Electronics and
Computer Science

UNIVERSITY OF Southampton

Coursework Specification

Module:	COMP1202: Programming 1		
Assignment:	Programming coursework	**Weighting:**	40%
Lecturers:	Jian Shi, Heather Packer, Thai Son Hoang		
Deadline:	6/Dec/2019	**Feedback:**	15/Jan/2020

Coursework Aims

This coursework allows you to demonstrate that you:

- Understand how to construct simple classes and implement methods.
- Are able to take simple pseudo-code descriptions and construct working Java code from them.
- Can write a working program to perform a complex task.
- Have an understanding of object-oriented programming.
- Can correctly use polymorphism and I/O.
- Can write code that is understandable and conforms to good coding practice.

Contacts

General programming queries should be made to your demonstrator in the timetabled labs.

Queries about the coursework specification should be made to Jian Shi (Jian.Shi@soton.ac.uk).

Any issues that may affect your ability to complete the coursework should be made known to **Jian Shi** (Jian.Shi@soton.ac.uk) or **Son Hoang** (T.S.Hoang@ecs.soton.ac.uk), ideally before the submission deadline.

Instructions

Late submissions will be penalised at 10% per working day.
No work can be accepted after feedback has been given.
You should expect to spend up to 50 hours on this assignment.
Please note the University regulations regarding academic integrity.

ECS Java Training School

Specification

The aim of this coursework is to construct a simple simulation of a Java training school. This school will admit students and they will receive training by a team of specialist instructors.

You are not required to have any knowledge of running and managing a training school in order to complete this coursework and no management accuracy is claimed in the representation of marketing practices presented here.

The ECS Java training school may bear some superficial similarities to a real school but is grossly simplified and in most cases likely to be quite different to how a real school might work.

Your task is to implement the specification as written.

In some cases, specific names are defined or specific properties given for classes. This is mainly to ensure that we can easily read your code, and find the parts that are most relevant to the marking scheme. You will not gain marks for deviating from the specification in order to increase the realism of the simulation. In fact, it may cost you marks and make it harder to assess what you have written. In some cases we wish to know that you can use specific Java constructs, so if it specifies an `ArrayList` then we want to see you use an `ArrayList`. There might be other ways of implementing the same feature but we are trying to assess your ability to use specific constructs. Where the specification is less explicit in the way something is implemented (for example, by omitted what data types to use) you are free to make your own choices but will be assessed on how sensible those choices are. Comments in your code should explain the choices you have made.

An FAQ will be kept on the notes pages of answers that we give to questions about the coursework. If issues are found with the specification it will be revised if necessary. If questions arise as to how certain aspects might be implemented then suggestions of approaches may be made but these will be suggestions only and not defined parts of the specification.

How the ECS Java training school works

For this coursework you are expected to follow the specification of the school, students and instructors as set out below. This will not correspond exactly to a real school or instructors in reality but we have chosen particular aspects for you to model **that help you to demonstrate your Java programming**.

The school offers training courses on various Java's related subjects. Some examples of subjects are showed in

Table 1. Each subject has a unique ID and belong to some area of specialism. The duration for the course associated with each subject is specified. The specialism determines the type of instructors that can deliver the course. For example, for the subject titled "Basics (variables, conditionals, methods, loop, etc.)" (with ID "1"), the duration for a course covering the subject is 5 days. Furthermore, the subject belongs to Specialism "1" and can be taught by any teachers.

Subject	ID	Specialism	Duration	Instructor
Basics (variables, conditionals, methods, loop, etc.)	1	1	5 days	Any Teacher
Lab 1 (Basics)	2	2	2 days	Any Teacher or Demonstrator
Arrays	3	1	3 days	Any Teacher
Algorithms	4	1	1 day	Any Teacher
Testing and Debugging	5	1	3 days	Any Teacher
Lab 2 (Advanced)	6	2	2 days	Any Teacher or Demonstrator
Object-Oriented 1 (class, inheritance, etc.)	7	3	6 days	OOTrainer
Object-Oriented 2 (encapsulation, exception, etc.)	8	3	7 days	OOTrainer
Lab 3 (Object-Oriented)	9	2	3 days	Any Teacher or Demonstrator
Graphics	10	4	5 days	GUITrainer
Controllers	11	4	2 days	GUITrainer
Lab 4 (GUI)	12	2	3 days	Any Teacher or Demonstrator

Table 1. Example of Subjects

There are a number of concepts, people, and procedures that contribute to this simulation. For our purposes these include:

Students: Students will be admitted to the school to study a number of subjects. These subjects need to be taught by the instructors. A student can only enrol to at most **ONE** course at a time. Once the students have graduated they can leave the school.

Courses: The school will create courses to teach different subjects. Each course is associated with exactly **ONE** subject. Each course will have a maximum 3 students and must be taught by an instructor.

Instructors: The school will be staffed by a number of instructors. The instructors may have particular specialisms that let them teach particular subjects. Some subjects can only be taught by instructors with the right specialism. An instructor will teach at most **ONE** course at a time.

School: For our purposes a school will manages the courses, students, and instructors.

Administrator: The school administrator is in charge of registering/deregistering students and instructors to the school.

The next sections will take you through the construction of the various required concepts. You are recommended to follow this sequence of development as it will allow you to slowly add more functionality and complexity to your final simulation. The first steps will provide more detailed information to get you started. It is important that you conform to the specification given. If properties and methods are written in

red then they are expected to appear that way for the test harness and our marking harnesses to work properly.

Part 1 – The Subject and Course classes

Subject

The first class you will need to create is a class that represents a *Subject.* The properties for **the `Subject` class** that you will need to define are:

- `id` – this is the unique ID of the subject,
- `specialism` – this is the specialism ID of the subject.
- `duration` – this is the duration (number of days) required for any course covering the subject.

Define these as you think appropriate, and create a constructor that initialises them. Define accessor methods `getID()`, `getSpecialism()`, and `getDuration()` to return the properties accordingly.

Now define the following additional property:

- `description` – this is the string description of the subject.

Define the accessor and mutator methods `getDescription()` and `setDescription(String)` for getting and setting this property accordingly.

Course

The second class you will need to create is to represent a *Course*. The properties of the course are:

- `subject` – this is the subject associated with the course.
- `daysUntilStarts` – this is the number of days until the course starts
- `daysToRun` – this is the number of days that the course still has to run.

Define these as you think appropriate, and create a constructor `Course(Subject, int)` that initialises them. Note that the initial value for `daysToRun` depend on the input *Subject*. Implement `getSubject()` method to return the subject associated with the course.

Implement the accessor method `getStatus()` to return the status of a course as follows:

- if the course has not started, then return **the negative of the number of days until the course starts**.
- if the course is currently running, then return **the number of days left until the course finishes**.
- If the course has finished, then return `0`.

Finally, define method `aDayPasses()` to advance one day for the course. The method must change the value of properties `daysUntilStarts` and `daysToRun` accordingly.

BY THIS STAGE you should have a `Subject` class, and a `Course` class connected to the `Subject` class.

Figure 1. Subjects and Courses

You can now test your `Subject` *and* `Course` *classes by creating a main method and to create some objects of these classes. You call* `aDayPasses()` *method on the* `Course` *objects and check how their status changes.*

Part 2 – The `Student` class

Person

The first step in this part is to create is an class that represents a *Person*. **The `Person` class** will be the basis for your `Student` and `Instructor` classes. This is an class that defines the basic properties and methods that all the different people classes will use. The properties that you will need to define are:

- `name` – this is the name of the person,
- `gender` – this defines whether the person is male (`'M'`) or female (`'F'`),
- `age` – this says how old the person is in years.

Define these as you think appropriate, and create a **constructor** that initialises them.

The *Person* class will also need to define some methods:

- `getName()` – returns the name of the person,
- `getGender()` – returns the gender (as a `char`) of the person,
- `setAge(int)` – set the age of the person,
- `getAge()` – returns the age of the person.

Student

To model people in different roles, you will use inheritance. Start off by creating *Students***.**

Define your second class, `Student`, that inherits from the `Person` class. An important property of *Students* are the certificates for the subjects that they have taken.

- `certificates` – this is the collections of subject IDs that the student has obtained.

Use an `ArrayList` to represent the certificates of a student. Create a constructor to initialise this property accordingly. To manage the student certificates, we will need a number of methods.

- `graduate(Subject)` – this adds the ID of the subject to the collections of certificates.
- `getCertificates()` – this returns the `ArrayList` of certificates obtained by the students.
- `hasCertificate(Subject)` – this checks whether or not the student has already obtained the certificate for the input subject.

We now need to connect the `Student` class with the `Course` class. Each course needs to maintain the collection of enrolled students. **Define an appropriate property** in the `Course` class and implement the following method.

- `enrolStudent(Student)` – this adds the student to the collection of enrolled students. The method return a `boolean` indicating if the registration is successful or not. Reasons for unsuccessful registration include:
  - The course is full (each course has a maximum of `3` students).
  - The course has already started.
- `getSize()` – this returns the number of students enrolled in the course.
- `getStudents()` – this returns the `Student[]` array of students enrolled in the course.

Finally, **modify the `aDayPasses()` method** to ensure that *if the course finishes then issues the certificate for the course's subject to all the students in the course.*

BY THIS STAGE you should have an `Person` class, and the first subclass of `Person`: `Student` connected to the `Course` class. Note that we will ensure that a student enrols to at most one course at a time later.

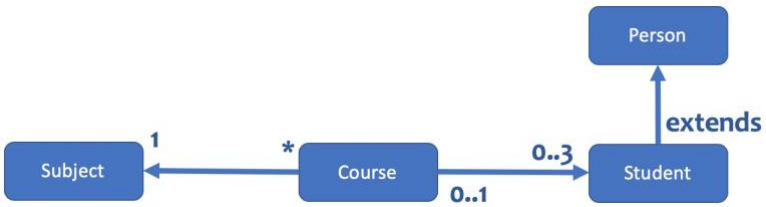
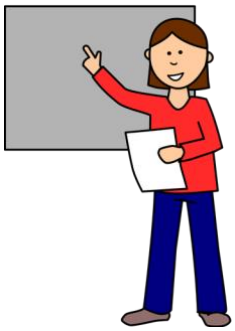


Figure 2. Students and Courses

You can now test your new `Student` class by creating a main method and to create some `Subject`, `Course`, and `Student` objects. You can register the students to the courses and call `aDayPasses()` method on the `Course` objects and check how the students obtain the certificates.

Part 3 – The `Instructor` classes



Instructors

To model the various types of *Instructors* you will need for your *School* (*Teachers, Demonstrators, OOTrainers* and *GUITrainers*) you will need to continue to use inheritance.

The diagram in Figure 3 shows you how the classes that you will create are related to each other. As you can see, both `Student` and `Instructor` inherit from `Person`. `Teacher` and `Demonstrator` are subclasses of `Instructor`, and `OOTrainer` and `GUITrainer` are subclasses of `Teacher`.

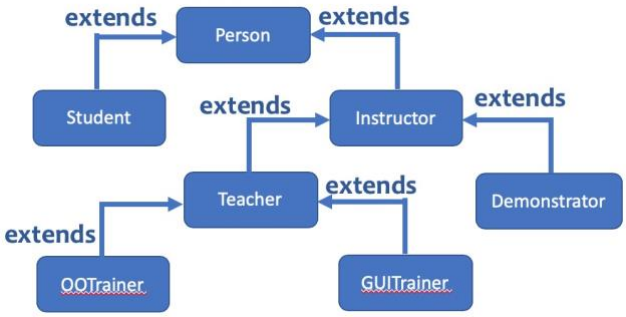


Figure 3. Inheritance Hierarchy of People

The first class to create is an **abstract** `Instructor` class. This is a class that defines the basic properties and methods that all the different instructor classes will use. The property that you will need to define are:

- `assignedCourse` – this is the course that is assigned to the instructor.

The `Instructor` class will also need to define some methods for use by the *School*.

- `assignCourse(Course)` – this assigns the input course to the instructor
- `unassignCourse()` – this removes the assigned course to the instructor
- `getAssignedCourse()` – this returns the assigned course to the instructor if any.

Define an **abstract method** `canTeach(Subject)` which returns a `boolean`. This method will be overridden by the other instructor sub-classes depending on their specialism. As we progress you may choose to add additional methods to your classes.

Teachers, Demonstrators, OOTrainers, and GUITrainers

Define the sub-classes of Instructor according to the hierarchy in Figure 3. In particular, each of the sub-classes, namely, `Teacher`, `Demonstrator`, `OOTrainer` and `GUITrainer` must implement the `canTeach(Subject)` method according to the following rules related to the subject's specialism IDs.

- `Teacher` can teach subjects with specialisms 1, 2.
- Demonstrator can teach subjects with specialism 2.
- OOTrainer can teach subjects with specialisms 1,2, 3
- GUITrainer can teach subjects with specialisms 1,2,4

To connect the `Instructor` class to the `Course` class, follow the steps below (you might want to add extra properties to the `Course` class to support the implementation of these methods):

- add a method `setInstructor(Instructor)` to the class. This method should have a return value indicating whether the assignment is successful, i.e., the instructor can teach the course,
- add a method `hasInstructor()` to return whether or not the course has an instructor,
- modify the `aDayPasses()` method to cancel the course when the course starts without any instructor (you might need to add some property to the course) or students,
- in `aDayPasses()` method, remember to unassign the instructor from the course when the course finishes,
- add a method `isCancelled()` to return weather or not the course has been cancelled.
- Remember to release the students (if any) and the instructor (if any) from the course when the course is cancelled.

BY THIS STAGE you should have an additional `Instructor` class, and various sub-classes, connected to the `Course` class.

You can now test your new `Instructor` classes by creating a main method and to create some instructors, courses, and students. You can assign the instructor and students to the courses and call `aDayPasses()` method on the `Course` object and check how the instructors' status changes.

Part 4 – The School

Our `School` class is where all our *Students* are taught and our *Instructors* work. The school must have a name. **Define the property and appropriate constructor for initialise that**. The school keeps information about the *Subjects*, *Courses*, *Students*, and *Instructors*. The `School` will need a number of methods. These should include:

- `add(Student)` – this adds a new student to the *School*.
- `remove(Student)` – this removes a student from the *School*
- `getStudents()` – this returns the students in the school.

Define similar methods for Subject, Course, Instructor accordingly, i.e., `add(Subject)`, `remove(Subject)`, `getSubjects()`, etc. Furthermore, we need to have utility methods for the simulation of the school.

- `toString()` – this *overrides* the method returning a *pretty-print string* of the school. It should contain information about subjects, courses, instructors, students and the relationship between them.
- `aDayAtSchool()` – this simulates events of a day at school (see below).

We are going to keep our simulation simple and try and avoid any complicated scheduling problems. No additional marks will be given for having really efficient scheduling systems. You are required to have a very

simple working process, it is not necessary to make this too complicated. A typical day at the school will have the following events occur.

- Create new courses: for any topic that does not have an open-for-registration course, create a new course for that subject to start in 2 days.
- Assign instructors and students to courses. The simplest way to do this is to:
    - Look at each course that requires an instructor, and go through the instructors until you find one that is free (i.e., not already teaching other courses), and can teach the course. Assign the instructor to the course.
    - Look at each students, if they are free, (i.e., not enrolled in any course) go through the courses until you find one that the student can join (e.g., not already full and the student has not got the certificate).
- Let the student learn (e.g., call aDayPasses() on each courses.)
- At the end of the day, removes any course that is cancelled or already finished.

You might need to add other properties and methods as appropriate.

BY THIS STAGE you should have implemented `School` class, connected to the `Course`, `Student`, `Instructor` classes.

You can now test your new `School` class by creating a main method and to create some instructors, subjects, students and add them to the school. Call the `aDayAtSchool()` method on the school and print the status of the school to see how it changes. You are advised to get all of this working before attempting to extend your functionality.

Part 5 – Running the simulation

In order to run our School as a simulation we are going to need a school *Administrator*. This class will have a *School*, will organise the day to day running of the *School*, i.e. run the simulation itself.

You should create a new class called `Administrator`. It should have a `School` instance and have a `run()` method which is used to run the simulation.

A typical day for a school administrator will have the following events occur.

- New students will be admitted to the school. Each day will have up to 2 students join the school. You will need to use a random number generator to decide the number of the students joining the school. You can do this with the `Toolbox` class that was provided with the labs, or by using the methods in the `java.util.Random` class directly.
- New instructor will join the school. The probabilities for a Teacher, Demonstrator, OOTrainer, and GUITrainer to join each day are 20%, 10%, 5%, and 5%, respectively. Again, use a random number generator for this.
- Run the school, i.e., call `aDayAtSchool()` method of the school.
- At the end of the day, students and instructors might leave the school
    - A free instructor (i.e., not assigned to any course) have 20% chance of leaving the school
    - If a student obtains the certificates for all subjects, the student will leave the school. Otherwise, if the student does not enrol in any course, he or she has 5% chance of leaving the school.

For properties of people, i.e., name, gender, age, you can use the random generator for creating them or use some fix value.

Define an overloaded version of `run()` that takes a **number of days as a parameter**, and simulates that number of days. After each day it should call report on the school to list information about courses

(including course status, list of enrolled students), students (certificates, enrolled course if any) and instructors (assigned course if any).
You can use print statements to record what is happening in your simulation. To slow things down a little, the following code will help you to pause for half a second between calls if you choose to include it.

```
try
{
      Thread.sleep(500);
      run();
}
catch (InterruptedException e)
{
}
```

BY THIS STAGE you should have implemented `Administrator` class, connected to a `School`
You can now test your new `Administrator` *class by creating a main method and to an instance of that for a School and call the corresponding* `run()` *methods.*

Part 6 – Reading a simulation configuration file

A good simulation will allow you to set the starting conditions for your simulation and one way of doing this is for the simulation to read in a simple configuration file. For this step you will need to use your file handling methods as well as split strings into different component parts.

Our basic configuration file will look like the example below. You may choose to extend this for your extensions, but for testing purposes your code should accept and use configuration files in this form. Each line gives information about a school student or an instructor.

```
school:ECS Java Training School
subject:Basics,1,1,5
subject:Lab 1,2,2,2
subject:Arrays,3,1,4
student:Peter,M,60
student:John,M,22
student:Annabelle,F,31
student:Maggie,F,58
student:Alex,M,23
Teacher:Yvonne,F,55
Demonstrator:Beth,F,45
OOTrainer:Chris,M,62
GUITrainer:Sarah,F,48
```

The format for the *school* is

`School:name`

The format for *Subjects is*

subject:description,subjectID,specialisationID,duration

The format for *Students is*

`student:name,gender,age`

The format for *Instructors* is

```
instructor_type:name,gender,age
```

Some example simulation files of varying complexity will be placed on the WIKI.

You should modify your main method so that it can take a file on the command line. This will enable you to start your simulation by taking the name of the configuration file and the duration (number of days) for the simulation.

```
java Administrator mySchool.txt 200
```

When the `Administrator` receives the configuration, it should read the file a line at a time. For each line the *file* will need to identify the Class, create a new class of this type, and set the appropriate parameters and add it to the School. You may find you need to create specific methods or indeed a helper class, to parse the configuration file and extract the information that the `Administrator` needs.

We are placing this code in the `Administrator` to make it simple, so your simulation is of an `Administrator` with a `School` in it.

You should now have a simulator that runs with an `Administrator` containing a `School` which has multiple Instructors *and* `Students` in it.

Exceptions

You should be trying to use exceptions in the construction of your simulator where possible. You should be catching appropriate I/O exceptions but also might consider the use of exceptions to correctly manage:

- The input of a configuration file that does not conform to the specified file format.
- …

Extensions

You are free to extend your code beyond the basic simulator as described above. You are advised that your extensions should not alter the basic structures and methods described above as marking will be looking to see that you have met the specification as we have described it. If you are in any doubt about the alterations, you are making please include your extended version in a separate directory.

Scheduling can become very complicated very quickly, so we have deliberately not made efficient scheduling a part of this coursework. Be warned if you attempt to do anything clever in this regard for your extension.

Some extensions that we would heartily recommend include:

- Implement the prerequisites for subjects and ensures that students can only enrol on a course if they have all the prerequisites for that course.
- Your Instructors might be able to teach more than one course in a day (up to a certain limit). The students can enrol in more than on course at a time (up to a certain limit). Modify your instructors, students and simulation to deal with this.
- Introduce a limit on the number of consecutive days that the students can learn and the instructors can teach. Once the limit has been reached, the student or the instructor must take a break.
- Depending on the number students, actively hire/fire the instructors accordingly. Introduce the cost for hiring/firing and maintaining the instructors and optimise the total cost of running a school.
- You might want to extend the configuration file and classes so some of the parameters of the simulation can be set in the configuration file. This might include some of the details about courses, students' enrolment, or instructors' allocation for instance.
- You might want to allow the simulation to save out the current state of a simulation to a file so that it can be reloaded and restarted at another time.

15% of the marks are available for implementing an extension. Any of the above examples would be suitable, but feel free to come up with one of your own if you like. It is not necessary to attempt multiple extensions in order to gain these marks. Please describe your extension clearly in the readme file that you submit with your code.

Space Cadets

You might want to add a GUI to your simulator so that you can visualise the state of the *School* at any given moment. **No marks are available for a GUI**, we put the suggestion forward simply for the challenge of it, although you can reasonably expect praise and glory for your efforts.

Submission

Submit **all Java source files** you have written for this coursework in a zip file **`school.zip`**. Do not submit class files. As a minimum this zip will include:

`Subject.java`, `Course.java`, `Instructor.java`, `Teacher.java`, `Demonstrator.java`, `OOTrainer.java`, `GUITrainer.java`, `School.java`, `Administrator.java`

Also include a text file **`readme.txt`** that contains a brief listing of the parts of the coursework you have attempted, describes how to run your code including command line parameters, and finally a description of the extensions you have attempted if any. Submit your files by **Friday 6th December 2019 16:00** to: https://handin.ecs.soton.ac.uk

Relevant Learning Outcomes

1. Simple object oriented terminology, including classes, objects, inheritance and methods.
2. Basic programming constructs including sequence, selection and iteration, the use of identifiers, variables and expressions, and a range of data types.
3. Good programming style
4. Analyse a problem in a systematic manner and model in an object oriented approach

Marking Scheme

Criterion	Description	Outcomes	Total
Compilation	Applications that compile and run.	1	20
Specification	Meeting the specification properly.	1,2,4	40
Extensions	Completion of one or more extensions.	1, 2,4	15
Style	Good coding style.	3	25