**Лабораторна робота № 1-2**

**Main**

import org.apache.commons.lang.StringUtils;  
  
import java.lang.annotation.Annotation;  
import java.lang.reflect.\*;  
  
public class Main {  
  
 public static void main(String []args) throws Exception {  
 Point\_Class point\_1 = new Point\_Class(23, 42);  
 Rectangular rectangle\_1 = new Rectangular(10.7, 20.8, 5.4, 3);  
 System.out.printf("%s\n%s\n\n",point\_1, rectangle\_1);  
  
 getting\_class\_fields(rectangle\_1);  
 getting\_class\_fields(point\_1);  
  
// getting\_method\_by\_name(rectangle\_1, "toString");  
// getting\_filed\_by\_name(rectangle\_1, "width");  
//  
// get\_annotations(rectangle\_1);  
// getting\_annotated\_methods(rectangle\_1);  
//  
// getting\_class\_methods(rectangle\_1);  
// getting\_class\_methods(point\_1);  
//  
// get\_constructors(rectangle\_1);  
// Rectangle\_constructor\_test();  
  
// check\_invocation(rectangle\_1);  
 }  
  
 public static void getting\_class\_methods(Object object) throws Exception {  
 for (Method method : object.getClass().getMethods()) {  
 if (method.isAnnotationPresent(AnnotationReflectable.class)) {  
 method.setAccessible(true);  
 System.out.println(object.getClass().getSimpleName() + "." +  
 method.getName() + "(" + get\_types(method) + ") = " + method.invoke(object));  
 }  
 }  
 System.out.println();  
 }  
  
 public static void getting\_class\_fields(Object object) throws Exception {  
 System.out.println("getting\_class\_fields");  
 for(Field field : object.getClass().getDeclaredFields()) {  
 field.setAccessible(true);  
 System.out.println(object.getClass().getSimpleName() + "." + field.getName() + " = " + field.get(object) +  
 "\ntype: " + field.getType() + "\nmodifier: " + Modifier.toString(field.getModifiers()));  
 }  
 System.out.println();  
 }  
  
 public static void getting\_annotated\_methods(Object object) throws Exception {  
 System.out.println("getting\_annotated\_methods");  
 System.out.println(object.getClass().getSimpleName());  
 for (Method method : object.getClass().getDeclaredMethods()) {  
 for (Annotation annotation : method.getDeclaredAnnotations()) {  
 System.out.println("\n" + annotation);  
 }  
 System.out.print(method.isAnnotationPresent(AnnotationReflectable.class) ?  
 method.getName() + "()\n\n" : method.getName() + "()\n"); //Arrays.toString(method.getTypeParameters())  
  
 }  
 }  
  
 public static void getting\_method\_by\_name(Object object, String name) throws Exception {  
 System.out.println("getting\_method\_by\_name");  
 if (name.startsWith("set")) {  
 throw new IllegalAccessException("Object immutable");  
 }  
 try {  
 Method method = object.getClass().getDeclaredMethod(name);  
 method.setAccessible(true);  
 System.out.println(method.getName() + " --> " + method.invoke(object));  
 } catch (NoSuchMethodException | InvocationTargetException | IllegalAccessException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public static void getting\_filed\_by\_name(Object object, String name) throws Exception {  
 System.out.println("getting\_method\_by\_name");  
 try {  
 Field field = object.getClass().getDeclaredField(name);  
 field.setAccessible(true);  
 System.out.println(field.getName() + " = " + field.get(object));  
 } catch (NoSuchFieldException | IllegalAccessException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public static void get\_annotations(Object object) throws ClassNotFoundException {  
 for (Method method : object.getClass().getDeclaredMethods()) {  
 if (method.isAnnotationPresent(AnnotationReflectable.class)){  
 for (Annotation annotation : method.getDeclaredAnnotations()) {  
 System.out.println(annotation); // System.out.println(Arrays.toString(method.getAnnotations()));  
 }  
 }  
 }  
 }  
  
 public static void get\_constructors(Object object) {  
 for (Constructor<?> constructor : object.getClass().getConstructors()) {  
 System.out.println(constructor.getName() + "(" + get\_types(constructor) + ")");  
 }  
 }  
  
 public static String get\_types(Constructor<?> constructor){  
 return StringUtils.join(constructor.getParameterTypes(), ", ");  
 }  
  
 public static String get\_types(Method method){  
 return StringUtils.join(method.getParameterTypes(), ", ");  
 }  
  
 public static void Rectangle\_constructor\_test(){  
 try {  
 Class[] params = {double.class, double.class, double.class, double.class};  
 System.out.println(Class.forName(Rectangular.class.getName()).  
 getConstructor(params).newInstance(5.55,8.2,8.4, 4.1));  
  
  
 } catch (ClassNotFoundException | InstantiationException | IllegalAccessException | NoSuchMethodException | InvocationTargetException e) {  
 e.printStackTrace();  
 }  
  
 }  
  
 public static void check\_invocation(Rectangular rectangular) throws Exception {  
 System.out.println("check\_invocation");  
 Class<Virtual> object = Virtual.class;  
 Virtual result = (Virtual) Proxy.newProxyInstance(object.getClassLoader(), new Class<?>[]{object},  
 new ImmutableProxyInvocationHandler(rectangular));  
  
 System.out.println(result.getHeight());  
 result.setWidth(1.7);  
 }  
  
 static class ImmutableProxyInvocationHandler implements InvocationHandler {  
  
 final Object target;  
  
 ImmutableProxyInvocationHandler(Object ref) {  
 this.target = ref;  
 }  
  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 if (method.getName().startsWith("set")) {  
 throw new RuntimeException("Object immutable");  
 }  
 return method.invoke(target, args);  
 }  
 }  
}

**Point\_Class**

import java.util.Objects;  
  
public class Point\_Class {  
 public double x;  
 public double y;  
  
 public Point\_Class(double x, double y) {  
 this.x = x;  
 this.y = y;  
 }  
  
 public Point\_Class() {  
 x = 0;  
 y = 0;  
 }  
 @AnnotationReflectable(name="Getting x")  
 public double getX() {  
 return x;  
 }  
 @AnnotationReflectable(name="Getting y")  
 public double getY() {  
 return y;  
 }  
 public void setX(double x) {  
 this.x = x;  
 }  
  
 public void setY(double y) {  
 this.y = y;  
 }  
  
 public void moving(double x, double y) {  
 this.x += x;  
 this.y += y;  
 }  
  
 @Override  
 @AnnotationReflectable(name="toString")  
 public String toString() {  
 return "point {x = " + x + "; y = " + y + '}';  
 }  
  
 @Override  
 public boolean equals(Object obj) {  
 if (this == obj)  
 return true;  
 if (obj == null || getClass() != obj.getClass())  
 return false;  
 Point\_Class point = (Point\_Class) obj;  
  
 return Double.*compare*(point.x, x) == 0 && Double.*compare*(point.y, y) == 0;  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(x, y);  
 }  
}

**Rectangular**

import java.util.Objects;  
  
public class Rectangular extends Point\_Class implements Virtual {  
  
 private double height;  
 private double width;  
  
 public Rectangular(double x, double y, double height, double width) {  
 super(x, y);  
 this.height = height;  
 this.width = width;  
 }  
  
 public Rectangular(double height, double width) {  
 super();  
 this.height = height;  
 this.width = width;  
 }  
  
 public Rectangular() {  
 super();  
 this.height = 0;  
 this.width = 0;  
 }  
  
 public Rectangular(Rectangular r) {  
 super(r.getX(), r.getY());  
 this.width = r.getWidth();  
 this.height = r.getHeight();  
 }  
  
 @AnnotationReflectable(name="Getting Height")  
 public double getHeight() {  
 return height;  
 }  
  
 public void setHeight(double height) {  
 this.height = height;  
 }  
  
 @AnnotationReflectable(name="Getting Width")  
 public double getWidth() {  
 return width;  
 }  
  
 public void setWidth(double width) {  
 this.width = width;  
 }  
  
 @AnnotationReflectable(name="Getting Area")  
 public double area() {  
 return width \* height;  
 }  
  
 @AnnotationReflectable(name="Getting Perimeter")  
 public double perimeter() {  
 return 2 \* (width + height);  
 }  
  
 public boolean isEmpty() {  
 return (width <= 0.0) || (height <= 0.0);  
 }  
  
 public boolean contains(double x, double y) {  
 double x0 = getX();  
 double y0 = getY();  
 return (x >= x0 && y >= y0 && x < x0 + getWidth() && y < y0 + getHeight());  
 }  
  
// @AnnotationReflectable(name = "moving")  
 @Override  
 public void moving(double x, double y) {  
 super.moving(x, y);  
 }  
//  
// @AnnotationReflectable(name = "toString")  
 @Override  
 public String toString() {  
 return "Rectangular {height = " + height + ", width = " + width + "}";  
 }  
  
 @Override  
 public boolean equals(Object obj) {  
 if (this == obj) return true;  
 if (obj == null || getClass() != obj.getClass() || !super.equals(obj)) return false;  
 Rectangular that = (Rectangular) obj;  
 return Double.*compare*(that.height, height) == 0 && Double.*compare*(that.width, width) == 0;  
 }  
  
 @Override  
 protected Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(super.hashCode(), height, width);  
 }  
}

**AnnotationReflectable**

import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
  
@Retention(RetentionPolicy.*RUNTIME*)  
@Target(ElementType.*METHOD*)  
public @interface AnnotationReflectable {  
 public String name();  
}

**Virtual**

public interface Virtual {  
  
 void moving(double x, double y);  
  
 default void moving() {  
 System.*out*.println("moving");  
 }  
  
 double getX();  
  
 double getY();  
  
 void setX(double x);  
  
 void setY(double y);  
  
 double getHeight();  
  
 void setHeight(double height);  
  
 double getWidth();  
  
 void setWidth(double width);  
  
 double area();  
  
 double perimeter();  
  
}

**Tests**

**Point\_ClassTest**

import org.junit.jupiter.api.AfterEach;  
 import org.junit.jupiter.api.BeforeAll;  
 import org.junit.jupiter.api.BeforeEach;  
 import org.junit.jupiter.api.Test;  
  
 import static org.junit.jupiter.api.Assertions.assertEquals;  
 import static org.junit.jupiter.api.Assertions.assertNotEquals;  
  
  
class Point\_ClassTest {  
  
 Point\_Class TestObject = new Point\_Class();  
  
 @Test  
 void getX() {  
 TestObject.setX(3);  
 assertEquals(TestObject.getX(), 3);  
 assertNotEquals(TestObject.getX(), 5);  
 }  
  
 @Test  
 void getY() {  
 TestObject.setY(6);  
 assertEquals(TestObject.getY(), 6);  
 assertNotEquals(TestObject.getY(), 5);  
 }  
  
 /\* Test  
 void setX() {  
 }  
  
 @Test  
 void setY() {  
 }  
 \*/  
  
 @Test  
 void moving() {  
 TestObject.setX(3);  
 TestObject.setY(4);  
 TestObject.moving(1, 3);  
 assertEquals(new Point\_Class(4, 7), TestObject);  
 assertNotEquals(new Point\_Class(3, 4), TestObject);  
 }  
  
 @Test  
 void testToString() {  
  
 TestObject.setX(3);  
 TestObject.setY(4);  
  
 assertEquals("point {x = 3.0; y = 4.0}", TestObject.toString());  
 assertNotEquals("point {x = 3.0; y = 4.0", TestObject.toString());  
 }  
  
}

**Point\_ClassTest**

import org.junit.jupiter.api.BeforeEach;  
 import org.junit.jupiter.api.Test;  
  
 import java.io.ByteArrayOutputStream;  
 import java.io.PrintStream;  
  
 import static org.junit.jupiter.api.Assertions.\*;  
  
  
class RectangularTest {  
  
 Rectangular TestObject = new Rectangular();  
 Rectangular Rectangle = new Rectangular(4, 9, 8.5, 7);  
  
// private final ByteArrayOutputStream output = new ByteArrayOutputStream();  
//  
// @BeforeEach  
// void setUp() {  
// System.setOut(new PrintStream(output));  
// }  
  
 @Test  
 void getHeight() {  
 TestObject.setHeight(7.9);  
 assertEquals(7.9, TestObject.getHeight());  
 assertNotEquals(2.9, TestObject.getHeight());  
 }  
  
 @Test  
 void getWidth() {  
 TestObject.setWidth(5);  
 assertEquals(5, TestObject.getWidth());  
 assertNotEquals(2.9, TestObject.getWidth());  
 }  
  
 /\*  
 @Test  
 void setWidth() {  
 }  
  
  
 @Test  
 void setHeight() {  
 }  
  
  
  
 @Test  
 void area() {  
 TestObject.setX(9);  
 TestObject.setY(4);  
 assertEquals(36, TestObject.area());  
 assertNotEquals(64.9, TestObject.area());  
 }  
  
 \*/  
  
 @Test  
 void area() {  
 assertEquals(59.5, Rectangle.area());  
 assertNotEquals(64.9, Rectangle.area());  
 }  
  
 @Test  
 void perimeter() {  
 assertEquals(31, Rectangle.perimeter());  
 assertNotEquals(69, Rectangle.perimeter());  
 }  
  
 @Test  
 void isEmpty() {  
 assertFalse(Rectangle.isEmpty());  
 assertNotEquals(true, Rectangle.isEmpty());  
 }  
  
 @Test  
 void contains() {  
 assertTrue(Rectangle.contains(5, 12));  
 assertFalse(Rectangle.contains(2, 12));  
 }  
  
 @Test  
 void moving() {  
 Rectangle.moving(1, 3);  
 assertEquals(new Rectangular(5, 12, 8.5, 7), Rectangle);  
 assertNotEquals(new Rectangular(3, 4, 8.5, 7), Rectangle);  
 }  
  
 @Test  
 void testToString() {  
 assertEquals("Rectangular {height = 8.5, width = 7.0}", Rectangle.toString());  
 assertNotEquals("Rectangular {height = 8.5, width = 7}", Rectangle.toString());  
 }  
  
  
}

**MainTest**

import org.junit.jupiter.api.BeforeAll;  
 import org.junit.jupiter.api.Test;  
 import java.util.ArrayList;  
 import java.util.Arrays;  
  
 import static org.junit.jupiter.api.Assertions.assertEquals;  
  
class MainTest {  
  
 Rectangular Rectangle = new Rectangular(5, 6, 7, 8);  
  
// ArrayList<String> Expected\_constructors = new ArrayList<String>(  
// Arrays.asList("Rectangular(class Rectangular)",  
// "Rectangular()", "Rectangular(double, double)", "Rectangular(double, double, double, double)"));  
  
 ArrayList<String> Expected\_methods = new ArrayList<>(Arrays.asList(  
 "Rectangular.area() = 56.0", "Rectangular.getHeight() = 7.0", "Rectangular.getWidth() = 8.0",  
 "Rectangular.getX() = 5.0", "Rectangular.getY() = 6.0", "Rectangular.perimeter() = 30.0")  
 );  
  
// ArrayList<String> Expected\_fields = new ArrayList<>(Arrays.asList(  
// "Rectangular.height = 7.0 type: double modifier: private", "Rectangular.width = 8.0 type: double modifier: private)"  
// ));  
//  
  
 @BeforeAll  
 static void setUp() {  
 }  
  
 @Test  
 void getting\_class\_methods() {  
 try {  
 assertEquals(Main.*getting\_class\_methods*(Rectangle), Expected\_methods);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 @Test  
 void check\_invocation() {  
 try {  
 Main.*check\_invocation*(Rectangle);  
 } catch (Exception e) {  
 assertEquals("java.lang.RuntimeException: Object immutable", e.toString());  
 }  
 }  
}