1 Submission Instructions

Create a folder named *asuriteid* where *asuriteid* is your ASURITE user id (for example, if your ASURITE user id is *jsmith6* then your would be named *jsmith6*) and copy all of your *.java* source code files to this folder. Do not copy the *.class* files or any data files. Next, compress the *asuriteid* folder creating a **zip archive** file named *asuriteid.zip* (e.g., *jsmith6.zip*). Upload *asuriteid.zip* to the Project 2 submission link by the project deadline. The deadline can be found in the *Course Schedule* section of BB or in the Syllabus. Consult the online syllabus for the late and academic integrity policies.

2 Learning Objectives

1. Read UML class diagrams and convert the diagram into Java classes [Ch. 12; Week 2: Lectures 1-3].
2. Identify and implement dependency, aggregation, inheritance, and composition relationships. [Ch. 12; Week 2: Lectures 1-3].
3. Properly use the public, private, and protected accessibility modifiers [Ch. 9; Week 2: Lectures 5-8].
4. Write Java code to override methods [Ch. 9; Week 2: Lectures 9-12].
5. Recognize when inheritance is present among classes in an OOD. [Chs. 9, 12; Week 2: Lectures 14-20].
6. Design and implement classes using inheritance. [Chs. 9, 12: Week 2: Lectures 1-26].
7. To write Java code to implement polymorphism in a class inheritance hierarchy. [Ch. 9; Week 3 Lectures 1-10].
8. To implement a Java interface [Ch. 10; Week 3: Lectures 5-10].

3 Background

At South Park University (located in beautiful South Park, CO) there are two categories of students: on-campus students and online students. On-campus students are categorized as residents (R) or nonresidents (N) depending on whether they reside within CO or they reside in a different state. The base tuition for on-campus students is $7575 for residents and $14,875 for non-residents. Some on-campus students, enrolled in certain pre-professional programs, e.g, law, dentistry, pharmacy, are charged an additional program fee which varies depending on the program. An on-campus students may enroll for up to 18 credit hours at the base rate but for each credit hour exceeding 18, they pay an additional fee of $475 for each credit hour over 18.

Online students are neither residents nor non-residents. Rather, their tuition is computed as the number of credit hours for which they are enrolled multiplied by the online credit hour rate which is $950 per credit hours. Furthermore, some online students enrolled in certain degree programs pay an online technology fee of $75 per semester.

4 Software Requirements

The software requirements for this project are:

1. Student information for South Park University is stored in a text file named *p02-students.txt*. There is one student record per line, where the format of a student record for an on-campus student is:

   `C` ***`id last-name first-name residency program-fee credits`***

   where:

`C`	Identifies the student as an on-campus student.
`id`	The student identifier number. A string of 13 digits.
`last-name`	The student's last name. A contiguous string of characters.
`first-name`	The student's first name. A contiguous string of characters.
`residency`	R if the student is a resident, N if the student is a non-resident.
`program-fee`	A program fee, which may be zero.
`credits`	The number of credit hours for which the student is enrolled.

   The format of a student record for an online student is:

   `O` ***`id last-name first-name tech-fee credits`*** where `O` identifies the student as an online student, and ***id***, ***last-name***, ***first-name***, and ***credits*** are the same as for an on-campus student. The ***tech-fee*** field is `T` if the student is to be assessed the technology fee or `-` if the student is not assessed the technology fee. Here is an example *p02-students.txt* file:

```
Sample p02-students.txt
C   8230123345450 Simons      Jenny    R  0     12
C   3873472785863 Cartman     Eric     N  750   18
C   4834324308675 McCormick   Kenny    R  0     20
O   1384349045225 Broflovski  Kyle     -  6
O   5627238253456 Marsh       Stan     T  3
```

2. The program shall read the contents of *p02-students.txt* and calculate the tuition for each student.

3. The program shall write the tuition results to an output file named *p02-tuition.txt* formatted thusly:

```
id last-name first-name tuition
id last-name first-name tuition
...
```

where *id* is the student identifier number, *last-name* and *first-name* are the student's name, and *tuition* is the computed tuition for the student. *id* shall be output left-justified in a field of width 16, *last-name* shall be output left-justified in a field of width 20, *first-name* shall be output left-justified in a field of width 15, and *tuition* shall be output right-justified in a field of width 8 with two digits after the decimal point. For the sample input file, this is the output file:

```
Sample p02-tuition.txt
1384349045225   Broflovski          Kyle            5700.00
3873472785863   Cartman             Eric           15625.00
4834324308675   McCormick           Kenny           8525.00
5627238253456   Marsh               Stan            2925.00
8230123345450   Simons              Jenny           7575.00
```

4. The records in the output file shall be sorted in ascending order by *id*.

5. If the input file *p02-students.txt* cannot be opened for reading (probably because it does not exist) then output an error message to the output window, close any open files, and then terminate the program, e.g.,

```
$ java Main
Sorry, could not open 'p02-students.txt' for reading. Stopping.
```

6. If the output file *p02-tuition.txt* cannot be opened for writing, then output an error message to the output window, close any open files, and then terminate the program, e.g.,

```
$ java Main
Sorry, could not open 'p02-tuition.txt' for writing. Stopping.
```

5 Software Design

Refer to and study the UML class diagram in §5.7. Your program shall implement this design.

5.1 Main Class

The main class is named *Main* and a template for *Main.java* is included in the project zip archive. The *Main* class shall contain the *main*() method which shall simply instantiate an object of the *Main* class and call *run*() on that object. Complete the code in *Main.java* by reading the UML class diagram, the comments, and implementing the pseudocode.

5.2 TuitionConstants Class

A class named *TuitionConstants* class is included in the project zip archive. This class declares several public static constants that are used in other classes. The constants are derived from the discussion in §3 Background.

5.3 Sorter Class

We shall discuss sorting algorithms later in the course, so this code may not make perfect sense at this time. Since I do not know which sorting algorithms you may have been exposed to in CSE100, CSE110, or non-ASU introduction to programming classes, I have provided all of the sorting code for you. If you are interested, it uses the insertion sort algorithm which is not very efficient for large lists but is efficient enough for small lists such as ours.

The *Sorter* class contains a public class method *insertionSort*() that can be called to sort a list of *ArrayList*<*Student*>. When sorting *Student*s we need to be able to compare one *Student A* to another *Student B* to determine if *A* is less than or greater than *B*. Since we are sorting by student id, we have the abstract *Student* class implement the *java.lang.Comparable*<*Student*> interface and we define *Student A* to be less than *Student B* if the *mId* field of *A* is less than the *mId* field of *B*. This is how we sort the *ArrayList*<*Student*> list by student identifier.

java.lang.Comparable<T> is a generic interface in the Java Class Library (it requires a type parameter *T* to be specified when the interface is implemented) that declares one method:

```
int compareTo(T obj)
```

where *T* represents a class type and *obj* is an object of the class *T*. The method returns a negative integer if *this T* (the object on which the method is invoked) is less than *obj*, zero if *this T* and *obj* are equal, or a positive integer if *this T* is greater than *obj*. To make abstract class *Student* implement the *Comparable* interface, we write:

```
public abstract class Student implements Comparable<Student> { ... }
```

Since *Student* implements *Comparable*<*Student*>, whenever *compareTo*() is called in *Sorter.keepMoving*() to compare two *Student* objects, either *OnCampusStudent.compareTo*() or *OnlineStudent.compareTo*() will be **polymorphically** called.

Also, study the comments for the *keepMoving*() method where I have used and discussed how to use the **ternary operator ?:** (which is inherited from the C language). There is a nice explanation of ?: on this web page.

5.4 Student Class

There is a template for the *Student* class is in the project zip archive. The *Student* class is an abstract class that implements the *java.lang.Comparable*<*T*> interface (see §5.3):

```
public abstract class Student implements Comparable<Student> { ... }
```

A *Student* object contains five instance variables (I preface my instance data members with a lowercase **m** for **m**ember):

mCredits	Number of credit hours the student is enrolled for.
mFname	The student's first name.
mId	The student's id number.
mLname	The student's last name.
mTuititon	The student's computed tuition.

Note that these data members are common to both *OnCampusStudents* and *OnlineStudents*. Most of the *Student* instance methods should be straightforward to implement (the majority of them are simple accessor/mutator methods) so we will only mention the two that are not so obvious:

`+`***`calcTuition`***`(): void`
An abstract method (that is why I have written it in italics here and in the UML class diagram) that is implemented by subclasses of *Student*. Abstract methods are generally not implemented in an abstract class, and this one is not. See the comments in the *calcTuition*() method header in *Student* for more information.

`+compareTo(pStudent: Student): int «override»`
Implements the *compareTo*() method of the *Comparable*<*Student*> interface. Returns -1 if the *mId* instance variable of *this Student* is less than the *mId* instance variable of *pStudent*. Returns 0 if they are equal (should not happen because id

numbers are unique). Returns 1 if the *mId* instance variable of *this Student* is greater than the *mId* instance variable of *pStudent*. The code for *compareTo*() is simple and is shown below. Read the *compareTo*() method comments in *Student* for more information. Note you will use the @Override annotation to prevent accidental overloading.

```
return getId().compareTo(pStudent.getId());
```

5.5 OnCampusStudent Class

The concrete *OnCampusStudent* class is a direct subclass of the abstract class *Student*. It declares two public int constants *RESIDENT* which is 1 and *NON_RESIDENT* which is 2. It adds new instance variables that are specific to on-campus students:

mResident	*RESIDENT* if the *OnCampusStudent* is a resident, *NON_RESIDENT* for non-resident.
mProgramFee	Certain *OnCampusStudent*'s pay an additional program fee. This value may be 0.

The *OnCampusStudent* instance methods are mostly straightforward to implement so we shall only discuss two of them.

`+OnCampusStudent(pId: String, pFname: String, pLname: String): «ctor»`
Must call the superclass constructor passing *pId*, *pFname*, and *pLname* as parameters.

`+calcTuition(): void «override»`
Must implement the rules described in §3 Background to calculate the tuition for either a resident or non-resident student. Note that we cannot directly access the *mTuition* instance variable of an *OnCampusStudent* because it is intentionally declared as private in *Student*. So how do we write to *mTuition*? By calling the protected *setTuition*() mutator method that is inherited from *Student*. Any why is *setTuition*() protected? Because it is only intended to be called from subclasses of *Student* and not from classes that are not part of the *Student* class hierarchy. The pseudocode for *calcTuition*() is:

```
Override Method calcTuititon() Returns Nothing
    Declare double variable t
    If getResidency() returns RESIDENT Then
        t = TuitionConstants.ONCAMP_RES_BASE
    Else
        t = TuitionConstants.ONCAMP_NONRES_BASE
    End if
    t = t + getProgramFee();
    If getCredits() > TuitionConstants.MAX_CREDITS Then
        t = t + (getCredits() - TuitionConstants.MAX_CREDITS) × TuitionConstants.ONCAMP_ADD_CREDITS
    End if
    Call setTuition(t)
End Method calcTuition()
```

5.6 OnlineStudent Class

The concrete *OnlineStudent* class is a direct subclass of the abstract class *Student*. It adds a new instance variable that is specific to online students:

mTechFee	Certain *OnlineStudent*'s pay an additional technology fee. This instance variable will be true if the technology fee applies and false if it does not.

The *OnlineStudent* instance methods are mostly straightforward to implement so we shall only discuss two of them.

`+OnlineStudent(pId: String, pFname: String, pLname: String): «ctor»`
Must call the superclass constructor passing *pId*, *pFname*, and *pLname* as parameters.

`+calcTuition(): void «override»`
Must implement the rules described in §3 Background.The pseudocode for *calcTuition*() is:

```
Override Method calcTuititon() Returns Nothing
    Declare double variable t = getCredits() × TuitionConstants.ONLINE_CREDIT_RATE
    If getTechFee() returns true Then
        t = t + TuitionConstants.ONLINE_TECH_FEE
    End if
    Call setTuition(t)
End Method calcTuition()
```

5.7 UML Class Diagram

The UML class diagram shown in Fig. 1 on the next page was created using UMLet. See the zip archive /uml folder for the UMLet file and the /img folder for a .EPS and .PNG image of the class diagram. We have the following relationships among the classes and *Comparable<Student>* interface. (See this web page for a good summary of the notation used in UML class diagrams.)

Main: The dashed lines with open arrowheads connecting *Main* with *Student*, *OnCampusStudent*, and *OnlineStudent* represent dependency relationships. *Main* is dependent on *Student* because *Student* objects are parameters to some of *Main*'s methods, in particular, *calcTuition*(), *readFile*(), and *writeFile*(). This makes *Main* dependent on *Student* because if the code in *Student* changes, it could affect the code in *calcTuition*(), *readFile*(), and *writeFile*. Note that the arrowhead points from *Main* to the class on which *Main* is dependent. *Main* also has a solid line with no symbols on the ends of the lines connecting to *Sorter*. This is an association relationship. *Main* is associated with *Sorter* because *Main.run*() calls *Sorter.insertionSort*(). Associations often include text describing the association and I have drawn the text **uses** indicating that *Main* **uses** the *Sorter* class.

Student: The solid line with a shaded diamond symbol connecting *Student* to *Main* represents a composition relationship. *Main* is composed of an *ArrayList* of *Student* objects, see *Main.run*(), and *Main* creates this *ArrayList* object, so when *Main* is deallocated by the garbage collector (we can say *Main* dies) the *Student* objects that were allocated in *Main* also die. We can say that the life cycle of *Main* and the *Student* objects of which *Main* is composed are the same so that is why this is a composition relationship and not an aggregation relationship (which is what I had originally drawn). *Student* also has a dotted line connecting it to the *Comparable<Student>* interface. The UML **classifier** «interface» serves to tell the reader that *Comparable<Student>* is an interface and not a class. An interface relationship is drawn with a dotted line with an open arrowhead pointing toward the interface being implemented.

OnCampusStudent and *OnlineStudent*: The solid lines connecting these two classes to *TuitionConstants* represents an association relationship and the word **uses** on each line simply tells the reader that these two classes use the *TuitionConstants* class. Both *OnCampusStudent* and *OnlineStudent* have a solid line with an unshaded triangle connecting them to *Student*. These lines represent generalization or inheritance relationships and since the triangle is on the end of the line near *Student* it indicates that *Student* is the general class and *OnCampusStudent* and *OnlineStudent* are the specific classes. An alternative way to describe this is to say that *Student* is the superclass and *OnCampusStudent* and *OnlineStudent* **inherit** from *Student* so they are subclasses of *Student*.

Sorter: The dotted line connecting *Sorter* to *Student* with an open arrowhead pointing toward *Student* indicates that *Sorter* is dependent on *Student*. *Sorter* is dependent on *Student* because *Sorter.insertionSort*(), *Sorter.keepMoving*() and *Sorter.swap*() all have method parameters which contain *Student* objects. This makes *Sorter* dependent on *Student* because if the code in *Student* changes, it could affect the code in *Sorter*.

6 Additional Project Requirements

1. Format your code neatly. Use proper indentation and spacing. Study the examples in the book and the examples the instructor presents in the lectures and posts on the course website.

2. Put a comment header block at the top of each method formatted thusly:

```
/**
 * A brief description of what the method does.
 */
```

3. Put a comment header block at the top of each source code file formatted thusly:

```
//**************************************************************************************************
// CLASS: classname (classname.java)
//
// DESCRIPTION
// A description of the contents of this file.
//
// COURSE AND PROJECT INFO
// CSE205 Object Oriented Programming and Data Structures, semester and year
// Project Number: project-number
//
// AUTHOR
// your-name (your-email-addr)
//**************************************************************************************************
```

Fig.1 Project 2 UML Class Diagram