Unfortunately, this assignment requires you to have done the previous one as it would not make sense otherwise. Make sure to have it completed before moving forward.

4.1 Repositories – aren't ours a bit redundant?

You may have noticed that our two repositories contain 90% (or more) the same code. Usually, this means we can work on abstracting them and in our case, we will be able to use generics as well.

There are a few hints which might trigger you to think that applying generics is a solution to a cleaner and more well designed code. For example, take a look at our *AnimalRepository*:

```
public void save(ArrayList<Animal> animals)
…
        for (XML_Parsable animal : animals)
        …
```

This we notice that we treat our "animals" as XML_Parsable objects. That means that if we have more entities which implement this XML_Parsable interface we might actually be able to *save* them using the exact same code, right? But still, there are a few things which are "animal-specific" here, right? For example

```
Constants.XML_TAGS.ANIMAL;
```

This tag should only be used when encoding animals, right? Also, the ***XML_FILENAME*** has to correspond to the animal file, mainly ***Animal.xml***. But at least for the save method, that's about it, right? I mean, honestly, is it really worth having 100 lines of duplicate code for only 2 variables? Isn't there a solution to all this? Of course there is – Generics + using a constructor to set our two variables which have to differ from repository to repository.

4.2 Abstracting and generalizing the *save* method

Create another (abstract class) repository called *EntityRepository* and place it in the same package as where the two other repositories are. Write the following:

```
package javasmmr.zoowsome.repositories;

public abstract class EntityRepository {

        private final String xmlFilename;
        private final String entityTag;

        public EntityRepository(String xmlFilename, String entityTag) {
                this.xmlFilename = xmlFilename;
                this.entityTag = entityTag;
        }

}
```

Now copy-paste your *save* method from either your AnimalRepository class or your EmployeeRepository class in this new repo and replace the two things which are now passed as constructor arguments: the xml filename and the entity tag. So instead of ***XML_FILENAME*** you should have `this.xmlFilename` and instead of `Constants.XML_TAGS.`***ANIMAL*** you should have `this.entityTag`.

This partially solves our problem. But still, our method takes as parameter a list of Animals. Let's try to use generics to solve this.
In the class definition, you should write

```
public abstract class EntityRepository<T extends XML_Parsable>
```

Notice the `<T extends XML_Parsable>` at the end. This simply states that we can add as a generic to this class any class which extends or implements XML_Parsable (in our case *implements* because XML_Parsable is an interface).

Where do we use this **T**? Well, why not in the return type of our save method. You should have something like:

```
public void save(ArrayList<T> entities) throws FileNotFoundException,
XMLStreamException { … }
```

Of course, it is worth changing the name from 'animals' to 'entities' just to avoid confusion. Don't forget to transform your for loop in:

```
for (XML_Parsable entitity : entities) { … }
```

After doing all this, you may find it pleasantly surprising that you have no errors. I mean come on, there should be at least something wrong there, right? Well, actually, no. That is perfectly *legit* code. Now let's see how to integrate this in the other repositories.
Change your AnimalRepository such that it looks like this:

```
public class AnimalRepository extends EntityRepository<Animal> {

	private static final String XML_FILENAME = "Animals.xml";

	public AnimalRepository() {
		super(XML_FILENAME, Constants.XML_TAGS.ANIMAL);
	}
	…
}
```

Completely remove the save method from the *AnimalRepository* class. Do the same thing but with the Employee repository (of course, keep in mind the different filenames and tags) and again, remove the *save* method from there as well. Re-run your program and voila. Everything should work exactly like before, however, you just did a code-makeover and your classes are now cleaner and less redundant. But still, we have the *load* method which we should take care of – surely, we will do this in a very similar fashion.

4.3 Abstracting and generalizing the *load* method

Start by moving the whole *load* method from your *AnimalRepository* class to the *EntityRepository* class. Do the same changes as you did before, replacing

```
public ArrayList<Animal> load()
```

with

```
public ArrayList<T> load()
```

and

```
ArrayList<Animal> animals = new ArrayList<Animal>();
```

with

```
ArrayList<T> entities = new ArrayList<T>();
```

You now may have a few errors – but that is to be expected.
Now that we almost successfully moved the *load* method in the *EntityRepository* you may ask – why do we still need the other 2 repositories? I mean clearly, we can at this stage get rid of them and simply make this a non-abstract generic class. Yes, we could, however, the *load* method behaves slightly different in the two repositories – we have 2 completely different and independent *switch* statements. Even though they both switch on some discriminant, this may change in the future. Also, consider grouping these two switches together! For 100 animals and 10 employee types you'd have a complete and utter *mess*! There should be a better solution to this. What if we let the concrete repository classes take care of the switching for us? What if we simply pass the current element we're (node) we're at in a method which is implemented only in the subclasses and let them take care of the switching for us? Let's see what I mean by that.

Create the following abstract method in the *EntityRepository*

```
protected abstract T getEntityFromXmlElement(Element element);
```

Instead of the whole switch statement in the *load* method, simply write:

```
...
for (int i = 0; i < nodeList.getLength(); i++) {
	Node node = nodeList.item(i);
	if (node.getNodeType() == Node.ELEMENT_NODE) {
		Element element = (Element) node;
		entities.add(getEntityFromXmlElement(element));
	}
}
```

Right after creating the abstract method, because your 2 repositories already extend this class you will get errors saying that you have to implement this unimplemented method. Do so!
For example, in your *AnimalRepository* class you would have:

```
@Override
protected Animal getEntityFromXmlElement(Element element) {
	String discriminant =
	element.getElementsByTagName(Constants.XML_TAGS.DISCRIMINANT).item(0)
							.getTextContent();
		switch (discriminant) {
		case Constants.Animal.Insect.Butterfly:
			Animal butterfly = new Butterfly();
			butterfly.decodeFromXml(element);
```

```
                return butterfly;
            default:
                break;
            }
            return null;
    }
```

Of course, this just contains one animal however by copy-pasting you should easily place the whole switch block here and instead of having the "*add to list"* statement, you'd have a return. Do this for the EmployeeRepository as well. Now gaze at how lovely your code became. But wait, there's more! Remember the method

```
    public static void createNode(XMLEventWriter eventWriter, String name, String
value) throws XMLStreamException { … }
```

Which I told you to place in the *AnimalRepository*? Well, if you placed it there, you would have had to import it from there in your Employee classes (and subclasses) which is rather messy as it was in the AnimalRepository class, you were dealing with employees, not animals, etc. etc. You can now safely put it in the *EntityRepository* class and import it everywhere from there. Now it makes more logical sense and it's less messy.

Hopefully you've seen the "power" of generics here, even though it may not have been so obvious.