CSCE 156 - Lab 06: Polymorphism

Handout

0. Prior to the Laboratory

1. Review the laboratory handout.
2. Read the following chapters sections from Data Structure and Abstractions with Java

3.6.1 The this Reference
3.6.3 The instanceof Operator
3.4.1 Constructors
3.4.2 Mutators and Accessors
4.1.1 Creating New Classes
4.1.3 Dynamic Dispatch and Polymorphism
4.1.8 Overriding a Method

The chapters in red cover new material which is the focus of this lab; the chapters in black were the focus of lab 4 but are essential to simplifying this lab.

Importing Your Project

A Java project has been started to implement a basic payroll system. Import this project into your Eclipse IDE by following either of the steps below:

From the GIT repository:

Go to File-> Import->Git-> Projects from Git.

(Note: If the "Git" import source is not listed, you might have to install a Git plugin for Eclipse)

1. Select "Clone URI" and enter the repository URI as the following

   https://git.unl.edu/csce-156/Lab06

1. Lab Objectives & Topics

Upon completion of this lab you should be able to:

- Use inheritance, composition (aggregation), simple polymorphism
- Planning, evaluating, and selecting different design strategies to efficiently and effectively solve problems

2. Problem Statement

Your company is responsible for designing billing software for a parking garage named SafePark. SafePark has a capacity of 20 vehicles and can service 3 types of vehicles; motorbikes, compact cars, and SUVs. Parking fees are based on the following table:

	One week or less	More than one week
Motorbike	$4.00 per day	$3.00 per day
Compact Car	$6.00 per day	$4.50 per day
SUV	$8.00 per day	$6.00 per day

For example, a Motorbike that leaves on the 8th day would pay $24 while an SUV that leaves on the 4th day would pay $32. Days begin on the day that the vehicle enters the garage.

To get you started, you have been provided with a framework with which to simulate the SafePark parking garage. Specifically, you have been provided the following partially completed classes:

- Vehicle - This class represents a generic vehicle, identified by a unique license plate (represented as a string).
- Garage - This class represents the SafePark garage which "parks" vehicles by storing them in an array of Vehicle objects.
- GarageSimulation - This class is where you will simulate the operation of the SafePark garage by creating vehicle objects, "parking" them and removing them from the garage and producing reports.

Activities

Familiarize yourself with the given code and run the GarageSimulation program. You'll need to complete several methods in each class to get the SafePark simulation working properly. *However,* how these methods will get implemented will depend greatly on some design decisions that you need to make up-front. The first thing to do is to define several subclasses of the Vehicle class.

Activity 1: Sub-classing & Simulating

Create three subclasses of the Vehicle class to support the three vehicle types described above: `Motorbike`, `CompactCar`, and `SUV`. Observe that since these are subclasses, Java requires you to define non-default constructors which (should) call super constructors. Modify the code in the `GarageSimulation` class to simulate the following parking schedule. Note: additional work will need to be done to make it fully functional. You should print a report each Saturday to answer the questions in your handout.

	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Week 1	Compact Car "ABC 123" enters SUV, "QED NEB" enters		Motorbike "XYZ 321" enters	Motorbike "QT 42" enters Compact Car "FOO 459" enters		"QT 42" leaves	
Week 2	"ABC 123" leaves			Compact Car "BAR 560" enters			
Week 3	"QED NEB" leaves		SUV, "CSE 444" enters Motorbike "QT 42" enters				

Activity 2: Class Design

The SafePark simulation needs to be finished by adding functionality to do the following.

1. Properly add (park) and remove cars from the garage
2. Keep track of number of days each vehicle has been in the garage (incremented for all parked vehicles by invoking one of the `addDay()` methods in the `Garage` class).
3. Properly compute the total fee for each vehicle
4. Produce a report on the garage's current state by printing information related to each parked vehicle (stall number, license plate, vehicle type, number of days in the garage, total current fee)

You will need to add functionality (state and behavior) to the Garage class and, depending on your design decisions, the Vehicle class and its three subclasses in order to support these features. Take a

moment to think about the problem and sketch out a design. While you do, think about how you should use Object Oriented Design to *model* the SafePark parking garage. In particular, think about the following:

- How can you determine the "type" of vehicle? Where is it important to distinguish the type of Vehicle and where is it not
- What's responsible for keeping track of the number of days a car has been in the garage?
- What's responsible for computing the total fee?

Design Paradigms

To further guide you in your design decisions, there are several competing paradigms that you may consider that essentially guide the answers to the questions above.

1. Days and fees are a function of the Garage—If your design is guided by this idea, then the state and logic responsible for keeping track of days and fees should be encapsulated in the `Garage` class.
2. Days and fees are a function of an individual vehicle—If your design is guided by this idea, then the state and logic responsible for keeping track of days and fees should be encapsulated in the `Vehicle` classes (and perhaps accessed/used in the Garage class).
3. You may be guided by a third alternative in which one entity is responsible for some things while another is responsible for others. This is perfectly fine, but whatever your decision, your code should model it correctly.

Keep in mind, there is really no *wrong* answer—your design could be influenced by a desire to more closely model the real-world problem or it could be motivated by conveniences in the context of OOP/Java (making it more in-line with accepted OOP practices). However, whatever your design decision, you should be able to *advocate* for it—argue that its advantages outweigh its disadvantages and/or that it is better than an alternative design paradigm.

Further Tools

Java provides several "meta-programming" tools that you may find helpful in your program implementation.

1. `instanceof` keyword - this built-in operator is used in conjunction with a variable and a class and evaluates to true if the variable is an instance of the given class (satisfies the *is-a* relationship, either through sub-classing or by implementing an `interface`). A full example of its usage:

```
if (rustyOldCar instanceof SUV) {

// rustyOldCar must be an instance of SUV } else if

            (rustyOldCar instanceof CompactCar) {

      // rustyOldCar must be an instance of CompactCar
```