Talon Knowlton

Kai Gonzalez

June 15, 2022

CSCI 4448 Project 2 Part 1

1. Main differences between Java interfaces and traditional OO interface design:
    a. Modern Java has default methods for interfaces. A default method is a method defined in an interface that specifies an actual implementation that can be overridden with custom code.
    b. Modern Java allows interfaces to specify static methods. Static methods can be used as factory methods.
    c. Modern Java allows interfaces to specify private methods. Private methods are useful as helper methods for the default methods specified in the interface.

```
package defaultmethods;
import java.time.*;
public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year, int hour, int minute, int second);
    // a private helper method for default method
    public LocalDateTime getLocalDateTime() {
        return dateAndTime;
    }
    // a static method for use in default method
    static ZoneId getZoneId (String zoneString) {
        try {
            return ZoneId.of(zoneString);
        } catch (DateTimeException e) {
            System.err.println("Invalid time zone: " + zoneString +
                "; using default time zone instead.");
            return ZoneId.systemDefault();
        }
    }
    // a default method
    default ZonedDateTime getZonedDateTime(String zoneString) {
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
    }
}
```

source: https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

2. Abstraction and encapsulation are both one of the four main principles of object oriented programming (the other two being inheritance and polymorphism). They both are used with the aim of creating code that is object oriented where you have multiple different pieces of code that each focus on their own individual tasks and communicate with each other. The difference between the two is that encapsulation is about *data hiding* whereas abstraction is about *implementation hiding*.

   In abstraction you are only showing the essential details and hiding everything else. The classic example is operating a car; you know how to interact and use the car, but that doesn't mean you need to know how it mechanically works. You can steer the car using the steering wheel and accelerate by using the gas pedal without knowledge of how those systems work. There is this abstract view that if you interact with the car in a certain way you will get some kind of known output.

   The same thing happens in programming where one piece of code will interact with other pieces of code through an interface without knowledge of the algorithms and implementations used by the other code. For example in the Analyzer class below we have many function calls to different math functions. When we call for example the sum() function we don't necessarily know how it works. All we know is that if we call the function and give it a single parameter of type ArrayList<Double>, we will get an output of the sum of the numbers in the arraylist. We may have an idea of how the algorithm in the sum() function works, but it's not necessary for us to know in order to use it.

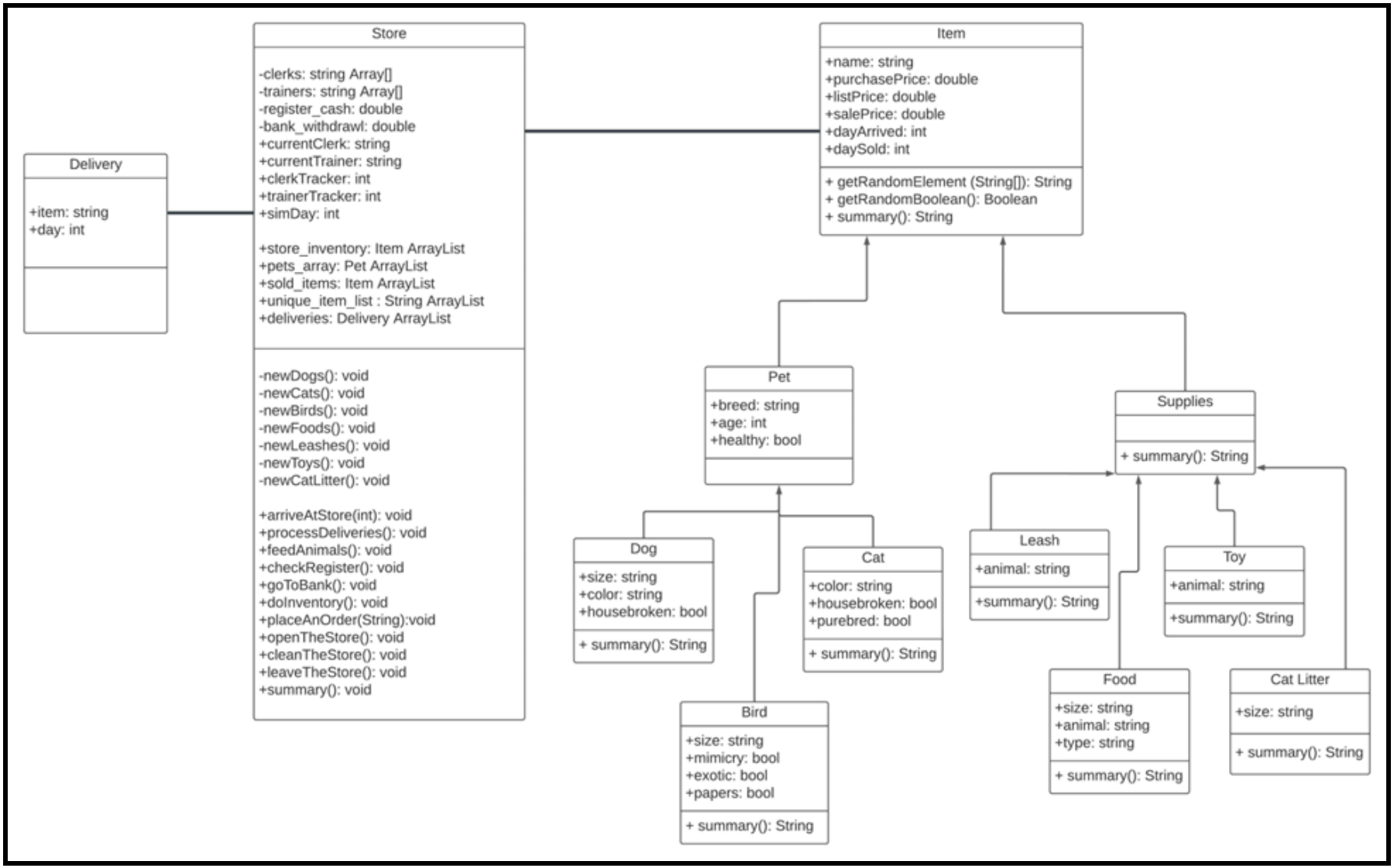
```java
class Analyzer{

    public void analyze(ArrayList<Double> mylist)
    {
        mylist.sort(Comparator.naturalOrder());//sort Array

        //call all functions
        double list_sum=(sum(mylist));
        double median=(median(mylist));
        double mean=(mean(mylist));
        double minimum=(minimum(mylist));
        double maximum=(maximum(mylist));
        double standard_deviation=(standard_deviation(mylist));

        //output the resulting calculations
        System.out.println("The list sum is " + list_sum);
        System.out.println("The list median is " + median);
        System.out.print(s: "The list mean is ");
        System.out.format(format: "%.3f",  mean);
        System.out.println(x: " ");
        System.out.print(s: "The list sample standard deviation is ");
        System.out.format(format: "%.3f", standard_deviation);
        System.out.println(x: " ");
        System.out.println("The list minimum is " + minimum);
        System.out.println("The list maximum is " + maximum);
        System.out.println(x: " ");

    }
```

In encapsulation you are instead hiding information within a class or certain areas of code with the goal that no other code can interact with it without explicit permission. You are essentially “encapsulating” data and protecting it from any outside attempts to modify it. This is done by using access modifiers such as private or protected which limit the access that other pieces of code can have within a class. In the example below we have two classes within a java file. We have the Scratch class which has the main function where the code starts and the Hidden class which has hidden information within a private integer. The “hidden_value” variable is protected from all outside attempts to change or view its value due to the fact that its access modifier is declared as private. The only way that this value can change or be viewed is from within the class. In this example this is achieved with a getter and setter function which give

explicit access to the hidden_value variable. This variable is completely encapsulated within its class and has complete control over who gets to view or change it.

```
class hidden
{
    private int hidden_value;

    public int get_hidden()
    {
        return hidden_value;
    }

    public void set_hidden(int hidden_value)
    {
        this.hidden_value = hidden_value;
    }


}

public class Scratch{
    Run | Debug
    public static void main(String[] args)
    {
        hidden s1 = new hidden();

        s1.set_hidden(hidden_value: 20);

        System.out.println("The hidden value is: " + s1.get_hidden());
    }
 }
```

3. Project UML Diagram


Store
-clerks: string Array[]
-trainers: string Array[]
-register_cash: double
-bank_withdrawl: double
+currentClerk: string
+currentTrainer: string
+clerkTracker: int
+trainerTracker: int
+simDay: int
+store_inventory: Item ArrayList
+pets_array: Pet ArrayList
+sold_items: Item ArrayList
+unique_item_list : String ArrayList
+deliveries: Delivery ArrayList
-newDogs(): void
-newCats(): void
-newBirds(): void
-newFoods(): void
-newLeashes(): void
-newToys(): void
-newCatLitter(): void
+arriveAtStore(int): void
+processDeliveries(): void
+feedAnimals(): void
+checkRegister(): void
+goToBank(): void
+doInventory(): void
+placeAnOrder(String):void
+openTheStore(): void
+cleanTheStore(): void
+leaveTheStore(): void
+summary(): void
Item
+name: string
+purchasePrice: double
+listPrice: double
+salePrice: double
+dayArrived: int
+daySold: int
+ getRandomElement (String[]): String
+ getRandomBoolean(): Boolean
+ summary(): String
Delivery
+item: string
+day: int
Pet
+breed: string
+age: int
+healthy: bool
Supplies
+ summary(): String
Dog
+size: string
+color: string
+housebroken: bool
+ summary(): String
Cat
+color: string
+housebroken: bool
+purebred: bool
+ summary(): String
Bird
+size: string
+mimicry: bool
+exotic: bool
+papers: bool
+ summary(): String
Leash
+animal: string
+summary(): String
Toy
+animal: string
+summary(): String
Food
+size: string
+animal: string
+type: string
+ summary(): String
Cat Litter
+size: string
+ summary(): String