1. Thinking back to the article we read about Singletons in Java, what was the special data type way to do a singleton in Java? The article said the main drawback was that it does not allow for lazy initialization. Could you do Singletons that same data type way in C++? If so, give a small working example (should be 2-3 brief files). If not, why? How if in any way, is that data structure/data type different in C++ than Java?

The special data type is the enum. The best way for Java to implement the Singleton Pattern is to use the enum data type because this guarantees that there will only ever be one instance of the class. This is because enums are inherently serializable. Lazy initialization can be implemented with an enum class, but the drawback is that it makes it globally accessible. This nullifies the singleton pattern as now multiple threads can instantiate multiple copies of the class. In a multi-threaded environment, there is no guarantee that an enum class will be instantiated once and only once.

In Java, the enum is a class that extends the Enumeration class. In C++, the Enum is just an alias for an integer. They are two vastly different objects. The beauty of Java is that you can use an enum just like a class. There is one exception, however, the enum class cannot be inherited from. The latest releases of C++ have released an Enum Class, but it is not as powerful or flexible as its Java sibling.

Here are several examples of a C++ enum class…

```
// C++ Enum class example

#include <iostream>
using namespace std;

int main()
{

    enum class Insect { Beetle,
                        Horsefly,
                        PreyingMantis };

    enum class Arachnid { Spider,
                          Scorpian };

    enum class Pet { Dog,
                                Goldfish,
                     Cat };

    int Beetle = 10;
    int Horsefly = 20;
    int PreyingMantis = 30

    // Object of the Enum Class
    Insect i = Insect::Beetle;

    // Comparison operation
    if (i == Insect::PreyingMantis)
        cout << "It's PreyingMantis\n";
    else
        cout << "It's not PreyingMantis\n";

    // another instantiation
    Pet p = Pet::Goldfish;

    return 0;
}
```

Another example:

Season.h

```
#include <iostream>
enum Season
{
    Summer,
    Spring,
    Winter,
    Autumn
};
enum Color{
    Blue,
    Pink,
    Green
};
```

Main.cpp

```
int main()
{
    Season s = Summer;
    Color c = Blue;
    if( s == c )
    {
        std::cout << "Equal" << std::endl;
    }
    else
    {
        std::cout << "Not Equal" << std::endl;
    }
    return 0;
}
```

Pulled from: https://www.codesdope.com/cpp-enum-class/

2. Which pattern(s) from class would be best for a rental car system from which clients could order multiple types of cars and add different features to a specific order? Certain car types already exist, but new cars with new features are always being added and we do not want to modify the ordering interface.

This sounds very much like our pizza construction application. I would use the Decorator Pattern, although other patterns could be applied as well. I could see implementing this with the Composite pattern as well. Each new feature would be a component class. But let's return to the Decorator pattern. I would first define an abstract class called BasicCar. Then I would add each additional feature as a concrete class just as we did with the pizza toppings. It might be advantageous to categorize various features into additional abstract classes in a hierarchical structure. The Decorator pattern allows you to embellish a basic object. This is what a typical rental car application does. It provides you with a basic car then allows you to add features to it. This is the intent of the Decorator pattern.

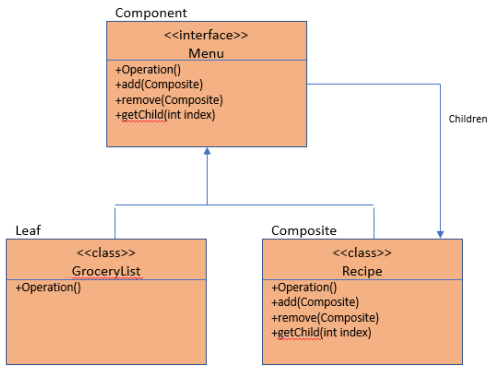
Another pattern that might work is the Composite Pattern. The base vehicle would be at the top of the hierarchy (the tree), and the leaves would represent the various finished completed rental cars. Another pattern we have not discussed that might fit is the Builder Pattern. The Builder object would represent the class that constructs the final rental car. The components would be the additional rental car features.

The Façade Pattern might work as well. The underlying interface would be a basic vehicle, a car, truck or bus. The façade class would be a RentalCarBuilder class. It would assemble the various final vehicles with each of the desired features.

The Factory Pattern could also be used. The factory class would be the basic vehicle. Each finished rental car could extend form the basic factory class. The factory class would provide a recipe for how to build the customer chosen vehicles. The factory class could be abstract thereby forcing the implementation of the construction methods needed to produce the final vehicle.

3. Which pattern(s) from class would be used for an Android phone application that has several pages for the user to interact with. Each of those pages can change and read from a collection of recipes loaded into memory or downloaded from the internet to memory. Recipes can be combined into a menu, and the menu can be dumped into a grocery list so that the user can know in one page everything they need to buy to make multiple recipes for one meal.

A document typically lends itself to a hierarchical relationship. The document is made up of paragraphs. Paragraphs are made up of words. Words of letters. This recipe arrangement is very similar to the document model. It lends itself to a hierarchical arrangement. That being the case, I would opt for the Composite Pattern for this situation. The Composite Pattern arranges objects in a hierarchical structure. In this case, the menu would be the root of the tree and the recipes and grocery list would be the leaves.



4. **What is the difference between the Decorator and Adapter pattern? How are they similar? What is the advantage of one over the other?**

The Decorator and the Adapter both deal with the composition of objects. They are both categorized as structural patterns. In fact, they are so similar that they are sometimes considered to be the same. However, their purposes are vastly different. The purpose of the Decorator pattern to embellish a basic object. You start with a base object then add features to it. The Adapter also deals with components, but rather than embellishing them; it attempts to make disparate objects work together that would normally have nothing to do

with each other. The Adapter pattern **converts** an interface. The Decorator simply **extends** an abstract class adding features to it.

5. **What does making an abstract class do? What does it protect a user of your code from accidentally doing? What is an interface? How are interfaces different in C++, Java, and Python?**

An abstract class is a class that is marked with the 'abstract' keyword. It can also contain abstract methods. An abstract class cannot be instantiated directly. This guarantees that the class will be used as the developer intended. This also guarantees that the class will not accidentally be instantiated. This protects the code from misuse. Abstract classes are extended from. Interfaces are implemented.

An interface is a syntactical entity that provides a blueprint for a behavior. It may include static data attributes and abstract function prototypes. An interface is implemented by a subsequent class. Interfaces are used to implement abstraction and multiple inheritance. Java has an interface structure built into the language, C++ and Python do not. The last statement is not entirely true. Later versions of Python have introduced a separately installed interface add-on to the language.

A class with all of its functions declared as 'virtual' in C++ is the equivalent of a named interface in Java. Java has a convenient 'interface' keyword. The virtual keyword in each function of a class in C++ makes an equivalent interface. Since Python permits multiple inheritance by default, there is no need for a separate interface structure; however, a special interface module can be installed into Python, which provides Java-like interface keywords. Python uses methods with empty implementations to simulate the concept of an interface.

6. I have code in an old version of Java. It uses an interface that new Java no longer uses. I do not want to rewrite all of this code to support the new Java interface. How can I call my functions from my old code with new Java code by adding some pattern classes and not touching the old code at all?

Perhaps the adapter pattern could be used. The adapter pattern converts the original interface behavior to entirely different behavior. It could possibly make the old behavior act like the new. The Command Pattern also might be a candidate. The old Java code (the original interface or abstract class) would be encapsulated into an object and the command would contain the enhanced (newer) code and features. The original encapsulated object could remain unchanged. As each new feature is added, a new command is simply plugged in and activated – just like adding new features on to a TV remote.

With the decorator pattern the original from Java interface and concreate classes could be superseded by classes that utilize the new features of Java. With each new release of Java, a new decorator class could be introduced that implemented the new java functions.

7. What is encapsulation? What is invocation? Which pattern allows me to encapsulate invocation? Do any other patterns aid in this?

Encapsulation is the process of taking everything that is required to complete a task and wrapping (hiding) it inside a class or object. This includes all of its data and its methods. The invoking process knows nothing about the contents of the requested object. Invocation is the process of activating the encapsulated object. In Java, objects are typically fired off with a 'new' keyword.

The Command Pattern is the pattern that encapsulates all of the components needed to perform an action. This action or event is triggered via a command.

8. I have a security system with multiple sensors. If someone breaks the laser beams, I need to spray gas at them to knock them out. If they break the glass around my diamonds, then I need a cage to fall around them. If they step on the plates outside of the cage after it has fallen, I need to release my pet lions, but if the smell sensor smells cordite I need to not release the lions because I do not want them to get hurt, instead, I want to release my robot vampire bats. Decide on a pattern to employ for this, and draw a picture of how it would work with everything. Label each part of your diagram with the corresponding part from the description and the terminology of your pattern to which those parts correspond.

The Observer Pattern would be ideal for this. This is very similar to the weather station example introduced in class. See diagram following code. When I saw the word sensor, I immediately thought Observer. The sensors are the Observers. They wait for a specific condition to occur. When It occurs, they send the condition to the Subject who will cause an action to occur. The action is executed against the intruder.

It would kind of be naïve to think that the Observer pattern was the one and only pattern that applied here. The Facade Pattern might also fit in here. The Façade pattern tries to simplify complex or convoluted logic. The Façade class sorts out the logic of this problem and presents it to the resulting action classes.

Then there is the Command Pattern. The Command pattern captures actions. Each of these actions in this scenario could be encapsulated into separate commands - "gas sprayed", "cage falls", and "bats released". These are the implementation of the interface's execute() method.

Most of the following code is cookie-cutter code for the Observer Pattern. It was taken from: https://dzone.com/articles/behavioral-design-patterns-observer. I added the code in the main method.

```
public interface Observer {
    void update(SensorData sensorData);
}

public interface Observable {
    void register(Observer observer);
    void unregister(Observer observer);
    void updateObservers();
}
```

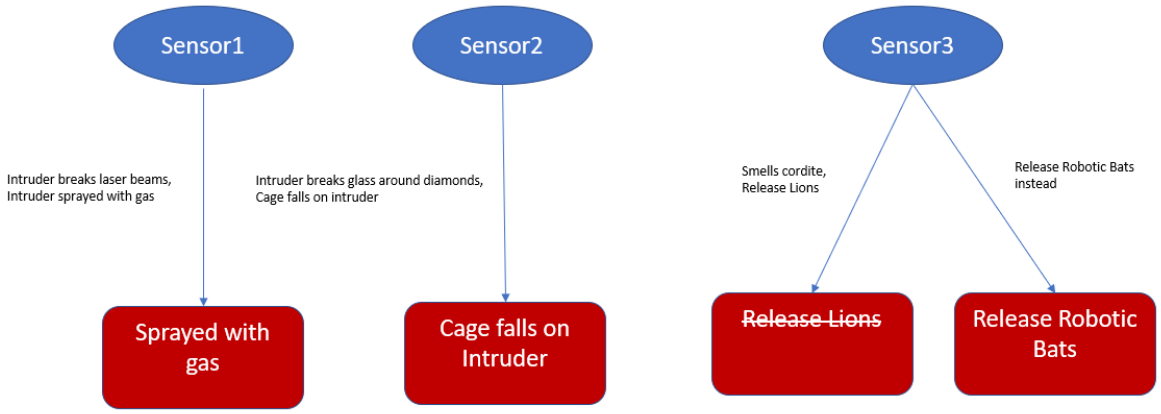
```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class SensorReceiver implements Observable {
    private List data = new ArrayList();
    private List observers = new ArrayList();
    @Override
    public void register(Observer observer) {
        observers.add(observer);
    }
    @Override
    public void unregister(Observer observer) {
        observers.remove(observer);
    }
    public void addData(SensorData sensorData) {
        data.add(sensorData);
    }
    @Override
    public void updateObservers() {
        /**
         * The sensor receiver has retrieved some sensor data and thus it will notify the
observer
         * on the data it accumulated.
         */
        Iterator iterator = data.iterator();
        while (iterator.hasNext()) {
            SensorData sensorData = iterator.next();
            for(Observer observer:observers) {
                observer.update(sensorData);
            }
            iterator.remove();
        }
    }
}

public class SensorLogger implements Observer {
    @Override
    public void update(SensorData sensorData) {

        System.out.println(String.format("Received sensor data %s:
%f",sensorData.getSensor(),sensorData.getMeasure()));
    }
}

public class SensorMain {
    public static void main(String[] args) {
        SensorReceiver sensorReceiver = new SensorReceiver();
        SensorLogger sensorLogger = new SensorLogger();
        sensorReceiver.register(sensorLogger);
        sensorReceiver.addData(new SensorData("Intruder breaks laserbeam","Sprayed with Gas"));
        sensorReceiver.addData(new SensorData("Intruder breaks glass around diamonds","Cage falls
on intruder"));
        sensorReceiver.addData(new SensorData("Smells cordite,","Do not Release Lions"));
        sensorReceiver.addData(new SensorData("Smells cordite,","Release Robotic Bats "));
        sensorReceiver.updateObservers();
}
```

Question #8: Security System – Observer Pattern



9. **Which other pattern(s) from class could we have used for accomplishing the functionality of decorating pizzas if we had never heard of the decorator pattern and books were outlawed in our area?**

If we are restricted to design patterns we have covered, I would say the Adapter pattern is the closest to the Decorator pattern in functionality. There is also a Builder Pattern. We have not talked about it in class. I don't think that we are even going to cover it in remainder of the class. The Builder pattern lends itself quite nicely to the composition of objects – like the building of various pizzas.

10. **How does the open closed principle apply to the Command Pattern?**

The open closed principle states that software should be open to enhancement or extension, but closed to modifications.

The prototype for the Command Pattern was the TV remote. Each button represents a discrete function. When you wish to add a new function or feature, simply add another button to the TV Remote. The TV Remote is the Interface or Abstract class in the Command Pattern. Each button on the remote represents a new command or class in the Command Pattern. How does this affect the open close principle? New enhancements can be added without changing the underlying implementation. When a new feature or function is called for, simply add a new command.