SOLID Coding Challenges

# Coding Challenges

Use Java and an Integrated Development Environment (STS/Eclipse, IntelliJ…) to complete the following coding exercises, based on SOLID principles. You may use books, notes and the web to research your solution, however you should not copy solutions directly from any source.

In cases where you need to display values to the screen, you may use System.out.println() to display the values to console.

## 01 Single Responsibility Principle

Suppose that you were working with the following class:

**public class** DataLoaderAndFormatter {

**public** String loadData() {

*/\* Code that gets the original data;*

*in the current implementation, it always returns the same String of data \*/*

**return "-Washington:1789 -Adams:1797 -Jefferson:1801"**;

}

**public** String formatData(String unformattedData) {

*/\* Formats the given data;*

*in the current implementation it just replaces whitespaces " " with new lines "\n" \*/*

**return** unformattedData.replace(**" "**, **"\n"**);

}

}



Why does this class violate the Single Responsibility Principle? How would you refactor it to solve this problem?

## 02 Single Responsibility Principle

Suppose that you were working with the following class:

**public class** CustomerAccountService {

**public void** activateAccount() {

*// Logic that activates a new account goes here*

}

**public void** sendMessage() {

*// Logic that sends a message to the customer goes here*

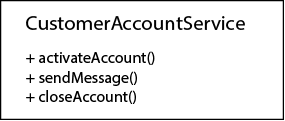
}

**public void** closeAccount() {

*// Logic that closes an account goes here*

}

}



How could you improve this class so it doesn't violate the Single Responsibility Principle?

## 03 Open / Closed Principle

Suppose that you were working on an 'Account' class representing a bank account. The class has fields to track the year when the account was created, the current balance of the account, and the current interest rate applied to that account. This interest rate is calculated using a 'calculateInterestRate' function, which follows a few logical rules based on the age of the account and its balance to find out what interest rate should be paid to the owner.

**public class** Account {

**private int yearOfCreation**;

**private double balance**;

**private double interestRate**;

**public double** getInterestRate() {

**return interestRate**;

}

**public** Account(**int** yearOfCreation, **double** balance) {

**this**.**yearOfCreation** = yearOfCreation;

**this**.**balance** = balance;

**this**.**interestRate** = **this**.calculateInterestRate(yearOfCreation, balance);

}

**private double** calculateInterestRate(**int** yearOfCreation, **double** balance) {

**if**(balance > 1000000) {

**return** 2.1;

} **else if**(yearOfCreation < 2010) {

**return** 0.1;

} **else** {

**return** 0.01;

}

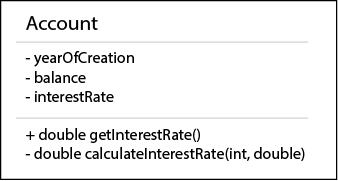
*// What if we wanted to modify it*

*// to give an interest rate of 1.4% for clients with a balance > 500,000*

*// without modifying anything else or having to recompile this class?*

}

}



What if we wanted to include a new rule to calculate interests rates, giving a 1.4% interest rate to clients whose balance were higher than $500,000?

If this class were following the Open for Extension / Closed for Modification principle, extending this class with a new logical rule wouldn't imply recompiling the class or modifying anything else.

How would you modify this class to comply with this principle, so new logical rules could be added without modifying anything else?

## 04 Open / Closed Principle

Suppose that you wanted to keep track of “to-do” tasks using a 'TaskList' class, containing a list of 'Task' objects, providing methods to check whether there are any more pending tasks, and to return the next task on the list.

In the following implementation, getting the next task on the list would return the oldest element from the list (the first element that was added.) This is a FIFO approach (First In, First Out.)

**TaskList.java:**

**import** java.util.ArrayList;

**import** java.util.List;

**public class** TaskList {

**private** ArrayList<Task> **list**;

*/\*\**

*\* Constructor*

*\** ***@param list*** *an array filled with Tasks*

*\*/*

**public** TaskList(List<Task> list){

**this**.**list** = **new** ArrayList<Task>(list);

}

**public boolean** hasMoreTasks() {

**return** (**this**.**list** != **null** && (!**this**.**list**.isEmpty()));

}

**public** Task getNextTask() {

**if**(**this**.hasMoreTasks()) {

**return this**.**list**.remove(0);

*// Returns the first element added to the list (oldest element)*

*// What if we wanted to get the most recent item added to the list?*

} **else** {

**return null**;

}

}

}

**Task.java:**

**public class Task {**

**private** String **description**;

**public** Task(String description) {

**this**.**description** = description;

}

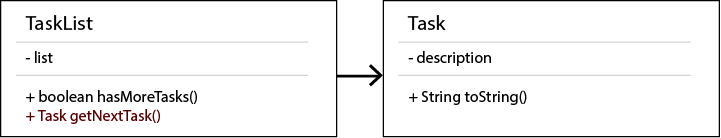
**public** String toString() {

**return this**.**description**;

}

}

But, what if we wanted to get the most recent task when asking for the next task (LIFO, Last In, First Out)? How could we refactor all this so we could have classes that worked in a FIFO approach, and other classes that worked in a LIFO approach, without having to modify the base TaskList class every time we changed the implementation of the method to get the next task in the list?



## 05 Liskov's Substitution Principle

Imagine we had created a class that represents and solves quadratic equations:

The 'solve' method obtains the two possible roots out of quadratic equations using the formula:

This would be the implementation of such class (and for the sake of simplicity, let's assume that we don't care about imaginary results:)

**QuadraticEquation.java:**

**public class** QuadraticEquation {

**protected int a**; *// ax2, quadratic coefficient*

**protected int b**; *// bx, linear coefficient*

**protected int c**; *// c, constant*

**public int** getA() {

**return a**;

}

**public void** setA(**int** a) {

**this**.**a** = a;

}

**public int** getB() {

**return b**;

}

**public void** setB(**int** b) {

**this**.**b** = b;

}

**public int** getC() {

**return c**;

}

**public void** setC(**int** c) {

**this**.**c** = c;

}

**public** QuadraticEquation() {

}

**public double**[] solve() {

**double** squareRootCalculation = Math.*sqrt*(Math.*pow*(**b**, 2) - 4 \* **a** \* **c**);

**double** root1 = (-**b** + squareRootCalculation) / (2 \* **a**);

**double** root2 = (-**b** - squareRootCalculation) / (2 \* **a**);

**double**[] solutions = **new double**[] {root1, root2};

**return** solutions;

}

}

Now, let's assume that we wanted to represent linear equations as well. A linear equation is a special case of a quadratic equation where the 'a' coefficient has a value of zero, being of the form:

Since a linear equation IS A quadratic equation, let's use inheritance, and make our LinearEquation class extend a QuadraticEquation:

**LinearEquation.java:**

*// 0x2 + bx + c = 0*

**public class** LinearEquation **extends** QuadraticEquation {

@Override

**public double**[] solve() {

**double** root = ((**double**)(-**c**) / (**double**)(**b**));

**double**[] solutions = **new double**[] {root};

**return** solutions;

}

}

Now, let's put our code to use in an example like the following:

**Main.java:**

**public class** Main {

**public static void** main(String[] args) {

QuadraticEquation equation = **new** QuadraticEquation();

equation.setA(2);

equation.setB(10);

equation.setC(5);

**double**[] solutions = equation.solve();

System.***out***.println(**"Solutions for the equation "**

+ equation.getA() + **"x2 + "** + equation.getB() + **"x + "** + equation.getC());

**for**(**int** i = 0; i < solutions.**length**; i++) {

System.***out***.println(solutions[i]);

}

*// We should be able to substitue the previous base class (supertype) with a subtype class and obtain the same results...*

QuadraticEquation linearEquation = **new** LinearEquation();

linearEquation.setA(2);

linearEquation.setB(10);

linearEquation.setC(5);

**double**[] solutionsForSameParameters = linearEquation.solve();

System.***out***.println(**"Solutions for the (same?) equation "**

+ linearEquation.getA() + **"x2 + "** + linearEquation.getB() + **"x + "** + linearEquation.getC());

**for**(**int** i = 0; i < solutionsForSameParameters.**length**; i++) {

System.***out***.println(solutionsForSameParameters[i]);

}

}

}

The code printed by running the previous example is:

Solutions for the equation 2x2 + 10x + 5

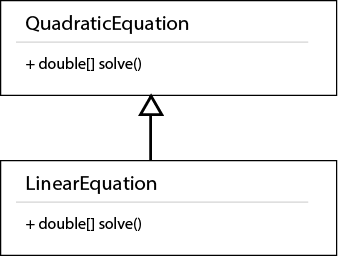
-0.5635083268962915

-4.436491673103708

Solutions for the (same?) equation 2x2 + 10x + 5

-0.5

Why does this violate Liskov's Substitution Principle? Why do we get an unexpected result when we replace an element from the superclass (QuadraticEquation) using an element from the subclass instead (LinearEquation)? How can we fix our implementation so we won't have this problem again?



## 06 Liskov's Substitution Principle

Let's create a class that represents a Triangle and that allows to calculate its perimeter:



**Triangle.java:**

**public class Triangle {**

**protected int sideOneLength**;

**protected int sideTwoLength**;

**protected int sideThreeLength**;

**public int** getSideOneLength() {

**return sideOneLength**;

}

**public void** setSideOneLength(**int** sideOneLength) {

**this**.**sideOneLength** = sideOneLength;

}

**public int** getSideTwoLength() {

**return sideTwoLength**;

}

**public void** setSideTwoLength(**int** sideTwoLength) {

**this**.**sideTwoLength** = sideTwoLength;

}

**public int** getSideThreeLength() {

**return sideThreeLength**;

}

**public void** setSideThreeLength(**int** sideThreeLength) {

**this**.**sideThreeLength** = sideThreeLength;

}

**public** Triangle() {

}

**public int** getPerimeter() {

**return sideOneLength** + **sideTwoLength** + **sideThreeLength**;

}

}

Now let's create another class representing and equilateral triangle where all sides measure the same. An Equilateral Triangle IS A Triangle, so we decide to inherit from our Triangle base class when creating it.



**EquilateralTriangle.java:**

**public class** EquilateralTriangle **extends** Triangle {

@Override

**public int** getPerimeter() {

**return** 3 \* **this**.**sideOneLength**;

}

}

Let's try to use these classes to represent the same triangle, checking whether Liskov's Replacement Principle has been followed:

**Main.java:**

**public class** Main {

**public static void** main(String[] args) {

**int** sideOneLength = 10;

**int** sideTwoLength = 5;

**int** sideThreeLength = 7;

Triangle myTriangle = **new** Triangle();

myTriangle.setSideOneLength(sideOneLength);

myTriangle.setSideTwoLength(sideTwoLength);

myTriangle.setSideThreeLength(sideThreeLength);

**int** myTrianglePerimeter = myTriangle.getPerimeter();

System.***out***.println(**"The perimeter of my triangle is: "** + myTrianglePerimeter);

*// An EquilateralTriangle IS A Triangle. Can we get the same results if we substitute it?*

Triangle equilateralTriangle = **new** EquilateralTriangle();

equilateralTriangle.setSideOneLength(sideOneLength);

equilateralTriangle.setSideTwoLength(sideTwoLength);

equilateralTriangle.setSideThreeLength(sideThreeLength);

**int** equilateralTrianglePerimeter = equilateralTriangle.getPerimeter();

System.***out***.println(**"The perimeter of my equilateral triangle is: "** + equilateralTrianglePerimeter);

}

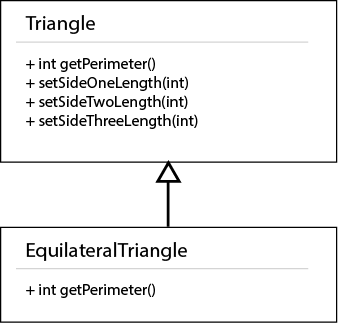
}

Running the previous code, for the only defined triangle, prints:

The perimeter of my triangle is: 22

The perimeter of my equilateral triangle is: 30

Why didn't we follow Liskov's Replacement Principle? How can we refactor our code so it prevents this problem?



## 07 Interface Segregation Principle

Let's assume that we have a class, 'DefaultHeaderBanner', part of a simple graphic interface, whose responsibility is to print a text-based banner with default values. These values are based on a user's preferences, and we want to display the full user name surrounded by some simple text graphics:

**DefaultHeaderBanner.java:**

**public class** DefaultHeaderBanner {

**public void** displayFullUserName() {

IUserPreferences defaultUserPreferences = **new** DefaultUserPreferences();

String fullUserName =

defaultUserPreferences.getFirstName() + **" "** +

defaultUserPreferences.getMiddleName() + **" "** +

defaultUserPreferences.getLastName();

System.***out***.println(**"--------------------------------"**);

System.***out***.println(**"User --> "** + fullUserName);

System.***out***.println(**"--------------------------------"**);

}

}

To get the information about the user name, we need to access information about this user's preferences. The following interface defines how to access these preferences:

**IuserPreferences.java:**

**public interface** IUserPreferences {

**public** String getFirstName();

**public** String getMiddleName();

**public** String getLastName();

**public** String getPassword();

**public** String[] getSecurityQuestionAndAnswer();

**public** String getFavoriteURL();

**public** String[] getBookmarkList();

}

Since our default header banner class is interested in default values related to the user name, we need to handle a concrete implementation of these preferences that contains the default values that we want to access. That is why the 'DefaultUserPreferences' class was created:

**DefaultUserPreferences.java:**

**public class DefaultUserPreferences implements IUserPreferences {**

@Override

**public** String getFirstName() {

**return "John"**;

}

@Override

**public** String getMiddleName() {

**return "X."**;

}

@Override

**public** String getLastName() {

**return "Doe"**;

}

@Override

**public** String getPassword() {

**return null**;

}

@Override

**public** String[] getSecurityQuestionAndAnswer() {

**return new** String[0];

}

@Override

**public** String getFavoriteURL() {

**return null**;

}

@Override

**public** String[] getBookmarkList() {

**return new** String[0];

}

}

We could then run this code in a Main class like this:

**Main.java:**

**public class** Main {

**public static void** main(String[] args) {

DefaultHeaderBanner defaultHeader = **new** DefaultHeaderBanner();

defaultHeader.displayFullUserName();

}

}

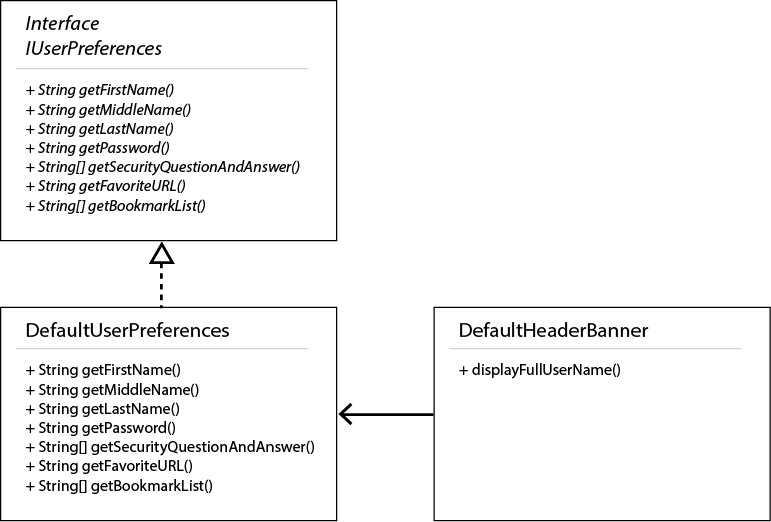
And this would print as a result:

--------------------------------

User --> John X. Doe

--------------------------------

Even when this code works, why doesn't this architecture follow the Interface Segregation Principle? How could we improve this interface, taking into account that our default header banner only needs to access name-related preferences?



## 08 Interface Segregation Principle

In the following application, there's a system in place to send notifications to the users via different platforms. In the current implementation we have defined Email notifications and SMS notifications (text messages.)

Since more implementations could be developed in the future, all kinds of notifications should implement a common interface, for sustainability and compatibility reasons.

Here's the code of such interface and their implementations, plus an example of how they would be used:

**INotification.java:**

**public interface INotification {**

**public** String getMessage();

**public** String getSubject();

**public void** sendToEmail(String emailAddress);

**public void** sendToPhone(String phoneNumber);

}

**EmailNotification.java:**

**public class EmailNotification implements INotification {**

**private** String **message**;

**private** String **subject**;

**public** EmailNotification(String message, String subject) {

**this**.**message** = message;

**this**.**subject** = subject;

}

@Override

**public** String getMessage() {

**return this**.**message**;

}

@Override

**public** String getSubject() {

**return this**.**subject**;

}

@Override

**public void** sendToEmail(String emailAddress) {

*/\* Code to send email would go here \*/*

System.***out***.println(**"Sending email..."**);

}

@Override

**public void** sendToPhone(String phoneNumber) {

*/\* Not implemented, not applicable \*/*

}

}

**SMSNotification.java:**

**public class SMSNotification implements INotification {**

**private** String **message**;

**public** SMSNotification(String message) {

**this**.**message** = message;

}

@Override

**public** String getMessage() {

**return this**.**message**;

}

@Override

**public** String getSubject() {

*/\* Not implemented, not applicable \*/*

**return null**;

}

@Override

**public void** sendToEmail(String emailAddress) {

*/\* Not implemented, not applicable \*/*

}

@Override

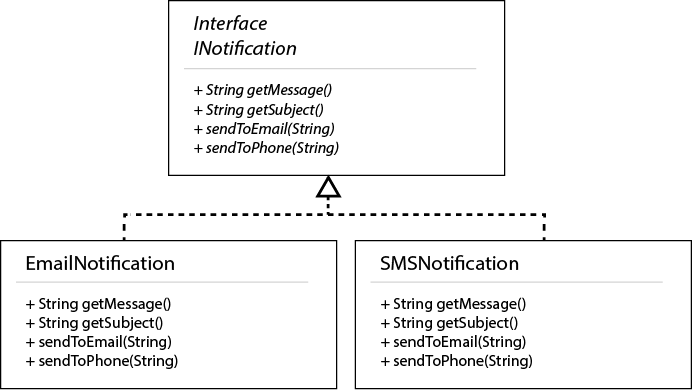
**public void** sendToPhone(String phoneNumber) {

*/\* Code to send SMS text message would go here \*/*

System.***out***.println(**"Sending text message to phone..."**);

}

}

****

What can be improved about this interface, so it better follows the Interface Segregation Principle?

## 09 Dependency Inversion Principle

In this example, we have a high-level class called 'DataLoader' whose purpose is to get back data using some database connection.

In our current case, we've been told that our client wants to use a MySQL database, and hence, we're giving our 'DataLoader' class a specific 'MySQLDatabaseConnection'.

Here's how the current system is implemented:

**DataLoader.java:**

**public class DataLoader {**

**public** DataLoader() {

}

**public** String loadData() {

MySQLDatabaseConnection connection = **new** MySQLDatabaseConnection();

**return** connection.getData();

}

}

**MySQLDatabaseConnection.java:**

**public class MySQLDatabaseConnection {**

**public** MySQLDatabaseConnection() {

*// A MySQL connection would be established here*

}

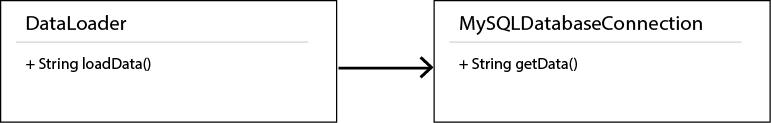
**public** String getData() {

*// Sample data, hardcoded*

**return "-Washington:1789 -Adams:1797 -Jefferson:1801"**;

}

}

****

But, what would happen if our client suddenly decided to use a different database engine, like Oracle or PostgreSQL? How could we apply the Dependency Inversion Principle to our design?

## 10 Dependency Inversion Principle

In this application, we're trying to create a financial system that would suggest our users what investments are worth selecting. We're modelling potential investments as objects characterized by a descriptive text, a risk level ranging 0 to 100 (where 0 is no risk, and 100 is high risk,) and a return ratio (as a decimal number, representing potential return on investment:)

**Investment.java**

**public class Investment {**

**private** String **description**;

*// Risk level - 0 = best, 100 = worst*

**private int riskLevel**;

*// Return on investment (percent)*

**private double returnRatio**;

**public** String getDescription() {

**return description**;

}

**public int** getRiskLevel() {

**return riskLevel**;

}

**public double** getReturnRatio() {

**return returnRatio**;

}

**public** Investment(String description, **int** riskLevel, **double** returnRatio) {

**this**.**description** = description;

**this**.**riskLevel** = riskLevel;

**this**.**returnRatio** = returnRatio;

}

}

Then, we want to create a Graphic User Interface (GUI) class that will display recommended investments on the screen. The first prototype of this class:

* Retrieves all potential investments from a database.
* Applies business logic rules to find all the recommended investments.
* Presents a list of recommended investments on the screen.

Here's what its code looks like:

**RecommendedInvestmentDisplayGUI.java:**

**public class** RecommendedInvestmentDisplayGUI {

**public void** displayRecommendedInvestments() {

ArrayList<Investment> recommendedInvestments = **this**.findAllRecommendedInvestments();

System.***out***.println(**"Recommended investments"**);

System.***out***.println(**"-----------------------"**);

**for**(**int** i = 0; i < recommendedInvestments.size(); i++) {

System.***out***.println(recommendedInvestments.get(i).getDescription());

}

}

**public** ArrayList<Investment> findAllRecommendedInvestments() {

Investment[] allInvestments = **this**.getAllInvestments();

ArrayList<Investment> recommendedInvestments = **new** ArrayList<Investment>();

**for**(**int** i = 0; i < allInvestments.**length**; i++) {

Investment currentInvestment = allInvestments[i];

**if**(**this**.isInvestmentRecommended(currentInvestment)) {

recommendedInvestments.add(currentInvestment);

}

}

**return** recommendedInvestments;

}

**public boolean** isInvestmentRecommended(Investment investment) {

**int** riskLevel = investment.getRiskLevel();

**double** returnRatio = investment.getReturnRatio();

**if**(riskLevel < 75 && returnRatio > 150) {

*// High risk, highly profitable investment*

**return true**;

}

**if**(riskLevel < 10 && returnRatio > 6) {

*// Low risk, decent profit investment*

**return true**;

}

*// Anything else isn't recommended with the current rules*

**return false**;

}

**public** Investment[] getAllInvestments() {

*// Would connect to a database and get a list of all possible investments*

**return new** Investment[] {

**new** Investment(**"Risky business"**, 100, 200),

**new** Investment(**"Startup"**, 50, 175),

**new** Investment(**"Savings account"**, 1, 2),

**new** Investment(**"Stocks account"**, 8, 7),

**new** Investment(**"High-frequency trading"**, 40, 20)

};

}

}

If we ran our code in a Main class like this:

**public class** Main {

**public static void** main(String[] args) {

RecommendedInvestmentDisplayGUI recommendedGUI = **new** RecommendedInvestmentDisplayGUI();

recommendedGUI.displayRecommendedInvestments();

}

}

This is what the system would output:

Recommended investments

-----------------------

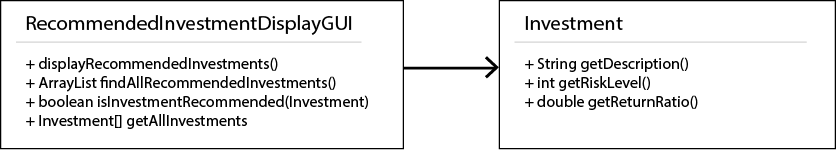
Startup

Stocks account

Analyzing the RecommendedInvestmentDisplayGUI class, we can see that it combines:

* Data access layer code (connecting to a database and retrieving the data.)
* Service layer code (business rules to determine what investments are recommended.)
* Presentation layer code (printing out the recommended investments.)

This isn't following the Single Responsibility Principle. Moreover, changing to a different database engine, or modifying / including new business rules to find other profitable investments, would imply recompiling the whole class, which doesn't follow the Open / Closed principle as well.



How would you redesign this class so it would follow the previously mentioned two principles AND also the Dependency Inversion Principle? Try to structure your code in meaningful packages as well.

## 11 Don't Repeat Yourself

Let's take an example from a previous case (Open/Closed Principle 1) where the interest rate assigned to existing Accounts was calculated using InterestRules. We were defining a specific interest rate for accounts with a balance over a million, and a different interest rate for accounts with a balance over half a million. This is the code of a suggested architecture for that:

**public** InterestCalculator() {

**this**.**interestRules** = **new** ArrayList<>();

*// How to eliminate code repetition in these rules?*

**this**.addRule(**new** InterestRuleOverMillion());

**this**.addRule(**new** InterestRuleOverHalfMillion());

}

And the key piece of code inside those two rules would be as follows:

**InterestRuleOverMillion.java:**

@Override

**public boolean** isRuleApplicable(**int** yearOfCreation, **double** balance) {

**return** (balance > 1000000);

}

**InterestRuleOverHalfMillion.java:**

@Override

**public boolean** isRuleApplicable(**int** yearOfCreation, **double** balance) {

**return** (balance > 500000);

}

Is it possible to eliminate code repetition on those two logical rules, whyle preserving the Open/Close principle? (This is a simple case, but imagine the benefit of that if the business logic calculations inside those rules were more complex.)

## 12 Don't Repeat Yourself

The following application is a temperature alert service that will send alert messages if the temperature surpasses certain thresholds. A temperature value is provided in Fahrenheit, and then it is compared with a threshold value in Celsius.

**TemperatureAlertService.java:**

**public class** TemperatureAlertService {

**public boolean** checkIsHighTemperature(**double** temperatureInFahrenheit) {

**double** temperatureInCelsius = ((temperatureInFahrenheit - 32) \* (5 / 9.0));

**if**(temperatureInCelsius > 100) {

**this**.sendAlert(**"Temperature alert "** + temperatureInCelsius, **"alert.service@example.com"**);

**return true**;

} **else** {

**return false**;

}

}

**public boolean** checkIsLowTemperature(**double** temperatureInFahrenheit) {

**double** temperatureInCelsius = ((temperatureInFahrenheit - 32) \* (5 / 9.0));

**if**(temperatureInCelsius < 0) {

**this**.sendAlert(**"Temperature alert "** + temperatureInCelsius, **"alert.service@example.com"**);

**return true**;

} **else** {

**return false**;

}

}

**private void** sendAlert(String message, String toAddress) {

*// It would use some email service to send the alert*

System.***out***.println(**"Sending alert..."**);

System.***out***.println(**"Message: "** + message);

System.***out***.println(**"To Address: "** + toAddress);

}

}

How would you refactor this class to apply the DRY principle?