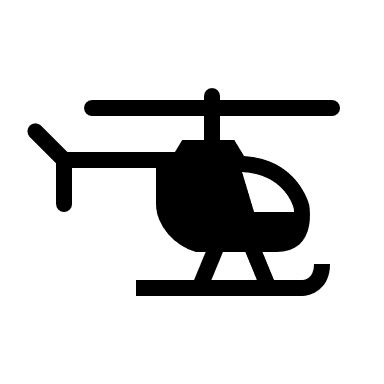
OOP Inheritance, abstraction and Polymorphism exercise

Note to the reader – This is a walk through exercise. Do not panic if some of this does not sink in. This is natural and it takes time and practice. Have fun. The IDE is your friend. Conal

**The "is-a" Relationship:**  
A Vehicle is loosely defined as a means of transport. A car is a vehicle. A motorcycle is a vehicle. A helicopter is a vehicle…

*Getters/Setters and validation will be left out intentionally. Object creation will be with constructors.*

**Step 1**  
Create an abstract Vehicle class with the following fields:

* make (String): The manufacturer of the vehicle.
* model (String): The specific model of the vehicle.
* year (Integer): The manufacturing year.
* isRunning (Boolean): Indicates whether the vehicle is currently running.

Use appropriate data types (such as the ones recommended above) and ensure encapsulation.

**Step 2**  
Create a constructor in the Vehicle class to initialize these fields.

**Step 3**  
Add the following abstract methods to Vehicle:

1. displayInfo: Must be implemented by child classes to print the vehicle's details.
2. honk: Must be implemented by child classes to provide a unique honking sound for each vehicle type.

**(Using abstract classes and methods ensures that child classes provide specific implementations. Additionally, abstract classes can provide common functionality that is shared across subclasses, such as start and stop)**

**A screen shot of a computer program

Description automatically generated**

**Step 4**  
Create the following concrete methods in the Vehicle class:

1. start: Simulates starting the vehicle. Update isRunning and print appropriate messages.
2. stop: Simulates stopping the vehicle. Update isRunning and print appropriate messages.

**A computer screen shot of a program code

Description automatically generated**

**Step 5**  
Create a Car class that:

* Extends Vehicle.
* Adds a field: numberOfDoors (Integer).
* Implements displayInfo to include numberOfDoors.
* Implements honk to print a unique car honking sound (e.g., "Beep beep!") or whatever you fancy.

A screen shot of a computer program

Description automatically generated

**Step 6**  
Repeat Step 5 for:

**Motorcycle:**

* Add a field: hasSideCar (Boolean).
* Customize displayInfo and honk.

**Helicopter:**

* Add a field: altitude (Integer).
* Add methods increaseAltitude and land.
* Customize displayInfo and honk.

**A screen shot of a computer code

Description automatically generated**

**A screen shot of a computer code

Description automatically generated**

**A screen shot of a computer program

Description automatically generated**

**Step 7**  
Create a Main class with a main method. In the Main class, we can create vehicles and actually demonstrate polymorphism:

* Create an array of Vehicle objects containing instances of Car, Motorcycle, and Helicopter.
* Iterate over the array and call honk and displayInfo for each object.

**A computer screen shot of a program code

Description automatically generated**

**Main class overview:**

Array of Vehicles:

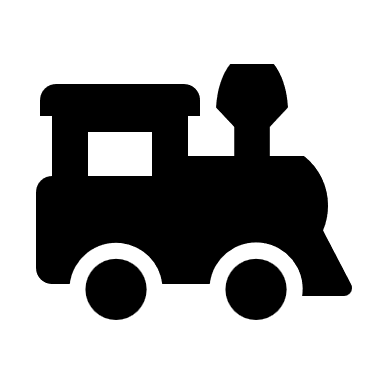
* The array contains different child class instances (Car, Motorcycle, Helicopter) using the parent class Vehicle. This is **Upcasting** (the process of converting a child class type to a parent class type).

Starting All Vehicles:

* The start method is called on each vehicle, demonstrating shared functionality inherited from the Vehicle class.

Polymorphism:

* The honk and displayInfo methods are overridden in each child class and dynamically dispatched (notes on this at bottom of document) based on the actual object type during runtime. Due to **polymorphism**, the actual start() method that gets executed is determined at runtime, based on the **actual type** of the object (Car, Motorcycle, or Helicopter), not the reference type (Vehicle).

****

**Some additional stuff we can do - what a time!**

**A screen shot of a computer program

Description automatically generated**

Specific Operations for Helicopters:

* Use **instanceof** to identify the Helicopter type and perform helicopter-specific operations (increaseAltitude and land).

Stopping All Vehicles:

* The stop method demonstrates shared functionality once again.

**Example output**

A screenshot of a computer

Description automatically generated

Finished!!!!!!



Notes:

**Dynamic Dispatch at work**

* **The reference type is Vehicle**, but the object type (the actual class of the object at runtime) is one of its subclasses (Car, Motorcycle, Helicopter).
* When you call methods start(), displayInfo(), honk() on the Vehicle reference, **Java runtime figures out which version of a method to call based on the object's actual class type (**magic just, I don’t know the inner workings**)**.

This is dynamic dispatch. The method that is executed is determined dynamically at runtime, not at compile time.