Appendix A
Exercises and Challenges

A.1 Exercises and Challenges for Part I

Exercises

1. *Programming paradigms*
   Indicate whether the following statements are true or not (and explain why):
   (a) All imperative languages are object-oriented.
   (b) There are object-oriented languages that are not procedural.
   (c) Procedural languages have to be compiled.
   (d) Declarative languages cannot run on the same processor are imperative languages, since that processor can execute an assignment instruction, which doesn't exist in declarative languages.

2. *The compiler as a program*
   A compiler itself is also a program.
   (a) Can the compiler itself be written in a higher programming language?
   (b) And if so, can that language be the same as the language that the compiler compiles?
   (c) If so, can the compiler compile itself?

3. *Names*
   In mathematics and physics, it is quite common to use fixed variable and constant names. Which ones? Is this useful?

4. *Classes and types*
   Name five standard classes, and for three of those, also name a method that belongs to it. Also name three types that are not classes.

5. *Comments*
   What are the two ways to write comments in a C# program?

A. Egges et al., *Learning C# by Programming Games*,
DOI 10.1007/978-3-642-36580-5, © Springer-Verlag Berlin Heidelberg 2013

6. *Concepts*
   Provide short definitions of the concepts ‘instruction’, ‘variable’, ‘method’, and ‘object’. Which two relations does the concept ‘class’ have with these concepts?

7. *Declaration, instruction, expression*
   What’s the difference between a declaration, an instruction and an expression?

8. *Statement versus instruction*
   Many programming books use the word ‘statement’ to indicate an instruction in a programming language. Why do you think we avoid that word in this book?

9. *Changing names*
   Look at the class DiscoWorld that we discussed in Chap. 4. What do we have to change if we wanted to change the name of this class into Hello? What is not necessary to change, but is logical to change anyway?

10. *Syntactical categories*
    Indicate for each of the following program fragments to which syntactical category it belongs: (M)ethod call, (D)eclaration, (E)xpression, (I)nstruction, and (A)ssignment. There may be 0, 1, or 2 correct answers for each.

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

11. *Relation between syntactical categories*
    Have another look at the five syntactical categories mentioned in the previous exercise, but now add a sixth one: C(onstant).
    (a) Which of the combinations of 2 categories are possible? For example: M+D is possible if there exists a program fragment that is both a method call and a declaration.
    (b) Which of the six categories are always together with which others?
    (c) Which of the six categories always ends with a semicolon? Which sometimes? Which never?

12. *Variable assignment*
Consider the following two variable declarations and assignments:

```
int x, y;
x = 40;
y = 12;
```

Indicate for each of the following groups of instructions what the values of x and y are when these instructions are executed after the above declarations and instructions.

y = x+1; x = y+1;	x = y; y = x;	x = y+1; y = x−1;	x = x+y; y = x−y; x = x−y;	y = x/3; x = y∗3;	y = 2/3∗x; x = 2∗x/3;	y = x%6; x = x/6;

In one of the cases, the values of x and y are swapped. Does this work for all possible values of x and y? If so, why is that? If not, in what cases does it fail?

13. *Multiplying and dividing*
Is there a difference between the following three instructions?

```
position = 300 − 3∗time / 2;
position = 300 − 3/2 ∗ time;
position = 300 − time/2 ∗ 3;
```

14. *Hours, minutes, seconds*
Suppose that the integer variable time contains a (possibly large) number of seconds. Write down a number of instructions that assign a value to the variables hours, minutes and seconds corresponding to their meaning, where the values of minutes and seconds should be smaller than 60. Secondly, write the instruction that performs the reverse operation. So, given the variables hours, minutes, and seconds, calculate the value that the variable time should contain.

15. *The game loop*
Which actions does the game loop consist of? Which actions are executed only once, and which actions are executed multiple times? What is the use of these different actions?

16. *Updating and drawing*
We could put all the code from the Update method in the Draw method, and leave out the Update method altogether. Why is it still useful to have different methods?

Challenges

1. *Changing colors*
   In this challenge, we're going to modify the DiscoWorld example explained in Chap. 4.
   (a) Change the program so that the color changes from black to blue instead of from black to red.
   (b) Now modify the program so that the color changes from black to purple. Can you also modify the program so that the color changes from *purple to black*?

2. *Drawing sprites in different locations*
   As a basis for this challenge, we use the SpriteDrawing example explained in Chap. 4.
   (a) Modify the program so that three balloons are drawn: one in the top right corner of the screen, one in the bottom left corner of the screen, and one in the middle of the screen. (Hint: use the Width and Height properties part of the Texture2D class.)
   (b) Find a few nice sprites on the Internet (confetti, clowns, candy, or any other party-related things) and draw them on the screen at different positions. Search for appropriate background music and play it when the application starts.

3. *Flying balloons*
   This challenge uses the FlyingSprites example (also from Chap. 4) as a basis.
   (a) Modify the program so that the balloon flies from the top to the bottom of the screen.
   (b) Now modify the program so that the balloon flies in circles around the point that indicates the center of the screen. Use the Sin or Cos methods that are available in the Math class. Define a constant that contains the speed at which the balloon turns around the center point. Also define a constant that contains the distance from the balloon to the center point (e.g. the radius of the circle).
   (c) Change the program so that the balloon flies in circles around a point moving from the left to the right.
   (d) Change the program so that a second balloon also turns in circles around that point, but in the opposite direction, and with a bigger radius. Feel free to try a couple of other things as well such as: a balloon that flies in a circle around another flying balloon, or use another Math method to change the position of the balloon. Don't go too crazy on this or you'll end up with a headache!
   (e) Finally, add a few flying objects (you could for example use the 'spr_ball_xx' sprites for that) that bounce off the edges of the screen, by using the Sign method in the Math class.

A.2 Exercises and Challenges for Part II

Exercises

1. *Keywords*
   (a) What does the word **void** mean, and when do we need this keyword?
   (b) What does the word **int** mean, and when do we need this keyword?
   (c) What does the word **return** mean in a C# instruction, and when do we need it?
   (d) What does the word **this** mean in a C# instruction, and when do we need it? In what kind of method can we not use this word?

2. *Type conversions*
   Suppose that the following declarations have been made:

```
int x;
string s;
double d;
```

   Expand the following assignments with the necessary type conversions (assuming that the string indeed represents a number):

```
x = d;
x = s;
s = x;
s = d;
d = x;
d = s;
```

3. *Methods with a result*
   (a) Write a method RemainderAfterDivision with two parameters x and y, which returns the value of x%y, *without* using the % operator.
   (b) Write a method Circumference that gives as a result the circumference of a rectangle, whose width and height are given as parameters.
   (c) Write a method Diagonal that gives as a result the length of the diagonal of a rectangle, whose width and height are given as parameters.
   (d) Write a method ThreeTimes, that returns three concatenated copies of a string passed as a parameter. So, ThreeTimes("hurray!") should result in the string "hurray!hurray!hurray!".
   (e) Write a method SixtyTimes. that returns sixty concatenated copies of a string passed as a parameter. Try to limit the number of instructions in that method.
   (f) Write a method ManyTimes. that returns a number of concatenated copies of a string passed as a parameter, where that number is also passed as a parameter (you may assume that this number is 0 or larger). So, ManyTimes("what?", 4) should result in "what?what?what?what?".

4. *Cuneiform*
   (a) Write a method Stripes with a number as a parameter (you may assume that this parameter is 0 or larger). The method should give as a result a string with as many vertical dashes as the parameter indicates. For example, the call **this**.Stripes(5) results in "|||||".
   (b) Write a method Cuneiform with a number as a parameter. You may assume that this parameter will always be 1 or bigger. The method should give as a result a string containing the number in a cuneiform notation. In that notation, every number is represented by vertical dashes and the digits are separated by a horizontal dash. Horizontal dashes are also placed at the beginning and at the end of the string. Here are a few examples:

   - **this**.Cuneiform(25) results in "−||−|||||−"
   - **this**.Cuneiform(12345) results in "−|−||−|||−||||−|||||−"
   - **this**.Cuneiform(7) results in "−|||||||−"
   - **this**.Cuneiform(203) results in "−||−−|||−"

   Hint: deal with the last digit first and then repeat for the rest of the digits.

5. *Sequences*
   (a) Write a method Total with a number n as a parameter that returns the total of the numbers from 0 until n as a result. If n has a value smaller than or equal to 0, the method should return 0.
   (b) The factorial of a natural number is the result of the multiplication of all the numbers smaller than that number. For example, the factorial of 3 equals $1 \times 2 \times 3 = 6$. Write a method Factorial which calculates the factorial of its parameter. You may assume that the parameter always is larger than or equal to 1.
   (c) Write a method Power that has two parameters: a number x and an exponent n. The result should be x^n, so x is multiplied n times with itself. You may assume that n is a positive integer. The method should also work if n equals 0, and if x isn't an integer number.
   Hint: use a variable for calculating the result, and don't forget to give that variable an initial value!
   (Note: many programming languages have the power operator ^. C# doesn't have this operator, although there is a method Math.Pow. You are not allowed to use that method here otherwise this exercise would be too easy :)).
   (d) We can approximate the hyperbolic cosine of a real number x as follows:

$$1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \frac{x^8}{8!} + \frac{x^{10}}{10!} + \cdots$$

   In this case, the notation 6! means factorial of 6. Write a method Coshyp that calculates this approximation by summing 20 of these terms and returning that value as a result.

6. *Prime numbers*
   (a) Write a method Even which indicates whether a number passed as a parameter is an even number. Determine what the best type is for the parameter and the return value.
   (b) Write a method MultipleOfThree that indicates whether its parameter is a multiple of three.
   (c) Write a method MultipleOf with two parameters x and y, that determines if x is a multiple of y.
   (d) Write a method Divisible with two parameters x and y that determines if x is divisible by y (so x/y should have no remainder).
   (e) Write a method SmallestDivider that determines the smallest integer number ≥ 2 by which the parameter can be divided.
   Hint: try the dividers one by one, and stop as soon as you found one.
   (f) Write a method that determines if a number is a prime number. This means that it is only divisible by 1 and by itself.

7. *Drawing the memory*
   Given are the following class definitions:

```
class One
{
    int x;
    public One()
    {
        x = 0;
    }
    public void SetX(int a)
    {
        x = a;
    }
}

class Two
{
    int x;
    One o;

    public Two(One b, int c)
    {
        o = b;
        x = c+1;
    }
    public One GetO()
    {
        return o;
    }
}
```

```
class Three : One
{
    Two p, q;
    public Three()
    {
        p = new Two(new One(), 1);
        p.GetO().SetX(7);
        q = new Two(p.GetO(), 2);
        q.GetO().SetX(8);
        p = new Two(this, 3);
        p.GetO().SetX(9);
    }
}
```

Make a drawing of what the memory looks like after executing:

```
Three t = new Three();
```

Just like the examples in this book, make a clear distinction between the name and the value of the variables: the name should be next to the boxes, and the value in it. Object-references should start with a clear dot inside the box of the reference variable, and point to the border of the object.

8. *Classes and inheritance*
Given are the following class definitions:

```
class A
{
    public float var1;
    protected int var2;
    private bool var3;

    public float Var1
    {
        get { return var1; }
    }
    public int Var2
    {
        get { return var2; }
        set { if (value > 0) var2 = value; }
    }

    public void MethodInA()
    {
        ...
    }
}
```

```
class B : A
{
    public int var4;
    private int var5;

    public void MethodInB()
    {
        ...
    }
}
```

(a) Indicate if the following expressions are allowed in MethodInA:

this.var1	**this**.var2	**this**.var3
this.var4	**this**.Var2	**base**.var1

(b) Indicate if the following expressions are allowed in MethodInB:

this.var1	**this**.var2	**this**.var3
this.var5	**this**.Var2	**base**.var1
base.var2	**base**.var3	**base**.Var2

9. *Type checking*
Are the types of expressions checked during the compilation phase, or when the program is running? There is a exception to this rule. In which case is this? And why is that exception necessary?

Challenges

1. *More flying balloons*
This challenge uses the Balloon2 program from Chap. 5 as a basis.
   (a) Modify the program so that the balloon follows the mouse pointer, but with a delay. Hint: add a balloon velocity vector to the program, and change the velocity depending on the distance of the balloon to the mouse pointer.
   (b) Modify the program so that the balloon cannot fly outside of the screen. Take into account the width and height of the sprite.
   (c) Change the program so that the balloon flies in circles around a point moving from the left to the right.
   (d) Add a few flying balloons (in any direction you like) that bounce off the edges of the screen. Use the **if**-instruction to reverse their velocities.

2. *Extending the Painter game*
This challenge uses the Painter game as a basis.

(a) Extend the game, so that there is a generic 'wind' parameter that varies throughout playing the game. When there is more wind, the paint cans and balloon rotate and move more heavily. If you define this 'wind' parameter as a vector, it can also be used to indicate the direction of the wind. The direction and strength of the wind then influences how the balloons move, but also how fast the ball moves.

(b) Make a two-player version of the game, where each player controls his/her own cannon (think of a good input mechanism for each player). You could make it a symmetrical game in that the players collaborate to give the paint cans the right color, and the player that colors the most paint can correctly wins the game. You could also make it an *asymmetrical* game, where the goal of one player is to correctly paint as many cans as possible, and the other player's goal is to obstruct the first player as much as possible.

A.3 Exercises and Challenges for Part III

Exercises

1. *Arrays*

(a) Write a method CountZeros which has an array of integers as a parameter. The result of the method is the number of zeros in the array.

(b) Write a method Add which has two integer arrays as parameters. You may assume that the length of each array is the same. The method adds the values in the two arrays and returns as a result another integer array. For example, given is an array array1 = {0, 3, 8, −4} and an array array2 = {10, 2, −8, 8}. The result of the method call add(array1, array2) will be another array {10, 5, 0, 4}.

(c) Write a method FirstPosition which has two parameters: a **string** and a **char**. The method returns the first position of the character in the string. If the character doesn't occur in the string, the method should return −1. For example:

```
int result = this.FirstPosition("arjan", 'a'); // returns 0
result = this.FirstPosition("jeroen", 'a'); // return −1
result = this.FirstPosition("mark", 'a'); // returns 1
```

2. ***string*** *methods*

(a) The **string** class contains the method ToUpper that returns a version of the string containing only capital letters. You can call this method as follows:

```
h = s.ToUpper();
```

If you had to write the class **string** yourself, how would you implement this method (using other methods in the **string** class?

(b) Another method in the **string** class is the Replace method. This method returns a new string in which each character that corresponds to the character given as the first parameter is replaced by the character given as the second parameter. For example:

```
"Good morning".replace('o','u') // returns "Guud murning"
"A+2+#?".replace('+','9') // returns "A929#?"
```

Implement the Replace method (using other methods of the **string** class where necessary).

(c) Now let's have a look at the method EndsWith, which returns whether a string ends with the string passed as a parameter. For example:

```
"Allyourbasearebelongtous".endsWith("belongtous") // returns true
```

Implement the EndsWith method. You may use existing methods in the **string** class, except for the Substring and IndexOf methods.

3. *Highway*
Have a look at the program below. The program should draw a line of cars on the highway, as can be seen from the screenshot below. Every third car is a truck, and every second truck is a combination with a trailer.

```
public class Highway : Game
{
    public SpriteBatch spriteBatch;
    public Texture2D wheel, car, truck, connection, trailer;

    static void Main()
    {
        Highway game = new Highway();
        game.Run();
    }

    public Highway()
    {
        graphics = new GraphicsDeviceManager(this);
        graphics.PreferredBackBufferWidth = 1800;
        graphics.PreferredBackBufferHeight = 80;
        Content.RootDirectory = "Content";
        // TODO: missing part of the constructor
    }
```

```
    protected override void LoadContent()
    {
        spriteBatch = new SpriteBatch(this.GraphicsDevice);
        wheel = Content.Load<Texture2D>("wheel");
        car = Content.Load<Texture2D>("car");
        truck = Content.Load<Texture2D>("truck");
        connection = Content.Load<Texture2D>("connection");
        trailer = Content.Load<Texture2D>("trailer");
    }

    protected override void Draw(GameTime gameTime)
    {
        GraphicsDevice.Clear(Color.Grey);
        spriteBatch.Begin();
        for (int t = 0; t < road.Length; t++)
            road[t].Draw(this, t*120, 60);
        spriteBatch.End();
    }
}

class MotorizedVehicle
{
    public void Draw(Game g, int x, int y)
    {
    }
}

class Car : MotorizedVehicle
{
    public void Draw(Game g, int x, int y)
    {
        g.spriteBatch.Draw(g.car, new Vector2(x, y - 30), Color.Blue);
        g.spriteBatch.Draw(g.wheel, new Vector2(x + 5, y - 10), Color.Red);
        g.spriteBatch.Draw(g.wheel, new Vector2(x + 25, y - 10), Color.Red);
    }
}

class Truck : MotorizedVehicle
{
    public void Draw(Game g, int x, int y)
    {
        g.spriteBatch.Draw(g.truck, new Vector2(x, y - 45), Color.Green);
        g.spriteBatch.Draw(g.wheel, new Vector2(x + 5, y - 10), Color.Red);
        g.spriteBatch.Draw(g.wheel, new Vector2(x + 20, y - 10), Color.Red);
        g.spriteBatch.Draw(g.wheel, new Vector2(x + 55, y - 10), Color.Red);
    }
}
```

```
class Combination : Truck
{
    public void Draw(Game g, int x, int y)
    {
        // the truck
        g.spriteBatch.Draw(g.truck, new Vector2(x, y − 45), Color.Green);
        g.spriteBatch.Draw(g.wheel, new Vector2(x + 5, y − 10), Color.Red);
        g.spriteBatch.Draw(g.wheel, new Vector2(x + 20, y − 10), Color.Red);
        g.spriteBatch.Draw(g.wheel, new Vector2(x + 55, y − 10), Color.Red);
        // the trailer
        g.spriteBatch.Draw(g.connection, new Vector2(x − 5, y − 10), Color.Black);
        g.spriteBatch.Draw(g.trailer, new Vector2(x − 45, y − 45), Color.Green);
        g.spriteBatch.Draw(g.wheel, new Vector2(x − 40, y − 10), Color.Red);
        g.spriteBatch.Draw(g.wheel, new Vector2(x − 20, y − 10), Color.Red);
    }
}
```

(a) One declaration is still missing. Write down this declaration and indicate where it should be placed in the program.
(b) Write down the missing part of the Highway constructor.
(c) Because the program still contains an error, nothing is drawn on the screen. What error is this, and how can it be corrected?
(d) The programmer has duplicated quite some code with copy and paste. Why is that not a good idea?
(e) How could this code copying have been avoided?
(f) We would like to continue improving the object-oriented approach of this program, so that also the 'wheel' and 'trailer' concepts are modeled with classes. How can that be done properly? Make sure that code duplication is avoided as much as possible, and that a loose trailer can never end up on the highway because of a programming error. You don't have to completely implement this, just indicate which extra classes are needed, how they are related to their subclasses and which declarations are needed in new or existing classes.

4. *Searching and sorting*
   (a) Write a method Largest which has as a parameter an array of doubles that gives as a result the *largest value* that occurs in the array.
   (b) Write another method IndexLargest which has as a result the index in the array of the largest value. If there are more than one largest values in the array, the method should return the index of the first one.
   (c) Write a method HowManySmallest which has as a parameter an array of doubles. The method should return how often the smallest value of the array occurs in it. For example, if the array contains the values 9, 12, 9, 7, 12, 7, 8, 25, 7, then the result of the method call will be 3 because the smallest value (7) occurs three times in the array.

(d) Write a special version of the IndexLargest method which doesn't look through the entire array, but only the first n elements, where n is passed as a parameter to the method.

(e) Write a method Sort, which has as a parameter an array of doubles. The return value type is **void**. After the method is called, the elements in the array should be sorted in increasing order.
Hint: first find the index of the largest value in the array (using the IndexLargest method). Then, swap this value with the value at the last index. After doing that, the largest value is already and the end of the array where it belongs. Then, find the largest value in the rest of the array and swap that value with the penultimate value in the array. And so on, until the entire array has been sorted.

(f) Write a method which has as parameters an array of integers and one single integer that returns as a result the index at which the single integer occurs first in the array. If the integer doesn't occur anywhere in the array, the method should return -1.

(g) (*more difficult*) If you know that the array is sorted, it is possible to search in a smarter way: look at the middle element in the array if it contains the value that you're looking for. If so, great! If not, you'll know if the element you're looking for is in the left part or in the right part of the array. This way, looking through the array becomes very efficient because at each iteration the part of the array you have to search through becomes half as big. Write this improved search method. Use two integers to keep track of the boundaries of the piece of array that you're searching in.

Challenges

1. *Weather control*
In this challenge, we will extend the Snowflakes program from Chap. 12.
   (a) Extend the program by allowing for more or less snowflakes depending on user input (up arrow means more snow, down arrow means less snow). Set a maximum number of snowflakes (for example 1000). Make sure that the program is robust (never more than 1000 snowflakes, and no crash when you reach 0 snowflakes).
   (b) Extend the program so that there is varying wind. You can implement 'wind' by not letting the snowflakes fall down straight but according to an angle. Vary the angle with which the flakes fall down over time using random numbers. Try to find a range of random values and a rate of change that looks realistic.

2. *More jewels*
This challenge is an extension of the Jewel Jam game.

(a) Extend the game so that sometimes all the colors of the jewels change. This can be something controlled by the player (if he/she is stuck), or it can be done randomly.
(b) When a valid combination of three jewels has been found, add a nice 'disappearing' effect for the jewels, such as making them smaller and smaller, or flying them outside of the screen in random directions.
(c) Add a feature where there is an extra restriction in some cases, such as only combinations allowed of the same color, or some jewels are not allowed in any combination.

3. *Tic-tac-toe*
Another nice example of a grid-based game is Tic-tac-toe. In this challenge, we'll go step-by-step through the important parts of building that game.
(a) Start by looking up some nice sprites that you can use for this game. In any case, you are going to need a sprite for the cross and a sprite for the circle. Once you've found these sprites, start by making a first version which only displays circles or crosses on a 3×3 grid on the screen. Here, you can reuse some of the classes that we built for the Jewel Jam game.
(b) Next, add player input to the game. When the player clicks in the screen, a cross should appear at that position in the grid. Make sure that you only add a cross to the grid if the position in the grid is free. In the first version, you can make it a two-player game, where each player takes a turn after the other by clicking somewhere. Therefore, you should keep track of whose turn it is, so that you can add the right element (cross or circle) to the grid.
(c) Now, you need to write the code that handles whether three symbols of the same type are in a row, column or diagonal. You can do this by using a couple of **for**-instructions that check each of the possible combinations.
(d) Finally, handle the different game states: indicate on the screen whose turn it is, show an overlay if the game has been finished, and add a restart option.
(e) (*difficult*) As a nice extension, you could program a single player mode, where the player has to play against the computer. The question is: how do you program the part where the computer decides what to do? A very well-known common way of implementing computer behavior in such games is by using the *Minimax* algorithm. The basic idea behind this algorithm is to represent all the possible ways that the game can be played as a tree, and then search through that tree to find an optimal path. The nodes in each tree represent a choice that is made by the player or by the computer, and the edges determine the sequence in which these choices are made.
In the case of tic-tac-toe, you can imagine the tree as follows. After starting the game, the first player has 9 possible locations to choose from. So from the root node there will be nine outgoing edges. After that, the second player can choose between the remaining 8 locations. So from each of the 9 nodes, there will be 8 outgoing edges. From each of these 8 sub nodes, there will be 7 outgoing edges, and so on, and so on. This will result in a huge tree containing all the possible outcomes of the game. For each path in this tree,

you can see immediately who has won (player 1 or player 2). So what you can do then, is to assign a cost or a reward value to each of the edges, for example $+10$ for an edge that helps leading to your victory, -10 for an edge that leads to your opponent's victory, and 0 for an edge that leads to a draw. Then, after the real player has made a choice, the computer can walk through the tree and find the path that minimizes the loss for the worst case (maximum loss) scenario (which also explains the name of the algorithm).
The Minimax algorithm is a classic AI algorithm that is used in many games where the computer has to make a choice between a number of alternatives and where there are discrete turns with a predetermined outcome. Many board games are suitable for this, such as chess, checkers, and so on. This algorithm is not always useful, since it means that you have to (partly) construct the tree of possible decisions and their outcomes, which can get quite complicated depending on the type of game. For example in chess, the number of different states is estimated at 10^{43} and the number of different nodes in the game tree at 10^{120}!

A.4 Exercises and Challenges for Part IV

Exercises

1. *Lists*
   The List<**string**> class contains (among others) methods that have the following headers:

   ```
   public void Reverse()
   public int LastIndexOf(string item)
   public bool Contains(string item)
   ```

   Suppose that you are the author of the List class and these methods have not yet been implemented. Implement these three methods. You may not use existing methods carrying the same name, or other varieties of these methods with these names, because we assume that these methods are not there yet. Apart from these methods, you may use any other method in the List class, as well as your own methods.

2. *Collections*
   Write a method RemoveDuplicates which receives as a parameter a List of **int** values. The method removes all of the duplicates of the numbers in the list that is provided as a parameter. For example, the list containing the numbers 0, 1, 3, 2, 1, 5, 2 becomes 0, 1, 3, 2, 5. Watch out: the return type of this method should be **void**!

3. *Classes and interfaces*
One of the following three combined declarations and assignments is correct. Which one is that, and why are the other two not correct?

```
List<int> a = new IList<int>(); // version 1
IList<int> b = new List<int>(); // version 2
IList<int> c = new IList<int>(); // version 3
```

Describe a situation in which the correct version of the above lines of code has an advantage over the also correct declaration and assignment:

```
List<int> d = new List<int>(); // version 4
```

4. *List and* ***foreach***
Provided is the following class:

```
class Counter
{
    int val = 0;
    public Counter(int v)
    { val = v;
    }
    public void Increment()
    {
        val++;
    }
}
```

and the following instructions in a class that uses the Counter class:

```
List<Counter> list = new List<Counter>();
for (int i=0; i<25; i++)
    list.Add(new Counter(i));
```

(a) Write a method Increment, which gets an IList<Counter> object as a parameter and which increments all the counters (using the Increment method in the Counter class). Provide a version that uses the **foreach** instruction, as well as a version that uses a **for** or a **while** instruction.
(b) Suppose that instead of passing a List<Counter> object, we would like to pass an object of type OwnList<Counter>, where the OwnList class is our own version of a list, which implements the IList interface. What do we need to change in the Increment method we wrote in the previous question so that we can use this method with our own list class?

5. *Strings*
The **string** class contains (among others) methods that have the following headers:

```
public string Substring(int startIndex, int length);
public string Substring(int startIndex);
public int IndexOf(char c)
```

Suppose that you are the author of the **string** class and these methods have not yet been implemented. Implement these three methods. You may not use any existing methods carrying the same name, or other varieties of these methods with these names, because we assume that these methods are not there yet. Apart from these methods, you may use any other method in the **string** class, as well as your own methods.

6. *Classes and inheritance*
   Consider the following two classes:

```
class A {
    public void Method1() { Console.WriteLine("A::Method1"); }
    public virtual void Method2() { Console.WriteLine("A::Method2"); }
}
class B : A {
    public void Method1() { Console.WriteLine("B::Method1"); }
    public override void Method2() { Console.WriteLine("B::Method2"); }
}
```

   What is the output of the following series of instructions?

```
A x = new A();
A y = new B();
B z = new B();
x.Method1();
x.Method2();
y.Method1();
y.Method2();
z.Method1();
z.Method2();
```

7. *Sidescrolling*
   Consider the following class:

```
class Scrolling : Game
{ GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    Texture2D background;
    Vector2 position;

    public Scrolling()
    { graphics = new GraphicsDeviceManager(this);
```

```
        Content.RootDirectory = "Content";
        position = Vector2.Zero;
    }

    protected override void LoadContent()
    { spriteBatch = new SpriteBatch(GraphicsDevice);
        background = Content.Load<Texture2D>("background");
    }

    protected override void Draw(GameTime gameTime)
    { GraphicsDevice.Clear(Color.White);
        spriteBatch.Begin();
        spriteBatch.Draw(this.background, this.position, Color.White);
        spriteBatch.End();
    }
}
```

The background sprite has the same height as the screen, but it is a lot wider. As a result, we can only draw a part of the sprite. The goal of this exercise is to implement side scrolling. We do this by using the mouse position. If the mouse is positioned left to the screen, the background moves to the right. If the mouse is positioned right to the screen, the background moves to the left. Write the Update method that results in this behavior.

8. *Decorator streams*
The Stream class has (among others) the following methods:

```
int ReadByte(); // returns the next byte, or −1 if there is no more byte
int Read(byte[] goal, int n); /∗ reads a maximum of n bytes and puts them in goal,
                                 and returns the number of bytes read. ∗/
```

In many cases it is more efficient to read an entire block with the Read method. On the other hand, it's easier to read a separate byte whenever you need it.
The class BufferedStream can help out here. The constructor method of these class has a Stream object as a parameter. When the ReadByte method is called on an object of type BufferedStream, it uses the Stream object it manages to read 1000 bytes. It then returns the first one it read, the rest is saved temporarily in an array (in other words, a buffer). The next time the ReadByte method is called, the byte can be retrieved from the array and we don't have to access the underlying file. Only when there are no more bytes left in the array, a new block of 1000 bytes is read.
Implement the BufferedReader class, containing a constructor method and the ReadByte method.

Challenges

1. *Penguin Pairs*
   In this challenge, we will extend the Penguin Pairs game.
   (a) Currently, the level menu consists of a single page displaying a maximum of fifteen levels. Extend the game such that multiple pages of levels are allowed. Add two buttons to the level menu screen to be able to navigate through the different pages.
   (b) Extend the game so that it also contains polar bears that can be moved by the player. Whenever a penguin collides with a polar bear, the penguin is so scared that it immediately starts moving away from the bear. Design a few different levels around this concept.
   (c) Add a 'hole' object to the game. Whenever the penguin goes in the hole, it appears from another hole elsewhere in the level and continues moving in the same direction. Think of a smart way to represent this in the text file. Especially, how do you represent the connections between the holes? Create a couple of levels that use holes.
   (d) Add a 'curve in the ice' item to the game that changes the direction of the penguin if it moves over it. For example, if a penguin is moving up through a curve object, and the curve is of the type 'turn left', the penguin will change direction so that it continues moving to the left. Add a few more levels to the game that use this item.
   (e) Introduce a conveyor belt, which moves the penguin a fixed number of steps in a certain direction. The penguin resumes in the original direction after it.

A.5 Exercises and Challenges for Part V

Exercises

1. *Text files and collections*
   Write a (console) application with the following specification. The program is started from the command line and the user specifies two filenames. The program reads the file corresponding to the first filename. Then it writes a text file with as a name the second filename. The output file will contain the text of the first file, but every word should be placed on a separate line. Also, the words will be ordered alphabetically (or actually: according to the Unicode order). Each word should be written to the file only once.
   We consider every group of characters without a space between them as a word. You may assume that there is exactly one space between each word in the input file.
   If the user provides too few or too many filenames, or there is an error during reading or writing, the program reports this to the user.

For example, if the input file contains the following two lines:

```
this IS an *%#$ example
of an input text file!
```

then the output should contain the following nine lines:

```
*%#$
IS
an
example
file!
input
of
text
this
```

In case this is too complicated, you can simplify the question as follows: use fixed file names instead of names specified by the user; leave out the sorting and the duplicate word removal; do not process the words, but the lines of the text; leave out reporting the error.

2. *Abstract classes and interfaces*
   What is the difference between an abstract class and an interface? Give an example of a situation in which you would use an abstract class. Give another example of a situation in which you would use an interface.

3. *Tetris blocks*
   Suppose that you are working on creating a Tetris game. In that game, a tetris block is represented by a two-dimensional array of boolean values. The first dimension indicates the column, the second dimension indicates the row. For example, here are two different tetris blocks:

```
true  true  false          false true  false false
false true  true           false true  false false
false false false          false true  false false
                           false true  false false
```

   (a) Write a method HorizontalMirror which receives as a parameter a two dimensional array of booleans. The goal of the method is that the values of the array are mirrored horizontally. After calling the method, the two blocks given as an example above will appear as follows:

```
false true  true           false false true  false
true  true  false          false false true  false
false false false          false false true  false
                           false false true  false
```

   For example, you could use this method as follows:

```
bool[,] tetrisblock = ...
this.Print(tetrisblock);
this.HorizontalMirror(tetrisblock);
this.Print(tetrisblock)
```

(b) Also implement the Print method. This method writes the contents of a two-dimensional boolean array to the console, in the way it is done in the examples given in this exercise.

4. *Abstract classes*
Given are the following classes:

```
abstract class A
{
    public abstract void Method1();
    public void Method2()
    {
        return;
    }
}
class B : A
{
    public override void Method1()
    {
        return;
    }

    public void Method3(A a)
    {
        a.Method1();
    }
}
```

Indicate for each of the following instructions whether or not it is allowed:

```
A obj;
obj = new A();
obj = new B();
obj.Method1();
obj.Method2();
obj.Method3(obj);
B otherObject = (B)(new A());
A yetAnotherObject = (A)obj;
obj.Method3(otherObject);
A[] list;
list = new A[10];
list[0] = new A();
list[1] = new B();
List<A> otherList = new List<A>();
```

Challenges

1. *Adding side scrolling to Tick Tick*
   If you look at the background sky image, you see that it is quite a bit larger than the actual screen size. One of the reasons for this is that we can use the sky image as a moving background for side scrolling. The goal of this challenge is to add side scrolling capabilities to the Tick Tick game.
   (a) The first thing you will have to do is extend the game environment framework so that you can define a *camera*. Using a virtual camera, we can specify which part of the game world we're seeing. What happens when we're side scrolling is that we move around this camera so that different parts of the world are seen. Add a Camera class to the GameManagement library that decides which part of the world we're currently seeing. In order to test your camera, try to run the program with a few different values for the camera position. Make sure that the camera works correctly in full-screen mode as well.
   (b) The next step is extending the TickTick... game so that we can read levels of any number of dimensions. Extend the game so that this works without problems.
   (c) Finally, we want to move the camera around based on the character position. There are a few different possibilities. Either you can always try to have the character in the middle of the screen, or you can move the camera when the character crosses a certain boundary, for example, past two-thirds of the visible screen. Add the automatically moving camera behavior to the game. Make sure that it is robust, and that the player never sees beyond the edges of the defined game world. For example: if the character falls out or jumps above the screen, the camera shouldn't move along with it.
   (d) Parallax scrolling is a way to create the illusion that there is a three dimensional world. You can achieve this effect by having several layers of mountains. The further away a layer of mountains is, the slower it will move when the viewport is moving. Add parallax scrolling to the game by introducing three different layers of mountains, each moving at a different speed. Of course, you don't have to use only mountains. If you find nice other sprites that could serve as a parallax layer, go right ahead.
   (e) Since our levels are now much bigger, extend the game so that for each level, you can define how much time the player has in the text file. Create a few different levels that use this ability.

2. *Other additions to Tick Tick*
   (a) Extend the Rocket class so that when the player jumps on the rocket, the rocket dies.
   (b) Add shooting behavior to the player. For example, the bomb could throw smaller bombs which would kill an enemy if it collides with the bomb.
   (c) Add a health indicator for the player. Every time the player touches an enemy, or if the player falls down from more than three tiles high, the health

is reduced with a certain amount. If the health reaches zero, the player dies. Add health packs to the game, so that the player can restore part of his/her health again.

(d) Add an item to the game that shields the player for a while so that her/his health is reduced less or not at all.

(e) Try to make some of the enemies smarter. For example, can you add behavior to the flames so that they can jump from one platform to another? Or can you make the rockets smarter so that they sometimes follow the player?

(f) Introduce a new type of tile: the moving tile. This is not an easy extension, because you have to make sure that if the player stands on the tile, he/she moves along with it. Also, you have to take into account that multiple moving tiles might appear next to each other to form a single moving platform. When a moving platform collides with another moving platform or a wall tile, it turns around and starts moving in the other direction.

(g) Add hidden levels to the game. By going to a particular place in the level, you can enter a hidden level. Think of a good class design where these hidden levels fit in. Of course, hidden levels should also be read from a file, just like the normal levels.

(h) Add a type of object to the game that results in the player walking much faster or slower for a while.

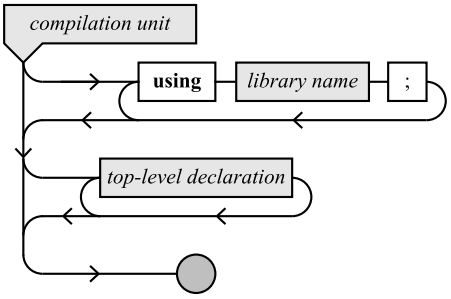
(i) Games often contain extras that don't really add anything to the gameplay, but that make the game a lot more fun. An example of such an extra is that a character says something ridiculous when you click on it. Even more fun is if the character says something different every time. Extend the game such that when you click on the player, the character says something funny. Write a class that allows to select a random sound and play it, so that there is more variety. You can use existing sound fragments, but of course you can also go crazy and record sounds yourself with a microphone.

Appendix B
Syntax Diagrams

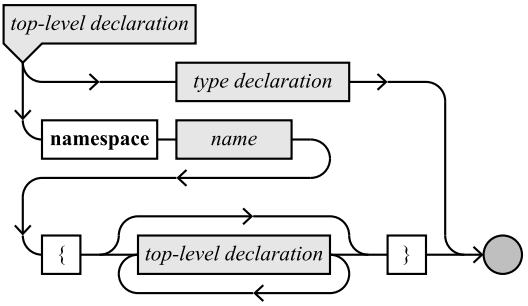
B.1 Introduction

In this appendix, we list a number of syntax diagrams that show how to create the most important C# constructs that are introduced in this book. Note that this is by no means a *complete* grammar of the C# language. It simply serves as an aid in constructing syntactically correct C# programs.

B.2 Compilation Unit

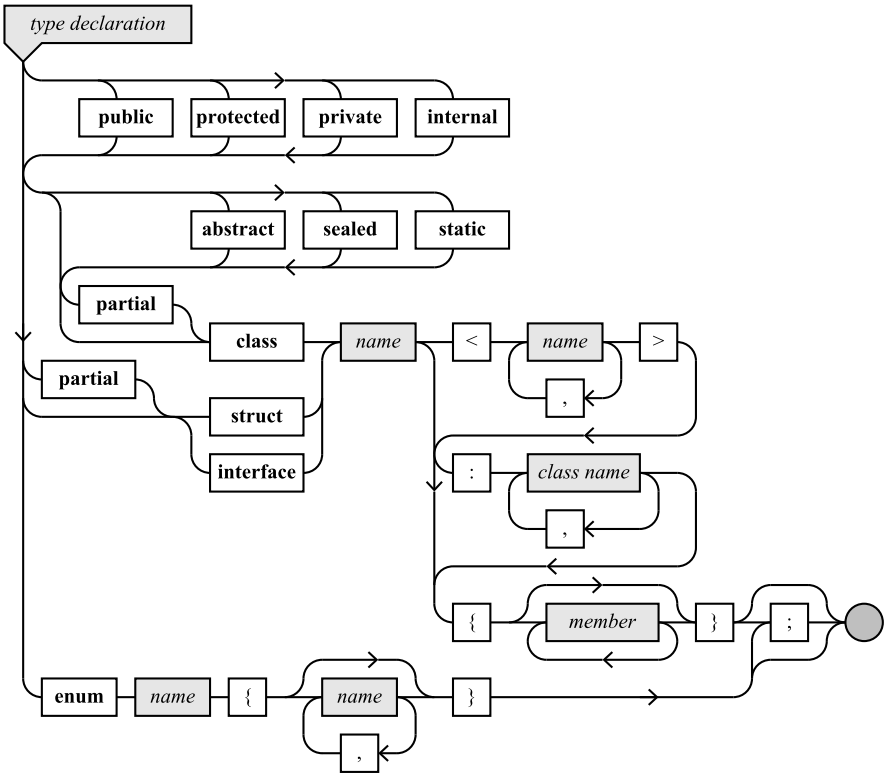


B.3 Top-Level Declaration

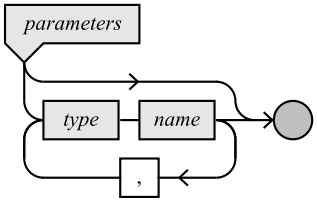


A. Egges et al., *Learning C# by Programming Games*,
DOI 10.1007/978-3-642-36580-5, © Springer-Verlag Berlin Heidelberg 2013

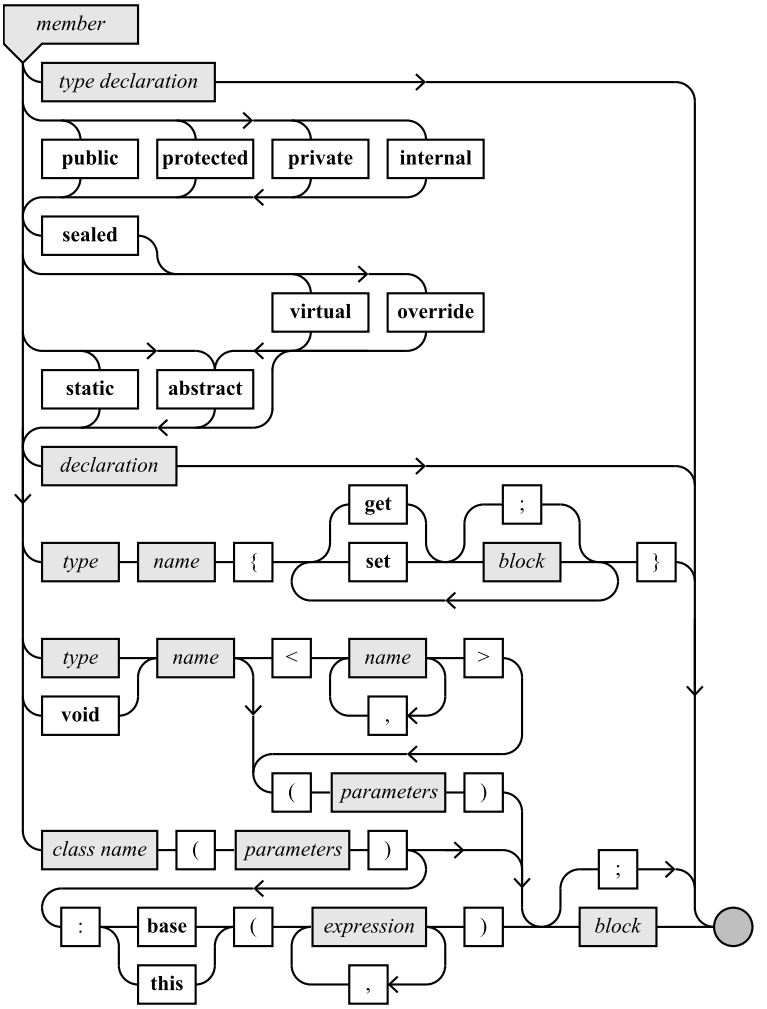
B.4 Type Declaration



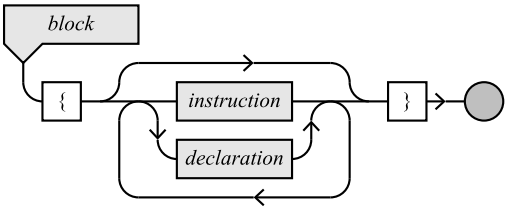
B.5 Parameters



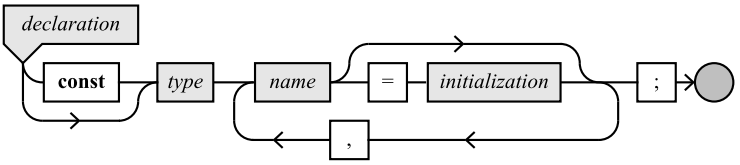
B.6 Member



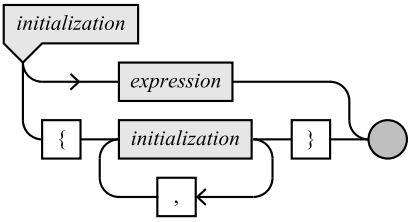
B.7 Block



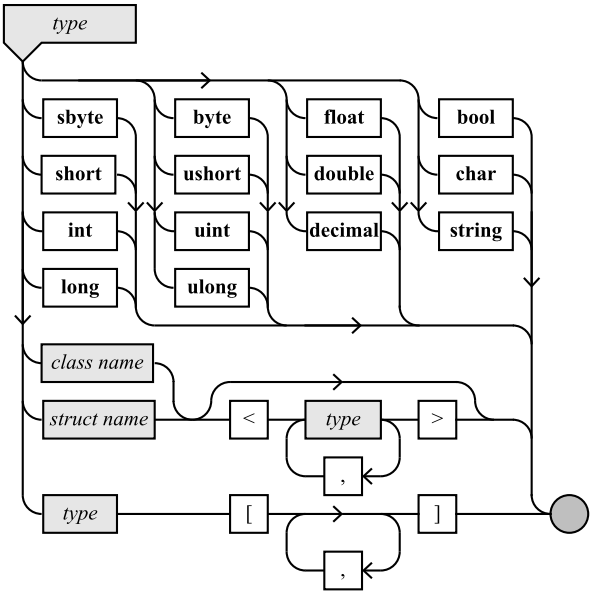
B.8 Declaration



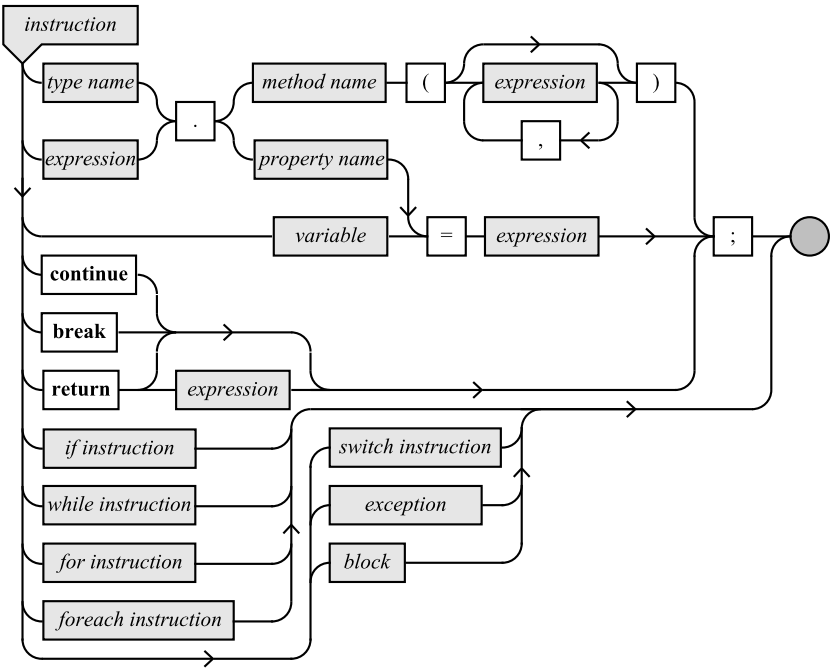
B.9 Initialization



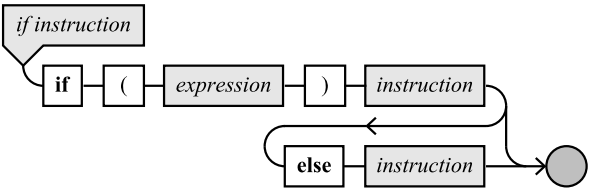
B.10 Type



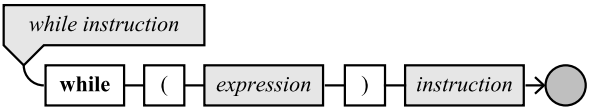
B.11 Instruction



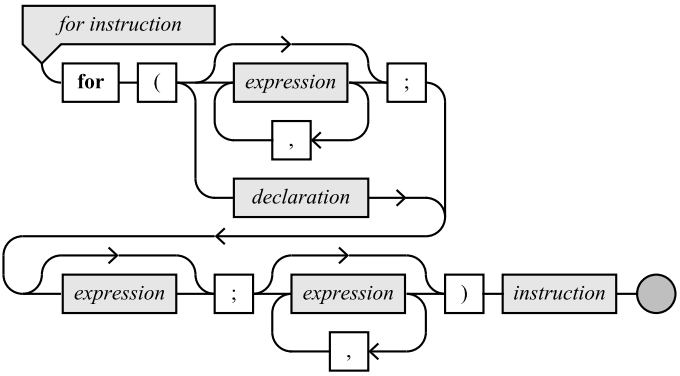
B.11.1 If-Instruction



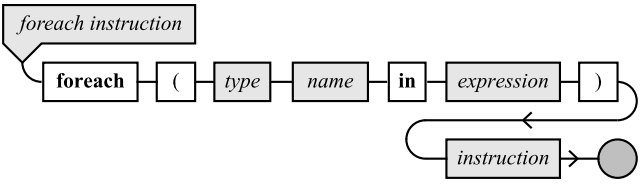
B.11.2 While-Instruction



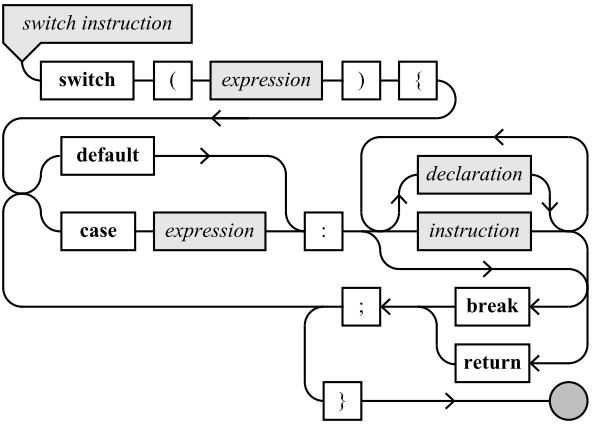
B.11.3 For-Instruction



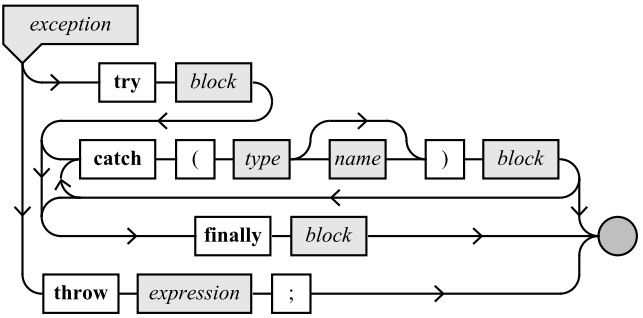
B.11.4 Foreach-Instruction



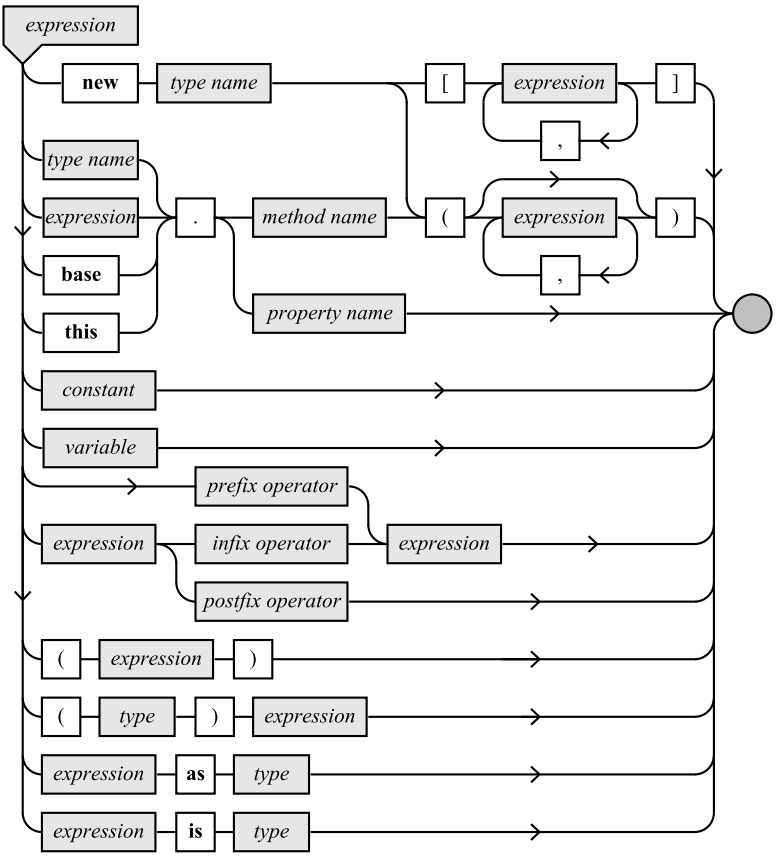
B.11.5 Switch-Instruction



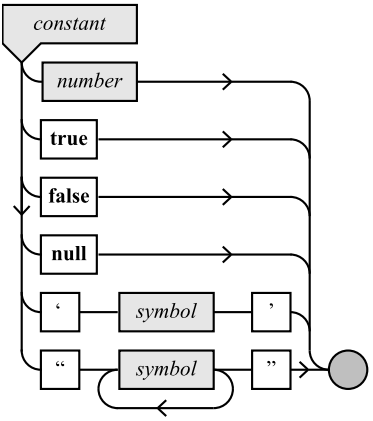
B.11.6 Try-Catch-Instruction



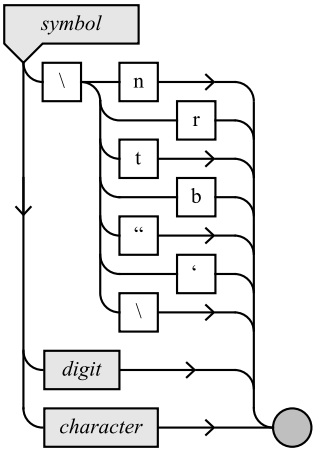
B.12 Expression



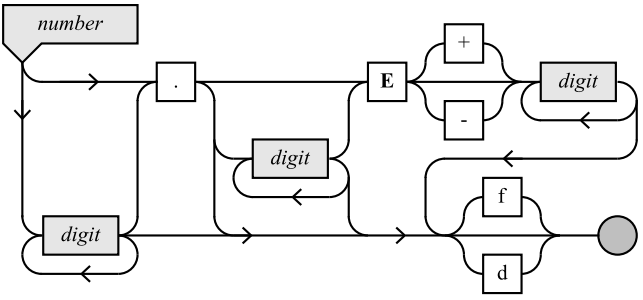
B.13 Constant



B.14 Symbol



B.15 Number



Further Reading

Ernest Adams. *Fundamentals of Game Design*. New Riders, second edition, 2010. ISBN 978-0321643377.

Paul J. Deitel and Harvey M. Deitel. *C# 2010 for Programmers*. Pearson Education, fourth edition, 2011. ISBN 978-0132618205.

Barbara Doyle. *C# Programming: From Problem Analysis to Program Design*. Cengage Learning, third edition, 2010. ISBN 978-0538453028.

Rob Miles. *Microsoft XNA Game Studio 4.0: Learn Programming Now!* Microsoft Press, second edition, 2011. ISBN 978-0735651579.

Tom Miller and Dean Johnson. *XNA Game Studio 4.0 Programming: Developing for Windows Phone 7 and Xbox 360*. Pearson Education, 2011. ISBN 978-0672333453.

Joel Murach. *Murach's C# 2010*. Mike Murach & Associates, 2010. ISBN 978-1890774592.

Benjamin Nitschke. *Professional XNA Programming: Building Games for Xbox 360 and Windows with XNA Game Studio 2.0*. Wiley Publishing, 2008. ISBN 0470261285.

Jeannie Novak. *Game Development Essentials: An Introduction*. Cengage Learning, third edition, 2012. ISBN 978-1111307653.

Nick Randolph, David Gardner, Michael Minutillo, and Chris Anderson. *Professional Visual Studio 2010*. Wiley Publishing, 2010. ISBN 978-0470548653.

Aaron Reed. *Learning XNA 4.0: Game Development for the PC, Xbox 360, and Windows Phone 7*. O'Reilly Media, 2011. ISBN 978-1449394622.

Scott Rogers. *Level Up!: The Guide to Great Video Game Design*. Wiley Publishing, 2010. ISBN 978-0470688670.

Daniel Schuller. *C# Game Programming: For Serious Game Creation*. Cengage Learning, 2011. ISBN 978-1435455566.

A. Egges et al., *Learning C# by Programming Games*,
DOI 10.1007/978-3-642-36580-5, © Springer-Verlag Berlin Heidelberg 2013

Jon Skeet. *C# in Depth*. Manning Publications, second edition, 2011. ISBN 978-1935182474.

Rod Stephens. *Stephens' C# Programming with Visual Studio 2010 24-Hour Trainer*. Wiley Publishing, 2010. ISBN 978-0470596906.

Glossary

abstract class class containing abstract methods, intended as base class for subclasses but not for creating instances, 303

abstract method method without body that must be redefined in a subclass, 304

animation rapid display of a sequence of images to create an illusion of movement, 351

app program that can be run on a smartphone, 32

applet program that can be run inside a web browser, 32

application program that can be run, 31

array (object containing a) numbered row of values of the same type, 190

asset resource used for developing a game, such as a sprite or a sound effect, 55

assignment instruction to change the value of a variable, 44

base class class that a class inherits from, 152

bit unit of memory in which two different values can be stored, 48

bitmap description of a picture consisting small colored dots called pixels (the word is imprecise, as with a bit only a black-or-white pixel can be designated; an alternative is "pixmap"), 261

block group of instructions and declarations that can be treated as a single instruction, 88

bool (type of a) logical value, i.e. either false or true, 84

bounding box two-dimensional box that encompasses an object, 365

bounding circle circle that encompasses an object, 365

branch conditional execution of an instruction, e.g. in an **if** or a **switch** instruction (named after the bifurcation that occurs in a diagram showing the flow of execution), 82

A. Egges et al., *Learning C# by Programming Games*,
DOI 10.1007/978-3-642-36580-5, © Springer-Verlag Berlin Heidelberg 2013

byte (type of a) value which is an integer number in the range $-128 \ldots 127$; or: unit of memory in which 256 different values can be stored, 48

cast forcing an expression to have a different, though related type (a more restricted numeric type, or a subclass), 49

char type of a) value that represents an Unicode character, 175

child class see subclass, 153

class group of declarations, methods, and properties, that serves as the type of an object, 33

code program text, 19

collision intersection between bounding volumes, 365

compiler program that checks and translates a program from source code to executable code, 18

condition expression that yields either **true** or **false**, 82

constructor method method, having the same name as its class, that is automatically called upon creation of a new instance, 104

declaration program fragment that introduces a name for a class, method, variable etc., 44

derived class see subclass, 153

double (type of a) floating-point value in double precision, 47

enum type of values belonging to an explicitly enumerated set, 81

exception abnormal event that may occur during program execution, that can be thrown when it occurs, and be catched to handle it, 346

executable code the program in its form as translated by the compiler, 23

expression program fragment that has a value, 45

false the one of the two bool values denoting falsehood, 84

float (type of a) value which is a number containing a floating point, 48

game program that gives its user a playful experience, 32

game engine collection of classes for governing commonly used structures and tasks in games, 22

game loop fundamental loop in a game that repeatedly calls the Update and Draw methods, 28

game state description of objects in the game at a particular in time; examples of game states are a level finished state or a game over state, 249

implementation class that complies to the methods specified in an interface; or: program that fulfills its specification, 25

infix operator operator that is written between two expressions (e.g., arithmetical, comparison, or logical), 46

inherit have access to members that were defined in a base class, 152

inheritance the fact that a subclass implicitly also declares the members of its base class, allowing to reuse code without copying it, 152

initialization assigning the initial value to a variable, 44
instance of a class object having the class as type, 73
instruction program fragment that changes memory in some way, 12
int (type of a) value which is an integer number, 44
interface group of method headers that specifies the methods that are needed in classes implementing it; or: the way a program interacts with a user, 186
interpreter program that checks a program from source code and executes it, 18
iteration repeated execution of an instruction, e.g. in a **while**, **for** or **foreach** instruction; or: one of the steps in an iteration, 142
iterator (type of a) value that can keep track of an iteration, 194
level part of a game progression that stands on its own, 297
library pre-defined classes that can be used in a program, 35
local declaration declaration of a variable that can only be used in the block that it is declared in, 158
loop repeated execution of an instruction, e.g. in a **while**, **for** or **foreach** (named after the cycle that occurs in a diagram showing the flow of execution), 142
member variable, method, or property belonging to an object, 76
member variable variable belonging to an object, 76
method group of instructions, with a name, that can manipulate an object, 14
method call instruction to execute the instructions in the body of the method, 36
namespace group of classes that can refer to each other without explicitly qualifying the library that they come from, 34
null the value of a reference when it is not referring to a particular object, 122
object group of variables, having a class as its type, 71
operator symbol that combines (one or) two expressions, 46
override a method redefine a method in a subclass, 157
parameter declaration in the header of a method that specifies the type of values that need to be passed when the method is called; or: expression that evaluates to the value that is passed in the method call, 36
parent class see base class, 152
partial class class of which the members are declared in more than one top-level declaration, 343
pixel picture element, a single dot in a picture described by a bitmap, 261
polymorphism the ability to process objects differently depending on their class type, 166
postfix operator operator that is written after an expression (e.g., increment, or decrement), 46

prefix operator operator that is written before an expression (e.g., minus, not, increment, or decrement), 46
private declaration declaration of a member that can be used only from the class that it is declared in, 158
property value related to an object that you can get and/or set, 39
protected declaration declaration of a member that can be used only from the class that it is declared in and its subclasses, 158
public declaration declaration of a member that can be used from within all other classes, 158
qualify specify the library that a class comes from, 35
recursion defining a method or property in terms of itself, 235
reference value with class type that refers to the actual instance of the class, 120
return value value that a method delivers after its execution, 97
rgb description of a color by its red, green, and blue components, 52
scope part of the program in which a declared name can be used, 53
sealed method overridden virtual method that cannot be redefined again in subclasses, 166
semantics rules that describe the meaning of programs in a language, 33
serious game game that can be used for training of professionals, 179
source code the text of the program as written by the programmer, i.e. before compilation, 19
sprite a two-dimensional image that is integrated into a larger scene, 55
static method group of instructions with a name that does not manipulate a particular object, 72
static variable member variable that is shared by all instances of the class, 117
string (type of a) value that represents a text, 176
struct type of objects that are accessed directly, i.e. without references, 123
subclass a class that inherits from another class, 153
super class see base class, 152
syntax rules that describe the form of programs in a language, 33
tile basic element of a two-dimensional level definition, 297
true the one of the two bool values denoting truth, 84
variable memory location with a name, 44
vector object denoting a point in two- or three-dimensional space, 59
virtual method method that can be redefined in a subclass, 158
void placeholder for the return type of a method that does not return a value, 99

Index

A. Egges et al., *Learning C# by Programming Games*,
DOI 10.1007/978-3-642-36580-5, © Springer-Verlag Berlin Heidelberg 2013