Name: _____ Andrew Id: _____

15-121 Assessment 5

60 minutes. No calculators, no notes, no books, and no computers.
Place answers on the answer sheet.

1. **Short Answer**

Answer the following in 1 sentence; longer answers will be marked incorrect.

(a) (1 point) How does increasing the number of buckets in a hash table affect performance, in terms of lookup speed and memory usage?

(b) (1 point) What is the difference between HashSet and TreeSet?

A. HashSet is faster but unordered; TreeSet is ordered but slower.

B. HashSet allows duplicates; TreeSet does not.

C. HashSet uses equals(); TreeSet uses compareTo().

D. Both A and C are correct.

(c) (2 points) Why does this code print false, even though the two Students contain identical values? What is the proper fix for this bug?

```
1  import java.util.HashSet;
2
3  class Student {
4      String name;
5      int id;
6
7      public Student(String name, int id) {
8          this.name = name;
9          this.id = id;
10     }
11
12     @Override
13     public boolean equals(Object o) {
14         if (!(o instanceof Student)) return false;
15         Student other = (Student) o;
16         return this.id == other.id;
17     }
18
19     public static void main(String[] args) {
20         HashSet<Student> students = new HashSet<>();
21         Student s1 = new Student("Alice", 101);
22         Student s2 = new Student("Alice", 101);
23
24         students.add(s1);
25         System.out.println("Contains s2? " + students.contains(s2)); // false!
26     }
27 }
```

2. (16 points) **Social Network** In this problem, you will build a tiny social network that tracks follower relationships between users. Unlike mutual friendships, the relationships here are directed — if user A follows user B, it does not necessarily mean that user B follows user A.

   For example:
   - If Alice follows Bob, Alice is a **follower of** Bob.
   - Bob is **followed by** Alice.

```
public class SocialNetwork {
    // Instance variables and constructor

    /**
     * Adds a one-way "follows" relationship from follower → followed.
     * Must run in O(1)
     *
     * @param follower the user who follows another
     * @param followed the user being followed
     * @return true if the relationship was newly added, false if it already existed
     */
    public boolean addFollow(String follower, String followed) {
        // You will write this code
    }

    /**
     * Returns all users that a given user follows.
     * Must run in O(m), where m is the number of users they follow.
     *
     * @param user the user whose followed list is requested
     * @return an array of usernames this user follows
     */
    public String[] getFollowing(String user) {
        // You will write this code
    }

    /**
     * Checks if one user follows another.
     * Must run in O(1)
     *
     * @param follower the potential follower
     * @param followed the potential followed user
     * @return true if follower follows followed, false otherwise
     */
    public boolean follows(String follower, String followed) {
        // You will write this code
    }

    /**
     * Finds common accounts that both users follow.
     * Must run in O(min(f1, f2)).
     *
     * @param user1 the first user
     * @param user2 the second user
     * @return an array of users followed by both user1 and user2
     */
    public String[] getCommonFollowing(String user1, String user2) {
        // You will write this code
    }
```

```
    public static void main(String[] args) {
        SocialNetwork sn = new SocialNetwork();

        sn.addFollow("alice", "bob");      // Alice follows Bob
        sn.addFollow("alice", "charlie");  // Alice follows Charlie
        sn.addFollow("bob", "charlie");    // Bob follows Charlie
        sn.addFollow("bob", "dave");       // Bob follows Dave

        System.out.println(Arrays.toString(sn.getFollowing("alice"))); // [bob, charlie]
        System.out.println(sn.follows("alice", "bob"));     // true
        System.out.println(sn.follows("alice", "dave"));    // false
        System.out.println(sn.getCommonFollowing("alice", "bob")); // [charlie]
    }
}
```

The main method, when executed, outputs:

```
[bob, charlie]
true
false
[charlie]
```

3. Free Response: `Movie` class.

   (a) (24 points) Write a complete Movie class that validates fields, supports hashing/equality, formatted output, and natural ordering.

   - A Movie has title (must be a valid string), year (must be between 1888 and 2025, inclusive), and a rating (must be between 0 and 10, inclusive and only whole numbers allowed). An `IllegalArgumentException` must be thrown on invalid input.
   - A Movie has a constructor that set all of its attributes.
   - The only way for code outside of `Movie` to read and modify the attributes of a `Movie` must be using appropriately named getters and setters. (You don't need to provide the getters, but must provide the setters.)
   - When printing a Movie, it should be formatted as follows:
     `"Title" (Year) [Rating/10]`
   - A `Movie` needs to be able to be properly used in `HashSet`s and `HashMap`s.
   - An array of `Movie`s must be sortable using `Arrays.sort(someArrayOfMovies)`. The natural order is based on the year (oldest first), then by rating (highest first), then by title (alphabetically).

   Write the `Movie` class to the above specifications. Pay careful attention to which classes you may need to extend or implement as well as which methods you need to write or override.

   Restriction: You may not use the the method `Objects.hash`.

   (b) (6 points) Without modifying your Movie class, write the code needed to sort an array movieArray by rating (high to low), breaking ties by title (A–Z). You do not need to write a complete example with a main function and a testcase, instead just focus on what is needed in order to use the `Arrays.sort` method to sort an array of `Movie`s named `movieArray`.