Software Construction SWC 1.1

Exercise Set

Table of Content

Exercise 1 3
Exercise 2 4
Exercise 3 5
Exercise 4 6
Exercise 5 7
Exercise 6 8
Exercise 7 9
Exercise 8 10
Exercise 9 11
Exercise 9a 13
Exercise 9b 15
Exercise 10 17
Exercise 11 19
Exercise 12 21
Exercise 13 23
Exercise 14 24
Exercise 15 25
Exercise 16 26
Exercise 17 27
Exercise 18 28
Exercise 18a 29
Exercise 18b 30
Exercise 19 32
Exercise 19a 33
Exercise 20 35
Exercise 21 36
Exercise 22 37
Exercise 23 38
Exercise 24 39
Exercise 25 40
Exercise 26 42
Exercise 27 44
Exercise 28 47
Exercise 29 50
Exercise 30 52
Exercise 31 54

Exercise 1

Download the C# project **CarVersion1** from the class website.

Compared to the **Sandbox** project, you will find that the single line of code inside the *InsertCodeHere.cs* file has been replaced with several **declarations of variables**. All the variables are supposed to represent facts about a car.

1. For each variable, consider if the purpose of the variable is clearly understandable from its name
2. For each variable, consider what actual values each variable can have. Is this consistent with the purpose of the variable?
3. For each variable, make a assignment statement that assigns a proper value to the variable. Do this in the place indicated by the comments in the code
4. Make sure that your program can compile; press F6, and make sure you get 0 errors (NOTE: You will probably get some warnings about the variables not being used for anything – ignore those warnings for now ☺).

Exercise 2

Reopen the **Sandbox** project (or download it from the class website). Make sure to remove any existing code between the two comment lines:

```
// The FIRST line of code should be BELOW this line
```
and
```
// The LAST line of code should be ABOVE this line
```

Consider now the problem of representing the properties of a **bank account**.

1. Come up with as many relevant properties for a bank account that you can think of (to get you started: name of account holder, bank account number, …)
2. Declare variables in the code to represent these properties. Remember to
    a. Choose a good name for each variable
    b. Choose a suitable type for each variable
3. Following the variable declarations, assign some proper values to each variable, and make sure to check that your final code can actually compile (F6, no errors)

Exercise 3

Download the C# project **Arithmetic** from the class website.

Compared to the **Sandbox** project, you will find that the single line of code inside the *InsertCodeHere.cs* file has been replaced with several **arithmetic statements** (8 cases in total). Note that the program does not print anything if you run it.

For each case, you should try to figure out what the final value of the variable in that case is. When ready, see the result by uncommenting the statement that prints out the value of the variable. You uncomment a statement by removing the initial pair of backslashes (//):

Change

```
// Console.WriteLine("The value of a is : {0}", a);
```

to

```
Console.WriteLine("The value of a is : {0}", a);
```

and likewise for the subsequent cases.

Remember that you run the program by pressing F5 or clicking on the small green triangle.

Exercise 4

Reopen the **Sandbox** project (or download it from the class website). Make sure to remove any existing code between the two comment lines:

```
// The FIRST line of code should be BELOW this line
```
and
```
// The LAST line of code should be ABOVE this line
```

Your task is to write arithmetic statements corresponding to the below cases (variables are indicated in **bold**):

1. Set **a** equal to the product of 5 and 7
2. Set **b** equal to 25, then increase the value to three times as much
3. Set **c** equal to integer division of 16 and 5 (16/5)
4. Set **d** equal to 2.5, then divide it by 3.5
5. Set **e** equal to **b** minus **a**, then multipy that value with **c**
6. Set **f** equal to the remainder after dividing 24 with 6
7. Set **g** equal to 35, then divide it with **f**

If you have forgotten how to print out the value of a variable, then have a look at exercise 3 (or look in the **Arithmetic** project)

Exercise 5

Download the C# project **LogicExp** from the class website.

Compared to the **Sandbox** project, you will find that the single line of code inside the *InsertCodeHere.cs* file has been replaced with several **logic statements** (8 cases in total). Note that the program does not print anything if you run it.

For each case, you should try to figure out what the final value (true or false) of the variable in that case is. When ready, see the result by uncommenting the statement that prints out the value of the variable. You uncomment a statement by removing the initial pair of backslashes (//):

Change

```
// Console.WriteLine("The value of bool1 is : {0}", bool1);
```

to

```
Console.WriteLine("The value of bool1 is : {0}", bool1);
```

and likewise for the subsequent cases.

Remember that you run the program by pressing F5 or clicking on the small green triangle.

Exercise 6

Reopen the **Sandbox** project (or download it from the class website). Make sure to remove any existing code between the two comment lines:

```
// The FIRST line of code should be BELOW this line
```
and
```
// The LAST line of code should be ABOVE this line
```

Your task is to write a couple of logic statements. First, you must declare two integer variables **a** and **b**, set **a** equal to 7, and set **b** equal to 12.
Now write logical expressions for each of the cases below. That is, the expression should become true if the condition in the case is true. In each case, declare a boolean variable and set it equal to the value of the expression, like: `Boolean bool1 = …;`

1. **a** is smaller than 10
2. **a** is larger than or equal to 7
3. **a** is not equal to 12
4. **a** is smaller than 10 AND **a** is larger than 5
5. **a** is smaller than 10 AND **b** is not equal to 10
6. **a** is larger than 10 OR **b** is larger than 10
7. **b** is in the interval from 0 to 20 AND **a** is smaller than 8
8. **a** plus **b** is larger than **a** minus **b**

If you have forgotten how to print out the value of a variable, then have a look at exercise 5 (or look in the **LogicExp** project).

If you have time left, try to vary the initial values of **a** and **b**, and see how it effects the results

Exercise 7

Download the C# project **CarVersion2** from the class website.

This exercise is only about observing how an existing class is used. The solution contains a new class called **Car**, in the file Car.cs.

1. Have a look inside the Car.cs file, and see what the **Car** class definition contains (constructors, methods, etc). See if you can figure out what each method does – the comments for each method should be helpful

The **Car** class is put to use in the usual place (in the ***myCode*** method in the **InsertCodeHere** class).

2. Observe how the **Car** class is used. Make sure you understand how:
    a. Objects are created
    b. Parameters are specified in the constructors
    c. Methods are called
    d. Parameters are specified when calling a method
    e. Return values are used

3. If you have time left, you can try to create a third **Car** object yourself, and call some methods on the object.

Exercise 8

Download the C# project **StudentManager** from the class website.

This exercise is about using an existing class. The solution contains a new class called **Student**, in the file Student.cs.

1. Have a look inside the Student.cs file, and see what the **Student** class definition contains (constructors, methods, etc). See if you can figure out what each method does – the comments for each method should be helpful
2. Now try to use the **Student** class yourself! Insert some new code into the solution (in the usual place for inserting code), that does the following:
    a. Creates a new student object
    b. Adds a few test scores to the student
    c. Prints out the name and average test score for the student (use ***Console.WriteLine(…)***)
3. After this, you should extend the code in the **Student** class, such that it can also hold information about the country for the student. More specifically, you need to:
    a. Add a new instance field to hold the country value
    b. Change the constructor to include a country value in the parameter list (also remember to use the new parameter to initialise the country instance field)
    c. Add a method ***GetCountry*** to return the country value
4. Finally, test your new code by updating the code from part 2., such that country information is also used here

Exercise 9

Download the C# project **Library** from the class website.

This exercise is about using and modifying an existing class. The solution contains a new class called **Book**, in the file Book.cs.

1. Have a look inside the Book.cs file, and see what the **Book** class definition contains. It is a fairly simple class, with some instance fields, one constructor and some ***Get…*** methods. The methods ***BorrowFromLibrary*** and ***ReturnToLibrary*** are noteworthy; they are to be used when a user loans a book from the library, and returns it again. Also, the method ***PrintInformation*** prints out all information about a book.
2. Now try to use the **Book** class yourself. Insert some code into the solution (in the usual place for inserting code), that does the following:
   a. Creates a new **Book** object
   b. Print out some information about the book (there is a method you can call for that, remember?)
   c. Loan the book, and print the book information again
   d. Return the book, and print the book information again
3. After this, we decide that we want to keep track of the number of times a book has been borrowed. To do this, you must add code to the **Book** class. More specifically, you need to:

   a. Add a new instance field ***numberOf Loans*** to store the number of times a book has been borrowed (what would be a good type to use for this field?)
   b. Update the constructor to give this instance field an initial value (what seems like an obvious initial value?)
   c. Figure out when we should change the value of the new field (*Hint: What method will be activated when you borrow a book?*)
   d. Add a method ***GetNumberOfLoans***, that will return the value of the new field
   e. Update the code that prints book information, so that it also includes the number of times the book has been loaned
4. Finally, test your new code by updating the code from part 2. You should try to loan and return a book several times, to see if the value for number of book loans is updated correctly
5. [Extra – only if you have time] It seems strange that we can call ***BorrowFromLibrary*** many times, without calling ***ReturnToLibrary*** in between those calls. That does not make sense in the real world. Could we somehow change the code to handle this (*Hint: That would probably require use of a **conditional statement**, which we have not really talked about yet…*)?

Exercise 9a

Download the C# project **TinyFacebook** from the class website.

This exercise is about <u>using</u> and <u>modifying</u> an existing class. The solution contains a new class called **Status**, in the file Status.cs.

1. Have a look inside the file Status.cs, and see what the **Status** class definition contains. See if you can identify the instance fields, the constructor, the methods that return information (accessors), and the methods that change the value of an instance field (mutators)

2. Now try to <u>use</u> the **Status** class yourself. Insert some code into the solution (in InsertCodeHere.cs), that does the following:
    a. Creates a new **Status** object.
    b. Prints out some information about the **Status** object: The status text, the number of likes, and the number of dislikes. In order to do this, you need to call some method on the **Status** object.
    c. Makes some calls of the methods which change the number of likes and dislikes.
    d. Prints out the information again – now you should see that the information has changed

3. After this, we decide that we want to keep track of the number of times the status has been read. To do this, we need to <u>add code</u> to the **Status** class. More specifically, you need to:

a. Add a new instance field ***numberOf Reads*** to store the number of times a status has been read
b. Update the constructor to give this instance field an initial value (what seems like an obvious initial value?)
c. Add a new method ***ReadStatus***, that will increase the number of times the status has been read by one (Hint: The method will be very similar to the ***AddOneLike*** method)
d. Add a method ***GetNumberOfReads***, that will return the value of the new instance field
e. Update your test code, so the new methods are tested as well.

4. [Extra – only if you have time] Add a method ***ReadStatus-Multiple***, that will increase the number of times the status has been read with a number given by the user (Hint: The method will then need to take this number as a **parameter**).

Exercise 9b

Download the C# project **SuperMarket** from the class website.

This exercise is about using and modifying an existing class. The solution contains a new class **FruitBox**, in the file FruitBox.cs.

1. Have a look inside the file FruitBox.cs, and see what the **FruitBox** class definition contains. See if you can identify the instance fields, the constructor, the methods that return information (accessors), and the methods that change the value of an instance field (mutators)

2. Now try to use the **FruitBox** class yourself. Insert some code into the solution (in the usual place for inserting code), that does the following:
   a. Creates a new **FruitBox** object.
   b. Prints out information about the fruit box – use the method ***GetBoxContentDescription*** for this purpose.
   c. Adds some fruit to the box, by calling the appropriate methods
   d. Prints out the information again – now you should see that the information has changed

3. We now wish to add a new method that can calculate the total price of the fruit in the box, and return the total price to the caller. The method should be named ***GetTotalPrice***. Write this method, and add some code to test the new method as well.

4. Now add two new methods that can <u>change</u> the price of bananas and apples, respectively. The new price must then be a parameter to the method. Remember to add code in InsertCodeHere.cs to test the new methods.

5. We now have a situation where the price of a piece of fruit can be different for two different boxes of fruit. Is that a good model of how things really are in a supermarket?

Exercise 10

Download the C# project **DiceGame** from the class website.

This exercise is about <u>using</u> an existing class, and <u>completing</u> an unfinished class. The solution contains two new classes called **Die** and **DiceCup,** in the files Die.cs and DiceCup.cs, respectively.

1. Investigate the **Die** class. It is complete, and fairly simple. Note that we use another class in the **Die** class, called **Random**. This class is from the **.NET class library**, so it is not as such part of the project – it is “given”. Try out the class; create a **Die** object (in the usual place for inserting code), call the methods ***RollDie*** and ***GetValue***, and print the value you get back. Notice that when you run the program again, you may get different values, just like for a real die.
2. Investigate the **DiceCup** class. This class is supposed to represent a cup with two dice in it. Therefore, the class has two instance fields, both of the type **Die**. The constructor of the class has also been completed. However, we would like to have three public methods in the DiceCup class as well:
    a. **RollDice**: Should roll both dices in the cup. No value is returned.
    b. **GetTotalValue**: Should return the total value of the two dice in the cup.
    c. **IsTotalValueLargerThan**: This method should take one integer value as input, and return either true or false. The return value should be true if the total value of the two dice is larger than the input value; otherwise, it should return false.

3. Your job is now to add these three methods to the **DiceCup** class. As a help, the “headers” for each method is already included as a comment in the class. Once you have completed the methods, you should of course add some code over in **InsertCodeHere** to test that the completed class works as expected
4. [Extra – only if you have time] How can you change the game such that it uses 10-sided dice instead? Can you even make it so that the user decides how many sides the dice have?

Exercise 11

Download the C# project **BankVersion1** from the class website.

This exercise is about defining a new class yourself! In the project, there is a file called BankAccount.cs, containing a definition of a class **BankAccount**. However, the definition is empty…

We have to fill out the class definition. The requirements to the bank account class are the following:

- It must have a **name**, which is the name of the account holder
- It must have a **balance**, which is a decimal number.
- It must have a method ***Deposit***, which you can use for depositing an amount to the account (that is, the balance should increase)
- It must have a method ***Withdraw***, which you can use for withdrawing an amount from the account (that is, the balance should decrease)
- It must have methods for returning the value of the account holder name, and the value of the balance (one method for each value)

Given these requirement, your tasks are:

1. Fill out the **BankAccount** definition, such that all requirements are fulfilled. This includes definition of instance fields, constructors and methods
2. Make some test code that tests your **BankAccount** class. That is, you should create a **BankAccount** object, call the withdraw/deposit methods, and use the available methods to check that the object behaves as expected with regards to the value of the balance
3. If you have time left, try to extend the class definition with methods for adding interest to the account. It is up to you to define requirements, etc..

NOTE: This is a fairly large and somewhat difficult exercise. Try to solve it in small steps – get a little bit to work, test it, and then proceed to the next little bit. Manage the complexity!

Exercise 12

Download the C# project **Geometry** from the class website.

This exercise is about defining a new class yourself! In the project, there is a file called Geometry.cs, containing a definition of a class **Geometry**. However, the definition is empty…

We have to fill out the class definition. The requirements to the **Geometry** class are the following (use Google if you cannot remember basic Geometry ☺):

- Given the two side lengths of a rectangle, calculate the perimeter of the rectangle
- Given the two side lengths of a rectangle, calculate the area of the rectangle
- Given the radius of a circle, calculate the area of the circle
- Given the four angles of a polygon with four sides, determine if the polygon is a rectangle (what conditions must the angles fulfill?)

Given these requirement, your tasks are:

4. Fill out the **Geometry** definition, such that all requirements are fulfilled. This includes definition of instance fields (are any needed?), constructors and methods
5. Make some test code that tests your **Geometry** class. That is, you should create a **Geometry** object, call the various methods, and check the results against manually calculated results
6. If you have time left, try to extend the class definition with various interesting geometric methods. It is up to you to define requirements, etc..

NOTE: This is a fairly large and somewhat difficult exercise. Try to solve it in small steps – get a little bit to work, test it, and then proceed to the next little bit. Manage the complexity!

Exercise 13

Download the C# project **CarVersion3** from the class website.

This exercise is only about observing how we can use **if**-statements to implement some business logic. The solution contains a class called **Car**, in the file Car.cs.

There are three methods in the **Car** class that use **if**-statements

- ***IsEconomic***
- ***IsFamilyCar***
- ***RentalPricePerDay***

1. For each of these three methods, make sure you understand what the possible return values are, and under what circumstances each value can be returned.

2. Read the code in InsertCodeHere.cs – it tests the **Car** class. Before running the program, try to predict the outcome.

3. Run the program, and check that the outcome is as you predicted. If not, you may need to read the **Car** class code again

4. Try to change some of the values used when creating the three **Car** objects, and see if you can predict the outcome of running the program

Exercise 14

Download the C# project **BankVersion2** from the class website.

This project contains a working **BankAccount** class. However, it has some problems…

1. Try to test the **BankAccount** class, by adding code in the **InsertCodeHere** class. Specifically, make some tests that make the balance go negative

2. Now change the code in the ***withdraw*** method, such that a withdrawal is only done if the balance is larger than or equal to the given amount. Remember to test that the change works as expected

3. This makes the **BankAccount** class more realistic, but there are still problems – you can call both ***withdraw*** and ***deposit*** with negative amounts (try it), which does not make much sense. Make changes to both methods, such that they only perform a withdrawal/deposit if the given amount is positive. Remember that for the ***withdraw*** method, the change made in part 2 must still work!

4. Test that all your changes work as expected.

Exercise 15

Download the C# project **Numerology** from the class website.

This project contains a class called **MysticNumbers**, with a single method ***ThreeNumbers***. All that is known about ***ThreeNumbers*** is that it takes three integers as input, and returns one of them

1. By reading the code for ***ThreeNumbers***, try to figure out what it does. Write some test code to see if you are right.

2. Write and test a new method ***TwoNumbers***, that does the same thing as ***ThreeNumbers***, but now only for two numbers.

3. Write and test a new method ***FourNumbers***, that does the same thing as ***ThreeNumbers***, but now for four numbers (tip – you can probably use the method ***TwoNumbers*** to make the code fairly short and easy).

4. Rewrite ***ThreeNumbers*** to use the ***TwoNumbers*** method. What code do you like best – the original code or the new code?

Exercise 16

Download the C# project **MedicalAnalyser** from the class website.

This project contains a class called **BodyAnalyser**, with a completed method ***CalculateBMI***, plus an unfinished method ***AnalyseBMI***. BMI stands for “Body Mass Index” (Google it ☺). The unfinished method does compile, but it is not implemented correctly!

Your job is to correctly implement the method ***AnalyseBMI***. The specification is as follows:

- If the BMI is less than 15, it must return ***“You are way too skinny!”***
- If the BMI is between 15 and 22, it must return ***“You are a bit skinny”***
- If the BMI is between 22 and 28, it must return ***“You are just fine!”***
- If the BMI is between 28 and 35, it must return ***“You are a bit overweight!”***
- If the BMI is more than 35, it must return ***“You are way too fat!”***

As always, remember to write some code that tests your work.

Exercise 17

Download the C# project **WhileLoopsPart1** from the class website.

This exercise is about observing how a few counter-controlled **while**-loops work, and then try to code a few on your own.

1. In the file InsertCodeHere.cs, four cases of **while**-loops (Case 1-4) have been implemented. Try to figure out what the output from each loop will be. When ready, uncomment the line in each loop that prints the current value of the variable, and see if you were right.

2. Following the four given cases are four additional cases (Case 5-8). Here you must implement a **while**-loop yourself, to produce the output given in the comment for each case.

Exercise 18

Download the C# project **WhileLoopsPart2** from the class website.

This exercise is about implementing a sentinel-controlled **while**-statement yourself

1. In the file CubeCalculator.cs, a new class **CubeCalculator** has been defined. It contains methods useful for calculating the cube of a number given to the program by the user. Read the comments in the class definition, and make sure you understand what each method does (it is <u>not</u> needed to understand <u>how</u> the methods work…).
2. Run the program – you will see that you can input a number, and the cube of that number is then calculated. The program even handles the situation where the input is not a number. All this is done by the code inserted in InsertCodeHere.cs; read that code, and make sure you understand what it does.
3. A major drawback of the program is that you have to start it again each time you want to calculate a new cube value. Change the code in InsertCodeHere.cs such that it becomes a sentinel-controlled **while**-loop. That is, the user can keep entering numbers until some particular value is entered. The exact details are left for you to decide…

Exercise 18a

Download the C# project **CorrectChangePart1** from the class website.

This exercise is about calculating the correct change when a customer pays a due amount with too much cash. Example: A customer has to pay 266 kr., but pays 500 kr.. The customer must then receive 234 kr. in change. The tricky part is to figure out how to pay this amount using ordinary bills and coins, and paying back as few bills and coins as possible. In the above case, the correct way to pay back correct change would be:

- 1 200-kr bill
- 1 20-kr coin
- 1 10-kr coin
- 2 2-kr coins

In the project, there are no extra classes, so the code needed for the problem can be added in InsertCodeHere.cs.

1. In the indicated place in the code, add code to calculate the correct change. To keeps things simple, we assume that you only use 100-kr bills, 10-kr coins and 1-kr coins. Remember to test your code with some different values for change

2. If you have time, include some more bills and coins, like 50-kr bills, 5-kr coins, etc..

Exercise 18b

Download the C# project **BeastBattle** from the class website.

In this exercise, we will see how several classes can work together for a more complex task. The project is supposed to model a very simple game, where a “hero” can battle a “beast” until either beast or hero is dead!

The project contains four classes, which are described in general terms here – see the code for more details:

- The **NumberGenerator** class, with the method ***GetRandom-NumberInInterval***. This is a helper class for generating random numbers.
- The **BattleLog** class, where individual strings can be “saved”, and later on printed out on the screen.
- The **Hero** class, which models a game character. It is a very simple character, since it just has a number of hit points (a typical game element; your character has some hit points, and if something damages the character, it loses some hit points. When all hit points are lost, the character dies…)
- The **Beast** class, which also models a game character. Actually, the class is very similar to the **Hero** class…

So, even though this is obviously a very simple setup, it does actually resemble the game mechanics from many popular role-playing games (RPG).

1. Study the classes in details, so you are sure of what they can do and how they work. Note in particular how the Hero and Beast classes make use of the **NumberGenerator** and **BattleLog** classes.

2. See if you can figure out how to code a battle between a Hero and a Beast (until the death!). This is done in InsertCodeHere.cs. The premade code creates one object of each of the classes. YOUR JOB is to add code that makes the **Hero** object battle against the **Beast** object. This will require some thinking, so before starting to code, to try sketch out how a battle proceeds in detail. You will probably need a while-loop in your code ☺.

3. When you can make the two objects battle each other, there are a number of things to consider afterwards:
   a. It seems like the Hero wins most of the time (depending of course on how you coded the battle…). Why is that? How could we make the battle more fair?
   b. It looks like the damage dealt by the Hero is always within the interval 10 to 30 points. How could we change that? Could we even let the creator of the Hero object decide this interval? Could this also be done for the number of hit points?
   c. Do we really need two separate classes for **Hero** and **Beast**? What would we need to add to obtain a more fundamental class like e.g. **Character**?
   d. How could you implement something like weapons and armor, healing abilities, etc.?

Exercise 19

Download the C# project **ForLoopsPart1** from the class website.

This exercise is about predicting the outcome of a couple of **for**-loops, and to try to change some given **while**-loops into **for**-loops

1. In the file InsertCodeHere.cs, two cases of **for**-loops (Case 1-2) have been implemented. Try to figure out what the output from each loop will be. When ready, uncomment the line in each loop that prints the current value of the variable, and see if you were right.

2. The code also contains two **while**-loops (Case 3-4). Run the program to see what the output from each loop is. Now change each **while**-loop to an equivalent **for**-loop, i.e. a **for**-loop that produces the same output.

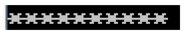
Exercise 19a

Download the C# project **DrawShapes** from the class website.

This exercise is about trying to draw some simple shapes on the screen, using for-loops to get the job done. A very simple class **DrawingTool** is provided to help with this.

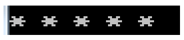
1. Study the class **DrawingTool**. As you can see, it is very simple. Are there any good reason to have such a class at all?
2. Using for-loops and the **DrawingTool** class, see if you can draw the following five simple shapes on the screen (they do become harder and harder, tough…). Just write the code in InsertCodeHere.cs as usual.

Shape A (10 stars in a row):



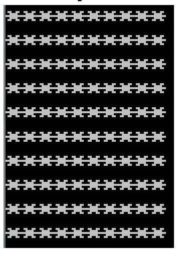
```
**********
```

Shape B (5 stars in a row, separated by spaces):



```
* * * * *
```

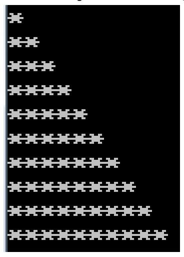
Shape C (10 rows of 10 stars each):



```
**********
**********
**********
**********
**********
**********
**********
**********
**********
**********
```

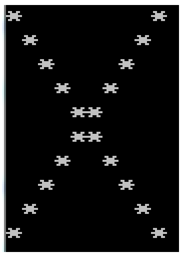
(Continues on next page…)

Shape D (So, what is this then…?):



```
*
**
***
****
*****
******
*******
********
*********
**********
```

Shape E (A big “X”, I guess…?):



```
*        *
 *      *
  *    *
   *  *
    **
    **
   *  *
  *    *
 *      *
*        *
```

Exercise 20

Download the C# project **SmallMathLibrary** from the class website.

This exercise is about implementing some loop-based methods in a class definition.

The project contains a new class **MathLib**, which contains four methods for various calculations:

- ***Faculty***
- ***IntervalSum***
- ***HighestSquareBelowLimit***
- ***IsPrime***

The method declarations are given, but the method bodies need to be completed. The purpose of each method is described in detail in the class definition itself (read it!). In InsertCodeHere.cs, some code has been added that tests the correctness of the methods in **MathLib**. Have a look at that code as well.

1. Implement the method ***Faculty***, and use the test to check that it works as expected
2. Implement the method ***IntervalSum***, and use the test to check that it works as expected
3. Implement the method ***HighestSquareBelowLimit***, and use the test to check that it works as expected
4. Implement the method ***IsPrime***, and use the test to check that it works as expected (NB: Difficult!)

Exercise 21

Download the C# project **ListExamples** from the class website.

This exercise is about predicting the result of applying some methods of the **List** class to a **List** object, and also about writing some code to use a **List** object

1. In the file InsertCodeHere.cs, a **List** object is created, and some elements are added and removed. At four points in the code (Case 1-4), you must predict the outcome of the ***WriteLine*** statement. When ready, you can uncomment the ***WriteLine*** statement, and see if your prediction was correct.
2. Following the cases above, four more cases are given (Case 5-8), where you must write pieces of code that use the **List** object, to retrieve various information about the elements in the list. Details for each case are given as comments in the code for each case.

Exercise 22

Download the C# project **BookCatalog_v1** from the class website.

This exercise illustrates the concept of a catalog. A catalog is a class that can store and use data of a certain type, without revealing the specific representation of data to the user of the catalog.

The project contains the simple data class **Book** (note however that we consider the instance field ***isbn*** to be a "key" for **Book**, i.e. no two **Book** objects can have the same ***isbn*** value).
Also, it contains the (incomplete) catalog class **BookCatalog**. The three public methods in **BookCatalog** allow the user to store and use **Book** objects in a simple way (see the comments in the code for more details about each method).

Given that we have chosen to use a **List<Book>** to store **Book** objects internally in the **BookCatalog** class, your job is now:

1. Complete the three methods in the **BookCatalog** class.
2. Study the test written in InsertCodeHere.cs, and figure out what you expect the test to output.
3. Run the program, and see if the output of the test matches your expectations (if not, you will have to examine the test and your code once again…)
4. Is there anything in the program that prevents a user from entering two **Book** objects with the same ***isbn*** value?

Exercise 23

Download the C# project **CarPoolManager** from the class website.

This exercise is about completing the definition of a class that uses a **Dictionary** object

The project contains the class **Car**. This is a very simple representation of a car, with only three instance fields; license plate, brand and model. This class does not need to be changed!

The project also contains the class **CarPool**. This class is supposed to represent a pool of cars; for this purpose, an instance field ***carList*** of type **Dictionary** is used to hold Key-Value pairs consisting of license plates and **Car** objects (since a car is uniquely identified by its license plate)

1. Look in the class definition of **CarPool**. You will see that three methods (***AddCarToPool***, ***RemoveCarFromPool***, ***LookupCar***) are not completed. Your job is to complete these methods, according to the specification given in the comments in the code.

2. In the file InsertCodeHere.cs, some code that tests the **CarPool** class has been inserted. Run the program, and check that your **CarPool** class behaves as expected.

Exercise 24

(Note: This exercise is intentionally similar to 20a…)
Download the C# project **BookCatalog_v2** from the class website.

This exercise illustrates the concept of a catalog. A catalog is a class that can store and use data of a certain type, without revealing the specific representation of data to the user of the catalog.

The project contains the simple data class **Book** (note however that we consider the instance field ***isbn*** to be a "key" for **Book**, i.e. no two **Book** objects can have the same ***isbn*** value).
Also, it contains the (incomplete) catalog class **BookCatalog**. The three public methods in **BookCatalog** allow the user to store and use **Book** objects in a simple way (see the comments in the code for more details about each method).

Given that we have chosen to use a **Dictionary<string,Book>** to store **Book** objects internally in the **BookCatalog** class, your job is now:

1. Complete the three methods in the **BookCatalog** class.
2. Study the test written in InsertCodeHere.cs, and figure out what you expect the test to output.
3. Run the program, and see if the output of the test matches your expectations (if not, you will have to examine the test and your code once again…)
4. Is there anything in the program that prevents a user from entering two **Book** objects with the same ***isbn*** value?

Exercise 25

Download the C# project **StudentInfoManager** from the class website.

This exercise is about completing the definition of a class that uses a **Dictionary** object

The project contains the class **Student**. This is a simple representation of a student, with three instance fields; student id, name and test scores. The first two are simple, but the "test scores" field is a **Dictionary** field, holding Key-Value pairs of course names (**String**) and scores (**int**).

The project also contains the class **StudentInfo**. This class is supposed to be able to retrieve various information about a group of students; for this purpose, an instance field ***students*** of type **Dictionary** is used to hold Key-Value pairs consisting of student ids and **Student** objects (since a student is uniquely identified by a student id)

1. The class **Student** is complete, and you need not change anything in it. However, take a good look at the **Student** class anyway, and make sure you understand how the methods work. Pay particular attention to the method ***GetScoreAverage***.

2. Look in the class definition of **StudentInfo**. You will see that five methods (***GetStudentCount***, ***AddStudent, GetStudent***, ***GetAverageForStudent, GetTotalAverage***) are not comple-ted. Your job is to complete these methods, according to the specification given in the comments in the code.

3. In the file InsertCodeHere.cs, some code that tests the **StudentInfo** class has been inserted. Run the program, and check that your **StudentInfo** class behaves as expected.

Exercise 26

Download the C# project **Company_v1** from the class website.

Initially, the project is rather simple – it just contains the class **Employee**. This class is supposed to serve as a base class for a number of derived classes.

1. Examine the **Employee** class. You can see that an employee just has a name and a monthly salary, both of which are set through the constructor. After construction, the values cannot be changed. Note that the method ***getSalaryPerMonth()*** is declared to be virtual.

2. Define a class **Worker**. The **Worker** class is supposed to be derived from the **Employee** class. A worker has a skill, which can be represented by a simple string, like “truck driver” or “maintenance”. The **Worker** class should thus have an instance field to represent this. The skill should be set when a **Worker** object is constructed, but it should also be possible to change the skill later. When implementing the **Worker** class, pay special attention to implementing the constructor correctly (it must call the base class constructor).

3. Define a class **Manager**. This class should also be derived from the **Employee** class. In addition to having a monthly salary, a manager also has a monthly bonus, which should be specified at construction. The bonus is paid out if the manager has worked more than 180 hours in a month. It should be possible to specify the number of worked hours

after the object has been created. When implementing the **Manager** class, pay special attention to implementing the method ***getSalaryPerMonth()*** correctly, since this should also include the bonus, if the bonus condition is fulfilled. Remember that the method is supposed to override the method in the base class.

4. Define a class **Director**. The **Director** class is supposed to be derived from the **Manager** class, i.e. not the **Employee** class. A director is just a manager who has a fixed monthly bonus of 20000. The condition for triggering the bonus is the same as for a manager. When implementing the **Director** class, pay special attention to implementing the constructor correctly. Does the class need anything else than a constructor?

5. Create a test of the classes, as usual in the ***myCode*** method in the **InsertCodeHere** class. More specifically, you should
   - Create a list which can hold **Employee** objects.
   - Create some **Worker** objects, and add them to the list
   - Create some **Manager** objects, set the hours worked, and add them to the list
   - Create some **Director** objects, set the hours worked, and add them to the list
   - Using a loop statement, print out the content of the arraylist, like “(name) has a salary of (salary)”

Exercise 27

Download the C# project **Company_v2** from the class website.

Initially, the project is rather simple – it just contains the class **Employee**. This class is supposed to serve as a base class for a number of derived classes, just in the above exercise

1. Examine the **Employee** class. You can see that an employee just has a name, nothing more. The rules for how a salary is calculated are now completely "free", and should be made concrete in the derived classes. However, there are some very general rules for salary calculation:
   - Part of the salary is considered to be a bonus
   - The bonus is paid if a certain condition is met.

2. The above rules for calculating the salary are quite vague, and the **Employee** class can therefore not contain any specific implementation of salary calculation. However, a number of abstract methods relating to salary calculation are included in the **Employee** class. Since the methods are abstract, the **Employee** class itself becomes abstract. Can you then create an **Employee** object?

3. We now wish to define a class **Worker**, to represent a worker-type employee. A worker has the following characteristics with regards to salary:
   - A worker is paid a fixed amount per hour
   - A worker works a fixed number of hours per month
   - A Worker does not receive a bonus

4. Given the above definitions, implement the **Worker** class – it will be derived from the **Employee** class. You will need to include some instance fields for hourly pay and hours worked per month, and provide concrete implementations for the abstract methods. Should you be able to create a **Worker** object?

5. We now wish to define a **Manager** class. A manager is somewhat more weakly defined than a worker with regards to salary:
   - A manager has a fixed monthly base salary
   - A manager has a fixed monthly bonus
   - The condition for when the bonus is paid out may vary, depending on the specific type of manager (so **Manager** might become a base class…?)

6. Given the above definitions, implement the **Manager** class – it will be derived from the **Employee** class. You will need to include some instance fields for monthly base salary and monthly bonus, and provide concrete implementations for <u>some</u> of the abstract methods. Should you be able to create a **Manager** object (probably not, and why not…?)? Should the **Manager** class then become an abstract class?

7. We now wish to define a **JuniorManager** class. This class will be derived from the **Manager** class. A junior manager will have the bonus paid out if (s)he has worked more than 180 hours during the month. So, we will need to add an instance field to hold the number of hours worked, a way of setting this value, and – very importantly – to provide a concrete implementation of the ***IsBonusPaidOut()*** method. Should you

be able to create a **JuniorManager** object (you probably should…)?

8. We now wish to define a **SeniorManager** class. This class will be derived from the **Manager** class. A senior manager will have the bonus paid out if (s)he has a “performance level” of at least 6 during the month. So, we will need to add an instance field to hold the performance level, a way of setting this value, and – very importantly – to provide a concrete implementation of the ***IsBonusPaidOut()*** method. Should you be able to create a **SeniorManager** object (you probably should…)?

9. Create a test of the classes, as usual in the ***myCode*** method in the **InsertCodeHere** class. More specifically, you should
    - Create a list which can hold **Employee** objects.
    - Create some **Worker** objects, and add them to the list
    - Create some **JuniorManager** objects, set the hours worked, and add them to the list
    - Create some **SeniorManager** objects, set the performance level, and add them to the list
    - Using a loop statement, print out the content of the list, like “(name) has a salary of (salary)”

Exercise 28

Download the C# project **HumansAndPirates** from the class website.

This exercise is about seeing further examples of inheritance and polymorphic behavior. Also, we will investigate how we can reduce the dependency between a class hierarchy and the code that uses the class hierarchy (the so-called “client code”).

The project contains no less than five new classes. In the first part, we concentrate on the following three classes:

IHuman: This is actually an interface, defining a very simple interface for humans (just one method ***SayHello***).

StandardHuman: Contains an implementation of the ***IHuman*** interface. The ***SayHello*** method returns “Hello”.

Pirate: A class that is derived from **StandardHuman**. It overrides the ***SayHello*** method to return the string “Ahoy, matey!” instead.

1. Look in the code in InsertCodeHere.cs, specifically the part called PART 1 (from the comment “PART 1 – Start” to the comment “PART 1 – End”). Observe how we can add both **StandardHuman** and **Pirate** objects to the list ***everybody***, even though it has the type **List<IHuman>**. Make sure that you understand why this is possible. Also note that this operation requires the client code (we call the code in InsertCodeHere.cs the “client code”) to have explicit knowledge about the classes **StandardHuman** and **Pirate**. Consider why this is a potential problem.

2. Run the program. You will see that it prints a mixed list of “Hello” and “Ahoy, matey!” strings. Make sure you understand why this happens. When you are done, remove the line of code indicated in the program.

3. In the next part of the program (PART 2), we make use of a class **HumanFactory**. This class (with a slightly spooky name) can create new objects of either type **StandardHuman** or **Pirate**, and return the newly created object to the caller. The creation is done when calling the method ***CreateHuman***. Read the code in the method, and see if you can figure out how it works. Why would we want such a class at all…?

4. Run the program. You will see that PART 2 also prints a mixed list of “Hello” and “Ahoy, matey!” strings. Make sure you understand why this happens. Also consider why this approach has reduced the dependency between the class hierarchy and the client code. When you are done, remove the line of code indicated in the program.

5. In the third part of the code (PART 3), we have made it possible for the user to decide which objects to put into the list. This is done be using a **while**-loop, which reads input from the user. See if you can work out how the loop works. Also consider if the dependency between the class hierarchy and the client code has changed as compared to PART 2.

6. Run the program, and try to input some different objects. Check that the printed strings match your input.

7. The program is now to be extended a bit, by introduction of a **Captain** class which is also derived from the **StandardHuman** class (see the code which is already included in the project). All three parts of the code in InsertCodeHere.cs must now be changed to include some **Captain** objects in the list. Make the needed changes (this includes changes to the **HumanFactory** class), and pay specific attention to how much you need to change the client code in each case.

8. [HARD] You probably discovered that in PART 3, we did not have to change anything in the client code…almost! The "prompt" to the user should actually be changed, so the user knows what options she has with regards to input. Unfortunately, this means that the changes were not quite isolated to the factory class (**HumanFactory**). However, if we are a bit creative, we can even isolate that change as well. See if you can figure out how to do that! (Tip: Maybe the factory class can help us produce the prompt string…)

Exercise 29

Download the C# project **BankWithExceptions** from the class website.

This exercise is about throwing and catching exceptions

The project contains a class **BankAccount**. It defines a fairly straightforward bank account, but there are a few restrictions in it (see the code). One restriction is that the balance must not become negative.

The project also contains three additional classes:

- **IllegalInterestRateException**
- **NegativeAmountException**
- **WithdrawAmountTooLargeException**

They are all exception classes, i.e. they inherit from **Exception**. The specific purpose of each exception class is described in the code.

The **BankAccount** class already uses the **WithdrawAmountTooLargeException** class, to prevent that the balance becomes negative (see the ***Withdraw*** method)

1. Modify the code in the **BankAccount** class, such that the additional exception classes are used properly

2. Update the test code found in InsertCodeHere.cs, such that method calls that could cause an exception to be thrown are enclosed in try-catch blocks

3. Test that the exceptions are thrown and handled properly, by adding some test code that will provoke the exceptions to happen

Exercise 30

Download the C# project **StaticExamples** from the class website.

This exercise is about using and defining static classes, methods and instance variables.

The project contains the class **ListMethods**, which defines two methods ***FindSmallestNumber*** and ***FindAverage***. The names should hopefully indicate what they do…

Code that tests the class is included in InsertCodeHere.cs. The class is tested in the traditional way; create an object, and call methods on the object.

1. Modify the **ListMethods** class such that it becomes a static class. Remember that a static class can only contain static methods.

2. Modify the code in InsertCodeHere.cs, such that it uses the **ListMethods** class as a static class. The output of running the program should of course be as before.

The project also contains a simple class **Car** (see the code). We would now like to track how the class is used. More specifically, we wish to track:

- How many objects of type **Car** have been created
- How many times has the method ***GetLicensePlate*** been called?
- How many times has the method ***GetPrice*** been called?

Note that it is the "grand totals" we wish to track, <u>not</u> the number of method calls on each object!

3. Add static instance fields to the **Car** class, to enable the tracking described above. Increment the value of each variable at the appropriate place in the class.

4. Add a static method that can print out the values of the static instance fields. It could be called ***PrintUsageStatistics***.

5. Test that your additions work, by including some test code in InsertCodeHere.cs. Create some **Car** objects, call methods on them, and finally call the static method to observe the usage statistics.

Exercise 31

Download the C# project **PartnerDesigner** from the class website.

This exercise is about discovering some problems in using the simple, built-in types like **String** and **int**, and trying to improve the code by using enumerated types instead.

In the project, the class **Human** has been included. The class contains four instance fields, one for:

- Gender
- Eye color
- Hair color
- Height category

You can thus create a **Human** object (your future partner…?) by specifying values for each of these four properties in the class constructor. Furthermore, you can get a textual description of a **Human** object by calling the method **GetDescription()**.

Some code that tests the **Human** class is as always included in InsertCodeHere.cs.

1. Examine the code in the **Human** class definition and in InsertCodeHere.cs, and see if you can predict the outcome of running the program

Running the program reveals some problems. In two cases, we have specified hair color where we should have specified eye color, and vice versa (unless you really want a partner with white eyes and blue hair…), and in one case, we have specified a non-existing height category

2. Change the **Human** class definition by adding enumerated types for gender, eye color, hair color and height category. Use these new types for the four instance variables. The constructor needs some changes as well. Also consider if you still need the methods **GetGenderDescription** and **GetHeightDescription**

3. Change the code in InsertCodeHere.cs, so it is compatible with the redesigned **Human** class. Observe how it is now only possible to specify legal values for each type.

4. Reflect a bit on the changes. Is there anything in the new code that is more complicated than it was in the original code? Was it always relevant to use an enumerated type?