OOP Design and Interfaces
IFT 194: HW 4

Brandon Doyle
bdoyle5@asu.edu
1215232174

Dr. Usha Jagannathan
Usha.Jagannathan@asu.edu

July 26, 2018

Summary

View the source of this document on GitHub.

7.1

We're asked to write a method that accepts two integer parameters and returns their average as a floating-point value. See Figure 1 for my solution below.

```java
package hw_4;

public class Average2
{
    public static void main(String[] args)
    {
        // Demonstrate average of 2 integers
        System.out.printf("%.2f\n", average2(5, 8));

        // Demonstrate average of 3 integers
        System.out.printf("%.2f\n", average3(5, 8, 8));
    }

    /**
     * Compute the average of two integers.
     *
     * @param a First integer.
     * @param b Second integer.
     * @return  A double representing the average of a and b.
     */
    public static double average2(int a, int b)
    {
        return (a + b) / 2.0;
    }

    /**
     * Compute the average of three integers.
     *
     * @param a First integer.
     * @param b Second integer.
     * @param c Third integer.
     * @return  A double representing the average value of the 3 input integers.
     */
    public static double average3(int a, int b, int c)
    {
        return (a + b + c) / 3.0;
    }
}
```

Figure 1: Average2.java

7.2

See again Figure 1 for my solution to finding the average of three integers. It is of course simple to extend the solution to an arbitrary number of arguments by writing a variadic method.

7.10

In Java, variables are pass-by-value. We just need to keep in mind whether the variable we're working with represents a reference to an object or contains a primitive value. Both references to objects and primitive values are *copied* to a method's scope. See also Figure 2 for a demonstration.

For example, the `void`-return `addOne(int[] arr)` and `addOne(ArrayList<Integer> arr)` methods have the ability to update entries of their input because the methods' argument variables

reference the same objects as those in `main` that were used to call the method.

On the other hand, calling `primAddOne` does not modify the value of `x` in the `main` method because we're effectively passing a *copy* of the primitive value to the method.

```java
package hw_4;

import java.util.ArrayList;

public class ArgumentDifferences
{
    public static void main(String[] args)
    {
        // Primitive list
        int[] arr = new int[10];
        for (int i = 0; i < arr.length; ++i) arr[i] = i;

        // List of objects
        var anotherArr = new ArrayList<Integer>();
        for (int i = 0; i < arr.length; ++i) anotherArr.add((Integer)i);

        for (int i : arr) System.out.printf("%d ", arr[i]);
        System.out.println();

        addOne(arr);

        for (int v : arr)
            System.out.printf("%d ", v);
        System.out.println();

        addOne(anotherArr);

        for (Integer v : anotherArr)
            System.out.printf("%d ", v);
        System.out.println();

        int x = 5;

        primAddOne(x);

        System.out.println(x);
    }

    /**
     * Add one to every element in an array.
     *
     * @param arr  An array of integers.
     */
    public static void addOne(int[] arr)
    {
        for (int i = 0; i < arr.length; arr[i]++, i++);
    }

    /**
     * Attempt to update every element of an ArrayList.
     *
     * @param arr  An ArrayList of integers.
     */
    public static void addOne(ArrayList<Integer> arr)
    {
        for (int i = 0; i < arr.size(); ++i)
            arr.set(i, (Integer)(i + 1));
    }

    /**
     * Attempt to update a number.
     *
     * @param number  Some integer.
     */
    public static void primAddOne(int number)
    {
        number += 1;
    }
}
```

Figure 2: ArgumentDifferences.java

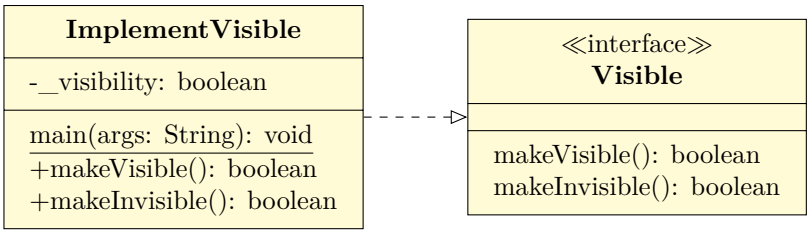

Figure 5: UML diagram of the code presented in Figure 4.

7.11

A static method is essentially instance-independent. In other words, the method's scope is practically anything that is not associated with an instance of its parent class. Therefore, in extension, these methods do not have access to data contained within particular instances of a class.

7.12

Yes, you can implement two interfaces with the same method signature. For example, consider Figure 3, in which I've written a conglomeration of interfaces and a class that implements them. I did, however, discover a conflict while implementing interfaces with default method declarations, which is why the `Example3` interface indeed `extends` the others. This ensures the method declarations are overridden.

7.13

In this section we're tasked with writing an interface `Visible` with two methods. Please see my solution in Figure 4. There may of course be more complex (or colorful) solutions to this implementation, but I chose a pretty straightforward one. The requirement is simply to write definitions for all (non-default) methods provided in the `interface` definition.

7.14

Here we're tasked with drawing a UML (Unified Modeling Language) diagram of the code in Section 7.13. See Figure 5 for my solution.

7.15

We're asked to write an interface with method declarations that should indicate whether an instance that implements it is broken. See Figure 6 for my solution.

```java
package hw_4;

/**
 * Attempt to implement two interfaces with duplicate methods.
 *
 * @author Brandon Doyle
 */
public class Implementer implements Example1, Example2, Example3
{
    public static void main(String[] args)
    {
        var inst = new Implementer();
        System.out.println(inst.addOne(5));
        System.out.println(inst.addAnother(6));
    }

    @Override
    public int addAnother(int x)
    {
        return x + 1;
    }
}

/**
 * An example interface with an ‘addOne‘ method declaration.
 *
 * @author Brandon Doyle
 */
interface Example1
{
    int addOne(int x);

    int addAnother(int x);
}

/**
 * Another interface with a repeatede ‘addOne‘ method.
 *
 * @author Brandon Doyle
 */
interface Example2
{
    int addOne(int x);

    int addAnother(int x);
}

/**
 * Let’s give default method declarations a try!
 *
 * @author Brandon Doyle
 */
interface Example3 extends Example1, Example2
{
    @Override
    default int addOne(int x)
    {
        return x + 1;
    }
}
```

Figure 3: Implementer.java. This example also demonstrates how the `main` method, which is a `static` method, is indeed not paired with any particular instance of `Implementer`; instead, we must create an instance to act upon.

```java
package hw_4;

/**
 * Implement the ‘Visible‘ interface provided below.
 *
 * @author Brandon Doyle
 */
public class ImplementVisible implements Visible
{
    private boolean _visibility = false;

    public static void main(String[] args)
    {
        var inst = new ImplementVisible();
        inst.makeInvisible();
        inst.makeVisible();
    }

    /**
     * Override these two methods from our ‘Visible‘ interface.
     */
    @Override
    public boolean makeVisible()
    {
        this._visibility = true;
        return this._visibility;
    }

    @Override
    public boolean makeInvisible()
    {
        this._visibility = false;
        return this._visibility;
    }
}

/**
 * Our interface.
 *
 * @author Brandon Doyle
 */
interface Visible
{
    boolean makeVisible();

    boolean makeInvisible();
}
```

Figure 4: ImplementVisibile.java

```java
package hw_4;


/**
 * An example of using my ‘Breakable‘ interface.
 *
 * @author Brandon Doyle
 */
public class Something implements Breakable
{
    private boolean _broke = false;

    public static void main(String[] args)
    {
        var inst = new Something();
        System.out.println("This instance is broken: " + inst.broken());
        inst.breakIt();
        System.out.println("This instance is broken: " + inst.broken());
        inst.unBreakIt();
        System.out.println("This instance is broken: " + inst.broken());
    }

    /**
     * If the instance is broken, ‘un-break’ it.
     */
    public void unBreakIt()
    {
        if (this._broke)
            this._broke = false;
    }

    /**
     * Break this instance of ‘Something‘.
     */
    @Override
    public void breakIt()
    {
        if (!this._broke)
            this._broke = true;
    }

    /**
     * Indicate whether this ‘Something‘ instance is broken.
     */
    @Override
    public boolean broken()
    {
        return this._broke;
    }
}


/**
 * Break somethin’.
 *
 * @author Brandon Doyle
 */
interface Breakable
{
    // Modified identifier to ‘breakIt‘ due to misuse of ‘break‘ keyword.
    void breakIt();

    boolean broken();
}
```

Figure 6: Something.java