Object-Oriented Programming

Worksheet 2

Tom Lin Jamie Willis

February 1, 2021

1 Equality and Reference

1. Utility classes like `java.lang.Math` are common. They contain no state, are not instantiable, and only contains static methods. They are simply used as a namespace to group related methods together.

   (a) Create a utility class called `TestUtils`, and implement the following static helpers:

   - `static void assertEquals(String, String, String)` - checks whether the 1st parameter (the *expected* value) is equal to the 2nd parameter (the *actual* value) using string equality. If the strings are not equal, throw an `AssertionError` with the 3rd parameter as a prefix and append as much detail as possible to describe what the *expectation* was and what *actual* value was given (i.e *`error`*`:expecting` *`expected`* `but was` *`actual`*).
   - `static void assertEquals(int, int, String)` - same as the string version but checks equality for ints instead.

   Test your implementation appropriately.

   (b) Implement the method `assertUniversalEquals` in `TestUtils`, the method should take the same amount of parameters like `assertEquals` but it should be able to compare **any** (both primitives and objects) two types, the following should compile:

   ```
   assertUniversalEquals(1, 2, "Err!");
   // Err!: expecting 1(class java.lang.Integer) but got 2(class java.lang.Integer)
   assertUniversalEquals("1", "2", "Err!");
   // Err!: expecting 1(class java.lang.String) but got 2(class java.lang.String)
   assertUniversalEquals("1", null, "Err!");
   // Err!: expecting 1(class java.lang.String) but got null
   assertUniversalEquals(1, "1", "Err!");
   // Err!: expecting 1(class java.lang.Integer) but got 1(class java.lang.String)
   assertUniversalEquals(null, null, "Err!"); // OK
   ```

   Test your implementation, you may copy the above snippet to verify behavior.

⋈† (c) `assertUniversalEquals` allows checking equality on any two type. Implement a version of `assertEquals` that ensures the expected and actual values have the same type at compile-time.

2. Explain the difference between object *reference* equality and object *value* equality. State their relation with the `==` operator and `Object.equals`[1].

3. Java's method invocation is *pass-by-value*, explain what this means in practice.

4. Demonstrate the effects of pass-by-value by implementing the following methods that all append the string `"b"` to the lone parameter, ideally we want to get the appended result back somehow without mutating fields; in cases where it would not be possible, explain why.

   (a) `void append(String a)`

   (b) `String append(final String a)`

   (c) `void concat(StringBox a)` where `StringBox` is `class StringBox { String value; }`

[1] `https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-`

2 Arrays

1. State whether an **existing** array can be resized.
2. State whether an array is considered an object or a primitive (Hint: Check the JLS §4).

3 Interfaces and abstract class

1. Create the interface `Formatter`, the purpose of this interface is to allow formatting of strings in a generic way. The interface should contain only one method: `String format(String)`.

   (a) Create the following implementations of the `Formatter` class:

   - `IdentityFormatter` - returns the input
   - `UppercaseFormatter` - returns to input but with all characters in uppercase
   - `ROT13Formatter` - returns the *ROT13*[2] encoded value of the input

   Test your implementation appropriately.

   (b) Create new instances of your formatter and place them in an array, print out the formatted version of the string `"Hello world!"` by iterating through the array. For `IdentityFormatter`, `UppercaseFormatter`, and `ROT13Formatter` respectively, the output should look like:

   ```
   Hello world!
   HELLO WORLD!
   Uryyb jbeyq!
   ```

†† (c) `Formatter` can be considered a form of *strategy pattern*. Compare and contrast this with *higher order functions*.

2. Unlike classes, an interface allows inheriting from multiple interfaces. Consider the following snippet:

   ```
   class A{}
   class B{}
   class AAndB extends A, B{} // Compile error: class cannot extend from multiple classes
   interface C{}
   interface D{}
   interface CAndD extends C, D{} // Compiles
   ```

   First, state why multiple inheritance is problematic for classes. Then, discuss why the same constraint does not apply to interfaces.

3. Consider the following snippet:

   ```
   interface Bird{ void fly(); }
   class Dodo implements Bird{ public void fly(){ System.out.println("I'm extinct"); } }
   class Kiwi implements Bird{ public void fly(){ System.out.println("I can't fly"); } }
   class BigExtinctKiwi extends Dodo, Kiwi{ } // does not compile
   ```

   Provide a way for `BigExtinctKiwi` to retain behaviors of both `Dodo` and `Kiwi` while still conforming to the `Bird` interface. The order of behavior is not important.

4. Discuss the differences between a class, an abstract class, and an interface. Clearly state when and where one would be preferred over the other.

4 Invariants

1. An invariant is some property that is usually established by the constructor and maintained by all public methods of a class. Consider the following mutable triangle class:

[2] `https://en.wikipedia.org/wiki/ROT13`

```java
package solutions.sheet2;

class MutableTriangle {

   private String kind;
   private double area;
   private int a, b, c;

   public MutableTriangle(int a, int b, int c) {
      this.a = a;
      this.b = b;
      this.c = c;
      this.kind = resolveKind(a, b, c);
      this.area = computeArea(a, b, c);
   }

   public void setA(int a) { this.a = a; }
   public void setB(int b) { this.b = b; }
   public void setC(int c) { this.c = c; }
   public int getA() { return a; }
   public int getB() { return b; }
   public int getC() { return c; }
   public String getKind() { return kind; }
   public double getArea() { return area; }

   static String resolveKind(int a, int b, int c) {
      if (a <= 0 || b <= 0 || c <= 0) return "Illegal";
      else if (a == b && b == c) return "Equilateral";
      else if (a + b < c || b + c < a || c + a < b) return "Impossible";
      else if (a + b == c || b + c == a || c + a == b) return "Flat";
      else if (a * a + b * b == c * c ||
            c * c + b * b == a * a ||
            c * c + a * a == b * b) return "RightAngled";
      else if (a == b || b == c || c == a) return "Isosceles";
      return "Scalene";
   }

   static double computeArea(int a, int b, int c) { // heron's formula
      double s = (a + b + c) / 2.0d;
      return Math.sqrt((s * (s - a) * (s - b) * (s - c)));
   }
}
```

Where the `double getArea()` and `void getKind()` method must return the correct area and the kind of triangle respectively for the given side length `a`, `b`, and `c`.

(a) First, identify the invariants, then, identify the methods that break those invariants.

(b) Without making the class immutable (i.e keep the setters), suggest ways of preserving the invariant.

† (c) Suppose another geometry class requires an instance of triangle that must be a equilateral:

```java
package solutions.sheet2;

class ImmutableTetrahedron { // a pyramid with equal side lengths
   private final MutableTriangle equilateral;

   public ImmutableTetrahedron(MutableTriangle equilateral) {
      if (!equilateral.getKind().equals("Equilateral"))
         throw new IllegalArgumentException("A tetrahedron must have equal side
             lengths");
```

```
 9        this.equilateral = equilateral;
10    }
11    public double getVolume() { return (Math.sqrt(2f) / 12) *
          Math.pow(equilateral.getA(), 3f); }
12 }
```

Explain why `ImmutableTetrahedron` is in fact not immutable. Provide a way to make the class actually immutable.

5 Enums

Note: Enums are not yet covered in the lectures but was included in slides of Lecture 6. Feel free to do these and ask questions about them in the labs.

⋈ 1. Give examples of what enums can be used for. State enum's advantages over traditional integer flags.

⋈ 2. Create an enum called `Sign`, the purpose of the class is two represent the sign of a numeric value (i.e `-` for negative), it should have two possible values: `POSITIVE` and `NEGATIVE`. The `toString` should print out `+` for `POSITIVE` and `-` for `NEGATIVE`, it should also contain the method `Sign flip()` that returns the opposite sign relative to the current sign.