```
// Project 3 added requirement – Code PDF
//Ben Zaeske, Peng Jiang, Mikayla Pickett

//HardwareStore.java
import java.util.*;

public class HardwareStore {

	private List<Tool> inventory;
	private List<Customer> customers;
	private List<Rental> completedRentals;
	private List<Rental> activeRentals;
	private int currentDay;
	private int revenue;
	private int casualRentCount;
	private int businessRentCount;
	private int regularRentCount;
	private int rentCount;

	public HardwareStore(List<Tool> inventory) {
		this.inventory = inventory;
		this.customers = new ArrayList<Customer>();
		this.completedRentals = new ArrayList<Rental>();
		this.activeRentals = new ArrayList<Rental>();
		this.currentDay = 34;
		this.revenue = 0;
		this.casualRentCount = 0;
		this.businessRentCount = 0;
		this.regularRentCount = 0;
```

```
            this.rentCount = 0;
        }

//----------------Simulation methods----------------
//Methods related to running the hardware store 35-day simulation

        public void runSimulation() {
            //Run a loop and call doDay each time.
            while (this.currentDay >= 0) {
                this.doDay();
                this.currentDay -= 1;
            }
            //Do final prints
        }

        //Simulate one day
        public void doDay() {
            int dayRevenue = 0;
            //1. print day number
            int day = 35 - this.currentDay;
            System.out.println("\n----(Day number: " + day + ")----\n");
            //2. notify each observer (customers) of day change;
            dayRevenue += this.notifyCustomers();
            //3. print completed rentals
            this.printCompletedRentals();
            //4. print active rentals
            this.printActiveRentals();
            //5. print count and list of tools left in inventory
            this.printInventory();
```

```
                //6. print the current day's revenue (dayRevenue) and add dayRevenue to this.revenue
                System.out.println("\n----(Revenue)----");
                this.revenue += dayRevenue;
                System.out.println("Store profit today: $" + dayRevenue);
                System.out.println("Total store profit after " + day + " days : $" + this.revenue);
                //If currentDay is 0 -> print completed rentals overall and by customer, total money the
store made
                if (this.currentDay == 0) {
                        System.out.println("\n----(End of Simulation)----");
                        System.out.println("Total revenue earned over 35 days: $" + this.revenue);
                        System.out.println("Total Rentals: " + this.rentCount);
                        System.out.println("Rentals by Customer Type");
                        System.out.println("Business Customers: " + this.businessRentCount);
                        System.out.println("Casual Customers: " + this.casualRentCount);
                        System.out.println("Regular Customers: " + this.regularRentCount);
                }
        }

        //Returns a rental's tools (baseTools) to the store's inventory, and moves the rental from active
to completed
        public void completeRental(Rental rental) {
                for (Tool tool : rental.baseTools) {
                        this.inventory.add(tool);
                }
                this.activeRentals.remove(rental);
                this.completedRentals.add(rental);
        }

//----------------Observer pattern methods----------------
```

```
	//Add a customer. Our equivalent of adding an observer
	public void addCustomer(Customer customer) {
		this.customers.add(customer);
		customer.store = this;
	}


	//Our equivalent of notifyObservers()
	//customer.update() for all customers. They will handle if they can rent or not and will handle returning their rentals.
	//Add the returned rental to activeRentals if not null. keep track of the day's revenue along the way (new rentals)
	public int notifyCustomers() {
		int dayRevenue = 0;
		for (Customer customer : this.customers) {
			//Update each customer
			Rental newRental = customer.update();
			//Update active rentals and rental counts if the customer makes a new rental
			if (newRental != null) {
				this.activeRentals.add(newRental);
				this.rentCount += 1;
				switch(customer.type) {
					case "business":
						this.businessRentCount += 1;
					case "regular":
						this.regularRentCount += 1;
					case "casual":
						this.casualRentCount += 1;
				}
```

```
                    //Update the day's revenue
                    dayRevenue += newRental.getCost();
                }
            }
            return dayRevenue;
        }

//----------------Helper methods----------------
//Methods which hold logic that may need to be repeated multiple times.

        //Because each tool needs a unique number/combination of extras, each tool needs to be retrieved one at a time
        //This should take no parameter because it should only return 1 tool
        //Should handle removing tool from store inventory
        public Tool getRandomTool() {
            //https://www.geeksforgeeks.org/java-util-random-nextint-java/
            Random rand = new Random();
            int randIndex = rand.nextInt(this.inventory.size());
            Tool randTool = this.inventory.get(randIndex);
            this.inventory.remove(randIndex);
            return randTool;
        }

        //helper method that prints completed rentals
        public void printCompletedRentals() {
            System.out.println("\n----(Completed Rentals)----");
            if (this.completedRentals.size() == 0) {
                System.out.println("No completed rentals.");
                return;
```

```
        }
        System.out.println("Total completed: " + this.completedRentals.size());
        for (Rental rental : this.completedRentals) {
            rental.printRental();
        }
}

//helper method that prints active rentals
public void printActiveRentals() {
        System.out.println("\n----(Active Rentals)----");
        if (this.activeRentals.size() == 0) {
            System.out.println("No active rentals.");
            return;
        }
        System.out.println("Total active: " + this.activeRentals.size());
        for (Rental rental : this.activeRentals) {
            rental.printRental();
        }
}

//helper method that prints count and list of all tools in the inventory
public void printInventory() {
        System.out.println("\n----(Store Inventory)----");
        System.out.println("Total tools left in inventory: " + this.inventory.size());
        for (Tool tool : this.inventory) {
            System.out.println(tool.getDescription());
        }
}
```

```
	public int getInventorySize()
	{
		return this.inventory.size();
	}

}



//Simulation.java
import java.util.*;
import java.io.*;

public class Simulation {

	public static void main(String[] args) throws FileNotFoundException {
		//Change output to 'simulation.out': https://www.geeksforgeeks.org/redirecting-system-out-println-output-to-a-file-in-java/
		PrintStream o = new PrintStream(new File("simulation.out"));
		//Save console
      PrintStream console = System.out;
      System.setOut(o);

		//Generate the inventory
		List<Tool> inventory = generateInventory();
		//Instantiate the HardwareStore
		HardwareStore store = new HardwareStore(inventory);
		//Make customers and add them to the hardwareStore
		generateCustomers(store);
```

```
        //Run the simulation:
        store.runSimulation();
}

//Helper function for making tools
public static List<Tool> generateInventory() {
        List<Tool> inventory = new ArrayList<Tool>();
        ToolFactory paintingTools = new PaintingToolFactory();
        ToolFactory concreteTools = new ConcreteToolFactory();
        ToolFactory woodworkTools = new WoodworkToolFactory();
        ToolFactory yardworkTools = new YardworkToolFactory();
        ToolFactory plumbingTools = new PlumbingToolFactory();
        for (int i = 0; i < 5; i++) {
                inventory.add(paintingTools.getInstance());
                inventory.add(concreteTools.getInstance());
                inventory.add(woodworkTools.getInstance());
                inventory.add(yardworkTools.getInstance());
                if (i < 4) {
                        inventory.add(plumbingTools.getInstance());
                }
        }
        return inventory;
}

//Helper function for generating customers
public static void generateCustomers(HardwareStore store) {
        CustomerFactory businessFactory = new BusinessCustomerFactory();
        CustomerFactory casualFactory = new CasualCustomerFactory();
        CustomerFactory regularFactory = new RegularCustomerFactory();
```

```
                for (int i = 0; i < 5; i++) {
                        if (i < 2) {
                                store.addCustomer(businessFactory.getInstance());
                        }
                        store.addCustomer(casualFactory.getInstance());
                        store.addCustomer(regularFactory.getInstance());
                }
        }

}



//Tool.java
//not github
import java.util.*;

//----------------------------Basic Tool Objects----------------------------
//Tool interface. Has methods to get the cost and description of a Tool
public interface Tool {
        public int cost();
        public String getDescription();
}

//-------Concrete Tool implementations below--------
class PaintingTool implements Tool {
        public String name;
        public String type;
        protected int cost;
```

```
	public PaintingTool(String name) {
		this.name = name;
		this.type = "Painting Tool";
		this.cost = 5;
	}
	public int cost() {
		return this.cost;
	}
	public String getDescription() {
		return this.name;
	}
}

class ConcreteTool implements Tool {
	public String name;
	public String type;
	protected int cost;
	public ConcreteTool(String name) {
		this.name = name;
		this.type = "Concrete Tool";
		this.cost = 20;
	}
	public int cost() {
		return this.cost;
	}
	public String getDescription() {
		return this.name;
	}
}
```

```
class PlumbingTool implements Tool {
	public String name;
	public String type;
	protected int cost;
	public PlumbingTool(String name) {
		this.name = name;
		this.type = "Plumbing Tool";
		this.cost = 15;
	}
	public int cost() {
		return this.cost;
	}
	public String getDescription() {
		return this.name;
	}
}

class WoodworkTool implements Tool {
	public String name;
	public String type;
	protected int cost;
	public WoodworkTool(String name) {
		this.name = name;
		this.type = "Woodwork Tool";
		this.cost = 15;
	}
	public int cost() {
		return this.cost;
```

```
	}
	public String getDescription() {
		return this.name;
	}
}

class YardworkTool implements Tool {
	public String name;
	public String type;
	protected int cost;
	public YardworkTool(String name) {
		this.name = name;
		this.type = "Yardwork Tool";
		this.cost = 10;
	}
	public int cost() {
		return this.cost;
	}
	public String getDescription() {
		return this.name;
	}
}

//----------------------------Tool Decorator----------------------------
//The following website was referenced:
//https://www.journaldev.com/1540/decorator-design-pattern-in-java-example

//Abstract ToolDecorator class.
abstract class ToolDecorator implements Tool {
```

```
	protected Tool tool;
	public ToolDecorator(Tool tool) {
		this.tool = tool;
	}
}

//------Concrete Tool Decorators below------

class ExtensionCord extends ToolDecorator {
	public ExtensionCord(Tool tool)
	{
		super(tool);
	}

	public String getDescription()
	{
		return tool.getDescription() + " + Extension Cord";
	}

	public int cost()
	{
		return 1 + tool.cost();
	}
}

class AccessoryKit extends ToolDecorator{
	public AccessoryKit(Tool tool)
	{
		super(tool);
```

```
	}

	public String getDescription()
	{
		return tool.getDescription() + " + Accessory Kit";
	}

	public int cost()
	{
		return 2 + tool.cost();
	}
}

class ProtectiveGearPackage extends ToolDecorator{
	public ProtectiveGearPackage(Tool tool)
	{
		super(tool);
	}

	public String getDescription()
	{
		return tool.getDescription() + " + Protective Gear";
	}

	public int cost()
	{
		return 3 + tool.cost();
	}
}
```

```
//----------------------------Option Factories----------------------------
//Option factories are used to add one of the three options onto a tool

abstract class OptionFactory {
	public abstract Tool addOption(Tool tool);
}

class ExtensionCordFactory extends OptionFactory {
	public Tool addOption(Tool tool) {
		Tool wrappedTool = new ExtensionCord(tool);
		return wrappedTool;
	}
}

class AccessoryKitFactory extends OptionFactory {
	public Tool addOption(Tool tool) {
		Tool wrappedTool = new AccessoryKit(tool);
		return wrappedTool;
	}
}

class ProtectiveGearFactory extends OptionFactory {
	public Tool addOption(Tool tool) {
		Tool wrappedTool = new ProtectiveGearPackage(tool);
		return wrappedTool;
	}
}
```

```
//----------------------------Tool Factories----------------------------
/*Tool factories are used to generate tools. They are used in the initialization
 * phase of the simulation to generate the 24 tools in the store's inventory
 */

abstract class ToolFactory {
	int toolsMade;
	public ToolFactory() {
		this.toolsMade = 0;
	}
	public abstract Tool getInstance();
}

class PaintingToolFactory extends ToolFactory {
	@Override
	public Tool getInstance() {
		this.toolsMade += 1;
		return new PaintingTool("Painting Tool " + this.toolsMade);
	}
}

class ConcreteToolFactory extends ToolFactory {
	@Override
	public Tool getInstance() {
		this.toolsMade += 1;
		return new ConcreteTool("Concrete Tool " + this.toolsMade);
	}
}
```

```
class PlumbingToolFactory extends ToolFactory {
	@Override
	public Tool getInstance() {
		this.toolsMade += 1;
		return new PlumbingTool("Plumbing Tool " + this.toolsMade);
	}
}

class WoodworkToolFactory extends ToolFactory {
	@Override
	public Tool getInstance() {
		this.toolsMade += 1;
		return new WoodworkTool("Woodwork Tool " + this.toolsMade);
	}
}

class YardworkToolFactory extends ToolFactory {
	@Override
	public Tool getInstance() {
		this.toolsMade += 1;
		return new YardworkTool("Yardwork Tool " + this.toolsMade);
	}
}



//Customer.java
import java.util.*;

//----------------------------Customer class----------------------------
```

```
//This is our Observer class
public class Customer {

	public String name;
	//Casual, regular, or business
	public String type;
	//True if they have space to make an additional rental
	public boolean canRent;
	public List<Rental> rentals = new ArrayList<Rental>();
	public HardwareStore store;
	public RentAlgorithm rentAlgorithm;

	public Customer(String name, String type, RentAlgorithm rentAlgorithm) {
		this.name = name;
		this.type = type;
		this.rentAlgorithm = rentAlgorithm;
	}

	//Returns a rental's tools (baseTools) to the store's inventory and updates the store's completedRental list. Removes the rental from the store's activeRentals list
	public void completeRental(Rental rental) {
		this.rentals.remove(rental);
		this.store.completeRental(rental);
	}

	//Returns max number of tools customer can rent
	public int checkMaxTools()
	{
```

```
        int maxTools = 3;
        if(this.type == "casual")
        {
                maxTools = 2;
        }

        //Check how much space customer has left
        int remainingSpace = 3;
        for(Rental rental : this.rentals)
        {
                remainingSpace -= rental.tools.size();
        }

        //If customer has more space to rent tools
        if(remainingSpace > 0)
        {
                //maxTools changed to available space
                if(remainingSpace <= maxTools)
                {
                        maxTools = remainingSpace;
                }
        }
        else
        {
                maxTools = 0;
        }

        return maxTools;
}
```

```
	public boolean getRentalStatus(int maxTools)
	{
		//If there's enough items in the store and the customer has room for more tools, they canRent = true
		if(this.store.getInventorySize() >= maxTools && maxTools!=0)
		{
			return true;
		}

		return false;
	}

	//Observer update method. Returns a new rental object or null if it can't rent or if it isn't randomly chosen
	public Rental update() {
		//Decreasing remainingDays, returning any rentals if remainingDays == 0
		for(int i = 0; i < this.rentals.size(); i++)
		{
			Rental currentRental = this.rentals.get(i);
			currentRental.remainingDays -= 1;
			if(currentRental.remainingDays == 0)
			{
				this.completeRental(currentRental);
			}
		}

		//helper functions to handle renting logic
		int maxTools = checkMaxTools();
```

```
                this.canRent = getRentalStatus(maxTools);

                Random rand = new Random();
                int willRent = rand.nextInt(2);

                if (willRent > 0 && this.canRent)
                {
                        Rental newRental =  this.rentAlgorithm.rent(this.store, maxTools, this.name);
                        this.rentals.add(newRental);
                        return newRental;
                }

                return null;
        }

}

//----------------------------RentAlgorithm Strategy pattern----------------------------

abstract class RentAlgorithm {

        //Makes a new rental object and returns it, removing 1-maxTools of tools from the
hardwareStore.
        //Adds 0-6 options to each tool.
        //returns completed rental
        public abstract Rental rent(HardwareStore store, int maxTools, String customerName);

}
```

```
class CasualRentAlgorithm extends RentAlgorithm {

	public Rental rent(HardwareStore store, int maxTools, String customerName) {
		//1-2 tools for 1-2 nights

		Random rand = new Random();
		int numTools = rand.nextInt(maxTools) + 1;

		List<Tool> tools = new ArrayList<Tool>();
		List<Tool> baseTools = new ArrayList<Tool>();

		//Casual customer will rent 1-2 nights
		//nextInt(2) = [0,1]
		//+1 = [1,2]
		int numDays = rand.nextInt(2)+1;

		for(int i = 0; i < numTools; i++)
		{
			Tool temp = store.getRandomTool();

			//add tool without extras to baseTools
			baseTools.add(temp);

			//Adding random number of options and random options
			int numOptions = rand.nextInt(6);
			for(int j = 0; j < numOptions; j++)
			{
				int optionType = rand.nextInt(3);
				switch(optionType)
```

```
                    {
                        case 0:
                            temp = new ExtensionCordFactory().addOption(temp);
                            break;
                        case 1:
                            temp = new AccessoryKitFactory().addOption(temp);
                            break;
                        case 2:
                            temp = new ProtectiveGearFactory().addOption(temp);
                            break;
                    }
                }

                //add single tool with option info to tools
                tools.add(temp);

            }
            Rental newRental = new Rental(baseTools, tools, numDays, customerName);
            return newRental;
        }

}

class BusinessRentAlgorithm extends RentAlgorithm {

    public Rental rent(HardwareStore store, int maxTools, String customerName) {
        //Rent 3 tools for 7 days
        Random rand = new Random();
        int numTools = 3;
```

```
int numDays = 7;

List<Tool> tools = new ArrayList<Tool>();
List<Tool> baseTools = new ArrayList<Tool>();

for(int i = 0; i < numTools; i++)
{
	Tool temp = store.getRandomTool();

	//add tool without extras to baseTools
	baseTools.add(temp);

	//Adding random number of options and random options
	int numOptions = rand.nextInt(6);
	for(int j = 0; j < numOptions; j++)
	{
		int optionType = rand.nextInt(3);
		switch(optionType)
		{
			case 0:
				temp = new ExtensionCordFactory().addOption(temp);
				break;
			case 1:
				temp = new AccessoryKitFactory().addOption(temp);
				break;
			case 2:
				temp = new ProtectiveGearFactory().addOption(temp);
				break;
		}
```

```
                }

                //add single tool with option info to tools
                tools.add(temp);

            }
            Rental newRental = new Rental(baseTools, tools, numDays, customerName);
            return newRental;
        }

}

class RegularRentAlgorithm extends RentAlgorithm {

        public Rental rent(HardwareStore store, int maxTools, String customerName) {
            //1-3 tools for 3-5 nights
            Random rand = new Random();
            int numTools = rand.nextInt(maxTools) + 1;

            List<Tool> tools = new ArrayList<Tool>();
            List<Tool> baseTools = new ArrayList<Tool>();

            //Regular customer will rent 3-5 nights
            //nextInt((5-3)+1) = [0,1,2]
            //+3 = [3,4,5]
            int numDays = rand.nextInt((5-3)+1)+3;

            for(int i = 0; i < numTools; i++)
            {
```

```
            Tool temp = store.getRandomTool();

            //add tool without extras to baseTools
            baseTools.add(temp);

            //Adding random number of options and random options
            int numOptions = rand.nextInt(6);
            for(int j = 0; j < numOptions; j++)
            {
                int optionType = rand.nextInt(3);
                switch(optionType)
                {
                    case 0:
                        temp = new ExtensionCordFactory().addOption(temp);
                        break;
                    case 1:
                        temp = new AccessoryKitFactory().addOption(temp);
                        break;
                    case 2:
                        temp = new ProtectiveGearFactory().addOption(temp);
                        break;
                }
            }

            //add single tool with option info to tools
            tools.add(temp);

        }
        Rental newRental = new Rental(baseTools, tools, numDays, customerName);
```

```
                return newRental;
        }

}


//----------------------------Rental class----------------------------

class Rental {
        //Save the base tool types
        public List<Tool> baseTools;
        //The list of tools once options are added on
        public List<Tool> tools;
        private int days;
        public int remainingDays;
        private int cost;
        private String customerName;
        public int id;

        //a customer has a list of rentals, so a rental doesn't need to keep track of the customer it belongs to
        public Rental(List<Tool> baseTools, List<Tool> tools, int days, String customerName) {
                //tools without options
                this.baseTools = baseTools;
                this.tools = tools;
                this.remainingDays = days;
                this.days = days;
                //Set the cost of the rental
                int cost = 0;
```

```
		for (int i = 0; i < this.tools.size(); i++) {
			cost += tools.get(i).cost();
		}
		this.cost = cost;
		this.customerName = customerName;
		//this.id = id;
	}

	//Prints tools + options (tools) for which customer, for how many days, and at what cost
	public void printRental()
	{
		System.out.println(this.customerName + " rented the following tools for " + this.days + " days at a cost of $" + this.cost + ":");
		for(Tool t: tools)
		{
			System.out.println(t.getDescription());
		}
	}

	public int getCost() {
		return this.cost;
	}

}

//---------- Customer Factories ----------
//Customer factories are used at the start of the simulation to create 12 generic customer objects

abstract class CustomerFactory {
```

```
	int customersMade;
	public CustomerFactory() {
		this.customersMade = 0;
	}
	public abstract Customer getInstance();
}

class BusinessCustomerFactory extends CustomerFactory {

	@Override
	public Customer getInstance() {
		this.customersMade += 1;
		return new Customer("Business Customer " + (this.customersMade), "business", new
BusinessRentAlgorithm());
	}

}

class CasualCustomerFactory extends CustomerFactory {

	@Override
	public Customer getInstance() {
		this.customersMade += 1;
		return new Customer("Casual Customer " + (this.customersMade), "casual", new
CasualRentAlgorithm());
	}

}
```

```
class RegularCustomerFactory extends CustomerFactory {


	@Override

	public Customer getInstance() {

		this.customersMade += 1;

		return new Customer("Regular Customer " + (this.customersMade), "regular", new
RegularRentAlgorithm());

	}


}
```