Computer Science 2
Object-Oriented Programming: Classes

1 Object-Oriented Programming Review

The idea of object-oriented programming is that we represent our data as objects. Each object is a specific instantiation of a class — the class defines the data components that each object of the class will have, but each object contains its own values for those components.

In Python, we generally used classes for standard data structures such as trees and graphs. However, we can also use classes for other types of data that is more complex than a single value. In addition, we can encode various behaviors that objects of that class can perform. By invoking a behavior on a particular object, we can change the values within the object, but in a clearly-defined way.

Unlike Python's classes, Java's classes allow the programmer to insist on additional constraints on how the classes are defined and used, using a concept called *encapsulation*. That is, we can create data within an object that is *private* to the object itself, so that other objects cannot modify the data directly — instead, we must invoke the behaviors on the object that will cause the data to be modified.

For example, consider a simulation of cars driving along a race course. In this case, a single Vehicle object may contain a number of simple data values. For example, speed, fuel level, color, steering direction, location on the track, and more can be contained within the Vehicle.

Alongside these data items, often called *states* of the Vehicle, the Vehicle class can define particular behaviors. For example, a Vehicle may be able to `brake`, `accelerate`, `refuel`. etc. By implementing these as behaviors within the Vehicle class, we can set it up so that the simulation can not set the `speed` value directly (0 to 60 in zero seconds?!), but instead has to use the `accelerate` and `brake` behaviors. These behaviors can then check that their parameters are reasonable and set the speed value appropriately.

We have already seen various Java classes with behaviors, such as the `Scanner` class with its `next()`, `nextInt()`, and so on. The nice thing about encapsulation is that we do not have to know how the `Scanner` works — nor can we accidentally mess it up by altering its internal state!

2 Initial implementation

To describe a class in Java, we will first start by laying out the states that are part of the class (these are just an example, depending on what we need from the Vehicle class there could be many other choices):

```
public class Vehicle {

    private String name;
    private double tankCapacity;
    private double fuelLeft;
    private int maxSpeed;

    private double speed;
    private double distanceTravelled;

}
```

The states are also called *instance variables*, since they have a separate existence for each object, or instance, of the class. Note that for each of our states, we have given it a data type, as required for every variable that is declared in a Java program. Also, we have declared these variables to be `private`. This means that objects of other classes cannot access these values directly. In general, we will declare all state variables as private to enforce the idea of encapsulation.

The next thing to add are the behaviors. These will also go inside the class scope. Note that while each object of the `Vehicle` class will have its own values for the state variables, the behaviors will work in the same way for any `Vehicle`.

Let's consider the `accelerate` behavior. Instead of setting the speed directly, it could take a parameter to indicate the amount of acceleration. The behavior can then check the vehicle's `maxSpeed` to make sure the acceleration is possible:

```
    public void accelerate(int factor) {
        speed += factor;
        if (speed > maxSpeed) {
            speed = maxSpeed;
        }
    }
```

Because the behavior is written inside the class, it can access the states directly by name. They are not declared within the function because they were already declared at the top of the class. This is important because it allows the object to "remember" the values from one behavior call to the next — whatever the speed is after one acceleration, that is the speed of the vehicle until the next time a behavior is called. The behavior itself is specified to be `public` so that it can be called from other classes.

Since the states are private, we cannot even access them from outside the class. Thus, if for some reason the code wants to know certain information, we need to provide a behavior for it. For example, we can write a behavior to return whether the vehicle is out of gas:

```
    public boolean outOfGas() {
        return fuelLeft == 0;
    }
```

3 Constructors

Of course, in order to create a `Vehicle` object in our program, we will invoke the construction process through a statement using the `new` keyword, such as

```
Vehicle v = new Vehicle();
```

Just as in Python, this invokes a constructor, and just as in Python, it is optional to write a constructor for our class. If (and only if) we do not write a constructor, the Java compiler will build one for us that takes no arguments and leaves all the state variables in the default value (`null` for object types, zero for numeric types). In general, we will want to write a constructor for all but the very simplest classes. In Java, the name of the constructor is simply the name of the class itself, and it has no return type:

```
    public Vehicle(String nameVal, int mpgVal,
                   int tankCapacityVal, int maxSpeedVal) {
        name = nameVal;
        mpg = mpgVal;
        tankCapacity = tankCapacityVal;
        maxSpeed = maxSpeedVal;

        fuelLeft = tankCapacity;
        speed = 0;
        distanceTravelled = 0;
    }
```

Note that we have set some of the instance variables using the parameters of the constructor, while others are simply set to default values. This constructor could be called from our simulation with a statement such as

```
Vehicle v = new Vehicle("Honda",32,12,110);
```

In Java, we are allowed to write multiple constructors, as long as they have different parameter lists. For example, we could write a second constructor that takes no arguments but generates reasonable default values:

```
    public Vehicle() {
        name = "Default";
        mpg = 30;
        tankCapacity = 15;
        maxSpeed = 100;

        fuelLeft = tankCapacity;
        speed = 0;
        distanceTravelled = 0;
    }
```

However, this looks like duplication of effort between the two constructors! Java allows us to chain constructors together, that is, to have one constructor use another. This is encouraged to avoid any cut/paste errors. Within a constructor, we can call another constructor simply through the use of the keyword `this`. So in this case, we could write our second constructor as:

```
public Vehicle() {
    this("Default",30,15,100);
}
```

Better yet, instead of just having "magic numbers" here, we can write some named constants so that it is clear what's going on.

```
public static final String DEFAULT_NAME = "Default";
public static final int DEFAULT_MPG = 30;
public static final double DEFAULT_CAPACITY = 15;
public static final int DEFAULT_MAX_SPEED = 100;

public Vehicle() {
    this(DEFAULT_NAME, DEFAULT_MPG, DEFAULT_CAPACITY,
             DEFAULT_MAX_SPEED);
}
```

Those constants are declared outside any function, but inside the class, just like instance variables. However, we have declared them differently in two ways. First of all, the keyword `final` means that these variables cannot be changed — they will retain their assigned value throughout the running of the progam. This is enforced by the Java compiler. Secondly, we have declared them to be `static`. This tells the compiler that it does not need to create a separate value for each `Vehicle` object, but rather that there is only one `DEFAULT_MPG` variable for the entire `Vehicle` class[1].

4 Other standard behaviors

Just as in Python, there are certain behaviors which are commonly implemented in all classes. (Actually, as we will see later on, all classes do have these particular behaviors, whether we write them or not.)

The first of these that we will see is the `toString()` behavior. This is similar to Python's `__str__()` function, producing a text representation of the object for printing. In Java, the `toString()` function has a required form — it must be declared as `public String toString() { ...}`. For our `Vehicle` class, we could write something like this:

1. So a variable that is `static` but not `final` can be changed by one object and that change will be seen by all other objects of that class! This is a "feature" that is not often useful, but is easy to write, so be careful!

```
    public String toString() {
        return name + " is going " + speed + " miles per hour and has " +
            fuelLeft + " gallons of gas left";

    }
```

Another important behavior is to define equality of objects in our class. In Java, there are two separate ideas of equality: we can have the exact same object under two names, or we could have two distinct objects that happen to have all the same internal state. Fortunately, Java allows us to test for both concepts. For example, our simulation could say this:

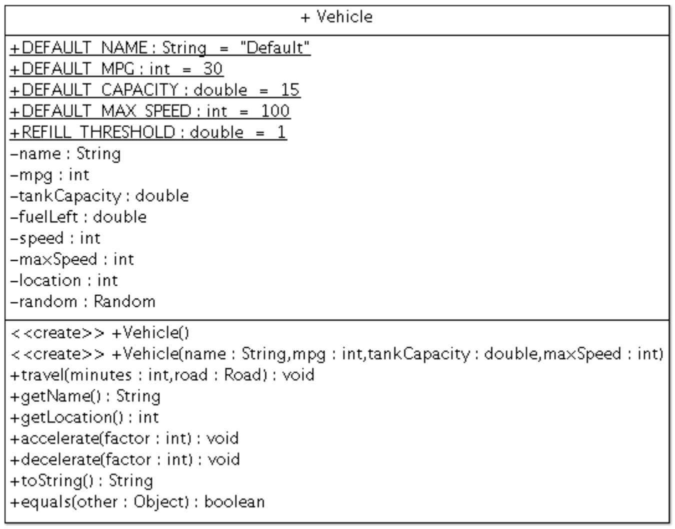
```
    Vehicle vehicle1 = new Vehicle("Honda", 30, 15, 100);
    Vehicle vehicle2 = new Vehicle("Honda", 30, 15, 100);
    // are they the same exact object?
    System.out.println(vehicle1 == vehicle2);
    // or just equivalent?
    System.out.println(vehicle1.equals(vehicle2));
```

While the first concept (==) is given to us by Java, it is up to us to come up with a definition for `equals()`. The `equals()` method must be defined as `public boolean equals(Object o) { ...}`. The idea is that any object might be passed in. But in order to decide if the object passed in is an equivalent Vehicle to ourselves, we have to first make sure that it is in fact a Vehicle (if not, it certainly isn't equal!). Once that is done, we can compare the contents of the two vehicles, like this:

```
    public boolean equals(Object o) {
        if (!(o instanceof Vehicle))
            return false;
        Vehicle ov = (Vehicle)(o);
        return name.equals(ov.name) && maxspeed == ov.maxspeed;
    }
```

5 UML

The Unified Modeling Language is a tool that allows us to graphically represent the classes and their relationships in a standard way. Here is the UML diagram for the final `Vehicle` class.


+ Vehicle
+DEFAULT_NAME : String = "Default"
+DEFAULT_MPG : int = 30
+DEFAULT_CAPACITY : double = 15
+DEFAULT_MAX_SPEED : int = 100
+REFILL_THRESHOLD : double = 1
-name : String
-mpg : int
-tankCapacity : double
-fuelLeft : double
-speed : int
-maxSpeed : int
-location : int
-random : Random
<<create>> +Vehicle()
<<create>> +Vehicle(name : String,mpg : int,tankCapacity : double,maxSpeed : int)
+travel(minutes : int,road : Road) : void
+getName() : String
+getLocation() : int
+accelerate(factor : int) : void
+decelerate(factor : int) : void
+toString() : String
+equals(other : Object) : boolean

6 Testing (Putting it all together)

Of course, a `Vehicle` class by itself doesn't do much — only when we have another class that uses it can we see what it's doing. The class `TestVehicle` is a simple one that creates some vehicles and runs some functions on them. The idea of *unit testing*, in which each class is tested independently (and each method within each class) to make sure it functions as intended, is a very important one in object-oriented programming.

The code in `CannonballRun` uses more of the functions in a more involved simulation scenario.