

7

Classes: large example

1 Example: student register

We illustrate program design on a larger scale, including the use of nested objects and arrays of objects. The example particularly illustrates the identification and design of classes.

We want to make a program that makes a set of course registers for students attending a certain university. Input comes from the keyboard in the form of one line per student. Each line contains (in order) student number, first name, second name, and three subjects which the student is taking. Subjects are drawn from biology, chemistry, computing, maths, physics, and stats. A typical input looks like

```
986987 Bill Smith maths stats computing
567245 Jane Kelly stats maths biology
965428 Fred Blogs biology stats maths
609128 Mary Wallace biology maths computing
754926 Carol Coutts maths stats computing
687753 Michael Moulds maths stats biology
297635 Pat Murphy maths stats computing
987636 Fred Flintstone maths stats biology
639373 Sonia Dee maths stats biology
```

Output is, for each subject with at least one student, a sorted list of the students taking that subject. Each list is sorted according to the usual phone-book ordering. A typical output looks like

```
                SUBJECT: biology

Blogs, Fred           965428
Dee, Sonia            639373
Flintstone, Fred     987636
Kelly, Jane           567245
Moulds, Michael      687753
Wallace, Mary        609128

                SUBJECT: computing

Coutts, Carol        754926
Murphy, Pat          297635
```

Smith, Bill	986987
Wallace, Mary	609128

SUBJECT: maths

Blogs, Fred	965428
Coutts, Carol	754926
Dee, Sonia	639373
Flinstone, Fred	987636
Kelly, Jane	567245
Moulds, Michael	687753
Murphy, Pat	297635
Smith, Bill	986987
Wallace, Mary	609128

SUBJECT: stats

Blogs, Fred	965428
Coutts, Carol	754926
Dee, Sonia	639373
Flinstone, Fred	987636
Kelly, Jane	567245
Moulds, Michael	687753
Murphy, Pat	297635
Smith, Bill	986987

To start, we identify the important concepts in the problem domain, such as

Student number	Person's name	Subject	Sets of subjects
Student	Student register	Class roll	

We can speculate how we might represent them, as follows:

Student number

A string is sufficient here, as there are sufficient operations on strings for the purpose -- reading and writing is about all that is needed. Not worth a class.

Person's name

A pair of strings suffice, one for the forename and one for the surname. (If we were not interested in sorting names, probably a single string would have sufficed.) Operations include reading, writing, comparison etc. Worth a class.

Subject

A string is sufficient to represent a subject. The only operation is probably that of reading. Not worth a class.

Set of subjects

Each student studies a set of subjects, which we can implement as an array of subjects. Operations include reading a set of three subjects, and testing whether a particular subject is included in the set. Introduce a class.

Student

A student is represented by a student number (string), a name, and a subject set, and so these will become instance variables. Operations include reading, writing, and comparison (of student names).

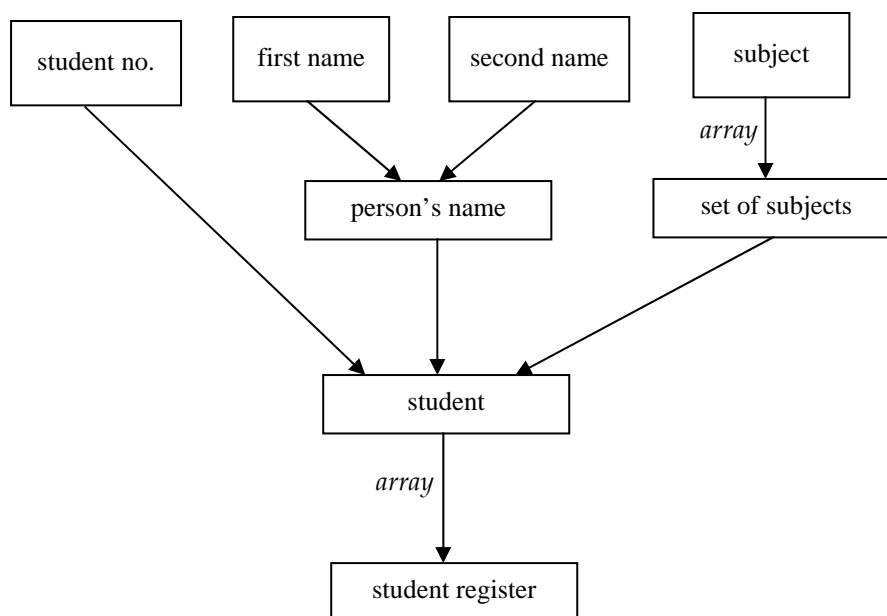
Student register

A student register is a database of all the students. We can implement it as an array of students. Operations include reading, sorting by name, et al. Introduce a class. Important: it commonly happens that we introduce a class to represent an individual item (here, a student), and a separate class to represent a collection of them (here, the student register); trying to put these notions in the same class doesn't work.

Class roll

A class roll is a list of the students taking a particular subject. There are no operations on a class roll other than to generate it from a student register, and so it will probably be convenient to do so by providing a method in the student register class.

The important components we have identified as being relevant to modeling the problem domain have a hierarchical relationship that can be depicted as follows :



We present each class in turn, highlighting some pertinent points in each. As always in these large examples, the details are not as important as the architecture.

```
class PersonName { // The name of a person

    private String first, second; // first and second names

    void get() { // Read name from keyboard
        first = Console.readToken(); second = Console.readToken();
    }

    void put() { // Write name in form "Smith, Fred" to length 25
        String name = second + ", " + first;
        System.out.printf("%-25s", name);
    }

    boolean lt(PersonName s) { // My name precedes s.name?
        return(second.compareTo(s.second)<0 ||
            (second.equals(s.second) && first.compareTo(s.first)<0));
    }

    boolean equals(PersonName s) { // My name equals s.name?
        return(first.equals(s.first) && second.equals(s.second));
    }
}
```

Class PersonName contains nothing new that needs explanation.

```
class SubjectSet { // Set of subjects

    private final int numSubjects = 3;
    private String[] subjects = new String[numSubjects]; // The set

    void get() { // Read set from keyboard.
        for (int i=0; i<numSubjects; i++) {
            subjects[i] = Console.readToken();
        }
    }
}
```

```

    }
}

boolean contains(String s) { // Is s a member of the set?
    int i = 0;
    while (i < numSubjects && !subjects[i].equals(s)) i++;
    return i < numSubjects;
}

public final static String[] subjectTitles =
    {"biology", "chemistry", "computing", "maths", "physics", "stats"};
public final static int numTitles = subjectTitles.length;
}

```

Observe the use of constant `numSubjects`: if the specification changes so that students take, say, four subjects, then only this constant needs to be changed and nothing else.

We have declared variables `subjectTitles` and `numTitles` to be static because we only need one version of each, no matter how many instances of the `SubjectSet` are created. Actually, the set of all possible subjects might deserve embodiment in its own class, but we have chosen to introduce it as a globally accessible array parked for convenience within class `SubjectSet`.

`Student numbers` is another example of a concept that arguably merits its own class, but one which pragmatically we can live without. If the problem specification made more of student numbers (for example by insisting they have some special format which we should check for) then we might consider it worthwhile to make a class for them. It is best not to add structure needlessly: structure for the sake of it can lead to obfuscation and computational inefficiency.

```

class Student { // A student

    private String studentNum; // Student number
    private PersonName name = new PersonName(); // name
    private SubjectSet courses = new SubjectSet(); // Subjects being studied

    void get() { // Read from keyboard.
        studentNum = Console.readToken(); name.get(); courses.get();
    }
}

```

```

void put() { // Display student name and number
    name.put();
    System.out.println(studentNum);
}

boolean lte(Student s) { // Do I precede s in phonebook order?
    return (name.lt(s.name) ||
        (name.equals(s.name) && studentNum.compareTo(s.studentNum) <=0));
}

boolean isTaking(String s) { // Am I taking subject s?
    return(courses.contains(s));
}
}

```

Method `isTaking()` is a small example of the added computational cost sometimes incurred by the hierarchical software architecture that arises in object-oriented programming. The method does no useful work other than passing the call to another component in the hierarchy.

```

class StudentRegister { // Register of students

    private final static int maxNumStudents= 1000;
    private Student[] register = new Student[maxNumStudents];
    private int numStudents = 0; // number of significant items in register

    void get() { // Read register of students, and sort
        while (!Console.endOfFile()) {
            register[numStudents] = new Student();
            register[numStudents].get();
            numStudents++;
        }
        sort();
    }

    void sort() { // sort register
        // Sort register[0..numStudents-1]
        int lft = 0;
        while (lft<numStudents) {

```

```

        int min = lft; int i = lft+1;
        while (i<numStudents) {
            if (register[i].lte(register[min])) min = i;
            i++;
        }
        Student temp = register[lft];
        register[lft] = register[min]; register[min] = temp;
        lft++;
    }
}

void putHeader(String s) { // Put header for subject s
    System.out.println(); System.out.println();
    System.out.println("        SUBJECT: " + s);
    System.out.println();
}

void put(String s) { // Display roll for subject s
    int total = 0; // Number of students taking this subject
    for (int i=0; i<numStudents; i++) {
        if (register[i].isTaking(s)) {
            if (total==0) putHeader(s);
            register[i].put();
            total++;
        }
    }
}
}
}

```

Rather unusually, it is attractive to sort the list of students immediately after it is input.

```

class ClassRolls {
    public static void main (String[] args) {
        StudentRegister register = new StudentRegister();
        register.get();
        for (int i=0; i<SubjectSet.numTitles; i++)
            register.put(SubjectSet.subjectTitles[i]);
    }
}

```

}

Changes

A good test of the architecture of a program is to ask a series of questions along the lines “If the customer asked for this change, or this addition, or this generalisation, how many classes would I have to inspect, how many would I have to change, and how small would the changes be?”. It is probably a bad sign if an intuitively small change in the specification leads to changes in many classes, or even to re-structuring the design. For the program above, consider the implications of small changes in the specification:

- (a) Class rolls to be in order of student number.
- (b) Names to be displayed in form “Bill Smith”.
- (c) The nationality of each student included in the input, and also in the output.
- (d) Allow the number of subjects each student takes to vary.
- (e) Do not assume that the possible subjects are known in advance; instead they are to be deduced from the student data.

We leave it as an exercise to show that these changes are kept fairly local to the relevant classes. One word of warning however: don’t try to anticipate too many changes in the specification – otherwise you will end up solving a much more general problem than was given by the customer.