Composition

by Sophia

WHAT'S COVERED

In this lesson, you will learn ways that Java classes can be used as components of other classes. You will also learn about building classes that make use of other classes as a way to add common functionality without using inheritance. Specifically, this lesson covers:

Table of Contents

- 1. Aggregation
- 2. Composition

1. Aggregation

Aggregation is the combining of Java objects into another class. For instance, in the lesson on interfaces, we worked with a DrinkOrder class that consisted of one or more Beverage objects in an ArrayList collection.

Just as a reminder, here is the DrinkOrder class:

```
import java.util.ArrayList;

public class DrinkOrder {
  // ArrayList can hold both Coffee & Tea objects since both implement the
  // Beverage interface
  private ArrayList<Beverage> order = new ArrayList<>();

  // add() method accepts an object any class that implements Beverage interface
  public void add(Beverage beverage) {
    order.add(beverage);
  }

  // Add up total for order
  public double getTotalPrice() {
    double total = 0;
    for(Beverage beverage : order) {
      total += beverage.getPrice();
    }
    return total;
  }

  // Return ArrayList of Beverages in order
  public ArrayList<Beverage> getOrder() {
    return order;
  }
}
```

Aggregation can be thought of as a structure where one class "has one or more" objects of another class. In the DrinkOrder<c/ode> example, a <code>DrinkOrder had one or more Beverage objects (which could have been coffee or tea). This "ownership" was just one way, though. A Coffee or Tea object could not really be said to have a DrinkOrder. The objects in an aggregation are relatively independent in that they could exist or be useful even if not part of the larger object.

TERM TO KNOW

Aggregation
Aggregation is the combining of Java classes to create another class (instances of the combined classes are used as attributes in the new class).

2. Composition

Composition is a more specific type of aggregation. In the case of**composition**, the ties between a component object and the larger object are much tighter, so that the component can't really exist apart from the larger object. If the larger containing object is deleted or destroyed, the objects inside it can practically be said to go out of existence. Think of the rooms in a house or the sections of a book as examples. If a house is destroyed, the rooms don't have separate existences. The same can be said of the parts of a book. If the book is shredded, the table of contents, chapters, and bibliography also go out of existence (at least practically).

Let's apply the concept of composition to the Coffee and Tea classes that we have seen previously. The Beverage class encapsulates the data about the name of the drink, the size, and price. This Beverage data is part of the information about coffee and tea, but a Beverage object doesn't really exist apart from a specific drink (a Coffee or Tea object).

TRY IT

Directions: Enter this code for the Beverage class (in Beverage.java) in Replit:

```
import java.text.DecimalFormat;

public class Beverage {
  private String name;
  private int size;
  private double price;
  // The DecimalFormat for the price format is a rare public attribute, but since it's
  // final, the object is constant so it can't be modified (even though it's public)
  public final DecimalFormat PriceFormat = new DecimalFormat("$0.00");

  Beverage(String name, int size, double price) {
    this.name = name;
    this.size = size;
    this.price = price;
  }

  public String getName() {
    return name;
  }

  public int getSize() {
    return size;
  }

  public double getPrice() {
    return price;
  }
}
```

TRY IT

Directions: Enter this code for Embeds a Beverage object (saved in a file named Coffee.java) in Replit:

```
public class Coffee {
  // Embedded Beverage object
  private Beverage beverage;
  private String roastType;
  private boolean decaf;

  public Coffee(String roastType, int size, boolean decaf, double price) {
    // Construct the embedded Beverage object
    this.beverage = new Beverage("coffee", size, price);
    this.roastType = roastType;
    this.decaf = decaf;
  }

  // The Coffee class's getName() calls the embedded object's getName()
  public String getName() {
    return beverage.getName();
  }

  // The Coffee class's getSize() calls the embedded object's getSize()
  public int getSize() {
    return beverage.getSize();
  }

  // The Coffee class's getPrice() calls the embedded object's getPrice()
  public double getPrice() {
    return beverage.getPrice();
  }

  public String getRoastType() {
    return roastType;
  }

  public boolean isDecaf() {
    return decaf;
  }

  public String toString() {
    // Calls to accessor methods to access data in embedded Beverage object
    String item = roastType + " coffee (" + getSize() + " oz.) " + beverage.PriceFormat.format(getPrice());
    if(decaf) {
      return "decaf " + item;
    }
    else {
      return item;
    }
  }
}

public class Tea {
  private Beverage beverage;
  private String teaType;
  private boolean iced;

  public Tea(String teaType, int size, boolean iced, double price) {
    // Construct the embedded Beverage object
    this.beverage = new Beverage("tea", size, price);
    this.teaType = teaType;
    this.iced = iced;
  }

  // The Tea class's getName() calls the embedded object's getName()
  public String getName() {
    return beverage.getName();
  }

  // The Tea class's getSize() calls the embedded object's getSize()
  public int getSize() {
    return beverage.getSize();
  }

  // The Tea class's getPrice() calls the embedded object's getPrice()
  public double getPrice() {
    return beverage.getPrice();
  }

  public String getTeaType() {
    return teaType;
  }

  public boolean isIced() {
    return iced;
  }

  public String toString() {
    // Note use of accessor methods that forward calls to get info from embedded object
    String item = teaType + " tea (" + getSize() + " oz.) " + beverage.PriceFormat.format(getPrice());
    if(iced) {
      return "iced " + item;
    }
    else {
      return "hot " + item;
    }
  }
}
```

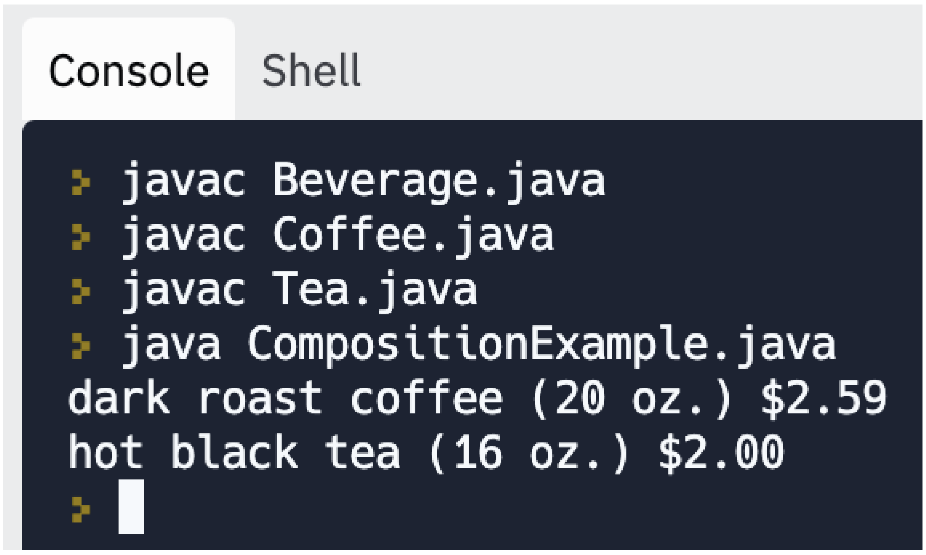
Here is a simple program for testing this version of the Coffee and Tea classes. Save this code in a file named CompositionExample.java.

TRY IT

Directions: Save this code in a file named CompositionExample.java:

```
class CompositionExample {
  public static void main(String[] args) {
    // Construct Coffee & Tea objects that have a Beverage objet embedded in them.
    // There's nothing unusual about the constructor calls.
    Coffee darkRoastCoffee = new Coffee("dark roast", 20, false, 2.59);
    Tea blackTea = new Tea("black", 16, false, 2.00);
    // Print out information about the drinks using their toString() methods
    System.out.println(darkRoastCoffee);
    System.out.println(blackTea);
  }
}
```

The output for this code looks like this:



```
> javac Beverage.java
> javac Coffee.java
> javac Tea.java
> java CompositionExample.java
dark roast coffee (20 oz.) $2.59
hot black tea (16 oz.) $2.00
>
```

REFLECT

Note how the code in the application's `main()` isn't visibly different, even though the code in the classes is different. This allows the programmer some latitude in designing a class without having an effect on how the class is used in other code.

TERM TO KNOW

Composition
Composition is a more specific type of aggregation where the component can't really exist apart from the larger object.

SUMMARY

In this lesson, you learned about approaches using **aggregation** and **composition** to class design that embed one object in another object. You also learned about using embedded objects as another way to provide common pieces of functionality, similar to how inheritance can be used to provide common functionality.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/

TERMS TO KNOW

Aggregation
Aggregation is the combining of Java classes to create another class (instances of the combined classes are used as attributes in the new class).

Composition
Composition is a more specific type of aggregation where the component can't really exist apart from the larger object.