Chapter 23

A Simple Database

We can use an array to implement a simple database of employee records. 23.1 provides a template from which employee record objects can be created.

```
import java.text.DecimalFormat;
import java.util.Scanner;

public class EmployeeRecord {
    private static int ID = 0;  //  Used to produce consecutive ID numbers
    public static final Scanner scanner = new Scanner(System.in);
    private int idNumber;
    private String lastName;
    private String firstName;
    private int salary;

    public EmployeeRecord(String last, String first, int pay) {
        idNumber = ID++;
        lastName = last;
        firstName = first;
        salary = pay;
    }
    //  Accessor methods
    public int getID() { return idNumber; }
    public String getLast() { return lastName; }
    public String getFirst() { return firstName; }
    public int getSalary() { return salary; }

    public void show() {
        DecimalFormat idFmt = new DecimalFormat("00000"),
                      salaryFmt = new DecimalFormat("$#####.00");
        System.out.println("ID:     " + idFmt.format(idNumber));
        System.out.println("Name:   " + lastName + ", " + firstName);
        System.out.println("Salary: " + salaryFmt.format(salary));
    }
```



```
    public static EmployeeRecord read() {
        String last, first;
        System.out.print("Enter last name:  ");
        last = scanner.next();
        System.out.print("Enter first name: ");
        first = scanner.next();
        System.out.print("Enter salary:     ");
        int pay = scanner.nextInt();
        scanner.nextLine();  //  Consume rest of line
        return new EmployeeRecord(last, first, pay);
    }
    public void edit() {
        String newLast, newFirst, newSalary;
        scanner.nextLine();  //  Clear line for new entry
        System.out.print("Last name [" + lastName + "]:");
        newLast = scanner.nextLine();
        System.out.print("First name [" + firstName + "]:");
        newFirst = scanner.nextLine();
        System.out.print("Salary [" + salary + "]:");
        newSalary = scanner.nextLine();
        lastName = (newLast.equals(""))? lastName : newLast;
        firstName = (newFirst.equals(""))? firstName : newFirst;
        salary = (newSalary.equals(""))? salary : Integer.parseInt(newSalary);
    }
}
```

Listing 23.1: `EmployeeRecord`—information about a single employee

`EmployeeRecord` defines a number of fields and methods:

- **Class variables**. The class variable `ID` is used to automatically generate consecutive unique identification numbers for each employee record object created. The first ID number assigned will be zero. The constructor is responsible for using and updating this master ID number. The process is identical to that used in the `Widget` class (9.2) in Section 9.4.

  The `scanner` constant, which gets user input, is shared by all `EmployeeRecord` objects. There is no need for each `EmployeeRecord` object to have its own object to receive keyboard input. It is `public` and `final`, so methods in other classes can use it but not reassign it.

- **Instance variables**. Each employee record object has a unique ID number, a last name, a first name, and a salary. Names and salaries need not be unique.

- **Constructor**. The constructor performs routine object initialization. A unique ID number is assigned outside of the control of the client. The other fields of the new object are assigned from client supplied values.

- **Accessor methods**. The `get...()` methods allow clients to read any instance variable of the object.

- **I/O methods**. The `show()` method uses a `java.text.DecimalFormat` object to format the output of the ID number and salary. The ID number always displays five digits (for example, ID = 5 displays as 00005), and the

salary field appears as currency (for example, salary = 45000 displays as $45000.00) even though it is stored as an integer. The `show()` method must be an instance method since it accesses instance variables.

The `read()` method allows a user to type in name and salary information from the keyboard. It is a class method because an instance of `EmployeeRecord` is not required for it to work; it creates and returns a new `EmployeeRecord` object.

The `edit()` method allows a user to modify the name fields or salary field of a record. The user is prompted for each field, and the current value is displayed as a default. If the user simply presses the Enter key, the empty string (`""`) is entered. The empty string will not replace an existing field. Notice that the user cannot change the ID number of an employee. This is not allowed because a user may accidentally reassign a used ID number.

A real-world employee record would contain many more fields including date of birth, hire date, social security number, etc.

The database class (23.2, `Database`) uses an array to store a collection of employee records.

```
public class Database {
    //  Array of employee records
    private EmployeeRecord[] employees;
    //  Current number of employees
    private int size;
    public Database(int max) {
        //  Create an array to hold all the employees
        employees = new EmployeeRecord[max];
        size = 0;  //  Currently empty
    }
    public EmployeeRecord retrieve(int id) {
        for ( int i = 0; i < size; i++ ) {
            if ( employees[i].getID() == id ) {
                return employees[i];  //  Return record
            }
        }
        return null;  //  Not found
    }
    public boolean insert(EmployeeRecord rec) {
        if ( size < employees.length ) {
            //  Still room to add
            if ( retrieve(rec.getID()) == null ) {  //  Disallow duplicate IDs
                employees[size++] = rec;
                return true;  //  Success
            } else {
                System.out.println("Duplicate ID.  Record not inserted");
            }
        } else {
            System.out.println("Database full.  Record not inserted");
        }
        return false;  //  Database full or duplicate record; could not insert
    }
    public void show() {  //  Print all database entries
        for ( int i = 0; i < size; i++ ) {
```



```
            employees[i].show();
            System.out.println("---------------------------");
        }
    }
    public void sortIDs() {  //  Sort the database by ascending IDs
        for ( int i = 0;  i < size - 1;  i++ ) {
            int smallestPos = i;
            for ( int j = i + 1;  j < size;  j++ ) {
                if ( employees[j].getID() < employees[smallestPos].getID() ) {
                    smallestPos = j;
                }
            }
            if ( smallestPos != i ) {   //  Swap
                EmployeeRecord temp = employees[i];
                employees[i] = employees[smallestPos];
                employees[smallestPos] = temp;
            }
        }
    }
    public void sortNames() {  //  Sort the database lexicographically
        for ( int i = 0;  i < size - 1;  i++ ) {
            int smallestPos = i;
            for ( int j = i + 1;  j < size;  j++ ) {
                String jName = employees[j].getLast() +
                               employees[j].getFirst(),
                       smallestName = employees[smallestPos].getLast() +
                                      employees[smallestPos].getFirst();
                if ( jName.compareToIgnoreCase(smallestName) < 0 ) {
                    smallestPos = j;
                }
            }
            if ( smallestPos != i ) {   //  Swap
                EmployeeRecord temp = employees[i];
                employees[i] = employees[smallestPos];
                employees[smallestPos] = temp;
            }
        }
    }
}
```

Listing 23.2: `Database`—uses an array to store a collection of employee records

The `Database` class provides two fields and a number of methods:

- **Instance variables**. An array (`employees`) stores individual employee records. The array's size is allocated when the database is created, but it is initially *logically* empty—it contains no records until they are inserted. The *physical* size of the array is determined by how many elements it can contain; this is fixed when the `new` operator is used to create the array. The *logical* size of the array is based on how many of the allocated spaces

are being used. The `size` variable keeps track of the array's logical size. Each time an element is added, `size` increases by one.

- **Construction**. The constructor creates the array and sets its logical size to zero. The physical size of the array is provided as a parameter from the client.

- **Insertion**. The `insert()` method allows clients to populate the database. The new employee record is placed "on the end" of the array. This means the new record goes in position `size`. The `size` variable is then incremented so that when the `insert()` method is called again the new record will go in the next available position.

  Insertion can fail for two reasons:

  – The logical size of the array equals its physical size. This means the array is full, and no more elements can be inserted.

  – The client attempts to insert a record with an ID number that matches an ID number already present in the database. Duplicate IDs are not allowed, so the insertion is not performed. Because of the way the `EmployeeRecord` class is designed, this situation should never arise; however, a redundant check like this one is not necessarily bad. The author of the `EmployeeRecord` class can be different from the author of the `Database` class. The `Database` programmer is exhibiting *defensive programming*; that is, attempting to avoid an error in the `insert()` method that is actually due to erroneous code elsewhere. This defensive programming is not free, however. The `retrieve()` method is relatively expensive since it will step through the entire array if the element to insert has a unique ID. After thorough testing this conditional check really should be removed.

- **Retrieval**. Employee records are retrieved by ID number. The `retrieve()` method returns a reference to the record (or `null` if not present).

- **Sorting**. The database can be sorted in one of two ways:

  – **ID number order**. This is the "natural" ordering since new records are given sequential ID numbers.

  – **Name order**. This is called lexicographical order (meaning "dictionary order"), commonly called alphabetical order. Since last names are not unique, the first name is concatenated to the last so both are considered when determining the proper order. The `String` method `compareToIgnoreCase()` is oblivious to capitalization and works as follows:

$$\texttt{a.compareToIgnoreCase(b)} \begin{cases} < 0 & \text{if } \texttt{a} \text{ is lexicographically less than } \texttt{b} \\ = 0 & \text{if } \texttt{a} \text{ is equal to } \texttt{b} \\ > 0 & \text{if } \texttt{a} \text{ is lexicographically greater than } \texttt{b} \end{cases}$$

- **Display**. The `show()` method steps through the array and has each employee record object show itself.

This database is extremely rudimentary. A real database would provide for more sophisticated client interaction. Most modern databases provide a query language that allow clients to perform complex queries.

`EmployeeDatabase` (23.3) represents client code that uses the `Database` and `EmployeeRecord` classes. It allows the user to enter single letter commands and prompts the user when more information is needed.

```
public class EmployeeDatabase {
    public static void main(String[] args) {
        Database db = new Database(10);  //  Create a new database
        boolean done = false;
        do {
```



```
            System.out.print("==>");
            String command = EmployeeRecord.scanner.nextLine();
            switch ( command.charAt(0) ) {
                case 'i':
                case 'I':
                    if ( !db.insert(EmployeeRecord.read()) ) {
                        System.out.println("Record not inserted");
                    }
                    break;
                case 'e':
                case 'E':
                    System.out.print("Enter ID number: ");
                    int id = EmployeeRecord.scanner.nextInt();
                    EmployeeRecord rec = db.retrieve(id);
                    if ( rec != null ) {
                        rec.edit();
                    }
                    break;
                case 'p':
                case 'P': db.show();    break;
                case 'n':
                case 'N': db.sortNames();    break;
                case 's':
                case 'S': db.sortIDs();    break;
                case 'q':
                case 'Q': done = true; break;
            }
        } while ( !done );
    }
}
```

Listing 23.3: `EmployeeDatabase`—allows a user to interactively exercise the employee database

23.1 Summary

- Add summary items here.

23.2 Exercises

