CSC 203 Common Final Exam

Fall 2024

December 11, 2024

Your name: _____

Rules:

- This exam is closed notes and book.
- You may not use any electronic devices during the exam.

Suggestions:

- Read the questions carefully. Be sure to answer all parts of a multi-part question.
- Provide definite, clear, complete, and (where possible) concise answers for all questions.
- Use any blank space available (including the backs of pages) to write your answers.
- Identify your answers *clearly*.
- State any assumptions made in solving a problem. **If you think there is a misprint, ask.**
- Problems are not necessarily ordered by difficulty.
- Be sure that you have all the pages. There are 15 pages including this one.

For grader use only.

#	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Total
Score											
Out of	8	3	7	4	5	2	10	10	8	8	65

All the best!

Question 1. (8 points) Instance and static members.

Answer the questions below, given the following class.

```
public class Course {
    public static final int DEFAULT_ENROLLMENT_CAP = 35;

    private String deptCode;
    private int number;
    private String title;
    private Set<Course> preRequisites;

    public Course(String deptCode, int number, String title) {
        this.deptCode = deptCode;
        this.number = number;
        this.title = title;
        this.preRequisites = Set.of(); // empty set
    }

    public Course(String deptCode, int number, String title, Set<Course> preReqs) {
        this(deptCode, number, title);
        this.preRequisites = preReqs;
    }

    @Override
    public String toString() {
        return this.deptCode + " " + this.number + " " + this.title;
    }

    // Assume that there are getter methods for the instance variables.
}
```

For each of the following, circle the correct answer and explain your reasoning. Assume that these lines are being executed in another file that's in the same folder as `Course.java`.

a) `System.out.println(Course.DEFAULT_ENROLLMENT_CAP);`

- This code (**will compile** / **will not compile** / **will crash**)
- Reasoning:

  Because the variable is static

b) `Course course = new Course();`
`System.out.println(course.toString())`

- This code (**will compile** / **will not compile** / **will crash**)
- Reasoning:

There is no constructor with no parameters
Missing semicolon from the second instruction

c) `Course csc101 = new Course("CSC", 101, "Fundamentals of Computer Science");`

- This code (**will compile** / **will not compile** / **will crash**)
- Reasoning:

It uses one of the constructors that I have in the class

d) `System.out.println(Course.toString());`

- This code (**will compile** / **will not compile** / **will crash**)
- Reasoning:

toString() is an instance method, so we need an object to call it not the class

Question 2. (3 points) Suppose you also have a `Student` class that contains the following instance variables (and appropriate getters):

- `String name`
- `Set<Course> coursesTaken`

Consider the method below.

Given a `Student` and a `Course`, it decides whether the `Student` is eligible to take the `Course`. A student is considered eligible to take the course if they have taken all the pre-requisites, or if the course has no pre-requisites.

```
public static boolean courseCanBeTaken(Course course, Student student) {
    return course.getPrerequisites().stream()
        .allMatch(preReq -> student.getCoursesTaken().contains(preReq));
}
```

Re-write the `courseCanBeTaken` method to be an instance method in either the `Student` or `Course` class (pick one).

In the student class:

```
public boolean courseCanBeTaken(Course course) {
    return course.getPrerequisites().stream()
        .allMatch(preReq -> this.getCoursesTaken().contains(preReq));
}
```

In the course class:

```
public boolean courseCanBeTaken(Student student) {
    return this.getPrerequisites().stream()
        .allMatch(preReq -> student.getCoursesTaken().contains(preReq));
}
```

Question 3. (3 + 3 + 1 = 7 points) `equals` and `hashCode`.

```
public class County {
    private String name;
    private double medianHouseholdIncome;

    public County(String name, double medianHouseholdIncome) {
        this.name = name;
        this.medianHouseholdIncome = medianHouseholdIncome;
    }

    // Assume there are getters for the instance variables above
}
```

a) Given the class above, override the `equals` method to compare two `County` objects based on all instance variables in the `County` class. You may use `Objects` methods in your solution.

```
public Boolean equals( Object obj ) {
  if ( obj == null ) { return false; }

  if ( !this.getClass().equals( obj.getClass() ) ) { return false }

  Country tmpCountry = ( Country ) obj;

  return this.getName().equals(tmpCountry.getName() && this.getMedianHouseholdIncome == tmpCountry.getMedianHouseholdIncome;
```

b) Override the `hashCode` method for the `County` class. You may NOT use `Objects.hash` in your solution, but you may use calls to other objects' `hashCode` methods.

```
public int hashCode(){
  int result = 1;
  result = result + 37 * this.getMedianHouseholdIncome;
  result = result + 37 * this.getName().hashCode();
  return result;
```

c) If two objects are equal, they should also produce the same hash code. Circle one:

true / **false**

Question 4. (4 points)

Consider the following code snippet, which uses the `County` class from the previous question. For the boolean expressions that follow, indicate their values by circling `true` or `false`. Assume that your implementation of `equals` is correct for this question.

```
County slo = new County("San Luis Obispo", 82514);
County sanLuisObispo = slo;

County monterey = new County("Monterey", 82013);
County mont = new County("Monterey", 82013);
```

a) `slo.equals(monterey)` (true / false)

b) `monterey.equals(mont)` (true / false)

c) `monterey == mont` (true / false)

d) `slo == sanLuisObispo` (true / false)

Question 5. (3 + 2 = 5 points) Inheritance.

Consider the `Book` and `PrintedBook` classes and then answer the questions that follow.

```
public class Book {
    private final String title;
    private String author;

    public Book(String title,
        String author) {
        this.title = title;
        this.author = author;
    }

    @Override
    public String toString() {
        return this.title + " by " +
            this.author + ".";
    }
}
```

```
public class PrintedBook extends Book {
    private final int pageCount;

    public PrintedBook(String title,
        String author,
        int pageCount) {
        super(title, author);
        this.pageCount = pageCount;
    }

    public int getNumberOfPages() {
        return this.pageCount;
    }
}
```

a) Write a `toString` method for the `PrintedBook` class. The required format of the output is as follows. For the book below...

```
new PrintedBook("A Tale of Two Cities", "Charles Dickens", 489);
```

...the expected string representation is `"A Tale of Two Cities by Charles Dickens. 489 pages."`

For full credit, do this *without* assuming direct access to the `title` and `author` instance variables in the `Book` class (through getter methods or otherwise).

```
public String toString(){
  return this.getTitle() + " by " + this.getAuthor() + "." + " " + this.getNumberOfPages() + "
pages.";
```

b) Here is an `equals` method for the `PrintedBook` class. Notice that it calls `super.equals(other)`. That is, it calls `Book`'s `equals` method.

```
@Override
public boolean equals(Object other) {
    return super.equals(other) && this.pageCount == ((PrintedBook) other).pageCount;
}
```

Consider the `equals` method for `Book` below. Fill in the blank so that this method can be used as see above in `PrintedBook` as well as any other `Book` subtypes we might create.

```
@Override
public boolean equals(Object other) {
    if (other == null) {
        return false;
    }

    if (other == this) {
        return true;
    }

    if (other instanceof Book) {
        Book book = (Book) other;
        return Objects.equals(this.title, book.title) &&
            Objects.equals(this.author, book.author);
    }

    return false;
}
```

Question 6. (2 points) Java memory model.

Consider the following `Point` class, a simple representation of a point in 2D space.

```
public class Point  {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // Assume getters and setters for x and y
}
```

Consider the code snippet below.

```
public class ExamQuestion {
    public static void main(String[] args) {
        Point p = new Point(1, 2);
        move(p);
        System.out.println(p.x + " " + p.y);
    }

    public static void move(Point p) {
        p = new Point(3, 4);
    }
}
```

What will be printed out after the `main` method above is executed? (Circle one)

(a) `"3 4"` (b) `"1 2"` (c) `"2 1"` (d) `"4 3"`

Question 7. (10 points) Class design.

The SLO Food Bank needs a software system to manage its operations. It can currently distribute food to different types of recipients: namely *households* or *organisations*.

Each recipient type has a name and email address. They must also have the ability to verify their eligibility and to request food from the Food Bank. The steps for verifying eligibility are different for the different recipient types.

Additionally, *households* need to keep track of their dietary restrictions, and *organisations* need to keep track of their maximum capacity (maximum number of people they can support).

Plan how you would represent these recipients in your code by drawing a diagram. You do not need to include any code. And you do not need to model the entire system, just the recipients. If you need to assume certain classes exist that you will use in your design, state those assumptions.

Your diagram should include:

- the classes, abstract classes, or interfaces you would create,
- relationships between them,
- and any members within them, like instance or static variables and abstract or implemented methods.

Be sure that your plan takes into account all necessary data and methods, supports loose coupling and abstraction, is extensible, and avoids code duplication.

Question 8. (3 + 3 + 4 = 10 points) Streams, lambdas, comparators.

The SLO Food Bank is also managing a large collection of `Donations`. Each `Donation` is defined as follows.

```
public class Donation {
    private final String donorName;
    private final String foodType;
    private final LocalDate expiryDate;
    private final int servings;

    public Donation(String foodType, String donorName, LocalDate expiryDate, int
    servings) {
        this.foodType = foodType;
        this.donorName = donorName;
        this.expiryDate = expiryDate;
        this.servings = servings;
    }

    /**
    * Gets the number of days from today's date to the expiry date.
    * Will be negative if the expiry date is in the past.
    */
    public int daysUntilExpiry() {
        LocalDate todaysDate = LocalDate.now();
        return todaysDate.until(this.expiryDate).getDays();
    }

    // Assume getter methods for donorName, foodType, and servings
}
```

Answer the following questions.

a) Write a method that takes a list of donations and determines how many of them are within 7 days of expiry. I.e., how many of them are *less than or equal to* 7 days of expiry. You must use streams and lambdas in your solution, not a loop.

Write your code in the method below.

```
public static int numExpiringWithinAWeek(List<Donation> donations) {

    return donations.stream()
                .filter( x -> x.daysUntilExpiry() <= 7 )
                .count();









}
```

b) Write a function that finds the total number of servings of non-expired food. Fill in the blanks.

```
public static int edibleServings(List<Donation> donations) {
    return donations.stream()

        .filter(x -> x.daysUntilExpiry() > 0          )

        .map(  x -> x.getNumServings()                )

        .reduce(   0, ( (result, current) -> result + current )  );
}
```

c) The Food Bank needs to prioritise which `Donations` must be distributed to recipients. We want to first distribute `Donations` that are closest to their expiry date, so as to avoid food wastage, then we want to distribute `Donations` that are most abundant (i.e., they have the most servings). For this question, assume that we have already filtered out expired food (so days until expiry will never be negative).

Fill in the blanks to define a `Comparator` that achieves the ordering above.

For full credit, use lambdas and the key extractor/method reference syntax.

```
Comparator<Donation> prioritiseDonations =

    Comparator.comparingInt(_____________________________________)

        .thenComparing((___________, ___________) -> _____________________________________);
```

Question 9. (8 points. A* search.)

Below is a grid of an imaginary world. Squares in gray are obstacles. White squares are clear to travel through. The `START` and `END` goals are labelled.

Complete **two** iterations of the A-star algorithm. By "iteration", we mean picking a node off the open list (priority queue), adding it to the closed list (list of visited nodes), and adding its valid neighbors to the open list.

1	2	3	4	5	6
7	8	9	10	11	12
13	**START (14)**	15	16	17	18
19	20	21	22	23	**GOAL (24)**
25	26	27	28	29	30
31	32	33	34	35	36

The distance heuristic used is Manhattan distance. Valid neighbours are *cardinal neighbours* (up, down, left, right) that have not yet been visited, that are within bounds of the world, and that are not obstacles.

In the following sub-questions,

- "G-value" refers to the node's known distance from the start
- "H-value" refers to the node's estimated distance to the end
- "Total" refers to the total (i.e., G-value + H-value)

If multiple equally-good options exist in the open list, choose the node whose node number is the *highest*.

Space and formatting for your answers are provided below.

As an example, below are the open list and closed list *before* the first iteration.

Node:	14				
G-value:	0				
H-value:	5				
Total:	5				

Closed list is empty before the first iteration, so the table is not shown.

a) After the **1st** iteration:

(Use only as many columns as you require.)

Open list:

Node:					
G-value:					
H-value:					
Total:					

Closed list:

Nodes:					

b) After the **2nd** iteration:

If a node is still in the open list, re-write it here and then add the additional nodes.

(Use only as many columns as you require.)

Open list:

Node:					
G-value:					
H-value:					
Total:					

Closed list:

Nodes:					

Question 10. (8 points) Exceptions.

Consider the following class definitions that contain try-catch blocks and exception-causing code.

```
public class Exceptional {
    public static void first(String text) {
        System.out.print("A");
        try {
            System.out.print("B");
            Item item = new Item(text);
            second(item.count);
        } catch (ArithmeticException e) {
            System.out.print("C");
        } catch (IllegalStateException e) {
            System.out.print("D");
        } finally {
            System.out.print("E");
        }
        System.out.print("F");
    }

    public static int second(int number) {
        System.out.print("1");
        try {
            System.out.print("2");
            if (number == 0) {
                throw new IllegalStateException();
            } else {
                int x = number % 2;
                int y = 2 / x;
                return y;
            }
        } catch (ArithmeticException e) {
            System.out.print("3");
        } finally {
            System.out.print("4");
        }
        System.out.print("5");
        return 0;
    }
}

public class Item {
    public int count;
    public Item(String item) {
        count = item.length();
    }
}
```

For each of the method calls below, write the printed output of the method call, and "Error" if a stack trace would be printed (i.e., if an error "escapes" the program).

a) `Exceptional.first(“hello”);`

b) `Exceptional.first(“null”);`

c) `Exceptional.first("");`

d) `Exceptional.first(null);`