LABORATORY 1

Logbook ADT

OBJECTIVES

In this laboratory, you

- examine the components that form an abstract data type (ADT) in Java.
- implement a programmer-defined ADT in Java.
- create a method that displays a logbook in calendar form.
- investigate how to overload methods in Java.

OVERVIEW

Because it is a pure object-oriented programming language, all Java programs contain one or more class (or ADT) definitions. Java defines many built-in classes and hundreds of methods. The purpose of this laboratory is for you to review how you can implement an abstract data type (ADT) of your own design while utilizing some of the built-in ADTs already implemented in Java. We use a monthly logbook as our example ADT. A **monthly logbook** consists of a set of entries, one for each day of the month. Depending on the logbook, these entries might denote a business's daily receipts, the amount of time a person spent exercising, the number of cups of coffee consumed, and so forth. A typical logbook is shown below.

February 2002						
					1 100	**2** 95
3 90	**4** 0	**5** 150	**6** 94	**7** 100	**8** 105	**9** 100
10 100	**11** 50	**12** 110	**13** 110	**14** 100	**15** 125	**16** 110
17 0	**18** 110	**19** 0	**20** 125	**21** 100	**22** 110	**23** 115
24 111	**25** 0	**26** 50	**27** 110	**28** 125		

When specifying an ADT, you begin by describing the **elements** (or attributes) that the ADT consists of. Then you describe how these ADT elements are organized to form the ADT's overall structure. In the case of the monthly logbook abstract data type—or Logbook ADT, for short—the elements are the entries associated with the days of the month and the structure is linear:

the entries are arranged in the same order as the corresponding days. In Java these elements are called the **data members** of the ADT (or class).

Having specified the ADT's data members, you then define its behavior by specifying the **operations** that are associated with the ADT. For each operation, you specify what conditions must be true before the operation can be applied (its requirements or **precondition**) as well as what conditions will be true once the operation has completed (its results or **postcondition**). The Logbook ADT specification below includes operations (or **methods** in Java) that create a logbook for a given month, store/retrieve the logbook entry for a specific day, and provide general information about the month.

Logbook ADT

Elements

A set of integer values for a logbook month and its associated calendar.

Structure

Each integer value is the logbook entry for a given day of the month. The number of logbook entries varies depending on the month for which data is being recorded. We will refer to this month as the **logbook month**. Each logbook month is actually a calendar month for a particular year. Thus each logbook month starts on a particular day of the week and has a fixed number of days in that month based on our Gregorian calendar.

Constructor

```
Logbook ( int month, int year )
```
Precondition:
Month is a valid calendar month between 1 and 12 inclusive.
Postcondition:
Constructor. Creates an empty logbook for the specified month—that is, a logbook in which all the entries are zero. If month is an invalid value, it will default to today's date.

LABORATORY 1: Prelab Exercise

Name ____

Hour/Period/Section ____

Date ____

The Logbook ADT specification provides enough information for you (or other programmers) to design and develop programs that use logbooks. Before you can begin using logbooks in your Java programs, however, you must first create a Java implementation of the Logbook ADT.

You saw in the Overview that an ADT consists of a set of elements and a set of operations that manipulate those elements. A Java **class** usually consists of a set of **data members** (or elements) and a set of **member methods** (or operations) that manipulate the data members. Thus, classes are a natural means for implementing ADTs.

How do you create a definition for a Logbook class from the specification of the Logbook ADT? You begin with the ADT elements and structure. The Logbook ADT specification indicates that you must maintain the following information about each logbook:

- the (month, year) pair that specify a particular logbook month
- the array of logbook entries for the month
- a calendar facility primarily for determining leap years and day-of-week on which the first day of the month falls

This information is stored in the data members of the Logbook class. The month and year are stored as integer values, the entries are stored as an array of integers, and the calendar facility will be based on Java's built-in GregorianCalendar class, which is derived (or inherited) from Java's Calendar class. We won't go into all the details of inheritance at this time, but because the GregorianCalendar class inherits from the Calendar class, an instance of the GregorianCalendar class can use all public and protected methods and variables in the Calendar class. This illustrates one big advantage of object-oriented programming—the ability to reuse existing ADTs instead of always writing your own.

```
class Logbook
{
    // Data members
    private int logMonth,                     // Logbook month
                logYear;                      // Logbook year
    private int[] entry = new int[31];        // Array of Logbook entries
    private GregorianCalendar logCalendar;    // Java's built-in Calendar class
}
```

By declaring the data members to be **private**, you prevent nonmember methods—that is, methods that are not members of the Logbook class—from accessing the logbook data directly. This restriction ensures that all references to the logbook data are made using the operations (or methods) in the Logbook ADT.

Having specified how the logbook data is to be stored, you then add definitions for the member methods corresponding to the operations in the Logbook ADT. These methods are declared as **public**. They can be called by any method—either member or nonmember—and provide a **public interface** to the logbook data. An incomplete definition for the Logbook ADT is given below. Note that it lacks implementation code for the class methods.

```
class Logbook
{
    // Data members
    private int logMonth,                       // Logbook month
                logYear;                        // Logbook year
    private int[] entry = new int[31];          // Array of Logbook entries
    private GregorianCalendar logCalendar;      // Java's built-in Calendar class

    // Constructor
    public Logbook ( int month, int year )      // Create a logbook
    {                                }

    // Logbook marking operations/methods
    public void putEntry ( int day, int value ) // Store entry for day
    {                                }
    public int getEntry ( int day )             // Return entry for day
    {                                }

    // General operations/methods
    public int month ( )                        // Return the month
    {                                }
    public int year ( )                         // Return the year
    {                                }
    public int daysInMonth ( )                  // Number of days in month
    {                                }

} // class Logbook
```

You need to know whether a given year is a leap year in order to determine the number of days in the month of February. To determine this information, a **facilitator method** (or helper method) has been added to the definition of the Logbook class. Note that the facilitator method is *not* an operation listed in the specifications for the Logbook ADT. Thus, it is included as a private member method rather than as part of the public interface. This facilitator method `leapYear()` can be implemented as follows using the built-in GregorianCalendar method for the Logbook class data member `logCalendar`.

```
return ( logCalendar.isLeapYear(logYear) );
```

Our current version of the incomplete definition for the Logbook class is shown as follows. Notice this version includes Java import statements for each of the built-in packages being used by the Logbook class. This incomplete definition is stored in the file *Logbook.java*.

```
import java.io.*;                     // For reading (keyboard) & writing (screen)
import java.util.*;                   // For GregorianCalendar class

class Logbook
{
    // Data members
    private int logMonth,                         // Logbook month
                logYear;                          // Logbook year
    private int[] entry = new int[31];            // Array of Logbook entries
    private GregorianCalendar logCalendar;        // Java's built-in Calendar class

    // Constructor
    public Logbook ( int month, int year )        // Create a logbook
    {                            }

    // Logbook marking operations/methods
    public void putEntry ( int day, int value )   // Store entry for day
    {                            }
    public int getEntry ( int day )               // Return entry for day
    {                            }

    // General operations/methods
    public int month ( )                          // Return the month
    {                            }
    public int year ( )                           // Return the year
    {                            }
    public int daysInMonth ( )                    // Number of days in month
    {                            }

    // Facilitator (helper) method
    private boolean leapYear ( )                  // Leap year?
    {                            }

} // class Logbook
```

This incomplete Logbook class definition provides a framework for the Logbook class. You are to fill in the Java code for each of the constructors and methods where the implementation braces are empty, or only partially filled (noted by "add code here …"). For example, an implementation of the `month()` method is given below.

```
public int month ( )
// Precondition: None.
// Postcondition: Returns the logbook month.
{
      return logMonth;
}
```

As you complete the class definition for the Logbook ADT, save your implementation of the member methods in the file *Logbook.java*.

The code in the file *Logbook.java* forms a Java implementation of the Logbook ADT. The following applications program uses the Logbook ADT to record and output a set of logbook entries. Note that this program is stored in its own file called *Coffee.java*. *Coffee.java* Class the application/driver class from where you can use your LogBook ADT

```
import java.io.*;

class Coffee
{
  // Records coffee intake for January 2002.
  public static void main ( String args[] ) throws IOException
  {
      int day;                  // Day loop counter

      // Coffee intake for January 2002
      Logbook coffee = new Logbook(1, 2002);

      // Record entries for the 1st and 15th of January 2002
      coffee.putEntry(1, 5);
      coffee.putEntry(15, 2);

      // Output the logbook entries.
      System.out.println("Month/Year : " + coffee.month() + "/" + coffee.year());
      for ( day = 1 ; day <= coffee.daysInMonth() ; day++ )
          System.out.println(day + " : " + coffee.getEntry(day));
  } // main( )

} // class Coffee
```

The statement

```
Logbook coffee = new Logbook(1, 2002);
```

invokes the Logbook class constructor to create a logbook for January 2002. Notice that in Java each instance (or object) of a class is created by using the `new` operator. In this case `coffee` is an instance of the Logbook class. As implemented, the constructor begins by verifying that a valid month value has been received. Then the constructor creates a new `logCalendar` for January 1, 2001 and sets the `logMonth` to 1 and `logYear` to 2002. You can use the assignment operator to perform this task, as in the following code fragment.

```
public Logbook ( int month, int year )
// Constructs an empty logbook for the specified month and year.
// Note:  Unlike mankind, Java's built-in Calendar numbers months
//        from January = 0
{
    int j;   // Loop counter

    // Verify that a valid month value was entered
    // If not, setup logbook for today's date
    ...........
    ...........
    else
    {
        // Assumes a default DAY_OF_MONTH as first day of month
        logCalendar = new GregorianCalendar(year, month -1, 1);
```

```
            logMonth = month;
            logYear = year;
        }

        // Set each entry in the logbook to 0.
        ...........
        ...........
    }
```

Note that Java's GregorianCalendar class numbers the months starting with 0 for January through 11 for December. Since people usually number the calendar months starting with 1 for January through 12 for December, during the execution of the coffee program `logMonth` is assigned the value 1 for January, and the month value for the GregorianCalendar constructor is adjusted accordingly by setting the month parameter to *`month -1`*. Once the constructor has created the `logCalendar` and assigned values to `logMonth` and `logYear`, it sets each element in the `entry` array to 0 and returns.

A side note: In-lab Exercise 2 will provide more information on how to create a logbook that defaults to today's date. For now you may want to simply implement the error-handling part of this constructor (when the month value is not between 1 and 12, inclusive) by picking arbitrary default values for `logMonth` and `logYear`.

Having constructed an empty logbook, the coffee program then uses the `putEntry()` method to record a pair of logbook entries for the first and fifteenth of January. It then outputs the logbook using repeated calls to the `getEntry()` method, with the `month()` and `year()` methods providing output headings.

Step 1: Implement the member methods in the Logbook class. Base your implementation on the incomplete Logbook class definition given earlier.

Step 2: Save your implementation of the Logbook ADT in the file *Logbook.java*. Be sure to document your code.

LABORATORY 1: Bridge Exercise

Name	
Hour/Period/Section	
Date	

Test your implementation of the Logbook ADT using the program in the file *TestLogbook.java*. This program supports the following tests.

Test	Action
1	Tests the constructor and the month, year, and daysInMonth operations.
2	Tests the putEntry and getEntry operations.

Step 1: Complete the test plan for Test 1 by filling in the expected number of days for each month.

Step 2: Test your implementation of the Logbook ADT in the file *Logbook.java* by compiling and running your test program *TestLogbook.java*.

Step 3: Execute the test plan. If you discover mistakes in your implementation of the Logbook ADT, correct them and execute the test plan again.

Test Plan for *Test1* (constructor, month, year, and daysInMonth Operations)

Test case	Logbook month	No. days in month	Checked
Simple month	`1 2002`	`31`	
Month in the past	`7 1998`		
Month in the future	`12 2020`		
Current month			
February (not leap year)	`2 1999`		
February (leap year)	`2 2000`		
An invalid month	`13 2002`		

Step 4: Complete the test plan for Test 2 by filling in the input data and expected result for each test case. Use a logbook for the current month.

Step 5: Execute the test plan. If you discover mistakes in your implementation of the Logbook ADT, correct them and execute the test plan again.

Test Plan for *Test2* (putEntry and getEntry Operations)

Test case	Logbook entries	Expected result	Checked

Record entries for the first and fifteenth of the month	`2 100`
	`15 200`		

Record entries for the first and last day of the month			
Record entries for all the Fridays in the month			

Change the entry for the first day	`1 100`
	`1 300`		

LABORATORY 1: In-lab Exercise 1

Name ______________________________

Hour/Period/Section ______________________________

Date ______________________________

The entries in a logbook store information about a specific month. A calendar provides a natural format for displaying this monthly data. That is why the GregorianCalendar class was conveniently included as a data member in the Logbook class: namely, the data member `logCalendar`.

> **`void displayCalendar ( )`**
> **Precondition:**
> None.
> **Postcondition:**
> Outputs a logbook using the calendar format shown below. Note that each calendar entry includes the logbook entry for the corresponding day.

```
                              2 / 2002

Sun        Mon        Tue        Wed        Thu        Fri        Sat

                                                        1 100      2 95

 3 90       4 0        5 150      6 94       7 100      8 105      9 100

10 100     11 50      12 110     13 110     14 100     15 125     16 110

17 0       18 110     19 0       20 125     21 100     22 110     23 115

24 111     25 0       26 50      27 110     28 125
```

In order to produce a calendar for a given month, you need to know on which day of the week the first day of the month occurs. To do so you will need to implement the facilitator method `dayOfWeek()` described below.

> **`int dayOfWeek ( int day )`**
> **Input parameter:**
> Day is a specific day in the logbook month.
> **Returns:**
> An integer denoting the day of the week on which the specified day occurs, where 0 corresponds to Sunday, 1 to Monday, and so forth.

First, you will need to set the logbook calendar to the day of the month specified by the input parameter for this facilitator method. To do so you may use the following method call for the GregorianCalendar class object (`logCalendar`).

```
logCalendar.set(logYear, logMonth −1, day);
```

Then the day of the week corresponding to the current logbook's `logCalendar` *month/day/year* can be found using the following method.

```
logCalendar.get(Calendar.DAY_OF_WEEK);
```

This method returns a value between 1 (Sunday) and 7 (Saturday). As noted in the description of the `dayOfWeek()` method given above, we would prefer that the value returned be between 0 (Sunday) and 6 (Saturday). So, you will need to adjust the returned value similar to the way the month parameter was adjusted for the creation of `logCalendar` in the Logbook constructor.

Step 1: Implement the facilitator method `dayOfWeek()` described above and add it to the file *Logbook.java*. This method is included in the incomplete definition of the Logbook class presented earlier.

Step 2: Implement the `displayCalendar()` method described above and add it to the file *Logbook.java*. This method is included in the incomplete definition of the Logbook class earlier.

Step 3: Activate Test 3 in the test program *TestLogbook.java* by removing the comment delimiter (and the character "`3`") from the lines that begin with "`//3`".

Step 4: Complete the test plan for Test 3 by filling in the day of the week for the first day of the current month.

Step 5: Execute the test plan. If you discover mistakes in your implementation of the displayCalendar operation, correct them and execute the test plan again.

Test Plan for *Test3* (displayCalendar Operation)

Test case	Logbook month	Day of the week of the first day in the month	Checked
Simple month	`1 2000`	6 (Saturday)	
Month in the past	`7 1998`	3 (Wednesday)	
Month in the future	`12 2021`	1 (Monday)	
Current month			
February (not leap year)	`2 2002`	5 (Friday)	
February (leap year)	`2 2000`	2 (Tuesday)	

LABORATORY 1: In-lab Exercise 2

Name ____

Hour/Period/Section ____

Date ____

Java allows you to create multiple methods with the same name so long as these methods have different numbers of arguments or different types of arguments—a process referred to as **method overloading**. The following Logbook ADT operations, for example, each shares the same name as an existing operation. They have fewer arguments than the existing operations, however. Instead of using an argument to specify the month/year (or day) to process, they use the current month/year (or day).

`Logbook ( )`
Precondition:
None.
Postcondition:
Default constructor. Creates an empty logbook for the current month/year.

`void putEntry ( int value )`
Precondition:
Logbook is for the current month/year.
Postcondition:
Stores the value as the logbook entry for today.

The default constructor for the built-in GregorianCalendar class creates a Calendar object for today's date. Then using the method *get* as we did in `dayOfWeek()` we can assign the correct value to `logYear` as follows.

```
logYear = logCalendar.get(Calendar.YEAR);
```

In a similar manner we can assign the correct value to `logMonth` using the parameter `Calendar.MONTH`. Also, `Calendar.DAY_OF_MONTH` contains today's day value for use in the overloaded method `putEntry()`.

Step 1: Implement these operations and add them to the file *Logbook.java*. Each method is included in the incomplete definition of the Logbook class.

Step 2: Activate Test 4 in the test program *TestLogbook.java* by removing the comment delimiter (and the character "`4`") from the lines that begin with "`//4`".

Step 3: Complete the test plan for Test 4 by filling in the expected result for each operation.

Step 4: Execute the test plan. If you discover mistakes in your implementation of these operations, correct them and execute the test plan again.

Test Plan for *Test4* (Overloaded Methods)

Test case	Expected result	Checked
Construct a logbook for the current month	Number of days in the current month:	
Record an entry for today	Day on which entry is made:	

LABORATORY 1: In-lab Exercise 3

Name ____________________

Hour/Period/Section ____________________

Date ____________________

What if we want to add the entries in several logbooks together to find a grand total of daily entries for a particular month? For instance, the code fragment on this page illustrates how we might add the daily entries in a logbook of citySales to the entries in a logbook of suburbSales to give us a logbook of daily salesTotals for the month of September, 2002. The following describes a method that will add the corresponding daily entries in two logbooks.

`void plus ( Logbook rightBook )`
Precondition:
The logbooks cover the same month/year.
Postcondition:
Adds each entry in rightBook to the corresponding entry in this (invoking object) logbook.

The following code fragment uses this operation to sum a pair of logbooks and then outputs the combined logbook entries.

```
Logbook citySales = new Logbook(9, 2002),     // City sales
        suburbSales = new Logbook(9, 2002),   // Suburban sales
        salesTotals = new Logbook(9, 2002);   // Combined sales for September 2002
int j;                                        // Loop counter

// Read in the city and suburban sales.
...

// Sum the city and suburban sales.
salesTotals.plus( citySales );                // Include city sales
salesTotals.plus( suburbSales );              // Include suburban sales

// Output the sum.
salesTotals.displayCalendar( );
```

Step 1: Implement the `plus( )` operation and add it to the file *Logbook.java*.

Step 2: Activate Test 5 in the test program *TestLogbook.java* by removing the comment delimiter (and the character “`5`”) from the lines that begin with “`//5`”.

Step 3: Complete the test plan for Test 5 by filling in the expected result. Use a logbook for the current month.

Step 4: Execute the test plan. If you discover mistakes in your implementation of the logbook addition operation, correct them and execute the test plan again.

Test Plan for *Test5* (plus Operation)

Test case	Expected result of adding `logDay200` to `logDay100`	Checked
The entries in logbook `logDay100` are equal to (100 * day) and the entries in logbook `logDay200` are equal to (200 * day)		

LABORATORY 1: Postlab Exercise 1

Name ____

Hour/Period/Section ______________________________

Date ____

Part A

In our implementation of Logbook, the facilitator method `leapYear( )` uses the built-in GregorianCalendar class method `isLeapYear( )`. This is possible because the Logbook class contains a data member from the GregorianCalendar class. If there were no GregorianCalendar (`logCalendar`) data member in the Logbook class, how would you implement Logbook's `leapYear( )` method?

```
private boolean leapYear ( )                 // Leap year?
{





}
```

Part B

In terms of time and space, what is the cost of defining the data member `logCalendar` to implement `leapYear( )`?

In terms of time and space, what is the cost (or savings) of implementing `leapYear( )` without declaring the GregorianCalendar class data member `logCalendar`?

LABORATORY 1: Postlab Exercise 2

Name	
Hour/Period/Section	
Date	

Part A

In our implementation of Logbook the facilitator method `dayOfWeek( )` uses several built-in GregorianCalendar class methods to return the correct value. This is possible because the Logbook class has a data member from the GregorianCalendar class. If there were no GregorianCalendar (`logCalendar`) data member, how would you implement `dayOfWeek( )`?

```
private int dayOfWeek ( int day )
// Returns the day of the week corresponding to the specified day.
{









}
```

Part B

What is gained/lost by implementing `dayOfWeek( )` without using the GregorianCalendar data member?