Wouter van Toll, Arjan Egges, Jeroen D. Fokker

Learning C# by Programming Games

Second Edition

Exercise Solutions for Part II:
Game Objects and Interaction

August 10, 2019

Springer

Solutions for Part II: Game Objects and Interaction

Chapter 5

1. *Drawing sprites in different locations*
   We won't give you the full program here, but we do provide some hints. If your sprite is stored under the variable name sprite, then:

   - GraphicsDevice.Viewport.Width − sprite.Width is the x-coordinate of the rightmost balloon;
   - GraphicsDevice.Viewport.Height − sprite.Height is the y-coordinate of the bottom balloon;
   - (GraphicsDevice.Viewport.Width − sprite.Width)/2 is the x-coordinate of the center balloon;
   - (GraphicsDevice.Viewport.Height − sprite.Height)/2 is the y-coordinate of the center balloon.

2. *Flying balloons*
   This is a rather 'open' exercise with many possible answers. Try to find a solution yourself, but if you get lost in the details, don't be afraid to just continue with Chapter 6 (or take a well-deserved break).

Chapter 6

1. *Enumerated types*
   An enum(erated type) is a data type with a fixed number of possible values. Each possible value is written as a word, which makes them more human-readable than integers in certain cases. Also, the compiler will prevent you from using values that you haven't defined.

   Here's another example:
   ```
   enum Direction { Left, Right, Up, Down };
   ```

   This example can be useful in a grid-based game where objects can move in only four directions. Instead of using integers with 'magic values' (1 for left, 2 for right, and so on), it is nicer to use such an enum here.

2. *Strange use of* ***if*** *and* ***else***
   This code gives the variable b the same value as a. Unless a has a value of 0: in that case, the code explicitly sets b to 0. In other words, this case actually does the same as the instruction b = a;! This means that the whole **if/else** structure is not needed, and you can just rewrite the code to this one instruction:

   ```
   b = a;
   ```

3. *Mixed seasoning*

   a. Oops: this code also shows the spring background if month > 6 and day < 21, for example on July 5 and August 10.
   b. Rewrite the **else if** line to the following: **else if** (month == 6 && day < 21). This way, the '¡ 21' case is only handled in June.
   c. We'll leave this for you to fill in. It's the same idea, but with different months!

4. *Bouncing balloons*
   In the later chapters of this book, you'll see many examples like this. For example, in Chapter 8 (section 2), we will implement the movement of the ball subject to gravity. Have a look at that section for inspiration!

Chapter 7

1. *Keywords*

   a. The word 'void' literally means 'emptiness'. In C#, we use it to indicate that a method doesn't have a return value.
   b. The word 'int' is an abbreviation of 'integer', which is used in C# to define the integer data type for storing whole numbers.
   c. The keyword **return** is used inside the body of a method to return a result. The program will then jump out of the method and return to the place where the method was called.
   d. The keyword **this** is used to indicate the object that we're currently manipulating. We may need it on several occasions, for example to let an object pass itself as a parameter to a method, or to resolve a name conflict between a member variable and a local variable.

2. *Properties in Painter*

   a. If you want to give the Angle property custom behavior, you'll have to fully write out the **get** and **set** parts again, and use a (private) member variable in the background:

   ```
   private float angle;
   public float Angle
   {
       get { return angle; }
       set
       {
           if (value >= 0 && value <= 2*Math.PI) angle = value;
       }
   }
   ```

b. Here's how to do it with a read-only property:

```
public Texture2D BarrelSprite { get { return cannonBarrel; } }
```

Another option would be to use a property with a private **set** part and a public **get** part. This allows you to get rid of the cannonBarrel member variable:

```
public Texture2D BarrelSprite{ get; private set; }
```

Both options are perfectly fine. The second is shorter, but bear in mind that it's no longer possible if you ever want to give the **get** or **set** part any custom code, such as in question (a).

c. Same story: either add a read-only property...

```
public SpriteBatch SpriteBatch { get { return spriteBatch; } }
```

...or replace the member variable by a property with two parts:

```
public SpriteBatch SpriteBatch { get; private set; }
```

3. *Methods with a result*

a.
```
double Circumference(double height, double width)
{
    return 2*height + 2*width;
}
```

b.
```
double Diagonal(double height, double width)
{
    return Math.Sqrt(height*height + width*width);
}
```

4. *Methods about dividers*

a.
```
bool Even(int x)
{
    return x%2 == 0;
}
```

b.
```
bool MultipleOfThree(int x)
{
    return x%3 == 0;
}
```

c.
```
bool MultipleOf(int x, int y)
{
    return x%y == 0;
}
```

d. This is actually the same question as the previous one. You could give the same answer, or you could let the Divisible method call the MultipleOf method, like this:

```
bool Divisible(int x, int y)
{
    return MultipleOf(x,y);
}
```

5. *Multiple flying balloons*
We'll leave this to you. If you get stuck, have a look at how this chapter handles the Cannon class again, and try to see if/how the Balloon class is any different.

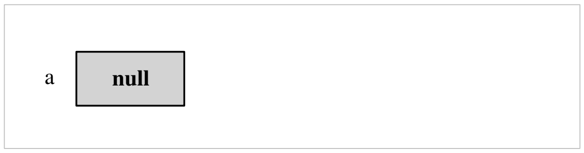
Chapter 8

1. *Static*

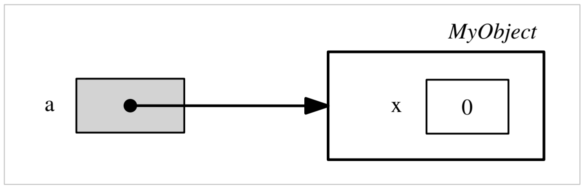
a. The keyword **static** can be used for member variables, methods, and properties in any class MyClass. If you mark such an element as **static**, it means that the element is shared by all instances of MyClass, so it does not manipulate (or use the data of) a specific instance.
b. MonoGame contains a lot of static (read-only) properties. For example, the Color class contains many static read-only properties for predefined colors, such as Color.Red. These pretty much act as constants, except that a new Color object is returned every time you call the property.
c. Because static methods don't manipulate a specific object, the 'current object being manipulated' is not defined there. Thus, the keyword **this** has no meaning in a static method, and the compiler will forbid you to use it.

2. *Memory models*

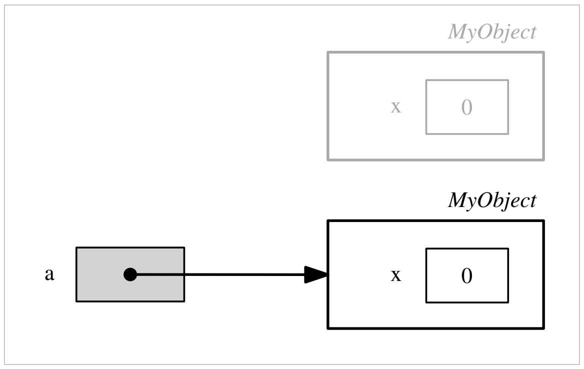
a. After the first instruction, the variable a does not point at anything yet:



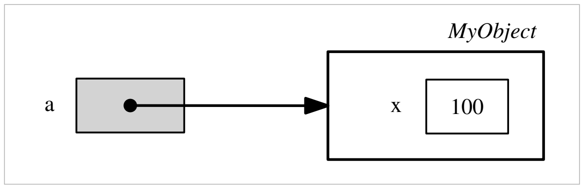
After the second instruction, a points to a MyObject instance:



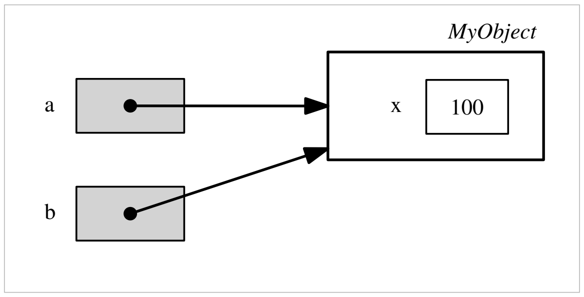
After the third instruction, a points to a new MyObject instance. The old MyObject instance has no more references to it, so it will eventually be removed via garbage collection:



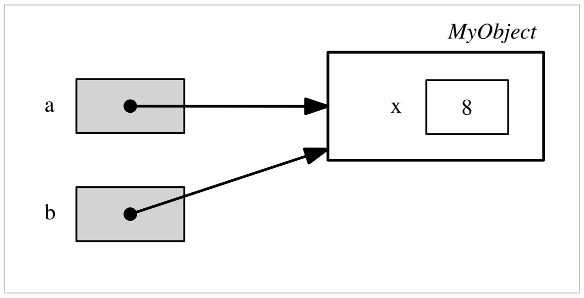
After the fourth instruction, the x member variable of the MyObject instance has changed:



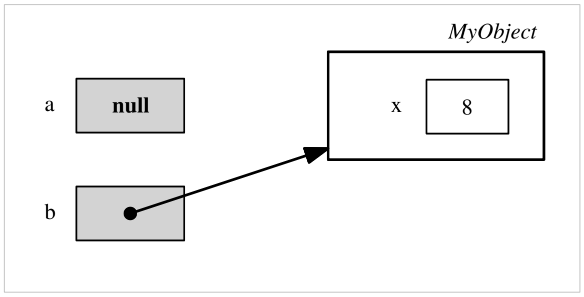
After the fifth instruction, a second variable b points to the same MyObject instance as a:



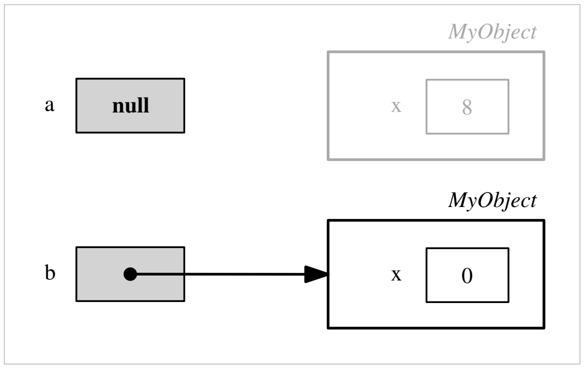
After the sixth instruction, the x member variable of the MyObject instance (which is 'shared' by a and b) has changed again:



After the seventh instruction, a does not point to anything anymore, while b still points to the same object:



After the eighth instruction, b points to a new MyObject instance, while a still points to nothing. The old MyObject instance has no more references to it, so it will eventually be removed via garbage collection:



b. If MyObject is a struct instead of a class, then the fifth instruction (MyObject b = a;) creates a second instance of MyObject, which is a copy of the other instance. Then, a and b will both point at different objects (both with an x value of 100). And of course, the sixth instruction (b.x = 8;) will change only one of those objects.

3. *Random numbers*
We'll assume that our code has access to an object of type Random via a static property Random, just like how we implemented it for Painter in this chapter.

a. To obtain a number between min and max, the trick is to get a random number between 0 and max − min, and to then add min to the result.

```
public void RandomDouble(double min, double max)
{
    return min + Random.NextDouble() * (max − min);
}
```

b. Getting random numbers is typically something you want to allow anywhere in your program. It's a bit pointless to give each object its own random number generator, so it's common to store that generator in a central place for all classes to use.

c. Sometimes, it's useful to ensure that a game (or another program) always draws exactly the same random numbers. An example is a program that runs scientific experiments that you want to repeat in the exact same way. Another example is a game level where you want seemingly random events that the player can still learn by heart (by replaying the level many times). To enable this, simply give your Random object the same seed each time the program starts.

   It can be good to use one Random object for those things that need to be the same every time, and a second Random object for the things that are allowed to change (such as explosions and other visual effects). You can then give the first Random object a fixed seed, and the second Random object an arbitrary one.

Chapter 9

1. *Results of loops*
   The first loop will give x the value 0 + 1 + 2 + ... + 9, which is 45.

   The second loop is similar, but it only adds even numbers to the sum. So, x will get the value 0 + 2 + 4 + 6 + 8, which is 20.

   The third loop will keep doubling x until it is larger than 1000, which happens as soon as x becomes 1024.

   The fourth loop is a nasty trick question. Here, x will remain 1, because the loop condition x > 50 already doesn't hold at the very beginning.

2. *An empty loop*
   If you remove the braces after the loop's header, then the compiler will think that this is a loop with one instruction inside it (and not zero). The first instruction that comes immediately *after* the loop will then be treated as if it is *inside* the loop. This will most likely give a compiler error. If you're unlucky, though, the result might actually be (syntactically) correct code, and your program will behave very strangely.

3. *Methods with loops*

   a. The idea here is to keep subtracting y until it is no longer possible:

```
int RemainderAfterDivision(int x, int y)
{
    int result = x;
    while (result >= y)
        result -= y;
    return result;
}
```

   You could also abuse the way in which integers are automatically rounded down. This gives a shorter program without any loops:

```
int RemainderAfterDivision(int x, int y)
{
    return x - (x / y) * y;
}
```

b.
```
int Total(int n)
{
    int result = 0;
    for (int i=1; i<=n; i++)
        result += i;
    return result;
}
```

By the way, you can actually calculate the same result without using a **while**- or a **for**-instruction:

```
int Total(int n)
{
    if (n <= 0) return 0;
    return n * (n+1) / 2;
}
```

c. This is basically the Total method, but with multiplication instead of addition. Make sure to start with a result of 1 (instead of 0), otherwise the method will always return zero!

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

d.
```
double Power(double x, int n)
{
    double result = 1;
    for (int i=0; i<n; i++)
        result *= x;
    return result;
}
```

e. By reusing the Power and Factorial methods, we can write down this method as follows:

```
double Coshyp(double x)
{
    double res=0;
    for (int t=0; t<40; t+=2)
        res += Power(x,t) / Factorial(t);
    return res;
}
```

4. *Prime numbers*

a. Here's one way to do it, via a **while** loop:

```
int SmallestDivider(int x)
{
    int divider = 2;
    while (x % divider != 0)
        divider++;
    return divider;
}
```

b. If you think about it, x is a prime number if x is its own smallest divider:

```
bool IsPrimeNumber(int x)
{
    return SmallestDivider(x) == x;
}
```

Chapter 10

1. *Inheritance*
   Inheritance means that you specify that some classes are specific versions of other classes. The main reason is to avoid code duplication: all the shared code will be inside the parent class. This makes your life as a programmer much easier, especially when you have to change things in the code you already have.

2. *Access modifiers*
   In the following tables, we will use Y for 'yes' and N for 'no'.

a.

this.var1 [Y]	**this**.var2 [Y]	**this**.var3 [Y]
this.var4 [N]	**this**.Var2 [Y]	**base**.var1 [N]

To explain the 'N's: var4 cannot be accessed in class A (because it only exists in class B), and the keyword **base** does not make sense in class A (because it does not have a parent class).

b.

this.var1 [Y]	**this**.var2 [Y]	**this**.var3 [N]
this.var5 [Y]	**this**.Var2 [Y]	**base**.var1 [Y]
base.var2 [Y]	**base**.var3 [N]	**base**.Var2 [Y]

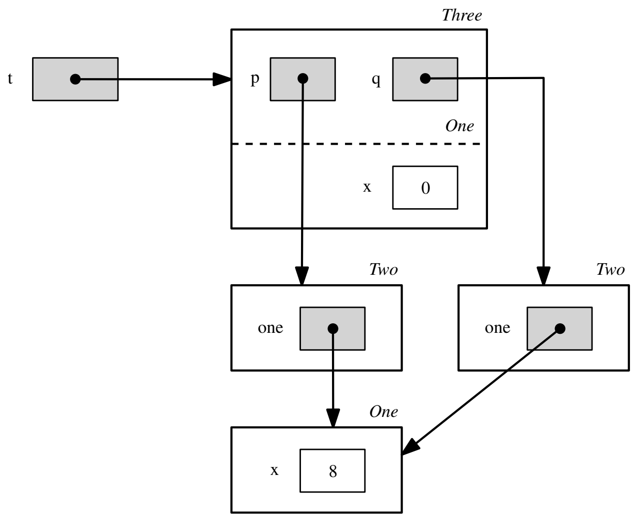
To explain the 'N's here: var3 is declared as private in class A, so you cannot access it from inside B (even if you use the keyword **base**).

3. *Polymorphism*
   The trick here is that Cat uses the keyword **override**, while Dog does not.

   a. This will print Woof! because bingo is of type Dog.
   b. This will print ??? because bingo2 is of type Animal, and the Dog class does not properly override the Speak method. In this case, there is no way for the program to know that one method is meant as a more specific version of the other.
   c. This will print Meow! because mittens is of type Cat.
   d. This will print Meow! because mittens2 is of type Animal, but the Cat class overrides the Speak method correctly. In this case, the program can find the more specific version of Speak. This is an example of polymorphism!

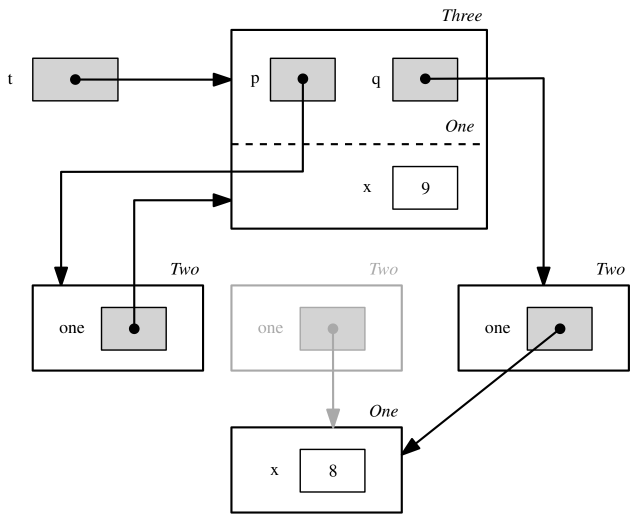
4. *Drawing the memory*
   We'll show the answer in two steps. A tricky thing to watch out for is that Three is a subclass of One, so an object of type Three will inherit a member variable x.

   Look at the constructor of the Three class. After the first four instructions, the member variables p and q will point at two different Two instances, but these Two instances will point at the same One instance. So, there is only a single One instance that has two arrows pointing towards it. This object will have its x member variable changed to 7, and then to 8. Take your time to understand the situation up to this point.



   The fifth instruction will let p point to a new Two instance, so one of the existing Two instances will be left without any references to it. C# will remove this object soon via garbage collection. (We'll draw it in gray in the upcoming drawing.)

   Note that the new Two object receives **this** in its constructor. So, its one member variable will end up pointing to the overall Three object! (This is allowed because Three is a subclass of One.) And therefore, the final instruction will actually set the x member variable of the overall Three object. This results in the following final drawing:



5. *Type checking*
   Types are mostly checked during the compilation phase. The compiler has to check that you are using data types correctly. Here's an example to prove it:

```
class A {...}
class B : A {...}
class C : A {...}
class Test
{
    void Main()
    {
        A a;
        B b;
        a = new B(); // this is allowed because B is a subclass of A
        b = new A(); // this is not allowed: compiler error
        b = a; // also not allowed: compiler error
        b = (B)a; // this is allowed!
    }
}
```

   With the cast in the final instruction, you indicate (as a programmer) that you know that the assignment is safe. During run-time, there will be a final check to determine if the variable a indeed contains an object of type B. This check cannot be done at compile time, because (in theory) the variable a could also store an object of type C, which cannot be cast to a B.

Chapter 11

1. *Methods with loops and strings*

   a.
```
string ThreeTimes(string s)
{
    return s + s + s;
}
```

   b.
```
string SixtyTimes(string s)
{
    string result = "";
    for (int i=0; i<60; i++)
        result += s;
    return result;
}
```

   c.
```
string ManyTimes(string s, int nr)
{
    string result = "";
    for (int i=0; i<nr; i++)
        result += s;
    return result;
}
```

2. *Type conversions*

```
x = (int)d;
x = int.Parse(s);
s = x.ToString();
s = d.ToString();
d = x; // no type conversion needed
d = double.Parse(s);
```

3. *Cuneiform*

   a.
```
string Stripes(int n)
{
    string s = "";
    for (int t=0; t<n; t++)
        s += "|";
    return s;
}
```

   b.
```
string Cuneiform(int x)
{
    string s = "";
    while (x > 0)
    {
        s = Stripes(x % 10) + "−" + s;
        x = x / 10;
    }
    return "−" + s;
}
```

4. *Extending the Painter 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!