Effective Java

Lab Instructions

SciSpike

Table of Contents

1 Create a ConfigDepot Class

Create a *ConfigDepot* class that provides a **Static Factory Method** like this:

```
public static Map<String, String> getConfig(String configFileName)
```

It should return a same map for the same input property file name. So you may call it repeatedly in your program. Internally, you will need to maintain a *Map<String, Map<String, String>>* that maps from a file name to a map of name value pairs (properties). The property file is assumed to be on the class paths.

To load a property file on the class paths, you may use:

```
InputStream is = ConfigDepot.class.getClassLoader().getResourceAsStream(configFileName);
```

To convert from *java.util.Properties* to *Map<String, String>*, you may use:

```
Map<String, String> rslt = new HashMap<String, String>((Map)properties);
```

The class may be used like the following:

```
Map<String, String> devConfig = ConfigDepot.getConfig("dev.properties");
System.out.printf("Development mode configurations:\n\t%s\n", devConfig);
System.out.printf("Development mode host:\n\t%s\n", devConfig.get("host"));
Map<String, String> prodConfig = ConfigDepot.getConfig("prod.properties");
System.out.printf("Production mode configurations:\n\t%s\n", prodConfig);
System.out.printf("Production mode host:\n\t%s\n", prodConfig.get("host"));
```

The property files should be put on your project's classpath, e.g. under your java project's src folder.

The above test should produce results similar to these:

```
Development mode configurations:
    {host=localhost, debug=true}
Development mode host:
    localhost
Production mode configurations:
    {host=rest.scispike.com, debug=false}
Production mode host:
    rest.scispike.com
```

2 Use Builder Pattern to Create Objects

In chapter 1, we demonstrated the use of the Builder pattern with a *Customer* class. Use it as an example and write a *Config* class that may take the following parameters as input to its constructor:

```
String protocol;
String host;
String port;
String contextRoot;
boolean useCache;
```

Of all these potential inputs, let's say only the *contextRoot* is mandatory.

Override the *toString* method to produce something like:

```
http://www.scispike.com:80/scispike/?useCache=true
```

A sample test should be able to construct the *Config* object like this:

```
Config config =
    new Config.Builder("scispike").host("www.scispike.com").useCache(true).build();
System.out.printf("URL for %s: %s\n", config.getHost(), config);
```

The printout should be something like this:

```
URL for www.scispike.com: http://www.scispike.com:80/scispike/?useCache=true
```

3 Create Singleton ConfigDepot

1. Use one the traditional singleton pattern methods to turn ConfigDepot in Lab1 into a singleton.
2. Use the single-element enum method to turn ConfigDepot in Lab1 into a singleton.

4 Create an Employee Class

Create an *Employee* class that contains at least these fields:

```
private String employeeNumber;
private String firstName;
private String lastName;
private Date dob;
private double salary;
```

Follow the instructions in the slides to override the *equals*, *hashCode*, and *toString* methods.

5 Create an Immutable Class

Create an *immutable* class *Payment* that has the following two fields:

```
private final float amount;
private final Date paymentDate;
```

In addition to provide the usual methods for an immutable class, it should support a method *adjustPaymentAmount* that takes a *float amount* as input and return a new immutable *Payment* instance.

6 Create an Encryption Strategy

Imagine you are asked to write a program that reads text files and as each line is read, encrypt it. Suppose one encryption method is Rot 13 (replace each letter with one that is 13 letters after it in the alphabet), and another is public key based. Of course, you can write two different programs to do this. But then you would have to duplicate the code that reads the file, etc. What if later you are asked to support yet another encryption strategy?

What you can do is to separate out the encryption part as a *strategy* by defining an *EncryptionStrategy* interface:

```
public interface EncryptionStrategy {
    String encrypt(String clearText);
}
```

You may then provide various implementation of the interface using a concrete encryption strategy, for example Rot13 encryption.

The file reading method can then take an *EncryptionStrategy* type as input, and as each line is read, the strategy's *encrypt* method is called.

Implement the above design idea:

1. Define the *EncryptionStrategy* interface
2. Implement the interface with a *Rot13Encryption* class. The core of the Rot13 encryption is, for any *char c*:

```
if (c >= 'a' && c <= 'm' || c >= 'A' && c <= 'M')
        c += 13;
else if (c >= 'n' && c <= 'z' || c >= 'N' && c <= 'Z')
        c -= 13;
```

3. Write a *SecureFileReader* class that reads a text file and then encrypt each line. It just has one method:

```
public void readAndEncrypt(String filename, EncryptionStrategy encryptor)
```

4. Write *main()* method for *SecureFileReader* to test it.

Note: with Rot13, encryption and decryption works the same way, that is, Rot13(Rot13(x)) = x.

7 Create a Decorator

Suppose you have implemented several encryption strategies in the previous lab, but you have no idea how each one is performing in comparison to one another. So you want to add some time measurement code to the encryption strategy. But that would require changing each existing encryption strategy class. Instead, define a wrapper/decorator class that wraps the *EncryptionStrategy* class and add the new code logic in the wrapper.

Implement the above design idea by creating an *InstrumentedRot13* class. It should implement the *EncryptionStrategy* interface and take an *EncryptionStrategy* type as an input for its constructor.

8 Create a Generic Stack

Create a generic *Stack<E>* class that exposes the following public API:

```
public void push(E obj)
public void pushAll(Collection<? extends E> src)
public E pop()
public void popAll(Collection<? super E> dst)
public boolean isEmpty()
public int size()
public String toString()
```

9 Create an enum Class for Eurozone Countries

The following table shows the current Eurozone countries, their population, and GDPs:

State	Population	GDP (million USD)
Austria	8,404,252	384,908
Belgium	10,918,405	468,522
Cyprus	838,896	24,910
Estonia	1,340,194	19,120
Finland	5,375,276	237,512
France	65,075,373	2,649,390
Germany	81,751,602	3,330,032
Greece	11,325,897	329,924
Ireland	4,480,858	227,193

State	Population	GDP (million USD)
Italy	60,626,442	2,112,780
Luxembourg	511,840	52,449
Malta	417,617	7,449
Netherlands	16,655,799	792,128
Portugal	10,636,979	227,676
Slovakia	5,435,273	87,642
Slovenia	2,050,189	48,477
Spain	47,190,493	1,460,250

Create an enum type EurozoneCountry with the above info. Each enum (country) should take a population and GDP (in millions) as input. It should provide the following API:

```
public long getPopulation()
public long getGDPInMillions()
public long getGDPPerCapita()
public static long getEurozonePopulation()
public static long getEurozoneGDPInMillions()
```

10 Create an Order System

Imagine you are asked to build a restaurant order system with the following data model:

1. An Order contains a customer name, a list of OrderItems, and a Table:

```
private String customerName;
private List<OrderItem> orderItems;
private Table table;
```

2. An OrderItem contains a MenuItem and quantity:

```
private MenuItem item;
private int quantity;
```

3. A MenuItem contains a name and a price:

```
private String name;
private double price;
```

4. A Table contains a table number and the number of guests it can sit:

```
private int number;
private int numberOfSeats;
```

Every day there are thousands of orders taken (think of Starbucks). But if you think about it, a restaurant's menu items are fixed (at least not changed often) and the tables don't change that often either. So if we just blindly create instances for the MenuItem and Table classes for every order, a lot of memory space is wasted for storing this repeated static information.

To overcome this, we should make MenuItem and Table classes immutable and the constructors private and provide only Static Factory Methods to get their instances:

```
// If table with the right table number exists, return it; otherwise, create one
// and save it in a map keyed off table number
public static Table getTable(int number) {...}

// If MenuItem with the right name exists, return it; otherwise, create one
// and save it in a map keyed off menu item name
public static MenuItem getMenuItem(String name) {...}
```

For the Order class, define the following methods:

```
public Order(String customerName)
public Order(String customerName, List<OrderItem> items, Table table)
public String getCustomerName()
public void setCustomerName(String customerName) // Make sure to add argument validation
public List<OrderItem> getItems()
public void changeOrder(List<OrderItem> items)
public Table getTable()
public void setTable(Table table)
public void addOrder(OrderItem oi)  // Make sure to add argument validation
public double getTotalPrice()
@Override public String toString()
```

To practive input validation, add some checking of the argument in the methods setCustomer and addOrder.

For OrderItem, implement this API:

```
public OrderItem(MenuItem item, int quantity)
public MenuItem getItem()
public void setItem(MenuItem item)
public int getQuantity()
public void setQuantity(int quantity)
@Override public String toString()
```

For MenuItem, implement this API:

```
private MenuItem(String name, double price)
public static MenuItem getMenuItem(String name) // Make sure to add argument validation
public String getName()
public double getPrice()
@Override public String toString()
```

For Table, implement this API:

```
private Table(int number, int numberOfSeats)
public static Table getTable(int number)
public int getNumber()
public int getNumberOfSeats()
@Override public String toString()
```

Write a Test program to test out your Order system.

11 Create a Stock Portfolio

Create a stock portfolio application with the following classes:

1. An *enum TradeAction* class that contains two enums: *BUY* and *SALE*.
2. A *Trade* class that represents one trading and contains these fields:

```
private String symbol;
private TradeAction action; // buy or sell
private BigDecimal price; // Use BigDecimal for currency
private int numOfShares;
private Date tradeDate;
```

3. A *Holding* class that represents one particular stock position that you hold. It contains these fields:

```
private String symbol;
private long numOfShares;
private BigDecimal cost;
```

4. A Portfolio class that represents all the holdings you have. It contains a HashMap of stock symbol to Holding's mapping:

```
private HashMap<String, Holding> holdings;
```

The *Trade* class is an immutable class that exposes these public API:

```
public Trade(String symbol, TradeAction action, BigDecimal price, int numOfShares, Date tradeDate)
public String getSymbol()
public TradeAction getAction()
public BigDecimal getPrice()
public int getNumOfShares()
public Date getTradeDate()
```

The Holding class exposes these API:

```
public Holding(String symbol) // need to initialize number of shares and cost
public String getSymbol()
public long getNumOfShares()
public BigDecimal getCost()
// Increase the number of shares and recalculate cost
public void addToHolding(long numOfShares, BigDecimal sharePrice)
// Reduce the number of shares and recalculate cost
public void removeFromHolding(long numOfShares, BigDecimal sharePrice)
// Before a sell-trade can be carried out, call this method to determine
```

```
// if there are enough shares.
public boolean canSell(long numOfSharesToSell)
@Override public String toString()
```

The Portfolio class exposes these API:

```
public Portfolio()
public int getNumberOfHoldings()
public String[] getHoldingSymbols()
public Holding getHolding(String symbol)
// Use the information in a trade to update the corresponding holding in the portfolio.
// Note: a trade can a buy or a sell.
public void modifyPortfolio(Trade trade)
// A client should call this method to determine if the portfolio has the corresponding holding
// and enough shares
public boolean canSell(String symbol, long numOfSharesToSell)
@Override public String toString()
```

Write a Test program to exercise your portfolio. One use case may look like the following:

```
Portfolio p = new Portfolio();
Trade t = new Trade("AAPL", TradeAction.BUY, new BigDecimal(550.00), 100, new Date());
p.modifyPortfolio(t);
System.out.println(p);

if (p.canSell("AAPL", 50)) {
    t = new Trade("AAPL", TradeAction.SALE, new BigDecimal(580.00), 50, new Date());
    p.modifyPortfolio(t);
    System.out.println(p);
}

t = new Trade("INTC", TradeAction.BUY, new BigDecimal(21.00), 100, new Date());
p.modifyPortfolio(t);
System.out.println(p);

System.out.println("Number of holdings in the portfolio: " + p.getNumberOfHoldings());

String[] symbols = p.getHoldingSymbols();

for (String s : symbols) {
    System.out.printf("%s: %s\n", s, p.getHolding(s));
}
```

12 Exception Chaining

In Lab 1, we asked you to write a ConfigDepot class that provided a static factory method to retrieve a configuration, using a config file name as id.

Since the code inevitably will read from a file, a lower-level IOException may throw if the file is not found, or not readable.

Let's pretend your *getConfig* static factory method is a higher level abstraction. It should throw an exception appropriate for its level when the lower level IOException occurrs.

Define a *ConfigNotAvailableException* that extends *Exception* and exposes these API:

```
// This constructor should pass the "cause" to its parent
public ConfigNotAvailableException(String configName, Throwable cause)
// The message should be constructed with the configName passed in in the constructor.
public String getMessage()
```