BSc (Hons) in Information Technology

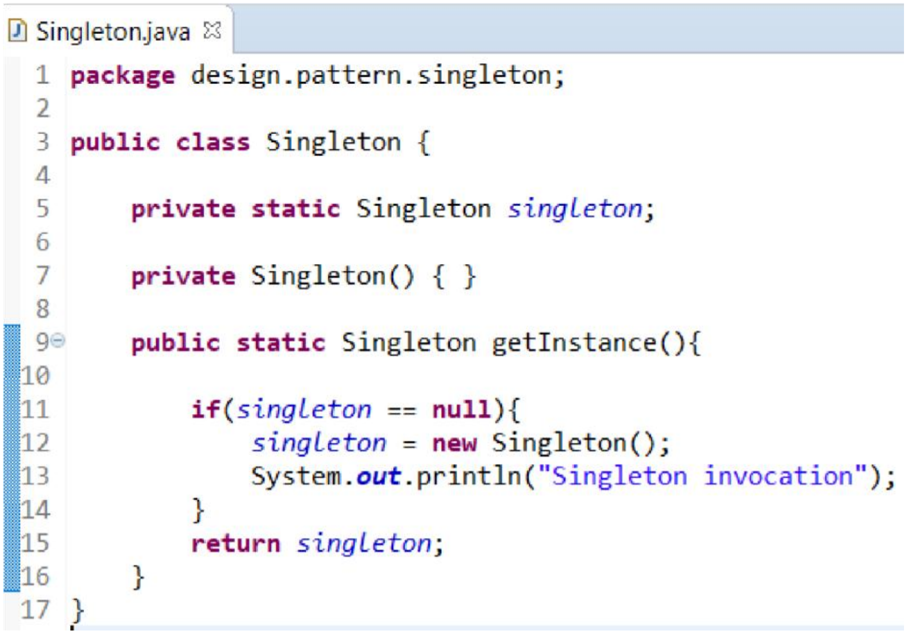
Year 2

Lab Exercise – Design Patterns

IT2030 – Object Oriented Programming Semester 1, 2020

Exercise 01 - Singleton

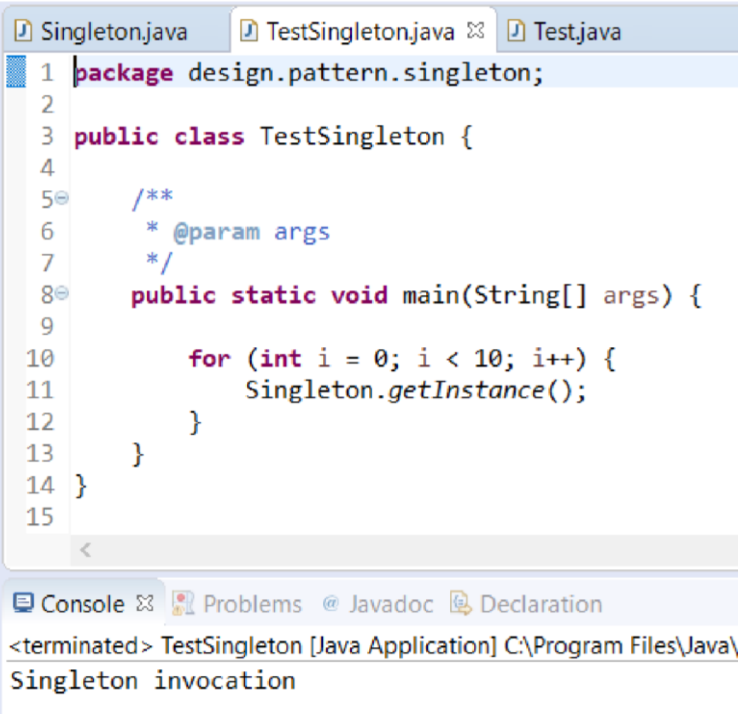
Implement the following code and check the output.



Singleton.java

```
package design.pattern.singleton;

public class Singleton {

    private static Singleton singleton;

    private Singleton() { }

    public static Singleton getInstance(){

        if(singleton == null){
            singleton = new Singleton();
            System.out.println("Singleton invocation");
        }
        return singleton;
    }
}
```

Run your program as follows.



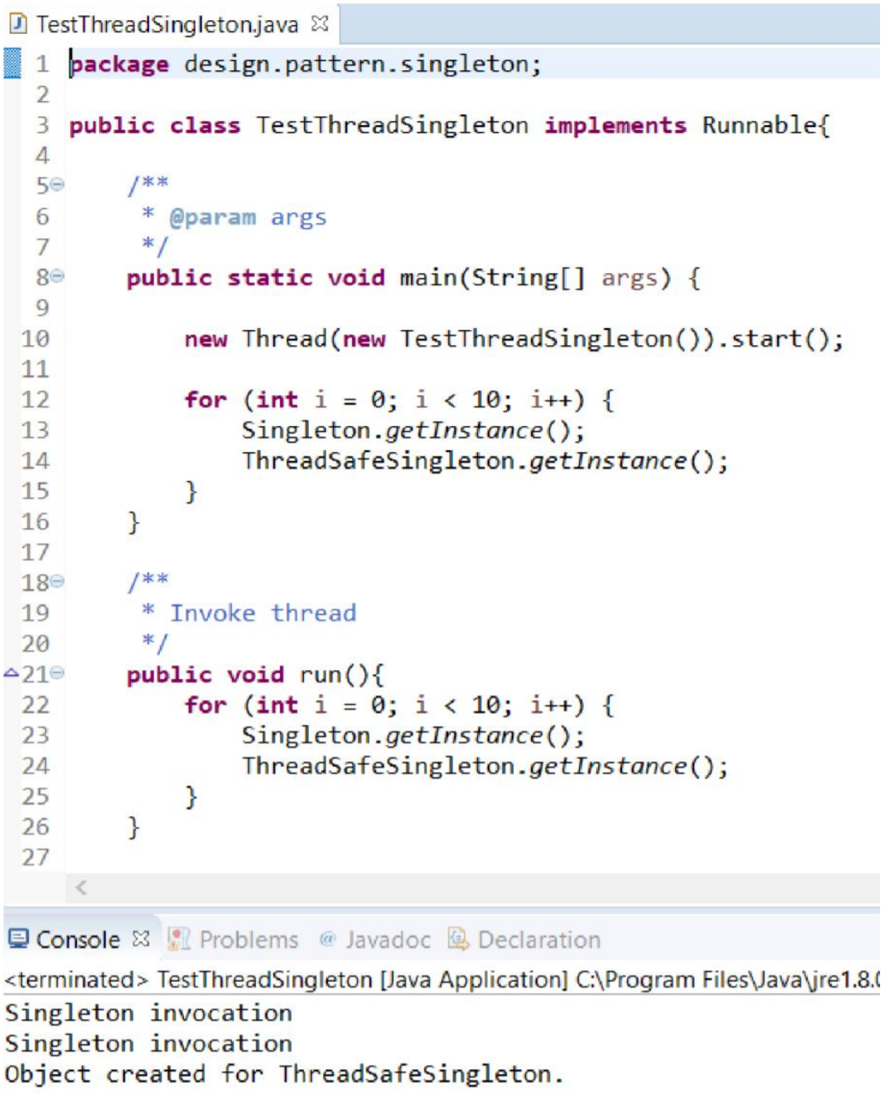
```
package design.pattern.singleton;

public class TestSingleton {

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

        for (int i = 0; i < 10; i++) {
            Singleton.getInstance();
        }
    }
}
```

```
<terminated> TestSingleton [Java Application] C:\Program Files\Java\
Singleton invocation
```

Ensure it creates only one instance even though you invoke it in the loop of 10 times.

Exercise 02 – Thread-safe Singleton

Now **modify the above program** to support **Thread safe** manner. You should implement synchronized blocks for the implementation. Now you should display output as follows.



```java
package design.pattern.singleton;

public class TestThreadSingleton implements Runnable{

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

        new Thread(new TestThreadSingleton()).start();

        for (int i = 0; i < 10; i++) {
            Singleton.getInstance();
            ThreadSafeSingleton.getInstance();
        }
    }

    /**
     * Invoke thread
     */
    public void run(){
        for (int i = 0; i < 10; i++) {
            Singleton.getInstance();
            ThreadSafeSingleton.getInstance();
        }
    }

```

```
<terminated> TestThreadSingleton [Java Application] C:\Program Files\Java\jre1.8.(
Singleton invocation
Singleton invocation
Object created for ThreadSafeSingleton.
```

Exercise 03 – Abstract Factory Pattern

1. Create an Interface called **Shape** with a method signature **draw()**
2. Implement 3-4 Concrete Classes of Shape
    a. Create classes for **Square**, **Circle**, **Triangle**, **Rectangle**, etc.
    b. Implement the **draw()** method
    e.g. Print the name of the shape inside the draw method of each class

3. Create a **ShapeFactory** class

a. Add a method called **getShape()** that accepts a **String** as a parameter and returns a **Shape**

b. Implement **getShape()** method to create the concrete shapes

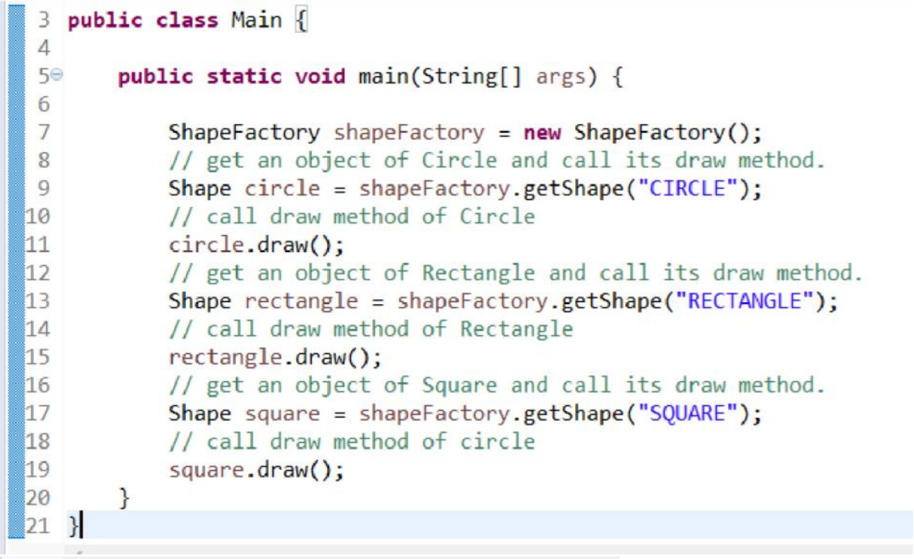
e.g. Check if the parameter is “SQUARE” and create an instance of Square class and return it

4. Create a Test class to test out the pattern functionality

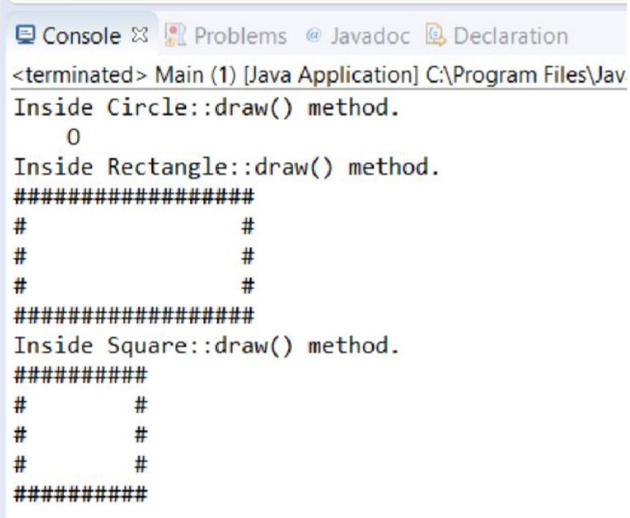
```
ShapeFactory shapeFactory = new ShapeFactory();
// get an object of Circle and call its draw method.
Shape shape1 = shapeFactory.getShape("SQUARE");
// call draw method of Circle
shape1.draw();
```

5. Complete the rest of the parts in design You should display the following output.

6. Create a Test class to test out the pattern functionality as below



```java
public class Main {

    public static void main(String[] args) {

        ShapeFactory shapeFactory = new ShapeFactory();
        // get an object of Circle and call its draw method.
        Shape circle = shapeFactory.getShape("CIRCLE");
        // call draw method of Circle
        circle.draw();
        // get an object of Rectangle and call its draw method.
        Shape rectangle = shapeFactory.getShape("RECTANGLE");
        // call draw method of Rectangle
        rectangle.draw();
        // get an object of Square and call its draw method.
        Shape square = shapeFactory.getShape("SQUARE");
        // call draw method of circle
        square.draw();
    }
}
```

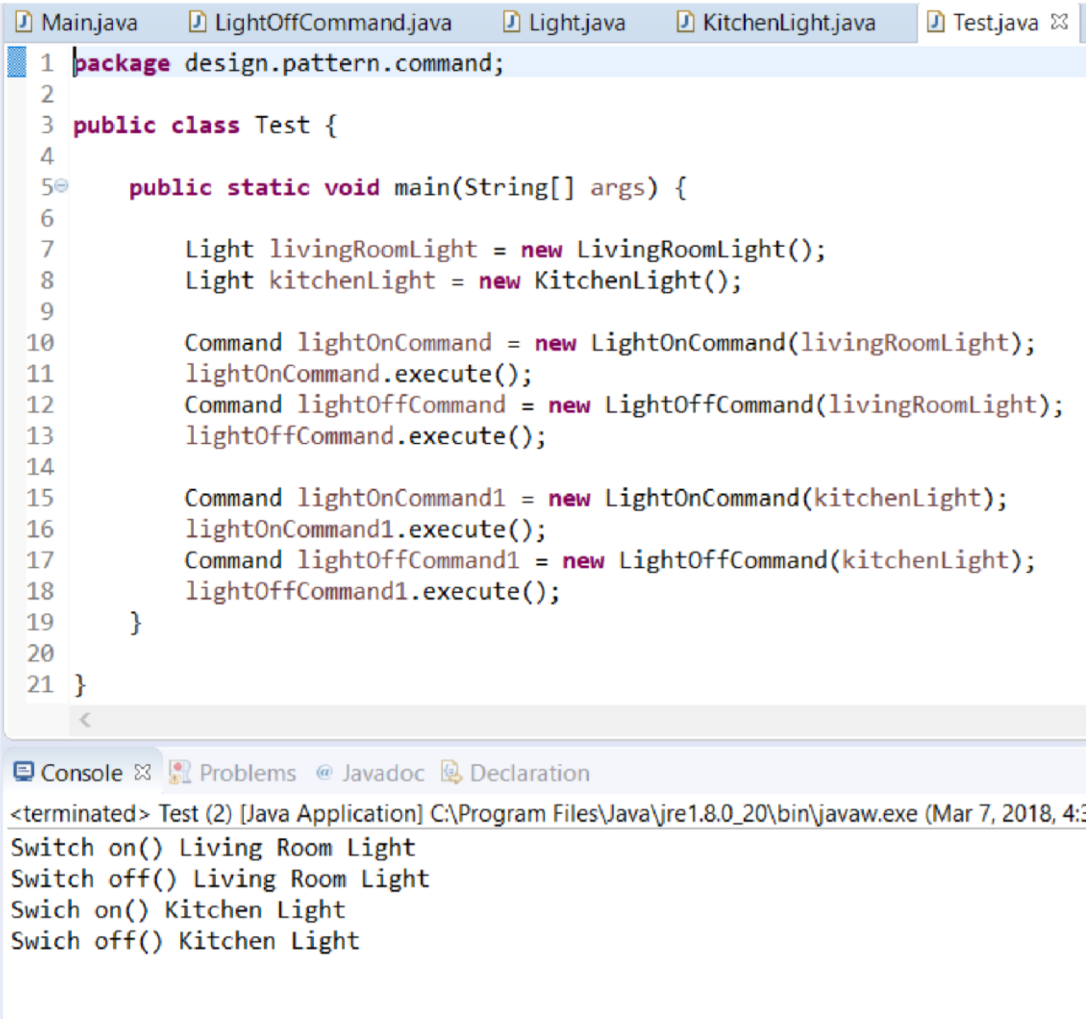


```
<terminated> Main (1) [Java Application] C:\Program Files\Jav
Inside Circle::draw() method.
    O
Inside Rectangle::draw() method.
##################
#                #
#                #
#                #
##################
Inside Square::draw() method.
##########
#        #
#        #
#        #
##########
```

Exercise 04 – Command Pattern

1. Create an Interface called **Command** with a method signature **execute()**
2. Create an Interface called **Light** with method signatures **on()** and **off()**
3. Implement 2 concrete classes named (**KitchenLight**, **LivingRoomLight**) and use **Light** interface with overriding methods **on()** and **off()** in each class.
4. Similarly use **Command interface** and implement 2 concrete classes named (**LightOnCommand**, **LightOffCommand**) and override **execute()** methods in each class.

5. Create **Test** class as below to check each light **on** and **off** commands with respect to the provided location.
6. Method **execute()** will run the given object for command class as below. Follow all above steps and execute relevant method.
7. Run this **Test** class and check the output should be as below.



```java
package design.pattern.command;

public class Test {

    public static void main(String[] args) {

        Light livingRoomLight = new LivingRoomLight();
        Light kitchenLight = new KitchenLight();

        Command lightOnCommand = new LightOnCommand(livingRoomLight);
        lightOnCommand.execute();
        Command lightOffCommand = new LightOffCommand(livingRoomLight);
        lightOffCommand.execute();

        Command lightOnCommand1 = new LightOnCommand(kitchenLight);
        lightOnCommand1.execute();
        Command lightOffCommand1 = new LightOffCommand(kitchenLight);
        lightOffCommand1.execute();
    }

}
```

```
<terminated> Test (2) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (Mar 7, 2018, 4:
Switch on() Living Room Light
Switch off() Living Room Light
Swich on() Kitchen Light
Swich off() Kitchen Light
```

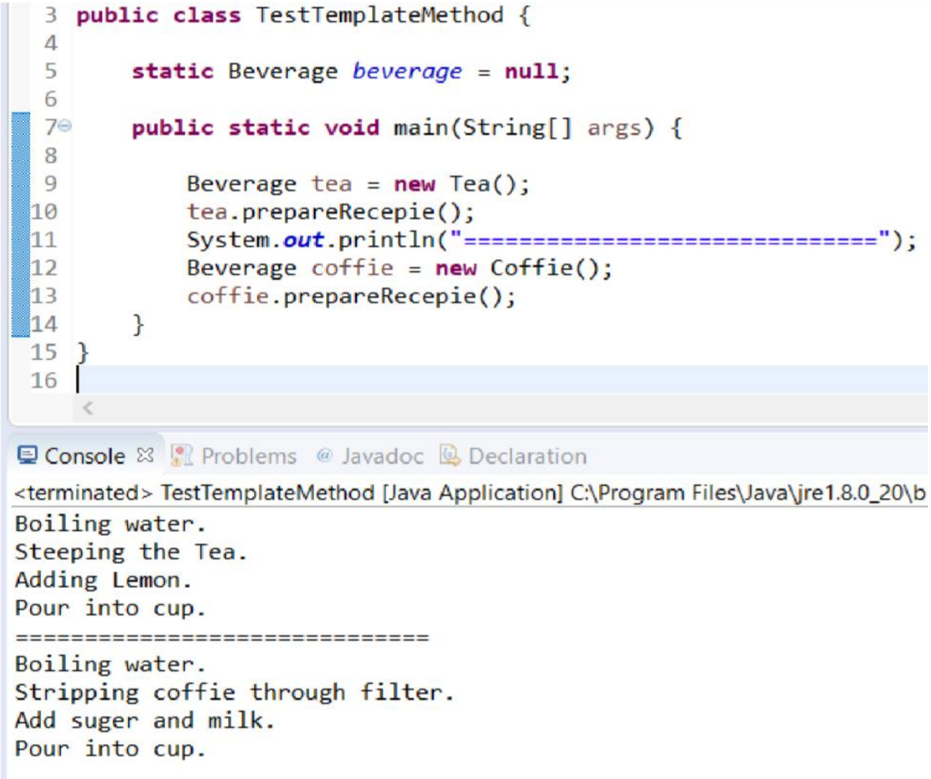
Exercise 05 – Template-method pattern

1. Create an abstract class called **Beverages** and extends that class using **Tea** and **Coffee** concrete classes

2. Within **Tea** and **Coffee** classes override both abstract methods **addCondiments()** and **brew()**
3. Now modify the **Beverage** class to implement **boilWater()** and **pourInCup()** methods. As per the below.

```
abstract void brew();

abstract void addCondiments();

void boilWater(){
    System.out.println("Boiling water.");
}

void pourInCup(){
    System.out.println("Pour into cup.");
}
```

4. Now you should impose the order of execution of these methods as below. This order of execution **should not be changed** implicitly or explicitly in any of these sub classes and it should work as life cycle methods.
5. Your modification should satisfy **step 4**
6. Now Implement **Test class** to test **above template method pattern** and you should be able to display the output below.



```
public class TestTemplateMethod {

    static Beverage beverage = null;

    public static void main(String[] args) {

        Beverage tea = new Tea();
        tea.prepareRecepie();
        System.out.println("==============================");
        Beverage coffie = new Coffie();
        coffie.prepareRecepie();
    }
}
```

```
<terminated> TestTemplateMethod [Java Application] C:\Program Files\Java\jre1.8.0_20\b
Boiling water.
Steeping the Tea.
Adding Lemon.
Pour into cup.
==============================
Boiling water.
Stripping coffie through filter.
Add suger and milk.
Pour into cup.
```

BSc (Hons) in Information Technology

Year 2

Lab Exercise – Design Patterns

IT2030 – Object Oriented Programming Semester 1, 2020

Exercise 06 – Builder Pattern

1. Create a **Query** class with SELECT, FROM WHERE and ORDER BY as properties
   a. Implement a method to print the complete query

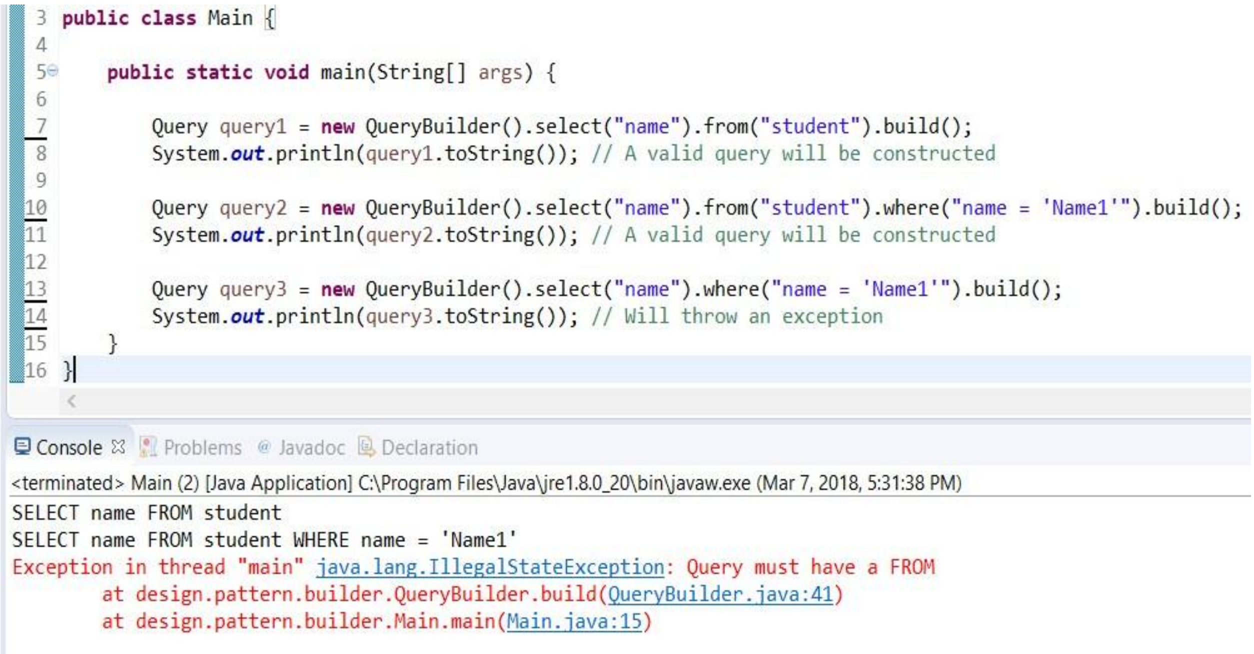
2. Create a **QueryBuilder** class with a property to hold a **Query** object
   a. Create the Query object inside the constructor of **QueryBuilder**
   b. Implement methods to set SELECT, FROM WHERE and ORDER to the Query object
   c. Each method should return a **QueryBuilder** object

3. Implement a method called **build()** in **QueryBuilder** that returns the **Query** object
   a. Check if the Query contains at least SELECT and FROM properties, if not it is not a valid query and prevent building the query by throwing an exception

4. Create a Test class to test out the pattern functionality

```
Query query1 = new QueryBuilder().select("name").from("student").build();
System.out.println(query1.toString()); // A valid query will be constructed
Query query2 = new QueryBuilder().select("name").from("student").where("name = 
'Name1'").build();
System.out.println(query2.toString()); // A valid query will be constructed
Query query3 = new QueryBuilder().select("name").where("name = 'Name1'").build();
System.out.println(query3.toString()); // Will throw an exception
```

5. You should display the following outputs. If you missed key word of the query, you should throw an exception as below.

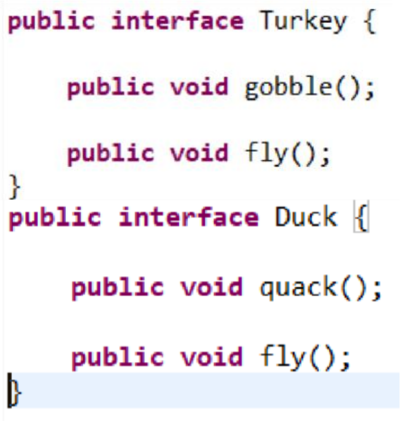


```
public class Main {

    public static void main(String[] args) {

        Query query1 = new QueryBuilder().select("name").from("student").build();
        System.out.println(query1.toString()); // A valid query will be constructed

        Query query2 = new QueryBuilder().select("name").from("student").where("name = 'Name1'").build();
        System.out.println(query2.toString()); // A valid query will be constructed

        Query query3 = new QueryBuilder().select("name").where("name = 'Name1'").build();
        System.out.println(query3.toString()); // Will throw an exception
    }
}
```

```
<terminated> Main (2) [Java Application] C:\Program Files\Java\jre1.8.0_20\bin\javaw.exe (Mar 7, 2018, 5:31:38 PM)
SELECT name FROM student
SELECT name FROM student WHERE name = 'Name1'
Exception in thread "main" java.lang.IllegalStateException: Query must have a FROM
	at design.pattern.builder.QueryBuilder.build(QueryBuilder.java:41)
	at design.pattern.builder.Main.main(Main.java:15)
```

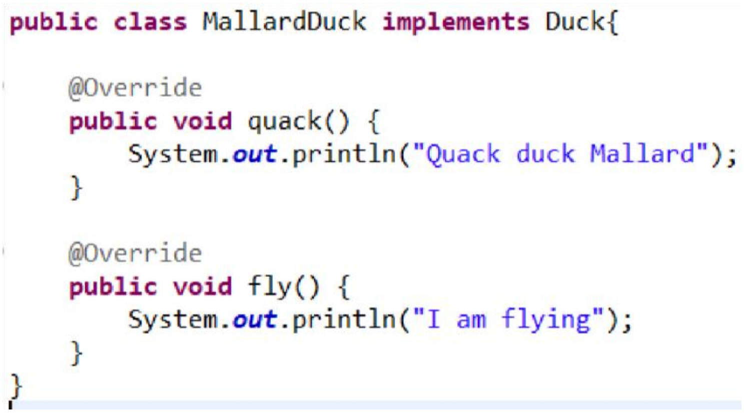
Exercise 07 – Adapter Pattern

1. Create two separate interfaces for **Duck** and **Turkey** as follows.



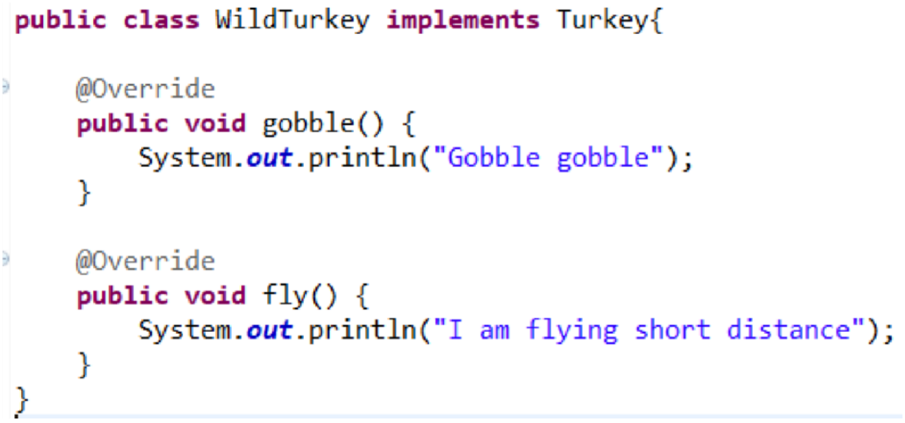
```
public interface Turkey {

    public void gobble();

    public void fly();
}
public interface Duck {

    public void quack();

    public void fly();
}
```

2. Then create concrete class **MallardDuck** with implementing **Duck interface** as follows.



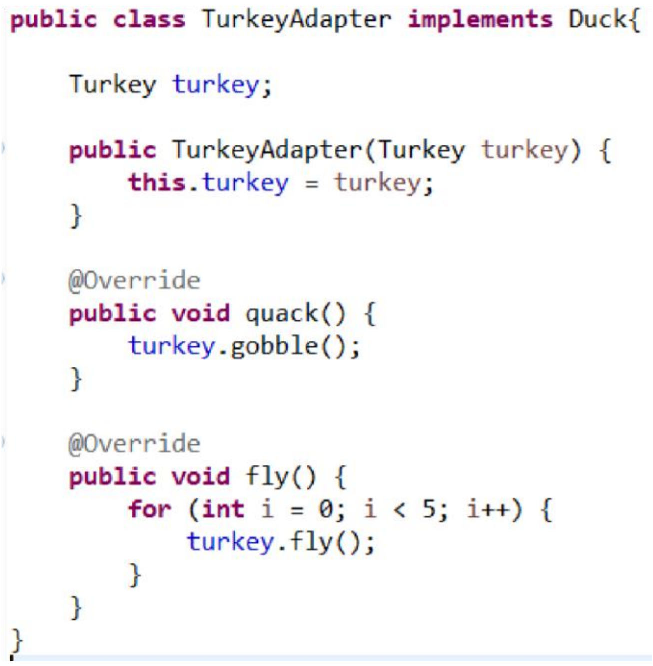
```
public class MallardDuck implements Duck{

    @Override
    public void quack() {
        System.out.println("Quack duck Mallard");
    }

    @Override
    public void fly() {
        System.out.println("I am flying");
    }
}
```

3. Similarly create another concrete class for **WildTurkey** with implementing **Turkey** interface as well.



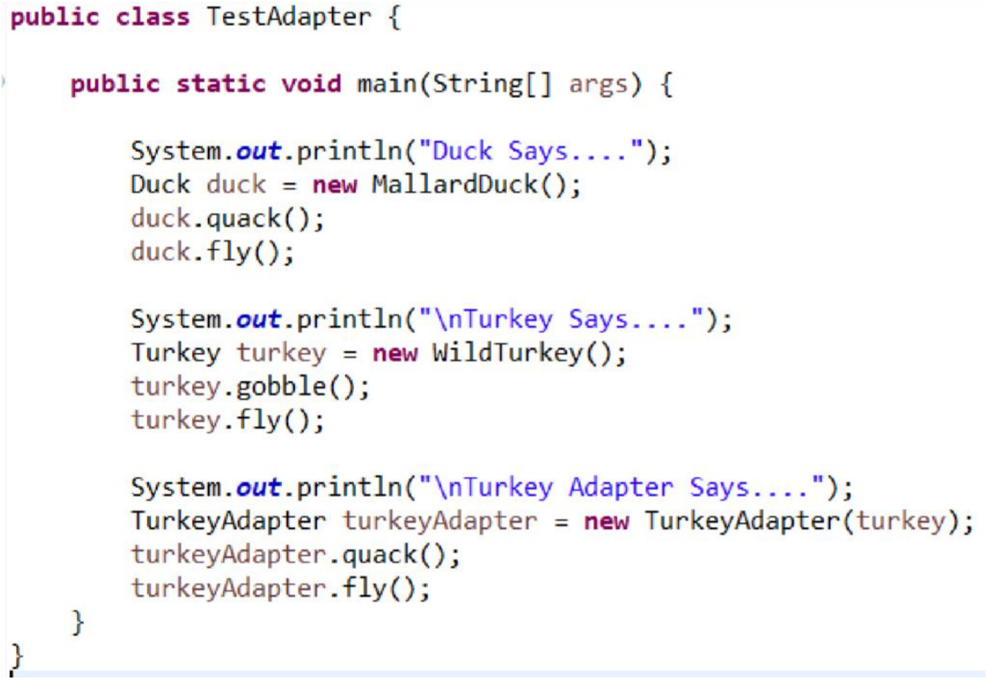
```
public class WildTurkey implements Turkey{

    @Override
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    @Override
    public void fly() {
        System.out.println("I am flying short distance");
    }
}
```

4. Now implement the Adapter for **Duck** and **Turkey** as follows and override both **quack()** and **fly()** method as below.

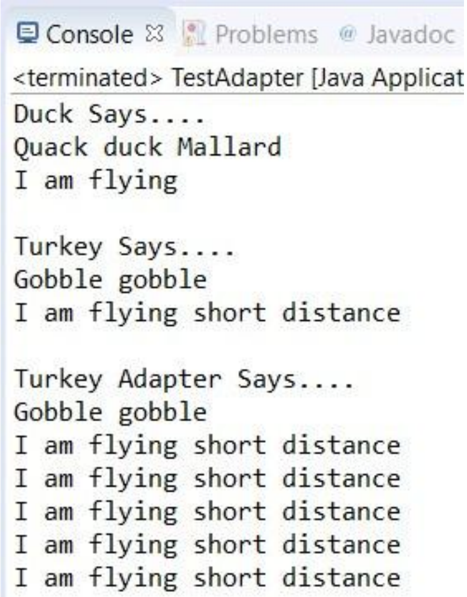


```java
public class TurkeyAdapter implements Duck{

    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    @Override
    public void quack() {
        turkey.gobble();
    }

    @Override
    public void fly() {
        for (int i = 0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

5. Now try out your Adapter implementation as per the **TestAdapter**. Check the output of Adapter pattern behavior.



```java
public class TestAdapter {

    public static void main(String[] args) {

        System.out.println("Duck Says....");
        Duck duck = new MallardDuck();
        duck.quack();
        duck.fly();

        System.out.println("\nTurkey Says....");
        Turkey turkey = new WildTurkey();
        turkey.gobble();
        turkey.fly();

        System.out.println("\nTurkey Adapter Says....");
        TurkeyAdapter turkeyAdapter = new TurkeyAdapter(turkey);
        turkeyAdapter.quack();
        turkeyAdapter.fly();
    }
}
```

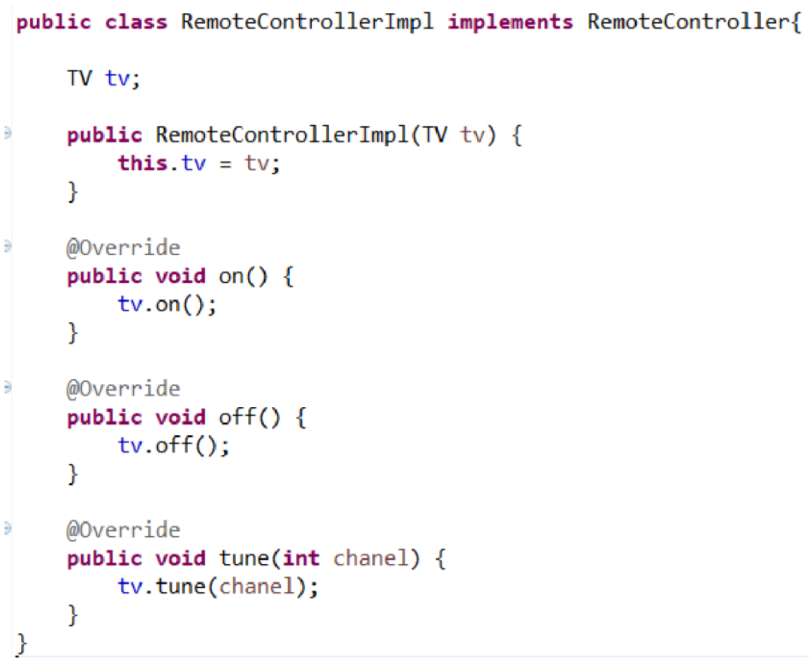


```
<terminated> TestAdapter [Java Applicat
Duck Says....
Quack duck Mallard
I am flying

Turkey Says....
Gobble gobble
I am flying short distance

Turkey Adapter Says....
Gobble gobble
I am flying short distance
I am flying short distance
I am flying short distance
I am flying short distance
I am flying short distance
```

Exercise 08 – Bridge Pattern

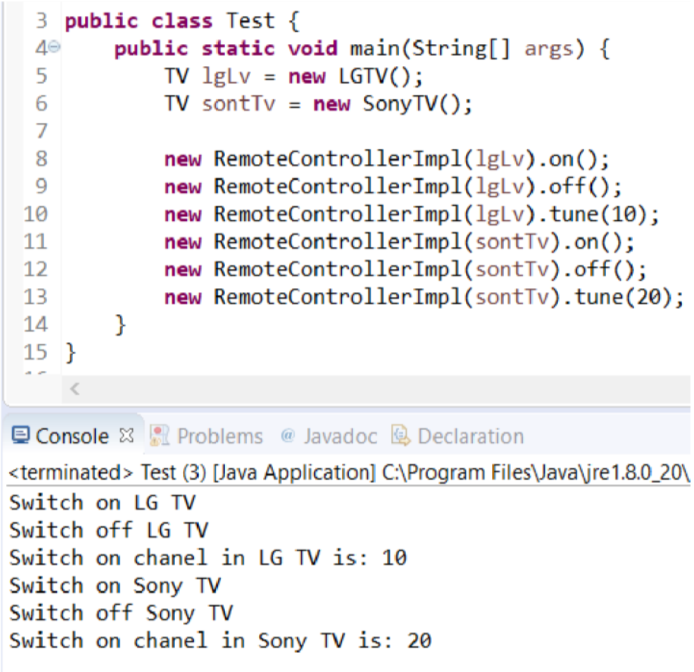
Use one remote controller for two types of **TVs** (**LG tv** and **Sony tv**) Implement how you can proceed with two types of TVs for both.

1. Create two interfaces for **TV** and **RemoteController** and implements operation **on(), off()** and **tune(int channel)**
2. Now implement 2 concrete classes for **LGtv** and **SonyTv** and implement above **on(), off()** and **tune(int channel)** operations in each class
3. Now implement the **RemoteContrlolerImpl** class as follows.



```java
public class RemoteControllerImpl implements RemoteController{

    TV tv;

    public RemoteControllerImpl(TV tv) {
        this.tv = tv;
    }

    @Override
    public void on() {
        tv.on();
    }

    @Override
    public void off() {
        tv.off();
    }

    @Override
    public void tune(int chanel) {
        tv.tune(chanel);
    }
}
```

4. Now create a Test class as follows and display the outputs below. Your implementation of above concrete classes should satisfy below outputs



```java
public class Test {
    public static void main(String[] args) {
        TV lgLv = new LGTV();
        TV sontTv = new SonyTV();

        new RemoteControllerImpl(lgLv).on();
        new RemoteControllerImpl(lgLv).off();
        new RemoteControllerImpl(lgLv).tune(10);
        new RemoteControllerImpl(sontTv).on();
        new RemoteControllerImpl(sontTv).off();
        new RemoteControllerImpl(sontTv).tune(20);
    }
}
```

<terminated> Test (3) [Java Application] C:\Program Files\Java\jre1.8.0_20\

```
Switch on LG TV
Switch off LG TV
Switch on chanel in LG TV is: 10
Switch on Sony TV
Switch off Sony TV
Switch on chanel in Sony TV is: 20
```

Exercise 09 – Strategy Pattern

Add 2 behaviors for Student class (**IFestival** and **IPRograms**) and add these behaviors are loosely coupled for the **Student** class. Each specific behavior may have its own way of **implementing algorithm** and it would not affect for the **adding** or **removing** behaviors.

Student class will be extended as **UndergraduateStudents** and **PostGraduateStudents**

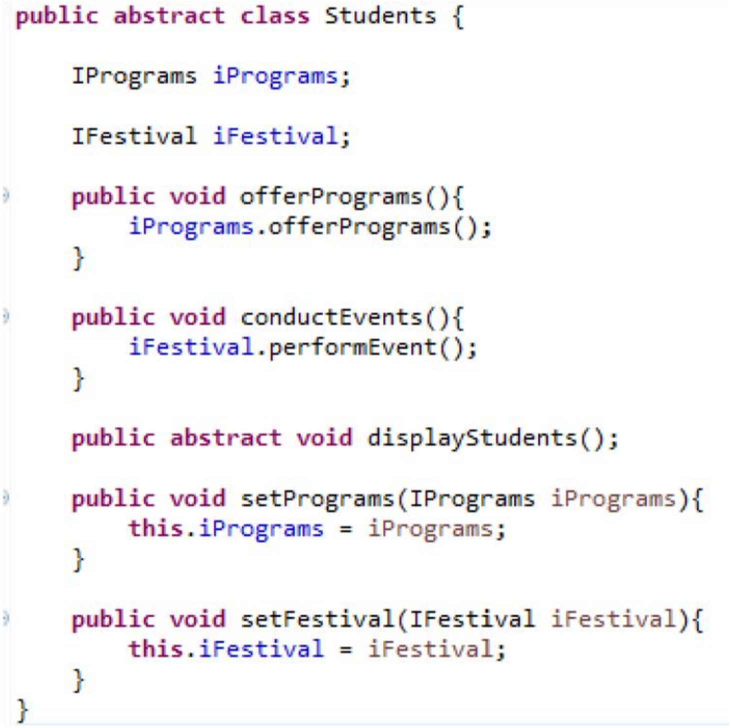
1. Create an interface called **IFestival** and declare method **performEvent()**

2. Create an interface called **IPrograms** and declare method **offerPrograms()**

3. Then create 3 concrete classes (**CodeFest**, **RoboFest** and **GameFest**) and implement the **IFestival** interface and override the **performEvent()** method in each class separately.

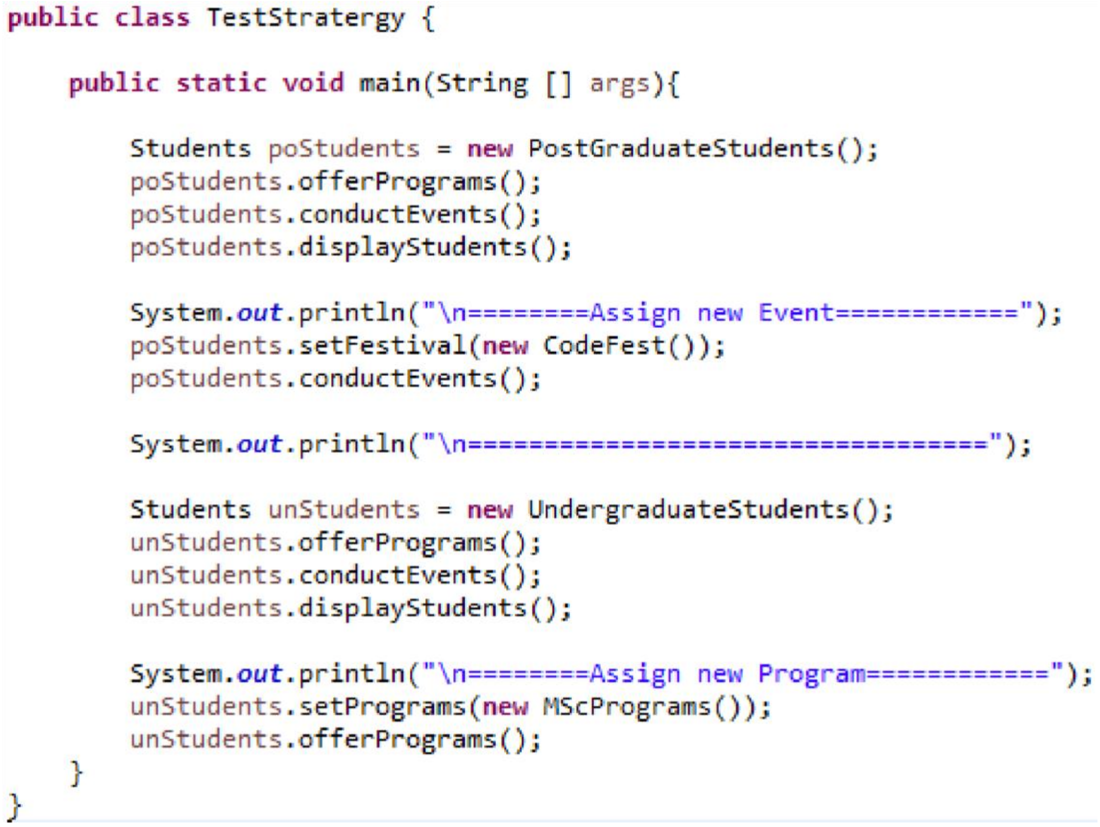
4. Now create another 3 concrete classes (**DoctoralPrograms**, **MScPrograms** and **BScPrograms**) and implement the **IPrograms** interface and override the **offerPrograms()** method in each class separately

5. Now implement an Abstract class of Student and let user to set behavior considering **aggregation relationship** as follows. (**All behaviors should be able to set dynamically**)



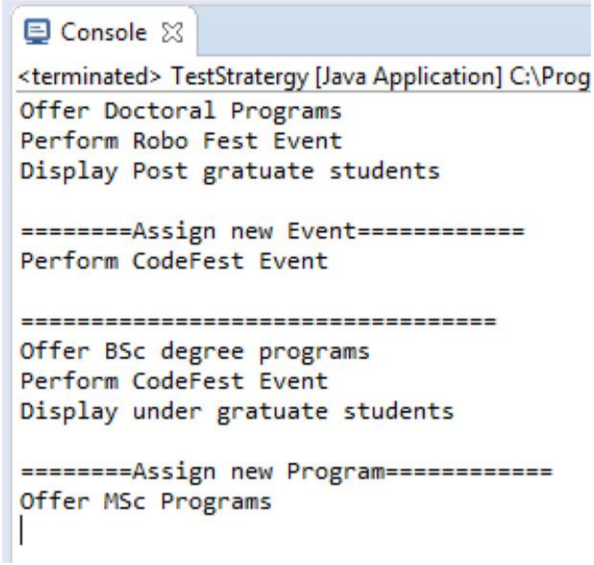
```
public abstract class Students {

    IPrograms iPrograms;

    IFestival iFestival;

    public void offerPrograms(){
        iPrograms.offerPrograms();
    }

    public void conductEvents(){
        iFestival.performEvent();
    }

    public abstract void displayStudents();

    public void setPrograms(IPrograms iPrograms){
        this.iPrograms = iPrograms;
    }

    public void setFestival(IFestival iFestival){
        this.iFestival = iFestival;
    }
}
```

6. Finally, you can implement **StratergyTest** class as follows and you should be able to **add or remove each behavior in dynamic manner using setters**. It should display output as follows.



```java
public class TestStratergy {

    public static void main(String [] args){

        Students poStudents = new PostGraduateStudents();
        poStudents.offerPrograms();
        poStudents.conductEvents();
        poStudents.displayStudents();

        System.out.println("\n========Assign new Event===========");
        poStudents.setFestival(new CodeFest());
        poStudents.conductEvents();

        System.out.println("\n==================================");

        Students unStudents = new UndergraduateStudents();
        unStudents.offerPrograms();
        unStudents.conductEvents();
        unStudents.displayStudents();

        System.out.println("\n========Assign new Program===========");
        unStudents.setPrograms(new MScPrograms());
        unStudents.offerPrograms();
    }
}
```

Output of Strategy



```
<terminated> TestStratergy [Java Application] C:\Prog
Offer Doctoral Programs
Perform Robo Fest Event
Display Post gratuate students

========Assign new Event===========
Perform CodeFest Event

==================================
Offer BSc degree programs
Perform CodeFest Event
Display under gratuate students

========Assign new Program===========
Offer MSc Programs
```

==========================END OF THE LAB=======================================