CSDS 132 – Programming in Java
Final Summary Sheet

Objectives:

- Understand how to use primitive and non-primitive types, and the differences between the two.
- Write classes that follow good coding practices and utilize inheritance and polymorphism.
- Read and use an API.
- Manipulate Strings, arrays, and linked lists.
- Understand how to use generic types.
- Understand how to throw and catch exceptions.

Conceptual Questions

Primitive Types

1. Which typecasts are automatic, and which ones require an explicit typecast?
    a. int → double

    b. long → double

    c. long → float

    d. float → double

    e. double → float

    f. double → char

    g. char → byte

    h. char → short

2. What is the resulting type of the following operations?
    a. (short) 1 + (short) 2

    b. (byte) 1 + (char) 'a'

    c. 1 + 2.4f

d. 3 + 4L

e. 43.2 + 12

Non-Primitive Types

3. Fill in the table below regarding differences between primitive and non-primitive types.

Type	Primitive	Non-Primitive
What value is stored?		
Testing for equality		
Comparing types		
How to declare/assign?		

4. Suppose we had the following statement: `Object o; o = "123";`
   a. What does the first line do?

   b. What does the second line do?

   c. What is the current type?

   d. What is the true type?

   e. If I call a static method on o, which method (current or true type) would be run?

   f. If I call a non-static method on o, which method (current or true type) would be run?

5. Which kind of non-primitive type (enum, class, abstract class, or interface) would you use for…
    a. … cardinal directions, where only values of north, south, east, and west should be allowed?

    b. … flying entities, where objects (like birds, airplanes, etc.) are not necessarily related to each other?

    c. … generic vehicles, which require a more specific type for instantiation?

    d. … cars, which don't require a more specific type for instantiation?

6. Regarding interfaces, abstract classes, and classes, which can be the true type? Which can be the current type?

7. What rules do we follow when typecasting for both non-primitive and primitive types (Hint: think about narrower vs. wider types)

Object-Oriented Programming

8. What is meant by polymorphism?

9. There are four access modifiers: public, private, protected, and package-private (default). Which one would you use for…
    a. … a class that should be accessed outside of its package?

b. … an interface that should only be accessed within the package or if extended?

c. … a class that should only be accessed within the package?

d. … a nested class that should only be accessed by its containing class and nothing else?

10. As we're studying for the final exam, it's only fitting that we talk about `final`. What does `final` mean…
    a. …when we make a field `final`?

    b. …when we make an instance method `final`?

    c. …when we make a class `final`?

11. Fll in the below table regarding extending classes and interfaces.

Type	**What keyword do we use? (extends or implements)**	**How many types can we extend/implement?**
class → abstract class		
class → interface		
interface → interface		

class → class		

12. See State.java.
    a. In the class definition, which are fields?

    b. Which are methods? Are these methods instance, static, or are there some of both?

    c. What is `State()`? What is the difference between the two?

    d. What class does `State` extend? What are the methods you are required to know about this class?

Memory

13. Let's first talk about the heap.
    a. What parts of classes are stored in the heap?

    b. What parts of instances are stored in the heap?

    c. When a non-static method is called, how does Java determine which method is run? (Think about true types!)

14. Let's talk about the stack.
    a. When is the stack used, and what information is stored on the stack?

b. What is added to the stack when compound statements are run?

15. Look at the equals() method in State. Describe the method call stack during its execution.

Loops

16. What are some rules and guidelines to follow when creating loops?

17. See the method below.

```
public String generateSequence(int k) {
      String s = "";
      for (int i = 1; i <= k; i++) {
            s += i;
            s += " ";
      }
      return s;
}
```

a. What is/are the precondition(s) of the loop?

b. What is/are the postcondition(s) of the loop?

c. What is/are the loop subgoal(s) of the loop?

d. What is wrong/could be changed for the above loop?

18. How would we test that loop?
    a. First, let's do test zero, test one, test many. What would we do to…
        i. … test zero?

        ii. … test one?

        iii. … test many?

    b. Second, let's do test first, test middle, test last. What would we do to…
        i. … test first?

        ii. … test middle?

        iii. … test last?

Arrays, ArrayLists, and LinkedLists

19. What's the difference between Arrays and ArrayLists?

20. Let's compare ArrayList/Arrays and LinkedList. (You'll learn more about this in CSDS233!)
    a. What are the advantages and disadvantages of Arrays/ArrayLists?

    b. What are the advantages and disadvantages of LinkedLists?

c. When would you use ArrayLists, and when would you use LinkedLists?

21. Let's say we have an array, and we wanted to search that array.
    a. We have 2 methods of searching: linear search and binary search. What is linear search, and what is binary search?

    b. When should we use binary search over linear search, and when should we use linear search over binary search?

22. As a parameter, what is the difference between `Object... arr` and `Object[] arr`?

Generics and Wildcards

For Questions 23 and 24, suppose we had the following hierarchy:

```
public class Device extends Object
public class Phone extends Device
public class CellPhone extends Phone implements
        Comparable<CellPhone>
public class IPhone extends CellPhone
public class AndroidPhone extends CellPhone
```

23. Which of the following typecasts are legal?
    a. LinkedList<CellPhone> → LinkedList<Phone>

    b. LinkedList<CellPhone> → List<CellPhone>

c. LinkedList<CellPhone> → List<Phone>

24. What classes can be the generic type if I declare my LinkedList…
    a. … as `LinkedList<T extends Device>`?

    b. … as `LinkedList<? extends CellPhone>`?

    c. … as `LinkedList<T extends Comparable<T>>`?

    d. … as `LinkedList<T extends Comparable<? super T>>`?

25. Support I declare a `LinkedList<?> list = new LinkedList<>();` Can I…
    a. … get the length of the LinkedList?

    b. … order the LinkedList?

    c. … print the objects of the LinkedList?

    d. … return the value at a specific index?

    e. … remove an element from the LinkedList?

f. … add an element to the LinkedList?

Iterable, Iterator, Comparable, Comparator

26. Fill out the below table regarding Iterable, Iterator, Comparable, and Comparator.

Interface	What does it mean when a class implements this interface?	What methods do I need to implement the interface?
Iterable		
Iterator		
Comparable		
Comparator		

27. The methods of Comparable and Comparator return 3 types of values: <0, =0, and >0. What do these values mean?

a. <0:

b. 0:

c. >0:

Exceptions

28. What are the 2 ways of handling an exception in a method?

29. What are the 2 ways that we can throw an exception?

30. How do we handle exceptions using a try/catch block?

31. We can add a `finally` block to a try/catch block. What does `finally` do?

32. See the following code:

```
public void method() {
     try {
          double num = Math.random(); // Returns a random
number between 0 and 1
          if (num > 0.8)
               throw new RandomException();
          else if (num > 0.5)
               throw new BadNumberException();

          System.out.print("A");
     } catch (RandomException e) {
          System.out.print("B");
     } catch (Exception e) {
          System.out.print("C");
     } finally {
          System.out.print("D");
     }
}
```

a. Suppose the num was 0.3. What would be printed?

b. What if the number was 0.6?

c. What if the number was 0.9?

d. Suppose we flipped the catch blocks so that the Exception catch block was the first catch block, followed by the RandomException catch block. What would change for a-c, if anything?

Nested Classes and Method References

33. Fill out the table below detailing nested classes and their different "types."

Type	What is it?	When should we use them?
Nested Classes		
Anonymous Classes		
Lambda Shortcuts		
Method References		

34. Let's say I have a class State and a nested class City inside State.
    a. Let's say this nested class was static and has a static int field population. How would I access it?

    b. Now, let's say that this nested class was non-static and it has a method getPopulation(). How would I make a new instance and call this method?

    c. Now, let's say that the nested class was non-static. Now, consider that I want to find the name of its State. From within the body of the nested class, how would I access the containing class's method getName()?

Threads and JavaFX

35. Why might we want to use threads?

36. What problem might occur if two threads try to access the same variables? How do we prevent this issue?

37. How does Java get a variable's value without the "volatile" keyword present? How does Java get the value when the "volatile" keyword is added?

38. Given the following variable declarations below, how would you write a synchronized block of code that decreases `myNum` by 5, using `numLock` as the lock?

```
int myNum = 100;
Object numLock = new Object();
```

39. Why might we want to write a synchronized block as we did in the previous question?

40. How do JavaFX properties help with multi-threads?

41. What is needed for each JavaFX property?

42. What does the bind method do in regards to JavaFX properties? Why is this useful?

Wrapper Classes and Optional

43. Why might we want to use Optional when working with potentially null objects?

44. Is unwrapping automatic with Optional? If not, how do we get the value wrapped in an Optional object?

45. What is the difference between a primitive type and its corresponding wrapper class? Why might we want to use wrapper classes?

46. Which of the following are legal declarations using wrapper classes?

```
a. Integer x = 5;
b. Double y = 5;
c. Double z = 5.0;
d. int primIntX = x;
e. int primIntZ = z;
f. double primDoubleX = x;
g. double primDoubleZ = z;
```

47. In the previous problem, what is really happening in statement (a)? What about in statement (d)?

Coding Questions

Creating Classes

48. Create a class named `Phone`.
    - It should have a double price, String operatingSystem, String name, and an int describing the number of pixels.
    - When you create a value of type Phone, all values must be specified by the code creating the value.
    - There should be a way to separately retrieve all variables.
    - There should be a way to change price value only.
    - Users/programmers should not be allowed to create a value of type Phone by itself.

Overriding Methods, Comparable/Comparator

49. Using your previous implementation of Phone,
    - Change it so your Phones are equal only if the operating system and the number of pixels are the same.
    - Change it so your Phones can be Comparable, and that phone1 is "smaller" than phone2 if phone1's number of pixels are less than phone2's number of pixels.
        - If the 2 phones have the same number of pixels, the "smaller" phone is the one with the less price.
    - Change it so your Phones can also be compared by price only. Name this method `compareByPrice`. In this case, phone1 is "smaller" than phone2 if phone1's price is less than phone2's price.
        - Use a lambda shortcut.
        - Null values should be considered and be considered "larger" than all other values.
50. Create 2 classes: `IPhone` and `AndroidPhone`.
    - Both classes should extend Phone and be able to be instantiated.
    - IPhone should have an operating system of "iOS", while AndroidPhone should have an operating system of "AndroidOS."
    - IPhone should have an additional field named "iOSVersion", which should be a String.
        - This should be taken in as input when a value of type IPhone is created.

        - This should be able to be accessed and changed.
    - AndroidPhone should have an additional field named “company”, which should be a String.
        - This should be taken in as input when a value of type AndroidPhone is created.
        - This should be able to be accessed, but not changed.

51. Using your previous implementation of IPhone and AndroidPhone,
    - Change it so AndroidPhones are only equal if their operating system, number of pixels, and company is the same.
    - Change it so IPhones are only equal if their operating system, number of pixels, and iOS version is the same.
    - Change it so AndroidPhones, when printed, return the company followed by the name, separated by a space.
    - Change it so IPhones, when printed, return the name and iOS version. For instance, if the name was “iPhone X” and the iOS version was “12.1”, it should return “iPhone X, iOS 12.1”.

List and Generic Stuff, Exceptions

52. Using your previous implementation of Phone, IPhone, and AndroidPhone,
    - Create a class named `PhoneStore` that sells IPhones, AndroidPhones, or both.
        - You might need to use a generic.
    - PhoneStore should have a field with an array of its respective type.
        - There should be no way of changing this value directly, but there should be a way to access this value.
    - When creating a PhoneStore, an int value numPhones should be specified, indicating the size of the array.
        - There should be no way of changing this value directly, but there should be a way to access this value.
        - You should make an array of the same length with length of numPhones.
            - To do this, you must do `T[] arr = (T[]) new Phone[numPhones];`
    - PhoneStore should have a method remove(int index), which should return the object at that specific array index.
        - If the element at that array index is null, this method should throw NoSuchElementException.
        - If the index provided is outside the bounds of the array, this method should also throw a NoSuchElementException.
    - PhoneStore should have a method add(T phone), which should add a phone to the array at the earliest available index.

- The “earliest available index” is defined as the lowest index where the object is null in the array.
- If all indices of the array are occupied, throw an IllegalStateException.

Iterable and Iterator

53. Using your previous implementation of PhoneStore,
    - Have a separate nested class PhoneStoreIterator that implements Iterator.
    - PhoneStoreIterator should be able to be iterated only if there is another non-null object to be read.
    - The next iteration of PhoneStoreIterator should skip over null values and return the next non-null value. If there is none, this should throw NoSuchElementException.
    - Make PhoneStore implement Iterable and implement the required methods, using the PhoneStoreIterator created above as an iterator.
54. Using your previous implementation of PhoneStore,
    - Add a method replenishStock(T phone) that adds the specified phone to the PhoneStore until the PhoneStore is “full.”
        - A PhoneStore is “full” if there are no null values at any position in the array.
55. Using your previous implementation of PhoneStore,
    - Make a separate class `PhoneMain`.
    - In PhoneMain, create a static method `printPhones()` that takes in a PhoneStore and prints out all the names of the line.
    - In PhoneMain, create a static method `totalPrice()` that returns a double representing the total price of all the phones at the store.

Using API, Method References

Using the following class and method(s):

java.util.Arrays

- **static void sort(Object[] a)**: Sorts the specified array of objects into ascending order, according to the natural ordering of its elements (using Comparable).
- **static void sort(T[] a, Comparator<? super T> c)**: Sorts the specified array of objects according to the order induced by the specified comparator.
- **static <T> Stream<T> stream(T[] array)**: Returns a sequential Stream with the specified array as its source.

java.util.Stream

- **void forEach(Consumer<? super T> action):** Performs an action for each element of this stream.

56. Using your previous implementation of PhoneMain,
    - In PhoneMain, create a static method printByPrice() that takes in a PhoneStore and prints out all the names of the line in increasing order by price.
        - This will involve sorting the phones by price.
        - Instead of using a loop for printing out the phones, try to use the above classes and use a method reference.

Strings/Arrays

57. Create a new class, `ToDo`.
    - The class should hold a String array of tasks, which stores descriptions for each task that must be done.
    - The class should have a constructor which takes in a String array as input and assigns the value of that array to *tasks*.
58. Using your previous implementation of ToDo,
    - Create a method, addToTask(String descToAdd), which concatenates descToAdd to each String in *tasks*.
    - Overload addToTask with a new method header, addToTask(String descToAdd, int start, int end), such that descToAdd are only added to the tasks at indices between start and end (inclusive).
        - If either start or end are out of the bounds of the array, catch the ArrayIndexOutOfBoundsException and print a message that says "Invalid start or end index."
59. Using your previous implementation of ToDo,
    - Create a method, makeExcited(int index), which takes the String in *tasks* at index and changes any periods to exclamation points.
        - For example, the String "Mow the lawn. Must be done after watering the plants. Need to complete before 5 pm." Will become "Mow the lawn! Must be done after watering the plants! Need to complete before 5 pm!"
    - Create a method, taskExists(String task), which checks if the given argument *task* exists in the *tasks* array. If it does exist, it returns true, otherwise it returns false.

Loops

Using the following class and method:

java.util.LinkedList

- **ListIterator<E> listIterator(int index)**: Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.

60. In a new class named Loops, create a method `printEachCharacter()` that takes in a `LinkedList<LinkedList<String>>>` and prints out all of the characters on their own line.

- Use Java's implementation of LinkedList.
- ListIterator extends Iterator, meaning that it has access to the same methods as Iterator.
- This will require 3 loops. Make one loop a `for` loop, one loop a `while` loop, and one loop a `foreach` loop.