Indiana University
CSCI-C212: Introduction to Software Systems (F25)
Joshua Crotts

Problem Set 8
October 22, 2025

Classes and Objects

Important Dates:

- Assigned: October 22, 2025
- Deadline: October 29, 2025 at 11:59 PM EST

Objectives:

- Students design classes as blueprints for objects.

What To Do:

Because we're introducing classes with this problem set, it is no longer appropriate to use the class name `ProblemX`. Instead, design classes with the given specification in each problem, along with the appropriate test suite. **Do not round your solutions!**

What You Cannot Use:

You cannot use any content beyond Chapter 4.1. Namely, do not use interfaces, inheritance, or anything that trivializes the problem. This problem set is to get you acclimated with immutable classes and data representation. Therefore, mutation and aliasing are off the table. All instance variables should be declared as `final`. Please contact a staff member if you are unsure about something you should or should not use. Any use of anything in the above-listed forbidden categories will result in a **zero** (0) on the problem set.

Warning About This Problem Set:

This problem set is purposefully short to give you a break before the next problem set, which is much harder and longer. Be prepared!

Problem 1:

Design the `Car` class, which stores a `String` representing the car's make, a `String` representing the car's model, and an `int` representing the car's year. Its constructor should receive these three values and store them in the instance variables. Be sure to write instance accessor methods for modifying all three fields. That is, you should write `getMake()`, `getModel`, and `getYear` to access the fields directly. Override the `equals`, `hashCode`, and `toString` methods: two `Car` objects are equal if their make, model, and year are the same. The hash code of a `Car` is the hash code of its instance variables. `toString` can be overridden however you wish, as long as it contains the three instance variables somehow.

Problem 2:

Design the `Dog` class, which stores a `String` representing the breed, a `String` representing its name, and an `int` representing its age in years. You should also store a boolean to keep track of whether or not the dog is a puppy. A dog is a puppy if it is less than two years old. Its constructor should receive these three values and store them in the instance variables. Be sure to write instance accessor methods for all three fields. That is, you should write `getBreed`, `getName`, and `getAge` to access the fields directly. You should also write the `boolean isPuppy` method to determine if the `Dog` is a puppy according to the above criteria. Override the `equals`, `hashCode`, and `toString` methods: two `Dog` objects are equal if their breed, name, and age are the same. The hash code of a `Dog` is the hash code of its instance variables. `toString` can be overridden however you wish, as long as it contains the three instance variables somehow.

Problem 3:

This exercise has nine parts. In this question you will design a class that represents a two-dimensional vector. Vectors have a direction and a magnitude, and are represented by two components: x and y.

(a) Design the `Vector` class, which stores two `double` values representing the x and y components of the vector. The constructor should receive these two values and assign them to the instance variables.

(b) Design the `double getX()`, `double getY()`, methods that return the respective components of the vector.

(c) Design the `double getMagnitude()` method that returns the magnitude of the vector, which is defined as $\sqrt{x^2 + y^2}$.

(d) Design the `Vector add(Vector v)` method that adds a given vector v to this vector. The result is a new `Vector` instance whose components are the sum of the respective components of both vectors.

(e) Design the `Vector subtract(Vector v)` method that subtracts a given vector v from this vector. The result is a new vector whose components are the difference of the respective components of both vectors.

(f) Design the `Vector scale(double s)` method that scales this vector by a given scalar s. The result is a new vector whose components are multiplied by s.

(g) Design the `Vector normalize()` method that normalizes this vector to the unit vector, i.e., a vector with a magnitude of 1. The result is a new vector whose components are divided by the magnitude of this vector.

(h) Design the `double dot(Vector v)` method that computes the dot product of this vector with a given vector v. The dot product is defined as $x_1 \cdot x_2 + y_1 \cdot y_2$, where (x_1, y_1) and (x_2, y_2) are the components of the two vectors.

(i) Finally, override the `equals`, `hashCode`, and `toString` methods.

- Two vectors are equal if their components are equal.
- The hash code of a vector is the sum of the hash codes of its components.
- The `toString` representation of a vector is the stringified version of the vector, e.g., `"(x, y)"`.

Problem 4:

The *bag*, or multi-set, data structure is a simple set-like data structure, with the exception that elements can be inserted multiple times into a bag. Order does not matter with a bag, nor does its internal structure. With bags, we can add items, remove items, query how many of an item there are, and return the number of items in the bag.

(a) Design the generic `Bag<T>` class. It would be inefficient to simply store a list of all the items in the bag. So, your bag will employ a `Map<T, Integer>`, which associates an item T with its count in the map. Instantiate the map as an instance variable as a `HashMap`.

(b) Design the `void insert(T t)` method, which inserts an item t into the bag. If the item already exists, its associated count is incremented by one.

(c) Design the `boolean remove(T t)` method, which removes an item t from the bag, meaning its associated count is decremented by one. If t does not exist, the method returns `false`, and otherwise returns `true`. If decrementing the count results in 0, then remove the key t.

(d) Design the `int count(T t)` method that returns the respective quantity of item t in the bag.

(e) Design the `int size()` method that returns the number of items in the bag. The method should *not* traverse over the keys and sum the values; this should be a "constant-time" operation. Don't overthink how to do this!

(f) Design the `boolean contains(T t)` method that returns whether the bag contains `t`.

(g) Design the `boolean isSubBagOf(Bag b)` method that determines `this` is a sub-bag of `b`. A bag b_1 is a sub-bag of b_2 if, for every element $i \in b_1$, then $i \in b_2$ and $\texttt{count}(b_1[i]) \leq \texttt{count}(b_2[i])$.