**Problem Statement**

* Problem statement:

*Write an application that reads in a text file organized in the manner shown below and displays the final course grades. The course grades are computed differently for the undergraduate and graduate students based on the formulas listed on page 710. The input text file format is as follows:*

*• A single line is used for information on one student.*

*• Each line uses the format*

*<Type> <Name> <Test 1> <Test 2> <Test 3>*

*where <Type> designates either a graduate or an undergraduate*

*student,<Name> designates the student’s first and last name, and*

*<Test i> designates the ith test score.*

*• End of input is designated by the word END. The case of the letters is*

*insignificant.*

Overall Plan

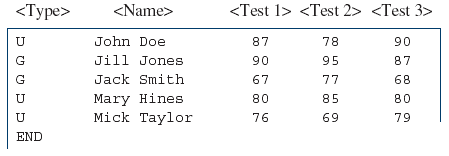
* Tasks

**1.** Read an input text file.

**2.** Compute the course grades.

**3.** Print out the result.

Input File Format



Development Steps

* We will develop this program in five steps:

1. Start with the program skeleton.Define the skeleton ComputeGrades classes.

2. Implement the printResult method.Define any other methods necessary to implement printResult.

3. Implement the computeGrade method.Define any other methods necessary to implement computeGrade.

4. Implement the readData method.Define any other methods necessary to implement readData.

5. Finalize and look for improvements.

Step 1 Design

* We start with a program skeleton.
* We will define two constructors so the programmer can create a roster of default size or the size of her choice.

Step 1 Code

Source Files: ComputeGrades.java

Step 1 Test

* We include a temporary output statement inside the (currently stub) method we define.
* We run the test main class and verify that the methods are called correctly.

Step 2 Design

* We design and implement the printResult method
* We use the helper class OutputBox for displaying the result.

**for** *each element i in the roster array* **{**

*output the name of* roster**[**i**]**;

*output the test scores of* roster**[**i**]**;

*output the course grade of* roster**[**i**]**;

*skip to the next line*;

**}**

Step 2 Code

Source Files: ComputeGrades.java

Step 2 Test

* We verify the temporary readData method is working correctly. This confirms that we are using the correct student classes and using their methods correctly.
* We verify the printResult method does indeed display the data in our desired format.

Step 3 Design

* We design and implement the computeGrade method.
* The code for actually determining the course grade is embedded in individual student classes
  + So the code to add to the ComputeGrades class is very simplistic.
  + This is a direct benefit of using polymorphism effectively.

Step 3 Code

Source Files: ComputeGrades.java

Step 3 Test

* We will repeat the same test routines from Step 2.
* Instead of seeing four asterisks, we should be seeing the correct grades.
* We test both the passing and not passing test scores.

Step 4 Design

* We design and implement the core functionality of the program—the readData method
* We can express its logic as

*get the filename from the user*;

**if (***the filename is provided***)**

*read in data and build the roster array;*

**else**

*output an error message*;

The buildRoster Method

* The logic of the workhorse private method buildRoster is as follows:

*set bufReader for input*;

**while (** !done **) {**

line = *get next line*;

**if (***line is END***) {**

done = **true**;

**} else {**

student = createStudent**(** line **)**;

**if (**student != **null) {**

roster**[**studentCount**]** = student; //add to roster

studentCount++;

**}**

**}**

The createStudent Method

* We use the StringTokenizer class to break down items in a single line of input

StringTokenizer parser = **new** StringTokenizer**(** line **)**;

String type;

**try {**

type = parser.nextToken**()**;

**if (**type.equals**(**UNDER\_GRAD**)** || type.equals**(**GRAD**)) {**

student = newStudentWithData**(**type, parser**)**;

**} else {** //invalid type is encountered

student = **null**;

**}**

**} catch (**NoSuchElementException e**) {** //no token

student = **null**;

**}**

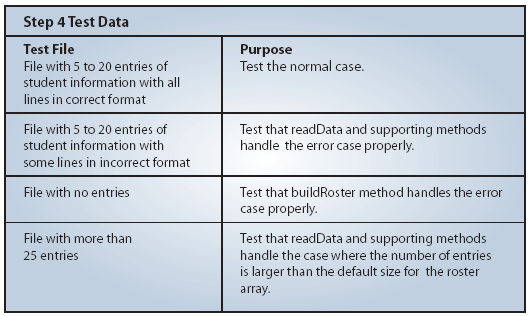
**return** student;

Step 4 Code

Source Files: ComputeGrades.java

Step 4 Test

* We run through a more complete testing routine in this step.We need to run the program for various types of input files. Some of the possible file contents are as follows:



Step 5: Finalize and Improve

* We finalize the program by correcting any remaining errors, inconsistency, or unfinished methods.
* We want to review the methods and improve them as necessarily.
* One problem (which would have been identified in step 4 testing) we need to correct is the missing method for expanding the roster array when the input file includes more student entries than the set default size of 25.
  + We leave this method as Exercise 3.
  + We also leave some of the possible improvements as exercises.

**Problem Statement**

* Problem statement:

*Write a class that manages file I/O of an AddressBook object.*

Development Steps

* We will develop this program in four steps:
  1. Implement the constructor and the setFile method.
  2. Implement the write method.
  3. Implement the read method.
  4. Finalize the class.

Step 1 Design

* We identify the data members and define a constructor to initialize them.
* Instead of storing individual Person objects, we will deal with a AddressBook object directly using Object I/O techniques.

Step 1 Code

Source Files: AddressBookStorage.java

TestAddressBookStorage.java

Step 1 Test

* We include a temporary output statement inside the setFile method.
* We run the test main class and verify that the setFile method is called correctly.

Step 2 Design

* Design and implement the write method
* The data member filename stores the name of the object file to store the address book.
* We create an ObjectOutputStream object from the data member filename in the write method.
* The write method will propagate an IOException when one is thrown.

Source Files: AddressBookStorage.java

TestAddressBookWrite.java

Step 2 Test

* We run the test program several times with different sizes for the address book.
* We verify that the resulting files indeed have different sizes.
* At this point, we cannot check whether the data are saved correctly or not.
  + We can do so only after finishing the code to read the data back.

Step 3 Design

* Design and implement the read method.
* The method returns an AddressBook object read from a file (if there's no exception)
* The method will propagate an IOException when one is thrown.

Step 3 Code

Source Files: AddressBookStorage.java

TestAddressBookRead.java

Step 3 Test

* We will write a test program to verify that the data can be read back correctly from a file.
* To test the read operation, the file to read the data from must already exist.
* We will make this test program save the data first by using the TestAddressBookWrite class from .

Step 4: Finalize

* We perform the critical review of the final program.
* We run the final test