



# University of Maryland College Park

## Department of Computer Science

### CMSC132 Summer 2025

### Exam #1

FIRSTNAME, LASTNAME (PRINT IN UPPERCASE):

KEY

STUDENT ID (e.g. 123456789):

#### Instructions

- Please print your answers and use a pencil.
- This exam is a closed-book, closed-notes exam with a duration of 1 hour and 20 minutes and 150 total points.
- **Do not remove the exam's staple.** Removing it will interfere with the scanning process (even if you staple the exam again).
- Write your directory id (e.g., terps1, not UID) at the bottom of pages with **DirectoryId**.
- Provide answers in the rectangular areas or the given line.
- Do not remove any exam pages. Even if you don't use the extra pages for scratch work, return them with the rest of the exam.
- Your code must be efficient and as short as possible.
- If you continue a problem on the extra page(s) provided, make a note on that problem.
- You don't need to use meaningful variable names; however, we expect good indentation.
- **You must write your name and id at this point (we will not wait for you after time is up).**
- You must stop writing once time is up.

#### Grader Use Only

#1	Problem #1 (Short Answer)	51	
#2	Problem #2 (Static Method)	30	
#3	Problem #3 (Class Implementation)	69	
<b>Total</b>	Total	150	

## Problem #1 (Short Answer) – 3 pts each

For question 1 to 8, assume the following 3 classes all in the same package.

```
public interface Device {
    void activate();
}

public class Gadget {
    private String serial;
    public void boot() {
        System.out.println("booting gadget");
    }
}

public class SmartGadget extends Gadget implements Device {
    private int signalStrength;

    public void boot() {
        System.out.println("booting smart");
    }

    public void boot(String os) {
        System.out.println("booting " + os);
    }

    public void activate() {
        System.out.println("activating smart");
    }

    public static void main(String[] args) {
        // Code goes here for Questions 1 to 6
    }
}
```

For questions 1 to 6, assume the code given is the only code placed in the main method above.

1. What is the output of the following code? **booting smart**  
`Gadget g = new SmartGadget();`  
`g.boot();`

Explain in one sentence **why** that output appears.

**Late binding allows for polymorphism and the overridden version of boot executes since the actual object is SmartGadget.**

2. Consider the following code:

```
SmartGadget s = new SmartGadget();
s.activate();
((Gadget) s).activate();
```

Will this code compile and run? If yes, what is the output. If no, explain which line fails and why (**2 sentences max**).

**Will not compile. ((Gadget) s).activate(); fails because the compiler does not see an activate method for Gadget objects.**

3. Consider the following code:

```
Device d = new SmartGadget();  
Gadget g = (Gadget) d;  
g.boot("Android");
```

Will this code compile and run? If yes, what is the output. If no, explain which line fails and why (2 sentences max).

Will not compile. `g.boot("Android");` fails because the compiler does not see a `boot` method that takes a string for `Gadget` objects (`g` is static type `Gadget`).

4. Write **C** if it will compile and **NC** if it would not compile next to each one. You do **not** need to explain why. Assume only one statement is placed in `main` at a time.

`Device d = new Gadget();` \_\_\_\_\_ **NC** \_\_\_\_\_

`Gadget g = new SmartGadget();` **C**

`Device d1 = (Device) (new SmartGadget());` **C**

5. Explain what happens when you run the following code:

```
Gadget g = new Gadget();  
SmartGadget s = (SmartGadget) g;  
s.boot();
```

Include whether it compiles, runs, or throws an exception — and why (2 sentences max).

throws an exception because the actual dynamic object referenced in `g` is a `Gadget` and therefore the cast is not valid at run time.

6. Suppose you want to call the `boot(String)` method on a `SmartGadget` object that is currently stored in a `Device` reference (variable name is `d`). Pass any string in you want. Write the necessary **cast and method call** in one line of code.

`((SmartGadget) d).boot("hello");` //the `()` must be in answer due to precedence

Directory ID:

7. Why does the following constructor cause a compilation error if added to `SmartGadget`? No more than 2 sentences.

```
public SmartGadget(String s, int signal) {
    serial = s;
    signalStrength = signal;
}
```

**serial is private field in the parent class Gadget and cannot be directly accessed in the subclass, SmartGadget**

8. Suppose you try to add this method to `SmartGadget`:

```
public boolean boot(String os) {
    System.out.println("booting " + os);
    return true;
}
```

Will it compile? If yes, explain why it is not a problem. If no, explain why. No more than 2 sentences.

**It will not compile because it is not a valid overload (override would not be the correct term here). There is no difference in the signature of this method and the public void boot(String os) already in the class.**

Assume the following class hierarchy for problems 9 to 14:

```
class Animal {}
class Mammal extends Animal {}
class Dog extends Mammal {}
```

Each class has a no-argument constructor.

Assume `import java.util.ArrayList;`

Answer each of the following **independently**.

For each, write one of:

- C – Compiles and runs with no exception
- CE – Compiles but throws an exception at runtime
- NC – Does not compile

9. `ArrayList animals = new ArrayList();`  
`animals.add("not an animal");`

**C**

10. `ArrayList<Mammal> mammalList = new ArrayList<>();`  
`ArrayList<? super Dog> dogs = mammalList;`  
`dogs.add(new Dog());` **C**
11. `ArrayList<? extends Animal> zoo = new ArrayList<Animal>();`  
`zoo.add(new Dog());` **NC**
12. `ArrayList<Dog> dl = new ArrayList<>();`  
`dl.add(new Dog());`  
`ArrayList<? extends Mammal> ml = dl;`  
`Animal a = ml.get(0);` **C**
13. `ArrayList<?> unknowns = new ArrayList<Dog>();`  
`unknowns.add(null);`  
`Object o = unknowns.get(0);` **C**
14. `ArrayList<Mammal> ml = new ArrayList<>();`  
`ml.add(new Dog());`  
`ml.add((Mammal) new Animal());` **CE**

15. You are running the **Bubble Sort** algorithm on the following array of integers to sort in ascending order:

[8, 3, 5, 2, 9]

Which two values are swapped during the third swap of the algorithm? **8 and 2**

16. Explain in no more than 3 sentences why bubble sort is  $O(n)$  in best case, but selection sort is  $O(n^2)$ .

Bubble sort only must do one linear scan to confirm no swaps and that can be done in  $O(n)$  time. Selection sort does the same work in as in the worst cases (i.e. picks the number, compares it to the remaining portion of the list, and write it back to the same place it was). Therefore, it will be  $O(n^2)$ .

17. Give the formal definition of Big-O.

A function  $f(n)$  is in  $O(g(n))$  if there exist positive constants  $M$  and  $N_0$  such that:

$M \cdot g(n) \geq f(n)$  for all  $n \geq N_0$

## **Problem #2 (Static method)**

Finish the static **rearrangeOddEvenUsingStack** method. The method rearranges the given queue so that all odd integers appear before all even integers, preserving the relative order within each group.

What you are allowed to use:

- One additional **Stack<Integer>** is permitted.
- No other data structures (e.g., additional queues, lists, arrays, etc.) may be used.
- Local **int** variables are allowed
- No private helper methods allowed

Only the following library methods are allowed:

1. You may use standard Queue methods: **offer**, **poll**, **size**
2. You may use standard Stack methods: **default Stack constructor**, **push**, **pop**, **isEmpty**

What exception to throw:

- If the input queue is **null**, throw an **IllegalArgumentException**. Any message is fine.

Assume needed import statements and no nulls elements in the queue.

```
public class QueueRearranger {  
  
    public static void rearrangeOddEvenUsingStack(Queue<Integer> q) {  
  
        //You will code this  
  
    }  
  
    public static void main(String[] args) {  
        Queue<Integer> q = new LinkedList<>();  
        q.offer(4); q.offer(3); q.offer(6);  
        q.offer(7); q.offer(2); q.offer(9);  
  
        System.out.println("Before: " + q); // Before: [4, 3, 6, 7, 2, 9]  
        rearrangeOddEvenUsingStack(q);  
        System.out.println("After:  " + q); // After:  [3, 7, 9, 4, 6, 2]  
    }  
}
```

```

public static void rearrangeOddEvenUsingStack (Queue<Integer> q) {

    if (q == null) {
        throw new IllegalArgumentException("Input queue cannot be null.");
    }

    Stack<Integer> stack = new Stack<>();
    int size = q.size();
    // Step 1: Separate odds and push evens onto stack
    int oddsCount = 0;
    for (int i = 0; i < size; i++) {
        int val = q.poll();
        if (val % 2 == 0) {
            stack.push(val);
        } else {
            q.offer(val);
            oddsCount++;
        }
    }
    // Step 2: Pop evens from stack and enqueue back (evens reversed at back)
    while (!stack.isEmpty()) {
        q.offer(stack.pop());
    }
    int evensCount = size - oddsCount;
    // Step 3: Rotate queue by oddsCount to bring evens to front (still reversed)
    for (int i = 0; i < oddsCount; i++) {
        q.offer(q.poll());
    }
    // Step 4: Now evens are at front in reversed order
    // Poll evens and push back into stack to reverse again to original order
    for (int i = 0; i < evensCount; i++) {
        stack.push(q.poll());
    }
    // Step 5: Enqueue evens from stack back (now in correct original order)
    while (!stack.isEmpty()) {
        q.offer(stack.pop());
    }
}

```

### Problem #3 (Class Implementation)

Assume the following classes are all in the same package. Assume all necessary import statements. No additional fields or methods allowed. Allowed library methods are listed in the problem. Use of **this** or **super** is not considered a library method. The use of the **length** field of an array is allowed. Local variables are ok as needed.

```
public class Book implements Comparable<Book> {

    private final String title;
    private final String author;
    private final int yearPublished;
    private final String[] chapters;
    private final int numPages;

    public Book() {
        //answer to #1 }

    public Book(String title, String author, int yearPublished,
                String[] chapters, int numPages) {
        //answer to #2 }

    public Book(Book other) {
        //answer to #3 }

    public boolean equals(Object obj) {
        //answer to #4 }

    @Override
    public int compareTo(Book other) {
        //answer to #5 }

    public Iterator<String> chapterIterator() {
        //answer to #6 }

    /**
     * Returns a string representation of the book. Make no changes.
     */
    @Override
    public String toString() {
        return title + " by " + author + " (" + yearPublished + "), "
            + chapters.length + " chapters, " + numPages + " pages";
    }

    /**
     * Returns the number of pages in the book. Make no changes.
     * You can use as needed.
     */
    public int getNumPages() {
        return numPages;
    }
}
```

```

public static void main(String[] args) {
    // Create books
    Book b1 = new Book("Algorithms Unplugged", "Turing", 1945,
        new String[]{"Intro", "Sorting", "Graphs"}, 300);
    Book b2 = new Book("Data Structures", "Knuth", 1975,
        new String[]{"Stacks", "Queues"}, 450);
    Book b3 = new Book("Zeta Functions", "Riemann", 1859,
        new String[]{"Complex Analysis"}, 150);

    // Copy constructor
    Book b4 = new Book(b1);

    // Print books
    System.out.println("Books:");
    System.out.println(b1);
    System.out.println(b2);
    System.out.println(b3);
    System.out.println("Copy of b1: " + b4);

    // Equality check
    System.out.println("\nb1.equals(b4): " + b1.equals(b4)); // true
    System.out.println("b1.equals(b2): " + b1.equals(b2)); // false

    // Chapter iteration
    System.out.println("\nChapters in b1:");
    Iterator<String> it = b1.chapterIterator();
    while (it.hasNext()) {
        System.out.println(" - " + it.next());
    }

    // Sorting with Comparable (natural order: year, title, author)
    List<Book> bookList = new ArrayList<>();
    bookList.add(b1);
    bookList.add(b2);
    bookList.add(b3);

    System.out.println("\nBooks sorted by natural order (year, title, author):");
    Collections.sort(bookList);
    for (Book b : bookList) {
        System.out.println(b);
    }

    // Sorting with BookPageComparator
    System.out.println("\nBooks sorted by number of pages:");
    Collections.sort(bookList, new BookPageComparator());
    for (Book b : bookList) {
        System.out.println(b);
    }

    // Triggering invalid input
    try {
        Book bad = new Book(null, "Someone", 2020, new String[]{"A"}, 100);
    } catch (IllegalArgumentException e) {
        System.out.println("\nException caught (null title): " + e.getMessage());
    }
}
}

```

## Output

Books:

Algorithms Unplugged by Turing (1945), 3 chapters, 300 pages  
Data Structures by Knuth (1975), 2 chapters, 450 pages  
Zeta Functions by Riemann (1859), 1 chapters, 150 pages  
Copy of b1: Algorithms Unplugged by Turing (1945), 3 chapters, 300 pages

b1.equals(b4): true  
b1.equals(b2): false

Chapters in b1:

- Intro
- Sorting
- Graphs

Books sorted by natural order (year, title, author):

Zeta Functions by Riemann (1859), 1 chapters, 150 pages  
Algorithms Unplugged by Turing (1945), 3 chapters, 300 pages  
Data Structures by Knuth (1975), 2 chapters, 450 pages

Books sorted by number of pages:

Zeta Functions by Riemann (1859), 1 chapters, 150 pages  
Algorithms Unplugged by Turing (1945), 3 chapters, 300 pages  
Data Structures by Knuth (1975), 2 chapters, 450 pages

Exception caught (null title): Invalid input!!

```
public class BookPageComparator implements Comparator<Book> {  
  
    @Override  
    public int compare(Book b1, Book b2) {  
        // answer to #7  
    }  
}
```

1. Implement the default constructor by calling the full constructor. Pass the following default values: "Untitled" as the title, "Unknown" as the author, 1900 as the year, an empty String array (new String[0]) for chapters, and 100 for the number of pages. Do not set fields directly. No library methods are allowed.

```
public Book() {  
  
    this("Untitled", "Unknown", 1900, new String[0], 100);  
}
```

2. Implement the full constructor by checking for invalid inputs: throw an `IllegalArgumentException("Invalid input!!")` if `title`, `author`, or `chapters` is null, if `yearPublished` is less than 1750, or if `numPages` is less than or equal to 0. If all inputs are valid, assign the parameters to the corresponding fields. If the input is valid, you may assign the `chapters` array directly without making a copy. No library methods are allowed.

```
public Book(String title, String author, int yearPublished,
String[] chapters, int numPages) {

    if (title == null || author == null || chapters == null || numPages <= 0 || yearPublished < 1750) {
        throw new IllegalArgumentException("Invalid input!!");
    }

    this.title = title;
    this.author = author;
    this.yearPublished = yearPublished;
    this.chapters = chapters;
    this.numPages = numPages;
}
```

3. Implement the copy constructor. If `other` is null, throw an `IllegalArgumentException("Invalid input!!")`. Otherwise, copy all fields from `other`. For the `chapters` array, create a shallow copy. No library methods allowed other than to throwing the exception.

```
public Book(Book other) {

    if (other == null) {
        throw new IllegalArgumentException("Invalid input!!");
    }

    this.title = other.title;
    this.author = other.author;
    this.yearPublished = other.yearPublished;
    this.numPages = other.numPages;

    this.chapters = new String[other.chapters.length];
    for (int i = 0; i < other.chapters.length; i++) {
        this.chapters[i] = other.chapters[i];
    }
}
```

4. Override `equals` to return `true` if the argument is the same object as the current one. If not, return `false` if the argument is not a `Book`. Otherwise, return `true` only if `yearPublished` is equal and both `title` and `author` are equal using their `String.equals` method (this is the only allowed library method).

```
public boolean equals(Object obj) {
    if (this == obj)
        return true; // Check if both references point to the same object

    if (!(obj instanceof Book)) {
        return false;
    }
    Book other = (Book) obj;
    return this.yearPublished == other.yearPublished
        && this.title.equals(other.title)
        && this.author.equals(other.author);
}
```

5. Implement `compareTo` to order `Book` objects by `yearPublished` (ascending). If the years are equal, compare `title` lexicographically using `String.compareTo` (only allowed library method). If titles are also equal, compare `author` lexicographically. Return a negative number if this book comes before `other`, zero if equal, or positive if after.

```
public int compareTo(Book other) {

    if (this.yearPublished != other.yearPublished) {
        return this.yearPublished - other.yearPublished;
    }
    int titleCompare = this.title.compareTo(other.title);
    if (titleCompare != 0) {
        return titleCompare;
    }
    return this.author.compareTo(other.author);
}
```

6. Implement `chapterIterator` to return an anonymous `Iterator<String>` over the `chapters` array. `hasNext` should return `true` if there are more elements to return. `next` should return the next chapter title and advance the index. If `next` is called when no elements remain, it must throw a `NoSuchElementException` with the message "No more chapters." You do not need to worry about the `remove` method. No library iterators or collections are allowed. Only allowed library method is throwing the exception. Private fields and local variables are allowed in the anonymous class.

```
public Iterator<String> chapterIterator() {  
  
    return new Iterator<String>() {  
        private int index = 0;  
  
        @Override  
        public boolean hasNext() {  
            return index < chapters.length;  
        }  
  
        @Override  
        public String next() {  
            if (!hasNext()) {  
                throw new NoSuchElementException("No more chapters.");  
            }  
            return chapters[index++];  
        }  
    };  
};
```

7. Implement `BookPageComparator` to compare two `Book` objects by number of pages in ascending order. Return a negative number if the first has fewer pages, zero if equal, or a positive number if more. You may use subtraction or `Integer.compare`. No other library methods are allowed.

```
public class BookPageComparator implements Comparator<Book> {  
  
    @Override  
    public int compare(Book b1, Book b2) {  
        return b1.getNumPages() - b2.getNumPages();  
        //or return Integer.compare(b1.getNumPages(), b2.getNumPages());  
    }  
}
```