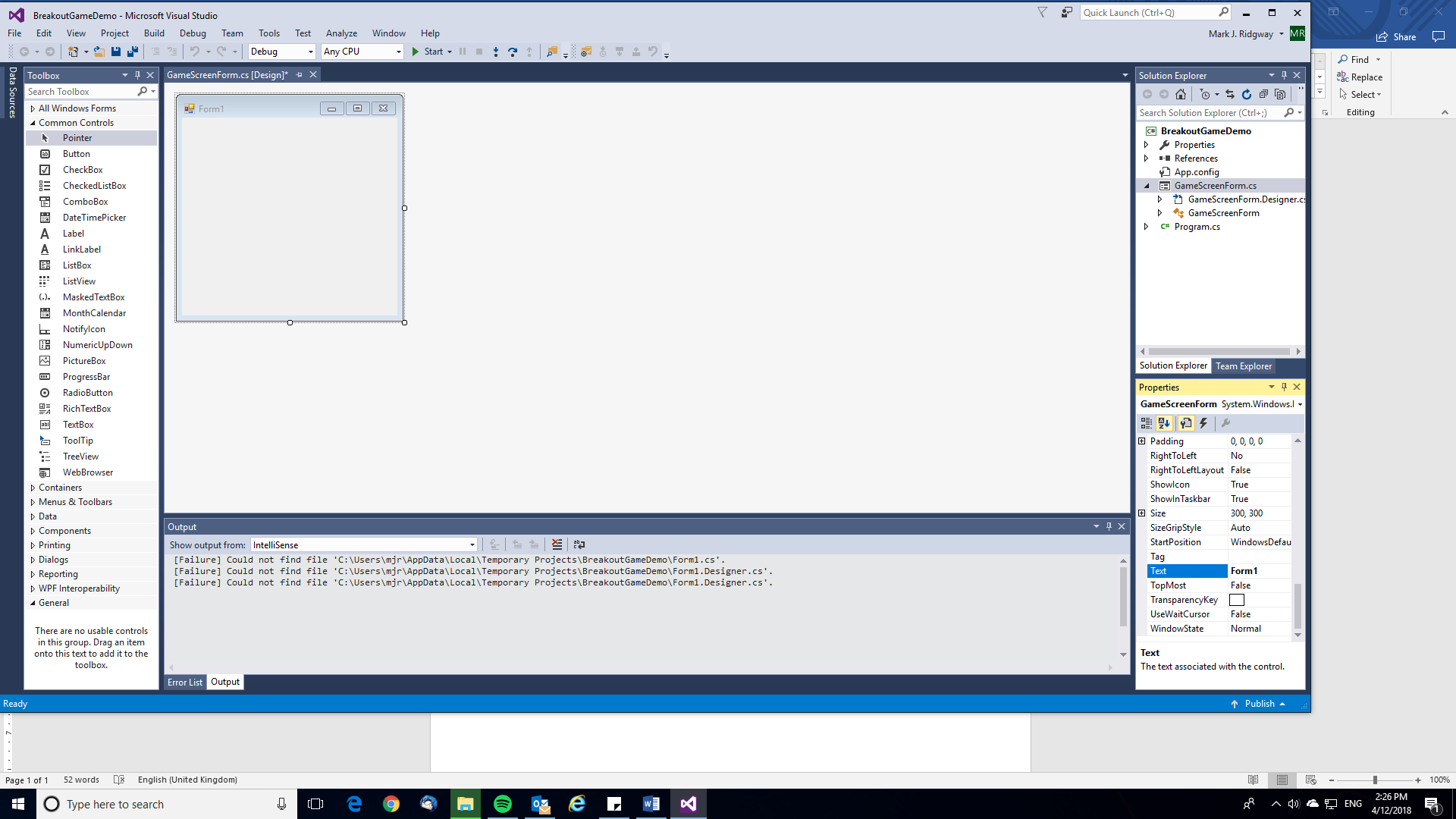
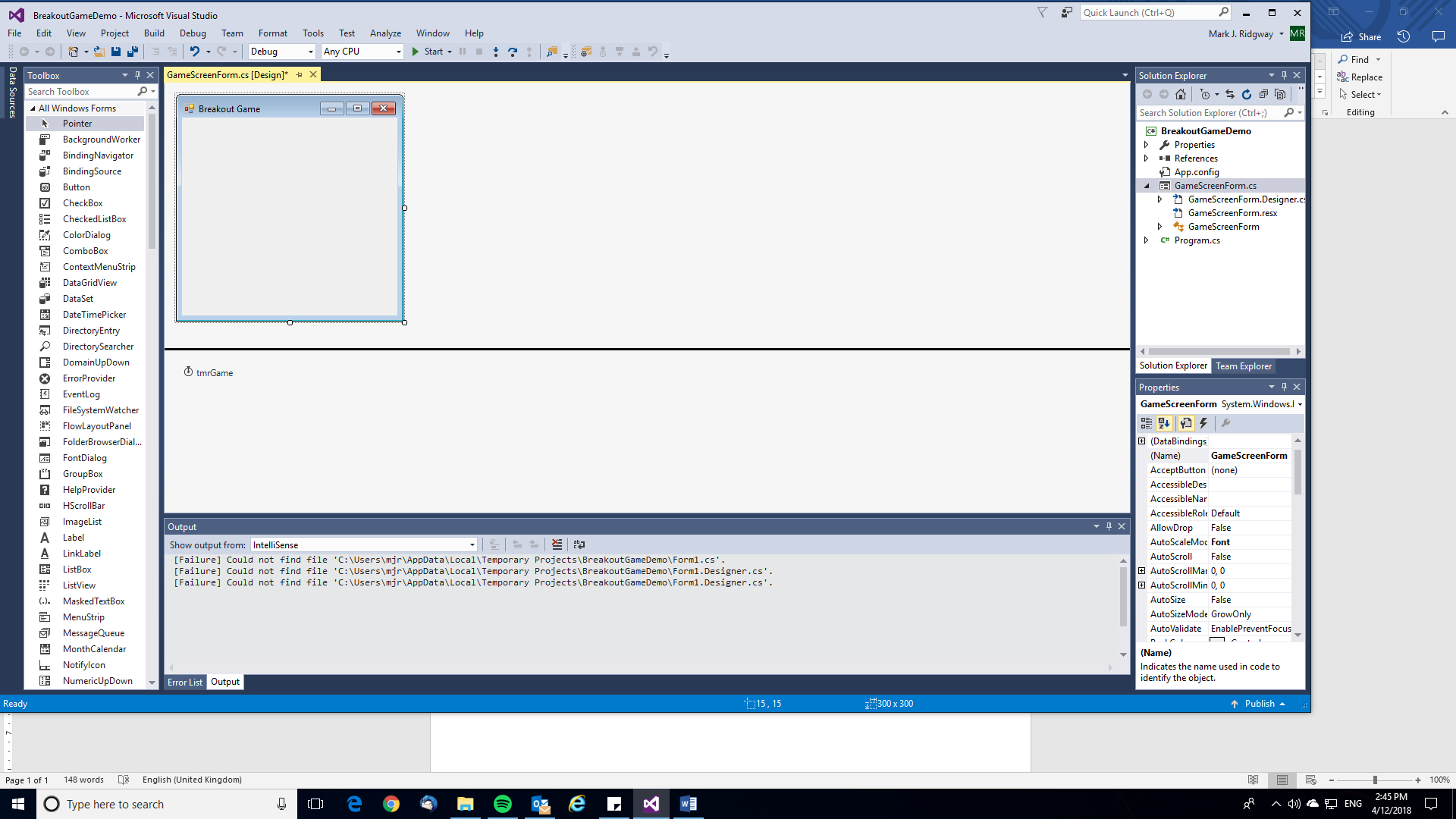
# Programming the Game Breakout (using Object Oriented Programming techniques)

## Setting up a new Project, Form and choosing meaningful names for forms/controls

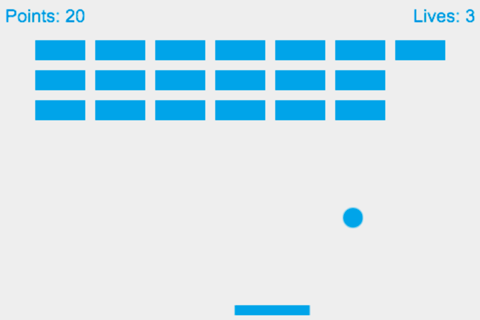
1. Create a new Visual C# Windows Form Application (remember to give it a Name before clicking OK – BreakoutGameDemo will do)

2. In the Solution Explorer on the right hand side of the screen right click Form1.cs and rename it – GameScreenForm.cs seems sensible.

3. You can see in the picture on the right the form’s Text property still contains “Form1”. Click inside the Form then on the right hand side in the Properties window locate the “Text” property and change it from “Form1” to “Breakout Game”

4. Add a Timer to the form from the Toolbox on the right. In the Toolbox click “All Windows Forms” then type in Timer. Click and drag the Timer control onto your form. Now click on the timer which will be called timer1 and change it’s name in the Properties window to “tmrGame”

## Creating the Classes for the various objects involved in the game



Paddle

Ball

Bricks

Bricks

You can see that we need a lot of bricks, one ball and one paddle. Depending on how you want to develop the game perhaps you will want more balls or paddles in future!

It makes sense to create a single class that can be used to create all the brick objects from. Remember a class is a template for a set of objects. As the bricks share a lot of the same properties as the other game objects (ball and paddle) it makes sense to make the class more generic – I’m going to call the class GameObject.

Bricks are static and the paddle and ball needs to be able to move. Instead of duplicating all the GameObject code and adding the extra I’m going to demonstrate use of inheritance to extend the GameObject class to have all the properties and methods a GameObject has plus the additional properties and methods that moveable game objects (like the ball and paddle) will need. I’ll call this derived class MoveableGameObject.

GameObject

MoveableGameObject

## Now to create the first class GameObject.

1. In the solution explorer window (on the right), right-click in the white space at the bottom, click Add, Class… **Name this class GameObject**

2. Edit the code of this new class GameObject. First of all we’ll need to add the System.Drawing namespace to allow us to access the relevant drawing library routines. See edited code below:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Drawing;

3. Next add the properties the class will require. These need to be set with the access modifier **protected** (rather than private) because later we’ll be extending this class using inheritance. The properties are basically the data that is being stored in the objects that will be created from this class. A GameObject will require a Boolean to show whether it is visible or not, a Rectangle which will contain both the position on the screen and the width and height and the Brush colour that will determine what colour it should be when painted on the screen.

namespace BreakoutGameDemo

{

public class GameObject

{

protected bool blnIsVisible;

protected Rectangle recPosition;

protected System.Drawing.Brush drawingBrushColour;

4. Next we will create a Constructor method that will be called automatically when an object of this class is instantiated (created). The purpose of the constructor is to ensure the object that is created is correctly initialised (e.g. in this case assigning data to the objects three properties). Notice the code required for a constructor (this can be examined and you will be introduced to typical pseudocode for this although C# code should be acceptable in the exam too).

namespace BreakoutGameDemo

{

public class GameObject

{

protected bool blnIsVisible;

protected Rectangle recPosition;

protected System.Drawing.Brush drawingBrushColour;

public GameObject(int x, int y,

int width, int height,

bool blnIsVisible, Brush drawingBrushColour)

{

this.blnIsVisible = blnIsVisible;

recPosition = new Rectangle(x, y, width, height);

this.drawingBrushColour = drawingBrushColour;

}

The first line public GameObject…. Indicates it is a constructor method as it has the same identifier as the name of the class. Inside the brackets there are parameters listed and when an object is instantiated values for these will have to be provided. Inside the constructor method the three properties are initialised. Notice use of the keyword **this.** This is required to show that we are referring to the name of the property rather than the name of the parameter (which has an identical name). this.blnIsVisible – refers to the property blnIsVisible whereas blnIsVisible refers to the parameter which acts as a local variable. recPosition doesn’t have a parameter of the same name so there is no need to use the keyword **this** on this line. Rectangle is the name of a class that exists in the namespace System.Drawing. This is a class provided with C# that we will use to help us represent the 2D game objects position and dimensions on the screen. As Rectangle is a class in order to use it we have to create a new object instance and assign it to our property recPosition.

recPosition = new Rectangle(x, y, width, height); The keyword **new** indicates that we are created a new object instance of the class Rectangle. We are providing Rectangle’s constructor method with the initial coordinates (x, y) and the dimensions (width, height) as parameters.

5. Right that’s the hard work for this class completed. Now we just need to code the other methods for the class. These are the **Get’s and Set’s** that allow the properties values to be accessed and changed by other objects and also a method to allow for an intersect to be checked between this GameObject’s Rectangle and another Rectangle.

public bool GetBlnIsVisible()

{

return blnIsVisible;

}

public Rectangle GetRecPosition()

{

return recPosition;

}

public Brush GetDrawingBrushColour()

{

return drawingBrushColour;

}

public void SetBlnIsVisible(bool isVisible)

{

blnIsVisible = isVisible;

}

public void SetRecPosition(Rectangle position)

{

recPosition = position;

}

public void SetDrawingBrushColour(Brush brushColour)

{

drawingBrushColour = brushColour;

}

public void MakeVisible()

{

blnIsVisible = true;

}

public void MakeInvisible()

{

blnIsVisible = false;

}

public bool CheckForIntersect(Rectangle recOther)

{

if (recPosition.IntersectsWith(recOther))

{

return true;

}

else

{

return false;

}

}

That’s it – the GameObject class is now complete.

## Now to create the second class MoveableGameObject which extends the GameObject class using OOP inheritance)

1. In the solution explorer window (on the right), right-click in the white space at the bottom, click Add, Class… **Name this class MoveableGameObject.** Edit the code of this new class MoveableGameObject. As before we’ll need to add the System.Drawing namespace to allow us to access the relevant drawing library routines.

using System.Drawing;

2. To implement inheritance and extend the GameObject class we need to change the line:

public class MoveableGameObject

To

public class MoveableGameObject : GameObject

3. Let’s now write the code for the extra four properties needed by MoveableGameObject which deal with speed, X velocity and Y velocity and whether the object “is glued” or not.

public class MoveableGameObject : GameObject

{

private float fltSpeed;

private float fltXVel;

private float fltYVel;

private bool blnIsGlued;

4. Next we’ll code the constructor method. Now as this is a derived class (one that inherits properties and methods from another) we’ll need to add a line of code in to call on the constructor of the base class (GameObject class) so it initialises the properties that are from GameObject. Then in the constructors code we’ll need to initialise the additional properties present in MoveableGameObject but not present in GameObject.

The full constructor code is given below…

public MoveableGameObject(int x, int y,

int width, int height,

bool blnIsVisible,

float fltSpeed, float fltXVel,

float fltYVel, bool blnIsGlued,

Brush drawingBrushColour)

: base(x, y, width, height, blnIsVisible, drawingBrushColour)

{

this.fltSpeed = fltSpeed;

this.fltXVel = fltXVel;

this.fltYVel = fltYVel;

this.blnIsGlued = blnIsGlued;

}

5. Finally we’ll give the code for the additional Get’s and Set’s and other methods required by MoveableGameObject. Notice how we don’t need to write any code for the methods that are inherited from GameObject. In particular take a look at the code to change the MoveableGameObject’s position on the screen. It allows for either an x and y coordinate to be provided or just an x coordinate and then set’s the Rectangle object to have a new location at a Point specified by the values passed in for x and y or just x along with the existing Y coordinate.

public float GetFltSpeed()

{

return fltSpeed;

}

public float GetfltXVel()

{

return fltXVel;

}

public float GetfltYVel()

{

return fltYVel;

}

public bool GetBlnIsGlued()

{

return blnIsGlued;

}

public void SetFltSpeed(float speed)

{

fltSpeed = speed;

}

public void SetFltXVel(float xVel)

{

fltXVel = xVel;

}

public void SetFltYVel(float yVel)

{

fltYVel = yVel;

}

public void Glue()

{

blnIsGlued = true;

}

public void UnGlue()

{

blnIsGlued = false;

}

public void ChangePosition(int x, int y)

{

recPosition.Location = new Point(x, y);

}

public void ChangePosition(int x)

{

recPosition.Location = new Point(x, recPosition.Y);

}

That’s it – the MoveableGameObject class is now complete.

## Writing the code to implement the game logic and instantiate the GameObject’s and MoveableGameObject’s necessary for the game to run

Finally we need to write the code for the GameScreenForm. This will need to create all the necessary GameObject’s and MoveableGameObject’s, accept keyboard and mouse input as well as contain the necessary game logic so the Form’s application behaves like a game – specifically the Breakout game.

Notice we added a Timer control to the Form at the beginning of this tutorial. Once the game starts this will be enabled and it’s Tick event (which can be set to an interval of our choosing in milliseconds) will be used to repeatedly execute the game logic at that set interval so that collisions between ball and walls, paddle or bricks can be checked, changes to objects made and the screen refreshed (paint event) to show changes (e.g. movement of ball, bricks disappearing after being hit etc...).

1. Right click on GameScreenForm.cs in the Solution Explorer on the right of Visual Studio and click View Code.

You’ll notice some odd code highlighted below **partial class**:

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace BreakoutGameDemo

{

public partial class GameScreenForm : Form

{

public GameScreenForm()

{

InitializeComponent();

}

}

}

C# provides the ability to have a single class coded in multiple .cs files using the partial modifier keyword. This allows C# to hide the automatically generated code for the Form in another file out of sight from the code that the programmer is responsible for. Notice how inheritance is being used again with GameScreenForm being derived from Form (which means it inherits all properties and methods of the Form class). If you want to see this automatically generated code look for GameScreenForm.Designer.cs – you’ll need to expand/explore GameScreenForm.cs in the solution explorer to find this file.

2. Right now for some coding. I’ll start off by declaring the properties for this GameScreenForm class:

private MoveableGameObject ball;

private MoveableGameObject paddle;

private GameObject[,] arrBricks;

This is declaring properties with the following identifiers:

* ball which when instantiated will refer to an object of the MoveableGameObject class.
* paddle which when instantiated will refer to an object of the MoveableGameObject class.
* arrBricks which will refer to a 2D array of GameObject’s (the bricks).

3. There is already a constructor method included which has the following code:

public GameScreenForm()

{

InitializeComponent();

}

We’ll need to add further code to initialise the GameScreenForm. I’ve decided to package this up in a separate procedure and call on it from the constructor. This is due to needing to execute this code to initialise the game again should the user run out of lives and want to have another game rather than exiting.

public GameScreenForm()

{

InitializeComponent();

InitialiseGameScreenForm();

}

Note below that onTick, onKeyDown, onMouseMove, onMouseClick and onPaint won’t be recognised yet as we haven’t coded those procedures which will be executed when the tick, keydown, mousemove, mouseclick and paint events occur. These are known as event handler procedures. This is something that seems complicated at first and unless you use this a lot you’ll need to look this up as it is something you will probably forget.

“An **event handler**, in **C#**, is a method that contains the code that gets executed in response to a specific **event** that occurs in an application. **Event handlers** are used in graphical user interface (GUI) applications to handle **events** such as button clicks and menu selections, raised by controls in the user interface.” <https://www.techopedia.com/definition/3220/event-handler-c>

private void InitialiseGameScreenForm()

{

tmrGame.Tick += new EventHandler(onTick);

this.KeyDown += new KeyEventHandler(onKeyDown);

this.MouseMove += new MouseEventHandler(onMouseMove);

this.MouseClick += new MouseEventHandler(onMouseClick);

this.Paint += new PaintEventHandler(onPaint);

DoubleBuffered = true;

KeyPreview = true;

this.MinimizeBox = false;

this.MaximizeBox = false;

this.BackColor = Color.SkyBlue;

this.FormBorderStyle = FormBorderStyle.FixedDialog;

this.Size = new Size(640, 482);

this.Text = "Breakout Game";

int intBrickRows = 5;

int intBrickColumns = 5;

int intBrickWidth = 80;

int intBrickHeight = 20;

int paddleWidth = 80;

int paddleHeight = 15;

int paddleX = 300;

int paddleY = 434;

int ballWidth = 20;

int ballHeight = 20;

int ballX = paddleX + paddleHeight / 2 - (ballHeight / 2);

int ballY = paddleY - ballWidth;

float defaultSpeed = 20;

float defaultXVel = (float)Math.Cos(defaultSpeed) \* defaultSpeed;

float defaultYVel = (float)Math.Cos(defaultSpeed) \* defaultSpeed;

paddle = new MoveableGameObject(paddleX, paddleY,

paddleWidth, paddleHeight,

true,

defaultSpeed,

0, 0, false, Brushes.Pink);

ball = new MoveableGameObject(ballX, ballY,

ballWidth, ballHeight,

true,

defaultSpeed,

defaultXVel, defaultYVel,

true, Brushes.Yellow);

arrBricks = new GameObject[intBrickRows, intBrickColumns];

tmrGame.Interval = 20;

tmrGame.Start();

int intXOffset = (intBrickWidth / 2) + 1;

int intYOffset = 100;

for (int intRow = 0; intRow <= arrBricks.GetUpperBound(0); intRow++)

{

for (int intColumn = 0;

intColumn <= arrBricks.GetUpperBound(1); intColumn++)

{

arrBricks[intRow, intColumn] =

new GameObject(intXOffset, intYOffset,

intBrickWidth, intBrickHeight,

true, Brushes.Red);

intXOffset += intBrickWidth + 10;

}

intYOffset += intBrickHeight + 10;

intXOffset = (intBrickWidth / 2) + 1;

}

}

4. Now we’ll code the event handlers that will deal with mouse or keyboard input. I’ll paste the code below and write a few comments explaining it.

Firstly let’s deal with the keydown event (which occurs when a key is pressed down on the keyboard):

private void onKeyDown(object sender, KeyEventArgs e)

{

if (e.KeyCode == Keys.Escape)

{

Application.Exit();

}

if (e.KeyCode == Keys.P)

{

if (tmrGame.Enabled)

{

tmrGame.Stop();

}

else

{

tmrGame.Start();

}

}

}

We only want to deal with the Escape and P keys so the code above will do this. Notice how the Escape key will exit the application and the P key will stop or start the timer which has the effect of pausing or continuing the game. The first line of code includes two parameters which need to be included the KeyEventArgs is the object that allows us to work out what key was pressed (stored in the KeyCode property of the KeyEventArgs object. The sender parameter needs to be included but we are not using it in our own code. The sender parameter contains a reference to the control that raised the event (in this case the KeyDown event e.g. perhaps this originated from entry into a textbox control or combobox control etc..).

Now the code for the event handler to deal with the mouseclick event. This code is so at the start of the game when the ball is glued to the paddle a user can left click the mouse to unglue the ball so it launches from the paddle so the game can commence.

private void onMouseClick(object sender, MouseEventArgs e)

{

if (e.Button == MouseButtons.Left)

{

if (ball.getBlnIsGlued() == true)

{

ball.UnGlue();

}

}

}

Next the event handler for the mousemove event. Seems more complicated than it is. If the X coordinate property of the MouseEventArgs object is within the limits of the GameScreenForm X coordinates (and not off the form) then the paddle changes to the position indicated by the X coordinate of the mouse position at the time the event was triggered. Also if the ball is glued then it positions the ball so it remains on top of the paddle (before the game starts you can move the paddle with the ball resting on top to get into your starting position).

private void onMouseMove(object sender, MouseEventArgs e)

{

if (e.X > 0 && e.X < this.Width - 80)

{

paddle.ChangePosition(e.X);

}

if (ball.getBlnIsGlued() == true)

{

ball.ChangePosition(paddle.GetRecPosition().X + 80 / 2 -

(ball.GetRecPosition().Width / 2), 432 - 16);

}

}

5. We will now write a function that will count the visible bricks in arrBricks and return that value.

public int GetBrickCount()

{

int intCount = 0;

foreach (GameObject brick in arrBricks) {

if (brick.GetBlnIsVisible() == true)

{

intCount += 1;

}

}

return intCount;

}

6. Next let’s write the code that will execute when the paint event is triggered. This is the onPaint event handler procedure.

private void onPaint(object sender, PaintEventArgs e)

{

for (int intRow = 0; intRow <= arrBricks.GetUpperBound(0); intRow++)

{

for (int intColumn = 0;

intColumn <= arrBricks.GetUpperBound(1); intColumn++)

{

if (arrBricks[intRow,intColumn].GetBlnIsVisible() == true)

{

e.Graphics.FillRectangle(

arrBricks[intRow, intColumn].GetDrawingBrushColour(),

arrBricks[intRow, intColumn].GetRecPosition());

}

}

}

e.Graphics.FillRectangle(ball.GetDrawingBrushColour(),

ball.GetRecPosition());

e.Graphics.FillRectangle(paddle.GetDrawingBrushColour(),

paddle.GetRecPosition());

}

The code above uses two nested for loops to enable every element of the 2D array to be visited. Each array element contains a GameObject object representing an individual brick. Simply if the object’s blnIsVisible property is set to True then that brick should be displayed on screen so will need to be painted. We do that by using the code:

e.Graphics.FillRectangle(arrBricks[intRow, intColumn].GetDrawingBrushColour(), arrBricks[intRow, intColumn].GetRecPosition());

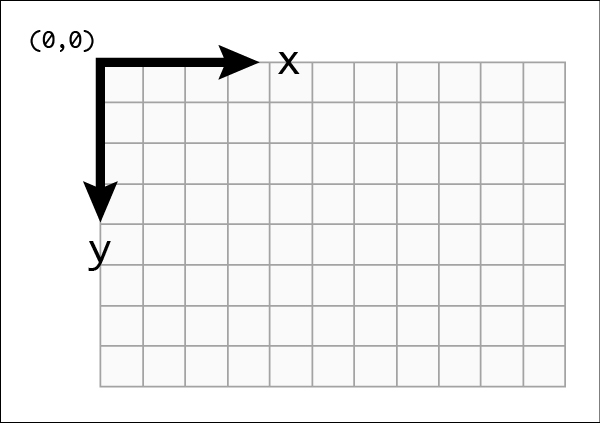
This code gets the brush colour and the rectangle from the corresponding location in arrBricks and then uses these values in a call to the FillRectangle method. FillRectangle will fill/colour a rectangle on the screen.

Notice the code at the end of the onPaint procedure which will paint the ball and paddle on the screen.

7. Right finally now for the last procedure. The event handler for the tick event. This procedure will contain all the game logic and the result of enabling the tmrGame control will have the effect of starting the “game loop” causing this code to be executed at set intervals (20 milliseconds).

This is by far the longest procedure so far. We could have divided this up further but that is down to personal preference.

Before I explain more be aware of what coordinates actually mean. Notice how the origin 0,0 is the top left of the screen.



A summary of the code is as follows:

* if the ball **is not** glued then change it’s position on the screen so that it appears to move
* Check if the ball has gone above the top of the screen (if the ball’s Y coordinate is less than the Y coordinate at the top of the screen (0)). If so then re-position ball on edge then bounce ball by flipping the ball’s Y velocity.
* Check the end condition if the ball falls through the bottom (obviously missing paddle). Note: the ball needs to be Glued so it no longer moves. Also the ball is placed back on the paddle ready to be launched again.
* Check if the ball has gone off the left of the screen and if so re-position on edge and flip the ball’s X velocity.
* Check if the ball has gone off the right of the screen and if so re-position on edge and flip the ball’s X velocity.
* Check if the ball’s rectangle has intersected with the paddle’s rectangle. If so re-position ball on top of paddle and then flip the ball’s Y velocity.
* Go through each element of the 2D array and for visible bricks check whether there has been an intersect between the ball’s rectangle and the brick’s rectangle. If an intersect occurs, then make that brick invisible and flip the ball’s X velocity or Y velocity (depending on a condition).
* Check the end condition of the game if the number of visible bricks is 0. If so stop the timer, show the mouse cursor and display a message box asking user if they want to play again. If they click Yes then execute the InitialiseGameScreenForm procedure. Otherwise they have clicked No so exit the application (which closes the form).
* The final line of code in the procedure is the most crucial.

this.Refresh();

Without this code the screen will not be painted and therefore changes to the position of the ball and visibility of the hit bricks will not be displayed to the user. this.Refresh() triggers the Paint event and causes the onPaint event handler procedure to execute which causes the game objects to be drawn on the screen accordingly.

Here’s the code:

private void onTick(object sender, EventArgs e)

{

if (!ball.GetBlnIsGlued())

{

ball.ChangePosition((int)

(ball.GetRecPosition().X + ball.GetfltXVel()),

(int)(ball.GetRecPosition().Y + ball.GetfltYVel()));

}

if(ball.GetRecPosition().Y < 0)

{

ball.ChangePosition(ball.GetRecPosition().X, 0);

ball.SetFltYVel(-ball.GetfltYVel());

}

if (ball.GetRecPosition().Y - ball.GetRecPosition().Height > this.Height)

{

ball.Glue();

ball.ChangePosition(

paddle.GetRecPosition().X + 72 / 2 –

(ball.GetRecPosition().Width / 2), 432 - 16);

}

if (ball.GetRecPosition().X < 0)

{

ball.ChangePosition(0,ball.GetRecPosition().Y);

ball.SetFltXVel(-ball.GetfltXVel());

}

if (ball.GetRecPosition().X + ball.GetRecPosition().Width > this.Width)

{

ball.ChangePosition(this.Width - ball.GetRecPosition().Width,

ball.GetRecPosition().Y);

ball.SetFltXVel(-ball.GetfltXVel());

}

if (ball.CheckForIntersect(paddle.GetRecPosition()))

{

ball.ChangePosition(ball.GetRecPosition().X,

ball.GetRecPosition().Y –

ball.GetRecPosition().Height);

ball.SetFltYVel(-ball.GetfltYVel());

}

for (int intRow = 0;

intRow <= arrBricks.GetUpperBound(0); intRow++)

{

for (int intColumn = 0;

intColumn <= arrBricks.GetUpperBound(1); intColumn++)

{

if (arrBricks[intRow, intColumn].GetBlnIsVisible())

{

if (ball.CheckForIntersect(

arrBricks[intRow, intColumn].GetRecPosition()))

{

arrBricks[intRow, intColumn].MakeInvisible();

if (ball.GetRecPosition().X + 10 <

arrBricks[intRow, intColumn].GetRecPosition().X ||

ball.GetRecPosition().X >

arrBricks[intRow, intColumn].GetRecPosition().X +

arrBricks[intRow, intColumn].GetRecPosition().Width)

{

ball.SetFltXVel(-ball.GetfltXVel());

}

else

{

ball.SetFltYVel(-ball.GetfltYVel());

}

}

}

}

}

if(GetBrickCount()==0)

{

tmrGame.Stop();

Cursor.Show();

if(MessageBox.Show("Would you like to play again?", "Play Again?",

MessageBoxButtons.YesNo,

MessageBoxIcon.Question) == DialogResult.Yes)

{

InitialiseBreakOutGameScreen();

}

else

{

Application.Exit();

}

}

this.Refresh();

}

Right that’s it you should now have a working Breakout game running in C# coded using OOP techniques including inheritance. A complete code listing is included below and a C# project containing this solution is available from your teacher (after you have attempted to get this coded yourself by following this guide).

Now spend some time turning this into an actual game:

* Introduce some kind of scoring system (1 point per brick hit?)
* Limit the number of lives available in a game e.g. to 3 lives (currently it is unlimited)
* Add some sound effects or retro music to make the game more interesting (feel free to checkout ideas for effects on youtube – plenty of Breakout game videos!)
* Change the speed of the ball (e.g. as you collect more points or by introducing a difficulty option that has an effect on the speed of the ball).
* Think of your own extensions/ideas and implement them!

Final Complete Code Listing

GameObject.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Drawing;

namespace BreakoutGameDemo

{

public class GameObject

{

protected bool blnIsVisible;

protected Rectangle recPosition;

protected System.Drawing.Brush drawingBrushColour;

public GameObject(int x, int y, int width,

int height, bool blnIsVisible,

Brush drawingBrushColour)

{

this.blnIsVisible = blnIsVisible;

recPosition = new Rectangle(x, y, width, height);

this.drawingBrushColour = drawingBrushColour;

}

public bool GetBlnIsVisible()

{

return blnIsVisible;

}

public Rectangle GetRecPosition()

{

return recPosition;

}

public Brush GetDrawingBrushColour()

{

return drawingBrushColour;

}

public void SetBlnIsVisible(bool isVisible)

{

blnIsVisible = isVisible;

}

public void SetRecPosition(Rectangle position)

{

recPosition = position;

}

public void SetDrawingBrushColour(Brush brushColour)

{

drawingBrushColour = brushColour;

}

public void MakeVisible()

{

blnIsVisible = true;

}

public void MakeInvisible()

{

blnIsVisible = false;

}

public bool CheckForIntersect(Rectangle recOther)

{

if (recPosition.IntersectsWith(recOther))

{

return true;

}

else

{

return false;

}

}

}

}

MoveableGameObject.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Drawing;

namespace BreakoutGameDemo

{

public class MoveableGameObject : GameObject

{

private float fltSpeed;

private float fltXVel;

private float fltYVel;

private bool blnIsGlued;

public MoveableGameObject(int x, int y,

int width, int height,

bool blnIsVisible,

float fltSpeed, float fltXVel,

float fltYVel, bool blnIsGlued,

Brush drawingBrushColour)

: base(x, y, width, height, blnIsVisible, drawingBrushColour)

{

this.fltSpeed = fltSpeed;

this.fltXVel = fltXVel;

this.fltYVel = fltYVel;

this.blnIsGlued = blnIsGlued;

}

public float GetFltSpeed()

{

return fltSpeed;

}

public float GetfltXVel()

{

return fltXVel;

}

public float GetfltYVel()

{

return fltYVel;

}

public bool GetBlnIsGlued()

{

return blnIsGlued;

}

public void SetFltSpeed(float speed)

{

fltSpeed = speed;

}

public void SetFltXVel(float xVel)

{

fltXVel = xVel;

}

public void SetFltYVel(float yVel)

{

fltYVel = yVel;

}

public void Glue()

{

blnIsGlued = true;

}

public void UnGlue()

{

blnIsGlued = false;

}

public void ChangePosition(int x, int y)

{

recPosition.Location = new Point(x, y);

}

public void ChangePosition(int x)

{

recPosition.Location = new Point(x, recPosition.Y);

}

}

}

GameScreenForm.cs

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace BreakoutGameDemo

{

public partial class GameScreenForm : Form

{

private MoveableGameObject ball;

private MoveableGameObject paddle;

private GameObject[,] arrBricks;

public GameScreenForm()

{

InitializeComponent();

InitialiseGameScreenForm();

}

private void InitialiseGameScreenForm()

{

tmrGame.Tick += new EventHandler(onTick);

this.KeyDown += new KeyEventHandler(onKeyDown);

this.MouseMove += new MouseEventHandler(onMouseMove);

this.MouseClick += new MouseEventHandler(onMouseClick);

this.Paint += new PaintEventHandler(onPaint);

DoubleBuffered = true;

KeyPreview = true;

this.MinimizeBox = false;

this.MaximizeBox = false;

this.BackColor = Color.SkyBlue;

this.FormBorderStyle = FormBorderStyle.FixedDialog;

this.Size = new Size(640, 482);

this.Text = "Breakout Game";

int intBrickRows = 5;

int intBrickColumns = 5;

int intBrickWidth = 80;

int intBrickHeight = 20;

int paddleWidth = 80;

int paddleHeight = 15;

int paddleX = 300;

int paddleY = 434;

int ballWidth = 20;

int ballHeight = 20;

int ballX = paddleX + paddleHeight / 2 - (ballHeight / 2);

int ballY = paddleY - ballWidth;

float defaultSpeed = 20;

float defaultXVel = (float)Math.Cos(defaultSpeed) \* defaultSpeed;

float defaultYVel = (float)Math.Cos(defaultSpeed) \* defaultSpeed;

paddle = new MoveableGameObject(paddleX, paddleY,

paddleWidth, paddleHeight,

true, defaultSpeed,

0, 0, false, Brushes.Pink);

ball = new MoveableGameObject(ballX, ballY,

ballWidth, ballHeight,

true,

defaultSpeed,

defaultXVel, defaultYVel,

true, Brushes.Yellow);

arrBricks = new GameObject[intBrickRows, intBrickColumns];

tmrGame.Interval = 20;

tmrGame.Start();

int intXOffset = (intBrickWidth / 2) + 1;

int intYOffset = 100;

for (int intRow = 0;

intRow <= arrBricks.GetUpperBound(0); intRow++)

{

for (int intColumn = 0;

intColumn <= arrBricks.GetUpperBound(1); intColumn++)

{

arrBricks[intRow, intColumn] = new

GameObject(intXOffset, intYOffset,

intBrickWidth, intBrickHeight,

true, Brushes.Red);

intXOffset += intBrickWidth + 10;

}

intYOffset += intBrickHeight + 10;

intXOffset = (intBrickWidth / 2) + 1;

}

}

private void onKeyDown(object sender, KeyEventArgs e)

{

if (e.KeyCode == Keys.Escape)

{

Application.Exit();

}

if (e.KeyCode == Keys.P)

{

if (tmrGame.Enabled)

{

tmrGame.Stop();

}

else

{

tmrGame.Start();

}

}

}

private void onMouseClick(object sender, MouseEventArgs e)

{

if (e.Button == MouseButtons.Left)

{

if (ball.GetBlnIsGlued() == true)

{

ball.UnGlue();

}

}

}

private void onMouseMove(object sender, MouseEventArgs e)

{

if (e.X > 0 && e.X < this.Width - 80)

{

paddle.ChangePosition(e.X);

}

if (ball.GetBlnIsGlued() == true)

{

ball.ChangePosition(

paddle.GetRecPosition().X + 80 / 2 - (

ball.GetRecPosition().Width / 2), 432 - 16);

}

}

public int GetBrickCount()

{

int intCount = 0;

foreach (GameObject brick in arrBricks)

{

if (brick.GetBlnIsVisible() == true)

{

intCount += 1;

}

}

return intCount;

}

private void onPaint(object sender, PaintEventArgs e)

{

for (int intRow = 0; intRow <= arrBricks.GetUpperBound(0); intRow++)

{

for (int intColumn = 0;

intColumn <= arrBricks.GetUpperBound(1); intColumn++)

{

if (arrBricks[intRow, intColumn].GetBlnIsVisible() == true)

{

e.Graphics.FillRectangle(

arrBricks[intRow, intColumn].GetDrawingBrushColour(),

arrBricks[intRow, intColumn].GetRecPosition());

}

}

}

e.Graphics.FillRectangle(ball.GetDrawingBrushColour(),

ball.GetRecPosition());

e.Graphics.FillRectangle(paddle.GetDrawingBrushColour(),

paddle.GetRecPosition());

}

private void onTick(object sender, EventArgs e)

{

if (!ball.GetBlnIsGlued())

{

ball.ChangePosition((int)(ball.GetRecPosition().X +

ball.GetfltXVel()),

(int)(ball.GetRecPosition().Y +

ball.GetfltYVel()));

}

if (ball.GetRecPosition().Y < 0)

{

ball.ChangePosition(ball.GetRecPosition().X, 0);

ball.SetFltYVel(-ball.GetfltYVel());

}

if (ball.GetRecPosition().Y - ball.GetRecPosition().Height > this.Height)

{

ball.Glue();

ball.ChangePosition(paddle.GetRecPosition().X + 72 / 2 –

(ball.GetRecPosition().Width / 2), 432 - 16);

}

if (ball.GetRecPosition().X < 0)

{

ball.ChangePosition(0, ball.GetRecPosition().Y);

ball.SetFltXVel(-ball.GetfltXVel());

}

if (ball.GetRecPosition().X + ball.GetRecPosition().Width > this.Width)

{

ball.ChangePosition(this.Width - ball.GetRecPosition().Width,

ball.GetRecPosition().Y);

ball.SetFltXVel(-ball.GetfltXVel());

}

if (ball.CheckForIntersect(paddle.GetRecPosition()))

{

ball.ChangePosition(ball.GetRecPosition().X,

ball.GetRecPosition().Y –

ball.GetRecPosition().Height);

ball.SetFltYVel(-ball.GetfltYVel());

}

for (int intRow = 0; intRow <= arrBricks.GetUpperBound(0); intRow++)

{

for (int intColumn = 0;

intColumn <= arrBricks.GetUpperBound(1); intColumn++)

{

if (arrBricks[intRow, intColumn].GetBlnIsVisible())

{

if (ball.CheckForIntersect(arrBricks[intRow,

intColumn].GetRecPosition()))

{

arrBricks[intRow, intColumn].MakeInvisible();

if (ball.GetRecPosition().X + 10 < arrBricks[intRow,

intColumn].GetRecPosition().X ||

ball.GetRecPosition().X > arrBricks[intRow,

intColumn].GetRecPosition().X + arrBricks[intRow,

intColumn].GetRecPosition().Width)

{

ball.SetFltXVel(-ball.GetfltXVel());

}

else

{

ball.SetFltYVel(-ball.GetfltYVel());

}

}

}

}

}

if (GetBrickCount() == 0)

{

tmrGame.Stop();

Cursor.Show();

if (MessageBox.Show("Would you like to play again?", "Play Again?",

MessageBoxButtons.YesNo,

MessageBoxIcon.Question) == DialogResult.Yes)

{

InitialiseGameScreenForm();

}

else

{

Application.Exit();

}

}

this.Refresh();

}

}

}