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

5.1 MVC

Let us now shift our focus from the lower levels of our architecture (repositories, services etc.) and take a look at how a user could interact with the application creating a relatively friendly UI. We will henceforth be using the **MVC** (**M**odel – **V**iew – **C**ontroller) architecture as it is widely used in all sort of applications.

We have already covered at least one letter from the above mentioned abbreviation, more precisely, the **M**. We created models for our application which ended up being either some sort of animal or employee. We kind-of had a **C**ontroller however it was a bit off what it is intended to be (as we will see shortly). With the risk of repeating myself, the reason for which software is designed in these kind of *boxes* of sort in most (99.9%) of the cases has nothing to do with what the end user (the person actually playing around with the application) sees. It's all about code readability, scalability and other non-functional requirements. You could (theoretically) write everything in one class/file but if that project is to be handed over to someone else, you're sure to gather enough hexes and curses from that unfortunate someone to last you a lifetime. Think about your code as a house. You have a kitchen in which certain activities are performed and in which you will like to find certain objects. Keeping spoons and forks in the bathroom would be rather strange, right? (At least for those of us who don't have particularly weird activities going on there.) Nevertheless, each room with its purpose and naturally, we like it that way. Naturally, we have this tendency of assigning a place for everything. Well, it's quite the same for software architecture – *everything has its place* and whenever you *eat in bed* instead of doing it in the kitchen, your mother (or in our case your team-lead / architect) will surely not enjoy that.
We are left with two more letters the **V**s and the **C**s.

5.2 Views

For this part we will be using the Java Swing library. Even though it may not be a *state of the art* library, it should suffice for our academic/learning purposes and needs.
Hopefully, by now you understand and know what a *JFrame* as it will be at the core of each of our views. Every view will be a *JFrame* and we'll tread them as *mobile phone* pages/screens, almost each of them having a *back* button as to facilitate navigation. (The terms *page* and *frame* will henceforth be used interchangeably)

5.3 Frame stack and the generic Frame class

In order to have some sort of control over which page gets shown, we need some sort of structure to hold our pages (at least for a while). Even though web pages and well browsers have a back/forward button, phones usually have only a back button and in case you wish to go to the screen you just came from, you'd have to click on the same button you clicked before and so on and so forth. The first thing which comes to mind in this case is a *Stack* where the *pop* operation

removes the last frame and sets the next one (if any) visible and the *push* operation renders the current page hidden and shows a new one which gets inserted at the top of the stack.
First of all, in the *javasmmr.zoowsome.views* package, let us create an interface called *ZooFrame_I* which has just one method.

```
public interface ZooFrame_I {

	public void goBack();

}
```

After that, we need to model a generic Frame. Every other frame in our application will need to extend this one and it will provide some base layout to all the frames.

```
public abstract class ZooFrame extends JFrame implements ZooFrame_I {

	@Override
	public void goBack() {

	}

}
```

As we can see, it extends the classic *JFrame* component from *Swing* and also implements the *ZooFrame_I* interface. Let's leave it like this, for now.

Switching to the *FrameStack* class, we need it to:

- Model a stack of frames with push/pop operations
- Be unique (as in, we cannot have more than one such stack taking care of our frames. Only once instance of this stack can be present in our application)

Surely, a stack is easily modelled using an array list but the second requirement seems maybe more interesting. One *design* pattern however should quickly *pop* (pun-intended) into your head: *singleton*. The next page contains the full implementation of the *FrameStack* class. Please create a separate package for this class called *javasmmr.zoowsome.views.utilities*

Notice the singleton pattern right at the start of the class. Examining the two methods (push and pop) we see that they actually perform exactly what we need them to.

Push:

- If the stack is not empty, peek at the first element of the stack and set its visibility to false
- Set the frame to be pushed to be visible
- Add said frame to the stack at the first position

Pop:

- If the stack is not empty we first dispose of all the resources which the current top frame is using (which also renders it invisible) and then remove it from our container
- If at this stage the stack is not empty, that means we have a page which we can navigate to (the previous one) and we thus render it visible

```
import java.util.ArrayList;
import javasmmr.zoowsome.views.ZooFrame;

public class FrameStack {

	private static FrameStack instance;
	private ArrayList<ZooFrame> stack;

	private FrameStack() {
		stack = new ArrayList<ZooFrame>();
	}

	public static FrameStack getInstance() {
		if (instance == null) {
			instance = new FrameStack();
		}
		return instance;
	}

	public ZooFrame peek() {
		return stack.get(0);
	}

	public void push(ZooFrame frame) {
		if (!stack.isEmpty())
			peek().setVisible(false);
		frame.setVisible(true);
		stack.add(0, frame);
	}

	public void pop() {
		if (!stack.isEmpty()) {
			peek().dispose();
			stack.remove(0);
			if (!stack.isEmpty()) {
				peek().setVisible(true);
			}
		}
	}

}
```

Switching back to our abstract ZooFrame class, I will again try to explain everything directly on the code snippet containing the whole class.

First of all, we see a `protected JPanel contentPanel;`. This will serve as a container for everything we will put on our frames/pages except a small bar at the top of each page which will contain the *back* button.
Our constructor contains just one parameter: the title of the frame.

```java
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javasmmr.zoowsome.services.factories.Constants.Frames;
import javasmmr.zoowsome.views.utilities.FrameStack;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;

public abstract class ZooFrame extends JFrame implements ZooFrame_I {

	protected JPanel contentPanel;


	public ZooFrame(String title) {
		FrameStack.getInstance().push(this);
		setTitle(title);
		setSize(FRAMES.WIDTH, FRAMES.HEIGHT);
		setLayout(new BorderLayout());
		setLocationRelativeTo(null);
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		contentPanel = new JPanel();
		contentPanel.setBorder(new EmptyBorder(5, 5, 5, 5));
		contentPanel.setBackground(Color.red);
		add(contentPanel, BorderLayout.CENTER);
	}

	public void setBackButtonActionListener(ActionListener a) {
		JPanel buttonPanel = new JPanel();
		buttonPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
		JButton backButton = new JButton("Back");
		buttonPanel.add(backButton);
		this.add(buttonPanel, BorderLayout.NORTH);
		backButton.addActionListener(a);
	}
}
```

In our constructor, we first push `this` (the current) frame to our stack, set the title and give the frame some dimensions. Notice that because I statically imported the *Frames* class from the *Constants* class, I was able to simply use `Frames.WIDTH` and `Frames.HEIGHT` instead of the writing something like `Constants.Frames.HEIGHT`. (Maybe not a big improvement in our case however in other situations, this static import may be useful). The default value for these two values is:

```java
public static final int HEIGHT = 500;
public static final int WIDTH = 700;
```

(Feel free to play around with the values)

Next we set the layout, add the back button and center the frame in the screen (`setLocationRelativeTo(null);`).

`setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);` simply forces the application to fully close without leaving any uncollected memory leaks when pressing the "X" button.
Last but not least, we initialize the contentPanel and give it the background color red jut for testing purposes.
The `setBackButtonActionListener` method will be used inside our controllers as we will see next.

5.4 Controllers

So. Many. Abstract. Things. And there's going to be one more still.

5.4.1 The abstract controller

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javasmmr.zoowsome.views.ZooFrame;
import javasmmr.zoowsome.views.utilities.FrameStack;

public class AbstractController {

	protected ZooFrame frame;

	public AbstractController(ZooFrame frame, boolean hasBackButton) {
		this.frame = frame;
		if (hasBackButton)
			frame.setBackButtonActionListener(new BackButtonListener());
	}

	private class BackButtonListener implements ActionListener {

		@Override
		public void actionPerformed(ActionEvent e) {
			FrameStack.getInstance().pop();
		}

	}
}
```

Each subsequent controller will have to inherit from this class. Each subclass will pass a frame to this controller and *whether or not* that frame has a back button. If it does have a back button, this abstract controller will call the `setBackButtonActionListener` method which will place a back button and set a listener to it.

5.5 Concrete stuff

Finally, we're reaching some concrete classes. Let's check out two very simple controllers and frames.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javasmmr.zoowsome.views.AddFrame;
import javasmmr.zoowsome.views.MainMenuFrame;

public class MainMenuController extends AbstractController {

	public MainMenuController(MainMenuFrame frame, boolean hasBackButton) {
		super(frame, hasBackButton);
		frame.setAddButtonActionListener(new AddButtonActionListener());
	}

	private class AddButtonActionListener implements ActionListener {

		@Override
		public void actionPerformed(ActionEvent e) {
			new AddController(new AddFrame("Add"), true);
		}

	}

}
```

The MainMenuController will be the first one to be created by the app. Notice how it also has only one constructor. It is no point in having a no-argument constructor. Why? Well, there is a 1 to 1 relation between a controller and a view/frame. A view cannot exist without a controller and there is no point in a controller existing without a view.
This controller is tightly linked to its associated frame, the *MainMenuFrame*. (See next page)

```java
import java.awt.GridLayout;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.SpringLayout;

public class MainMenuFrame extends ZooFrame {

	private JButton btnAdd;
	private JButton btnList;
	private JButton btnSaveAndExit;

	public MainMenuFrame(String title) {
		super(title);
		contentPanel.setLayout(new GridLayout(0, 3, 0, 0));

		JPanel panel = new JPanel();
		// adding empty panel to fill grid layout
		contentPanel.add(panel);

		JPanel pan = new JPanel();
		contentPanel.add(pan);
		SpringLayout slPanel = new SpringLayout();
		pan.setLayout(slPanel);

		btnAdd = new JButton("Add");
		slPanel.putConstraint(SpringLayout.NORTH, btnAdd, 65, SpringLayout.NORTH, pan);
		slPanel.putConstraint(SpringLayout.WEST, btnAdd, 93, SpringLayout.WEST, pan);
		pan.add(btnAdd);

		btnList = new JButton("List");
		slPanel.putConstraint(SpringLayout.NORTH, btnList, 150, SpringLayout.NORTH, pan);
		slPanel.putConstraint(SpringLayout.WEST, btnList, 94, SpringLayout.WEST, pan);
		pan.add(btnList);

		btnSaveAndExit = new JButton("Save and Exit");
		slPanel.putConstraint(SpringLayout.NORTH, btnSaveAndExit, 264, SpringLayout.NORTH, pan);
		slPanel.putConstraint(SpringLayout.WEST, btnSaveAndExit, 69, SpringLayout.WEST, pan);
		pan.add(btnSaveAndExit);

		JPanel panel_2 = new JPanel();
		contentPanel.add(panel_2);
		setVisible(true);	}

	public void setAddButtonActionListener(ActionListener a) {
		btnAdd.addActionListener(a);
	}

	public void setListButtonActionListener(ActionListener a) {
		btnList.addActionListener(a);
	}

	public void setSaveAndExitButtonActionListener(ActionListener a) {
		btnSaveAndExit.addActionListener(a);
	}

}
```

The other controller/frame pair contains a completely empty page with no components whatsoever.

```
public class AddController extends AbstractController {

	public AddController(AddFrame addFrame, boolean hasBackButton) {
		super(addFrame, hasBackButton);
	}
}
```

```
public class AddFrame extends ZooFrame {

	public AddFrame(String title) {
		super(title);
	}

}
```

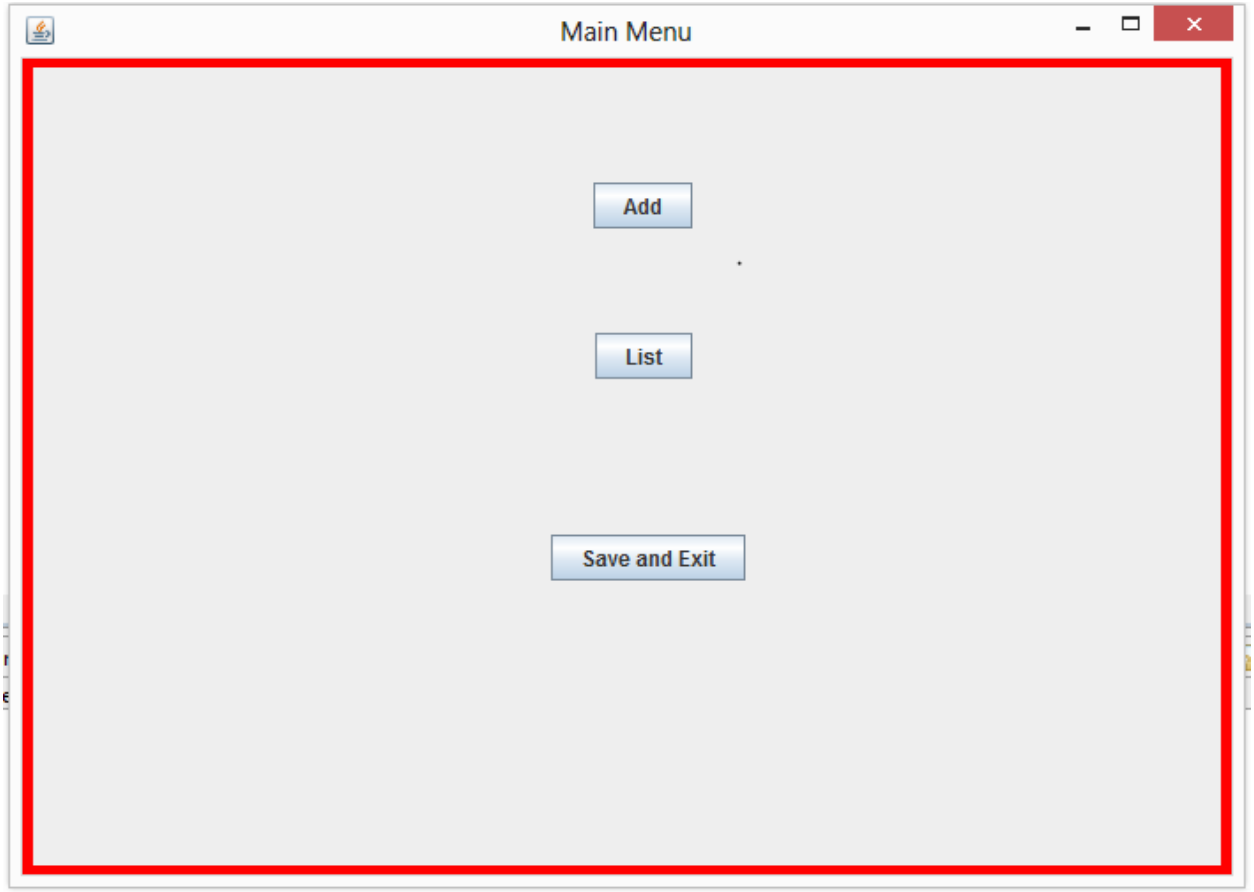
The new main method of the application should now look like:

```
import javasmmr.zoowsome.views.MainMenuFrame;

public class Main {
	public static void main(String[] args) throws Exception {

		new MainMenuController(new MainMenuFrame("Main Menu"), false);

	}
}
```

Create this Main class and place it in the *controllers* package.

You may be wondering why we are not saving these in variables. Well, the idea is that we should never ever use the Controller object nor the View object separately. They form a bond together and should not be manipulated in any other way. We could save them in objects but since the initialization code is directly written in the constructor, there is absolutely no point in doing so. There may be other more elaborate ways of binding these two together, however, these approaches do not fall in the scope of this course.

At this stage the application should look something like this:



And by pressing the "Add" button, the menu frame should disappear and an empty one with just the back button should pop-out.

5.6. Requirements

- Using this architecture, construct views and controllers, creating forms for adding any type of animal or employee and listing all animals and employees with all their attributes. Use your imagination for this! You can build it in any way you want to, however, make sure that you can create specific animal types and that it's somehow intuitive to do.
- Don't forget about the layered architecture. Your controllers should not do much in terms of business logic but pass that responsibility to services / repositories

Ps: this in itself is not a terribly easy task so don't discourage.

Twist:

- By using the Java Reflection API and a predefined template for a form, automatically generate the non-abstract forms for any animal. This basically means that you should

provide one controller and one view for all the forms and they should somehow create all the necessary fields on the front-end (view) in order to generate any form for any animal given only the type of the animal to be created. If you don't understand this, ***ask questions on the forum.***

- Whoever successfully completes this requirement and will be a student of mine next semester will receive from the get-go 2 points on the final mark at the lab (and maybe chocolate or candy or something)