Continuing the implementation of last chapters' assignment, we will this time start adding some functionality to our zoo.

1.1.Every animal a possible killer

First off, create another *final* field of type *Double* in the Animal class called *maintenanceCost*. Keep in mind, final fields ***have to be initialized in a constructor*** and ***their values cannot be changed once initialized!*** This maintenance cost will hold how many hours per week will this animal require attention from one (or more) employees of the zoo. Values for this field should range from 0.1 to 8.0 and initialization should be done in the appropriate factory classes. Observe what happens when you add a final field to this class.
On the same note, add another *final* field of type *double* to the animal class called *dangerPerc*. This field will represent how dangerous an animal is in %. For example, a lion could be 90% dangerous meaning, in 90% of the times you encounter a lion head-on, you will get eaten by that lion. Unfortunate, I know. Try to set some values based on how dangerous you think an animal is – we will need this field later on. Also, these values must be in the [0, 1] interval.

Next thing we have to do is create an interface for this Animal class. Let's call our interface *Killer* and may it contain 1 method declaration:

```
public boolean kill();
```

After creating this interface and its method, make the Animal class implement this interface. Now this method returns whether or not the animal will *kill* the person (or animal) it is interacting with, so, in order to do that, you will have to:

- Generate a random number between 0 and 1
- If the generated number is strictly bellow the *dangerPerc* field, then return true (meaning, the animal kills the interacting entity)
- Else, return false.

We will use this method in the following section where zoo employees will be added.

1.2. Every animal taken care of

A very simple requirement: add a *takenCareOf* (Boolean) field to the Animal class and create getters/setters for it. Make sure that by default, it is set to *false* at creation time.

1.3. Employees architecture

Of course, ***Zoowsome*** cannot run only on animals! It needs a manager, investors but most importantly, *caretakers*. All these are *employees* of the zoo, so, naturally, we will create our entities (classes) in the *javasmmr.zoowsome.models.employees* package.

Consider creating a more general *abstract Employee* class. This class should contain 4 fields: *name (String), id (Long + unique!), salary (BigDecimal)* and *isDead (Boolean).* For now, generating a pseudo-random 13 digit code should suffice for the id field when creating an Employee, however, later in the application when storing employees somewhere will become a problem, duplicate checks will have to be done. After creating a getter/setter for all these fields and creating a constructor with all 3 fields as parameters, let's move on to an implementation of this class. Also, make sure that at creation time, each employee is ***alive***.

1.4. Caretakers and factories

Create a *Caretaker* class which extends the *Employee* class but has one extra field: *workingHours* (Double) which represents the number of hours a caretaker can work for in a week. (Of course, do not forget to create a getter/setter for this field)

Use the same architecture based on factories and abstract factories to create your employees (in this case, you will only have to deal with caretakers, for now, however, make sure to have an *EmployeeAbstractFactory* with an *EmployeeFactory getEmployeeFactory(String type)* method similar to what the *SpeciesFactory* did in last chapter's assignment and, for each type of employee, both add specific entries in the Constants class and create a factory for it. For now, you will only have a Caretaker factory to create but if need be, you'll easily extend this to more than 1 type of employee using this architecture. Be sure to do all this in the *javasmmr.zoowsome.services.factories* package. (One idea would be to split the *factories* package in two different packages to separate Animal factories from Employee factories – this is however optional)

1.5. Caretaker interface

Before moving on to our requirement for this assignment, let's create an interface for our *Caretaker* class which will allow it to actually take care of any animal. Let's name this interface *Caretaker_I* (***_I*** is needed to differentiate between the interface and the actual class). This interface should have just one method:

```java
public String takeCareOf(Animal animal);
```

Curious thing that this should return a String, yes, but all shall be explained.
As you might already suspect, every caretaker will have to … well, take care of animals. But there are at least 3 possibilities which might occur here!

1) The caretaker inspects the animal and sees that he/she does not have enough time to take care of this animal
2) The animal kills the caretaker when he/she tries to interact with it
3) The caretaker successfully spends his/her time taking care of the animal

Seeing that we have 3 possibilities here, we need to be informed of what actually happened when we tried to take care of an animal, so, one way of doing it is by returning some sort of standardized String (message) back as a result. So, in order to do this, in your Constants class, add the following class:

```java
public static final class Employees {

    public static final class Caretakers {
        public static final String TCO_SUCCESS = "SUCCESS";
        public static final String TCO_KILLED = "KILLED";
```

```
                public static final String TCO_NO_TIME = "NO_TIME";
            }
        }
```
right beneath the *Animal* class.
As you might have guessed, *TCO* stands for *T*aking *C*are *Of* and each message represents an outcome of the *takeCareOf* method. The pseudocode for this method reads like this:

```
    takeCareOf(Animal a) {

        if (a.kill()) {
            return Constants.Employees.Caretakers.TCO_KILLED;
        }

        if (this.workingHours < a.getMaintenanceCost()){
            return Constants.Employees.Caretakers.TCO_NO_TIME;
        }

        Set the animal takenCareOf flag to true
        Subtract the maintenance cost from the caretakers working hours

        return Constants.Employees.Caretakers.TCO_SUCCESS;

    }
```

A rather interesting remark is that in most cases when dealing with web applications, this is the way server-side validations work. For example, if you have a form in which you have to fill in your username, password (+ repeat password) and your email, this information might be send to the server and validations will be done on those fields. If something is wrong with your email for example, the server may append to the response something like "error.email = not valid" and because you have this message in the response from the server, your form will become invalid and the email field will turn red. This is of course a very trivial and a not-so "exactly how it - actually works" example, but the point is, if more than 1 thing can go wrong and you need to do a specific and different action based on every outcome, you might need to return something like a string value OR an integer to signal what went wrong!

1.6. Russian-rouletting with Animals and Caretakers

Now that you have caretakers and animals, we can simulate how a bunch of caretakers might … take care of all the animals we have created. Still, in the *MainController* create a few Caretakers and iterate through all the animals and try to take care of them all. The code should look like this:

```
for each caretaker c {
    for each animal a
    {
        if(caretaker is not dead AND the animal has not been taken care of){
            result = c.takeCareOf(a)
            if(result.equals(Constants.Employees.Caretakers.TCO_KILLED)){
                    declare the caretaker dead
```

```
                  }
              else if(result.equals(Constants.Employees.Caretakers.TCO_NO_TIME){
                                  skip to the next animal, try to take care of that
                  }
              else {
                  a.setTakenCareOf(true)
                  }
              }
         }
}
```

At the end, go through the whole list of animals again and see whether or not all have been taken care of. Print out something relevant to the console in each case.

Ps: out of curiosity, try deleting an element from an ArrayList while iterating through it. Does it work without using an *Iterator*? If not, why?

Twist 1:

Just to add a slight degree of realism and introduce the concept of *predisposition to kill*. Let us assume that some animals are predisposed to killing at certain times in the day. For example, assume that spiders get a 25% chance of killing someone if they encounter that someone between 11 PM and 6 AM. With this being the case, you will need to:

- Add a ***public double*** *getPredisposition();* method in your Killer interface. In the Animal class class, this method returns 0 because we do not know which animal we are dealing with so by default, all animals have a 0% extra *predisposition to kill.*
- This method will need to be invoked from the *kill()* method in such a way that it adds up to the initial *dangerPerc* returned.
- For every animal that could have some sort of predisposition at certain times in the day, you will need to return a specific (your choice) value by overriding the getPredisposition() method. Take the above example: check if the time is between 11 PM and 6 AM for spiders and return 0.25.
- You will need to find a way to get the current system time and check whether or not it is between these predefined intervals.
- The best thing is that, even if you override only this specific method in a subclass but do not override the *kill* method, the overridden implementation of getPredisposition() will be taken into consideration in the *kill* method. Try it out.
- You may even consider cases in which some animals are predisposed to kill on certain days of the week, intervals of month and so on – feel free to choose whatever approach you want.

What will this twist accomplish? You'll get to familiarize yourself a bit with working with *Time* and *Dates* in Java.