CS 61B
Spring 2026

Lists, Maps, Arrays, References

Discussion 02: January 26, 2026

1 The Mystery of the Walrus

(a) Consider the code below. Next to each blank, write down the expected output. Alternatively, if it's impossible to predict the output, write "unknown".

Implementations for **obliterate**, **IntSquasher**, **shamble**, and **agglutinate** are unknown.

```
public class Walrus {
    public static void main(String[] args) {
        int x = 10;
        obliterate(x);

        System.out.println(x);                 10

        int y = 20;
        IntSquasher isq = new IntSquasher(y);

        System.out.println(y);                 20


        int[] z = new int[]{1, 2, 3};
        shamble(z[0]);

        System.out.println(z[0]);              1

        agglutinate(z);

        System.out.println(z[1]);              unknown

    }
}
```

(b) Consider the class `MyInteger` below.

```
public class MyInteger {
    public int val;
    public MyInteger(int val) {
        this.val = val;
    }

    @Override
    public String toString() {
        return String.valueOf(this.val);
    }
}
```

If `z` was instantiated as

```
MyInteger[] z = new MyInteger[] {MyInteger(1), MyInteger(2), MyInteger(3)};
```

Would any of your answers change? If so, which ones, and why? If not, why not?

`int` is not a reference type. **`MyInteger`** is! When you pass **`x[0]`** into **`shamble`**, it could access **`z[0].val`** and modify it, changing the value of **`z[0]`** in **`main`**'s scope. As such, **`System.out.println(z[0])`** now prints an unknown value!

(c) Implementations for **invertify**, **scrub**, and **feed** are unknown.

```
public class WalrusReview {
    public int v;
    public static String name;

    public WalrusReview(int v) {
        this.v = v;
        name = "Scott";
        v = -10;
    }

    public static void main(String[] args) {
        int z = 10;
        WalrusReview wr = new WalrusReview(z);

        System.out.println(z);                      10

        System.out.println(wr.v);                   10

        invertify(wr.v);

        System.out.println(wr.v);                   10

        scrub(WalrusReview.name);

        System.out.println(WalrusReview.name);   Scott

        z = 10;
        wr = new WalrusReview(z);
        feed(wr);

        System.out.println(z);                      10

        System.out.println(wr.v);                   unknown

        System.out.println(WalrusReview.name);   unknown
    }
}
```

2 Ranking Players

Fill in **rankedAbove**, which takes in a list of **Player**s and returns a map from each **Player** to their rank. The player with the highest score has rank 1, the player with the next-highest score has rank 2, and so on. Assume no two players have the same score. For example, if we have a list of players with **score**s of 500, 800, 1200, and 100, then these players would have ranks 3, 2, 1, and 4, respectively, and **rankedAbove** would return the following **Map**:

```
{  Player with score 500  : 3, Player with score 800  : 2,
   Player with score 1200 : 1, Player with score 100  : 4  }
```

Syntax hints (you may not need all of these):

- A `Set` has the operations `add` and `contains`. You can instantiate one using `new HashSet`.
- A `map` has the operations `put`, `containsKey`, and `get`. You can instantiate one using `new HashMap`.
- A `list` has the operations `get` and `set`. You can instantiate one using `new ArrayList`.
- You can iterate over a `List<Integer> c` or a `Set<Integer> c` using `for int x : c`.
- `someMap.keySet()` will return the `Set` of all keys in the map `someMap`.

```
public class Player {
    public double score;
    public static Map<Player, Integer> rankedAbove(List<Player> players) {

        Map<Player, Integer> results = new HashMap<>(); ;

        for (Player p1 : players ) {

            results.put(p1, 1);

            for (Player <| p2 : players) {

            if (p2.score > p1.score) {

            results.put(p1, results.get(p1) + 1);

            }

            }

            UNUSED

        }
        return results;
    }
}
```

3 Static Books

Suppose we have the following **Book** and **Library** classes.

```java
class Book {
    public String title;
    public Library library;
    public static Book last = null;

    public Book(String name) {
        title = name;
        last = this;
        library = null;
    }

    public static String lastBookTitle()
{
        return last.title;
    }
    public String getTitle() {
        return title;
    }
}
```

```java
class Library {
    public Book[] books;
    public int index;
    public static int totalBooks = 0;

    public Library(int size) {
        books = new Book[size];
        index = 0;
    }

    public void addBook(Book book) {
        books[index] = book;
        index++;
        totalBooks++;
        book.library = this;
    }
}
```

(a) For each modification below, determine whether the code of the `Library` and `Book` classes will compile or error if we **only** made that modification, i.e. treat each modification independently.

1. Change the `totalBooks` variable to **non** `static`

   Compile

2. Change the `lastBookTitle` method to **non** `static`

   Compile

3. Change the `addBook` method to `static`

   Error, cannot access instance variable books in a static method.

4. Change the `last` variable to **non** `static`

   Error, cannot access instance variable last in a static method.

5. Change the `library` variable to `static`

   Compile

(b) Using the original **Book** and **Library** classes (i.e., without the modifications from part a), write the output of the **main** method below. If a line errors, put the precise reason it errors and continue execution.

```
public class Main {
    public static void main(String[] args) {

        System.out.println(Library.totalBooks);                 0

        System.out.println(Book.lastBookTitle());               RE, NullPointerException

        System.out.println(Book.getTitle());                    CE, does not compile

        Book goneGirl = new Book("Gone Girl");
        Book fightClub = new Book("Fight Club");

        System.out.println(goneGirl.title);                     Gone Girl

        System.out.println(Book.lastBookTitle());               Fight Club

        System.out.println(fightClub.lastBookTitle());          Fight Club

        System.out.println(goneGirl.last.title);                Fight Club

        Library libraryA = new Library(1);
        Library libraryB = new Library(2);
        libraryA.addBook(goneGirl);

        System.out.println(libraryA.index);                     1

        System.out.println(libraryA.totalBooks);                1

        libraryA.totalBooks = 0;
        libraryB.addBook(fightClub);
        libraryB.addBook(goneGirl);

        System.out.println(libraryB.index);                     2

        System.out.println(Library.totalBooks);                 2

        System.out.println(goneGirl.library.books[0].title); Fight Club
    }
}
```

Click here for visualizer link

4 Country Club

Avik wants to keep track of the students in UC Berkeley's clubs. Each club is represented by the `Club` class below, which maps every student in that club to their home country.

```
public class Club {
    public Map<Student, Country> countryMap;
    ...
}

public class Student { ... }
public class Country { ... }
```

On the next page, implement `countByCountry`, which takes in a list of `Club`s, and returns a map from each `Country` to the number of unique students from that country. The map should only contain countries that appear in the `countryMap`s.

If a `Student` is in multiple clubs, then each of those clubs will map that student to the same `Country`. Make sure to avoid counting the same `Student` twice if they are in multiple clubs.

You may assume that there is at least one club, and each club has at least one student.

Here is an example with 2 clubs and 3 total students:

Club	**Country Map**
Chess Club	`{` Aditya`:` Scotland, Natalia`:` Brazil, Rushil`:` Scotland `}`
Climbing Club	`{` Natalia`:` Brazil `}`

`countByCountry` should return the following map: `{` Brazil`:` 1, Scotland`:` 2 `}`.

Syntax hints (you may not need all of these):

- A `Set` has the operations `add` and `contains`. You can instantiate one using `new HashSet`.
- A `map` has the operations `put`, `containsKey`, and `get`. You can instantiate one using `new HashMap`.
- A `list` has the operations `get` and `set`. You can instantiate one using `new ArrayList`.
- You can iterate over a `List<Integer> c` or a `Set<Integer> c` using `for int x : c`.
- `someMap.keySet()` will return the `Set` of all keys in the map `someMap`.

```
public static Map<Country, Integer> countByCountry(List<Club> allClubs) {

    Map<Country, Integer> counts = new HashMap<>();

    Set<Student> uniqueStudents = new HashSet<>();

    for (Club club : allClubs) {

        for (Student s : club.countryMap.keySet()) {

            Country c = club.countryMap.get(s);

            if (!uniqueStudents.contains(s)) {

                if (!counts.containsKey(c)) {

                    counts.put(c, 1);

                } else {

                    counts.put(c, counts.get(c) + 1);

                }

                uniqueStudents.add(s);

             }

            UNUSED

            UNUSED
        }
    }

    return counts;
}
```