

Development of distributed services - Project II

Jan Magne Tjensvold

October 29, 2007

Chapter 1

Project II

1.1 Introduction

I choose to do all three assignments instead of just two because, quite frankly, they were a bit easy. In each of the assignments all the business logic was implemented as class libraries (compiles to .dll files) while the user interfaces was implemented as console applications. The program code for each of the assignments can be viewed in Appendix A at the end of the report.

I have learned a few new lessons while programming with C#. First of all the documentation is lacking compared to Java. It is not as well organized and at times it can be hard to find the information you are looking for. One example is when I was trying to find out how to implement the IComparer interface correctly. I wanted to find out which values I should return from the Compare() method. Looking at the definition of the IComparer interface in Visual Studio 2005 I got the following cryptic information.

```
// Returns:  
//     Value Condition Less than zerox is less than y.Zerox  
//     equals y.Greater than zerox is greater than y.  
int Compare(T x, T y);
```

After some inspection I noticed that some spaces are missing, amongst other things. It looks a bit amateurish, but I guess it will improve with newer versions just like Java has done.

1.2 Assignment 1

Assignment 1 was about modeling a bottle and methods for filling, emptying and pouring its contents to other bottles.

1.2.1 1b - Simple bottle client

The following is the console output for the bottle client from assignment 1b:

```
Bottle Client 1b
=====

Initial state
  bottle1: 0/2
  bottle2: 0/7
Step 1: Fill up bottle2
  bottle1: 0/2
  bottle2: 7/7
Step 2: Pour from bottle2 to bottle1
  bottle1: 2/2
  bottle2: 5/7
Step 3: Empty bottle1
  bottle1: 0/2
  bottle2: 5/7
```

At the end the 7 liter bottle contains 5 liter.

1.2.2 1c - Advanced bottle client

In assignment 1c I implemented three different methods for solving the problem. The last one is the most interesting one. It picks one of the methods at random at each step and keeps on doing so until the 5 liter bottle contains 4 liter. The following is the console output for the program on a lucky day:

```
Bottle Client 1c
=====

Solve by Precognition
-----

Initial state
  bottle1: 0/3
  bottle2: 0/5
Step 1: Fill bottle2 with 4 liters
  bottle1: 0/3
  bottle2: 4/5
Problem solved in 1 step

Solve by Predefined Steps
```

Initial state
 bottle1: 0/3
 bottle2: 0/5
Step 1: Fill up bottle2
 bottle1: 0/3
 bottle2: 5/5
Step 2: Pour from bottle2 to bottle1
 bottle1: 3/3
 bottle2: 2/5
Step 3: Empty bottle 1
 bottle1: 0/3
 bottle2: 2/5
Step 4: Pour from bottle2 to bottle1
 bottle1: 2/3
 bottle2: 0/5
Step 5: Fill up bottle2
 bottle1: 2/3
 bottle2: 5/5
Step 6: Pour from bottle2 to bottle1
 bottle1: 3/3
 bottle2: 4/5
Problem solved in 6 step

Solve by Randomization

Initial state
 bottle1: 0/3
 bottle2: 0/5
Step 1: Fill up bottle2
 bottle1: 0/3
 bottle2: 5/5
Step 2: Empty bottle1
 bottle1: 0/3
 bottle2: 5/5
Step 3: Fill up bottle2
 bottle1: 0/3
 bottle2: 5/5

[...]

Step 151: Empty bottle2

```
    bottle1: 1/3
    bottle2: 0/5
Step 152: Pour from bottle1 to bottle2
    bottle1: 0/3
    bottle2: 1/5
Step 153: Fill up bottle1
    bottle1: 3/3
    bottle2: 1/5
Step 154: Pour from bottle1 to bottle2
    bottle1: 0/3
    bottle2: 4/5
Problem solved in 154 steps
```

On a bad day the output from the randomization solver will look more like this:

Solve by Randomization

```
Initial state
    bottle1: 0/3
    bottle2: 0/5
Step 1: Fill up bottle1
    bottle1: 3/3
    bottle2: 0/5
Step 2: Empty bottle1
    bottle1: 0/3
    bottle2: 0/5
Step 3: Empty bottle2
    bottle1: 0/3
    bottle2: 0/5

[...]

Step 119036: Pour from bottle1 to bottle2
    bottle1: 0/3
    bottle2: 1/5
Step 119037: Empty bottle1
    bottle1: 0/3
    bottle2: 1/5
Step 119038: Fill up bottle1
    bottle1: 3/3
    bottle2: 1/5
Step 119039: Pour from bottle1 to bottle2
```

```
bottle1: 0/3
bottle2: 4/5
Problem solved in 119039 steps
```

1.3 Assignment 2

All the algorithms were implemented as a single class library. For assignment 2c I choose to implement the binary search algorithm. For the initial bubble sort run the array specified in the assignment was used:

```
Int[] tall = { 8, 4, 2, 6, 1 }
```

Later an array of 10 elements with random values is used. This means that each time the program is run the results will be different. Below you can see the output during one run of the console program:

```
BubbleSort
=====
```

```
Sorting an array of integers:
```

```
Before: {8, 4, 2, 6, 1}
```

```
After:  {1, 2, 4, 6, 8}
```

```
Doing a linear search with an array of integers:
```

```
Array: {1, 2, 3, 3, 2, 8, 1, 8, 4, 1}
```

```
The first occurrence of 2 was found at index 1 in the search array
```

```
Sorting another array of integers:
```

```
Before: {1, 2, 3, 3, 2, 8, 1, 8, 4, 1}
```

```
After:  {1, 1, 1, 2, 2, 3, 3, 4, 8, 8}
```

```
Doing a binary search with a sorted array of integers:
```

```
Array: {1, 1, 1, 2, 2, 3, 3, 4, 8, 8}
```

```
2 was found at index 4 in the search array
```

As you might notice the binary search algorithm finds the value 2 at index 4 instead of index 3, which is where it first occurs. This comes from the way the binary search algorithm works with sorted arrays containing duplicate values. See the program code for more detail.

1.4 Assignment 3

Assignment 3 revolved around the game of dice and the game of poker.

1.4.1 3a - Game of dice

Console output for a game of dice is shown below:

```
Dice Game
=====
```

```
Enter starting cash: 100
```

```
Enter bet between 1 - 100 (0 to exit): 100
You won this round
Dice: 4 3 3 4 Cash: 200
```

```
Enter bet between 1 - 200 (0 to exit): 150
You won this round
Dice: 5 4 4 2 Cash: 350
```

```
Enter bet between 1 - 350 (0 to exit): 250
You won this round
Dice: 6 2 2 5 Cash: 600
```

```
Enter bet between 1 - 600 (0 to exit): 400
You won this round
Dice: 6 5 6 5 Cash: 1000
```

```
Enter bet between 1 - 1000 (0 to exit): 200
You lost this round
Dice: 6 4 3 1 Cash: 800
```

```
Enter bet between 1 - 800 (0 to exit): 250
You won this round
Dice: 3 4 3 5 Cash: 1050
```

```
Enter bet between 1 -1050 (0 to exit): 400
You won this round
Dice: 2 6 3 4 Cash: 1450
```

```
Enter bet between 1 - 1450 (0 to exit): 800
You won this round
Dice: 4 4 5 3 Cash: 2250
```

```
Enter bet between 1 - 2250 (0 to exit): 2250
You lost this round
Dice: 1 5 5 1 Cash: 0
```

1.4.2 3b - Poker

For the poker game I used standard one letter codes for ranks and suits. Ranks go from A-1-2-...-9-T-J-Q-K and for the suits I used the first letter in their name. For example 'Jd' represents a jack of diamonds and '6s' is a six of spades. I also decided to implement the full poker hand scoring system as shown in Figure 1.1 on page 9 instead of the subset suggested in the assignment.

In the program code the ranks, suits and scores are defined as enums. Each rank and suit is assigned a value on the form 2^n to facilitate a quick hand scoring algorithm. It uses the value of the ranks and suits of each card and logically ORs them together. This is an idea Øystein Johansen presented to me and it appears to work very well for most scores. It does include some additional complexities when you want to distinguish between two pairs and three of a kind and between a full house and four of a kind. Distinguishing a royal flush from a straight flush is also a bit tricky, as well as finding ace-high straights in general. See the program code to find out how these problems were solved.

The output of the basic poker program in action is shown below:

```
Poker Game
=====

Hand: Ad 5h Td Js Qh
You have: High Card

Do you want to try again? (Y/N) y

Hand: Ad 6d 7s 7h 7c
You have: Three of a Kind

Do you want to try again? (Y/N) y

Hand: Ah 2d 3h 4h Ks
You have: High Card

Do you want to try again? (Y/N) y

Hand: As 4s 6h 6c 9s
You have: One Pair

Do you want to try again? (Y/N) y

Hand: 2d 3s 8s Ts Th
```


You have: One Pair

Do you want to try again? (Y/N) n

1.4.3 3c - Poker with card swapping

For assignment 3c swapping of cards during the game was added. See the below console output of one such game session:

```
Poker Game With Card Swapping
=====
```

```
Hand: 6d 9d Ks Kh Kd
You have: Three of a Kind
```

```
Type card numbers from 1 - 5 separated by space to swap (enter to skip)
1 2
```

```
New cards: 3s 3h
```

```
New hand: 3s 3h Ks Kh Kd
You now have: Full House
```

```
Nice improvement from three of a kind
```

```
Do you want to try again? (Y/N) y
```

```
Hand: 3d 7d Js Qd Ks
You have: High Card
```

```
Type card numbers from 1 - 5 separated by space to swap (enter to skip)
3 5
```






```
New cards: 3s Qs
```

```
New hand: 3s 3d 7d Qs Qd
You now have: Two Pair
```

```
Nice improvement from high card
```

```
Do you want to try again? (Y/N) n
```

♠ ♥ **POKER** ♦ ♣
HAND RANKINGS

 Royal Flush	10♥ J♥ Q♥ K♥ A♥
 Straight Flush	4♣ 5♣ 6♣ 7♣ 8♣
 Four of a Kind	K♠ K♥ K♣ K♦ 3♠
 Full House	10♥ 10♠ 10♦ A♠ A♣
 Flush	10♠ K♠ 2♠ 6♠ 7♠
 Straight	7♣ 8♠ 9♦ 10♠ J♥
 Three of a Kind	5♠ 5♥ 5♣ J♦ A♦
 Two Pair	A♠ A♥ 3♣ 3♠ J♣
 One Pair	Q♦ Q♥ 2♥ 8♠ 9♣

© Jeremy Voros (jeremyvoros@gmail.com) (some rights reserved) Creative Commons Attribution-Share Alike 3.0 License

Figure 1.1: Complete poker hand ranking/scoring

Appendix A

Program code

```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace Bottle
6 {
7     /**
8      * A class representing a bottle with certain contents and capacity.
9      */
10    public class Bottle
11    {
12        private double capacity; // Maximum capacity of the bottle.
13        private double contents = 0; // Current contents in the bottle.
14
15        /**
16         * Creates a standard bottle with capacity of 1.
17         */
18        public Bottle(): this(1)
19        {
20        }
21
22        /**
23         * Creates a bottle with the specified capacity.
24         */
25        public Bottle(double capacity)
26        {
27            if (capacity <= 0) throw new ArgumentException(
28                "negative or zero capacity specified", "capacity");
29            this.capacity = capacity;
30        }
31
32        /**
33         * Returns the total capacity of this bottle.
34         */
35        public double TotalCapacity()
36        {
37            return capacity;
38        }
39
40        /**
41         * Returns the remaining capacity of this bottle.
42         */
43        public double RemainingCapacity()
44        {
45            return capacity - contents;
46        }
47
48        /**
49         * Returns the current contents of the bottle.
50         */
51        public double Content()
52        {
53            return contents;
54        }
55
56        /**
57         * Fill the bottle to its maximum capacity. Returns the amount that was
58         * added to the original amount in order to fill it up.
59         */
60        public double FillUp()
61        {
62            return capacity - Fill(capacity);
63        }
64
65        /**
66         * Fill the specified amount into the bottle. If the filling causes the
67         * bottle to exceed its capacity it will return the exceeding amount.
68         */
69        public double Fill(double amount)
70        {
71            if (amount < 0) throw new ArgumentException(
72                "negative fill amount specified", "amount");
73
74            amount = contents + amount;
75

```

```

76         if (amount > capacity)
77         {
78             contents = capacity;
79             return amount - capacity;
80         }
81         else
82         {
83             contents = amount;
84             return 0;
85         }
86     }
87
88     /**
89     * Empties the bottle and returns the amount the bottle contained
90     * before it was emptied.
91     */
92     public double Empty()
93     {
94         double amount = contents;
95         contents = 0;
96         return amount;
97     }
98
99     /**
100    * Pours the entire contents of this bottle into another bottle. If the
101    * destination bottle has enough capacity left to take the entire
102    * contents of this bottle then the resulting contents of this bottle
103    * will be zero. If the destination bottle only has capacity enough to
104    * take some of the contents from this bottle, then the destination
105    * bottle will be filled to its maximum capacity and the remaining
106    * contents will be kept in this bottle so that nothing is spilt.
107    */
108    public void PourInto(Bottle destination)
109    {
110        PourInto(destination, contents);
111    }
112
113    /**
114    * Pours the specified amount of this bottle into another bottle. If
115    * the amount specified is more than the available contents in this
116    * bottle the entire contents of this bottle is poured into the other
117    * bottle if possible. If the destination bottle does not have enough
118    * capacity left to take the specified amount from this bottle then the
119    * destination bottle will be filled to its maximum capacity and the
120    * remaining contents willW be kept in this bottle so that nothing is
121    * spilt.
122    */
123    public void PourInto(Bottle destination, double amount)
124    {
125        if (contents < amount) amount = contents;
126        contents = contents - amount + destination.Fill(amount);
127    }
128
129    /**
130    * Pours from the source bottle to this bottle.
131    * See pourInto(Bottle) for more formation.
132    */
133    public void PourFrom(Bottle source)
134    {
135        // I would have preferred to only define PourInto(Bottle), but
136        // assignment 1a defines PourFrom(Bottle) which is counterintuitive
137        // in my opinion. When you hold a real bottle it's not physically
138        // possible to make a bottle you're not holding pour its content
139        // into your own bottle. At least not without external help. In my
140        // opinion this method defies logic, but luckily it doesn't need any
141        // real logic since it's easily mapped to PourInto(Bottle).
142        source.PourInto(this);
143    }
144
145    /**
146    * Pours a certain amount from the source bottle to this bottle.
147    * See pourInto(Bottle, double) for more formation.
148    */
149    public void PourFrom(Bottle source, double amount)
150    {

```

```
151         source.PourInto(this, amount);
152     }
153
154     public override string ToString()
155     {
156         return contents + "/" + capacity;
157     }
158 }
159
160 }
161
```

```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace BottleClient1b
6 {
7     using Bottle;
8
9     /**
10     * A program using the bottle class as described in assignment 1b.
11     */
12     class Program
13     {
14         // Initialize the two bottles.
15         static Bottle bottle1 = new Bottle(2.0);
16         static Bottle bottle2 = new Bottle(7.0);
17
18         static void Main(string[] args)
19         {
20             Console.WriteLine();
21             Console.WriteLine("Bottle Client 1b");
22             Console.WriteLine("=====");
23             Console.WriteLine();
24
25             Console.WriteLine("Initial state");
26             WriteBottleInfo();
27
28             Console.WriteLine("Step 1: Fill up bottle2");
29             bottle2.FillUp();
30             WriteBottleInfo();
31
32             Console.WriteLine("Step 2: Pour from bottle2 to bottle1");
33             bottle2.PourInto(bottle1);
34             WriteBottleInfo();
35
36             Console.WriteLine("Step 3: Empty bottle1");
37             bottle1.Empty();
38             WriteBottleInfo();
39         }
40
41         /**
42         * Prints out information about the two bottles to the console.
43         */
44         static void WriteBottleInfo()
45         {
46             Console.WriteLine("  bottle1: " + bottle1);
47             Console.WriteLine("  bottle2: " + bottle2);
48         }
49     }
50 }
51

```

```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace BottleClient1c
6 {
7     using Bottle;
8
9     /**
10    * A program using the bottle class to solve the problem in assignment 1c.
11    */
12    class Program
13    {
14        static Bottle bottle1;
15        static Bottle bottle2;
16
17        static void Main(string[] args)
18        {
19            Console.WriteLine();
20            Console.WriteLine("Bottle Client 1c");
21            Console.WriteLine("=====");
22            Console.WriteLine();
23
24            SolveByPrecognition();
25            SolveByPredefinedSteps();
26            SolveByRandomization();
27        }
28
29        /**
30        * Initializes bottle1 with a capacity of 3 liters and bottle2 with a
31        * capacity of 5 liters.
32        */
33        static void Initialize()
34        {
35            bottle1 = new Bottle(3.0);
36            bottle2 = new Bottle(5.0);
37            Console.WriteLine("Initial state");
38            WriteBottleInfo();
39        }
40
41        /**
42        * Prints out information about the two bottles to the console.
43        */
44        static void WriteBottleInfo()
45        {
46            Console.WriteLine("  bottle1: " + bottle1);
47            Console.WriteLine("  bottle2: " + bottle2);
48        }
49
50        /**
51        * Given that assignment 1a gives room for additional helper methods
52        * Bottle.Fill(double) was implemented to simplify the code. This was
53        * done before it was made apparent that its functionality would fit
54        * perfectly for 1c. However 1c does not specify the Fill(double)
55        * method as one of the legal methods which can be used. Still, it's a
56        * perfect way to solve the problem. Short, concise and to the point.
57        */
58        static void SolveByPrecognition()
59        {
60            Console.WriteLine();
61            Console.WriteLine("Solve by Precognition");
62            Console.WriteLine("-----");
63            Console.WriteLine();
64            Initialize();
65
66            Console.WriteLine("Step 1: Fill bottle2 with 4 liters");
67            bottle2.Fill(4);
68            WriteBottleInfo();
69
70            Console.WriteLine("Problem solved in 1 step");
71        }
72
73        /**
74        * To stay within the bounds of the assignment a set of predefined
75        * steps using the specified methods was discovered reasonably easy.

```



```
151         bottle2.Empty();
152         break;
153     case 4:
154         Console.WriteLine("Pour from bottle1 to bottle2");
155         bottle2.PourFrom(bottle1);
156         break;
157     case 5:
158         Console.WriteLine("Pour from bottle2 to bottle1");
159         bottle1.PourFrom(bottle2);
160         break;
161     }
162     WriteBottleInfo();
163 }
164 while (bottle2.Content() != 4);
165
166 Console.WriteLine("Problem solved in " + step + " steps");
167 }
168 }
169 }
170 }
```

```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace Algorithms
6 {
7     /**
8      * A small collection of sorting and searching algorithms.
9      */
10    public class Algorithms
11    {
12        /**
13         * Use the Bubble sort algorithm to sort a list of integers in
14         * ascending order.
15         */
16        public static void BubbleSort(int[] list)
17        {
18            for (int i = list.Length - 1; i > 0; i--)
19                for (int j = 0; j < i; j++)
20                    if (list[j] > list[j + 1]) Swap(list, j, j + 1);
21        }
22
23        /**
24         * Swap the elements at index a and b in the list.
25         */
26        public static void Swap(int[] list, int a, int b)
27        {
28            int temp = list[a];
29            list[a] = list[b];
30            list[b] = temp;
31        }
32
33        /**
34         * Does a linear search through an array of integers and returns the
35         * index of the first occurrence of the specified value. If the value
36         * cannot be found it returns -1.
37         */
38        public static int LinearSearch(int[] list, int value)
39        {
40            for (int i = 0; i < list.Length; i++)
41                if (list[i] == value) return i;
42            return -1;
43        }
44
45        /**
46         * Does a binary search through a sorted array of integers and returns
47         * the index of the specified value. If the value being searched for
48         * occurs multiple times (in sequence) this algorithm will may not
49         * return the index of the first occurrence. If the value cannot be
50         * found it returns -1.
51         */
52        public static int BinarySearch(int[] list, int value)
53        {
54            int low = 0;
55            int high = list.Length - 1;
56            int mid;
57
58            while (low <= high)
59            {
60                mid = (low + high) / 2;
61                if (list[mid] > value)
62                    high = mid - 1;
63                else if (list[mid] < value)
64                    low = mid + 1;
65                else
66                    return mid;
67            }
68
69            return -1;
70        }
71    }
72 }
73

```

```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace BubbleSort
6 {
7     using Algorithms;
8
9     /**
10    * A program to show how the sorting and searching algorithms are used.
11    */
12    class Program
13    {
14        static void Main(string[] args)
15        {
16            Console.WriteLine();
17            Console.WriteLine("BubbleSort");
18            Console.WriteLine("=====");
19
20            Console.WriteLine();
21            Console.WriteLine("Sorting an array of integers:");
22            int[] sort = {8, 4, 2, 6, 1};
23            Console.WriteLine("Before: " + ArrayToString(sort));
24            Algorithms.BubbleSort(sort);
25            Console.WriteLine("After:  " + ArrayToString(sort));
26
27            Console.WriteLine();
28            Console.WriteLine(
29                "Doing a linear search with an array of integers:");
30            Random ran = new Random();
31            int[] search = RandomArray(10, 0, 9, ran);
32            Console.WriteLine("Array: " + ArrayToString(search));
33            int value = ran.Next(0, 10);
34            int result = Algorithms.LinearSearch(search, value);
35            if (result < 0)
36                Console.WriteLine("No occurrence of " + value
37                    + " was not found in the search array");
38            else
39                Console.WriteLine("The first occurrence of " + value
40                    + " was found at index " + result + " in the search array");
41
42            Console.WriteLine();
43            Console.WriteLine("Sorting another array of integers:");
44            Console.WriteLine("Before: " + ArrayToString(search));
45            Algorithms.BubbleSort(search);
46            Console.WriteLine("After:  " + ArrayToString(search));
47
48            Console.WriteLine();
49            Console.WriteLine(
50                "Doing a binary search with a sorted array of integers:");
51            Console.WriteLine("Array: " + ArrayToString(search));
52            result = Algorithms.BinarySearch(search, value);
53            if (result < 0)
54                Console.WriteLine("No occurrence of " + value
55                    + " was not found in the search array");
56            else
57                Console.WriteLine(value + " was found at index " + result
58                    + " in the search array");
59        }
60
61        /**
62        * Create an array of the given size with random values.
63        */
64        static int[] RandomArray(uint size, int min, int max, Random random)
65        {
66            int[] array = new int[size];
67            for (uint i = 0; i < size; i++) array[i] = random.Next(min, max + 1);
68            return array;
69        }
70
71        /**
72        * Convert an array of integers to a string.
73        */
74        static String ArrayToString(int[] list)
75        {

```

```
76     StringBuilder sb = new StringBuilder("{ " + list[0]);
77     for (uint i = 1; i < list.Length; i++) sb.Append(", " + list[i]);
78     sb.Append("}");
79     return sb.ToString();
80 }
81 }
82 }
83 }
```

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace DiceGame
6 {
7     /**
8      * A six sided die.
9      */
10    public class Die
11    {
12        private static Random random = new Random();
13        private int lastValue = -1;
14
15        /**
16         * Roll the die. It returns the resulting value.
17         */
18        public int Roll()
19        {
20            lastValue = random.Next(1, 7);
21            return lastValue;
22        }
23
24        /**
25         * Read the last value of the die. If it has never been rolled the last
26         * value will be -1.
27         */
28        public int LastValue()
29        {
30            return lastValue;
31        }
32
33        /**
34         * Returns a string representation of the current state of the die.
35         */
36        public override string ToString()
37        {
38            return lastValue.ToString();
39        }
40    }
41 }
42
```

```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace DiceGame
6 {
7     /**
8      * A game of dice.
9      */
10    public class DiceGame
11    {
12        private const uint DICE_COUNT = 4;
13        private const int DIE_LOSE = 1;
14        private Die[] dice = new Die[DICE_COUNT];
15        private uint cash;
16
17        /**
18         * Creates a game with 100 cash.
19         */
20        public DiceGame(): this(100)
21        {
22        }
23
24        /**
25         * Creates a game with the specified amount of starting cash.
26         */
27        public DiceGame(uint startingCash)
28        {
29            if (startingCash == 0) throw new ArgumentException(
30                "starting cash cannot be zero", "startingCash");
31            cash = startingCash;
32
33            for (int i = 0; i < dice.Length; i++)
34                dice[i] = new Die();
35        }
36
37        /**
38         * Places a bet and rolls the dice. The bet must be a non-zero amount,
39         * but not larger than the current cash balance. The player lose if at
40         * least one die is 1, otherwise the player wins. Returns true if the
41         * player won and false if the player lost.
42         */
43        public bool BetAndRoll(uint bet)
44        {
45            if (bet > cash) throw new ArgumentException(
46                "cannot bet more than current cash balance", "bet");
47            if (bet == 0) throw new ArgumentException(
48                "cannot bet zero cash", "bet");
49
50            bool win = true;
51            foreach (Die die in dice)
52            {
53                int value = die.Roll();
54                if (value == DIE_LOSE) win = false;
55            }
56
57            if (win)
58                cash += bet;
59            else
60                cash -= bet;
61
62            return win;
63        }
64
65        /**
66         * Returns the current cash balance.
67         */
68        public uint Cash()
69        {
70            return cash;
71        }
72
73        /**
74         * Returns the last value of the specified die. The game has DICE_COUNT
75         * dice so only an index of 0 to DICE_COUNT - 1 is valid.

```

```
76     */
77     public int DieValue(int index)
78     {
79         return dice[index].LastValue();
80     }
81
82     /**
83     * Returns a string representation of the current state of the game.
84     */
85     public override string ToString()
86     {
87         StringBuilder sb = new StringBuilder("Dice: ");
88         foreach (Die die in dice) sb.Append(die + " ");
89         sb.Append("Cash: ");
90         sb.Append(cash);
91         return sb.ToString();
92     }
93 }
94 }
95
```



```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace ConsoleDiceGame
6 {
7     using DiceGame;
8
9     /**
10     * A console based game of dice.
11     */
12     class Program
13     {
14         static void Main(string[] args)
15         {
16             Console.WriteLine();
17             Console.WriteLine("Dice Game");
18             Console.WriteLine("=====");
19             Console.WriteLine();
20
21             uint cash;
22             do
23                 Console.Write("Enter starting cash: ");
24             while (!uint.TryParse(Console.ReadLine(), out cash) || cash == 0);
25
26             DiceGame game = new DiceGame(cash);
27             Console.WriteLine();
28
29             // Main game loop.
30             do
31             {
32                 uint bet;
33                 do
34                     Console.Write("Enter bet between 1 - " + game.Cash()
35                     + " (0 to exit): ");
36                 while (!uint.TryParse(Console.ReadLine(), out bet)
37                 || bet > game.Cash());
38
39                 if (bet == 0) break;
40                 if (game.BetAndRoll(bet))
41                     Console.WriteLine("You won this round");
42                 else
43                     Console.WriteLine("You lost this round");
44
45                 Console.WriteLine(game);
46                 Console.WriteLine();
47             }
48             while (game.Cash() > 0);
49         }
50     }
51 }
52 }
53

```

```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace Cards
6 {
7     /**
8      * A playing card.
9      */
10    public class Card : IComparable<Card>
11    {
12        private Deck.Suit suit;
13        private Deck.Rank rank;
14
15        /**
16         * Create a card with the given suit and rank.
17         */
18        public Card(Deck.Suit suit, Deck.Rank rank)
19        {
20            this.suit = suit;
21            this.rank = rank;
22        }
23
24        /**
25         * Return the suit of this card.
26         */
27        public Deck.Suit Suit()
28        {
29            return suit;
30        }
31
32        /**
33         * Return the rank of this card.
34         */
35        public Deck.Rank Rank()
36        {
37            return rank;
38        }
39
40        public int CompareTo(Card other)
41        {
42            int diff = rank - other.rank;
43            if (diff == 0) diff = suit - other.suit;
44            return diff;
45        }
46
47        public override bool Equals(Object obj)
48        {
49            return (suit.Equals(obj) && rank.Equals(obj));
50        }
51
52        public override int GetHashCode()
53        {
54            return suit.GetHashCode() * rank.GetHashCode();
55        }
56
57        public override string ToString()
58        {
59            return RankToString(rank) + SuitToString(suit);
60        }
61
62        public static String RankToString(Deck.Rank rank)
63        {
64            switch (rank)
65            {
66                case Deck.Rank.Ace: return "A";
67                case Deck.Rank.Deuce: return "2";
68                case Deck.Rank.Three: return "3";
69                case Deck.Rank.Four: return "4";
70                case Deck.Rank.Five: return "5";
71                case Deck.Rank.Six: return "6";
72                case Deck.Rank.Seven: return "7";
73                case Deck.Rank.Eight: return "8";
74                case Deck.Rank.Nine: return "9";
75                case Deck.Rank.Ten: return "T";

```

```
76         case Deck.Rank.Jack:    return "J";
77         case Deck.Rank.Queen:   return "Q";
78         case Deck.Rank.King:    return "K";
79     }
80     throw new ArgumentException("invalid rank specified", "rank");
81 }
82
83 public static String SuitToString(Deck.Suit suit)
84 {
85     switch (suit)
86     {
87         case Deck.Suit.Spades:    return "s";
88         case Deck.Suit.Hearts:    return "h";
89         case Deck.Suit.Diamonds:  return "d";
90         case Deck.Suit.Clubs:     return "c";
91     }
92     throw new ArgumentException("invalid suit specified", "suit");
93 }
94 }
95 }
96 }
```

```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace Cards
6 {
7     /**
8     * A hand of playing cards drawn from a deck.
9     */
10    public class Hand
11    {
12        public const int HAND_SIZE = 5;
13
14        private Card[] cards = new Card[HAND_SIZE];
15        private Deck deck;
16
17        /**
18        * Creates a hand by drawing cards from the specified deck.
19        */
20        public Hand(Deck deck)
21        {
22            this.deck = deck;
23            for (int i = 0; i < HAND_SIZE; i++)
24                cards[i] = deck.Draw();
25            Array.Sort(cards);
26        }
27
28        public Hand(Card[] cards)
29        {
30            this.cards = cards;
31        }
32
33        /**
34        * Swap the cards at the specified indices in the card list. The new
35        * cards are drawn before the old ones are placed back in the deck.
36        * This prevents the player from accidentally drawing one of the cards
37        * he is trying to sway. The list of the new cards drawn from the deck
38        * is returned.
39        */
40        public Card[] SwapCards(uint[] cardIndex)
41        {
42            Card[] newCards = new Card[cardIndex.Length];
43            for (int i = 0; i < cardIndex.Length; i++)
44                newCards[i] = deck.Draw();
45
46            // Put back old cards and replace them with the new ones.
47            for (int i = 0; i < cardIndex.Length; i++)
48            {
49                uint index = cardIndex[i];
50                deck.PutBack(cards[index]);
51                cards[index] = newCards[i];
52            }
53            Array.Sort(cards);
54            Array.Sort(newCards);
55
56            return newCards;
57        }
58
59        /**
60        * Returns an array of all the cards in the hand.
61        */
62        public Card[] Cards()
63        {
64            return cards;
65        }
66
67        /**
68        * Puts all the cards in this hand back into the deck. After calling
69        * this method you will be unable to use the hand because all the cards
70        * will be removed.
71        */
72        public void PutBack()
73        {
74            foreach (Card card in cards)
75                deck.PutBack(card);

```

```

76
77     cards = null;
78     deck = null;
79 }
80
81 public Deck.Score ScoreHand()
82 {
83     return ScoreHand(this);
84 }
85
86 public static Deck.Score ScoreHand(Hand hand)
87 {
88     Deck.Suit suitBits = 0;
89     Deck.Rank rankBits = 0;
90     for (int i = 0; i < HAND_SIZE; i++)
91     {
92         rankBits |= hand.cards[i].Rank();
93         suitBits |= hand.cards[i].Suit();
94     }
95
96     uint rankSeqSize, rankSeqEnd;
97     uint numRanks = BitCount((uint)rankBits,
98         (uint)Enum.GetValues(typeof(Deck.Rank)).Length,
99         out rankSeqSize, out rankSeqEnd);
100     uint numSuits = BitCount((uint)suitBits,
101         (uint)Enum.GetValues(typeof(Deck.Suit)).Length);
102
103     switch (numRanks)
104     {
105         // If the number of different ranks is less than 5 then we
106         // have at least one pair and the possible scores are:
107         // OnePair, TwoPair, ThreeOfAKind, FullHouse and FourOfAKind.
108         case 4:
109             return Deck.Score.OnePair;
110         case 3: // TwoPair or ThreeOfAKind
111             if (RankCount(hand) == 2)
112                 return Deck.Score.TwoPair;
113             else
114                 return Deck.Score.ThreeOfAKind;
115         case 2: // FullHouse or FourOfAKind
116             if (numSuits == 3 || RankCount(hand) == 3)
117                 return Deck.Score.FullHouse;
118             else
119                 return Deck.Score.FourOfAKind;
120         case 5:
121             // Possible scores:
122             // HighCard, Straight, Flush, StraightFlush and RoyalFlush.
123
124             bool flush = (numSuits == 1);
125             bool straight = (rankSeqSize == HAND_SIZE);
126
127             // Special check for ace-high straight.
128             if (rankSeqSize == HAND_SIZE - 1
129                 && rankSeqEnd == Enum.GetValues(typeof(Deck.Rank)).Length
130                 && (rankBits & Deck.Rank.Ace) == Deck.Rank.Ace)
131             {
132                 if (flush)
133                     return Deck.Score.RoyalFlush;
134                 else
135                     return Deck.Score.Straight;
136             }
137
138             if (straight && flush)
139                 return Deck.Score.StraightFlush;
140             else if (straight)
141                 return Deck.Score.Straight;
142             else if (flush)
143                 return Deck.Score.Flush;
144             else
145                 return Deck.Score.HighCard;
146         default:
147             throw new ApplicationException(
148                 "unable to determine hand score");
149     }
150 }

```

```

151
152 /**
153  * Returns the number of 1 bits in an unsigned integer along with the
154  * size of the largest sequence of consecutive 1 bits and on which bit
155  * the sequence ends. The bit index is counted from the least
156  * significant bit, which is 1, to the most significant bit, which is
157  * 32.
158  */
159 private static uint BitCount(uint val, uint maxBits,
160     out uint maxSequence, out uint endSequence)
161 {
162     if (maxBits > 32) throw new ArgumentException(
163         "maximum number of bits cannot exceed 32", "maxBits");
164
165     uint bits = 0;
166     uint curSequence = 0;
167     maxSequence = 0;
168     endSequence = 0;
169
170     for (uint i = 0; i < maxBits; i++)
171     {
172         if ((val & 1) == 1)
173         {
174             bits++;
175             curSequence++;
176             if (curSequence > maxSequence)
177             {
178                 maxSequence = curSequence;
179                 endSequence = (i + 1);
180             }
181         }
182         else
183             curSequence = 0;
184
185         val >>= 1;
186     }
187     return bits;
188 }
189
190 /**
191  * Returns the number of 1 bits in an unsigned integer.
192  */
193 private static uint BitCount(uint val, uint maxBits)
194 {
195     uint x, y;
196     return BitCount(val, maxBits, out x, out y);
197 }
198
199 /**
200  * Counts the number of ranks which occurs most often in this hand.
201  */
202 private static uint RankCount(Hand hand)
203 {
204     uint[] bucket = new uint[(uint)Enum
205         .GetValues(typeof(Deck.Rank)).Length];
206     uint max = 0;
207
208     foreach (Card card in hand.cards)
209     {
210         uint index = (uint)Math.Log((double)card.Rank(), 2);
211         uint test = ++bucket[index];
212         if (test > max)
213             max = test;
214     }
215     return max;
216 }
217
218 /**
219  * Returns a string representation of the hand.
220  */
221 public override string ToString()
222 {
223     StringBuilder sb = new StringBuilder(cards[0].ToString());
224     for (int i = 1; i < cards.Length; i++)
225         sb.Append(" " + cards[i]);

```

```
226         return sb.ToString();
227     }
228
229     public static String ScoreToString(Deck.Score score)
230     {
231         switch (score)
232         {
233             case Deck.Score.HighCard: return "High Card";
234             case Deck.Score.OnePair: return "One Pair";
235             case Deck.Score.TwoPair: return "Two Pair";
236             case Deck.Score.ThreeOfAKind: return "Three of a Kind";
237             case Deck.Score.Straight: return "Straight";
238             case Deck.Score.Flush: return "Flush";
239             case Deck.Score.FullHouse: return "Full House";
240             case Deck.Score.FourOfAKind: return "Four of a Kind";
241             case Deck.Score.StraightFlush: return "Straight Flush";
242             case Deck.Score.RoyalFlush: return "Royal Flush";
243         }
244         throw new ArgumentException("invalid score specified", "score");
245     }
246 }
247 }
248
```

```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace Cards
6 {
7     /**
8      * A deck of playing cards.
9      */
10    public class Deck
11    {
12        /** Card suits. */
13        public enum Suit
14        {
15            Spades = 0x1,
16            Hearts = 0x2,
17            Diamonds = 0x4,
18            Clubs = 0x8,
19        }
20
21        /** Card ranks. */
22        public enum Rank
23        {
24            Ace = 0x1,
25            Deuce = 0x2,
26            Three = 0x4,
27            Four = 0x8,
28            Five = 0x10,
29            Six = 0x20,
30            Seven = 0x40,
31            Eight = 0x80,
32            Nine = 0x100,
33            Ten = 0x200,
34            Jack = 0x400,
35            Queen = 0x800,
36            King = 0x1000,
37        }
38
39        /** Hand score. */
40        public enum Score
41        {
42            HighCard = 1,
43            OnePair = 2,
44            TwoPair = 3,
45            ThreeOfAKind = 4,
46            Straight = 5,
47            Flush = 6,
48            FullHouse = 7,
49            FourOfAKind = 8,
50            StraightFlush = 9,
51            RoyalFlush = 10,
52        }
53
54        private List<Card> cards = new List<Card>(
55            Enum.GetValues(typeof(Suit)).Length
56            * Enum.GetValues(typeof(Rank)).Length);
57
58        private Random random = new Random();
59
60        /**
61         * Create a deck with the standard set of cards.
62         */
63        public Deck()
64        {
65            foreach (Suit suit in Enum.GetValues(typeof(Suit)))
66                foreach (Rank rank in Enum.GetValues(typeof(Rank)))
67                    cards.Add(new Card(suit, rank));
68        }
69
70        /**
71         * Draw a card from the deck. This returns the card by first removing
72         * it from the deck.
73         */
74        public Card Draw()
75        {

```



```

76         int index = random.Next(0, cards.Count);
77         Card card = cards[index];
78         cards.RemoveAt(index);
79         return card;
80     }
81
82     /**
83     * Puts a card back into the deck.
84     */
85     public void PutBack(Card card)
86     {
87         if (cards.Contains(card))
88             throw new ArgumentException("card is already in deck", "card");
89
90         cards.Add(card);
91     }
92
93     /**
94     * Deal the specified number of hands from this deck.
95     */
96     public Hand[] Deal(uint numHands)
97     {
98         if (numHands == 0)
99             throw new ArgumentException("cannot draw zero hands", "numHands");
100
101         Hand[] hands = new Hand[numHands];
102         for (int i = 0; i < hands.Length; i++)
103             hands[i] = new Hand(this);
104
105         return hands;
106     }
107
108 }
109 }
110 }
111

```

```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace ConsolePokerGame
6 {
7     using Cards;
8
9     /**
10     * A console based poker game.
11     */
12     class Program
13     {
14         static void Main(string[] args)
15         {
16             Console.WriteLine();
17             Console.WriteLine("Poker Game");
18             Console.WriteLine("=====");
19
20             Deck deck = new Deck();
21
22             bool retry = true;
23             // Main game loop.
24             while (retry)
25             {
26                 Hand hand = deck.Deal(1)[0];
27                 Console.WriteLine();
28                 Console.WriteLine("Hand: " + hand);
29                 Deck.Score score = hand.ScoreHand();
30                 Console.WriteLine("You have: " + Hand.ScoreToString(score));
31                 Console.WriteLine();
32
33                 Console.Write("Do you want to try again? (Y/N) ");
34                 retry = Console.ReadLine().ToUpperInvariant().Equals("Y");
35                 hand.PutBack(); // Be nice and put the hand back into the deck.
36             }
37         }
38     }
39 }
40

```

```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace ConsolePokerGameWithSwap
6 {
7     using Cards;
8
9     /**
10     * A console based poker game where you can swap cards.
11     */
12     class Program
13     {
14         static void Main(string[] args)
15         {
16             Console.WriteLine();
17             Console.WriteLine("Poker Game With Card Swapping");
18             Console.WriteLine("=====");
19
20             Deck deck = new Deck();
21
22             bool retry = true;
23             // Main game loop.
24             while (retry)
25             {
26                 Hand hand = deck.Deal(1)[0];
27                 Console.WriteLine();
28                 Console.WriteLine("Hand: " + hand);
29                 Deck.Score score = hand.ScoreHand();
30                 Console.WriteLine("You have: " + Hand.ScoreToString(score));
31                 Console.WriteLine();
32
33                 // Card swap loop.
34                 bool parseError = true;
35                 while (parseError)
36                 {
37                     Console.WriteLine("Type card numbers from 1 - 5 separated "
38                     + "by space to swap (enter to skip)");
39                     string swap = Console.ReadLine();
40                     if (swap.Trim().Length == 0) break;
41                     uint[] indices = TryParse(swap);
42                     if (indices == null) continue;
43
44                     parseError = false;
45                     Card[] newCards = hand.SwapCards(indices);
46                     Console.WriteLine();
47                     Console.WriteLine("New cards:");
48                     foreach (Card card in newCards) Console.Write(" " + card);
49                     Console.WriteLine();
50
51                     Console.WriteLine();
52                     Console.WriteLine("New hand: " + hand);
53                     Deck.Score newScore = hand.ScoreHand();
54                     Console.WriteLine("You now have: "
55                     + Hand.ScoreToString(newScore));
56
57                     // Improvement comments.
58                     Console.WriteLine();
59                     if (newScore > score)
60                         Console.WriteLine("Nice improvement from "
61                         + Hand.ScoreToString(score).ToLower());
62                     else if (newScore == score)
63                         Console.WriteLine("That didn't help much.");
64                     else
65                         Console.WriteLine("You're supposed to swap to improve "
66                         + "your hand. Not ruin it.");
67
68                     Console.WriteLine();
69                 }
70
71                 Console.WriteLine("Do you want to try again? (Y/N) ");
72                 retry = Console.ReadLine().ToUpperInvariant().Equals("Y");
73                 hand.PutBack(); // Be nice and put the hand back into the deck.
74             }
75         }
76     }
77 }

```

```

76
77  /**
78  * Tries to parse a list of card indices. If any errors are detected it
79  * simply returns null.
80  */
81  public static uint[] TryParse(String input)
82  {
83      char[] delims = {' ', ',', '.'};
84      string[] parts = input.Split(delims, Hand.HAND_SIZE,
85          StringSplitOptions.RemoveEmptyEntries);
86
87      if (parts == null || parts.Length == 0)
88          return null;
89
90      List<uint> indices = new List<uint>(parts.Length);
91
92      // Parse each index.
93      foreach (string index in parts)
94      {
95          uint result;
96          if (!uint.TryParse(index.Trim(), out result)) return null;
97          result--; // Zero based index in code, but not in input.
98          if (result < 0 || result > Hand.HAND_SIZE) return null;
99          if (indices.Contains(result)) return null;
100         indices.Add(result);
101     }
102     return indices.ToArray();
103 }
104 }
105 }
106

```