Wouter van Toll, Arjan Egges, Jeroen D. Fokker

Learning C# by Programming Games

Second Edition

Exercise Solutions for Part III: Structures and Patterns

August 10, 2019

Springer

Solutions for Part III: Structures and Patterns

Chapter 12

1. *Coordinate conversion*
   Screen coordinates indicate a pixel position on the screen. World coordinates indicate a position in the game world. These two coordinate systems may be the same in simple games, but they can be different if your game needs to be adapted to different screen sizes, or if your game has a camera that can zoom and/or pan.

   Conversion from world to screen coordinates needs to happen when you draw the game world on the screen. Conversion from screen to world coordinates needs to happen when you handle user interaction, such as mouse clicks.

2. *Side-scrolling games*

   a. At the very least, the ScreenToWorld method should add cameraPosition to the final result. After all, this method currently calculates world coordinates *relative* to the camera's top-left corner. And now that this top-left corner can have a different position than $(0, 0)$, you should add it to the final position that you calculate. So, you should turn the final line of ScreenToWorld into this:

   return (screenPosition − viewportTopLeft) ∗ screenToWorldScale + cameraPosition;

   But this version is probably incomplete: it assumes that the world is (still) scaled to fit on the screen. So, whenever the camera is not at $(0, 0)$, part of the screen will show a blank background. Let's move on to question b for the full version.

b. When the camera's view is described by a full Rectangle (let's name it cameraRectangle, one more thing changes. When computing screenToWorldScale, we should take into account that the camera shows only cameraRectangle.Width by cameraRectangle.Height world units on the screen, instead of the full game world. If we assume that the aspect ratio of cameraRectangle is always okay (that is: the game world does not get distorted horizontally or vertically), using Width is enough:

```
Vector2 ScreenToWorld(Vector2 screenPosition)
{
    Vector2 viewportTopLeft =
        new Vector2(GraphicsDevice.Viewport.X, GraphicsDevice.Viewport.Y);

    Vector2 cameraTopLeft = new Vector2(cameraRectangle.X, cameraRectangle.Y);

    float screenToWorldScale =
        cameraRectangle.Width / (float)GraphicsDevice.Viewport.Width;

    return (screenPosition − viewportTopLeft) ∗ screenToWorldScale + cameraTopLeft;
}
```

Chapter 13

1. *Lists*
   After the five insertions, the list contains 5,4,3,2,1.
   After the first removal, the list contains 4,3,2,1 (we've removed the first element).
   After the second removal, the list contains 4,2,1 (we've removed the second element).
   After the third removal, the list contains 4,2 (we've removed the third element).

2. *Methods with arrays*

   a.
   ```
   int CountZeros(int[] arr)
   {
       int counter = 0;
       for (int i=0; i<arr.Length; i++)
           if (arr[i] == 0)
               counter++;
       return counter;
   }
   ```

   b.
   ```
   int[] AddArrays(int[] arr1, int[] arr2)
   {
       int[] res = new int[arr1.Length];
       for (int i=0; i<arr1.Length; i++)
           res[i] = arr1[i] + arr2[i];
       return res;
   }
   ```

3. *Uppercase and lowercase*

   a. Here's a method that checks if all characters in a string are uppercase letters. Note that we can stop as soon as we've found a wrong character!

```
bool IsAllUppercase(string str)
{
    for (int n=0; n<str.Length; n++)
    {
        char c = str[n];
        if (c < 'A' || c > 'Z')
            return false;
    }
    return true;
}
```

   b. To turn an uppercase letter into a lowercase letter, you should 'move it forward' in the ASCII table, by adding the difference between lowercase and uppercase letters. This difference is a fixed number, and you can calculate it as 'a' − 'A'.
   Combined with a loop, we get the following:

```
string ToLowercase(string str)
{
    string result;
    for (int n=0; n<str.Length; n++)
    {
        char c = str[n];
        if (c >= 'A' && c <= 'Z')
            result += (char)(c + ('a' − 'A'));
        else
            result += c;
    }
    return result;
}
```

   Note that a character simply stays the same if it is not an uppercase letter.

4. *More **string** methods*

   a. Here's a version with a **while** loop:

```
int FirstPosition(string s, char c)
{
    int pos = 0;
    while (pos < s.Length)
    {
        if (s[pos] == c)
            return pos;
        pos++;
    }
    return −1;
}
```

   Of course, you can do it with a **for** loop as well:

```
int FirstPosition(string s, char c)
{
    for (int pos=0; pos < s.Length; pos++)
    {
        if (s[pos] == c)
            return pos;
    }
    return −1;
}
```

b.
```
public string Replace(string s, char x, char y)
{
    string result = "";
    for (int pos = 0; pos < s.Length; pos++)
    {
        if (s[pos]==x)
            result += y;
        else
            result += s[pos];
    }
    return result;
}
```

c. Here's our solution. If part is longer than full, then we know the answer immediately. Otherwise, we loop over all characters in part and check if they match the corresponding character in **string**. Make sure to use the right offset value for this!

```
public bool EndsWith(string full, string part)
{
    if (part.Length > full.Length)
        return false;

    int offset = full.Length − part.Length;
    for (int pos=0; n<part.Length; pos++)
    {
        if (part[pos] != full[pos + offset])
            return false;
    }
    return true;
}
```

5. *Collections*

For this question, we use a helper method IndexOf from the List class. This helper method is pretty easy to program yourself, though: it's essentially the FirstPosition method of the previous question, and the IndexOf method that will appear in the next question.

```
void RemoveDuplicates(List<int> list)
{
    for (int i=list.Count−1; i>=0; i−−)
    {
        int pos = list.IndexOf(list[i]);
        if (pos != −1 && pos < i)
            list.RemoveAt(i);
    }
}
```

This version of RemoveDuplicates is not extremely efficient, by the way. Can you see why? And can you think of a nicer version? *Hint:* have a look at our answers for the rest of this chapter.

6. *Searching in arrays*
   In this question, we will always assume that the given array is not empty.

   a.
```
double Largest(double[] a)
{
    double result = a[0];
    for (int t=1; t<a.Length; t++)
        if (a[t] > result)
            result = a[t];
    return result;
}
```

   b.
```
double IndexLargest(double[] a)
{
    int result = 0;
    for (int t=1; t<a.Length; t++)
        if (a[t] > a[result])
            result = t;

    return result;
}
```

   c. This is basically the FirstPosition method that you wrote for strings a few questions ago.

```
double IndexOf(double[] a, double val)
{
    for (int t=0; t<a.Length; t++)
        if (a[t] == val)
            return t;

    return -1;
}
```

   d. This one may be a bit tricky to get right. Keep track of the smallest number *and* how often you've found it. Whenever you find a number that is even smaller, restart your counting.

```
int HowManySmallest(double [] a)
{
    int result = 1;
    double smallest = a[0];
    for (int t=1; t<a.Length; t++)
    {
        if (a[t] < smallest)
        {
            result = 1;
            smallest = a[t];
        }
        else if (a[t]==smallest)
            result++;
    }
    return result;
}
```

a. It's useful to first create a variant of IndexLargest with a second parameter, indicating the array index at which the search should stop:

```
int IndexLargest(double [] a, int n)
{
    int result = 0;
    for (int t=1; t<n; t++)
        if (a[t] > a[result])
            result = t;
    return result;
}
```

With this method, you can implement Sort as follows:

```
void Sort(double [] a)
{
    for (int t=a.Length; t>0; t--)
    {
        int p = indexLargest(a,t);
        double backup = a[t-1];
        a[t-1] = a[p];
        a[p] = backup;
    }
}
```

This way of sorting is called *bubble sort*, because the array elements are shifted ('bubbled') one by one to the right position. There are many other ways to sort an array (including more efficient ways!), but discussing that would go too far for this book.

b. If you follow the question's instructions very closely, you should end up with something like this:

```
int IndexOf(double [] a, double x)
{
    int low = 0;
    int high = a.Length;
    while (high > low)
    {
        int mid = (low+high)/2;
        if (a[mid]==x)
            return mid;
        else if (a[mid]<x)
            low = mid+1;
        else
            high = mid;
    }
    return -1;
}
```

This way of searching is called *binary search*, because you always look at the middle element and then decide between one of the two halves (which is a 'binary' decision).

If you think this was an interesting puzzle, have you considered a career in computer science?

Chapter 14

1. *Hierarchies and hierarchies*
   A class hierarchy has to do with *inheritance*: it's a tree-like structure that shows how classes are subclasses of each other. For example, you could have a Food class for describing all kinds of food, and subclasses Fruit, Vegetable, and so on for the more specific types of food that exist.

   A game-object hierarchy has to do with objects *containing* other objects in a (usually recursive) way. For example, a garage can contain several cars, and each car contains an engine, wheels, and so on.

   It's a common (beginner's) mistake to confuse the concepts of inheritance and containment, for example by making Car a subclass of Garage, and Wheel a subclass of Car. We'll leave it to you to imagine the horrible consequences this could have...

2. *Global positions without recursion*
   With a **while** loop, the idea is to add up the LocalPosition of all objects that lie (hierarchically) between this object and the root. Use a helper variable that points to the object you're currently looking at, and keep using the Parent property to go up and up:

```
Vector2 GlobalPosition
{
    get
    {
        Vector2 result = LocalPosition;
        GameObject p = Parent;
        while (p != null)
        {
            result += p.LocalPosition;
            p = p.Parent;
        }
    }
}
```

   Watch out: do *not* use GlobalPosition inside GlobalPosition! This would make the property (sort of) recursive again, which is not what we asked for in this question.

3. *Loops and recursion*

   a. This is the same solution as in Chapter 9.

```
int Factorial(int n)
{
    int result = 1;
    for (int t=1; t<=n; t++)
        result *= t;
    return result;
}
```

b. The key observation here is that $n!$ is the same as $(n-1)! \times n$. Once you get that far, the recursive version is very similar to SumRecursive from this chapter of the book. Have a look at the explanation there if you get confused.

```
int Factorial(int n)
{
    if (n == 1)
        return n;
    return Factorial(n−1) ∗ n;
}
```

4. *Recursive reversal*

a. Assume that the string s has at least 2 characters. Let's use the term 'middle part' for the part of s that excludes the first and last character. The reversed version of s can be described recursively as: the last character of s, plus the reversed version of the middle part, plus the first character of s.

If s has zero or one characters, then the reversed version of s is simply s itself! This is the base case of our recursion. In total, this gives the following method:

```
string Reverse(string s)
{
    if (s.Length < 2)
        return s;

    string middlePart = s.Substring(1, s.Length−2);
    return s[s.Length−1] + Reverse(middlePart) + s[0];
}
```

b. The previous version of Reverse uses the Substring method in every recursive step, so it basically makes many copies of (smaller and smaller parts of) the input string. To prevent this, give the Reverse method two **int** parameters, say first and last, that denote the index of the first and last character that we're currently looking at. Instead of passing a smaller string to the recursive call, we just make the values of first and last grow towards each other. The recursive part looks like this:

```
return s[last] + Reverse2(s, first+1, last−1) + s[first];
```

One annoying detail is that we now have two base cases. If first and last are exactly the same, then we're interested in a substring of length 1, and we should return exactly that character. If first is larger than last, then we've handled all characters of s, and we should return an empty string. Here is the full recursive method:

```
string Reverse2(string s, int first, int last)
{
    if (first > last)
        return "";
    if (first == last)
        return s[first].ToString();

    return s[last] + Reverse2(s, first+1, last−1) + s[first];
}
```

And to be really complete, you'd have to add a method around it that gives first and last the correct starting values:

```
static string Reverse(string s)
{
    return Reverse2(s, 0, s.Length−1);
}
```

5. *Recursive searching*

```
int IndexOf(double[] a, double val)
{
    return IndexOfRecursive(a, val, 0, a.Length);
}

int IndexOfRecursive(double[] a, double val, int low, int high)
{
    if (high <= low)
        return −1;

    int mid = (low+high)/2;
    if (a[mid]==x)
        return mid;

    if (a[mid]<x)
        return IndexOfRecursive(a, val, mid+1, high);

    return IndexOfRecursive(a, val, low, mid);
}
```

Chapter 15

1. *Playing with a grid*

   a. In this question, it's unclear where the game's Random object is. You could create one yourself inside the constructor, like this:

   ```
   Random r = new Random();
   ```

   ...or you could assume that it's already been created and that you can access it (statically) via a property. Anyway, the important part of this question is how to fill the grid:

   ```
   for (int x=0; x<width; x++)
       for (int y=0; y<height; y++)
           numbers[x,y] = r.Next(1, 6);
   ```

   b. We'll assume that pos indicates a position that actually exists. (Otherwise, you can add some initial **if** checks to handle special cases.)

   ```
   void Increment(Point pos)
   {
       numbers[pos.X, pos.Y]++;
       if (numbers[pos.X, pos.Y] > 5)
           numbers[pos.X, pos.Y] = 0;
   }
   ```

c.
```
bool AllZeros()
{
    for (int x=0; x<numbers.GetLength(0); x++)
        for (int y=0; y<numbers.GetLength(1); y++)
            if (numbers[x, y] != 0)
                return false;
    return true;
}
```

d. Here, the easiest approach is to use *recursion*! Set the current grid cell to 0, but keep its old value n as a back-up. If n was already zero, then we've already visited this cell, and the recursion should stop here.

If n was not zero yet, we should check all four neighbors, taking into account that you may have reached the edge of the grid. For every valid neighbor, visit that neighbor recursively by calling RemoveIsland for that position. The result looks like this:

```
void RemoveIsland(Point pos)
{
    int n = numbers[pos.X, pos.Y];
    if (n == 0)
        return;

    numbers[pos.X, pos.Y] = 0;

    if (pos.X > 0)
        RemoveIsland(new Point(pos.X−1, pos.Y));

    if (pos.X+1 < numbers.GetLength(0))
        RemoveIsland(new Point(pos.X+1, pos.Y));

    if (pos.Y > 0)
        RemoveIsland(new Point(pos.X, pos.Y−1));

    if (pos.Y+1 < numbers.GetLength(1))
        RemoveIsland(new Point(pos.X, pos.Y+1));
}
```

e. We'll leave this part to you!

Chapter 16

We will not give solutions to the programming challenges at the end of each part. Have fun extending the game, and don't be afraid to challenge yourself!

In fact, *all* exercises for this chapter are all open programming challenges, and we will leave them to you.