Object-Oriented Programming

Worksheet 1

Tom Lin Jamie Willis

February 1, 2021

1 Introduction

These worksheets are designed to give you practice for the COMS10017 Object-Oriented Programming course. If you have taken the Functional programming course, these will be very similar in format to the Haskell worksheets.

You are encouraged to sit in groups and work together to complete these worksheets. Answers will be released together with the next worksheet. Be aware that while these worksheets are formative, there is a summative assignment where you must work *in your pairs* to avoid plagiarising.

Difficulties will be shown by daggers and symbols:

No daggers	Straightforward question. Reviewing the lecture slides should get you through these.
†	More involved questions. Answering these required some thought and may not be immediately clear from materials in the lectures.
††	Hard and challenging questions. There may be multiple solutions and will take you some time.
†††	Hard, time-consuming and potentially open-ended questions. Try these if you are feeling ambitious.
⋈	Questions that could be better solved with more advanced language features. You may not be able to solve these cleanly right now, and they are worth revisiting later.

You are encouraged to attempt all questions regardless of difficulty.

2 Objects and Classes

1. Suppose we have a real life generic computer keyboard with which has the traditional LEDs for `CapsLock`, `NumLock`, and `ScrollLock`.

   (a) Informally, identify all potential states this keyboard may have. These could be hardware implementation details or simply appearance.

   (b) Informally, from a computer's standpoint, identify all properties and behaviors this keyboard should have.

2. Suppose you've written a very basic program that prints the string `Hello, World!`:

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, world!");
   }
}
```

   (a) Explain the significance of all the *modifiers* (e.g. `public`, `static`) for the *method* and *class* declarations.

† (b) Discuss why an entry point like `main`, in the context of OOP, makes sense to be marked `static`.

3. A critter has a name and responds to the environment. For example, a cat says "meow" when poked. Critters also eat other critters to stay alive.

   (a) Write a class called `Critter` that when called with `poke()` prints `"I was poked"` to console (stdout).

(b) Allow the `Critter` to be named such that calling `poke()` prints `"Cow was poked"` where `Cow` is the name. Make sure the name can never change.

(c) Allow the `Critter` to eat other critters by having a method called `eat(Critter c)`. The predator should print to the console detailing the situation.

4. Language design usually has trade-offs and limitations. For example, instead of having objects only, Java opted for having primitives for performance reasons.

   (a) First, explain the difference between a class and an instance of a class.

   (b) Then, explain the purpose of primitives and compare it to classes and instances.

† (c) Explain what `null` is. Give example of a type where null is not a valid value.

3 Smart data

1. Consider a 2-dimensional vector (Euclidean vector):

$$x = (x_1, x_2)$$

   This can be represented in the object-oriented world using a class.

   (a) Create a class called `Vector2D` which stores two doubles to form a 2D vector. To ensure correctness, create a separate class called `Vector2DTest` and test your `Vector2D` there. The class should implement the following methods:

   - `double distance(Vector2D v)` – the distance [1] between this and the other vector, you may want to look into the built-in math methods in `java.lang.Math`.
   - `Vector2D add(Vector2D v)` – the sum of this and the other vector.
   - `Vector2D scale(double f)` – multiply this vector by a fixed factor.

   Also implement the `String toString()`[2] method for the class which returns an appropriate representation of the class as a `String` (similar to `Show` in Haskell).

   Note that none of the methods should mutate (change) the values of fields but return a new `Vector2D` instead. Concretely, the following should be observed:

```
Vector2D a = new Vector2D(10.0, 10.0);
System.out.println(a.toString()); // "Vector2D(10.00, 10.00)"
System.out.println(a.add(a).toString()); // "Vector2D(20.00, 20.00)"
System.out.println(a.toString()); // "Vector2D(10.00, 10.00)"
```

   (b) The fields in `Vector2D` should never be mutated after assignment in the constructor. By marking those fields private, we shield them from mutation from the users of the class, but you can still accidentally mutate them within the class. State how this can also be avoided.

† 2. The methods defined in `Vector2D` present a very nice property: we can chain method calls to form an expression that is similar to built-in numeric operators (e.g. `+-*/`):

```
double x = new Vector2D(10.0, 10.0)
   .add(new Vector2D(5.0, 5.0))
   .add(new Vector2D(5.0, 5.0))
   .scale(3)
   .distance(new Vector2D(5.0, 5.0));
```

   Because `Vector2D` is immutable, the class can be considered *referentially transparent*, meaning that substitution of any variable with the expression retains the same semantic. This property enables equational reasoning which makes code less error-prone.

[1] $\textbf{distance}(x, y) = \textbf{distance}(y, x) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$

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

The opposite of immutable classes is mutable classes, where the internal state can change and thus they are *referentially opaque*. This may be desirable under very limited circumstances for performance or in cases where the underlying domain is better modeled with mutability.

(a) Create a new class `IntBox` which contains a single `int x` which is **not** final. The constructor should accept the starting value of `x`, and methods should modify `x` directly.

The class should implement the following methods:

- `IntBox add(int v)` – adds `v` to the current value
- `IntBox subtract(int v)` – subtracts `v` from the current value
- `IntBox multiply(int v)` – multiplies `v` with the current value

Remember to implement the `toString()` method so that the values inside the class can be observed easily. The following should be observed:

```
IntBox a = new IntBox(10).add(10);
System.out.println(a); // "IntBox(20)"
System.out.println(a.multiply(10)); // "IntBox(200)"
System.out.println(a); // "IntBox(200)"
```

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

† (b) A *copy constructor* is a kind of constructor that takes the class it resides in as parameter and copies the content(state) of the class. For a class with one field, it would look something like:

```
class Foo{
   int a;
   Foo(int a){ this.a = a; }
   Foo(Foo that){ this.a = that.a; } // copy constructor
}
```

This is useful for creating snapshots of a mutable class like `IntBox`. Implement a copy constructor for your `IntBox` class.

4 Control structures

1. Create a class, called `Triangle`, which represents a triangle with three sides. The class should be initialisable with three `int`s and should implement the `String toString()` method.

   (a) Implement the method `triangleType()` taking no parameters and returning the type of the triangle as a `String`. Possible return values are: `"Equilateral"`, `"Isosceles"`, `"Flat"`, `"Impossible"`, `"Scalene"`, and `"RightAngled"`. Change your `toString()` method by additionally calling this method so that the text representation also includes the triangle type.

† (b) Implement the method `static void makeTriangles(int n)`, which should print out permutations of triangles with sides of length from 1 to `n`. For example, `makeTriangles(2)` prints:

```
Equilateral(1, 1, 1)
Flat(1, 1, 2)
Flat(1, 2, 1)
Isosceles(1, 2, 2)
Flat(2, 1, 1)
Isosceles(2, 1, 2)
Isosceles(2, 2, 1)
Equilateral(2, 2, 2)
```

Explain why this method should be marked `static`.

2. Exception is an unusual control structure. The concept is similar to multiple returns from a method where failure is a possible value (e.g. *Go*). An equivalent to exceptions is to return a value that could be either the result or an error (similar to Haskell's `Either`).

†† (a) Write the method `static ErrorOrResult parseIntOrFail(String input)`, where the input string will be parsed into an integer; the `ErrorOrResult` class should allow either the integer or the reason of failure to be retrieved. It should contain the following methods:

- `boolean failed()` – whether the parsing failed.
- `String error() throws NoSuchElementException` – returns the error message if the result is a failure or throw a `NoSuchElementException`.
- `int result() throws NoSuchElementException` – returns the integer if parsing succeeded or throw a `NoSuchElementException`.

You may reuse `java.lang.Integer.parseInt(String)` to implement this method. Remember to implement `toString` and test your implementation suitably.

(b) Explain the difference between a checked exception, and an unchecked exception.

(c) Demonstrate rethrowing of an exception.