3.7. Case Study 1: Pizza Delivery 2U

In this case study we look at how to design a class from scratch for both a specific purpose, and so that it can be used more widely later. This dual requirement is a common one and, if met, enables us to get the maximum benefit from our programming effort. We start with a concrete specification of the problem, with a reasonable solution, and then move towards a more durable product. In the process, we shall learn more about the use of objects in both for-loops and parameters.

Problem

Pizza Delivery 2U is interested in giving more accurate delivery times to its customers when they order pizzas. From the time of the call to the pizza arriving, there are two factors to be taken into account: the length of the queue at the pizza oven (i.e. the number of orders already in the system), and the area in which the customer lives. Customers in areas close by will have faster deliveries than those in areas further away. Pizza 2U makes deliveries in four identifiable areas, and would like to have a quick look-up table next to the phone for times from 9am to 11pm and queue lengths from 0 to 5, in these four areas.

Solution

This sounds like a tall order, but in fact, we can bring considerable experience already to our assistance:

- recognizing repetition and controlling it through methods and loops;
- knowing how to print a table, as in Example 3.3 (Temperature conversion);
- knowing how to convert times and add them, as in Example 2.6 (Fleet timetables).

With these in hand we can break the solution down to printing one table for area *x*, four times (*x* varying from 1 to 4). Then the table itself will show queue lengths across the top and the times in the given range down the side. If we take times in 15 minute intervals, we shall get a table something like the following:

```
Pizza 2U Delivery estimates for Area 1
=======================================
Time now          Queue length
0     1     2     3     4     5
900      915   920   925   930   935   940
915      930   935   940   945   950   955
930      945   950   955   1000  1005  1010
945      1000  1005  1010  1015  1020  1025
...
```

Initial design

Attacking the problem from the top, we can postulate the main part of the program as drawing a table consisting of headings and then a number of lines, and this being repeated for each of the four areas. Each line is handled by a `printaLine` method that we do not specify yet, and the only real thinking we need to do is for the loop that controls `printaLine`. Since we want rows labelled for times from 9.00 to 23.00, in intervals of 15 minutes, this is not so difficult. The constructor for the first version of the program is:

```
PizzaDelivery1 () {
  for (int area = 1; area<= 4; area++) {
    printHeading(area);
                                          startTable();
    for (int time = 900; time < 2300; time = addTime(time,15))
      printaLine(time, area);
    }
  }
```

The methods that we now have to fill in are underlined. The first two concern headings, and are simple enough. Let us next look at `printaLine`. It has a much simpler structure than the similar method, `outaLine`, in the temperature table program, because there is only one value per column and each row starts with a provided time. The loop will run over the queue lengths (0 to 5, as specified in the problem formulation) and all we have to do is work out what value gets printed in the column. The answer is that it is the time of the order plus all the unavoidable times for

- waiting in a queue – 5 minutes times the length of the queue;
- cooking the pizza – 5 minutes;
- handling the order – 10 minutes;
- delivering it – 5 minutes per area crossed.

After some thought, we can come up with the following formula for one value in the table:

```
addTime (orderTime, queue*makeTime + processTime + a*driveTime)
```

In other words, we postulate an `addTime` method. The method will be typed in order to return the new time.

Now we can work on `addTime`. It takes a time, represented as an integer but for a 24 hour clock, and adds minutes to it. So the method header must be:

```
int addTime (int t, int m) {
```

We deliberately use `t` as the formal parameter identifier, to keep the method very general. Now we can capitalize on the work already done in Example 2.6 in converting times. There were two statements that converted times from the 24 hour clock to minutes and back again. These were:

```
arriveMins = arrive24 / 100 * 60 + arrive24 % 100;
newArrive24 = newArriveMins / 60 * 100 + newArriveMins % 60;
```

If we formulate these expressions as methods, they become:

```
int timeInMins (int t) {
  return t / 100 * 60 + t % 100;
}

int time24 (int t) {
  return t / 60 * 100 + t % 60;
}
```

Now the `addTime` method can take the easy path: it calls one of the others to convert the given time to minutes, then adds the minutes specified, then converts the whole thing back again. This is done in:

```
int addTime (int t, int m) {
  return time24(timeInMins(t) + m);
}
```

Of course, with such a simple method, we could have just replaced its contents wherever we needed to, so that

```
addTime (time,15)
```

would have become

```
time24(timeInMins(time) + 15)
```

However, the use of the identifier `addTime` adds to the readibility of the program, and we do use the method more than once, so it is worthwhile.

Model diagram

The model diagram for the program is shown in Figure 3.5. The arrows in the method section indicate who calls whom. Unlike the curio store programs, there are no additional objects in this program, the focus being on methods. But the model diagram does tell us quite a bit apart from providing a neat summary of the methods and their parameters, as it shows the calling structure as well. Once the calling structure becomes more complicated than this, we can create a separate diagram, as shown in Figure 3.6.

Figure 3.5. Model diagram for `PizzaDelivery1` .

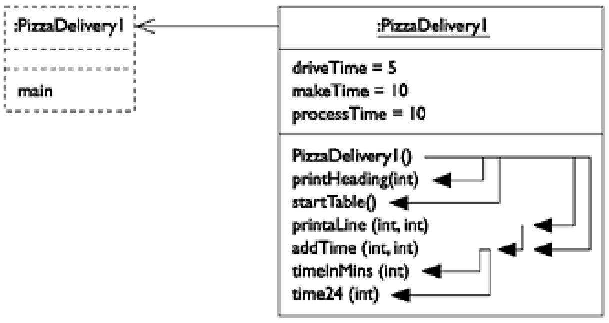
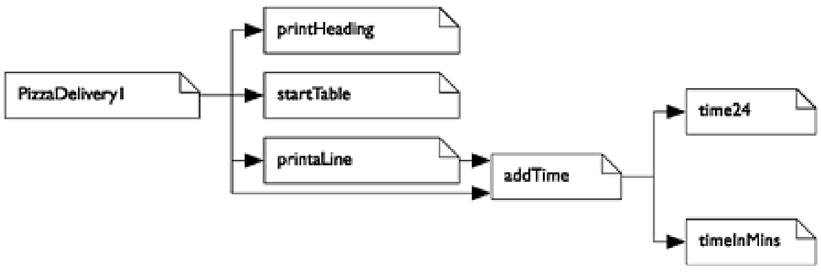


Figure 3.6. Call model diagram of `PizzaDelivery1` methods.



Program 1

Finally, we can put the whole program together in Java itself:

```
class PizzaDelivery1 {

  /* The Pizza Delivery Program version 1    J M Bishop  May 2000
   * ====================================
   *
   * Works out the time for delivering pizzas based on
   * queue length and area of delivery.
   * Illustrates reuse of methods, method calling and loops
   * Test version with 1 area and times from 900 to 1300
   */

  int driveTime = 5;    // per area;
  int makeTime = 5;     // in a queue
  int processTime = 10; // per order

  PizzaDelivery1 () {
    for (int area = 1; area<= 1; area++) {
      printHeading(area);
```

```
        startTable();
        for (int time = 900; time < 1300; time=addTime(time,15))
          printaLine(time, area);
      }
    }

    void printaLine (int t, int a) {
      System.out.print(t+"\t");
      for (int queue = 0; queue <= 5; queue++)
        System.out.print(addTime(t,
          queue*makeTime+processTime+a*driveTime) + "\t");
      System.out.println();
    }

    int timeInMins (int t) {
      return t / 100 * 60 + t % 100;
    }

   int time24 (int t) {
      return t / 60 * 100 + t % 60;
    }

    int addTime (int t, int m) {
      return time24(timeInMins(t)+m);
    }

    void printHeading (int area) {
      System.out.println("Pizza 2U Delivery estimates for "+
        Area "+area);
      System.out.println("================================="+
       "=====\n");
    }

    void startTable () {
      System.out.println("Time now        Queue length");
      System.out.print("        ");
      for (int queue = 0; queue <=5; queue++)
        System.out.print("  "+queue+"  ");
      System.out.println();
    }

    public static void main (String [] args) {
      new PizzaDelivery1 ();
    }
  }
```

Testing

The program has been set up to only print a table for one area, and to stop the times at 1300, just to keep the table manageable. The output looks like this:

```
Pizza 2U Delivery estimates for Area 1
=======================================
Time now        Queue length
0     1     2     3     4     5
900      915   920   925   930   935   940
915      930   935   940   945   950   955
930      945   950   955   1000  1005  1010
945      1000  1005  1010  1015  1020  1025
1000     1015  1020  1025  1030  1035  1040
1015     1030  1035  1040  1045  1050  1055
1030     1045  1050  1055  1100  1105  1110
```

```
1045     1100  1105  1110  1115  1120  1125
1100     1115  1120  1125  1130  1135  1140
1115     1130  1135  1140  1145  1150  1155
1130     1145  1150  1155  1200  1205  1210
1145     1200  1205  1210  1215  1220  1225
1200     1215  1220  1225  1230  1235  1240
1215     1230  1235  1240  1245  1250  1255
1230     1245  1250  1255  1300  1305  1310
1245     1300  1305  1310  1315  1320  1325
```

How do we check the results? We do it in two ways. Firstly, we take the statements in the program and reason about them. This we have more or less done during the development. The other way of checking a program like this is to do some hand calculations. Take a random spot in the table, e.g. at 1100 with a queue of 2. The delivery time can be calculated as:

1100 + (2*5 + 10 + 1*5) = 1100 + 25 = 1125

which tallies with the results. More checking should be done at the borders of the table, e.g. at the end of a row, and also when the time goes over the hour, as in the row starting with 1145.

The problem revisited

Given the above investigation, we would like to assess whether a class for times would be a good idea. Although using integers to represent times did work, it was incorrect, because ultimately 915 did not mean nine hundred and fifteen minutes, but nine hours and 15 minutes. It would be better to represent a time as two integers – the hours and the minutes. This implies that we should have a class for times.

So the task is to construct such a class, and in order to test it out, we shall use it in our existing Pizza Delivery program. This way we can compare outputs to ensure they remain the same, and we can assess how much more complex using a class makes the programming. The fact is that the program will look slightly more complex, but it will have been built in a more structured way and will be more maintainable and flexible.

Java's `Date` and `Calendar` classes

Before embarking on the development of a class for times, we should look to see whether Java itself provides one. We do not wish to duplicate what has already been set up. If we go to Java's online documentation and search for 'time', we find that it occurs first in a `Date` class. `Date` has only a few methods, which are shown in the form below. It represents the time as the number of milliseconds since 1 January 1970. It will not be convenient to create times if we have first to calculate milliseconds.

Java's `Date` class

```
Date ()
Date (long Date)
long getTime ()
void setTime(long
                                                time)
boolean after (Date when)
boolean before (Date when)
int compareTo (Date anotherDate)
// plus the usual methods for toString etc.
```

Then Java also has a `Calendar` class which has extensive methods for handling dates and times, and for doing arithmetic on them, but not for handling times alone: a time must always be prefaced by a year, month and day. As such, `Calendar` is more elaborate than we need. We can therefore go ahead with the design of a `Time` class, confident that we are not duplicating effort.

Class design for `Time`

In designing a class, we look at

- the variables that make up its object;
- its constructors;
- the methods required by programs that might use it.

Looking at the variables, we find that there is no need for anything beyond the hour and minute for the purposes we have in mind. Of course, the class could be extended to cope with seconds, and this is taken up in the Problems.

The constructors are interesting. We need to supply the hour and minute information to the class. The class can, if necessary, deduce the hour and minute from a variety of formats. These could be:

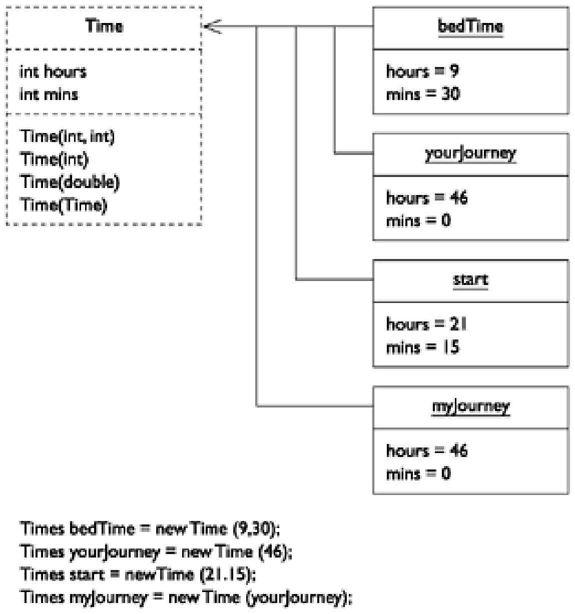
- two integers – the hour and minute, e.g. (11, 30);
- one integer, where the first hundreds and thousands are used for the hours, as we did before, e.g. 1130;
- a real number, where the fractional part represents the minutes, e.g. 11.30.

Constructors can be provided for all of these. Finally, we recognize that one time can also be created from another time, which gives us a fourth constructor. These are, then:

```
class Time {
  int hour;
  int min;

  Time (int h, int m) {
    hour = h;
    min = m;
  }

  Time (int mins) {
    hour = mins / 60;
    min = mins % 60;
  }

  Time (double t) {
    hour = (int) t;
    min = (int) t*100 - hour*100;
  }

  Time (Time t) {
    hour = t.hour;
    min = t.min;
  }
  ...  and so on
```

Figure 3.7 shows the model diagram for `Time` together with some objects created using the different constructors.

Figure 3.7. Model diagram of the construction of some `Time` objects.



The first method to be developed is going to be `addTime`. The question is: how does `addTime` look in a class, and how would it be called? The answer is: quite differently from before, in all respects. We have already seen that when a class is instantiated to create an object, method calls for that object use the dot operator. In the case of the time to which we are adding minutes, the time itself is the object in question. Therefore the call

```
time = addTime(time,15)
```

will become

```
time.addTime(15)
```

The object will be altered by adding the parameter of 15 minutes to its data. From this, we can deduce that `addTime` is going to be a void method. The addition of the minutes itself turns out to be simpler than before, because we already have the time split into its constituent parts:

```
void addTime (int m) {
  min += m;
  hour+= min / 60;
  min %= 60;
}
```

The method makes good use of the special assignment operators, which update the left-hand side variable. Thus the first statement could read: 'add m to min' or 'min becomes min plus m'.

What other methods are required? Well, we shall want to display a time, and could set up a `write` method, but in general it is not good practice to have output statements in general classes such as `Time`. Instead, we shall define a `toString` method, which can be automatically called by `println`, as described in Section 3.1. There may be other methods, but we shall add them after considering how to test `Time` in a program.

Testing the class

It is useful to be able to test a new class with a program that already works. So we take our first Pizza Delivery program and rework it. Looking at the original program, we can see that the changes are going to occur where times are mentioned, which are underlined in the following extract:

```
                                         PizzaDelivery1 () {
  .. omit some lines
  for (int time = 900; time < 1300; time=addTime (time, 15))
      printaLine(time, area);
```

```
    }
  }

  void printaLine (int t, int a) {
    System.out.print(t+"\t");
    for (int queue = 0; queue <= 5; queue++)
      System.out.print(addTime(t,
        queue*makeTime + processTime + a*driveTime) + "\t");
    System.out.println();
  }
```

For-loops with objects

The first change is that we rename the program, so as to keep both versions. The crucial next question is how do we loop over times, when they are objects? It turns out that the for-loop is far more powerful than just an integer counter, and its general form is:

For-loop

```
for (start; check; update) {
  body
}
```

start and *update* are assignments, and *check* is some condition based on the variables mentioned in *start* and *update*.

The loop begins by executing the *start* statement, then the *check*. If the *check* is true, do the *body* followed by *update* and *check*. When *check* returns false, exit the loop.

In other words, the start statement does not have to assign a number to an integer: it can be any appropriate initialization. So let us define two time objects to start with, as:

```
Time open  = new Time (9.00);
Time close = new Time (13,0);
```

We have deliberately used two different constructor calls here, just to emphasize their use. The first sends over a real number, and the second uses two parameters. Now, the start of the loop is simply

```
Time ofDay = open
```

where `ofDay` is going to be the loop variable. Working with the translation that we have already made from integers to objects, the update section of the loop is:

```
ofDay.addTime(15)
```

How do we specify the condition? The obvious formulation is

```
ofDay < close    // won’t work
```

but as we discussed in the previous section, we cannot use conditional operators such as < on objects. The condition has to be defined and implemented as a method, so that the call would be

```
ofDay.lessThan(close)
```

where `lessThan` returns a boolean value based on a condition that it evaluates. We can now write out the full loop with its body, as:

```
for (Time ofDay = open; ofDay.lessThan(close); ofDay.addTime(15)) {
  printaLine(ofDay, area);
}
```

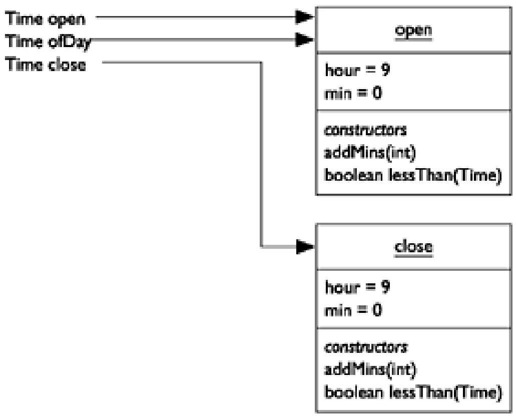
As with any other for-loop, ofDay starts at open, then starts the cycle of

- check against `close` ;
- execute the body;
- update with 15 mins by calling `addTime` .

A model diagram showing the objects of the for-loop is given in Figure 3.8. As ofDay.addTime is called each time round the loop, 15 minutes gets added to the ofDay object. However, as can be seen from the diagram, ofDay is the same object as open. Thus open will change as well. If open changes like this, we will need to recreate it for each area table. Instead, what we need to do is to create a copy of open for ofDay, which can be done easily through our fourth constructor, and the loop becomes:

```
for (Time ofDay = new Time(open); ofDay.lessThan(close);
       ofDay.addTime(15)) {
  printaLine(ofDay, area);
}
```

Figure 3.8. Model diagram of an object loop for times.



This is an excellent example of how the diagrams have shown up a problem, which we can then remedy in the Java implementation.

Objects as parameters

Moving on to printaLine, it would seem that the translation of the call to addTime would be simple, and the equivalent for-loop would be:

```
for (int queue = 0; queue <= 5; queue++) {
  t.addTime(queue*makeTime + processTime + a*driveTime);
  System.out.print(t + "\t");
}
```

If we put a print statement in this loop and show the values of the four times we are dealing with, we will get a trace of the values of the variables concerned as the two nested loops – one in the constructor and one in printaLine.

```
open      close     ofDay     t          queue
09:00     13:00     09:00     09:00      0
09:00     13:00     09:15     09:15      0
09:00     13:00     09:35     09:35      1
09:00     13:00     10:00     10:00      2
09:00     13:00     10:30     10:30      3
```

```
09:00       13:00       11:05       11:05         4
09:00       13:00       11:45       11:45         5
09:00       13:00       12:00       12:00         0
09:00       13:00       12:15       12:15         1
```

If we use this loop and run the program, the table part of the output will be:

```
9:0       9:15       9:35       10:0       10:30       11:5       11:45
12:0      12:15      12:35      13:0       13:30       14:5       14:45
```

which is wrong. The second line should begin with the time at 9:15, not 12:00. The 12:00 has obviously crept in by `ofDay`, an actual parameter to `printaLine`, being contaminated by `t`, its formal parameter.

Here we see the effect of passing objects as parameters, and how it is different from passing primitive typed values. In `PizzaDelivery1`, when the integer time value was passed into `printaLine`, the method received a copy of the value, and could work on it, without disturbing the original. `t` did change on each line, getting the six different values required for the table. But back in the constructor, it still had its original value for the order time, which was then incremented and became the start of the next line.

However, when an object is passed, its value is not copied, only a reference to where the original is. So any changes to the formal parameter affect the actual parameter directly. To obtain the effect we need, though, is quite simple: we must remember to make a local copy of the parameter inside the method ourselves. This is where our fourth constructor – often called a **copy constructor** – comes in again. The correct method is:

```
void printaLine (Time t, int a) {
  Time ofDelivery;
  System.out.print(t+"\t");
  for (int queue = 0; queue <= 5; queue++) {
    ofDelivery = new Time(t);
    ofDelivery.addTime(queue*makeTime + processTime + a*driveTime);
    System.out.print(ofDelivery + "\t");
  }
  System.out.println();
}
```

Each time we want to enter a new value in the table, we create a new time object with a copy of `t` in it, alter it, print it out, and then Java disposes of it for us when `ofDelivery` is reset again to `t`. The trace now becomes

```
open      close       ofDay       t           ofDelivery  queue
09:00     13:00       09:00       09:00       null        0
09:00     13:00       09:00       09:00       09:15       0
09:00     13:00       09:00       09:00       09:20       1
09:00     13:00       09:00       09:00       09:25       2
09:00     13:00       09:00       09:00       09:30       3
09:00     13:00       09:00       09:00       09:35       4
09:00     13:00       09:00       09:00       09:40       5
09:00     13:00       09:15       09:15       null
09:00     13:00       09:15       09:15       09:30       0
09:00     13:00       12:15       09:15       09:35       1
```

which can be checked as correct.

Program 2

Finally we list the entire class and test program, all put together:

```
class PizzaDelivery2 {

  /* The Pizza Delivery Program version 2    J M Bishop  May 2000
   * ===================================
   *
```

```
 * Works out the time for delivering pizzas based on
 * queue length and area of delivery.
 *
 * Illustrates the design of a Time class and the use of such
 * objects in methods and for-loops.
 */

int driveTime = 5;    // per area;
int makeTime = 5;     // in a queue
int processTime = 10; // per order

PizzaDelivery2 () {
  // Set the loop for one area only
  for (int area = 1; area<= 1; area++) {
   printHeading(area);
   startTable();
   Time open = new Time (9.00);
   Time close = new Time (13,0);
   for (Time ofDay = open; ofDay.lessThan(close);
          ofDay.addTime(15)) {
      printaLine(ofDay, area);
   }
  }
}

void printaLine (Time t, int a) {
  Time ofDelivery;
  System.out.print(t+"\t");
  for (int queue = 0; queue <= 5; queue++) {
    ofDelivery = new Time(t);
    ofDelivery.addTime(queue*makeTime+processTime+a*driveTime);
    System.out.print(ofDelivery + "\t");
  }
  System.out.println();
}
class Time {
  int hour;
  int min;

  Time (int h, int m) {
    hour = h;
    min = m;
  }

  Time (int mins) {
    hour = mins / 60;
    min = mins % 60;
  }
  Time (double t) {
    hour = (int) t;
    min = (int) t*100 - hour*100;
  }

  Time (Time t) {
    hour = t.hour;
    min = t.min;
  }

  void addTime (int m) {
    min +=m;
    hour+= min / 60;
    min %= 60;
  }
```

```
      boolean lessThan (Time t) {
        return hour < t.hour | (hour==t.hour & min < t.min);
      }

      public String toString () {
        return hour + ":" + min + " ";
      }
    }

    void printHeading (int area) {
      System.out.println("Pizza 2U Delivery estimates for "+
        "Area "+area);
      System.out.println("=================================="+
        "=====\n");
    }
    void startTable() {
      System.out.println("Time now        Queue length");
      System.out.print("        ");
      for (int queue = 0; queue <=5; queue++)
        System.out.print("  "+queue+"  ");
      System.out.println();
    }

    public static void main (String [] args) {
      new PizzaDelivery2 ();
    }
  }
```

The output is:

```
Pizza 2U Delivery estimates for Area 1
=======================================
Time now Queue length
0        1        2        3        4        5
9:0      9:15     9:20     9:25     9:30     9:35     9:40
9:15     9:30     9:35     9:40     9:45     9:50     9:55
9:30     9:45     9:50     9:55     10:0     10:5     10:1
9:45     10:0     10:5     10:10    10:15    10:20    10:2
10:0     10:15    10:20    10:25    10:30    10:35    10:4
10:15    10:30    10:35    10:40    10:45    10:50    10:5
10:30    10:45    10:50    10:55    11:0     11:5     11:1
10:45    11:0     11:5     11:10    11:15    11:20    11:2
11:0     11:15    11:20    11:25    11:30    11:35    11:4
11:15    11:30    11:35    11:40    11:45    11:50    11:5
11:30    11:45    11:50    11:55    12:0     12:5     12:1
11:45    12:0     12:5     12:10    12:15    12:20    12:2
12:0     12:15    12:20    12:25    12:30    12:35    12:4
12:15    12:30    12:35    12:40    12:45    12:50    12:5
12:30    12:45    12:50    12:55    13:0     13:5     13:1
```