Electronics and
Computer Science

UNIVERSITY OF
Southampton

Coursework Specification

Module:	COMP1202: Programming 1		
Assignment:	Programming coursework	**Weighting:**	30%
Lecturers:	David Millard (dem), Mark Weal (mjw), Rikki Prince (rfp)		
Deadline:	9/Dec/2015	**Feedback:**	8/Jan/2016

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 it is understandable and conforms to good coding practice.

Contacts

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

Queries about the coursework specification should be made to Mark Weal (mjw@ecs.soton.ac.uk).

Any issues that may affect your ability to complete the coursework should be made known to Mark Weal (mjw@ecs.soton.ac.uk) or David Millard (dem@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 Zoo

Specification

The aim of this coursework is to construct a simple simulation of a zoo. This zoo will have enclosures of animals that are looked after by zookeepers. The animals will eat food, produce waste and have special treats to keep them healthy.

You are not required to have any knowledge of zoos or animals or zookeeping in order to complete this coursework and no accuracy is claimed in the representation of zoos and animals presented here.

The zoo may bear some superficial similarities to a real zoo but is grossly simplified and in most cases likely to be highly different to how a real zoo might work.

Your task is to implement the specification as written.

In some cases, specific interfaces are defined or specific properties given for classes. This is mainly to ensure that the code you are producing will work with the test harnesses we will provide to help you test your work and make sure you are on the right track. You will not gain marks for deviating from the specification in order to increase the realism of the zoo, in fact it may well cost you marks and make it much 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 may well be other ways of implementing the same thing but we are trying to assess your ability to use specific constructs. Where the specification is less explicit in the way something is implemented you are free to make your own choices but will be assessed on how sensible those choices are. Comments in your code can be used to explain why you have made the choices you have.

An FAQ will be kept on the notes pages of answers that I give to questions I receive 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 it will be made clear these are suggestions and not defined parts of the specification.

How ECS Zoo works

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

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

Animals: All good zoos will have a range of exciting animals for people to see and your zoo is no exception. Each animal will need food to eat, produce waste, live a happy life for a defined period of time, and sometimes have offspring.

Enclosures: The animals have to live somewhere. Each enclosure can hold a number of animals, if appropriate of different types. At any given moment it will have an amount of food available and a pile of waste to be cleared up.

Zookeepers : The zookeepers are responsible for looking after the animals. They will be trained on looking after specific types of animals and will be responsible for feeding the animals, cleaning out the enclosures.

Zoo: The zoo contains the code that orchestrates the overall zoo, deciding which animals go in which enclosures, ordering the construction of new enclosures and hiring Zookeepers to look after the animals.

The next sections will take you through the construction of the various people, animals and buildings you require. You are recommended to follow this sequence of development as it will allow you to slowly add more functionality and complexity to your simulation. The first steps will provide more detailed information to get you started. It is important that you conform to the specification given. If methods are written in red then they are expected to appear that way for the test harness to work properly.

Part 1 – Modelling Animals

The first class you will need to create is an abstract class that represents an *Animal*. You may be able to re-use some of the code you have created in Lab7 for this, **but note this specification is slightly different** so it will need modifying to work accordingly.

The abstract *Animal* class will be the basis for all your different animal classes.

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

- `age` – This integer says how old the *Animal* is in months.
- `gender` – This char says whether the *Animal* is Male ('m') or Female ('f').
- `eats` – specifies the types of food that the *Animal* will eat. This should be an array of Strings representing the types of food that the *Animal* can eat.
- `health` – this species how healthy the animal is. It is a value between 0 and 10, with 10 being very healthy and 0 being dead. Its health reduces by 2 every month but increases if it eats, and/or receives a special treat from a zookeeper.
- `lifeExpectancy` – An integer that specifies the average age that the *Animal* lives to in months.

You can add constructors to the *Animal* class that creates an animal with some default values.

The abstract *Animal* class will also need to define some methods for use by the *Zoo* and *Zookeepers*. These methods can be overridden by the other specific sub-classes. The methods you need to create are *getAge(), getGender(), getLifeExpectancy(), canEat(String),* that returns true if the Animal can eat that food, *eat()*, *decreaseHealth()*, treat(), and the abstract method *aMonthPasses()*,which will have a return type of `boolean`. *aMonthPasses()* will be called on each Animal in the zoo once each month and will contain all the code that enables the animal to do what they need to do and to change its state over time. Principally, for *Animals* it will allow them to eat, excrete and grow old. We will fill in the details of this method later.

You should now have an *Animal* class. As we progress you may need to, and choose to add additional methods to your abstract class. Some aspects, such as the lifeExpectancy, will be specific for each type of *Animal* you go on to create.

Part 2 – Modelling food and the Foodstore

There are different types of food that give different amounts of energy to the *Animals*. Different Animals will also only eat certain types of Food. An animal can eat one item of food each month if it is available. Table 1. below lists different food that the animals in your zoo might want to eat. Each food type has different properties, it gives a certain amount of health when eaten and leads to a certain amount of waste being produced.

name	health	waste
hay	+1	4
steak	+3	4
fruit	+2	3
celery	0	1
fish	+3	2
ice cream	+1	3

Table 1: Types of food and relevant information

Food is held in a *Foodstore* class. Each *Enclosure* will have a *Foodstore* where the animals get their food from, and the Zoo will have a *Foodstore* where food is delivered to the Zoo.

Your *Foodstore* class will need a number of access methods. *addFood(String, int)* that adds a number of items of food of the specified type to the *Foodstore*, i.e. addFood("steak",5). *takeFood(String)*, which is used by the animals when they eat in their Enclosure, and also by the *Zookeepers* as they move food from the *Zoo* foodstore to the *Enclosure* foodstore. You may find you want to add other access functions for the foodstores later on.

How you model the storage in the *Foodstore* is up to you. It would be fine to have an int variable for each type of food, holding the quantity of that food in the *Foodstore*. If you want to make your code more flexible you might choose to use a structure such as a *Hashtable* with the key being the food name and the value being the quantity.

The health given by food and waste produced will be modelled in the eat method of the animals later on to keep things simple.

At this stage you should now be able to create a *Foodstore* and add varying quantities of various types of food to it.

Part 3 – Modelling Enclosures

Our *Enclosure* class is where our *Animals* will live inside. You should use an `ArrayList` to represent the *Animals* in the *Enclosure*. For the purposes of our simulation we will allow our *Enclosures* to have no more than 20 *Animals* in them. Your code should take this into account and restrict the *Enclosures* appropriately. The *Enclosure* also has a *Foodstore* containing varying amounts of food in it, potentially of different types. The *Zookeepers* will fill the *Foodstore* from the *Zoo Foodstore*. Each Enclosure will also accrue a certain amount of Animal waste, which can be stored in an int.

The *Enclosure* will need a number of access methods. These should include:

- *addAnimal (`Animal`)* – this adds an *Animal* to the *Enclosure*.
- *removeAnimal(`Animal`)* – removes the specified *Animal* from the *Enclosure*.
- *removeWaste(`int`)* – removes the specified amount of waste from the *Enclosure*.
- *addWaste(int)* – adds an amount of waste (if animal has eaten).
- *getWasteSize()* – returns the amount of waste in the *Enclosure.*
- *getFoodstore()* – returns the *Foodstore* for the *Enclosure*.
- *size()* – this returns the number of *Animals* in the *Enclosure*.
- *aMonthPasses()* –this method is called as part of the simulation to trigger a new month for the Enclosure. When it is called, it will in turn call aMonthPasses() on all the Animals in the Enclosure.

You should now have an *Enclosure* class that you can add *Animals* to.

N.B. Now you have an Enclosure class, you will need to add an additional property to your Animal class to hold the animals enclosure. An access method of *setEnclosure(Enclosure)* will also be needed. The *Animal* needs to know what *Enclosure* it is in so it can access the *Foodstore* to eat.

We can now move on to create some actual *Animals* to add to our *Enclosure*. You can then test this using a main method in your *Enclosure* class to create new *Animals* and add them to the *Enclosure*.

Part 4 – Modelling different Animals

To model the various types of *Animal* you will need for your *Zoo* you will use inheritance. For this simple simulation is may seem more complicated than is necessary, but the encapsulation of different animals in their own classes will allow the simulation to be extended more easily at a later date.

The following simple diagram Fig 1. shows you how some of the classes that you will create are related to each other. As you can see, both *BigCat* and *Ape* inherit from *Animal* and are both Abstract classes. This allows us to model the fact that they both eat the same type of food at the abstract level. *Chimpanzee* and *Gorrilla* are a subclass of *Ape*. *Elephant*, and the other animals you might choose to include, are direct extensions of *Animal* in this model.

You already have your abstract *Animal* class, we now need to create some specific instances of Animals. We are going to use some further abstract classes however as some Animals share common eating habits. For each of your animal classes you will need to write code in the aMonthPasses() method. A typical month for an animal will follow the following procedure

- Reduce health by 2
- Eat a piece of food if available
- Produce an amount of waste if food eaten.

Some of this you may choose to implement in the superclass if appropriate.

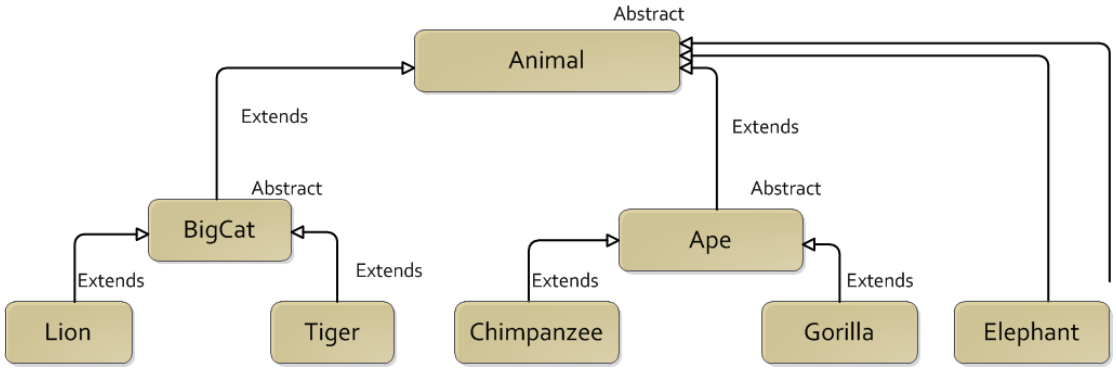

Fig 1: Partial Class diagram for Animal classes.

Animal	**LifeExpectancy (months)**	**Eats**	**Treat**
Lion (BigCat)	24	steak, celery	stroked() +2 health
Tiger (BigCat)	24	steak, celery	stroked() +3 health
Elephant	36	hay, fruit	bath() +5 health
Giraffe	28	hay, fruit	neckMassage() +4 health
Penguin	15	fish, ice cream	watchAFilm () +2 health
Chimpanzee (Ape)	24	fruit, ice cream	playChase() +4 health
Bear	18	fish, steak	hug() +3 health
Gorilla (Ape)	32	Fruit, ice cream	painting() +4 health

Table 2: Types of animal, food eaten and special treat.[1]

[1] All animal information has been meticulously researched by asking a 4yr old and a 6yr old. Any factual inaccuracies are to be disregarded if found.

Each *Animal* can also be treated and so needs to implement a method that can be called to treat the *Animal*. In the abstract class you should have implemented a treat() method and you can over-ride this in the subclasses to call the appropriate specific method for the *Animal*.

You should now have some *Animals* of different types that you can add to *Enclosures*.

Part 5 – Modelling the Zoo Lifecycle

In order to run our Zoo as a simulation we are going to need a *Zoo* class. This class will have a number of *Enclosures* containing *Animals*, a collection of *Zookeepers* assigned to those enclosures, and will organise the day to day running of the zoo, i.e. run the simulation itself.

You should create a new class called *Zoo*. It should have an Array of *Enclosures* and later on, an ArrayList of *Zookeepers*. It should have a aMonthPasses() method which is used to run the Simulation.

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 month at the zoo will have the following events occur.

- Each Enclosure will have aMonthPasses() called on it. This will in turn then call the method on all the Animals in the Enclosure.
- Each Zookeeper (once these have been implemented) will have aMonthPasses() called on it. This will allow them to perform their duties.
- Each Animal in the zoo will be checked to see if its health is 0. If so, it will be removed from the zoo.
- The zoo Foodstore can be restocked.

The list of events above represent the algorithm for running your zoo. This should be implemented inside the aMonthPasses() method of your *Zoo*.

In a go() method you can loop round the simulation by calling aMonthPasses(). You can use print statements to record what is happening in your simulation. To slow things down a little, the following code will allow you to pause for half a second between months if you choose to include it.

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

Part 6 – Modelling the Zookeepers

The *Zookeeper* comes in three different types. There is a general Zookeeper, and two specialist Zookeepers called the PlayZookeeper and the PhysioZookeeper. The Class diagram for the zookeepers can be seen in Figure 2. below. Note, the Zookeeper is not an abstract class and you will be able to create instances of *Zookeeper*. They will do the basic zookeeper

duties but won't be able to carry out some of the more specialist duties of the other two.

You may need to create some member variables and access methods for the zookeepers but the main ones you will require is an *Enclosure*, so the *Zookeeper* knows which *Animals* they are looking after, and they will need to have access to the Zoo *Foodstore* .

Fig 2: Class diagram for Zookeeper classes.

The Zookeeper should also have aMonthPasses() method, which is called every month by the Zoo as part of the simulation. Inside this method the Zookeeper will carry out their duties which will involve:

- Moving up to 20 items of food from the *Zoo Foodstore* to the *Enclosure Foodstore*.
- Removing up to 20 units of waste from the *Enclosure*.
- Giving a treat to up to two animals.

The generic Zookeeper is able to call stroke() or hug() on any Animal that has that treat method. Only the PlayZookeeper is able to call the watchAFilm(), playChase() or painting() methods on Animals which have that method. PhysioZookeepers are able to call neckMassage() or bath().

Your Animal class should have a treat() method and if it makes things easier you can just call the treat() method on the Animal to give it its treat, relying on it calling the appropriate method inside the class. You should implement some mechanism to ensure that only specialist zookeepers are able to carry out certain types of treat.

You should now have a *Zookeeper* class. One this is implemented you will need to revisit your *Zoo* class to add the ArrayList of *Zookeepers* and add the additional calls to the *Zookeepers* in the aMonthPasses() method.

Part 7 – 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 an enclosure, zoo animal or a zookeeper.

```
zoo:
steak:50
hay:30
enclosure:50
hay:20
steak:5
celery:10
lion:M,12,5,1
lion:F,10,4
playZookeeper:
```

The format for the *Enclosure* is

enclosure:waste

for food

type:quantity

for *Animals*

animal:gender,age,health,enclosure

for *ZooKeepers* is

type:

Some example simulation files of varying complexity will be placed on the WIKI. There are currently no properties for the zookeepers but this leaves scope for future extension.

You should modify the main method in your *Zoo* class so that it can take a file on the command line. This will enable you to start your simulation by typing

java Zoo myZoo.txt

When the *Zoo* 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 where appropriate, and set the appropriate parameters. If creating a new *Enclosure* it should added to the *Zoo*, if creating a new *ZooKeeper*, add it to the *Zoo*. If no enclosures have been created then it is assumed the food is for the Zoo Foodstore. The last enclosure created is assumed to be the assigned enclosure for any food, animals or zookeepers that follow until a new *Enclosure* is created. 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 *Zoo* needs.

We are placing this code in the *Zoo* to make it simple. You could create a new Class *Simulator* to perform this function if you wish, provided it is clear in the Readme.txt file you supply where the code can be found.

You should now have a simulator that runs with a *Zoo* containing multiple *Enclosures* containing *Animals and Zookeepers* 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.
- Attempts to add too many animals.
- A *Zookeeper* treating an *Animal* they're not qualified to treat.
- An *Animal* trying to eat food of the wrong type.
- …

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 and resource management 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:

- Try modifying the simulator so that *animals* can breed. If a male and female animal of the same time are in the same enclosure then there is a random chance that they will have a baby.
- Try adding code so that your zookeepers can be assigned to more than one enclosure and look after different types of animals.
- 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 food properties for instance.
- You might want to allow the simulation to save out the current state of a zoo to a file so that it can be reloaded and restarted at another time. To do this you may be able to use the current designed configuration file, or perhaps you need to extend it in some way.

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 *Zoo* at any given moment. **No marks are available for a GUI**, we put the suggestion forward simply for the challenge of it.

Submission

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

Zoo.java, Enclosure.java, FoodStore.java, Animal.java, BigCat.java, Lion.java, Zookeeper.java.

Also include a text file **zoo.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 **Wednesday 9th December 2015 17:00** 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