OOP Examples Sheet

Robert Harle

Michaelmas 2025

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..!). While these exercises will be useful, there is no substitute for actually coding. In addition to these questions, you should aim to build small programs that exercise your Java skills.

- 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.

The lectures will proceed at *roughly* one section per lecture. Consult your supervisor for an appropriate set for each supervision.

Section 1: Intro to Java

1.1. On your personal system, install and configure a JDK (OpenJDK recommended) and an IDE (IntelliJ IDEA recommended). Make sure you can build a hello world application and compile and run it in both a terminal and your IDE. Also locate the `jshell` program and create a hello world (one line of code).

1.2. (W) Compare and contrast a typical functional language and a typical imperative language.

1.3. (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;
```

1.4. (W) Naming Conventions. The lecture notes contrast Python's `snake_case` and Java's `CamelCase`. Why is adhering to a language's standard naming convention important? Provide a Java class `StudentReportGenerator` with one `static final` constant `DEFAULT_YEAR` set to `2025`, one instance field `reportTitle`, and one method `generateReport`, all following Java conventions.

1.5. (W)

(a) Write a class `MathUtil` with two overloaded methods named `max`. One should accept two `int` parameters, and the other should accept two `double` parameters.

(b) Explain why you could *not* add a third method `float max(int a, int b)` to this class while the `int max(int a, int b)` method already exists.

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

1.7. 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 should use at least one `break` statement.

1.8. For this questions, you should write procedurally (i.e. use static functions, no need to make custom classes

(a) Write a Java function that creates an array-of-arrays to represent an $n \times n$ matrix of floats. The matrix should be initialised to be a unit matrix.

(b) Write another Java function that transposes such an array-of-arrays matrix. Your function should be in-place i.e. use $O(1)$ space.

1.9. (W) Explain why this code prints 0 rather than 7.

```
public class Test {
    public int x = 0;
    public void Test() {
        x = 7;
    }
    public static void main(String[] args) {
        Test t = new Test();
        System.out.println(t.x);
    }
}
```

Section 2: Class Design and Encapsulation

2.1. (W) Explain why we should use `private` state in conjunction with `public` getter and setter methods.

2.2. (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 disadvantages of this approach compared to Java.

2.3. 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?

2.4. 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.

2.5. (W) Single Responsibility Principle (SRP) The lecture notes discuss the Single Responsibility Principle (SRP). The following `Report` class violates `SRP`. Explain why, and refactor it into separate classes that better adhere to `SRP` and exhibit high cohesion.

```
public class Report {
    private String data;

public void fetchDataFromDatabase() {
    System.out.println("Fetching data...");
    this.data = "raw data";
}

public void parseXMLData() {
    System.out.println("Parsing data as XML...");
    // logic to parse this.data
}

public void printToConsole() {
    System.out.println("--- REPORT ---");
    System.out.println(this.data);
    System.out.println("--- END ---");
}
}
```

2.6. Immutability and Records. The lecture notes introduce the `record` keyword (Java 16+) as a concise way to create immutable classes.

(a) Create an immutable class `StudentRecord` containing name, age and college without using `record`. Provide the class with a `print()` method that prints the details to the screen.

(b) Re-implement `StudentRecord` as a `record`.

(c) The notes mention that a `record` automatically generates `equals()`, `hashCode()`, and `toString()`. Why is it critical that an immutable class used as a data carrier provides a correct, value-based implementation of `equals()` and `hashCode()`? (Hint: consider its use in a `HashSet` or as a key in a `HashMap`).

2.7. (*) 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 text below describes a design for a set of integers. Draw out the UML diagram.

SearchSet

- -numElements: int
- +SearchSet()
- +insert(int): void
- +getNumElements(): int
- +contains(int): boolean
- (relationship) -head (0..1) → **BinaryTreeNode**

BinaryTreeNode

- -value: int
- +BinaryTreeNode(int)
- +getValue(): int
- +setValue(int): void
- +getLeft(): BinaryTreeNode
- +getRight(): BinaryTreeNode
- +setRight(BinaryTreeNode): void
- +setLeft(BinaryTreeNode): void
- (relationship) -right (0..1) → **BinaryTreeNode**
- (relationship) -left (0..1) → **BinaryTreeNode**

(b) Implement this in Java, using the names of entities to decide what they should do. Make your `main` method test that the code works.

(c) 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.

Section 3: Pointers, References and Memory

3.1. (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?

3.2. (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;
```

3.3. (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]);
}
```

3.4. The notes mention the confusion over passing by value and reference. Write Java code to demonstrates that the variable declared by the code `int[] test` can be viewed as a reference that gets copied when passed as an argument.

3.5. A programmer proposes a new imperative language whereby all variables are passed by reference (even those of primitive type). Discuss the advantages and disadvantages of this design.

Section 4: Inheritance

4.1. (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.

4.2. (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.

4.3. (W) Write a small Java program that demonstrates constructor chaining using a hierarchy of three classes as follows: A is the parent of B which is the parent of C. Modify your definition of A so that it has exactly one constructor that takes an argument, and show how B and/or C must be changed to work with it.

4.4. (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.

4.5. 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.

4.6. Consider the following class hierarchy. Explain what will be printed by the `main` method. Why does `a.value` behave differently from `a.printValue()`, even though `a` holds a `Sub` object?

```
public class Super {
    public String value = "Super";

    public void printValue() {
        System.out.println("Super.printValue: " + value);
    }
}

public class Sub extends Super {
    public String value = "Sub"; // Field Shadowing

    @Override
    public void printValue() {
        System.out.println("Sub.printValue: " + value);
    }
}

public class TestShadowing {
```

```
    public static void main(String[] args) {
        Super a = new Sub();
        System.out.println("a.value = " + a.value); a.printValue();
    }
}
```

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

4.8. (*) 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.

Section 5: Polymorphism

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

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

5.3. (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.

5.4. (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`.

5.5. 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.

5.6. An alternative implementation of a 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.

5.7. (*)

(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.

5.8. 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.

Section 6: Object lifecycle, Garbage Collection and Copying Objects

6.1. The lectures described the main mechanism to mark objects that can be deleted (or to mark objects that cannot be deleted), but not the deletion process. Compare and contrast the following approaches in terms of performance. Your answer should consider the costs associated with marking and deleting.

(a) The mark-sweep schemes delete the marked objects, leaving their memory available. A list of free memory chunks is maintained.

(b) The mark-sweep-compact schemes delete the objects and then compact the momory by moving surviving objects to be adjacent to each other (i.e. no free memory between them).

(c) The mark-copy schemes maintain two memory regions, one of which is active. During the marking process, surviving objects are copied into the other region. Once marking is complete, references are upated to the second region, which becomes active, and the first region is deleted.

6.2. It is often recommended that we make classes immutable wherever possible. How would you expect this to impact garbage collection?

6.3. (W) Adapt your `OOPLinkedList` class to be cloneable.

6.4. (W) Explain the purpose of marker interfaces.

6.5. A student forgets to use `super.clone()` in their `clone()` method:

```
public class SomeClass extends SomeOtherClass implements Cloneable {
    private int[] mData;

    public Object clone() {
        SomeClass sc = new SomeClass();
        sc.mData = mData.clone();
        return sc;
    }
}
```

Explain what could go wrong, illustrating your answer with an example.

6.6. Consider the following code:

```
public class MyClass {
    private String mName;
    private int[] mData;

    // Copy constructor
    public MyClass (MyClass toCopy) {
        this.mName = toCopy.mName;
        // TODO
    }
}
```

(a) Complete the copy constructor.

(b) Make `MyClass clone()`-able (you should do a deep copy).

(c) Why might the Java designers have disliked copy constructors?

(d) Under what circumstances is a copy constructor unambiguously a good solution?

6.7. Consider the class below. What difficulty is there in providing a deep `clone()` method for it?

```
public class CloneTest {
    private final int[] mData = new int[100];
}
```

Section 7: Collections, Comparison

7.1. (W) Using the Java API documentation or otherwise, compare the Java classes Vector, LinkedList, ArrayList and TreeSet.

7.2. (W) Write an immutable class that represents a 3D point (x, y, z). Give it a natural order such that values are sorted in ascending order by z, then y, then x

7.3. Write a Java class that can store a series of student names and their corresponding marks (percentages) for the year. Your class should use at least one Map and should be able to output a List of all students (sorted alphabetically); a List containing the names of the top P% of the year as well; and the median mark.

7.4. You need to implement a `RecentFiles` list for an application. This list will store the paths to the 10 most recently opened files. Every time a new file is opened, its path is added to the *start* of the list, and if the list is now over 10 items, the *last* item is removed. Which implementation of `List` — `ArrayList` or `LinkedList` — would be the more performant choice for this specific task, and why?

7.5. (W) Explain why the following code excerpts behave differently when compiled and run (may need some research):

```
String s1 = new String("Hi");
String s2 = new String("Hi");
```

```
System.out.println(s1 == s2);

String s3 = "Hi";
String s4 = "Hi";
System.out.println(s3 == s4);
```

7.6. The user of the class Car below wishes to maintain a collection of Car objects such that they can be iterated over in some specific order.

```
public class Car {
    private String manufacturer;
    private int age;
}
```

(a) Show how to keep the collection sorted alphabetically by the manufacturer without writing a Comparator.

(b) Using a Comparator, show how to keep the collection sorted by (manufacturer, age). i.e. sort first by manufacturer, and sub-sort by age.

7.7. (*) Write a Java program that reads in a text file that contains two integers on each line, separated by a comma (i.e. two columns in a comma-separated file). Your program should print out the same set of numbers, but sorted by the first column and subsorted by the second.

Section 8: Generics

8.1. (W) Explain in detail why Java's Generics not support the use of primitive types as the parameterised type? Why can you not instantiate objects of the template type in generics (i.e. why is `new T()` forbidden?)

8.2. (W) Rewrite your `OOPList` interface and `OOPLinkedList` class to support lists of types other than integers using Generics. e.g. `OOPLinkedList<Double>`.

8.3. (W) Explain the notion of wildcards in Java Generics.

8.4. (*)The following code will not compile. It is intended to take a list of `Doubles` and add them to a list of `Numbers`

.

```
import java.util.List;
import java.util.ArrayList;

public class Example {
    public static void main(String[] args) {
        List<Double> doubles = List.of(1.1, 2.2, 3.3); List<Number>
            numbers = new ArrayList<>();

        // This call fails to compile
        addItems(doubles, numbers);
        System.out.println(numbers);
    }
```

```
    // This method signature is too restrictive
    public static <T> void addItems(List<T> src, List<T> dest) {
        for (T t : src) {
            dest.add(t);
        }
    }
}
```

1. Explain *why* the call `addItems(doubles, numbers)` fails to compile.
2. Fix the signature of the `addItems` method using bounded wildcards so that the `main` method compiles and runs correctly.

8.5. (*) Java provides the `List` interface and an abstract class that implements much of it called `AbstractList`. The intention is that you can extend `AbstractList` and just fill in a few implementation details to have a Collections-compatible structure. Write a new class `CollectionArrayList` that implements a mutable Collections-compatible Generics array-based list using this technique. Comment on any difficulties you encounter.

Section 9: Coupling, Errors and Exceptions

9.1. (W) The following code captures errors using return values. Rewrite it to use exceptions.

```
public class RetValTest {
    public static String sEmail = "";

    public static int extractCamEmail (String sentence) {
        if (sentence == null || sentence.length() == 0)
            return -1; // Error sentence empty

        String tokens[] = sentence.split(" "); // split into tokens
        for (int i = 0; i < tokens.length; i++) {
            if (tokens[i].endsWith("@cam.ac.uk")) {
                sEmail = tokens[i];
                return 0; // success
            }
        }
        return 2; // Error no cam email found
    }

    public static void main(String[] args) {
        int ret = RetValTest.extractCamEmail("My email is rkh23@cam.ac.
            uk");
        if (ret == 0) System.out.println("Success: " + RetValTest.sEmail
            );
        else if (ret == -1) System.out.println("Supplied string empty");
        else System.out.println("No @cam address in supplied string");
    }
}
```

9.2. (W) You are writing a method for a banking application `void withdraw(double amount)` that withdraws money from an account. If the `amount` is larger than the account's balance, the withdrawal should fail.

(a) Should you create an `InsufficientFundsException` that is **checked** (i.e., `extends Exception`) or **unchecked** (i.e., `extends RuntimeException`)?

(b) Justify your choice, referring to the lecture notes on how clients are expected to handle each type.

9.3. Write a Java function that computes the square root of a double number using the Newton-Raphson method. Your function should make appropriate use of exceptions and assertions.

9.4. Comment on the following implementation of `pow`, which computes the power of a number:

```
public class Answer extends Exception {
    private int mAns;
    public Answer (int a) { mAns = a; }
    public int getAns() { return mAns; }
}

public class ExceptionTest {
    private void powaux(int x, int v, int n) throws Answer {
        if (n == 0) throw new Answer(v);
        else powaux(x, v * x, n - 1);
    }

    public int pow(int x, int n) {
        try { powaux(x, 1, n); }
        catch (Answer a) { return a.getAns(); }
        return 0;
    }
}
```

9.5. (*) In Java `try...finally` blocks can be applied to any code no `catch` is needed. The code in the `finally` block is guaranteed to run after that in the `try` block. Suggest how you could make use of this to emulate the behaviour of a destructor (which is called immediately when indicate we are finished with the object, not at some indeterminate time later).

9.6. By experimentation or otherwise, work out what happens when the following method is executed.

```
public static int x() {
    try { return 6; }
    finally { ... }
}
```

Section 10: Design Patterns

10.1. (W) Explain the difference between the State pattern and the Strategy pattern.

10.2. In lectures the examples for the State pattern used academic rank.

(a) Explain the problems with the first solution of using direct inheritance of `Lecturer` and `Professor` from `Academic` rather than the State pattern.

(b) Explain why a call to `getRank()` could not simply return the `mRank` object.

(c) Alternatives are to return a separate rank indicator (e.g. an int); to return a copy of the rank object; or return an immutable version of the object. Using an appropriate design pattern, which you should identify, show how to achieve the immutable version, and identify any problems that might occur

10.3. A drawing program has an abstract `Shape` class. Each `Shape` object supports a `draw()` method that draws the relevant shape on the screen (as per the example in lectures). There are a series of concrete subclasses of `Shape`, including `Circle` and `Rectangle`. The drawing program keeps a list of all shapes in a `List<Shape>` object.

(a) Should `draw()` be an abstract method?

(b) Write Java code for the method called by the main application to draw all the shapes on each screen refresh.

(c) Show how to use the Composite pattern to allow sets of shapes to be grouped together and treated as a single entity.

(d) Which design pattern would you use if you wanted to extend the program to draw frames around some of the shapes? Show how this would work.

10.4. For some applications the Decorator pattern has the weakness that decorated objects can be decorated. For example, wrapped shop items should not be wrapped again! Show how to prevent a wrapped item from being wrapped. It is easy to solve at runtime using a flag try to solve it at compile time (so the compiler won't allow it rather than the JVM throwing an exception).

Section 11: Lambdas, Method References and Streams

11.1. Refactor the following code to use method references instead of explicit lambdas.

```
public class StringUtils {
    public static void main(String[] args) {
        List<String> words = List.of("Apple", "Banana",              "
            Cherry");

        // 1. Print each word
        words.forEach(s -> System.out.println(s));

        // 2. Get list of lengths
        List<Integer> lengths = words.stream()
            .map(s -> s.length())
            .toList();
```

```
        // 3. Create new String objects
        List<String> copies = words.stream()
            .map(s -> new String(s))
            .toList();
    }
}
```

11.2. Using Streams.You have a `List<Book>`:

```
public record Book(String title, String author, int pages) {}

List<Book> library = List.of(
    new Book("Moby Dick", "Herman Melville", 720),
    new Book("1984", "George Orwell", 328),
    new Book("Ulysses", "James Joyce", 730),
    new Book("War and Peace", "Leo Tolstoy", 1225)
    );
```

Write a *single* Java stream pipeline that returns a `List<String>` containing only the **titles** of the books that have **more than 500 pages**, **sorted alphabetically** by title.