COSC2135 - Programming 1
Study Period 1, 2014
Assignment 3

NOTE: This assignment is to be undertaken individually – no group work is permitted.

Background information

For this assignment you need to write an object-oriented console application in the Java programming language which adheres to basic object-oriented programming principles shown below:

a) Setting the visibility of all instance variables to private.

b) Setting the visibility of all methods in your classes to public

c) Encapsulating both the data for a class and the methods that work with and/or manipulate that data within the same class.

d) Using superclass methods to retrieve and/or manipulate superclass properties from within subclass methods where necessary.

e) Taking advantage of polymorphism wherever possible when invoking methods upon objects that have been created.

The overall scenario we are modelling is that of university employee management system, where you can have *standard* employees and *academic* employees.

- The standard employee type represents administrative or service provider roles in the university - the pay scale for standard employees is broken down into five levels and the pay scale level for a standard employee is decided by the requirements of their role.

  The role that is being performed by a standard employee may change over time, resulting in their current pay scale level being adjusted (ie. their pay scale level can be increased or decreased as required).

- An academic employee type represents academic teaching staff in the university. The pay scales for academic staff are also broken down into five levels, but for academics the pay scale reflects the length of time they have worked for the university, as well as their performance in the areas of teaching and research.

  An academic employee's pay scale level will move up to the next level when the university decides that they have earned a promotion and once an academic employee's pay scale has reached the highest level (Professor) then they will not be able to be promoted further.

This task is divided up into several stages, each of which corresponds to the concepts discussed during specific weeks of the course as shown below:

- ***Stage 1 - Employee class (Week 5 - writing classes)***
- ***Stage 2 - Employee Data System (Week 6 - working with arrays of objects)***
- ***Stage 3 - AcademicEmployee class (Week 7 - Inheritance / subclasses)***
- ***Stage 4 - Employee Data System updates (Week 8 - Polymorphism)***
- ***Stage 5 - Updating Pay Scale Level Functionality (Week 10 - Exception Handling)***
- ***Bonus marks functionality (Week 11 - File Handling)***

Stage 1: Employee Class Design
(already implemented)

Both standard and academic employees share common details, such as the employee number, employee name, employee role (ie. their position such as cleaner, receptionist, etc.) and current pay scale level, which will be encapsulated in this superclass.

An `Employee`'s pay scale level can be set to any level in the range 'A' to 'E' and their pay scale level may be changed up or down as the role they perform in the university changes.

A design for a basic `Employee` class has already been put together and the programmer who was previously working on the project has already implemented the corresponding class, so **you do not have to implement this class yourself**.

A) Instance Variables

Instance variables have been defined for the **employeeNumber** (a `String`), **name** (a `String`), **role** (a `String`) and **level** (a `char`).

B) Constructor

A constructor has been defined for the `Employee` class which accepts the **employee number**, **name, role** and (**pay scale) level** for the new `Employee` as parameters and stores the supplied information in the corresponding instance variables.

C) Accessors (getters)

Simple accessors (getters) have been implemented for each of the instance variables in this class.

D) Mutators (setters)

A simple mutator `public void setRole(String role)` has been implemented, which allows the `Employee` **role** to be updated to the value that was passed in as a parameter.

E) Operations

(i) A method `public boolean updateLevel(char level)` has been implemented, which updates the level for an `Employee` to the new level that was passed in as a parameter (as long as it is within the valid range 'A' - 'E') and returns a true result.

If the new level that was passed in as a parameter is outside the valid range ('A' - 'E') then the level is left unchanged and a `false` result is returned.

(ii) A method `public double getSalary()` has been implemented, which returns the salary for the current `Employee` based on their current `level` - the salaries for the different pay scale levels are listed in the table below:

Pay Scale Level	Salary
A	$40,000
B	$45,000
C	$55,000
D	$65,000
E	$75,000

F) Helper methods

A helper method `public void printDetails()` has been implemented, which prints a summary showing all of the details (instance variables) as well as the salary for the current `Employee` to the screen.

Stage 2 - EmployeeDataSystem Class Functionality (~~45~~ 40 marks)

Note: The total marks for this stage were corrected on Thursday May 15.

Now that we have the attribute and functionality required for a basic employee modelled and encapsulated in the `Employee` class described above in Stage 1, you will need to begin the implementation of the `EmployeeDataSystem` application class, which will use an array of `Employee` references called `employees` to store and manage employees that are added to the system by the user.*****

****You may also use a JCF collection instance (eg. `ArrayList` or `HashMap`) to store and manage the employees that the user is adding to the system if you so wish - it is, however, not a requirement to do so for this task and using a simple array of `Employee` references as described above is still perfectly acceptable.***

The programmer who was previously working on this project has implemented a menu feature for the `EmployeeDataSystem` class, for which options are presented to A) add a new employee to the system, B) display a summary of all employees currently in the system and C) Update the pay scale level / role for an employee.

A screen shot showing the presentation of this menu is shown below.

```
***** Employee Data System Menu *****
A. Add New Employee
B. Display Employee Summary
C. Update Employee Pay Scale Level / Role
D. Add New Academic Employee
E. Record PhD for Academic Employee
X. Exit
```

Your task is to work on the implementation of this initial `EmployeeDataSystem` class by implementing the functionality for **the first three** features shown in the menu only (**the other two features will be addressed in Stage 4 of this specification**).

A description of these functionality that needs to be implemented for each of these features is provided below.

A) Add New Employee Feature

This feature should prompt the user for the general employee details (employee number, name, role and pay scale level), after which a new `Employee` object should be created and stored in the next available location in the array of `Employee` references `employees`.

Note that the variable `employeeCount` should be updated to reflect the adding of the new `Employee` object to the `employees` array / collection and it may also help determine where the next available [empty] spot is in the array (this doesn't apply if you are using a JCF collection instance to store the `Employee` objects).

(10 marks)

B) Display Employee Summary Feature

This feature should display the details for all objects currently stored in the array / collection of `Employee` references described above to the screen (by calling the `printDetails`() method for each `Employee` object in the array/collection).

(5 marks)

C) Update Employee Pay Scale Level / Role

This feature should begin by prompting the user to enter the employee number to search for, after which it should proceed to locate the corresponding `Employee` object from amongst the set of `Employee` objects currently stored in the `employees` array / collection.

If an Employee object with the specified employee number **was not** found then a suitable error message indicating that the employee search was unsuccessful should be displayed to the user.

If an employee object with the specified employee number **was** found then the user be prompted to enter the new pay scale level for the `Employee`, after which the system should attempt to update the `Employee's` pay scale level by invoking the `updateLevel`() method for the `Employee` object in question in an appropriate manner.

The result that is returned by the `updateLevel`() method should be trapped / checked and a suitable message should be displayed to the user indicating whether the pay scale level was updated successfully or not.

If the pay scale level was updated successfully then the user should also be prompted to enter the new role for the `Employee` (or to just hit enter is they wish to leave the role unchanged). If a new role was entered then the role for the `Employee` should also be updated by invoking the `setRole()` method in an appropriate manner.

(25 marks)

Stage 3 - AcademicEmployee subclass implementation (45 marks)

An `AcademicEmployee` is managed somewhat differently to a standard `Employee` in that they will gradually be promoted up through the different levels that apply for academics and thus their pay scale level can only be changed by promoting the `AcademicEmployee` to the next level up.

An `AcademicEmployee` who has completes studies for a doctoral qualification should also be referred to as a "doctor" (eg. "**Dr** Donald E Knuth" as opposed to just "Donald E Knuth") and once an `AdacemicEmployee` has been awarded a doctoral qualification (PhD) then it cannot be removed.

You should address these new requirements by designing and implementing an `AcademicEmployee` subclass (which extends the basic `Employee` class from Stage 1), and which encapsulates the required additional properties and functionality as described as follows:

A) Extend the `Employee` class to define a new class `AcademicEmployee`, which includes a new instance variable `hasPHD` (a boolean).

NOTE: You should not redefine any of the instance variables that were defined previously in the `Employee` superclass in this `AcademicEmployee` subclass.

(5 marks)

B) Implement a constructor which accepts the **employee number** and **name** for the new `AcademicEmployee` as parameters and stores the supplied information in the corresponding instance variables.

You may assume that all new `AcademicEmployee's` that are added to the system will start at the lowest pay scale level by default, thus this constructor should pass along the values **'A'** and **"Associate Lecturer"** for the **level** and **role** respectively.

The `hasPHD` instance variable should start off being `false` to begin with (this happens by default).

NOTE: You should invoke the superclass constructor in an appropriate manner to initialise the instance variables for the employee number, name, role and level of the new `AcademicEmployee`.

(5 marks)

C) Override the accessor for the employee name that was defined originally in the `Employee` superclass in this `AcademicEmployee` subclass so that it inserts the title (prefix) of "Dr" before the name **if** the `AcademicEmployee` possesses a doctoral qualification (PhD) and returns the corresponding result. If the `AcademicEmployee` does not possess a doctoral qualification (PhD) then just their name should be returned.

Note: You will need to invoke the superclass name accessor to retrieve the name of the `AcademicEmployee` initially so that you can do what is instructed above in this overriding version of the name accessor in the `AdacemicEmployee` subclass.

(5 marks)

D) Override the method `public boolean updateLevel(char level)` in this `AcademicEmployee` subclass so that the method functions in the following manner.

If the current `payScaleLevel` for the `AcademicEmployee` is already at highest level (ie. level 'E') **OR** if the new level that has been passed in as a parameter is not one level above the `AdacemicEmployee's` current `payScaleLevel` then the method should return a `false` result.

If the new pay scale level is valid then this method should invoke the superclass version of the `updateLevel`() method to change the pay scale level, after which it should also automatically update the role for the `AcademicEmployee` to represent their new role in the university - these roles are defined in the table below:

Pay Scale Level	Role
A	Associate Lecturer
B	Lecturer
C	Senior Lecturer
D	Associate Professor
E	Professor

Once the role for the `AcademicEmployee` has been updated then this method should return a `true` result.

(10 marks)

E) Override the method `public double getEmployeeSalary()`, so that it returns the salary for the current Employee based on their current pay scale level - the salaries for the different pay scale levels are listed in the table below:

Pay Scale Level	Salary
A	$50,000
B	$60,000
C	$70,000
D	$80,000
E	$90,000

(10 marks)

F) Implement a method `public boolean recordPHD()`, which records the awarding of a doctorate qualification (PhD) for an `AcademicEmployee`. If the `AcademicEmployee` does not already have a doctorate qualification (PhD) recorded (ie. `hasPHD` is currently set to `false`) then the hasPHD instance variable should be set to `true` and a `true` result should be returned, otherwise a `false` result should be returned..

(5 marks)

G) Override the method `public void printDetails()` so that it prints a summary of all of the details for the current `AcademicEmployee` to the screen. ~~*(including the* `projectCount` *and* `projectList` *for the* `AcademicEmployee`*).*~~

NOTE: The overriding version of the `printDetails()` method in this `AcademicEmployee` subclass will need to use the `super` "reference" invoke the corresponding method from the `Employee` superclass to display the basic `Employee` details first.

(5 marks)

NOTE: reference to projectCount and projectList instance variables here was removed on Monday May 2.

Stage 4 - EmployeeDataSystem class updates (~~35~~ 40 marks)

Note: The total marks for this stage were corrected on Thursday May 15.

The next stage of this task is to update the `EmployeeDataSystem` class implementation described in stage 2 above so that it re-factors the update pay scale level / role feature and incorporates the other two features as required to allow the user to add and work with with `AcademicEmployee` objects:

```
***** Employee Data System Menu *****
A. Add New Employee
B. Display Employee Summary
C. Update Employee Pay Scale Level / Role
D. Add New Academic Employee
E. Record PhD for Academic Employee
X. Exit

Enter your selection:
```

A description of these functionality that needs to be implemented for each of these features is provided below.

A) Update Employee Pay Scale / Level Feature

Update this feature so that the user is only prompted to enter a new role for the employee in question if the specified employee is **not** an `AcademicEmployee`.

If the specified employee is an `AcademicEmployee` then a message should be displayed indicating that the role has been updated automatically, otherwise the feature should proceed to prompt the user to enter the new role for the employee as described in Stage 2.

(5 marks)

B) Add New Academic Employee Feature

This feature should prompt the user to enter all relevant details for the new academic employee (ie. **employee number** and **employee name**).

Once the user has entered the required details the program should then create a new `AcademicEmployee` object accordingly, passing along the details the user has supplied.

After this the object that has been created should be added to the next (empty) spot in the array or collection instance you are using to store the `Employee` / `AcademicEmployee` objects in.

(10 marks)

C) Record PhD feature

This feature should begin by prompting the user to enter the employee number to search for, after which it should proceed to locate the corresponding `AcademicEmployee` object from amongst the set of `Employee` objects currently stored in the `employees` array / collection.

If an object with the specified employee number **was not** found then a suitable error message indicating that the employee search was unsuccessful should be displayed to the user.

If a matching object was found but it is not an `AcademicEmployee` object then the program should display a suitable error message stating that the system cannot record a PhD for a standard employee.

If an employee object with the specified employee number **was** found AND it was an `AcademicEmployee` (ie. the program makes it past the previous checks for failed search / incorrect object type) then the system should attempt to invoke the `recordPHD`() method for the `AcademicEmployee` object in question in an appropriate manner.

The result that is returned by the `recordPHD`() method should be trapped / checked and a suitable message should be displayed to the user indicating whether the PhD was recorded successfully for the `AcademicEmployee` or not.

~~*(20 marks)*~~

(25 marks)

Note: The weighting of this requirement was changed from 20 marks to 25 marks on Thursday May 15.

Stage 5 – Exception handling (5 + 10 + 5 = 20 marks)

A) Initially you should define your own exception type `PayScaleException` which represents an issue that occurs when attempting to make an update the pay scale for an employee.

This `PayScaleException` type should allow a suitable error message to be specified when a new `PayScaleException` object is created.

(5 marks)

B) You should then proceed to update the `Employee/AcademicEmployee` class implementations of the `updateLevel()` method updating the pay scale level, so that a `PayScaleException` containing a suitable error message is generated and thrown when the specified pay scale level is invalid as described in for the `Employee` / `AcademicEmployee` classes (ie. when they would normally return a `false` result).

This will initially require the rewriting of the program logic in the ~~`updatePayScaleLevel()`~~ **`updateLevel()`** methods described in the specifications of the `Employee` and `AcademicEmployee` classes described previously, so that an `PayScaleException` is thrown at the points where these methods would normally return a `false` result (the exceptions that are thrown should be set up with a message that indicates what the issue is - ie. invalid pay scale level / academic already at maximum pay scale level)

Once you have done so there will not be any need to return a value indicating the success or failure of the attempted operation, so the return type for the ~~`updatePayScaleLevel()`~~ ***`updateLevel()`*** methods should also be changed to `void`.

(5 + 5 = 10 marks)

C) The `PayScaleException` should then be allowed to propagate back up to the `EmployeeDataSystem` class (**ie. the exception should not be caught locally within the `updateLevel()` method(s)**), where it will need to be caught and handled in an appropriate manner (by displaying the error message contained within the `PayScaleException` object that has propagated up from the relevant method call).

Note: You should try to limit the scope of your try block (in your try-catch structure) to the method call which could cause an exception to be thrown and any code that depends on that method call executing successfully (ie. without throwing an exception).

(5 marks)

Strategies for converting your program from the traditional "return a value" approach to the exception handling approach for signalling errors will be discussed during the week 10 live chat session.

Use of appropriate coding style (5 marks)

You should adhere to the following coding style guidelines when implementing your program for this assignment.

A. Different levels of program scope (methods, control structures, etc) should be indented consistently using levels of 3 or 4 spaces (do not use tabs).

B. A new level of indentation should be added for each new class/method/control structure that is "opened".

C. Indentation should return to the previous level of at the end of a class/method/control structure (before the closing brace if one is being used).

D. Block braces should be aligned and positioned consistently in relation to the method/control structure they are opening/closing.

E. Lines of code should not generally exceed 80 characters in length (including indentation) - lines which will exceed this limit are split into two or more segments where required.

F. Expressions should be well spaced out – this includes assignment statements, arithmetic expressions and parameter lists supplied to method calls.

G. Source code should be spaced out into logically related code segments.

H. Identifiers used in the program for class/method/variable/constant names should adhere to the naming conventions discussed in the course notes.

I. Identifiers themselves should be meaningful without being overly explicit (long) – you should avoid using abbreviations in identifiers as much as possible (an exception to this rule is a "generic" loop counter used in a for-loop).

J. Each file in your program should have a comment at the top listing your name, student number and a brief (1-2 line) description of the contents of the file (generally the purpose of the class you have implemented).

K. You should include brief (1-2 line) comments describing what each logically related segment of code in your program is doing.

An example of a good comment:

```
// calculate the interest earned and new balance
// value after the first month

interest  = ......;

balance   = ......;
```

An example of a bad comment:

```
// declare an int variable called 'x' and set
// it to the value '1'

int x = 1;
```

L. Comments should be placed on the line above the statement(s) or code segment they refer to.

Bonus Marks - File Handling Functionality (12 + 18 = 30 marks)

Your program should incorporate file handling functionality so that it writes the details for each object currently in the employee data system out to file when the program terminates (ie. when the user selects the "Exit" option in the menu).

The data that was previously written out to file should then be read back in automatically when the program is started up again and this employee information within should be used to create an appropriate set of `Employee` / `AcademicEmployee` objects, which should be stored in the array or collection of `Employee` references described in Stage 2).

If the employee data file is not found in the local folder then the program should display a message indicating that no employee data was loaded in from file and continue on to the program menu.

The data should be written out to and read in from the same text file, which contains the details for both `Employee` and `AcademicEmployee` objects in the `EmployeeDataSystem`. The format that you write data out in is entirely at your own discretion as long as it is done to a text file **(strategies for handling the writing out and reading in of data for different types of objects in the same array / collection will be discussed during the live chat session in week 11)**.

One aspect of this task is to record any changes that have been made during the previous run of the `EmployeeDataSystem` application, so your file handling functionality must be able to handle the writing out and reading in of all details for both types of employee in such a way that the state of the system at the point where the program was last exited is reconstructed in full when the program is started up again.

As these file reading / writing features are advanced (bonus marks) functionality your implementation of these features must be at least partially functional in order for your submission to be awarded any bonus marks (ie. your program needs to be able to write object details out for all objects to some degree to get any marks for the file writing feature and the program also needs to be able to read in and reconstruct the existing objects in the system to some degree in order to get any marks for the file reading feature).

IMPORTANT NOTE:

You are <u>not</u> permitted to use serialisation or binary files when implementing your file handling mechanism - you must use a `PrintWriter` for writing data out and a `BufferedReader` or `Scanner` when reading the data back in.

What to submit

If you have developed the program using a text editor and compiling/running manually at a command prompt then you only need to submit the **source code** (.java) files for the **`Employee`**, **`AcademicEmployee`** **`and`** **`EmployeeDataSystem`** classes, **as well as any helper classes that may be required to run compile and/or run your program**.

If you are using eclipse then you should export your entire eclipse project to a zip archive and submit the resulting zip file - **do this from within eclipse while it is running**, not by fiddling around in the eclipse workspace directly as you may corrupt your entire workspace if you do something wrong.

If you are not sure of how to do this then you should ask – the process of exporting an entire project to an archive file will also be demonstrated during one of the live chat sessions.

All students are also advised to check the contents of their zip files by opening them and viewing the files contained within before submitting to make sure they have done it correctly and that the correct (latest) version of the source code file is present to avoid any unpleasant surprises later on – we are obliged to accept each submission in the form it is sent to us in, so make sure you submit the final version of your program!

Submission information

This assignment will be marked out of a total of **150 marks (plus up to an additional 30 bonus marks)** and contributes **15% (plus up to an additional 3%)** towards your final result for this course. This assignment is not a hurdle in/of itself - rather it contributes towards the practical work component for this course (**of which you must obtain 50% or better overall in order to pass the course**).

This assignment is due to be submitted to **weblearn** by **11:59pm on Sunday May 25 AEST - this is the end of week 12.**

There will be **a late submission period of 5 days** for this assignment, which will expire at **11:59pm on Friday May 30 AEST**. Late submissions that are received before the end of this late submission period will **attract a late penalty of 10% per day (or part thereof) of the marks awarded on the assignment**, unless an extension has been granted by the school or as part of an existing EAA provision (see below for details).

Submissions that are received after the late submission period expires at **11:59pm on Friday May 30 AEST** will not be assessed unless some prior arrangement has been organised by the **CSIT OUA administrator** (ouacsit@rmit.edu.au) in response to a request for an extension.

Your instructor does not have the authority to negotiate or approve extensions, so all such requests should be sent to the email address listed above.

Special Note for students who have had EAA provisions organised by the DLU

If you are a student who has EAA provisions organised by the DLU then you must negotiate any extension that you may need with the instructor well in advance of the deadline (at least 72 hours in advance of the on-time submission deadline would be preferred).

All questions regarding this assignment should be directed to the appropriate Assignment 3 Discussion forum on blackboard.