Wouter van Toll, Arjan Egges, Jeroen D. Fokker

Learning C# by Programming Games

Second Edition

Exercise Solutions for Part V: Animation and Complexity

August 10, 2019

Springer

Solutions for Part V: Animation and Complexity

Chapter 22

1. *Exceptions*
   An exception is an 'unexpected' event that a program encounters at run-time. This can have many different causes. Simple examples include trying to use an object that is **null**, or trying to access an array index that doesn't exist. Other examples are a failing network connection, a file that cannot be found, or a file with an incorrect format that cannot be read properly.

   It's good practice to use exception handling *only* for things that happen beyond the programmer's control, such as in the second category of examples we just mentioned.

2. *Exceptions and file reading*

   a. Here's the method:

```
int ReadSum(string filename)
{
    int result = 0;

    StreamReader r = new StreamReader(filename);

    // read the lines one by one; build up the result
    string line = r.ReadLine();
    while (line != null)
    {
        result += int.Parse(line);
        line = r.ReadLine();
    }

    return result;
}
```

   ...and here's a usage example:

```
int sum = ReadSum("someFile.txt");
Console.WriteLine("The sum is " + sum + "!");
```

b. Change the following line:

```
StreamReader r = new StreamReader(filename);
```

into this:

```
StreamReader r;
try { r = new StreamReader(filename); }
catch (Exception e)
{
    Console.WriteLine("The file does not exist.");
    return 0;
}
```

c. Change the following line:

```
result += int.Parse(line);
```

into this:

```
try { result += int.Parse(line); }
catch (Exception e) { }
```

d. In the changes from the previous subquestion, change the **catch** block into this:

```
catch (Exception e) { throw new InvalidLineException(); }
```

e. In general, if you want to prevent your program from crashing at all times, you should use **try** and **catch** whenever you call a method that may cause exceptions without already handling them itself.
In the case of ReadSum, this 'outer' exception handling could look like this:

```
try
{
    int sum = ReadSum("someFile.txt");
    Console.WriteLine("The sum is " + sum + "!");
}
catch (Exception e)
{
    Console.WriteLine("Something went wrong!");
}
```

You can make it more specific by dealing with InvalidLineException in a special way:

```
try
{
    int sum = ReadSum("someFile.txt");
    Console.WriteLine("The sum is " + sum + "!");
}
catch (InvalidLineException e)
{
    Console.WriteLine("One of the lines was invalid.");
}
catch (Exception e) // something else went wrong
{
    Console.WriteLine(e.Message);
}
```

f. This is a matter of taste. It really depends on your application, too. If you're writing software for a bank, and there's a wrong line in a file with transactions, what should you do? If you stop reading as soon as something goes wrong, then you'll skip the remaining (and possibly valid!) lines, which could mean that people do not receive their money.

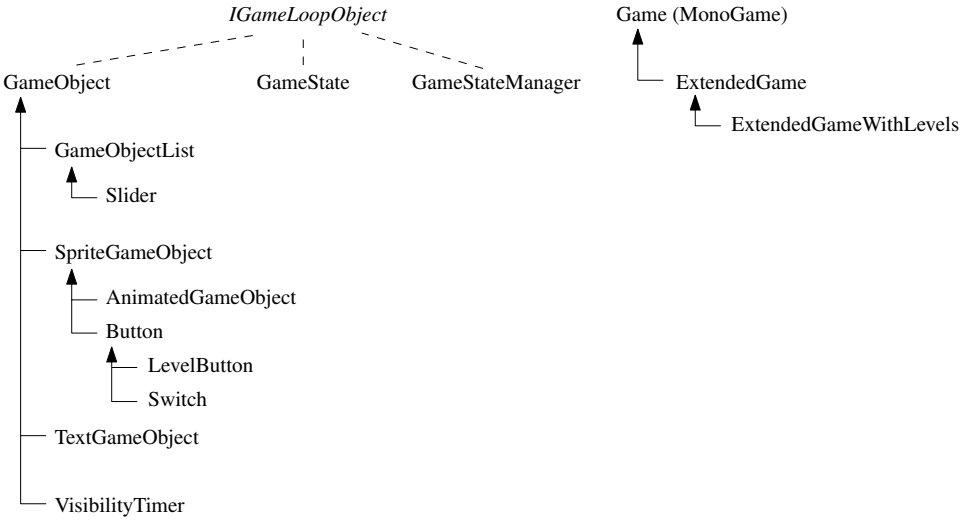
   The bottom line is: there are many possible approaches, so try to think of the consequences of each approach, and then determine the nicest one.

3. *Using the new engine with Penguin Pairs*
   We'll leave the details of this to you. You should be able to find several places where 'Penguin Pairs'-specific code is now already covered by the engine.

4. *Exceptions in Tick Tick*
   In Tick Tick, exceptions can be useful for handling unexpected events such as level files not being available, level files containing invalid content, or the progress file not being accessible for writing. Note that these three examples are all related to file I/O. This is a common area for exception handling!

Chapter 23

1. *Class hierarchy of the engine*
   Here's a diagram that shows only those classes that have some inheritance relation. You could make an even bigger diagram that contains all other classes as well.



Chapter 24

1. *Bounding volumes*
   A bounding volume is a simplified representation of a game object. It can be used for physics calculations such as collision detection. The main advantage of bounding volumes is that they make those calculations easier and therefore faster. The main disadvantage is that they are not perfect versions of your actual game objects, which may have consequences for the gameplay.

2. *Collision detection between rectangles*
   Here's our version. We've assumed that we are writing inside the Rectangle class itself, but you could also create a version with *two* rectangles as parameters.

```
public bool Intersects(Rectangle other)
{
    bool xOverlap = (this.Left <= other.Right && this.Right >= other.Left);
    bool yOverlap = (this.Top <= other.Bottom && this.Bottom >= other.Top);
    return xOverlap && yOverlap;
}
```

Chapter 25

1. *Improving the class hierarchy*
   Ah, another open programming exercise. You know what to do!

   You could create a GameCharacter class that becomes the superclass of both Player and Enemy. This class could take care of most things that happen in the constructor: storing a reference to the level, storing the start position, and loading an animation.

   If you ever want to create enemies that can jump (and that should therefore experience gravity and handle collisions with tiles), you could move more Player code into the GameCharacter class, and give GameCharacter member variables that denote if (and how strongly) a character should respond to gravity and collisions.

Chapter 26

We will not give solutions to the programming challenges at the end of each part. For this chapter, the exercises contain some truly challenging extensions, as an ultimate test to see how much you've learned in this book. Try to see how far you can get!

Thanks again for using this book!

– Wouter, Arjan, and Jeroen