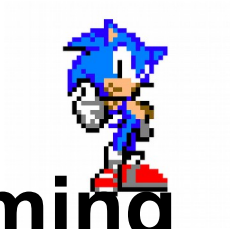
Introduction to Game Programming



Session 5 – Using functions to arrange our code

Nacho Iborra

IES San Vicente

Index of contents

1. Introduction

Now it's time to have a look at our code. At this moment, our *Main* function is quite big, and it needs a "refactoring". We must divide the code into functions so that each function does one thing.

In this session, we will divide all the work that we have done into the corresponding functions: print borders, print texts, bricks... Each function will have some restrictions (return type, argument types...) that we need to follow. We will define each function, adapt the existing code to fit this function, and then replace the original code in the *Main* function with this new function.

1.1. Previous step with the structs

As we are going to use *struct* variables inside some functions, the compiler may complain that the visibility of these structs is more restrictive than the functions that use them. In other words, structs are not public, whereas functions are. So, to solve this problem, add the *public* modifier before the *structs* definition:

```
public struct brick
{
    ...

public struct ball
{
    ...
```

2. Functions to draw the static part

Firstly, we are going to add some basic functions to draw all the static things that we placed before the game loop:

- Right border
- Game title, score and lives texts
- Bricks initialization and drawing
- Ball initialization

2.1. Printing the right border

Create a function called `PrintBorder` with no parameters nor return type. Put inside this function all the lines that draw the grey border of the right half:

```
public static void PrintBorder()
{
    Console.BackgroundColor = ConsoleColor.Gray;

    for (int i = 40; i <  79; i++)
    {
        Console.SetCursorPosition(i, 0);
        Console.Write(" ");
        Console.SetCursorPosition(i, 23);
        Console.Write(" ");
    }
    for (int i = 0; i < 24; i++)
    {
        Console.SetCursorPosition(40, i);
        Console.Write(" ");
        Console.SetCursorPosition(79, i);
        Console.Write(" ");
    }

    Console.ResetColor();
}
```

and then, call this function from the Main function, instead of all the lines of code that you have placed inside it:

```
public static void Main()
{
    …
    // Print right panel

    PrintBorder();
    …
```

2.2. Printing right texts

Create now another function called `PrintTexts` with no parameters nor return type, and do the same with the texts written inside the right half:

```
public static void PrintTexts()
{
    Console.ForegroundColor = ConsoleColor.Red;
```

```
    Console.SetCursorPosition(56, 4);
    Console.Write("ARKANOID");

    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.SetCursorPosition(42, 7);
    Console.Write("LIVES");

    Console.ForegroundColor = ConsoleColor.Green;
    Console.SetCursorPosition(42, 12);
    Console.Write("POINTS");

    Console.ResetColor();
}

public static void Main()
{
    …
    // Print right panel

    PrintBorder();
    PrintTexts();
```

2.3. Initializing bricks

Now we are going to define a function called `CreateBricks`. It will have no parameters, and it will return an array of brick structures, with the brick array already created.

```
public static brick[] CreateBricks()
{
    brick[] bricks = new brick[40];

    for (int i = 0; i < bricks.Length; i++)
    {
        bricks[i].x = (byte)((i % 10)*4);
        bricks[i].y = (byte)(2 + (i/10));
        bricks[i].destroyed = false;
        if (i >= 14 && i <= 15)
        {
            bricks[i].numberOfHits = 2;
            bricks[i].color = ConsoleColor.Yellow;
            bricks[i].score = 50;
        } else if (i%2 == 0 && (i/10)%2 == 0 || i%2 == 1 && (i/10)%2 == 1) {
            bricks[i].numberOfHits = 1;
            bricks[i].color = ConsoleColor.Blue;
            bricks[i].score = 20;
        } else {
            bricks[i].numberOfHits = 1;
            bricks[i].color = ConsoleColor.Red;
            bricks[i].score = 10;
        }
    }
    return bricks;
}
```

In the Main function, we just define a variable called *bricks*, and assign it to the result of this new function:

```
// Bricks
brick[] bricks = CreateBricks();
```

2.4. Drawing bricks

Define another function called DrawBricks. It will get a brick array as a parameter, and it will have no return type. Inside this function, we will explore and draw the brick array.

```
public static void DrawBricks(brick[] bricks)
{
    for (int i = 0; i < bricks.Length; i++)
    {
        Console.SetCursorPosition(bricks[i].x, bricks[i].y);
        Console.BackgroundColor = bricks[i].color;
        Console.Write("    ");
    }
}
```

Remember to call this function in the place where you had the corresponding lines of code before.

2.5. Initializing the ball

To finish with this subsection, we are going to define a new function call InitBall. It will have no parameters, and return a *ball* structure with the corresponding initial values:

```
public static ball InitBall()
{
    ball gameBall;

    gameBall.x = 20;
    gameBall.y = 22;
    gameBall.xDirection = 0;
    gameBall.xAngle = 0;
    gameBall.yDirection = -1;

    return gameBall;
}
```

In the Main function, just define a *ball* variable, and assign it to the result of this function:

```
ball gameBall = InitBall();
```

a

3. Get user input and move the ball

In this subsection we are going to define two functions to get the user input (and modify character X coordinate if necessary), and move the ball in each loop iteration.

3.1. Getting user input

We'll start by creating a function called `GetUserInput`. It will have two *ref* parameters: the character X coordinate, and the exitGame flag. Inside this function, and depending on the key pressed by the player, we can either change the X coordinate or set the *exitGame* flag to *true*.

```
public static void GetUserInput (ref int characterX, int characterY,
                                 ref bool exitGame)
{
    ConsoleKeyInfo key;

    Console.SetCursorPosition(0, 24);

    if (Console.KeyAvailable)
    {
        do
        {
            key = Console.ReadKey(false);
        } while (Console.KeyAvailable);

        Console.SetCursorPosition(characterX, characterY);
        Console.ResetColor();
        Console.Write("     ");

        if (key.Key == ConsoleKey.LeftArrow && characterX > 0)
            characterX--;
        if (key.Key == ConsoleKey.RightArrow && characterX + BAR_WIDTH < 40)
            characterX++;
        if (key.Key == ConsoleKey.Escape)
            exitGame = true;
    }
}
```

Besides, note that the variable *key* will no longer be needed from the Main function, since we only need it for this piece of code. So, we can declare it inside this function. You may also need to define constan BAR_WIDTH globally:

```
public class MyGame
{
    public struct brick
    {
        ...
    }

    public const int BAR_WIDTH = 5;
```

Finally, remember to call `GetUserInput` from step #2 of the game loop:

```
// 2 Read input and calculate player's new position

GetUserInput(ref characterX, characterY, ref exitGame);
```

3.2. Moving the ball

Define now a function called `MoveBall`. It will have one *ref* parameters (ball structure):

```
public static void MoveBall(ref ball gameBall)
{
    Console.ResetColor();
    Console.SetCursorPosition(gameBall.x, gameBall.y);
    Console.Write(" ");

    if (gameBall.xDirection < 0 && gameBall.xAngle == -2 && gameBall.x < 2)
        gameBall.x -= 1;
    else if (gameBall.xDirection > 0 && gameBall.xAngle == 2 && gameBall.x > 37)
        gameBall.x += 1;
    else
        gameBall.x += (byte) gameBall.xAngle;

    gameBall.y += (byte) gameBall.yDirection;
}
```

Use this new function in step #3 of the game loop:

```
// 3 Move enemies and other objects

if (moveBall)
{
    MoveBall(ref gameBall);
}
```

4. Checking collisions

In this section we are going to define functions to check all the possible types of collisions:

- Collision between the ball and the bar
- Collision between the ball and the bricks
- Collision between the ball and the scene boundaries

4.1. Collision between the ball and the bar

Define a function called `CollisionBallBar` with three parameters: the ball (*ball* structure, as *ref*), and the bar X and Y coordinates:

```
public static void CollisionBallBar(ref ball gameBall, int characterX, int
characterY)
{
    if (gameBall.y == 22 && gameBall.x >= characterX &&
        gameBall.x <= characterX + BAR_WIDTH)
    {
        gameBall.yDirection = (sbyte)-gameBall.yDirection;
        switch (gameBall.x - characterX)
        {
            case 0:
                    gameBall.xDirection = -1;
                    gameBall.xAngle = -2;
                    break;
            case 1:
                    gameBall.xDirection = -1;
                    gameBall.xAngle = -1;
                    break;
            case 2:
                    gameBall.xDirection = 0;
                    gameBall.xAngle = 0;
                    break;
            case 3:
                    gameBall.xDirection = 1;
                    gameBall.xAngle = 1;
                    break;
            case 4:
                    gameBall.xDirection = 1;
                    gameBall.xAngle = 2;
                    break;
        }
    }
}
```

4.2. Collision between the ball and the bricks

This function will be called `CollisionBallBricks` and it will have three parameters: the *ball* variable (as *ref*), the *bricks* array and the total score (also as *ref*)

```
public static void CollisionBallBricks(ref ball gameBall, brick[] bricks, ref
int totalScore)
{
    for (int i = 0; i < bricks.Length; i++)
    {
```

```
        if (!bricks[i].destroyed)
        {
            // Lower and upper border
            if ((gameBall.y == bricks[i].y + 1 && gameBall.yDirection < 0 ||
                 gameBall.y == bricks[i].y - 1 && gameBall.yDirection > 0) &&
                 gameBall.x >= bricks[i].x && gameBall.x < bricks[i].x + 4)
            {
                bricks[i].numberOfHits--;
                gameBall.yDirection = (sbyte)-gameBall.yDirection;
            }
            // Left and right border
            if ((gameBall.x == bricks[i].x-1 && gameBall.xDirection > 0 ||
                 gameBall.x == bricks[i].x + 4 && gameBall.xDirection < 0) &&
                 gameBall.y == bricks[i].y)
            {
                bricks[i].numberOfHits--;
                gameBall.xDirection = (sbyte)-gameBall.xDirection;
                gameBall.xAngle = (sbyte)-gameBall.xAngle;
            }
            // Check if brick must be destroyed, then destroy the brick
            if (bricks[i].numberOfHits == 0)
            {
                bricks[i].destroyed = true;
                totalScore += bricks[i].score;
                Console.SetCursorPosition(50, 12);
                Console.ForegroundColor = ConsoleColor.White;
                Console.Write(totalScore);
                Console.ResetColor();
                // Remove brick from the scene (erase it)
                Console.BackgroundColor = ConsoleColor.Black;
                Console.SetCursorPosition(bricks[i].x, bricks[i].y);
                Console.Write("    ");
                Console.ResetColor();
            }
        }
    }
}
```

4.3. Collision between the ball and the boundaries

Finally, define a function called `CollisionBallBoundaries`. It will have a single parameter of type *ball* (as *ref*):

```
public static void CollisionBallBoundaries(ref ball gameBall)
{
    // Collision between the ball and left/right bounds
    if (gameBall.x == 0 || gameBall.x == 39)
    {
        gameBall.xDirection = (sbyte)-gameBall.xDirection;
        gameBall.xAngle = (sbyte)-gameBall.xAngle;
    }

    // Collision between the ball and the upper bound
    if (gameBall.y == 0)
    {
        gameBall.yDirection = (sbyte)-gameBall.yDirection;
    }
}
```

4.4. Calling all this stuff from the Main function

Finally, you need to call these functions in step #4 of the game loop:

```
// 4 Check collisions and update game state

if (moveBall)
{
    CollisionBallBar(ref gameBall, characterX, characterY);
    CollisionBallBricks(ref gameBall, bricks, ref totalScore);
    CollisionBallBoundaries(ref gameBall);
}
```

5. To think a little: updating state

To finish with this session, define functions to:

- Print the score whenever it is called. This function will replace the code within `CollisionBallBricks` function where you print the score in the appropriate place.
- Print the number of lives and update lives. At the beginning, we will have 3 lives (you must print them in the right panel). Whenever the ball gets out of reach (Y coordinate greater or equal than character Y coordinate), it must decrease the number of lives, and place the bar and the ball in their initial positions. Besides, if the number of lives is 0, the game must finish. You may need to create functions or code to:
  - Print the number of lives
  - Reset ball and bar positions after losing a life
  - Exit the game if we get 0 lives

You can modify the definition of some function(s) in order to achieve these goals.