DUE DATE: **By the end of your lab session**

CSE-271: Object-Oriented Programming

Exercise #5

Max Points: 20

Name: Jacob Igel

For your own convenient reference – You should first save/rename this document using the naming convention **`MUid_Exercise5.docx`** (example: `raodm_Exercise5.docx`) prior to proceeding with this exercise.

Objectives: The objectives of this exercise are to:

1. Explore the concepts of inheritance & interfaces
2. Experiment concepts of polymorphic method calls
3. Gain familiarity with Java's approach for interfaces
4. Developing Java classes meeting interface requirements

Fill in answers to all of the questions. For some of the questions you can simply copy-paste appropriate text from Eclipse output into this document. You may discuss the questions or seek help from your neighbor, TA, and/or your instructor.

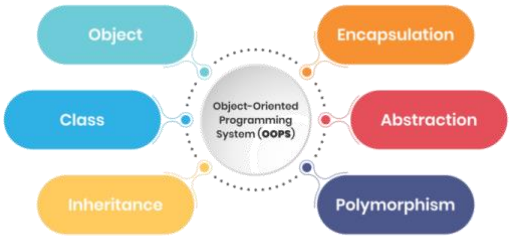
Part #0: One time setup of Eclipse (IDE) – Only if needed

We already configured `Eclipse`'s source formatter and Checkstyle plug-in as part of Lab #1. If your `Eclipse` is not configured (because you are using a different computer) then use the instructions from Lab #1 to configure `Eclipse`.

Part #1: Generic concepts of Object-oriented programming

Estimate time: < 30 minutes

Background: Object-oriented Programming (OOP) is a programming paradigm that is widely used and adopted by several mainstream programming languages, such as: C++, C#, JavaScript, Java, and Python. Hence, understanding the generic concepts underlying OOP is very important for your future careers, immaterial of the programming-language that you may be working with. Moreover, clearly and concisely explaining concepts is a very important skills for your future job-interviews and in your jobs. Hence, the exams also involve such questions, and in the labs, we will practice this style of questions.



Exercise: Briefly (2-to-3 sentences each) respond to the following questions regarding generic concepts of object-oriented programming (OOP).

1. In object-oriented programming (OOP) parlance, what is polymorphism?

   "Inheriting-class can override(different from overload) methods to perform different operations"

   Child classes can override methods in the parent class to customize functionality. Overrides apply even when methods are class referenced from the parent-class.

   Source: PowerPoint

2. What is an `abstract` method? When would you use an abstract method?

   An abstract method can be present only in an abstract class. If a class has an abstract method, it must be marked as an abstract class. An abstract class need not have any abstract methods. You would use an abstract method to implement certain methods.

   Source: PowerPoint

3. What is an abstract class? Briefly (1-to-2 sentences) describe <u>two different</u> scenarios when we would want a class to be an abstract class?

   An abstract class is an incomplete class that we use to "enforce derived classes to implement certain methods thereby enforcing standard API"

   Source: PowerPoint

4. What is upcasting?

   Upcasting is the typecasting of a child object to a parent object. Upcasting can be done implicitly and it gives us the flexibility to access the parent class members.

   Source: https://www.geeksforgeeks.org/upcasting-vs-downcasting-in-java/

5. What is down-casting? How do we achieve safe down-casting in Java?

   Down casting means the typecasting of a parent object to a child object. Down casting cannot be implicit. To achieve safe down casting, we use the instanceof operator/the getClass() method to check the data type.

   Source: https://www.geeksforgeeks.org/upcasting-vs-downcasting-in-java/

6. What is the difference between using `instanceof` operator versus checking the `class` (of a Java object)?

   Instance of - instanceof is a boolean operator can be used to safely check compatibility of a class without typecasts

   getClass() - It returns a Class object that contains information used by the JVM for loading and processing classes.

   Source: PowerPoint

7. What is an `interface` in Java?

   An interface is a completely "abstract class" that is used to group related methods with empty bodies. Interfaces are “pure” abstract classes – none of the methods can have any definition

   Source: https://www.w3schools.com/java/java_interface.asp & PowerPoint

8. State two similarities between an `abstract` class and an `interface` in Java
   1. Upcasting and Downcasting work for objects whether the parent is an abstract class or an interface
   2. They can both contain static and final instance variables

9. State at least two differences between an `abstract` class and an `interface` in Java?
   1. Classes have to extend an abstract class while you have to implement an interface
   2. You can implement an abstract interface with an abstract class but you cannot do the other way around.

Part #2: Inheritance and polymorphism in Java

Estimated time: < 25 minutes

Background: The generic OOP concepts (covered in previous part) are supported by different object-oriented programming-languages in slightly different ways. Java has a specific approach for supporting the 4-key tenants of OOP using Java classes. However, similar syntax and semantics also apply to other object-oriented programming-languages.

Exercise: Consider the following for related Java classes shown below:

```
public class Boat {
  public void print() {
    System.out.println("Slow boat");
  }
  public void doIt() {
    System.out.println("Rowing");
  }
```

```
public class Yacht extends Boat {
  @Override
  public void print() {
    super.print();
    System.out.println("Quick boat");
  }
}
```

```
}
```

```
public class Ship extends Yacht {
  @Override
  public void print() {
    super.print();
    System.out.println("Fast boat");
  }
  @Override
  public void doIt() {
    super.doIt();
    System.out.println("Steaming");
  }
}
```

```
public class WaterWorks {
  public static void main(String[] a) {
    Boat boats[] = {new Boat(),
          new Yacht(), new Ship()};
    for (Boat b: boats) {
       b.doIt();
       b.print();
       System.out.println("---");
    }
  }
}
```

1. What constructors are being used to instantiate objects in `WaterWorks.main()` method?

   The default constructors of boat, yacht, and ship are being sed to instantiate objects in Water Works.

2. Illustrate the output from `WaterWorks.main()` method in the space below:

   Rowing
   Slow boat

   Rowing
   Slow boat
   Quick boat

   Rowing
   Steaming.
   Slow boat
   Quick boat
   Fast boat
   ---

3. Given the above inheritance hierarchy (of `Boat`, `Yacht`, and `Ship`), complete the following method to create different objects from a list of class names supplied to you as a string:

```
/**
  * This method create different objects from a list of class names supplied
  * to you as a string. For example, if given the string "Ship Boat Boat"
  * this method returns an array list with those objects (in that exact
  * order).
  *
  * @param names The names of the objects to be created. Each one is
  *     separated by one-or-more white spaces.
  *
  * @return An array list containing the corresponding objects. The objects
  *     are returned in exactly the same order in which they are specified.
```

```
   */
public ArrayList<Boat> createObjects(String names) {
    ArrayList<Boat> list = new ArrayList<Boat>();
```

```
 try (Scanner in = new Scanner(names)) {
      while (in.hasNext()) {
      final String name = in. next();
   if(names.equals("Boat") {
      list.add(new Boat());
   } else if(names.equals("Yacht") {
      list.add(new Yacht());

   } else if(names.equals("Ship") {
      list.add(new Ship());

   } else {
      throw new IllegalArgumentException("Invalid name " + name);
   }


}
```

```
    return list;
}
```

4. Consider the following for related Java classes shown below:

```
public interface Swimmer {
  public void swim();
  public String getName();
}
```

```
public class Fish implements Swimmer{
  @Override
  public void swim() {
    System.out.println("Swimming");
  }
  @Override
  public String getName() {
    return "Fish";
  }
}
```

```
public class Shark extends Fish {
  @Override
  public String getName() {
    return "Shark";
  }
}
```

```
public class Aquarium {
  public static void doIt(Swimmer s){
    System.out.println(s.getName());
    s.swim();
  }

  public static void main(String[] a) {
    Fish f = new Fish();
    Fish s = new Shark();
    doIt(f);
    doIt(s);
  }
}
```

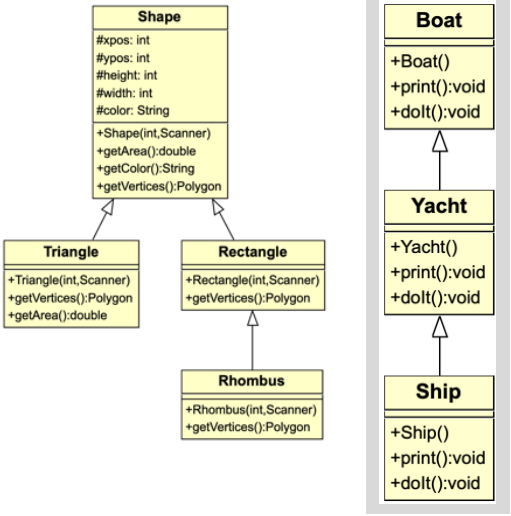
Illustrate the output from `Aquarium.main()` method in the space below:

```
Fish
Swimming
Shark
```

Swimming

5. A company has developed a sophisticated algorithm to optimally pack different types of shapes into another shape – in computing, this is called a "bin packing" algorithm which is a NP-complete problem. They have built their `magicPack` algorithm to use a custom `Shape` hierarchy that has a method call `getVertices` that provides the polygon defining the `Shape` as shown below:

```
/**
  * Optimally packs a subset of shapes into a given bin
using a
  * proprietary algorithm. Some shapes may not fit into the
bin.
  *
  * @param bin The bounding box or bin into which a subset
of
  *      shapes are to be fit.
  * @param toFit The set of shapes to be fit into the bin.
  * @return The subset of shapes that were actually fit.
  */
public ArrayList<Shape> magickPack(Polygon bin,
                                   ArrayList<Shape> toFit)
{
   // Proprietary algorithm uses the getPolygon method
   // defined for each shape!
}
```



Now, a new high-paying client wants to use `magickPack` to work with their `Boat` hierarchy (see UML above). Describe how the company can use a Java interface to enable their `magicPack` algorithm to work with the `Shape` hierarchy or the `Boat` hierarchy. **Must show pertinent code fragments (not full source code) to earn full points**.

```
public interface Geometry {
        /** Return verticies of polygon for this geometry */
        public Polygon getVertices();
    }

public ArrayList<Geometry> magickPack(Polygon bin,
                        ArrayList<Geometry> toFit) {

}
```

Alter shape to fit the Geometry interface, use the boat class to implement the Geometry interface, use the interface

Part #3: Programming with inheritance & interfaces

Estimated time: 30 minutes

DUE DATE: By the end of your lab session

Background: A key advantage of inheritance and polymorphism is that it enables effective reuse of methods or algorithms. Interfaces are an integral part of OOP and play an important role in enabling effective reuse of methods. In this part of the exercise you will be working in interfaces to obtain a better understand of their use in Java.

Setup: First, you need to setup a Java project in Eclipse, download the starter code. In this part of the exercise, you are given the following set of classes:
You have been provided with the following classes that represent a hierarchy of items that are going to be used in this exercise.

File Name	***Description***
`Controllable.java`	A simple interface to enable turning devices on and off. **Do not modify this one class**.
`RemoteControl.java`	Another interface that extends `Controllable` to add a couple of more features.
`LightBulb.java`	A simple class that implements `Controllable`.
`CDPlayer.java`	A simple class that implements `RemoteControl`.
`LivingRoom.java`	A class that contains a set of controllable items. This is an independent class for testing the aforementioned classes.

Exercise: First, review the classes supplied to you and note the various methods and instance variables defined in the different classes and then proceed with the following steps (while making notes as you go along).

In the following steps, you may need to add methods to different classes. For these new methods you add, implement the methods to just print simple messages similar to other methods in the class (or other classes). See sample output further below for messages or discuss with your instructor.

As you fix issues, make notes on the issues being fixed along with the generic rule associated it. For example, let's say a variable was not defined correctly, then you should make the following note:

- "*Issue*: Variable `i` was not defined" and
- "*Generic rule*: In Java, variables must be defined with the correct data type before they are used."

Step #1: Fix syntax errors in `LightBulb.java`

The supplied `LightBulb.java` intentionally has bugs in it. First troubleshoot the `LightBulb.java` class and fix all syntactic errors so that it compiles successfully. You won't be able to run the program yet (need to fix errors in other classes as well first).

Briefly describe what issue(s) needed to be fixed and the generic rule associated with it (see earlier note for example or seek clarifications from your instructor/TA)

Issue: Off method not implemented

Generic rule: All of the methods in the interface must be implemented for a non-abstract class to work.

Step 2: Fix syntax errors in `RemoteControl.java` Interface class

The supplied `RemoteControl.java` interface class intentionally has bugs in it. Troubleshoot the compile issues in `RemoteControl.java` class and fix all syntactic errors so that it compiles successfully. You won't be able to run the program yet (need to fix errors in other classes as well first).

Briefly describe what issue(s) needed to be fixed and the generic rule associated with it (see earlier note for example or seek clarifications from your instructor/TA)

Issue: Had to change protected to public
Generic rule: interfaces need to have public or abstract modifiers

Step 3: Fix syntax errors in `CDPlayer.java`

The supplied `CDPlayer.java` class intentionally has bugs in it. Troubleshoot the compile issues in `RemoteControl.java` class and fix all syntactic errors so that it compiles successfully.

Briefly describe what issue(s) needed to be fixed and the generic rule associated with it (see earlier note for example or seek clarifications from your instructor/TA)

Issue: Had to implement on and off to the method.
Generic rule: All of the methods in the interface must be implemented for a non-abstract class to work.

Step 4: Run `LivingRoom.java` and fix bugs

- Once you have successfully completed the previous tasks, you should have eliminated all syntax errors in the program. Now, you can run the program to execute `LivingRoom.main()` method that serves as the test harness.
- When you run this program, it will crash with an exception in `LivingRoom.on()` method. Study the class and the method and appropriately troubleshoot the bugs to get the `on()` method working correctly. The `instanceof` keyword will be handy here.

- Briefly describe what issue(s) needed to be fixed and the generic rule associated with it (see earlier note for example or seek clarifications from your instructor/TA)

Issue: * * * Testing 1 * * *
----[Turning devices on]----
Light0: Light bulb is on
Exception in thread "main" java.lang.ClassCastException: class LightBulb cannot be cast to class RemoteControl (LightBulb and RemoteControl are in unnamed module of loader 'app')

```
        at LivingRoom.on(LivingRoom.java:45)
        at LivingRoom.main(LivingRoom.java:74)
```

Generic rule: You have to check the type of data prior to type casting.

- If you correctly troubleshoot this issue, then you will be able to obtain the following output from the program:
  Expected output after Step 4 is complete:

```
* * *    Testing 1    * * *
----[  Turning devices on  ]----
Light0: Light bulb is on
Light1: Light bulb is on
Player2: CD Player is on
Player2: Playing music.
----[  Turning devices off  ]----

* * *    Testing 2    * * *
----[  Turning devices on  ]----
Light0: Light bulb is on
Player1: CD Player is on
Player1: Playing music.
Light2: Light bulb is on
Light3: Light bulb is on
----[  Turning devices off  ]----
```

Step 5: Implement and test `LivingRoom.off()` method

Once you have successfully completed the previous task and verified the output, use the `on()` method as a reference to implement the `off()` method as follows:

- The `off()` method should turn-off all the appliances.
- However, if you have an appliance that implements `RemoteControl` interface then ensure you stop the appliance before you turn it off.z
- **Expected output after Step 5 is complete:**

```
* * *    Testing 1    * * *
----[  Turning devices on  ]----
Light0: Light bulb is on
Light1: Light bulb is on
Player2: CD Player is on
Player2: Playing music.
----[  Turning devices off  ]----
Light0: Light bulb is off
Light1: Light bulb is off
Player2: Stopping music.
Player2: CD Player is off

* * *    Testing 2    * * *
----[  Turning devices on  ]----
Light0: Light bulb is on
Player1: CD Player is on
Player1: Playing music.
Light2: Light bulb is on
Light3: Light bulb is on
----[  Turning devices off  ]----
Light0: Light bulb is off
Player1: Stopping music.
Player1: CD Player is off
Light2: Light bulb is off
Light3: Light bulb is off
```

Part #4: Submit to Canvas via CODE plug-in

Estimated time: < 5 minutes

Exercise: You will be submitting the following files via the Canvas CODE plug-in:

1. This MS-Word document saved as a PDF file – **Only submit PDF file**.
2. The Java source files: `RemoteControl.java`, `LightBulb.java`, `CDPlayer.java`, `LivingRoom.java` that you modified in this exercise.

Ensure you actually complete the submission on Canvas by verifying your submission