Solutions to the Exercises

Part I

1. *Programming paradigms*

   (a) Not true, Fortran is an imperative language but not object-oriented.

   (b) Not true, objects are modified by methods which are a kind of procedure.

   (c) Not true, PHP is a procedural language and it is interpreted, not compiled.

   (d) Not true, a programming language can hide certain aspects of the underlying processor.

2. *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 you have an executable.

3. *Names*
   Mathematical constants: e, π. Mathematical variables: x for something that varies, f for a function, ϕ for an angle. Physical constants: the speed of light c, gravity constant g. Physical variables: velocity v, acceleration a, mass m. If everyone uses the same symbols for certain constants and variables, then it is much easier to communicate.

4. *Classes and types*
   Standard classes: GraphicsDeviceManager, Game, SpriteBatch, GameTime, Console. Methods: GraphicsDeviceManager.Clear, SpriteBatch.Draw, Console.WriteLine. Three types that are not a class: **int**, **double**, **bool**.

5. *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 ∗/.

6. *Concepts*
   An instruction is a programming construct 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.

7. *Declaration, instruction, expression*
   A declaration determines the type of a variable. An instruction can be executed, which will change the memory. An expression can be calculated to determine its value.

8. *Statement versus instruction*
   A 'statement' means something like a 'remark' or a 'point of view', while we give the computer a *command* or an *instruction* to do something. A statement can 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;.

9. *Changing names*
   We have to change the name of the class, the name of the constructor method, and the call to the constructor method from Main. It is not necessary to change the name of the .cs file, but in general it is a good idea to assign the name of the class to the file in which it is defined.

10. *Syntactical categories*

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

11. *Relation between syntactical categories*
    The following combination of categories are possible: EC, IA, IM, EM. An assignment always goes together with an instruction. A constant is also an expression. An assignment, a declaration and an instruction always end with a semicolon. An expression and a constant never end with a semicolon. A method call sometimes ends with a semicolon, depending on whether it is called as a part of an expression or an instruction.

12. *Variable assignment*

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;	

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

13. *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.

14. *Hours, minutes, seconds*

```
int hours = 0;
int minutes = 0;
int seconds = 0;
while (time >= 3600)
{
      hours++;
      time -= 3600;
}
while (time >= 60)
{
      minutes++;
      time -= 60;
}
seconds = time;
```

Or simpler:

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

The reverse operation is much easier: `time = 3600*hours + 60*minutes + seconds;`.

15. *The game loop*
The game loop consists of LoadContent, Update, and Draw. LoadContent is executed once, the other two methods are executed in a loop. In the LoadContent method the game assets are loaded. In the Update method the game world is updated. In the Draw method the game world is drawn on the screen.

16. *Updating and drawing*
The first advantage is that it is easier to read for the programmer if the code for updating the game world is separated from the code for drawing the game world. Secondly, a game engine can make certain optimizations to the performance of the game by splitting the code. For example, if the game world didn't change, then it doesn't have to be redrawn.

Part II

1. *Keywords*
   The word 'void' means 'empty'. We use this word to indicate that a method doesn't have a return value. The word 'int' is an abbreviation of integer, which is used to define the integer type. The word 'return' is used in the body of a method to return a result, as well as return the control of the program to the caller of the method. The word 'this' is used to indicate the object that we're currently manipulating. The 'this' reference is needed so that we can access the variables stored in that object. We cannot use the 'this' word in a static method, since a static method doesn't manipulate an object.

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. *Methods with a result*

   (a)
```
int RemainderAfterDivision(int x, int y)
{
    return x − y∗(x/y);
}
```

   (b)
```
double Circumference(double height, double width)
{
    return 2∗height + 2∗width;
}
```

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

   (d)
```
string ThreeTimes(string s)
{
    return s + s + s;
}
```

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

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

4. *Cuneiform*

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

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

5. *Sequences*

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

It is also possible to do this without using a **while**- or a **for**-instruction:

```
int Total(int n)
{
    return n * (n+1) / 2;
}
```

(b)
```
int Factorial(int n)
{
    int res, t;
    res = 1;
    for (t=2; t<=n; t++)
```

```
        res *= t;
    return res;
}
```

(c)
```
double Power(double x, int n)
{
    double result = 1.0;
    for (int i=0; i<n; i++)
        result *= x;
    return result;
}
```

(d) By reusing the Power and Factorial methods, we can write down this method as follows:

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

This can be done slightly more efficiently by incorporating the factorial and power operations inside the method:

```
double Coshyp(double x)
{
     double res, a, b; int s, t;
     a=1; b=1; res=0;
     for (t=0; t<40; t+=2)
     {
          res += a/b;
          a *= x*x;
          b *= (t+1)*(t+2);
     }
     return res;
}
```

6. *Prime numbers*

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

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

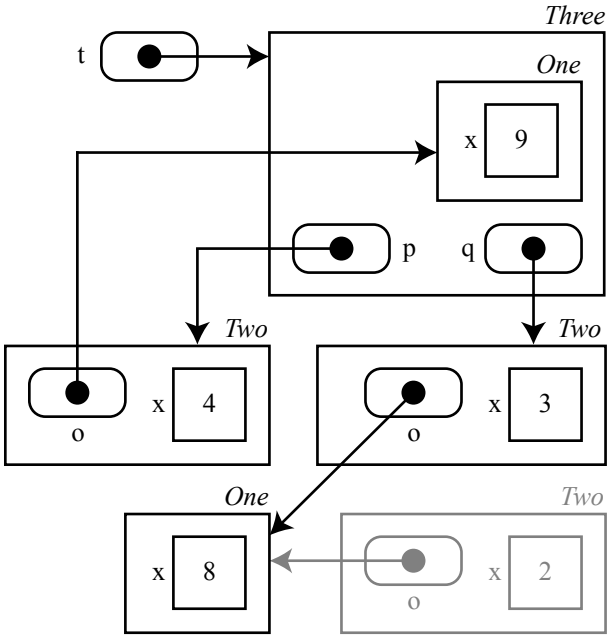
(c)
```
bool MultipleOf(int x, int y)
{
    return x%y == 0;
}
```

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

(e)
```
int SmallestDivider(int x)
{
    int divider = 2;
    while (!this.Divisible(x,divider))
        divider++;
    return divider;
}
```

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

7. *Drawing the memory*



The Two instance at the bottom right is greyed out. Because it is no longer reachable through an object reference, it will be disposed of automatically by C#'s garbage collection. You may therefore leave it out of the drawing.

8. *Classes and inheritance*

(a)

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

(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]

9. *Type checking*
Types are checked during the compilation phase. You can store an object of a subclass in a variable that has the type of a superclass:

```
class A {...}
class B : A {...}
class Test
{
    static 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 you indicate as a programmer that you know because of the previous instructions 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 the subclass type. This check cannot be done at compile time, since analysing where the object comes from is not always possible.

Part III

1. *Arrays*

   (a)
```
int CountZeros(int[] arr)
{
    int counter = 0;
    for (int i=0; i<arr.Length; i++)
        if (arr[i] == 0)
            counter++;
    return counter;
}
```

   (b)
```
int[] Add(int[] arr1, int[] arr2)
{
    int[] res = new int[arr1.Length];
    for (int i=0; i<arr1.Length; i++)
        res[i] = arr1[i] + arr2[i];
    return res;
}
```

   (c)
```
int FirstPosition(string s, char c)
{
    int pos = 0;
    while (pos < s.Length)
    {
        if (s[pos] == c)
            return pos;
        pos++;
    }
    return -1;
}
```

2. **string** *methods*

   (a)
```
public string ToUpper()
{
    string res = "";
    for (int n=0; n<this.Length; n++)
    { char c = this[n];
        if (c>='a' && c<='z')
            c = (char)(c-32);
        res += c;
    }
    return res;
}
```

   (b)
```
public string Replace(char x, char y)
{
    string res;
    int n;
    char c;
    res = "";
    for (n=0; n<this.Length; n++)
```

```
    {
        c = this[n];
        if (c==x)
            c = y;
        res += c;
    }
    return res;
}
```

(c)
```
public boolean EndsWith(string s)
{
    int n, sl, tl;
    tl = this.Length;
    sl = s.Length;
    if (sl>tl)
        return false;
    for (n=0; n<sl; n++)
    { if (s[n] != this[tl−sl+n])
            return false;
    }
    return true;
}
```

3. *Highway*

(a) One declaration is still missing. Write down this declaration and indicate where it should be placed in the program. In the class Highway, the following declaration should be added:

```
MotorizedVehicle[] road = new MotorizedVehicle[15];
```

(b)
```
for (int t = 0; t < road.Length; t++)
{ if (t % 3 != 0)
        road[t] = new Car();
    else if (t % 6 == 0)
        road[t] = new Truck();
    else road[t] = new Combination();
}
```

(c) The element type of the array is MotorizedVehicle, so in the Draw method of Highway the empty Draw method of the MotorizedVehicle class is called. This method should have been declared as **virtual** and in the subclass as **override**.

(d) We run into version management problems. If the shape of the truck changes later on, the code needs to be updated in two places.

(e) In the class Combination the trailer could have been drawn with the call **base**.Draw(g,x,y).

(f) Add a new class Wheel, independent of the other classes. Add a new class Vehicle as a superclass of the MotorizedVehicle class. Within that class define an array of Wheel objects. Add a new class Trailer as a subclass of Vehicle, and declare an instance of it in the Combination class.

4. *Searching and sorting*

(a)
```
double Largest(double [] a)
{
    double result = a[0];
    for (int t=1; t<a.Length; t++)
        if (a[t] > result)
            result = a[t];
    return result;
}
```

(b)
```
int IndexLargest(double [] a)
{
    int result = 0;
    for (int t=1; t<a.Length; t++)
        if (a[t] > a[result])
            result = t;
    return result;
}
```

(c)
```
int HowManySmallest(double [] a)
{
    int result = 1;
    double smallest = a[0];
    for (int t=1; t<a.Length; t++)
        if (a[t] < smallest)
        { result = 1;
            smallest = a[t];
        }
        else if (a[t]==smallest)
            result++;
    return result;
}
```

(d)
```
int IndexLargest(double [] a, int n)
{
    int result = 0;
    for (int t=1; t<n; t++)
        if (a[t] > a[result])
            result = t;
    return result;
}
```

(e)
```
void Sort(double [] a)
{
    for (int t=a.Length; t>0; t--)
    {
        int p = IndexLargest(a,t);
        double h = a[p];
        a[t-1] = a[p];
        a[p] = h;
    }
}
```

(f)
```
int First(int [] a, int x)
{
    for (int t=0; t<a.Length; t++)
        if (a[t]==x)
            return t;
    return -1;
}
```

(g)
```
int First(int [] a, int x)
{
    int low = 0;
    int high = a.Length;
    while (high>low)
    {
        int mid = (low+high)/2;
        if (a[mid]==x)
            return mid;
        else if (a[mid]<x)
            low = mid+1;
        else high = mid;
    }
    return -1;
}
```

Part IV

1. *Lists*

```
public void Reverse()
{
    for (int i=this.Count−2; i>=0; i−−)
    {
        this.Add(this[i]);
        this.RemoveAt(i);
    }
}
public int LastIndexOf(string item)
{
    int i=this.Count−1;
    while (i>=0)
    {
        if (this[i] == item)
            return i;
        i−−;
    }
    return i;
}
public bool Contains(string item)
{
    for (int i=0; i<this.Count; i++)
        if (this[i] == item)
            return true;
    return false;
}
```

2. *Collections*

```
void RemoveDuplicates(List<int> list)
{
    for (int i=list.Count−1; i>=0; i−−)
    {
        if (list.IndexOf(list[i], 0, i−1) != −1)
            list.RemoveAt(i);
    }
}
```

3. *Classes and interfaces*
   In version 1 and 3 an object of type IList is created. This isn't possible because IList is an interface and not a class. Version 2 is correct, because the List class implements the IList interface, so it may be assigned to a variable of the type IList.

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

4. *List and* **foreach**

(a)
```
void Increment(IList<Counter> list)
{
    for (int i=0; i<list.Count; i++)
        list[i].Increment();
}
void Increment(IList<Counter> list)
{
    foreach (Counter c in list)
        c.Increment();
}
```

(b) We don't have to change anything, because the class OwnList implements the IList interface. As a result, all the properties and methods needed, such as the Count property and the bracketed element access, are implemented.

5. *Strings*

```
public string Substring(int startIndex, int length)
{
    string s = "";
    for (int i=startIndex; i<startIndex + length && i<this.Length; i++)
        s += this[i];
    return s;
}
public string Substring(int startIndex)
{
    return this.Substring(startIndex, this.Length);
}
public int IndexOf(char c)
{
    for (int i=0; i<this.Length; i++)
        if (this[i] == c)
            return i;
    return -1;
}
```

6. *Classes and inheritance*

```
A::Method1
A::Method2
A::Method1
B::Method2
B::Method1
B::Method2
```

7. *Sidescrolling*

```
protected override void Update(GameTime gameTime)
{
    MouseState m = Mouse.GetState();
    if (m.X < 0)
        position.X+= 5;
    else if (m.X > GraphicsDevice.Viewport.Width)
        position.X-= 5;
```

```
        position.X = MathHelper.Clamp(position.X, GraphicsDevice.Viewport.Width − background.Width, 0);
    }
```

8. *Decorator streams*

```
class BufferedStream : Stream
{
    Stream subject;
    byte[] buffer = new byte[1000];
    int count, max;

    public BufferedStream(Stream s)
    { subject = s;
        count = 0; max=0;
    }

    public override int ReadByte()
    {
        if (count==max)
        { // no more bytes available, read a new block
            max = subject.Read(buffer, buffer.Length);
            if (max==0)
                return −1; // nothing left to read
            else count = 0;
        }
        byte res = buffer[count];
        count++;
        return res;
    }
}
```

The final three lines can also be combined to:

```
return buffer[count++];
```

You would think that the counter is incremented too soon, or not at all because we already return from the method. This is not the case, it works exactly as you would hope: the *old* value of count is used to index the array, and *then* the value of count is incremented. If we used the expression ++count instead, the counter would be incremented *before* indexing the array, but of course in this case it is not what we want.

Part V

1. *Text files and collections*

```
public class Word
{
    static void Main(string [] names)
    {
        if (names.length!=2)
            Console.WriteLine("Usage: Word input output");
        else
        { try
            {
                TextReader input = new StreamReader(names[0]);
                TextWriter output = new StreamWriter(names[1]);
                SortedSet<string> everything = new SortedSet<string>();
                string line;

                while ((line=input.readLine())!=null)
                    foreach (string word in line.Split(" "))
                        everything.Add(word);
                foreach (string t in everything)
                    output.writeLine(t);
            }
            catch (Exception e)
            { Console.WriteLine("An error occured.");
            }
        }
    }
}
```

2. *Abstract class and interfaces*
   An interface only contains method and property headers. This is also possible in an abstract class, if you write the **abstract** keyword in front of the header of the method. However, in an abstract class you can provide the body of some of the methods, which isn't possible with interfaces. Another difference is that you can declare member variables in an abstract class, and not in an interface.

   An abstract class is used as the basis of a hierarchy, where some member variables or methods are already defined. An example is the GameObject class. It already contains member variables to indicate the visibility and the position. Another example is the Stream class, where the Read method is defined using the yet to be implemented ReadByte method.

   An interface is used if you want to specify what the *possibilities* are, without already defining how these possibilities should be implemented. A good example of this is the ICollection interface. It defines what a collection should be able to do, but it doesn't define *how* the collection should do it.

3. *Tetris blocks*

   (a)
```
public void HorizontalMirror(bool[,] block)
{
    for (int x = 0; x < block.GetLength(0) / 2; x++)
    {
        for (int y = 0; y < block.GetLength(1); y++)
        {
```

```
                bool tmp = block[x, y];
                block[x, y] = block[block.GetLength(0) - 1 - x, y];
                block[block.GetLength(0) - 1 - x, y] = tmp;
            }
        }
    }
```

(b)
```
    public void Print(bool[,] array)
    {
        for (int y = 0; y < array.GetLength(1); y++)
        {
            for (int x = 0; x < array.GetLength(0); x++)
                if (array[x,y])
                    Console.Write("true ");
                else
                    Console.Write("false ");
            Console.WriteLine();
        }
    }
```

Instead of using the **if** instruction inside the **for** loop, you could have also used the ToString method of the boolean type. The method would then be given as follows:

```
public void Print(bool[,] array)
{
    for (int y = 0; y < array.GetLength(1); y++)
    {
        for (int x = 0; x < array.GetLength(0); x++)
            Console.Write(array[x,y].ToString() + " ");
        Console.WriteLine();
    }
}
```

4. *Abstract classes*

```
A obj; // yes, declaring a reference is always allowed
obj = new A(); // no, it's not allowed to make an instance of an abstract class type
obj = new B(); // yes, it's allowed to make an instance of a non-abstract subclass
obj.Method1(); // yes, the compiler knows that any subclass will have this method
obj.Method2(); // yes, the method is inherited
obj.Method3(obj); // no, the compiler can't verify that the object has this method
B otherObject = (B)(new A()); // no, it's not allowed to make an instance of an abstract class type
A yetAnotherObject = (A)obj; // yes, casting to a type higher in the hierarcy is allowed
obj.Method3(otherObject); // no, the compiler can't verify that the object has this method
A[] list; // yes
list = new A[10]; // yes, no instances of A are created here
list[0] = new A(); // no, it's not allowed to make an instance of an abstract class type
list[1] = new B(); // yes, it's allowed to make an instance of a non-abstract subclass
List<A> otherList = new List<A>(); // yes, again no instances of A are created here
```