Assignment #8: Order Management

Due Dates:

8A: Friday, December 31, 2:00pm

8B: Friday, January 7, 2:00pm

Overview

In this assignment you will build the back end of a business that sells products and also provides professional services. We use the word "services" to refer to things that you hire a person to do for you on an hourly basis, and "products" are physical items that you order and are shipped to you. In this assignment you will practice the following:

1) **Implementing an interface**
2) **Writing `equals` and `hashCode` methods**
3) **Using collections**
4) **Using java packages**

What You Will Submit

Unlike the first 7 homework assignments, this one requires you to write your code inside a package. The name of the package that all of your code must be a part of is edu.yu.cs.intro.orderManagement

You must submit 6 .java files:

1) `OrderManagementSystem.java`
2) `Warehouse.java`
3) `Order.java`
4) `Product.java`
5) `Service.java`
6) `ServiceProvider.java`

One other file will be needed to compile and run your program (`Item.java`) but it may not be changed at all from the interface I provided you. I will simply use my own copy of that file to compile and run your program.

Important Points

1) You may not make **ANY** changes to the interface I have provided you or the to the signatures of any methods that you have been required to implement. If you do, the code will not even compile when I go to test it and you will get a zero
2) You MUST implement `equals` and `hashCode` for both the Product class and the Service class. It is only their itemNumber (a.k.a. serviceID or productID) that uniquely identifies them, and you must take that into account when you write the `equals` and `hashCode` methods
3) You should avoid writing code that iterates through entire collections – use collections and the methods on the various collections interfaces wisely to make your program as fast and efficient as possible
4) Do not use an array anywhere in your project except for when creating the return value of a method that returns an array – use collections wherever you need a data structure
5) You can add whatever private and protected methods you want to all the classes, but you MAY NOT add additional public methods

6) Avoid replicating information, i.e. don't create multiple objects with the same exact info, but do keep in mind that if a single object is stored in multiple collections, that is not replicating info, rather it is simply having multiple ways to view/access a single object. This is a useful fact to take advantage of when trying to make your code efficient with respect to finding information relevant to a given method.
7) Keep in mind that an order is only complete once all of its products AND services have been shipped/allocated
8) Keep in mind that my test code will be in the same package as your code, so I can call any/all protected and public methods
9) Note that many of the methods throw unchecked exceptions under certain circumstances. This is specified in the comments. Not throwing those exceptions in the described circumstances is a bug/error/point-losing-thing

Classes and API

Read every line, including the comments, very carefully!

Item

```
package edu.yu.cs.intro.orderManagement;

public interface Item {
    int getItemNumber();
    String getDescription();
    double getPrice();
}
```

OrderManagementSystem

```
package edu.yu.cs.intro.orderManagement;


/**
 * Takes orders, manages the warehouse as well as service providers
 */
public class OrderManagementSystem {

    /**
     * Creates a new Warehouse instance and calls the other constructor
     *
     * @param products
     * @param defaultProductStockLevel
     * @param serviceProviders
     */
    public OrderManagementSystem(Set<Product> products, int defaultProductStockLevel,
Set<ServiceProvider> serviceProviders) {}


    /**
     * 1) populate the warehouse with the products.
     * 2) retrieve set of services provided by the ServiceProviders, to save it as the set of
services the business can provide
     * 3) create map of services to the List of service providers that provide them
     *
     * @param products                 - set of products to populate the warehouse with
     * @param defaultProductStockLevel - the default number of products to stock for any product
     * @param serviceProviders         - set of service providers and the services they provide, to
make up the services arm of the business
```

```
     * @param warehouse               - the warehouse that we will store our products in
     */
    public OrderManagementSystem(Set<Product> products, int defaultProductStockLevel,
Set<ServiceProvider> serviceProviders, Warehouse warehouse) {}

/**
 * Accept an order:
 * 1) See if we have ServiceProviders for all Services in the order. If not, reject the order.
 * 2) See if we can fulfill all Items in the order. If so, place the product orders with the
warehouse and handle the service orders inside this class
 * 2a) We CAN fulfill a product order if either the warehouse currently has enough quantity in
stock OR if the product is NOT on the "do not restock" list.
 *  In the case that the current quantity of a product is < the quantity in the order AND the
product is NOT on the "do not restock" list, the order management system should
 *  first instruct the warehouse to restock the item, and then tell the warehouse to fulfill this
order.
 * 3) Mark the order as completed
 * 4) Update the busy status of service providers involved...
* @throws IllegalArgumentException if any part of the order for PRODUCTS can't be fulfilled
* @throws IllegalStateException if any part of the order for SERVICES can't be fulfilled
 */
  public void placeOrder(Order order) {}

  /**
   * Validate that all the services being ordered can be provided. Make sure to check how many instances of a given service are being requested in
the order, and see if we have enough providers for them.
   * @param services the set of services which are being ordered inside the order
   * @param order the order whose services we are validating
   * @return itemNumber of the first requested service encountered that we either do not have a provider for at all, or for which we do not have an
available provider. Return 0 if all services are valid.
   */
  protected int validateServices(Collection<Service> services, Order order) {}

  /**
   * validate that the requested quantity of products can be fulfilled
   * @param products being ordered in this order
   * @param order the order whose products we are validating
   * @return itemNumber of product which is either not in the catalog or which we have insufficient quantity of. Return 0 if we can fulfill.
   */
  protected int validateProducts(Collection<Product> products, Order order) {}

  /**
   * Adds new Products to the set of products that the warehouse can ship/fulfill
   * @param products the products to add to the warehouse
   * @return set of products that were actually added (don't include any products that were already in the warehouse before this was called!)
   */
  protected Set<Product> addNewProducts(Collection<Product> products) {}

  /**
   * Adds an additional ServiceProvider to the system. Update all relevant data about which Services are offered and which ServiceProviders provide
which services
   * @param provider the provider to add
   */
  protected void addServiceProvider(ServiceProvider provider) {}

  /**
   *
   * @return get the set of all the products offered/sold by this business
   */
  public Set<Product> getProductCatalog() {}

  /**
   * @return get the set of all the Services offered/sold by this business
   */
  public Set<Service> getOfferedServices() {}

  /**
   * Discontinue Item, i.e. stop selling a Service or Product.
   * Also prevent the Item from being added in the future.
   * If it's a Service - remove it from the set of provided services.
   * If it's a Product - still sell whatever instances of this Product are in stock, but do not restock it.
   * @param item the item to discontinue see {@link Item}
   */
  protected void discontinueItem(Item item) {}

  /**
   * Set the default product stock level for the given product
   * @param prod
   * @param level
   */
  protected void setDefaultProductStockLevel(Product prod, int level) {}
```

Warehouse

```
package edu.yu.cs.intro.orderManagement;

/**
 * Stocks products, fulfills product orders, manages stock of products.
 */
public class Warehouse {

    /**
     * create a warehouse, initialize all the instance variables
     */
    protected Warehouse(){}

    /**
     * @return all unique Products stocked in the warehouse
     */
    protected Set<Product> getAllProductsInCatalog(){}

    /**
     * Add a product to the warehouse, at the given stock level.
     * @param product
     * @param desiredStockLevel the number to stock initially, and also to restock to when
subsequently restocked
     * @throws IllegalArgumentException if the product is in the "do not restock" set, or if the
product is already in the warehouse
     */
    protected void addNewProductToWarehouse(Product product, int desiredStockLevel){ }

    /**
     * If the actual stock is already >= the minimum, do nothing. Otherwise, raise it to minimum OR
the default stock level, whichever is greater
     * @param productNumber
     * @param minimum
     * @throws IllegalArgumentException if the product is in the "do not restock" set, or if it is
not in the catalog
     */
    protected void restock(int productNumber, int minimum){}

    /**
     * Set the new default stock level for the given product
     * @param productNumber
     * @param quantity
     * @return the old default stock level
     * @throws IllegalArgumentException if the product is in the "do not restock" set, or if it is
```

```java
not in the catalog
     */
    protected int setDefaultStockLevel(int productNumber, int quantity){}

    /**
     * @param productNumber
     * @return how many of the given product we have in stock, or zero if it is not stocked
     */
    protected int getStockLevel(int productNumber){}

    /**
     * @param itemNumber
     * @return true if the given item number is in the warehouse's catalog, false if not
     */
    protected boolean isInCatalog(int itemNumber){}

    /**
     *
     * @param itemNumber
     * @return false if it's not in catalog or is in the "do not restock" set. Otherwise true.
     */
    protected boolean isRestockable(int itemNumber){}

    /**
     * add the given product to the "do not restock" set
     * @param productNumber
     * @return the current actual stock level of the product
     */
    protected int doNotRestock(int productNumber){}

    /**
     * can the warehouse fulfill an order for the given amount of the given product?
     * @param productNumber
     * @param quantity
     * @return false if the product is not in the catalog or there are fewer than quantity of the
products in the catalog. Otherwise true.
     */
    protected boolean canFulfill(int productNumber, int quantity){}

    /**
     * Fulfill an order for the given amount of the given product, i.e. lower the stock levels of
the product by the given amount
     * @param productNumber
     * @param quantity
     * @throws IllegalArgumentException if {@link #canFulfill(int, int)} returns false
     */
    protected void fulfill(int productNumber, int quantity){}
```

Order

```java
package edu.yu.cs.intro.orderManagement;

/**
 * Represents an order placed by a customer. An item in the order can be an instance of either
Product or Service
 */
public class Order {

    public Order(){}

    /**
     * @return all the items (products and services) in the order
     */
    public Item[] getItems(){}

    /**
     * @param b
     * @return the quantity of the given item ordered in this order. Zero if the item is not in the
order.
     */
    public int getQuantity(Item b){ }

    /**
     * Add the given quantity of the given item (product or service) to the order
     * @param item
     * @param quantity
     */
    public void addToOrder(Item item, int quantity){}

    /**
     * Calculate the total price of PRODUCTS in the order. Must multiply each item's price by the
quantity.
     * @return the total price of products in this order
     */
    public double getProductsTotalPrice(){}

    /**
     * Calculate the total price of the SERVICES in the order. Must multiply each item's price by
the quantity.
     * @return the total price of products in this order
     */
    public double getServicesTotalPrice(){}

    /**
     * @return has the order been completed by the order management system?
     */
    public boolean isCompleted() {}

    /**
     * Indicate if the order has been completed by the order management system
     * @param completed
     */
    public void setCompleted(boolean completed) {}
}
```

Product

```
package edu.yu.cs.intro.orderManagement;

/**
 * A "physical" item that is "stocked" in the warehouse.
 */
public class Product implements Item {

    public Product(String name, double price, int productID){}

    @Override
    public int getItemNumber() {}

    @Override
    public String getDescription() {}

    @Override
    public double getPrice() {}

    @Override
    public boolean equals(Object o) {}

    @Override
    public int hashCode() {}
}
```

Service

```
package edu.yu.cs.intro.orderManagement;

/**
 * An implementation of item which represents a Service provided by the business.
 * Has a price per billable hour as well a number of hours this service takes.
 * The price returned by getPrice must be the per hour price multiplied by the number of hours the
service takes
 */
public class Service implements Item {

    public Service(double pricePerHour, int numberOfHours, int serviceID, String description){}

    /**
     * @return the number of hours this service takes
     */
    public int getNumberOfHours(){}

    @Override
    public int getItemNumber() {}

    @Override
    public String getDescription() {}

    @Override
    public double getPrice() {}

    @Override
    public boolean equals(Object o) {}

    @Override
    public int hashCode() {}
}
```

ServiceProvider

```
package edu.yu.cs.intro.orderManagement;

/**
 * 1) has a Set of services that it can provide
 * 2) can only work on one order at a time - once assigned to a customer, can’t take another
assignment until 3 other orders have been placed with the order management system
* 3) is uniquely identified by its ID
 */
public class ServiceProvider implements Comparable<ServiceProvider>{
    /**
     *
     * @param name
     * @param id unique id of the ServiceProvider
     * @param services set of services this provider can provide
     */
    public ServiceProvider(String name, int id, Set<Service> services){ }

    public String getName(){}

    public int getId(){}

    /**
     * Assign this provider to a customer. Record the fact that he is busy.
     * @throws IllegalStateException if the provider is currently assigned to a job
     */
    protected void assignToCustomer(){}

    /**
     * Free this provider up - is no longer assigned to a customer
     * @throws IllegalStateException if the provider is NOT currently assigned to a job
     */
    protected void endCustomerEngagement(){}

    /**
     * @param s add the given service to the set of services this provider can provide
     * @return true if it was added, false if not
     */
    protected boolean addService(Service s){}

    /**
     * @param s remove the given service from the set of services this provider can provide
     * @return true if it was removed, false if not
     */
    protected boolean removeService(Service s){}

    /**
     *
     * @return a COPY of the set of services. MUST NOT return the Set instance itself, since that
would allow a caller to then add/remove services to/from the set
     */
    public Set<Service> getServices(){}

    @Override
    public boolean equals(Object o) {}

    @Override
    public int hashCode() {}

    @Override
    public int compareTo(ServiceProvider other) {}

}
```

Demo Program

This is not an exhaustive test – just a demo.

Note that for the assertions in this program to work, you must run it with the –ea option, i.e.:

```
java -ea edu.yu.cs.intro.orderManagement.Demo
```

```
package edu.yu.cs.intro.orderManagement;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

public class Demo {
    private Set<Product> products;
    private Set<ServiceProvider> providers;
    private Set<Service> allServices;
    private Map<Integer, Product> idToProduct;
    private Map<Integer, Service> idToService;
    private Warehouse warehouse;

    public static void main(String[] args) {
        Demo dd = new Demo();
        dd.runDemo();
    }

    public Demo(){
        this.warehouse = new Warehouse();
        this.products = new HashSet<>();
        this.idToProduct = new HashMap<>();
        this.idToService = new HashMap<>();
        this.allServices = new HashSet<>();
        this.providers = new HashSet<>();
    }

    void runDemo(){
        OrderManagementSystem system = new
OrderManagementSystem(this.products,5,this.providers,this.warehouse);
        //populate our system with products and services
        this.createDemoProducts();
        system.addNewProducts(this.products);
        this.createDemoServiceProviders();
        for(ServiceProvider p : this.providers){
            system.addServiceProvider(p);
        }
        //make sure all the products added are in the catalog
        Set<Product> catalog = system.getProductCatalog();
        assert this.products.size() == catalog.size();
        assert catalog.containsAll(this.products);
        //make sure all the services are in the services offered
        Set<Service> services = system.getOfferedServices();
        assert this.allServices.size() == services.size();
        assert services.containsAll(this.allServices);

        //create an order
        Order order = new Order();
        order.addToOrder(this.idToProduct.get(1),3); //will use out of 5 of product #1
        order.addToOrder(this.idToService.get(6),1); //will use the only service provider for #6
        system.placeOrder(order);
        assert this.warehouse.getStockLevel(1) == 2;
        assert order.isCompleted();

        //place another order, should throw IllegalStateException
        order = new Order();
        order.addToOrder(this.idToService.get(6),1); //provider for #6 not available - should throw
exception
        boolean caught = false;
        try{
            system.placeOrder(order);
        }catch (IllegalStateException e){
            caught = true;
        }
        assert caught;
        assert !order.isCompleted();

        //force it to throw exception for ordering more than available of a discontinued item
        system.discontinueItem(this.idToProduct.get(1));
        order = new Order();
        order.addToOrder(this.idToProduct.get(1),3); //only 2 left of product #1
        caught = false;
        try{
            system.placeOrder(order);
        }catch (IllegalArgumentException e){
            caught = true;
        }
        assert caught;
        assert !order.isCompleted();

        //order more than available of a current item, make sure it ups the stock level and
fulfills it
        assert this.warehouse.getStockLevel(2) == 5;
        order = new Order();
        order.addToOrder(this.idToProduct.get(2),10);
        system.placeOrder(order);
        assert order.isCompleted();
        assert this.warehouse.getStockLevel(2) == 0;
        this.warehouse.restock(2,10);
        assert this.warehouse.getStockLevel(2) == 10;

        //place 2 more order2 to make 3 orders since service provider for 6 was all busy. Should
then be able to place order for service #6
        order = new Order();
        order.addToOrder(this.idToProduct.get(3),1);
        system.placeOrder(order);
        assert order.isCompleted();
        order = new Order();
        order.addToOrder(this.idToProduct.get(4),1);
        system.placeOrder(order);
        assert order.isCompleted();

        order = new Order();
        order.addToOrder(this.idToService.get(6),1);
        system.placeOrder(order);
        assert order.isCompleted();

    }


    private void createDemoProducts(){
        this.products.add(new Product("prod1",1,1));
        this.products.add(new Product("prod2",2,2));
        this.products.add(new Product("prod3",3,3));
        this.products.add(new Product("prod4",4,4));
        this.products.add(new Product("prod5",5,5));
        this.products.add(new Product("prod6",6,6));
        this.products.add(new Product("prod7",7,7));
        for(Product p : this.products){
            this.idToProduct.put(p.getItemNumber(),p);
        }
    }

    private void createDemoServiceProviders(){
        Service s1 = new Service(1,1,1,"srvc1");
        Service s2 = new Service(2,1,2,"srvc2");
        Service s3 = new Service(3,1,3,"srvc3");
        Service s4 = new Service(4,1,4,"srvc4");
        Service s5 = new Service(5,1,5,"srvc5");
        Service s6 = new Service(6,1,6,"srvc6");

        Set<Service> srvcSetAll = new HashSet<>();
        srvcSetAll.add(s1);
        srvcSetAll.add(s2);
        srvcSetAll.add(s3);
        srvcSetAll.add(s4);
        srvcSetAll.add(s5);
        srvcSetAll.add(s6);
        this.allServices.addAll(srvcSetAll);
        for(Service srvc : this.allServices){
            this.idToService.put(srvc.getItemNumber(),srvc);
        }

        this.providers.add(new ServiceProvider("p1",1,srvcSetAll));

        Set<Service> srvcSetThree = new HashSet<>();
        srvcSetAll.add(s1);
        srvcSetAll.add(s2);
        srvcSetAll.add(s3);
        this.providers.add(new ServiceProvider("p2",2,srvcSetThree));

        Set<Service> singleService = new HashSet<>();
        srvcSetAll.add(s1);
        this.providers.add(new ServiceProvider("p2",3,singleService));
        }

}
```