# Polymorphism - Individual Exercise

The purpose of this exercise is to provide you the opportunity to practice writing code using the principle of polymorphism.

## Learning Objectives

After completing this exercise, students will understand:

- How to design and create super classes.
- How to implement sub classes.
- How to use classes polymorphically.

## Evaluation Criteria & Functional Requirements

- The project must not have any build errors.
- Appropriate variable names and data types are being used.
- Code is presented in a clean, organized format.
- The principle of polymorphism is being applied appropriately.
- The code meets the specifications defined below.
- The output for each program matches what is defined in the requirements.

In order to practice applying polymorphism, your task is to create class implementations for the exercises defined below. In any of the cases, you may add attributes (i.e. properties) and other supporting methods to the classes in order to fully implement them.

#### Notes for All Classes

- X in the set column indicates it **should have a `set` accessor**.
- Nothing in the set column indicates the property is a derived property.
- Readonly properties do not require a `set` accessor.

### Toll Booth Calculator

#### Vehicle Class

| Method | Return Type |
|---|---|
| `calculateToll(int distance)` | `double` |

| Attribute | Type | Get | Set | Description |
|---|---|---|---|---|
| `type` | string | X | | The name of the vehicle type (ex. Car) |

#### Car

Create a `Car` class that is a subclass of `Vehicle`. Additionally it has the following attributes and constructors.

| Attribute | Type | Get | Set |
|---|---|---|---|

| Attribute | Type | Get | Set |
|---|---|---|---|
| hasTrailer | boolean | X | |

| Constructor | Description |
|---|---|
| `Car(boolean hasTrailer)` | Creates a new car indicating whether or not it is pulling a trailer. |

Tolls for cars are based upon distance.

```
toll = distance * 0.020
if pulling a trailer then toll = toll + 1.00
```

#### Truck

Create a `Truck` class is a subclass of `Vehicle`. Additionally it has the following attributes and constructors.

| Attribute | Type | Get | Set |
|---|---|---|---|
| numberOfAxles | int | X | |

| Constructor | Description |
|---|---|
| `Truck(int numberOfAxles)` | Creates a new truck indicating how many axles it has. |

Tolls for trucks are based upon the number of axles.

> | Axles | Per Mile |
> |---|---|
> | 4 | 0.040 |
> | 6 | 0.045 |
> | 8+ | 0.048 |
>
> toll = rate per mile * distance

#### Tank

Create a `Tank` class that is a subclass of `Vehicle`.

All military vehicles travel free on the toll roads.

```
toll = 0
```

#### TollCalculator.java

Create a command line program named TollCalculator. The application will not accept any input.

Following the approach discussed in the lecture, create a `List<Vehicle>` that represents all of the vehicles that travel through a particular tollbooth. Using a random number for distance (10 to 240) calculate the tolls for each vehicle so that you can:

- Indicate each vehicle type, the distance traveled, and the calculated toll
- Indicate the sum of all miles traveled and total tollbooth revenue
- You should not need to check the type of Vehicle in the main method when displaying the output

*Expected output using **random** distances*

```
Vehicle             Distance Traveled       Toll $
------------------------------------------------
Car                 100                     $2.00
Car (with trailer)  75                      $2.50
Tank                240                     $0.00
Truck (4 axels)     150                     $6.75
Truck (6 axels)     101                     $4.55
Truck (8 axels)     180                     $8.64

Total Miles Traveled: 846
Total Tollbooth Revenue: $24.44
```

### Postage Calculator

#### DeliveryDriver class

| Method | Return Type | Description |
|---|---|---|
| `calculateRate(int distance, double weight)` | `double` | calculateRate takes the distance being traveled (in miles) and the weight in ounces to calculate the rate. |

| Attribute | Type | Get | Set | Description |
|---|---|---|---|---|
| `name` | string | X | | The name of the service type (ex. 1st Class) |

#### Postal Service

Create a PostalService class that is a subclass of DeliveryDriver. The Postal Service deals with pounds and ounces.

The rate is calculated based on the following rate table:

```
|--------|-----------|-----------|-----------|
| Weight |           |           |           |
| Not    | 1st Class | 2nd Class | 3rd Class |
| Over   | Per Mile  | Per Mile  | Per Mile  |
|--------|-----------|-----------|-----------|
|  2 oz. |   0.035   |   0.0035  |   0.0020  |
|  8 oz. |   0.040   |   0.0040  |   0.0022  |
| 15 oz. |   0.047   |   0.0047  |   0.0024  |
|  3 lbs.|   0.195   |   0.0195  |   0.0150  |
|  8 lbs.|   0.450   |   0.0450  |   0.0160  |
|--------|-----------|-----------|-----------|
| Over   |           |           |           |
| 8 lbs. |   0.500   |   0.0500  |   0.0170  |
|--------|-----------|-----------|-----------|

rate = per mile rate * distance
```

**HINT** Consider how multiple classes per delivery type (1st class, 2nd class, 3rd class) could help. How might your code be impacted if a new delivery type is added?

**FexEd**

Create a FexEd class that is a subclass of DeliveryDriver.

FexEd charges a flat rate for all packages, but may apply extra charges based upon weight and distance.

```
rate = 20.00
If distance > 500 miles then rate = rate + 5.00
If weight > 48 ounces then rate = rate + 3.00
```

**SPU**

Create an SPU class that is a subclass of DeliveryDriver.

Each class will determine the rate based on the rate type, weight (in lbs), and distance.

```
If "4-Day Ground" then rate = (weight * 0.0050) * distance.
If "2-Day Business" then rate = (weight * 0.050) * distance.
If "Next Day" then rate = (weight * 0.075) * distance.
```

**PostageCalculator.java**

Create a command line program named PostageCalculator. The application will accept the weight of the package, whether or not the weight is in pounds or ounces, and the distance the package will be travelling. The output will display the rates for all possible delivery methods based on the inputs provided by the customer.

You *should not* need to check the type of DeliveryDriver in the main method when displaying the output.

*Expected output*

```
Please enter the weight of the package? 15
(P)ounds or (O)unces? O
```

```
What distance will it be traveling? 340

Delivery Method                 $ cost
--------------------------------------
Postal Service (1st Class)      $15.98
Postal Service (2nd Class)      $1.60
Postal Service (3rd Class)      $0.82
FexEd                           $20.00
SPU (4-Day Ground)              $1.59
SPU (2-Day Business)            $15.94
SPU (Next Day)                  $23.91
```

## Getting Started

- Import the polymorphism-exercises project into Eclipse.
- You should create a folder for each command line application you are working on inside of the `src/main/java/com/techelevator` folder.
    - One folder will be named `PostageCalculator`
    - The other folder will be named `TollBoothCalculator`
    - The code for each of the programs, to include classes for each, will be in each of their respective folders.
- Each of the programs will be a console application. This will be how you will verify your code works as expected.

## Tips and Tricks

- When writing your program, a question to ask yourself to ensure you are writing your code polymorphically is, "If a new requirement were provided, how much of my code will I need to change?" If the answer to this question is, "quite a bit", then you may have the opportunity to improve the quality of your code. Your code should be "open to extension, and closed for modification.open-closed-principle." In other words, what will happen if the Postal Service adds a new Economy delivery type? What would happen to the Toll Booth calculator if a new requirement were added for another type of vehicle?
- As you work on this assignment, you will need to verify that the application is working as expected for all of the scenarios outlined in the requirements. To do this, you will be practicing a form of testing referred to as "Manual Testing". Ideally, you will write automated unit tests to verify the applications you work on behave as expected. However, manual testing is equally important, and is a skill that is important on software development teams.
- Be sure you are verifying all of the execution paths for the application you are working on. For instance, verify that the postage calculator determines the correct rate when the distance is greater than 500 miles, and when the distance is less than 500 miles. Verify that the rates for each of the classes is being calculated correctly as well. Use the rate tables specified in the requirements for guidance.

---