**CSE-271: Object-Oriented Programming**

**Exercise #5**

Max Points: 20

|  |  |
| --- | --- |
| **Name:** | Raobot |

|  |  |
| --- | --- |
| Pin | 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. |

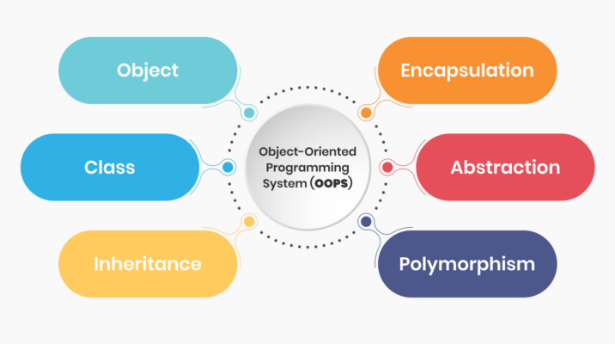
|  |
| --- |
| **Objectives**: The objectives of this exercise are to:   1. Explore the concepts of inheritance & interfaces 2. Experiment concepts of polymorphic method calls 3. Gain familiarity with Java’s approach for interfaces 4. Developing Java classes meeting interface requirements   Fill in answers 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

|  |  |
| --- | --- |
| Eclipse Logo A2 by dj-fahr on DeviantArt | 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?

|  |
| --- |
| Polymorphism means "many forms" – *i.e.*, it allows us to perform a single action in multiple ways. In OOP, polymorphism occurs when methods (methods cannot be static or final) are selectively overridden in subclasses (or child classes) to perform different operations suitable to the subclass. |

1. What is an abstract method? When would you use an abstract method?

|  |
| --- |
| An abstract method does not have any implementation but specifies the interface (i.e., parameters, return value, and exceptions) for an overridable method. We use abstract methods to stipulate (or enforce) certain methods (or attributes of a class) but we are unable to provide a good generic (or common) implementation for this method in the parent class. Abstract methods must be defined in an abstract class. |

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

|  |
| --- |
| An abstract class contains zero or more abstract methods (defined directly or inherited). Abstract methods cannot be instantiated using the new operator. We can use abstract classes in the following scenarios:   * When we want to stipulate a generic API for classes without providing implementation for methods (either because a common/generic implementation does not make sense or we want the derived class to consciously implement it). * When we derive from a parent class and implement only a subset of methods or only partially implement interfaces. * intentionally want only the child class(es) to be instantiated. |

1. What is upcasting?

|  |
| --- |
| The process of creating a parent class (or base class) reference from a derived class is called upcasting. Upcasting is a safe operation and can be reliably performed. Hence, in Java, upcasting is automatically performed when passing arguments to method calls. |

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

|  |
| --- |
| The process of creating a subclass (or child class or derived class) reference from a derived class is called upcasting. Upcasting is not a safe operation and can cause ClassCastException, if incorrectly performed. Hence, we must use instanceof operator or the getClass() method to check the data type prior to down casting. |

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

|  |
| --- |
| Checking the class object results in checking if an object’s data type is exactly a given class. On the other hand, the instanceof operator checks for a given class-and-descendants – i.e., checks many subclasses (or child classes) |

1. What is an interface in Java?

|  |
| --- |
| An interface class specifies the interface (i.e., parameters, return value, and exceptions) for an overridable method. |

1. State two similarities between an abstract class and an interface in Java

|  |
| --- |
| 1. Both of these mechanisms essentially stipulate the common functionality (or API) for their subclasses. 2. The methods (not final or static methods) in abstract class or an interface are overridden in a child class. 3. Both abstract classes and interfaces can contain static and final instance variables. 4. Upcasting and down-casting work consistently for objects immaterial of whether the parent is an abstract class or an interface. |

1. State at least two differences between an abstract class and an interface in Java?

|  |
| --- |
| 1. Classes must extend an abstract class and have to implement an interface. 2. Abstract class can implement an interface but not vice versa 3. Methods in an interface are all abstract methods (unless they are default or static methods). However, an abstract class can have non-abstract and final methods. 4. An abstract class can have static, final, and other instance variables. An interface can have only final (*i.e.*, effectively constant) and static instance variables. 5. Attributes of an interface are all public. An abstract class can have public, protected, package-private (when no access modifier is specified), and private instance variables. |

# 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 or no-argument constructors of the Boat, Yacht, and Ship classes are being used to instantiate objects. |

1. 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  --- |

1. 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>(); |
| // Using a scanner to easily extract word-by-word without creating  // a whole bunch of words in the middle (which would be the case  // with split).  **try** (Scanner sc = **new** Scanner(names)) {  **while** (sc.hasNext()) {  **final** String name = sc.next();  **switch** (name) {  **case** "Boat": list.add(**new** Boat());  **break**;  **case** "Yacht": list.add(**new** Yacht());  **break**;  **case** "Ship": list.add(**new** Ship());  **break**;  **default**:  **throw** **new** IllegalArgumentException("Invalid name " + name);  }  }  } |
| **return** list;  } |

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

|  |  |  |
| --- | --- | --- |
| **public** **interface** Swimmer {  **public** **void** swim();  **public** String getName();  } |  | **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);  }  } |
|  |  |
| **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";  }  } |  |

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

|  |
| --- |
| Fish  Swimming  Shark  Swimming |

1. 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**.

|  |
| --- |
| An interface provides a convenient way to enable algorithms to work with different object hierarchies. For example, the company can create a new Geometry interface with a getVertices() method and revise the magicPack method as shown below:  **public** **interface** **Geometry** {  /\*\* Return verticies of polygon for this geometry \*/  **public** Polygon getVertices();  }  **public** ArrayList<**Geometry**> **magickPack**(Polygon bin,  ArrayList<**Geometry**> toFit) {  // Proprietary algorithm uses the getPolygon method  // defined for each **Geometry**!  }  Next, the Shape class can be modified to implement the Geometry interface ­– the existing getVertices() method would already work just fine here and so no extra implementation is necessary in this hierarchy.  Next, the Boat class can be modified to implement the Geometry interface and each subclass needs to suitably override the getVertices method.  Since both hierarchies implement the Geometry interface, they can now be used with the magicPack algorithm. This is the big benefit of interfaces for designing solutions and enabling effective reuse of software. |

# Part #3: Programming with inheritance & interfaces

*Estimated time: 30 minutes*

**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).

|  |  |
| --- | --- |
| Questions RTL | 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. |

|  |  |
| --- | --- |
| Information | 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*: The method off in Controllable interface was not implemented in the LightBulb class.  *Generic rule*: When a concrete class (i.e., not abstract) implements an interface, all the methods in the interface must be implemented. |

**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*: The RemoteControl interface class must have public access-modifier.  *Generic rule*: An interface class can have only public or abstract access-modifier. |

**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*: The method off in Controllable interface was not implemented in the LightBulb class.  *Generic rule*: When a concrete class (i.e., not abstract) implements an interface, all the methods in the interface must be implemented. |

**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*: The method on was performing an incorrect type cast resulting in the following exception:  Exception in thread "main" java.lang.ClassCastException: class LightBulb cannot be cast to class RemoteControl  at LivingRoom.on(LivingRoom.java:43)  at LivingRoom.main(LivingRoom.java:77)  *Generic rule*: In Java, objects can be type casted only if they are of the correct data type. Otherwise, it results in a ClassCastException (it is an unchecked exception derived from the RuntimeException class). Hence, it is important to check the data type of a class using instanceof or the getClass() method 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