Wouter van Toll, Arjan Egges, Jeroen D. Fokker

Learning C# by Programming Games

Second Edition

Exercise Solutions for Part IV: Menus and Levels

August 10, 2019

Springer

Solutions for Part IV: Menus and Levels

Chapter 17

1. *Abstract classes and interfaces (1)*

   An abstract class is a class of which you cannot create any instances; it is purely used for other classes to inherit from. An interface is a list of method headers that other classes can implement; it does not contain any implementations itself. Also, a class can only inherit from one class, but it can implement multiple interfaces.

   There are countless examples that you could give for both concepts. If you cannot come up with anything, consider reading this chapter from the book again.

2. *Abstract classes and interfaces (2)*

   a. False: an abstract class could have all of its methods filled in already. It makes sense if at least one method is *virtual* (so that it can be overwritten by a subclass), but even this is not mandatory.
   b. False: an abstract class can contain private methods that are 'helpers' for other (non-private) methods. Sure, subclasses cannot call such a method directly, but it can still be used in the abstract class itself.
   c. True: an abstract method *must* be filled in by subclasses, and the keyword **private** would prevent this from being possible.
   d. False: an abstract class can contain non-abstract methods that are already filled in.
   e. True: by definition, an interface is just a list of method headers.

3. *Abstract classes: What is allowed?*

```
A obj; // Yes, declaring a reference is always allowed
obj = new A(); // No, you cannot create instances of A, because it's an abstract class
obj = new B(); // Yes, you can create instances of a non−abstract subclass
obj.Method1(); // Yes, the compiler know that any subclass will have this method
obj.Method2(); // Yes, all subclasses of A inherit this method, and it is public
obj.Method3(obj); // No, the compiler can't verify that obj has this method
B otherObject = (B)(new A()); // No, (again) you cannot create instances of A
A yetAnotherObject = (A)obj; // Yes, casting to a type higher in the hierarchy is allowed
obj.method3(otherObject); // No, (again) the compiler can't verify that obj has this method
A[] list; // Yes, you're not creating the specific array elements yet
list = new A[10]; // Yes, you're not creating the specific array elements yet
list[0] = new A(); // No, (again) you cannot create instances of A
```

```
list[1] = new B(); // Yes, (again) you can create instances of a non−abstract subclass
List<A> otherList = new List<A>(); // Yes, again no instances of A are created here
```

4. *Loading game states dynamically*
   This is not a big change. Nothing's stopping you from calling AddGameState much later in the game! A nice addition would be a helper method that returns if the game state with a given name already exists:

   ```
   bool GameStateExists(string id)
   {
       return GetGameState(id) != null;
   }
   ```

   That way, you'll know when you need to call AddGameState first.

   And if you're on it anyway, it's a good idea to also allow the *unloading* of game states, by adding a method **void** RemoveGameState(**string** id). Try to fill in the details of this yourself.

5. *Lists and interfaces*

   a. Versions 1 and 3 are incorrect. We try to create an instance of IList, but this is not allowed because IList is an interface and not a class.

      Version 2 is correct, because the List class implements the IList interface, so it may be assigned to a variable of the type IList.

   b. Version 2 may be better than version 4 if you want to keep the possibility to choose another implementation of the IList interface at a later stage. By using the interface, you make sure that you do not accidentally use methods or properties from List that are not specified in IList.

Chapter 18

1. *More UI elements*
   Some examples of other UI elements are scrollbars, drop-down menus, and radio buttons. This last option is probably the easiest to implement.

2. *Pressing a level button*
   One solution is to give each level button a reference to the LevelMenuState that contains it, and to give LevelMenuState a public method (such as LevelButtonClicked(**int** index)) that a level button can call when it is pressed.

3. *Better buttons*
   This open programming exercise should not be too difficult with all the things you've learned so far in this book. Once again, we'll leave the details to you!

Chapter 19

1. *File reading*

```
int ReadSum(string filename)
{
    // read the file's single line
    StreamReader r = new StreamReader(filename);
    string line = r.ReadLine();
    r.close();

    // get the separate numbers
    string[] numbers = r.Split(' ');

    // convert to integers and keep track of the total
    int total = 0;
    foreach (string number in numbers)
        total += int.Parse(number);

    return total;
}
```

2. *File writing*

```
void WriteMississippi(string filename, int n)
{
    StreamWriter w = new StreamWriter(filename);

    for (int i=1; i<=n; i++)
        w.WriteLine(i + " Mississippi");

    w.close();
}
```

3. *Closing a reader or writer*
   As soon as you close a reader or writer, the associated stream is no longer 'locked' and the target can be used by other programs (or by other parts of your own program). So, closing a stream as soon as possible is considered good practice because it stops your code from keeping the stream all to itself.

   Also, sometimes it's not entirely clear *when* a C# object goes out of scope. It's easy if you create the reader/writer locally inside a method, but what if your method gives the reader/writer back as a result? Who should be in charge of 'deleting' it then? If you don't watch out, you may end up locking a file without knowing it— unless you properly use the Close method.

4. *Making the **switch***
   Basically, look for any situation where several simple **if** and **else** instructions appear in a row, with each instruction checking the same variable. A good example is the getSpriteNameForStatus method in LevelButton. With a **switch** instruction, it could look like this:

```
static string getSpriteNameForStatus(LevelStatus status)
{
    switch (status)
    {
        case LevelStatus.Locked:
            return "Sprites/UI/spr_level_locked";
        case LevelStatus.Unlocked:
            return "Sprites/UI/spr_level_unsolved";
        default:
            return "Sprites/UI/spr_level_solved@6";
    }
}
```

Chapter 20

1. *The keyword '**is**'*

   a.
```
if (animal is Dog)
    Console.WriteLine("Woof!");
else if (animal is Cat)
    Console.WriteLine("Meow!");
else // it's a fox
    Console.WriteLine("Ring ding ding ding ding diding diding!");
```

   b.
```
// Add this in the Animal class:
public abstract void Speak();
// And fill it in all subclasses, such as like this in Dog:
public void Speak() { Console.WriteLine("Woof!"); }
// The body of the foreach loop then becomes easy:
animal.Speak();
```

   c. The second approach is better for several reasons. First, it delegates the concept of 'speaking' to each specific animal, which is conceptually nicer. Second, if you ever add another Animal subclass (such as Elephant), then this class is obliged to implement the Speak method too, and this new Speak implementation will automatically be supported in the **foreach** loop, thanks to polymorphism!

      By contrast, the first approach would force you to add another **else if** instruction to the loop— otherwise, elephants would accidentally make fox noises! And trust us, that is *not* a world you want to live in.

Chapter 21

1. *Classes and access modifiers*

   a. A public class can be used everywhere, whereas an internal class can only be used inside the same assembly. This is relevant when a program includes another codebase, for example as a library.
   b. In general, it's a good assumption that classes will not be used by other programs. The only exception to this rule is when you deliberately write a library that other programs are *supposed* to use.
   c. Well, this goes a bit too far for this book, but it is technically possible to write *nested* classes like this:

   ```
   class Outer
   {
       class Inner { ... }
   }
   ```

   where the Inner class is defined 'inside' the Outer class. This can be useful if the functionality of Inner has so much to do with Outer that it isn't really relevant anywhere else. In such cases, you can define the class Inner as **protected** or **private**, to define whether any subclasses of Outer can also use Inner.

   But for any class that is *not* nested inside another class, it is simply not possible in C# to mark them as protected or private.

2. *Extending the Penguin Pairs game*
   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!