mBallSpeedY;

	//Check if ball hits either the left side or right side of screen
	//But only do if ball is moving towards that side of screen
	//If it does that => change the direction of ball in the X direction
	if((mBallX <= mBall.getWidth() / 2 && mBallSpeedX < 0) || (mBallX >=
mCanvasWidth - mBall.getWidth() / 2 && mBallSpeedX > 0) )
	{
			mBallSpeedX = -mBallSpeedX;
	}

	distanceBetweenBallAndPaddle = (mSmileyBallX - mBallX) * (mSmileyBallX -
mBallX) + (mSmileyBallY - mBallY) *(mSmileyBallY - mBallY);

	//Check if actual distance is lower than the allowed => collision
	if(mMinDistanceBetweenRedBallAndBigBall >= distanceBetweenBallAndPaddle)
	{

		/*Get the present velocity (this should also be the velocity going away
after the collision)*/
		float velocityOfBall = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);
		//Change the direction of the ball
		mBallSpeedX = mBallX - mSmileyBallX;
		mBallSpeedY = mBallY - mSmileyBallY;
		//Get the velocity after the collision
		float newVelocity = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);
		/*using the fraction between the original velocity and present velocity
to calculate the needed*/
		/*speeds in X and Y to get the original velocity but with the new
angle*/
		mBallSpeedX = mBallSpeedX * velocityOfBall / newVelocity;
		mBallSpeedY = mBallSpeedY * velocityOfBall / newVelocity;

			//Increase score
		updateScore(1);
	}

	//Loop through all SadBalls
	for(int i = 0; i < mSadBallX.length; i++)
	{
```

```
            /*Perform collisions (if necessary) between SadBall in position i and
the red ball*/
            /*Get actual distance (without square root - remember?) between the
mBall and the ball being checked*/
            updateBallCollusion (mSadBallX[i], mSadBallY[i]);
      }
      /*If the ball goes out of the top of the screen and moves towards the top of
the screen => */
      //change the direction of the ball in the Y direction
      if(mBallY <= mBall.getWidth() / 2 && mBallSpeedY < 0)
      {
            mBallSpeedY = -mBallSpeedY;
      }

      //If the ball goes out of the bottom of the screen => lose the game
      if(mBallY >= mCanvasHeight)
      {
            setState(GameThread.STATE_LOSE);
      }
}
```

You can use the debugger to look at how this code will be executed, as you have done before.

A breakpoint can be placed inside the 'for' loop for sad balls (

Figure 8). Double-click on the left-hand side edge of the editor as shown in

Figure 8 to create a breakpoint. Now, start the program in the debug mode and see how the execution takes place (Figure 6). To step into a function you can use F5 (Figure 9), and to step over you can use F6 (

Figure 11). To resume you can use F8.

If you need more help with debugging you can refer to the 'Debugging help sheet' provided in Week 2 or look at the section 'Using the debugger to view the control flow of function calls' in this document.

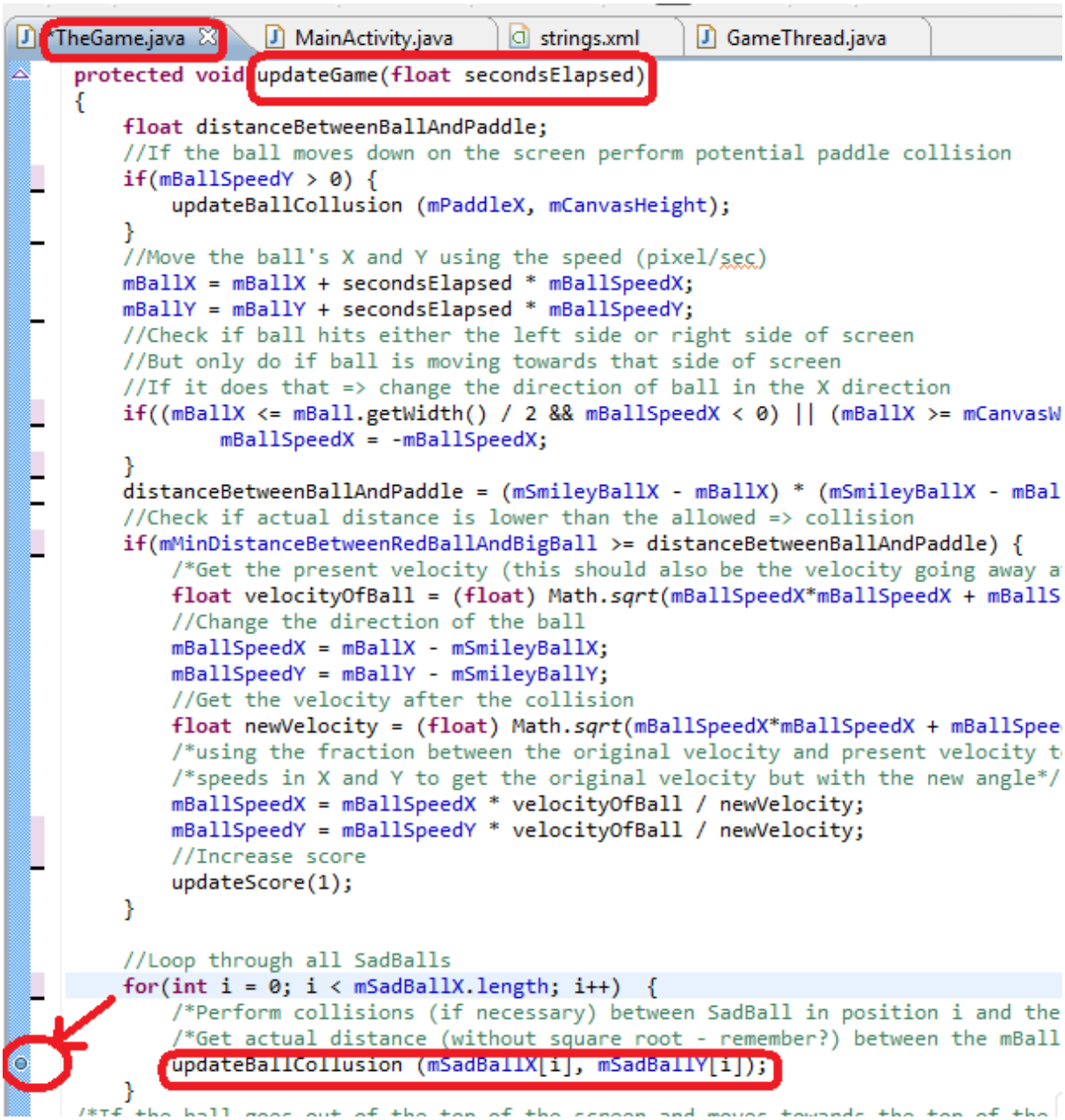

Figure 14: Breakpoint at Function updateBallCollusion(x, y)

Using return values in functions

There is another place in the code where the function 'updateBallCollusion' can be used.

```
distanceBetweenBallAndPaddle = (mSmileyBallX - mBallX) * (mSmileyBallX - mBallX) +
(mSmileyBallY - mBallY) *(mSmileyBallY - mBallY);
if(mMinDistanceBetweenRedBallAndBigBall >= distanceBetweenBallAndPaddle)
{
	/*Get the present velocity (this should also be the velocity going away after
the collision)*/
	float velocityOfBall = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);
	//Change the direction of the ball
	mBallSpeedX = mBallX - mSmileyBallX;
	mBallSpeedY = mBallY - mSmileyBallY;
	//Get the velocity after the collision
```

```
        float newVelocity = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);
        /*using the fraction between the original velocity and present velocity to
calculate the needed*/
        /*speeds in X and Y to get the original velocity but with the new angle*/
        mBallSpeedX = mBallSpeedX * velocityOfBall / newVelocity;
        mBallSpeedY = mBallSpeedY * velocityOfBall / newVelocity;

        //Increase score
        updateScore(1);
}
```

If the 'updateBallCollusion' function returned a value to say whether there was a collision or not you could use it here.

Currently, the 'updateBallCollusion' function does not return a value (the return type is void)

void updateBallCollusion(float x, float y)

Now this function can be converted to return a boolean value.

boolean updateBallCollusion(float x, float y)

The function is now defined to return a value, but inside the function statements need to be written to make sure the function returns the values. If there had been a collision it would return true, and if not it would return false.

```
private boolean updateBallCollusion(float x, float y)
{
        /*Get actual distance (without square root - remember?) between the mBall and
the ball being checked*/
        float distanceBetweenBallAndPaddle = (x - mBallX) * (x - mBallX) + (y -
mBallY) *(y - mBallY);

        //Check if the actual distance is lower than the allowed => collision
        if(mMinDistanceBetweenRedBallAndBigBall >= distanceBetweenBallAndPaddle) {

          /*Get the present velocity (this should also be the velocity going away
after the collision)*/
        float velocityOfBall = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);

        //Change the direction of the ball
        mBallSpeedX = mBallX - x;
        mBallSpeedY = mBallY - y;

        //Get the velocity after the collision
        float newVelocity = (float) Math.sqrt(mBallSpeedX*mBallSpeedX +
mBallSpeedY*mBallSpeedY);

        //using the fraction between the original velocity and present velocity to
calculate the needed
        //speeds in X and Y to get the original velocity but with the new angle.
        mBallSpeedX = mBallSpeedX * velocityOfBall / newVelocity;
        mBallSpeedY = mBallSpeedY * velocityOfBall / newVelocity;

        //====== A collision happened so we return true here
        return true;
    }
    //====== No collision happened so we return false here
    return false;
}
```

When a 'return' statement in a function is executed, it returns the control back to the caller. In other words, the function stops executing, and the value given in the return statement is returned to the point where the function was called.

Next, the returned value in the 'updateGame' needs to be used. So, instead of using the 'if' condition:

if(mMinDistanceBetweenRedBallAndBigBall >= distanceBetweenBallAndPaddle)

...in the code, the returned value of the 'updateBallCollusion' function can be used to detect a collision. Thus, replacing the if condition with the new function's return value.

There two possible ways of achieving this:

boolean myBoolean = updateBallCollusion(mSmileyBallX, mSmileyBallY);

if (myBoolean)

if(updateBallCollusion(mSmileyBallX, mSmileyBallY))

Both take the returned value (true or false) of the 'updateBallCollusion' function, and use it in the if statement.

Here is the enhanced 'updateBallCollusion' function being used in the updated game's section (this was identified at the beginning of the section 'Using return values in functions').

```
protected void updateGame(float secondsElapsed)

{
  //If the ball moves down on the screen
  if(mBallSpeedY > 0) {
       //Check for a paddle collision
        updateBallCollusion(mPaddleX, mCanvasHeight);
  }
  //Move the ball's X and Y using the speed (pixel/sec)
  mBallX = mBallX + secondsElapsed * mBallSpeedX;
  mBallY = mBallY + secondsElapsed * mBallSpeedY;

  //Check if the ball hits either the left side or the right side of the screen
  //But only do something if the ball is moving towards that side of the screen
  //If it does that => change the direction of the ball in the X direction
  if((mBallX <= mBall.getWidth() / 2 && mBallSpeedX < 0) || (mBallX >= mCanvasWidth -
mBall.getWidth() / 2 && mBallSpeedX > 0) )
  {
       mBallSpeedX = -mBallSpeedX;
  }

       //Check for SmileyBall collision
       if(updateBallCollusion(mSmileyBallX, mSmileyBallY))
       {
          //Increase score
          updateScore(1);
        }

        //Loop through all SadBalls
        for(int i = 0; i < mSadBallX.length; i++)
       {
             /*Perform collisions (if necessary) between SadBall in position i and
the red ball*/
             updateBallCollusion(mSadBallX[i], mSadBallY[i]);
        }
```

```
        /*If the ball goes out of the top of the screen and moves towards the top of
the screen => */
        //change the direction of the ball in the Y direction
        if(mBallY <= mBall.getWidth() / 2 && mBallSpeedY < 0)
        {
                mBallSpeedY = -mBallSpeedY;
        }

        //If the ball goes out of the bottom of the screen => lose the game
        if(mBallY >= mCanvasHeight) {
                setState(GameThread.STATE_LOSE);
        }
}
```

Again, the debugger can be used to see the execution of the sequence.

You should now have a good understanding of how repeating code blocks in the name can be removed by using functions.