Introduction to Object Oriented Programming and Building Revenue Management Models

1. Object oriented programming

The essence of object oriented programming is to create objects that are very much like robots. Just like a robot carries a certain amount of data (for example, its name, memory capacity, friends) and it performs certain actions (for example, clean, fix another robot, cook dinner), a software object carries a certain amount of data and performs certain actions. Also, an object can alter the data that itself or other objects carry by performing certain actions.

We begin by creating robot objects in JAVA. This is just for illustration and the analogy that we drew between robots and objects in the previous paragraph has nothing to do with the fact that we create robot objects. The following piece of code shows how we can create a simple robot object in JAVA. This robot object has a name and a health level (these are the data that the object carries) and it can perform the actions of saying its name and health level, changing its health level by a certain amount and attacking another robot (these are the actions that the object can perform).

```
package RevenueManagement.CrashCourse;

public class Robot
{
    // name of the robot
    String name;
    // health level of the robot
    int health;

    // populates a robot with a certain name and initial health level
    public Robot ( String name_ , int health_ )
    {
        name = name_;
        health = health_;
    }

    public void speak()
    {
        System.out.println( "Hi, my name is " + name + "." );
        System.out.println( "My health level is " + health + "." );
    }

    // changes the health of the robot by a certain amount
    public void changeHealth ( int increment )
    {
        health = health + increment;
    }

    // attacks a certain robot
    // returns the health level of the robot after the attack
    public int attack ( Robot opponent )
    {
```

```
        int opponentHealth = opponent.getHealth();
        // health variable below corresponds to this robot's health variable
        if ( opponentHealth <= health )
        {
            opponent.changeHealth( -5 );
        }
        else
        {
            changeHealth( -5 );
        }
        return health;
    }

    // returns the name of the robot
    public String getName()
    {
        return name;
    }

    // returns the health level of the robot
    public int getHealth()
    {
        return health;
    }
}
```

We can now start creating instances of a robot and use them. The following piece of code creates three robots whose names are john, marry and tom_walker, and makes them speak and attack each other.

```
package RevenueManagement.CrashCourse;

public class Main
{
    public static void main ( String[] args )
    {
        // john's initial health level is 100
        Robot john = new Robot ( "john" , 100 );
        Robot marry = new Robot ( "marry" , 105  );
        Robot tom = new Robot ( "tom_walker" , 70 );

        john.speak();
        System.out.println();

        // john attacks marry, marry is stronger
        john.attack( marry );
        john.speak();
        marry.speak();
        System.out.println();
```

```
        // john attacks tom, tom is stronger
        john.attack( tom );
        john.speak();
        tom.speak();
        System.out.println();
    }
}
```

The most important element in the piece of code above is the method signature

```
public static void main ( String[] args ).
```

Any executable JAVA program should have a method that starts with this signature and this method is the entry point for the program. That is, the program gets to be executed starting from this method. Once this method starts being executed, we can create objects and have them alter the data that they carry. Since the name of the first class is Robot, the first class should be included in a file named Robot.java. Similarly, the second class should be included in a file named Main.java. Noting the package name, these files should be in the directory path . . . \RevenueManagement\CrashCourse\. For illustration, we assume that these files are included in the directory path C:\huseyin\research\RevenueManagement\CrashCourse\. In this case, we can execute the JAVA program above by using the command

```
java -classpath C:\huseyin\research RevenueManagement.CrashCourse.Main
```

from DOS prompt. The output of the program is the following.

```
Hi, my name is john.
My health level is 100.

Hi, my name is john.
My health level is 95.
Hi, my name is marry.
My health level is 105.

Hi, my name is john.
My health level is 95.
Hi, my name is tom_walker.
My health level is 65.
```

The next thing we look at is how we can manage a large number of robots efficiently. For this purpose, we construct a robot keeper that keeps all the robots in an array. Whenever we populate a robot, we add it to the robot keeper so that all of our robots are kept in a list. Presumably, we do not need more than one instance of the class RobotKeeper, since we probably need only one list that keeps all the robots. This will give us a chance to illustrate the use of static data and methods. The following piece of code gives a possible way to implement a robot keeper.

```
package RevenueManagement.CrashCourse;

public class RobotKeeper
{
    //use an array of robots as a list
    Robot[] list;
    // current size of the list
    int size;
    // this keeps the unique instance of robot keeper
    static RobotKeeper instance;

    // populates a robot keeper with a certain capacity
    public RobotKeeper ( int capacity )
    {
        list = new Robot[ capacity ];
        size = 0;
        instance = this;
    }

    // adds a robot to the list if there is capacity
    public void addRobot ( Robot robot )
    {
        // if there is capacity add the robot and increment the size
        // so that the next added robot does not overwrite an earlier robot
        if ( size < list.length )
        {
            list[ size ] = robot;
            size ++;
        }
    }

    // returns the robot with a certain index
    public Robot getRobot ( int index )
    {
        return list[ index ];
    }

    // returns the unique instance of robot keeper
    public static RobotKeeper getInstance()
    {
        return instance;
    }
}
```

The most interesting thing in the piece of code above is the static modifier. If a variable in a class is tagged with the static modifier, then all instances of the class have access to the same instance of the variable. In this way, we can construct global variables that can be accessed easily from anywhere in the code. We note that the method getInstance() is also declared as static. By doing so, we allow ourselves to call this method through the method call RobotKeeper.getInstance() and we do not have to refer to a particular instance of the class

RobotKeeper. This is very powerful because whenever we need to access our robot keeper, which is probably unique, we can simply use the method call RobotKeeper.getInstance(). The following piece of code shows how we can utilize the robot keeper.

```
package RevenueManagement.CrashCourse;

public class Main
{
    public static void main ( String[] args )
    {
        // create a robot keeper with 21 robot capacity
        new RobotKeeper( 21 );
        // create 20 robots and add them to the robot keeper
        for ( int i = 0 ; i < 20 ; i ++ )
        {
            Robot cr = new Robot ( "Robot" + i , (int) ( 100 * Math.random() ) );
            RobotKeeper.getInstance().addRobot( cr );
        }

        Robot john = new Robot( "john" , 100 );
        RobotKeeper.getInstance().addRobot( john );

        Robot r05 = RobotKeeper.getInstance().getRobot( 5 );
        r05.speak();
        System.out.println();

        Robot r20 = RobotKeeper.getInstance().getRobot( 20 );
        r20.speak();
        System.out.println();
    }
}
```

The output of the program is the following.

```
Hi, my name is Robot5.
My health level is 75.

Hi, my name is john.
My health level is 100.
```

---

2. Reading from and writing to files

There are many ways to access files in JAVA. We use the class BufferedReader to read from files and the class PrintWriter to write to files. For illustration, we assume that we keep a list of robots in the following format in a text file. This file may be created by using any text editor.

```
[Robot_0] [700]
[Robot_1] [100]
[Robot_2] [500]
[Robot_3] [400]
[Robot_4] [200]
[Robot_5] [100]
[Robot_6] [700]
[Robot_7] [300]
[Robot_8] [700]
[Robot_9] [100]
[Robot_10] [200]
```

The first entry in each line is the name of the robot and the second entry is the initial health level. We ignore the square brackets when reading this file. The following class reads the file above, creates robots with the names and initial health levels in the file and puts them into the robot keeper.

```
package RevenueManagement.CrashCourse;

import java.io.*;
import java.util.*;

public class RobotReader
{
    // a buffered reader is used to read a file line by line
    BufferedReader bur;
    // a string tokenizer is used to split a line into tokens
    StringTokenizer tokenizer;
    // We will refer to these characters as deliminators and will not read them
    String dels;

    public RobotReader( File inFile )
    {
        try
        {
            bur = new BufferedReader ( new InputStreamReader ( new FileInputStream ( inFile ) ) );
            // we treat spaces, tabs and square brackets as deliminators
            dels = " \t []";
        }
        catch ( Exception e )
        {
            throw new Error ( "exception thrown " + e.getClass() + " " + e.getMessage() );
        }
    }

    // reads the robots from the input file, creates them and adds them to the robot keeper
    // assumes that a robot keeper has already been created
    public void read()
```

```
    {
        try
        {
            String line = bur.readLine();
            while ( line != null )
            {
                // tokenizer is getting ready to tokenize the line that has just been read
                tokenizer = new StringTokenizer( line , dels );
                // extract the robot name
                String name = tokenizer.nextToken();
                // extract the health level
                // note that string tokenizer always returns a string
                // we need to convert the string into integer
                String healthString = tokenizer.nextToken();
                int health = Integer.parseInt( healthString );
                // create the robot and add it to robot keeper
                Robot robotCreated = new Robot( name , health );
                RobotKeeper.getInstance().addRobot( robotCreated );
                // read next line
                line = bur.readLine();
            }
        }
        catch ( IOException e )
        {
            throw new Error ( "exception thrown " + e.getClass() + " " + e.getMessage() );
        }
    }
}
```

We use the class StringTokenizer to divide a line into words and we process each word one by one. We continue reading the file until the line we read is null. The following piece of code shows how we can use the robot reader.

```
package RevenueManagement.CrashCourse;

import java.io.*;

public class Main
{
    public static void main ( String[] args )
    {
       new RobotKeeper( 10 );
       // create the file to read the robots from
       File inFile =
        new File ( "c:\\huseyin\\research\\RevenueManagement\\CrashCourse\\Data\\list.txt" );
       RobotReader rr = new RobotReader( inFile );
       rr.read();
       // loop over all robots and make them speak
       for ( int i = 0 ; i < 10 ; i ++ )
       {
```

```
            Robot rc = RobotKeeper.getInstance().getRobot( i );
            rc.speak();
        }
    }
}
```

The following output of the program verifies that we can indeed read the file correctly.

```
Hi, my name is Robot_1.
My health level is 100.
Hi, my name is Robot_2.
My health level is 500.
Hi, my name is Robot_3.
My health level is 400.
Hi, my name is Robot_4.
My health level is 200.
Hi, my name is Robot_5.
My health level is 100.
Hi, my name is Robot_6.
My health level is 700.
Hi, my name is Robot_7.
My health level is 300.
Hi, my name is Robot_8.
My health level is 700.
Hi, my name is Robot_9.
My health level is 100.
Hi, my name is Robot_10.
My health level is 200.
```

To write to a file, we can use the class PrintWriter. Assuming that the file we would like to write information in is C:\huseyin\research\RevenueManagement\CrashCourse\Data\out.txt, we can create and use an instance of the class PrintWriter through the following piece of code. It is important to flush frequently so that the information is thoroughly written into the file even if the program is terminated abnormally without having a chance to close the files properly.

```
File outFile =
       new File ( "C:\\huseyin\\research\\RevenueManagement\\CrashCourse\\Data\\out.txt" );
PrintWriter pw =
       new PrintWriter ( new OutputStreamWriter ( new FileOutputStream ( outFile ) ) );
pw.println( "Robot 1 died." );
pw.flush();
```

---

3. Building revenue management models

In this section, we show how we can use the techniques in the previous section to build revenue management models. Most of the crucial code in this section is presented in pseudo code format. We begin by developing a class to represent a fare class. The following piece of code gives a sketch of a class that represents a fare class.

```
package RevenueManagement.SingleLeg.General;

public class FareClass
{
    double fare;
    // the largest possible demand that we can observe from this fare class
    int maxDemand;
    // pdf and cdf's of the demand from this fare class
    double[] pdf;
    double[] cdf;

    // construct a fare class with a certain fare
    // we deal with the maximum demand and the probability distribution for the demand later
    public FareClass ( double fare_ )
    {
        ...
    }

    // sets the pdf for this fare class
    // once pdf is set, maximum demand and cdf is automatically computed
    public void setProbability ( double[] pdf_ )
    {
        pdf = pdf_;
        maxDemand = pdf.length - 1;
        // compute cdf using pdf
        ...
    }

    public double getFare()
    {
        ...
    }

    public int getMaxDemand()
    {
        ...
    }

    public double[] getPDF()
    {
        ...
    }

    public double[] getCDF()
    {
        ...
    }
```

```
    // returns the inverse of the cdf at a given point
    // useful for EMSR, Littlewood type of computation and
    // generating samples from the demand distribution
    public int getCDFInverse ( double r )
    {
        for ( int i = 0 ; i < cdf.length ; i ++ )
        {
            if ( r < cdf[ i ] )
            {
                return i;
            }
        }
        ...
    }
}
```

The following class, which we refer to as the fare class keeper, is used to keep the fare classes in a list. It is very similar to the robot keeper.

```
package RevenueManagement.SingleLeg.General;

public class FareClassKeeper
{
    FareClass[] fcs;
    int size;
    static FareClassKeeper instance;

    public FareClassKeeper ( int noFareClasses )
    {
        ...
    }

    // adds a fare class to the fare class keeper
    public void addFareClass ( FareClass fc )
    {
        ...
    }

    // returns the fare class with a specific index
    public FareClass getFareClass ( int fc )
    {
        ...
    }

    // returns the number of fare classes
    public int getNoFareClasses()
    {
        ...
    }
```

```
    public static FareClassKeeper instance()
    {
        return instance;
    }
}
```

We also construct a class, which we refer to as the global parameter keeper, to keep a few global parameters. For our case, this class keeps only the flight capacity and an instance of the class PrintWriter to write some information into a file. Finally, we construct a problem reader class that reads the problem data from a file, creates all the fare classes and puts them into the fare class keeper. Global parameter keeper and problem reader are not very interesting. One interesting point in the implementation of the problem reader is that it automatically skips the lines that are blank or start with the character #. These lines are treated as comments. This is similar to the style of commenting in AMPL.

We adopt the following format to store the problem data.

```
# no of fare classes
4

# capacity
6

# fares, order is important and high (late arriving) fare comes first
100
90
80
70

# probabilities, order is important and high (late arriving) fare comes first
# the first entry is the maximum demand
1 0.5 0.5
2 0.3 0.3 0.4
3 0.1 0.4 0.2 0.3
2 0.3 0.5 0.2

# output file to dump any information that we like
C:\huseyin\research\RevenueManagement\SingleLeg\Data\out.txt
```

The problem parameters above represent a problem with four fare classes. The probabilities that there will be 0, 1 or 2 demands from the 4-th fare class are respectively 0.3, 0.5 and 0.2.

4. Dynamic programming formulation for the revenue management problem

In this section, we present a dynamic programming formulation of the revenue management problem that is slightly different from the formulation that we used before. Although there

is nothing wrong with the earlier formulation, our formulation here is more amenable to computer implementation.

We assume that there are n fare classes. We let p_j be the fare for fare class j and D_j be the demand for fare class j. We use x_j to denote the capacity remaining just before we observe the demand for fare class j. Since the n-th fare class arrives first, we always have $x_n = C$, where C is the capacity of the flight. We let z_j be the number of seats that we sell to fare class j. Clearly, z_j is a decision variable.

Given that we have x_j units of capacity remaining just before we observe the demand for fare class j and the demand for fare class j turns out to be D_j, the expected revenue obtained from the fare classes $j, j-1, \ldots, 1$ satisfy the equation

$$V_j(x_j, D_j) = \max_{z_j \leq \min\{x_j, D_j\}} p_j \, z_j + \mathbb{E}\big\{V_{j-1}(x_j - z_j, D_{j-1})\big\}, \tag{1}$$

with the boundary condition that $V_0(\cdot, \cdot) = 0$. We note the constraint $z_j \leq \min\{x_j, D_j\}$ ensures that the number of seats that we sell to fare class j does not exceed the remaining capacity and the demand for fare class j. An expectation of the form $\mathbb{E}\big\{V_j(x_j, D_j)\big\}$ can be written as

$$\mathbb{E}\big\{V_j(x_j, D_j)\big\} = \sum_{k=0}^{\infty} \mathbb{P}\big\{D_j = k\big\} \, V_j(x_j, k)$$

and the quantity on the right side above only depends on j and x_j. For notational brevity, we let $\bar{V}_j(x_j) = \mathbb{E}\big\{V_j(x_j, D_j)\big\}$, in which case (1) can be written as

with the boundary condition that $\bar{V}_0(\cdot) = 0$. Taking the expectations of both sides above, we obtain

In our model, we compute the value functions $\big\{\bar{V}_j(\cdot) : j = 1, \ldots, n\big\}$. Replacing the expectation on the right side above with a sum, we have

$$\bar{V}_j(x_j) = \sum_{k=0}^{\infty} \mathbb{P}\{D_j = k\} \left\{ \max_{z_j \leq \min\{x_j, k\}} p_j \, z_j + \bar{V}_{j-1}(x_j - z_j) \right\}.$$

For many cases, the demand random variable takes on only finitely many possible values and we can truncate the summation above. Once we compute the value functions $\{\bar{V}_j(\cdot) : j = 1, \ldots, n\}$, if we have x_j units of capacity left and we observe a demand of D_j for fare class j, then we solve the problem

$$\max_{z_j \leq \min\{x_j, D_j\}} p_j \, z_j + \bar{V}_{j-1}(x_j - z_j) \tag{2}$$

to decide how many seats we should sell to fare class j.

5. Computer implementation of the dynamic programming formulation
The following is a pseudo code for computing the value functions $\{\bar{V}_j(\cdot) : j = 1, \ldots, n\}$. We assume that the value functions are kept in a two-dimensional array, where the first index represents the fare class and the second index represents the capacity.

---

for $j = 1$ to n do
    for $x = 0$ to C do
        set $v[j][x] = 0$
        for $k = 0$ to the maximum value of demand for fare class j do
            given that the remaining capacity is x and the demand is k,
              find the best action z to take
              (for the moment denote this by $z =$ best action(j, x, k))
            set $v[j][x] = v[j][x] + \mathbb{P}\{D_j = k\} \times \{p_j \times z + v[j-1][x-z]\}$
        next k
    next x
next j

The following is the pseudo code for the method best action(j, x, k). Mathematically speaking, this is equivalent to finding the optimal solution to problem (2) after we replace x_j with x and D_j with k.

---

set best value $= -\infty$
set best action $= \infty$
for $z = 0$ to minimum of x and k do
    set current value $= p_j \times z + v[j-1][x-z]$
    if current value $\geq$ best value then
        set best value = current value
        set best action $= z$
    end if
next z
return best action

---

6. EMSR-a heuristic for the revenue management problem

In this section, we briefly review the EMSR-a heuristic for the revenue management problem. Our notation is closer to the notation of the book and is slightly different from the notation that we used before. Given a fare class j and another fare class $k < j$, we compute y_{jk} as

$$y_{jk} = F_k^{-1}\left(1 - \frac{p_j}{p_k}\right). \tag{3}$$

This is the number of seats that we want to protect for fare class k when we make the decisions for fare class j. Therefore, the total number of seats that we want to protect when we make the decisions for fare class j is

$$v_j = y_{j,j-1} + y_{j,j-2} + \ldots + y_{j1}.$$

That is, when we make the decisions for fare class j, we protect v_j seats for fare classes $j-1, j-2, \ldots, 1$. We have $v_1 = 0$ by definition.

We now consider the problem of making the seat allocation decisions by using the protection levels $\{v_j : j = n, \ldots, 1\}$. Given that v_j seats are protected for fare classes $j-1, j-2, \ldots, 1$, if the remaining capacity is x_j just before we observe the demand for fare class j, then we have $\max\{x_j - v_j, 0\}$ seats that can be used to cover the demand for fare class j. Consequently, if the demand for fare class j is D_j, then we sell

(4)

seats to fare class j. In this case, the capacity that we have left for fare class $j-1$ is

$$x_j - \min\{D_j, \max\{x_j - v_j, 0\}\}.$$

7. Computer implementation of EMSR-a heuristic

We note that the class FareClass includes a method that computes the inverse of the cumulative distribution function of the demand for this fare class at any given point. In particular, when we deal with the discrete cumulative distribution function $F(\cdot)$, $F^{-1}(r)$ is equal to k if and only if k satisfies

$$F(k-1) < r \leq F(k).$$

We note that for the continuous cumulative distribution function $F(\cdot)$, $F^{-1}(r)$ is equal to k if and only if k satisfies $F(k) = r$ and the condition above can be visualized as the extension of the condition $F(k) = r$ to the discrete case. Consequently, we can easily carry out the computation in (3) and the following is the pseudo code for computing the protection levels $\{v_j : j = n, \ldots, 1\}$. We assume that $\{y_{jk} : j = n, \ldots, 1,\ k = j-1, \ldots, 1\}$ are kept in a two-dimensional array and $\{v_j : j = n, \ldots, 1\}$ are kept in a one-dimensional array.

```
for j = n down to 1 do
    for k = j − 1 down to 1 do
        set y[j][k] = F_k^{−1}(1 − p_j/p_k)
    next k
next j
```

```
for j = n down to 1 do
    set v[j] = 0
    for k = j − 1 down to 1 do
        set v[j] = v[j] + y[j][k]
    next k
next j
```

8. Building a simulator

For any seat allocation policy, it is important that we build a simulator to simulate the performance of the policy. The simulator loops over the fare classes starting from the fare class that arrives first. For our case, this is the n-th fare class. For each fare class j, the simulator generates a sample of the demand from that fare class. Letting x_j be the remaining capacity that we have just before observing the demand for fare class j and $\hat{D}_j$ be the sample of the demand for fare class j, we then decide how many seats to sell to fare class j. In particular, if we use the dynamic programming formulation, then we solve problem (2) after replacing D_j with $\hat{D}_j$. The optimal solution z_j to this problem tells us how many seats we should sell to fare class j. We also note that this is equivalent to executing the method best action$(j, x_j, \hat{D}_j)$. Similarly, if we use the EMSR-a heuristic, then the number of seats that we sell to fare class j is $z_j = \min\{\hat{D}_j, \max\{x_j - v_j, 0\}\}$. In either case, the revenue obtained by the policy increases by $p_j\, z_j$, the remaining capacity just before we observe the demand for fare class $j-1$ goes down to $x_j - z_j$ and we repeat the whole process again until we reach the first fare class. Of course, we make multiple simulation runs to get a feel for the expected revenue. The following is the pseudo code for the simulator.

```
for i = 1 to maximum number of iterations do
    set revenue = 0
    set capacity = C
    for j = n down to 1 do
        sample the demand for fare class j
        given the capacity and the demand sample, let z be the number of
          seats sold to fare class j
        set capacity = capacity − z
        set revenue = revenue + p_j × z
    next j
    report revenue
next i
```

One final gap is how to generate a sample of the demand for fare class j. If the cumulative distribution function of the demand for fare class j is $F_j(\cdot)$ and $\hat{U}$ is a sample from the uniform distribution between 0 and 1, then $F_j^{-1}(\hat{U})$ is a sample of the demand for fare class j. The standard JAVA class Random can be used to generate samples from the uniform distribution between 0 and 1. Generating samples of different random variables is a topic that is extensively studied in simulation courses.