Lab Questions

Chapter 9: Interfaces and Polymorphism

Tutorial

Using an interface to share methods

1.1 *It is often the case that two or more classes share a common set of methods. For programming purposes we might wish to treat the objects of those classes in a similar way by invoking some of their common routines.*

For example, the `Dog` *and* `Cat` *classes listed below agree on the* `void` *method* `speak`*. Because* `Dog` *and* `Cat` *objects have the ability to "speak," it is natural to think of putting both types of objects in an* `ArrayList` *and invoking* `speak` *on every object in the list. Is this possible? Certainly we could create an* `ArrayList` *of* `Dog` *that would hold all the* `Dog` *objects, but can we then add a* `Cat` *object to an* `ArrayList` *of* `Dog`*?*

Try running the main program below as it is written. Run it a second time after uncommenting the line that instantiates a `Cat` *object and tries to add it to the* `ArrayList`*.*

```
import java.util.*;

public class AnimalRunner
{
    public static void main(String[] args)
    {
        ArrayList<Dog> dogcatList = new ArrayList<Dog>();
        dogcatList.add(new Dog("Fred"));
        // dogcatList.add(new Cat("Wanda"));
    }
}

-------------------
public class Dog
{
    private String name;

    public Dog(String name)
    {
        this.name = name;
    }

    public void speak()
    {
       System.out.println("Woof! Woof!");
```

```
    }

    public String toString()
    {
        return "Dog:   " + name;
    }
}

-------------------
public class Cat
{
    private String name;

    public Cat(String name)
    {
        this.name = name;
    }

    public void speak()
    {
       System.out.println("Meow! Meow!");
    }

    public String toString()
    {
        return "Cat:   " + name;
    }
}
```

Our experiment to add `Cat` objects to an `ArrayList` of `Dog` objects failed. Perhaps we should try using the original Java `ArrayList` without generics? Try running the code below as it is written along with the `Dog` and `Cat` classes defined above. Run it a second time after uncommenting the line that invokes `speak`.

```
import java.util.*;

public class AnimalRunner
{
    public static void main(String[] args)
    {
        ArrayList dogcatList = new ArrayList();
        dogcatList.add(new Dog("Fred"));
        // dogList.add(new Cat("Wanda"));
        for (Object obj : dogcatList)
        {
            // obj.speak();
        }
    }
}
```

The experiment shows that we are now able to add `Dog` *and* `Cat` *objects to the* `ArrayList`*, but there is a compile error on the line* `obj.speak` *because* `obj` *is an* `Object` *reference variable and the class* `Object` *doesn't contain a* `speak` *method. We need a reference variable that can refer to* `Dog` *and* `Cat` *objects and which also allows us to invoke* `speak`*. The solution to the problem uses interfaces.*

First create an interface called `Speakable` *that contains a* `void speak()` *method signature. Be sure to modify the* `Dog` *and* `Cat` *classes to indicate that they implement the* `Speakable` *interface. For example, in the case of the* `Dog` *class, we will code* `public class Dog implements Speakable`*. Be sure to make a similar change in the declaration of the* `Cat` *class.*

The term `Speakable` *can be used to create* `Speakable` *references. Using generics, create an* `ArrayList` *of* `Speakable` *objects in the* `main` *method. Modify the* `for` *loop so that it iterates over* `Speakable` *objects. Try adding the* `Dog` *and* `Cat` *objects and invoking the* `speak` *method on each object. Does this work?*

Your answer :

```
ArrayList<Speakable> speakers = new ArrayList<Speakable>();

speakers.add(new Dog("Wanda"));
speakers.add(new Cat("Cosmo"));

for (Speakable s : speakers) {
      s.speak();
}

// Yes it works
```

Casting class objects

1.2 Compile and execute the code listed below as it is written. Run it a second time after uncommenting the line `obj.speak();`.

```
public class AnimalRunner
{
   public static void main(String[] args)
   {
      Dog d1 = new Dog("Fred");
```

```
        d1.speak();
        Object obj = new Dog("Connie");
        // obj.speak();
    }
}
```

The uncommented line causes a compile error because `obj` is an `Object` reference variable and class `Object` doesn't contain a `speak()` method. This is the case, in spite of the fact that `obj` refers to a `Dog` object and that class `Dog` has a `speak()` method. For safety, the Java compiler limits the methods that can be called using a reference variable to only those methods that belong to the class that was used to create the reference variable. So `obj` can only invoke `Object` methods.

Use the following method to make Connie speak: Create a second `Dog` reference variable `d2`. Cast `obj` to a `Dog` and assign the new reference to `d2`. Now invoke `speak` on `d2`. What happens if you create a `Cat` object and try to cast it to a `Dog`?

Your answer :

We cannot cast Dog to Cat because they are different classes. If they both extended a class like "Animal" I believe it would work.

What methods do you need to add to `BankAccount` to implement `Comparable`?

2.1. *The `Comparable` interface is a commonly used interface in Java. Look up the `Comparable` interface in the API documentation.*

If you wanted to modify the `BankAccount` class so that it implements the `Comparable` interface, what method(s) do you need to implement?

Give the method signature(s), that is, the return type(s), the method name(s), and the method parameter(s) needed.

Your answer :

int compareTo(T o)

Return type: int

Method name: compareTo

Method parameters: the object to compare to

Implement `compareTo` for `BankAccount`

2.2. *The `compareTo` method compares two parameters, the implicit and explicit parameter. The call a.compareTo(b) returns*

a positive integer if `a` is larger than `b`

a negative integer if `a` is smaller than `b`

0 if `a` and `b` are the same

Implement the `compareTo` method of the `BankAccount` class so that it compares the balances of the accounts. Some of the code has been provided for you:

```
public class BankAccount implements Comparable<BankAccount>
{
    private double balance;

    . . .

    /**
        Compares two bank accounts.
        @param other the other BankAccount
       @return 1 if this bank account has a greater balance than the other one,
        -1 if this bank account is has a smaller balance than the other one,
        and 0 if both bank accounts have the same balance
    */
    public int compareTo(BankAccount other)
    {
        . . .
    }
}
```

Your answer :

```
public int compareTo(BankAccount other) {
    if (this.balance > other.balance) {
        return 1;
    } else if (this.balance < other.balance) {
        return -1;
    }

    return 0;
}
```

Sorting bank accounts

2.3. *The `sort` method of the `Collections` class can sort a list of objects whose classes implement the `Comparable` interface.*

Here is the outline of the required code.

```
import java.util.ArrayList;
import java.util.Collections;
 . . .

// Put bank accounts into a list
ArrayList<BankAccount> list = new ArrayList<BankAccount>();
list.add(ba1);
list.add(ba2);
list.add(ba3);

// Call the library sort method
Collections.sort(list);

// Print out the sorted list
for (int i = 0; i < list.size(); i++)
{
    BankAccount b = list.get(i);
    System.out.println(b.getBalance());
}
```

Using this outline, write a test program that sorts a list of five bank accounts.

Your answer :

```
import java.util.*;

public class SuperSecretBank {
    public static void main(String[] args) {
        ArrayList<BankAccount> superSecretAccounts = new Ar-
rayList<BankAccount>();
        superSecretAccounts.add(new BankAccount(9281));
        superSecretAccounts.add(new BankAccount(82819273));
        superSecretAccounts.add(new BankAccount(8438492));
        superSecretAccounts.add(new BankAccount(0.01));
        superSecretAccounts.add(new BankAccount(8483));
        Collections.sort(superSecretAccounts);
    }
}
```

Modifying the sorting criterion in compareTo

2.4. *Change your* `compareTo` *method by switching the positive and negative return values. Recompile and run the test program again. What is the outcome of executing your test program? Explain the changed output.*

Your answer :

It will sort the list in the opposite order. A bank account with a lower balance will technically be greater than a bank account with a higher balance.

Why can't you sort rectangles?

3.1. *Consider the following program that sorts* `Rectangle` *objects:*

```
public class SortDemo
{
    public static void main(String[] args)
    {
        Rectangle rect1 = new Rectangle(5, 10, 20, 30);
        Rectangle rect2 = new Rectangle(10, 20, 30, 15);
        Rectangle rect3 = new Rectangle(20, 30, 45, 10);

        // Put the rectangles into a list
        ArrayList<Rectangle> list = new ArrayList<Rectangle>();
        list.add(rect1);
        list.add(rect2);
        list.add(rect3);

       // Call the library sort method
        Collections.sort(list);

        // Print out the sorted list
        for (int i = 0; i < list.size(); i++)
        {
            Rectangle r = list.get(i);
            System.out.println(r.getWidth() + " " + r.getHeight());
        }
    }
}
```

When you compile the program, you will get an error message. What is the error message? What is the reason for the error message?

Your answer :

There was no Rectangle imported, so I assume it is “java.awt.Rectangle” and imported that.

This is the error I got:

```
error: no suitable method found for sort(ArrayList<Rectangle>)

        Collections.sort(list);
```

What methods are required to implement Comparator?

3.2. *Unfortunately, you cannot modify the* `Rectangle` *class so that it implements the* `Comparable` *interface. The* `Rectangle` *class is part of the standard library, and you cannot modify library classes.*

Fortunately, there is a second `sort` *method that you can use to sort a list of objects of any class, even if the class doesn't implement the* `Comparable` *interface.*

```
Comparator<T> comp = . . .;
    // for example, Comparator<Rectangle> comp = new RectangleComparator();
Collections.sort(list, comp);
```

`Comparator` *is an interface. Therefore,* `comp` *must be constructed as an object of some class that implements the* `Comparator` *interface.*

What method(s) must that class implement? (: Look up the `Comparator` *interface in the API documentation.)*

Your answer :

```
int compare(T o1, T o2)

boolean equals(Object obj)
```

Implement a Rectangle comparator

3.3. *Implement a class* `RectangleComparator` *whose* `compare` *method compares two rectangles.*

The method should return:

a positive integer if the area of the first rectangle is larger than the area of the second rectangle

a negative integer if the area of the first rectangle is smaller than the area of the second rectangle

0 if the two rectangles have the same area

Part of the code has been provided for you below:

```
import java.util.Comparator;
import java.awt.Rectangle;

public class RectangleComparator implements Comparator<Rectangle>
{
    /**
        Compares two Rectangle objects.
        @param r1 the first rectangle
        @param r2 the second rectangle
        @return 1 if the area of the first rectangle is larger than the area of
                the second rectangle, -1 if the area of the first rectangle is
                smaller than the area of the second rectangle or 0 if the two
                rectangles have the same area
    */
    public int compare(Rectangle r1, Rectangle r2)
    {
        . . .
    }
}
```

Your answer :

```
import java.util.Comparator;
import java.awt.Rectangle;


public class RectangleComparator implements Comparator<Rectangle> {
```

```
    /**
     * Compares two Rectangle objects.
     * @param r1 the first rectangle
     * @param r2 the second rectangle
* @return 1 if the area of the first rectangle is larger than the area of the second
rectangle,
*         -1 if the area of the first rectangle is smaller than the area of the
second rectangle,
     *         or 0 if the two rectangles have the same area
     */
    public int compare(Rectangle r1, Rectangle r2) {
        double area1 = r1.getWidth() * r1.getHeight();
        double area2 = r2.getWidth() * r2.getHeight();

        if (area1 > area2) {
            return 1;
        } else if (area1 < area2) {
            return -1;
        }

        return 0;
    }
}
```

Testing the rectangle comparator

3.4. *Write a test program that adds the three rectangles below to a list, constructs a rectangle comparator, sorts the list, and prints the sorted list and the expected values.*

```
    Rectangle rect1 = new Rectangle(5, 10, 20, 30);
    Rectangle rect2 = new Rectangle(10, 20, 30, 15);
    Rectangle rect3 = new Rectangle(20, 30, 45, 10);
```

What is your test program?

Your answer :

```
import java.awt.Rectangle;
import java.util.*;

public class SortDemo {
    public static void main(String[] args) {
        ArrayList<Rectangle> list = new ArrayList<Rectangle>();
```

```
        Rectangle rect1 = new Rectangle(5, 10, 20, 30);
        Rectangle rect2 = new Rectangle(10, 20, 30, 15);
        Rectangle rect3 = new Rectangle(20, 30, 45, 10);

        list.add(rect1);
        list.add(rect2);
        list.add(rect3);

        Comparator<Rectangle> comparator = new RectangleComparator();
        Collections.sort(list, comparator);

        // Print out the sorted list
        for (int i = 0; i < list.size(); i++) {
            Rectangle r = list.get(i);
            System.out.println(r.getWidth() + " " + r.getHeight());
        }
    }
}

class RectangleComparator implements Comparator<Rectangle> {

    /**
     * Compares two Rectangle objects.
     * @param r1 the first rectangle
     * @param r2 the second rectangle
     * @return 1 if the area of the first rectangle is larger than the area of
the second rectangle,
     *         -1 if the area of the first rectangle is smaller than the area of
the second rectangle,
     *         or 0 if the two rectangles have the same area
     */
    public int compare(Rectangle r1, Rectangle r2) {
        double area1 = r1.getWidth() * r1.getHeight();
        double area2 = r2.getWidth() * r2.getHeight();

        if (area1 > area2) {
            return 1;
        } else if (area1 < area2) {
            return -1;
        }

        return 0;
    }
}
```

Making Rectangle comparator an inner class

3.5. *A very specialized class, such as the* `RectangleComparator`*, can be defined inside the method that uses it.*

Reorganize your program so that the `RectangleComparator` *class is defined inside the* `main` *method of your test class.*

What is your `main` *method now?*

Your answer :

public static void main(String[] args) {

```
        // yo this is so ugly don't make us do this again!!!
        class RectangleComparator implements Comparator<Rectangle> {

            /**
            * Compares two Rectangle objects.
            * @param r1 the first rectangle
            * @param r2 the second rectangle
            * @return 1 if the area of the first rectangle is larger than the area
of the second rectangle,
            *         -1 if the area of the first rectangle is smaller than the
area of the second rectangle,
            *         or 0 if the two rectangles have the same area
            */
            public int compare(Rectangle r1, Rectangle r2) {
                double area1 = r1.getWidth() * r1.getHeight();
                double area2 = r2.getWidth() * r2.getHeight();

                if (area1 > area2) {
                    return 1;
                } else if (area1 < area2) {
                    return -1;
                }

                return 0;
            }
        }

        ArrayList<Rectangle> list = new ArrayList<Rectangle>();

        Rectangle rect1 = new Rectangle(5, 10, 20, 30);
        Rectangle rect2 = new Rectangle(10, 20, 30, 15);
        Rectangle rect3 = new Rectangle(20, 30, 45, 10);

        list.add(rect1);
        list.add(rect2);
        list.add(rect3);

        Comparator<Rectangle> comparator = new RectangleComparator();
        Collections.sort(list, comparator);
```

```
        // Print out the sorted list
        for (int i = 0; i < list.size(); i++) {
            Rectangle r = list.get(i);
            System.out.println(r.getWidth() + " " + r.getHeight());
        }
    }
```

4. *There are at least two good reasons why you might choose to build a class as an inner class inside of another containing class:*

1) The inner class will never be used to instantiate objects outside of the containing class, so we are effectively hiding the inner class from other classes.

2) The inner class needs access to data that is defined inside the containing class. In this case, the inner class is positioned in the scope of the shared variables of the containing class. As a result the containing class variables don't need to be passed to the inner class. This is particularly handy if lots of information must be shared, as is often the case for listener classes in a graphics application.

There is one good reason not to build inner classes: Inner classes break the object-oriented notion of "data hiding" – the idea that class variables are private and are only accessible by public accessor and mutator methods in the same class. When we build inner classes we should remember that we are breaking an important object-oriented design principle and we should have a compelling reason to design in this way.

In this problem we will gain some experience working with inner classes. The `Person` *class below contains an inner class called* `Memory` *in which we can store* `String` *objects that represent events in the life of a* `Person` *object. We have decided that class* `Memory` *will never be used outside of* `Person` *and that the* `Person` *class contains data that is needed by the* `Memory` *class. Here is the code.*

```
import java.util.*;

public class Person
{
    private String name;
    private int age;
    private Memory mem;

  public Person(String name, int age)
    {
        this.name = name;
        this.age = age;
        mem = new Memory();
    }

    public String toString()
    {
        return "Name:   " + name + '\n' +
```

```
                "Age:   " + age + '\n';
    }

    public String getName()
    {
        return name;
    }

    public int getAge()
    {
        return age;
    }

    public void tellAll()
    {
        mem.dumpMemory();
    }

    public void rememberAnEvent(String s)
    {
         mem.addLifeData(s);
    }

    // Start of inner class Memory
    private class Memory
    {
        ArrayList<String> lifeData;

     private Memory()
        {
            lifeData = new ArrayList<String>();
            lifeData.add("Name: " + name);
            lifeData.add("Age:   " + age);
        }

        public void addLifeData(String datum)
        {
            lifeData.add(datum);
        }

        public void dumpMemory()
        {
            for (String s: lifeData)
            {
                System.out.println(s);
            }
        }
    }
}
```

Here is a driver for the `Person` class.

```
public class PersonRunner
{
    public static void main(String[] args)
    {
        Person aperson = new Person("Bob", 33);
        aperson.tellAll();
        aperson.rememberAnEvent("I was born in 1970.");
        aperson.rememberAnEvent("I finished school in 2003.");
        aperson.tellAll();
    }
}
```

Notice that the private data in `Person` includes instance variables `name` and `age`. Look inside the inner class `Memory` and notice that `name` and `age` are referenced directly in the `Memory` constructor without invoking an accessor method. This is only possible because the `Memory` constructor is in the scope of `name` and `age`.

In order to understand the effects of an inner class, remove class `Memory` from `Person` and make it a standalone class. Make all the changes that are needed in each class to make the new design work.

Your answer :

```
import java.util.ArrayList;

public class Person {
    private String name;
    private int age;
    private Memory mem;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
        mem = new Memory(name, age);
    }

    public String toString() {
        return "Name: " + name + '\n' +
                "Age: " + age + '\n';
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public void tellAll() {
        mem.dumpMemory();
    }
```

```
    public void rememberAnEvent(String s) {
        mem.addLifeData(s);
    }
}

class Memory {
    private ArrayList<String> lifeData;

    public Memory(String name, int age) {
        lifeData = new ArrayList<String>();
        lifeData.add("Name:" + name);
        lifeData.add("Age:" + age);
    }

    public void addLifeData(String datum) {
        lifeData.add(datum);
    }

    public void dumpMemory() {
        for (String s : lifeData) {
            System.out.println(s);
        }
    }
}
```