Midterm CPE 203 – Fall 2018

Answer Key

1. `Object.equals()` and `Object.hashCode()`

Complete the following class:

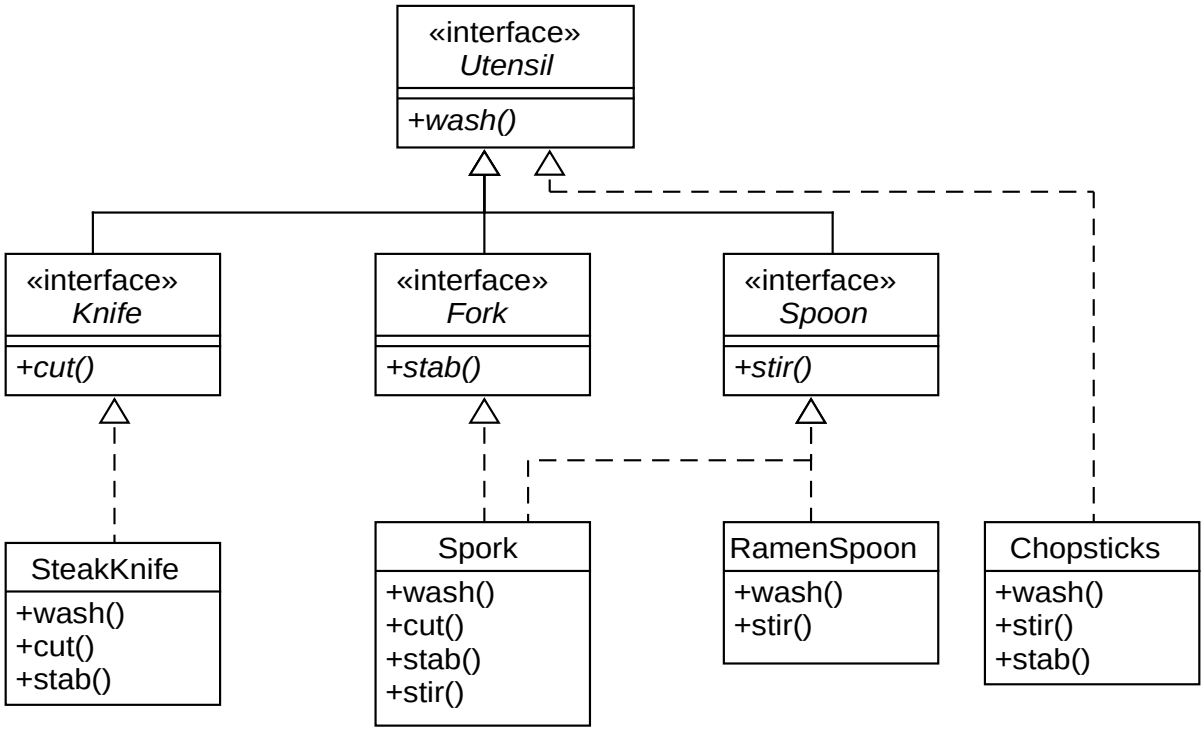
```
public final class PostItMessage {
    public final String message;     // guaranteed to not be null
    public final int x;
    public final int y;

    public PostItMessage(String message, int x, int y) {
        // implementation not shown, but it is correct and reasonable
    }
    /**
     * Returns true if other is a PostItMessage instance with equivalent
     * values for the three fields
     */
    public boolean equals(Object other) {
        if (other instanceof PostItMessage) {
            PostItMessage op = (PostItMessage) other;
            return message.equals(op.message) && x == op.x && y == op.y;
        } else {
            return false;
        }
    }

    /**
     * Returns a value consistent with the definition of equals()
     * such that PostItMessage instances can be used as the key in
     * a hash table.
     */
    public int hashCode() {
        return Objects.hash(message, x, y);
    }
```

A - equals checks null
B - equals checks message.equals()
C - equals check message somehow, even if incorrectly
D - equals checks x
E - equals checks y
F - equals checks instanceof
G - equals does downcast
H - hashCode uses message correctly
I - hashCode uses x correctly
J - hashCode uses y correctly
K - hashCode combines the three values appropriately
L - hashCode combines at least two of the values appropriately

V - small error: you got message.hashCode() syntax wrong
W - small error: other error getting a hash value
X - small error: you tried to compare two strings using ==
Y - you tried to say x.hashCode(), hash(x) or similar
Z - you used getClass()/reflection in equals

2. Classes, Interfaces and Types

Consider the following type hierarchy:



Inside a method, the following references have been declared an initialized to something that is not null and that is not shown. Other code that is not shown might include additional classes or interfaces not show in this diagram.

```
Utensil utensil = ... ;
Knife knife =  ... ;
Fork fork = ... ;
Spoon spoon = ... ;
SteakKnife steakKnife = ... ;
Spork spork = ... ;
RamenSpoon ramenSpoon = ... ;
Chopsticks chopsticks = ... ;
Car car = ... ;
```

2. Continued

For each code fragment below, write **A** if the fragment will always compile and run, **M** if the fragment will compile but might fail at runtime, and **F** if the fragment will fail to compile. You may assume that the declared methods will not fail when called. Each code fragment is independent; an assignment statement in one does not affect the following fragments.

Code Fragment	A/M/F
`utensil.wash();`	**A**
`knife.wash();`	**A**
`chopsticks.wash();`	**A**
`((Fork) spork).stab();`	**A**
`((Fork) spork).stir();`	**F**
`((Spork) utensil).stab();`	**M**
`((Spoon) chopsticks).stir();`	**M**
`utensil = spork;`	**A**
`utensil = fork;`	**A**
`fork = spork;`	**A**
`knife = spork;`	**F**
`spork = utensil`	**F**
`fork = utensil`	**F**
`((Utensil) chopsticks).stab()`	**F**
`((Chopsticks) utensil).stab()`	**M**

3. Creating Classes and Methods

Assume the following partial class/interface definitions for `League`, `Team` and `Fan`. Read all the code before you begin

a) Fill in all the methods marked **TODO** .

```
public class League {
    private final String name;
    private final Map<String, Team> teamsByName;
    private final List<Fan> fans;

   /* TODO
    * Finish implementing the constructor
    */
    public League(String name) {
        teamsByName = new HashMap<String, Team>()

A:          this.name = name;
B:          fans = new ArrayList<Fan>()

    }

    /* TODO
     * Implement this
     */
    public void addTeam(Team t){

C:      teamsByName.put(t.getName(), t);

    }

    /* TODO
     * Implement this
     */
    public Team getTeam(String name){

D:      return teamsByName.get(name);


    }


    /* TODO
     * Implement this
     */
    public void addFan(Fan fan){

E:      fans.add(fan);


    }
```

```
        /* TODO
         * This method of League returns a string containing the league name
         * and total number of teams, in the following format:
         *     "League Name (237 teams)"
         */
        public String toString() {

A:          return name + "(" + teamsByName.size() + " teams)";



        }
    }
    /* TODO
     * Implement all methods of Team
     */
    public class Team {
        private final String name;
        private int totalPlayerSalaries;  // in dollars

        public Team(Sring name, int totalPlayerSalaries) {

B:          this.name = name;
C:          this.totalPlayerSalaries = totalPlayerSalaries;

        }

        public String getName() {

D:           return name;


        }

        public int getTotalPlayerSalaries() {

E:           return totalPlayerSalaries;

        }

        public void addToPlayerSalaries(final int amount) {

F:          totalPlayerSalaries += amount;


        }

        public void subtractFromPlayerSalaries(final int amount) {

G:           totalPlayerSalaries -= amount;

        }
    }
```

```
public class Fan {

    private final Team team;
    // other fields not shown

    public Fan(Team team, int id) {
        // Implementation not shown.  It initializes other data within
        // Fan by looking up the id in a fixed data structure that is
        // not shown.
    }

    public String getName() { /* Implementation not shown */ }

    public Team getTeam()   {  return team;  }

    /**
     * Returns the number of years this fan has been with their
     * team.
     */
    public int getYearsWithTeam() {
        // Implementation not shown
    }

    public String toString() {
        // Implementation not shown
    }

    // Other details of the Fan class not shown
}
```

1. **(3 continued)** Answer the following questions using the above classes.

b. Fill in the method below so that it creates and returns a `League` object with two teams and three fans, one for the first team and two for the second. Use any names and other values you like. This method is part of a class that is not `League`, `Team` or `Fan`.

```
public League createTestLeague() {

A:      League league = new League("Foo Frobbers");
B:      Team t1 = new Team("Team 1", 27);
        Team t2 = new Team("Team 2", 42);
C:      Fan f1 = new Fan(t1, 101);
        Fan f2 = new Fan(t1, 102);
        Fan f3 = new Fan(t2, 103);
D:      league.addTeam(t1);
        league.addTeam(t2);
E:      league.addFan(f1);
        league.addFan(f2);
        league.addFan(f3);



F:       return league;









}
```

c. Add a method to the `League` class to increase the total player salary number for each team by a fixed amount. Fill in the implementation of the following method, which is now part of `League`.

```
public void increaseTeamSalary(final int fee)
{


    for (Team team : teamsByName.values()) {
        team.addToPlayerSalaries(fee);
    }



G:  uses teamsByName.values(), or other valid way of getting teams
H:  iterates
I:  calls team.addToPlayerSalaries(fee)









}
```

4. Sorting

a. Printing the Fans

Implement the following method, which is contained in a new class called `Printer` that is not shown. You may not add any new definitions outside the method. This method should print the fans using `Fan.toString()`, one per line, sorted by team, and within each team, sorted by fan name.

```
public void printFans(ArrayList<Fan> fans) {

    Comparator<Fan> c = (Fan left, Fan right) -> {
        int result = left.getTeam().getName().compareTo(
                              right.getTeam().getName());
        if (result == 0) {
            result = left.getName().compareTo(right.getName());
        }
        return result;
    };

    fans.sort(c);

    for (Fan f : fans) {
        System.out.println(f);
    }
}
```

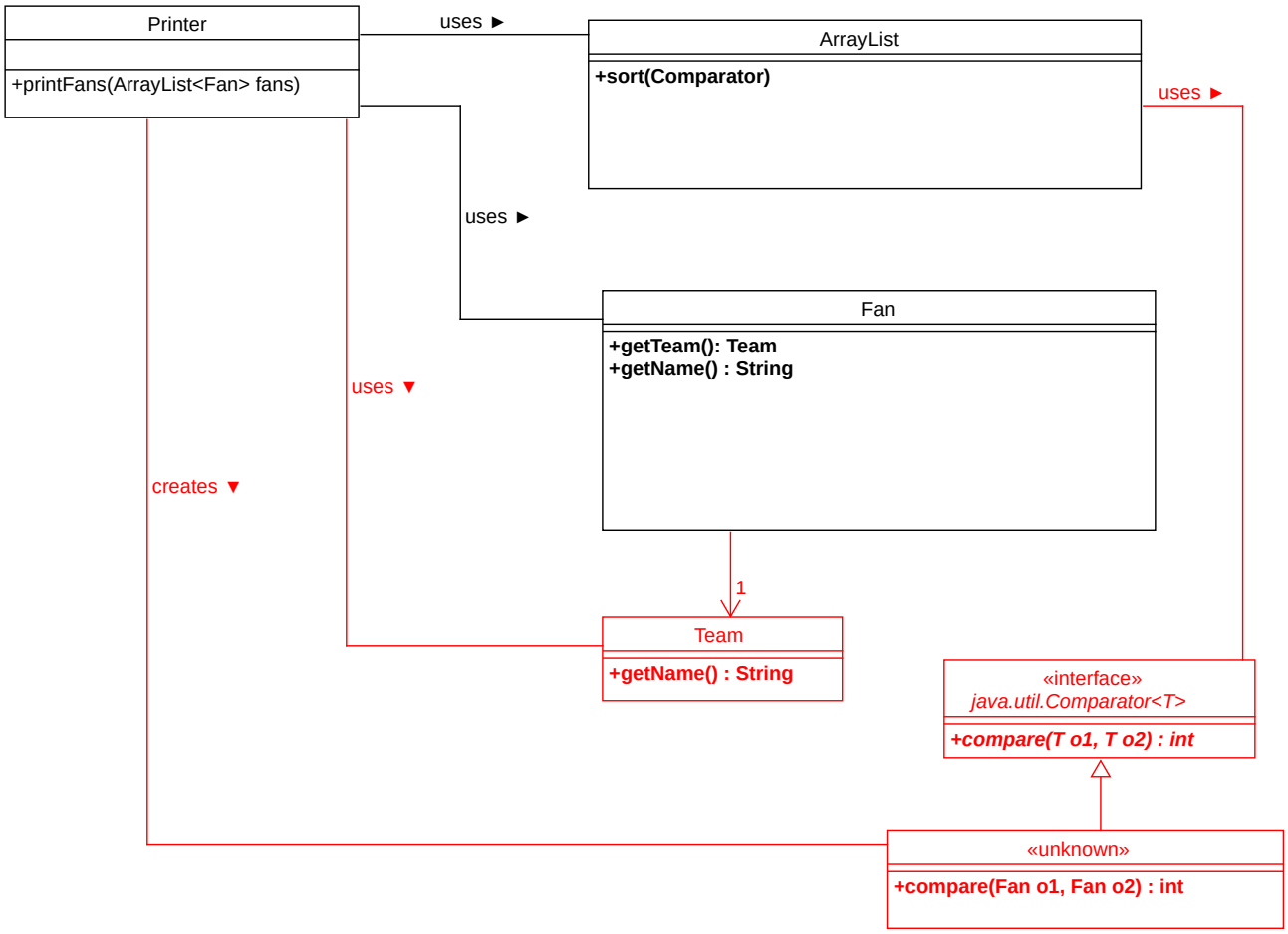
A: Creates comparator
B: Comparator takes two arguments
C: Comparator compares team name
D: Uses String.compareTo() or compareToIgnoreCase()
E: Compares fan name if team name matches
F: comparator returns result
G: Uses ArrayList.sort() or equivalent
H: Iterates over fans
I: System.out.println(f) or System.out.println(f.toString())

Note: Some students didn't use comparators, even though we just finished covering them in lab. These midterms tried to do a brute-force sort, but in all cases the sorting code was buggy enough so that the partial credit given was appropriate, despite the mismatch in the rubric. Remember also, "your code will be graded on functionality, design and efficiency."

One fairly common error was to sort by team name, and then sort by fan name ignoring the team name, thereby obliterating the results of the first sort.

b. UML

Complete the following UML diagram showing `Printer` and any other classes or interfaces that are relevant to the your code for `Printer.printFans()`. Classes in `java.lang`, like String, need not be shown. Anything with an unknown name can be shown with the name "<<Unknown>>".



A: ArrayList.sort()
B: Fan.getTeam()
C: Fan.getName()
D: Team class
E: Team.getName()
F: Comparator<T>
G: Class implementing Comparator

Note that a key extractor is a class that implements the Function<T, U> interface. It's not possible to make a key extractor for Fan.getTeam().getName(), but if it were somehow, then that key extractor would be an anonymous subtype of Function. A key extractor for Fan::getName() is a class that implements Function<T, U>.

Class String

- java.lang.Object
  - java.lang.String
- All Implemented Interfaces:
  Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

Method Summary

All Methods Static Methods Instance Methods Concrete Methods Deprecated Methods

Modifier and Type	Method and Description
char	charAt(int index) Returns the char value at the specified index.
int	compareTo(String anotherString) Compares two strings lexicographically.
int	compareToIgnoreCase(String str) Compares two strings lexicographically, ignoring case differences.
String	concat(String str) Concatenates the specified string to the end of this string.
boolean	equals(Object anObject) Compares this string to the specified object.
boolean	equalsIgnoreCase(String anotherString) Compares this String to another String, ignoring case considerations.
int	hashCode() Returns a hash code for this string.