# CS 61B Fall 2023

# Midterm 1

Thursday September 21, 2023

## Intro: Welcome to CS61B Midterm 1!

Your Name: _____

Your SID: _____ Location: _____

SID of Person to your Left: _____ Right: _____

Formatting:

- ◯ indicates only one circle should be filled in. ☐ indicates more than one box may be filled in. **Please fill in the shape completely.** If you change your response, **erase as completely as possible**.
- Anything you write that you ~~cross out~~ will not be graded.
- You may not use ternary operators, lambdas, streams, or multiple assignment.

Tips:

- This midterm is worth **100 points**, and there are a lot of problems on this exam. Work through the ones with which you are comfortable first. Do not get overly captivated by interesting design issues or complex corner cases you're not sure about.
- Not all information provided in a problem may be useful, and you may not need all lines.
- **We will not give credit for solutions that go over the number of provided lines or that fail to follow any restrictions given in the problem statement.**
- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- Unless otherwise stated, all given code on this exam compiles. All code has been compiled and executed before printing, but in the unlikely event that we do happen to catch any bugs in the exam, we'll announce a fix.

Write the statement below in the same handwriting you will use on the rest of the exam: "I have neither given nor received help on this exam (or quiz), and have rejected any attempt to cheat. If these answers are not my own work, I may be deducted 9,876,543,210 points on the exam."

_____

_____

_____

_____

_____

Signature: _____

# 1 Hot Curry

**(10 Points)**

(a) In Java, every **`if`** condition must also have an **`else`** clause.
○ True ○ False

(b) `System.out.println` returns a `String`.
○ True ○ False

(c) An instance method with no arguments can call static methods.
○ True ○ False

(d) An `int[]` of length 1 behaves exactly the same as an `int` when passed into a method.
○ True ○ False

(e) You can call a private method from a public method in the same class.
○ True ○ False

(f) You can call a public method from a private method in the same class.
○ True ○ False

(g) Passing all tests always means your code is failproof.
○ True ○ False

(h) An interface can extend a class.
○ True ○ False

(i) You can always cast a class to its parent class and/or parent interface(s).
○ True ○ False

(j) A class can be both `Comparable` and `Iterable`.
○ True ○ False

(k) What is the result of **`new`** `Dog() == ` **`new`** `Dog()`, assuming there are no errors?
○ `true` ○ `false`

(l) The following code compiles:

```
class Ch<E> {
    static E se() {
        return null;
    }
}
```

○ True ○ False

# 2 Java Cat-astrophe

**(15 Points)**

For each of the following lines, write the result of each statement: Write "CE" if a compiler error is raised on that line, "RE" if a runtime error occurs on that line, "OK" if the line runs properly but doesn't print anything, and the printed result if the line runs properly and prints something. If a line errors, assume that the program continues to run as if that line did not exist. Blank lines will receive no credit.

```
public class Cat {
    public void scratch(Cat c) { System.out.println("scratchy scratch"); }
    public void meow(Cat c) { System.out.println("meow"); }
}

public class Siamese extends Cat {
    public void scratch(Cat c) { System.out.println("big scratch"); }
    public void scratch(Siamese s) { System.out.println("fat scratch"); }
    public void scratch(Calico c) { System.out.println("fighting scratch"); }
    public void meow(Cat c) { System.out.println("purr"); }
}

public class Calico extends Cat {
    public void scratch(Cat c) { System.out.println("regular scratch"); }
    public void scratch(Calico c) { System.out.println("light scratch"); }
    public void meow(Siamese s) { System.out.println("meep"); }
}
```

```
Cat midori = new Cat();                          a. _____
Cat tofu = new Siamese();                        b. _____
Calico fish = new Cat();                         c. _____
Calico cliff = new Calico();                     d. _____
Calico minou = new Siamese();                    e. _____
Siamese luna = new Siamese();                    f. _____

cliff.meow(luna);                                g. _____
((Cat) cliff).meow(luna);                        h. _____
midori.meow(luna);                               i. _____
((Cat) cliff).scratch(tofu);                     j. _____
Cat.scratch(tofu);                               k. _____
midori.scratch(cliff);                           l. _____
midori.scratch((Calico) tofu);                   m. _____
cliff.meow(midori);                              n. _____
tofu.scratch(midori);                            o. _____
```

# 3 Printer Problems

**(25 Points)**

It's midterm season, and the Soda printers are working overtime to print out everyone's files. Help the printers print everything out in the right order!

We've defined a `PrintJob` class below.

```
public class PrintJob {
    List<String> pages; // a List of Strings representing the pages of the printout
    int numCopies; // an int denoting the number of copies to make.
    public PrintJob(List<String> pages, int copies) {
        assert pages.size() > 0 && copies > 0; // pages.size() and copies will be greater than 0.
        this.pages = pages;
        this.numCopies = copies;
    }
}
```

A `Printer` can be modeled as an iterator, behaving as follows:

- **public** `Printer()`: Creates a new printer with no print jobs. Contains a `jobs` deque.
- **public void** `sendJob(PrintJob job)`: Sends a print job to the back of the `jobs` deque.
- **public** `String next()`: Returns the next page to be printed. The printer should print a page from the first job in the `jobs` deque. When the job is completed, the job should be removed from the `jobs` deque, and the printer should begin printing the next PrintJob.
  - In order to print a job, exactly `numCopies` of the strings in `pages` should be printed, with the pages collated. For example, let's say we had a PrintJob with `pages = List.of("1", "2", "3")` and `numCopies = 4`. We expect `Printer` to output in the following order: `123123123123` (instead of `111122223333`)
- **public boolean** `hasNext()`: Returns if the printer has a next page to print.

For example, if we ran the following program:

```
Printer p = new Printer();
p.sendJob(new PrintJob(List.of("N", "a"),13));
p.sendJob(new PrintJob(List.of("Bat", "ma", "n"), 2));
for (int i = 0; i < 3; i++) {
    System.out.println(p.next());
}
p.sendJob(new PrintJob(List.of("!"), 5));
while (p.hasNext()) {
    System.out.print(p.next());
}
```

We should get the following output:

```
N
a
N
aNaNaNaNaNaNaNaNaNaNaBatmanBatman!!!!!
```

(a) Implement the `Printer` class methods.

```
public class Printer implements Iterator<String> {
    public Deque<PrintJob> jobs;
    public int currPage;
    public int currCopy;

    public Printer() {
        this.jobs = new ArrayDeque<PrintJob>();
        _____1_____;
        _____2_____;
    }

    public void sendJob(PrintJob job) {
        _____3_____;
    }

    @Override
    public boolean hasNext() {
        return !this.jobs.isEmpty();
    }

    @Override
    public String next() {
        if (_____4_____) {
            throw new NoSuchElementException("No more pages to print.");
        }
        PrintJob currJob = _____5_____;
        String nextPage = _____6_____;
        this.currPage = _____7_____;
        if (currPage == currJob.pages.size()) {
            _____8_____;
            _____9_____;
        }
        if (currCopy == currJob.numCopies) {
            _____10_____;
            _____11_____;
            _____12_____;
        }
        return nextPage;
    }
}
```

(b) Halfway through printing the midterm, you realize that there's a mistake! Fortunately, you kept a reference to the PrintJob `pj` you sent, so you update the PrintJob. For each of the following updates, **select all** options that could happen.

You may assume that the PrintJob had started, but not completed when you updated the PrintJob (at least one page of the Job has been printed, and at least one page of the Job has yet to be printed). The PrintJob gets modified between `next()` and `hasNext()` calls (i.e. not during `next()` or `hasNext()` calls). After the update, you call `next()` until `hasNext()` `returns` **`false`**.

i. You add finitely many additional pages to the end of the exam (ex. with `pj.pages.addLast();`)
☐ The PrintJob finishes as if the job hadn't been updated
☐ The PrintJob finishes as if the job had been updated from the start
☐ The PrintJob finishes, but some copies look like the old version, and some copies look like the new version
☐ The PrintJob never finishes
☐ The Printer crashes
○ None of the above

ii. You remove some (but not all) pages from the end of the exam (ex. with `pj.pages.removeLast();`)
☐ The PrintJob finishes as if the job hadn't been updated
☐ The PrintJob finishes as if the job had been updated from the start
☐ The PrintJob finishes, but some copies look like the old version, and some copies look like the new version
☐ The PrintJob never finishes
☐ The Printer crashes
○ None of the above

iii. You increase the number of copies (ex. with `pj.numCopies += 100;`)
☐ The PrintJob finishes as if the job hadn't been updated
☐ The PrintJob finishes as if the job had been updated from the start
☐ The PrintJob never finishes
☐ The Printer crashes
○ None of the above

iv. You decrease the number of copies (ex. with `pj.numCopies -= 100;`) After this change, `pj.numCopies` is still positive.
☐ The PrintJob finishes as if the job hadn't been updated
☐ The PrintJob finishes as if the job had been updated from the start
☐ The PrintJob never finishes
☐ The Printer crashes
○ None of the above

This page is intentionally left (mostly) blank.

# 4 Card Deck

**(30 Points)**

Eric wants to play with a deck of cards but doesn't have one on hand. Instead of buying a new deck, we decide to create a deck ourselves by implementing a `CardDeck` class (representing a standard deck of playing cards). Each card is further defined by the nested `Card` class below.

```
public class CardDeck {
    public class Card {
        public static final String[] SUITS = new String[]{"Hearts", "Clubs", "Diamonds", "Spades"};
        public static final String[] RANKS = new String[]{"Ace", "2", "3", "4", "5",
        "6", "7", "8", "9", "10", "Jack", "Queen", "King"};
        public int suitIndex;
        public int rankIndex;

        public Card(int suitIndex, int rankIndex) {
            this.suitIndex = suitIndex;
            this.rankIndex = rankIndex;
        }

        @Override
        public boolean equals(Object obj) {
            if (obj instanceof Card card) {
                return this.suitIndex == card.suitIndex && this.rankIndex == card.rankIndex;
            }
            return false;
        }
    }

    public List<Card> cards;

    public CardDeck() {
        cards = new ArrayList<>();
        createDeck();
    }

    private void createDeck() { /* part a */ }

    public void faroShuffle() { /* part b */}

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof CardDeck cardDeck) {
            return this.cards.equals(cardDeck.cards);
        }
        return false;
    }
}
```

(a) Implement the function `createDeck`, which sets `cards` to an ArrayList containing a deck of 52 unique cards, one for each combination of the 13 ranks and 4 suits. Your solution may add the cards in any order, and may assume that `cards` begins empty. You may not have to use all lines provided, but you will recieve no credit if you go over the number of lines provided.

```
private void createDeck() {
    _____
    _____
    _____
    _____
    _____
    _____
    _____
    _____
    _____
}
```

(b) A faro shuffle involves splitting the deck into two equal halves and then interweaving them perfectly. For example, a faro shuffle of `ints [1, 2, 3, 4, 5, 6]` would result in `[1, 4, 2, 5, 3, 6]`. Write the `faroShuffle` which faro-shuffles the `List cards`. You may assume that there are always exactly 52 cards in the deck.

```
public void faroShuffle() {
    List<Card> cardCopy = new ArrayList<>(cards);
    for (int i = 0; i < cards.size() / 2; i += 1) {

        this.cards.set(_____1_____);

        this.cards.set(_____2_____);
    }
}
```

(c) For each of the following, how many iterations will the while loop run? **Hint: If you perform 8 faro shuffles on a deck of 52 cards, the deck will return to its original order.** If it does not terminate, please write "inf". Assume that `faroShuffle` in part (b) has been correctly implemented for `CardDeck`.

1

```
CardDeck deck = new CardDeck();
CardDeck deck2 = deck;
deck2.faroShuffle();
while (deck2 != deck) {
    deck2.faroShuffle();
}
```

2

```
CardDeck deck = new CardDeck();
CardDeck deck2 = deck;
deck2.faroShuffle();
while (!deck2.equals(deck)) {
    deck2.faroShuffle();
}
```

3

```
CardDeck deck = new CardDeck();
CardDeck deck2 = new CardDeck();
deck2.faroShuffle();
while (deck2 != deck) {
    deck2.faroShuffle();
}
```

4

```
CardDeck deck = new CardDeck();
CardDeck deck2 = new CardDeck();
deck2.faroShuffle();
while (!deck2.equals(deck)) {
    deck2.faroShuffle();
}
```

(d) We'll now write a comparator to play a very simple game of whichever card has the higher value. In this game, we consider Ace = 1, Jack = 11, Queen = 12, King = 13, and all of the number cards to equal their respective number (2 through 10). If the ranks are the same, we will then look at their suit. We consider that Hearts < Clubs < Diamonds < Spades. Which of the following lines should replace the numbered boxes in the below code?

```
class SimpleGameComparator implements Comparator<Card> {
    @Override
    public int compare(Card c1, Card c2) {
        if ([1]) {
            return [2];
        }
        return [3];
    }
}
```

[1]

- ◯ `c1.rankIndex == c2.rankIndex`
- ◯ `c1.suitIndex == c2.suitIndex`

[2]

- ◯ `c1.rankIndex - c2.rankIndex`
- ◯ `c1.suitIndex - c2.suitIndex`
- ◯ `c2.rankIndex - c1.rankIndex`
- ◯ `c2.suitIndex - c1.suitIndex`

[3]

- ◯ `c1.rankIndex - c2.rankIndex`
- ◯ `c1.suitIndex - c2.suitIndex`
- ◯ `c2.rankIndex - c1.rankIndex`
- ◯ `c2.suitIndex - c1.suitIndex`

# 5 I Signed an NDA

**(20 Points)**

High-dimensional nested arrays are quite cumbersome in Java. For example, a 9-dimensional `int` array `arr` must be declared as `int[][][][][][][][][] arr`. Angel wants to devise a class to store his $n - D$ data.

(a) Complete the constructor for `NDArray`, which takes in a dimension $D$ and a width $W$, such that the `NDArray` represents a $\underbrace{W \times W \times ... \times W}_{\text{D times}}$ array. You may assume that $D >= 1$ and $W >= 1$.

```
public class NDArray {
    public int value;
    public int dimension;
    public NDArray[] arr;

    public NDArray(int D, int W) {
        dimension = D;

        if (_____1_____) { return; }

        arr = _____2_____;

        for (_____3_____) {

            _____4_____;
        }
    }
}
```

(b) Angel now needs a way to get items from the `NDArray`. Complete `get`, which is an instance method and returns the item at `List<Integer> coords`. For example, if we have a `NDArray nda` of dimension 2 representing $\begin{bmatrix}[5, & 4], \\ [1, & 9]\end{bmatrix}$ then `nda.get(List.of(0, 1))` should return `4`. You may assume that each individual coordinate is between `0` and `W-1`, inclusive. Hint: You may use the `subList` method of `List`.

```
public int get(List<Integer> coords) {
    if (coords.size() != dimension) {
        throw new IllegalArgumentException();
    }
    if (_____1_____) {

        _____2_____;
    }
    int index = coords.get(0);

    return _____3_____;
}
```

**Nothing on this page is worth any points.**

---

# 61Bonus Question

**(0 Points)**

How many Faro shuffles do you need to perform on a deck of 1000000 cards in order to return the deck to its original order?

# Feedback

**(0 Points)**

Leave any feedback, comments, concerns, or drawings below!

# Deque Interface API

```
public interface Deque<E> {
    /** Inserts the specified element at the front of this deque. */
    void addFirst(E e);

    /** Inserts the specified element at the end of this deque. */
    void addLast(E e);

    /** Retrieves and removes the first element of this deque. */
    E removeFirst();

    /** Retrieves and removes the last element of this deque. */
    E removeLast();

    /** Returns the number of elements in this deque. */
    int size();

    /** Returns true if this deque contains no elements. */
    boolean isEmpty();
}

// Implementations of Deques covered in class
public class ArrayDeque<E> implements Deque<E> {...}
public class LinkedListDeque<E> implements Deque<E> {...}
```

# List Interface API

```
public interface List<E> {
    /** Constructor that takes in a List and creates a new List copy with the same elements. */
    List<E> (List<E> e);

    /** Appends the specified element to the end of this list. Runs in constant time. */
    void add(E e);

    /** Returns the element at the specified position in this list. */
    E get(int index);

    /** Replaces the element at the specified position in this list with the specified element. */
    E set(int index, E element);

    /** Returns the number of elements in this list. */
    int size();

    /** Returns true if this list contains no elements. */
    boolean isEmpty();
```

```
    /** Returns a view of the portion of this list between the specified fromIndex, inclusive, and
    toIndex, exclusive. */
    List<E> subList(int fromIndex, int toIndex);

    /** Lists are defined to be equal if they contain the same elements in the same order. */
    boolean equals(Object o);

    /** Returns an immutable list containing an arbitrary number of elements. For example,
    List<Integer> example = List.of(1, 2, 3); is a valid assignment. */
    static <E> List<E> of(E... elements);
}

// Implementations of Lists covered in class
public class ArrayList<E> implements List<E> {...}
public class LinkedList<E> implements List<E> {...}
```

# Math Class API

```
public class Math {
    /** Returns the smaller of two int values. */
    public static int min(int a, int b) { ... }

    /** Returns the greater of two int values. */
    public static int max(int a, int b) { ... }
}
```

# Integer Class API

```
public class Integer {
    /** A constant holding the minimum value an int can have, -2^31. */
    public static final int MIN_VALUE = -2147483648;

    /** A constant holding the maximum value an int can have, 2^31-1. */
    public static final int MAX_VALUE = 2147483647;
}
```

Q3: You may also use E getFirst() and E getLast() in the Deque interface:
getFirst: Retrieves the first element of the deque, but does not remove it
getLast: Retrieves the last element of the deque, but does not remove it

Q3b: You may assume for 3b ONLY that List has addLast and removeLast correctly implemented. This should only affect the examples given

Q4c: You may assume that creating a new CardDeck always returns the cards in the same order