Wouter van Toll, Arjan Egges, Jeroen D. Fokker

Learning C# by Programming Games

Second Edition

Exercise Solutions for Part I: Getting Started

August 10, 2019

Springer

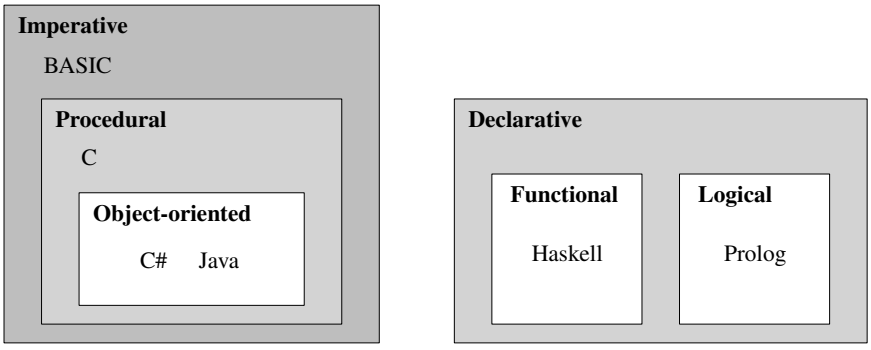
Solutions for Part I: Getting Started

Chapter 2

1. *Syntax and semantics*
   Syntax is the set of *'grammatical' rules* of a programming language: it defines what you can and cannot write in a (syntactically) correct program. Semantics refers to the *meaning* of a program: it defines what your code is actually trying to do. A compiler can only check the syntax of your program, and not the semantics.

   In the 'Edit-Compile-Run' cycle, developers first write their code code (syntax) and then compile it to check if it is syntactically correct. Finally, they run and test the program to see if it works as intended (semantics). If something is wrong, the cycle starts over.

2. *Programming paradigms: Overview*
   The diagram below contains the answers to both sub-questions.



3. *Programming paradigms: Statements*

   a. Not true: for example, Fortran is an imperative language that is not object-oriented.
   b. True: objects are modified by methods, which are a kind of procedure.
   c. Not true: for example, PHP is a procedural language that is interpreted, not compiled.
   d. Not true: a programming language can hide certain aspects of the underlying processor.

4. *The compiler as a program*

   a. Yes, if you already possess a compiler or interpreter for that language. Only the first compiler ever had to be written in machine code.
   b. Yes, that is possible.
   c. Yes it can, assuming that you have another compiler available to compile it for the first time, so that you can have an executable.

Chapter 3

1. *Instructions, methods, classes*
   An *instruction* is a piece of code that changes the computer's memory in some way. It is a part of the code that 'does something'. A *method* is a reusable group of instructions with a name. The instructions inside it will be executed whenever the method is called. A *class* is a reusable group of methods (plus member variables and some other things) with a name. It defines the behavior of the objects in your program.

2. *Comments*
   Comments can be written on a single line with the // notation, or spread over multiple lines by starting the comment with /∗ and ending it with ∗/.

   The most important reason to use comments is to make your code understandable for human readers: not just other programmers, but also yourself!

3. *The game loop*

   a. A classical game loop consists of two methods: Update (for changing the state of the game world) and Draw (for drawing the game world on the screen in its current state).
   b. In MonoGame, there are extra methods such as LoadContent (the first place where you can load the assets of your game). Its counterpart is UnloadContent, which we do not use in this book. Finally, there is Initialize, another start-up method for setting up various aspects of your game.
   c. In a *fixed-timestep* loop, each iteration simulates the passing of a fixed amount of time (such as $1/60$ seconds). This makes sure that the game behaves the same on all computers, regardless of how powerful they are. However, slower computers may not be able to run the game in real-time.
      In a *variable-timestep* loop, the computer simulates as many iterations as possible. This ensures real-time performance on all devices, but it does mean that the game behaves differently on fast and slow computers.
   d. The first advantage is that the code is easier to read: Update handles one task, and Draw handles the other. Secondly, this forces the programmer to not mix up these tasks: a drawing instruction inside the Update method will simply have no effect. Thirdly (although we have not discussed this in the book), a game engine can sometimes make certain optimizations for the performance of the game. For example, if the game world doesn't change, then it doesn't have to be redrawn.

4. *The smallest program (for real, this time)*
   Here is a very small program of 29 characters:
   class A{**static void** Main(){}}

It has all the necessary ingredients, but no instructions. This is syntactically correct (although it's a bit useless, of course). If you're really eager, you can shorten this even further by renaming the Main method to something like B, and then changing the project settings so that B is marked as the program's main method.

Chapter 4

1. *Names*
   Examples of *constants* are e and π in mathematics, and c (the speed of light) and g (gravity constant) in physics. These are numbers with a fixed value that are useful in certain areas.

   For *variables*, it is common to use the symbol x for a standard variable, f for a function, and ϕ or θ for an angle. In physics, we often use v for velocity, a for acceleration, and m for mass.

   If everyone uses the same symbols for certain constants and variables, then it is much easier to communicate.

2. *Concepts*
   An instruction is a piece of code that can be executed. A variable is a location in memory with a name. A method is a group of instructions with a name. An object is a group of variables that belong together. A class is a group of methods, and also the type of an object that these methods can modify.

3. *Declaration, instruction, expression*
   An *instruction* is a programming construct that can be executed, so that the memory changes. A *declaration* is an instruction that 'promises' the existence of a variable with a certain type. An *expression* is a piece of code with a value (that can be evaluated).

4. *Statement versus instruction*
   The word 'statement' can also mean something like a 'remark' or a 'point of view', or a quote that you can agree or disagree with. We use the word 'instruction' to emphasize that we give the computer a *command* to do something.

   For example, a statement could be something like: "the grass is purple". This is different from giving the computer a command or an instruction to *change* the color of grass into purple: `grass.Color = Color.Purple;`.

   In fact, the word 'statement' would actually make more sense for a *Boolean expression*, as you will discover in later chapters.

5. *Changing names*
   We have to change the name of the class, the name of its constructor method, and the call to this constructor method from Main. Apart from that, it makes sense to change the name of the *.cs* file of your program, because it's common to let the filename and the class name match. However, this is technically not necessary.

6. *Playing with colors*

   a. These colors are (from top to bottom) red, green, blue, yellow, magenta (a kind of purple), cyan, white, and orange. Try them out in an example program!
   b. To use blue instead of red, the game should change the color's third component (instead of the first one).
   c. To use purple, the game should change the red *and* blue components.
   d. To let the color change in the opposite order, you'll either have to let the variable decrease (instead of increase), or use a more complicated expression (involving `255 − ...` in the `Draw` method.

7. *Syntactical categories*
   Here are our answers:

int x; [DI]	**int** 23; []	(y+1)∗x [E]	**new** Color(0,0,0) [EM]
(**int**)x [E]	23 [E]	(x+y)(x−1) []	**new** Color black; []
int(x) []	23x0 []	x+1=y+1; []	Color blue; [DI]
int x []	x=23; [AI]	x=y+1; [AI]	GraphicsDevice.Clear(Color.White); [IM]
int x, **double** y; [DI]	"x=23;" [E]	spriteBatch.Begin(); [IM]	Content.RootDirectory = "Content"; [AI]
int x, y; [DI]	x23 [E]	Math.Sqrt(23) [EM]	Color.CornflowerBlue [E]
"/" [E]	0x23 [E]	"\\" [E]	Color.CornflowerBlue.ToString() [EM]
"\" []	23%x [E]	(x%23) [E]	game.Run() [M]
"//" [E]	x/∗23∗/ [E]	"" [E]	23=x; []

   The fragments marked with [] are invalid code. Some fragments leave room for (nearly philosophical) discussion:

   - x, y, x23, etc. are valid as variable names, but of course, they are only valid code if you also declare these variables somewhere.
   - game.Run() could also be seen as an *expression* of type **void**, depending on whether you count **void** as a datatype.
   - You could argue that a combination of *two* declarations (such as **int** x, y;) is not a declaration anymore.
   - Not all books or websites agree that every declaration is an instruction, and some would prefer the answer [D] wherever we wrote [DI].

   But try not to worry too much about these details.

8. *Relation between syntactical categories*

   a. The following combinations of categories are possible: DI, AI, IM, EM, ADI. The last one (for a combined declaration and assignment) doesn't appear in this table, though.
   b. A declaration and an assignment are both a specific type of instruction, so D and A always imply I.
   c. An instruction *always* ends with a semicolon, and therefore an assignment and a declaration do that as well. An expression *never* ends with a semicolon. A method call *sometimes* ends with a semicolon, depending on whether it is part of an expression or part of an instruction.

9. *Variable assignment*
   In each group, always execute the instructions from top to bottom. You should then get the following results:

y = 41 x = 42	x = 12 y = 12	x = 13 y = 12	x = 12 y = 40
y = 13 x = 39	y = 0 x = 26	y = 4 x = 6	

   Watch out: we're slightly abusing the = symbol here. It has nothing to do with the assignment operator anymore. Instead, you should interpret it as the classical 'equals' that you know from mathematics. So, the symbols in this table are not meant as C# code.

   The swap between x and y works for all cases (as long as the two numbers are not too big). You can see that this works if you replace the constants 40 and 12 by symbols a and b. If you fill in these values on the right hand side the final situation will be y = a and x = b. However, the sum of x and y shouldn't be so big that x+y doesn't fit into an **int** anymore.

10. *Multiplying and dividing*
    Mathematically speaking, there is no difference. But if we are dealing with variables of type **int**, then the rounding of the division will result in a different outcome. For example, consider the case where time contains the value 5. Then 3∗5/2 = 15/2 = 7. But 3/2∗5 = 1∗5 = 5, and 5/2∗3 = 2∗3 = 6.

11. *Hours, minutes, seconds*
    The following instructions do the job:

```
int hours = time / 3600;
int minutes = (time % 3600) / 60;
int seconds = time % 60;
```

    The reverse operation is easier:

```
time = 3600∗hours + 60∗minutes + seconds;
```