OOP Examples Sheet 1

Dr Andrew Rice
Michaelmas 2019
(Material written by Dr Robert Harle)

These exercises follow the notes and are intended to provide material for supervisions. For the majority of students this course has two challenges: the first is understanding the core OOP concepts; the second is applying them correctly in Java. The former can be addressed through traditional academic study; the latter requires a combination of knowledge and experience. To get the most out of this course you will need to put in effort programming *from scratch* (not just copy/pasting from StackOverflow..!).

Questions marked (**W**) are warm up questions. They mostly ask you to regurgitate the notes just to check your core understanding. Questions marked (*) are intended to be more time consuming and/or challenging. Consult your supervisor for an appropriate set for you.

Practical exercises are listed on the course materials page on the department webpages.

Practical exercises associated with this supervision

- Fibonacci
- Matrices
- Chess
- ProductOfOtherElements (Daily Coding Challenge - optional)
- PalindromePairs (Daily Coding Challenge - optional)

Lecture 1: Solving practical exercises

1.1. **(W)** Set up SSH, git, and IDE and your account on chime

1.2. **(W)** Complete the Fibonacci task that was shown in lectures

(a) Why would one argue that the provided tests for Fibonacci are not sufficient?
(b) Why would it be a bad idea to write a test that times whether FibonacciTable runs faster than Fibonacci?

1.3. What does instruction coverage mean as a test coverage metric? Give an example showing how 100% instruction coverage does not mean your program is bug free.

Lecture 2: Types, Objects and Classes

2.1. **(W)** Compare and contrast a typical functional langauage and a typical imperative language.

2.2. **(W)** Identify the primitives, references, classes and objects in the following Java code:

```
double d = 5.0;
int i[] = {1,2,3,4};
LinkedList<Double> l = new LinkedList<Double>();
Double k = new Double();
Tree t;
float f;
Computer c = null;
```

2.3. **(W)** Write Java code to test whether your Java environment performs tail-recursion optimisations or not.

2.4. Write a static function lowestCommon that takes two long arguments and returns the position of the first set bit in common, where position 0 is the LSB. If there is no common bit, the function should return -1. For example lowestCommon(14,25) would be 3. Your solution whould use at least one break statement.

2.5. Complete the Matrices programming task

(a) Why is it neccessary for the assertions to have the form assertThat(actual).isWithin(tolerance).of(expected);?
(b) Why have the static factory methods been put in the Shapes class rather than Matrix?

Lecture 3: Designing Classes

3.1. **(W)** Explain why we should use private state in conjuction with public getter and setter methods.

3.2. **(W)** Python does not offer a private access modifier making everything public. Instead, programmers should follow the convention that a variable name should be prefixed with an underscore ("_") to signal it should not be accessed or edited externally. Discuss the advantages and disadvatages of this approach compared to Java.

3.3. This question considers representing a 2D vector in Java.

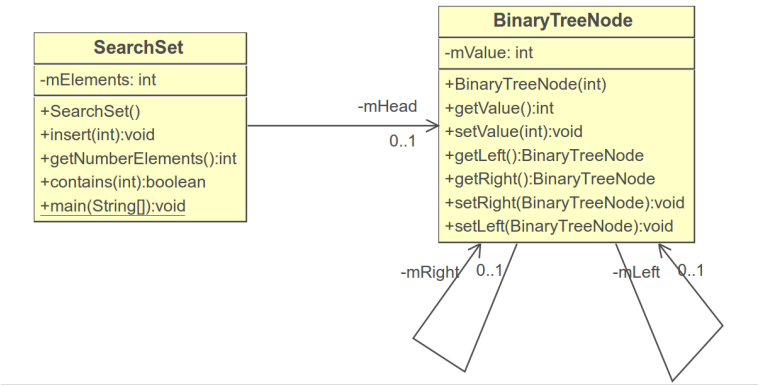
(a) Develop a mutable class Vector2D to embody the notion of a 2D vector based on floats (do not use Generics). At a minimum your class should support addition of two vectors; scalar product; normalisation and magnitude.
(b) What changes would be needed to make it immutable?
(c) Contrast the following prototypes for the addition method for both the (i) mutable, and (ii) immutable versions.
    - `public void add(Vector2D v)`
    - `public Vector2D add(Vector2D v)`
    - `public Vector2D add(Vector2D v1, Vector2D v2)`
    - `public static Vector2D add(Vector2D v1, Vector2D v2)`
(d) How can you convey to a user of your class that it is immutable?

3.4. You met the idea of linked lists in FoCS.

(a) Write a class OOPLinkedList that encapsulates a linked list of integers. Your class should support the addition and removal of elements from the head, querying of the head element, obtaining the n^{th} element and computing the length of the list. You may find it useful to first define a class OOPLinkedListElement to represent a single list element. Do not use Generics.
(b) Give the UML class diagram for your code.

3.5. **(*)** In mathematics, a *set* of integers refers to a collection of integers that contains no duplicates. You typically want to insert numbers into a set and query whether the set contains numbers. One approach is to store the numbers in a binary search tree.

(a) The diagram below represents this approach. Implement it in Java, using the names of entities to decide what they should do. Make your main method test that the code works.



(b) The BinaryTreeNode class can be reused for other solutions. Create a class FunctionalArray that uses BinaryTreeNode to cretae a functional array of ints. Your class should have a constructor that creates a tree of a given size (passed as an argument); a void set(int index, int value) method; and a int get(int index) method. You should make the functional array zero-indexed to match java's normal arrays (i.e. the first element has index 0). Requests for indices outside the limits should result in an exception.

Lecture 4: Pointers, References and Memory

4.1. **(W)** Pointers are problematic because they might not point to anything useful. A null reference doesn't point to anything useful. So what is the advantage of using references over pointers?

4.2. **(W)** Draw some simple diagrams to illustrate what happens with each step of the following Java code in memory:

```
Person p = null;
Person p2 = new Person();
p = p2;
p2 = new Person();
p=null;
```

4.3. **(W)** Explain the result of the following code.

```
public static void add(int[] xy,int dx, int dy) {
  xy[0]+=dx;
  xy[1]+=dy;
}

public static void add(int x,int y,int dx, int dy) {
  x=x+dx;
  y=y+dy;
}

public static void main(String[] args) {
  int xypair[] = {1,1};

  add(xypair[0], xypair[1], 1, 1);
  System.out.println(xypair[0]+" "+xypair[1]);

  add(xypair,1,1);
  System.out.println(xypair[0]+" "+xypair[1]);
}
```

Lecture 5: Inheritance

5.1. **(W)** A student wishes to create a class for a 3D vector and chooses to derive from the Vector2D class (i.e. public void Vector3D extends Vector2D). The argument is that a 3D vector is a "2D vector with some stuff added". Explain the conceptual misunderstanding here.

5.2. **(W)** If you don't specify an access modifier when you declare a member field of a class, what does Java assign it? Demonstrate your answer by providing minimal Java examples that will and will not compile, as appropriate.

5.3. **(W)** Suggest UML class diagrams that could be used to represent the following.Think careully about the directions of and multiplicities on your arrows.

(a) A shop is composed of a series of departments, each with its own manager. There is also a store manager and many shop assistants. Each item sold has a price and a tax rate.
(b) Vehicles are either motor-driven (cars, trucks, motorbikes, electric bikes) or human-powered (bikes, skateboards,scooters). All cars have 3 or 4 wheels and all bikes have two wheels. Every vehicle has an owner. Some vehicles must have road tax.

5.4. Consider the Java class below:

```
package questions;

public class X {
  MODIFIER int value = 3;
};
```

Another class Y attempts to access the field value in an object of type X. Describe what happens at compilation and/or runtime for the range of MODIFIER possibilities (i.e. public, protected, private and unspecified) under the following circumstances:

(a) Y subclasses X and is in the same package;
(b) Y subclasses X and is in a different package;
(c) Y does not subclass X and is in the same package;
(d) Y does not subclass X and is in a different package.

5.5. Create a class OOPSortedLinkedList that derives from OOPLinkedList but keeps the list elements in ascending order.

5.6. **(*)** Create a class OOPLazySortedLinkedList that derives from OOPSortedLinkedList but avoids performing any sorting until data are expressly requested from it, whereupon it first sorts its contents and then returns the result.

Lecture 6: Polymorphism

6.1. **(W)** Explain the differences between a class, an abstract class and an interface in Java.

6.2. **(W)** Explain what is meant by (dynamic) polymorphism in OOP and explain why it is useful, illustrating your answer with an example.

6.3. **(W)** A programming language designer proposes adding 'selective inheritance' whereby a programmer manually specifies which methods or fields are inherited by any subclasses. Comment on this idea.

6.4. **(W)** A Computer Science department keeps track of its CS students using some custom software. Each student is represented by a Student object that features a pass() method that returns true if and only if the student has all 20 ticks to pass the year. The department suddenly starts teaching NS students, who only need 10 ticks to pass. Using inheritance and polymorphism, show how the software can continue to keep all Student objects in one list in code without having to change any classes other than Student.

6.5. Complete the Chess practical exercise

(a) Which are the parts of the program which are poorly designed with switch statements for pieces?

(b) In what ways is your new code easier to maintain than previously? What drawbacks have arisen from this new approach?

6.6. In the second of the shape drawing examples in lectures, we relied on the existence of an instanceof operator that allowed us to check which class the object really was. The built-in ability to do this is called *reflection*. However, not all OOP languages supprt reflection. Show how we could modify the shape classes to allow us to determine the true type without using relection.

6.7. An alternative implementation of a list to a linked list uses an array as the underlying data structure rather than a linked list.

(a) Write down the asymptotic complexities of the array-based list methods.
(b) Abstract your implementation of OOPLinkedList to extract an appropriate OOPList interface.
(c) Implement OOPArrayList (which should make use of your interface).
(d) When adding items to an array-based list, rather than expanding the array by one each time, the array size is often doubled whenever expansion is required. Analyse this approach to get the asymptotic complexities associated with an insertion.

6.8. (*)

(a) Create a Java interface for a standard queue (i.e. FIFO).
(b) Implement OOPListQueue, which should use two OOPLinkedList objects as per the queues you constructed in your FoCS course. You may need to implement a method to reverse lists.
(c) Implement OOPArrayQueue. Use integer indices to keep track of the head and the tail position.
(d) State the asymptotic complexities of the two approaches.

6.9. Imagine you have two classes: Employee (which embodies being an employee) and Ninja (which embodies being a Ninja). You need to represent an employee who is also a ninja (a common problem in the real world). By creating only one interface and only one class (NinjaEmployee), show how you can do this *without* having to copy method implementation code from either of the original classes.