CSDS 132 – Programming in Java
Final Summary Sheet

* **Note**: Not all of these concepts will be covered on the final. Concepts that are unlikely to show up on the final are marked with an asterisk (*).

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
    automatic

    b. long → double
    automatic

    c. long → float
    automatic

    d. float → double
    automatic

    e. double → float
    Typecast needed

    f. double → char
    Typecast needed

    g. char → byte
    typecast

h. char → short
typecast

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

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

   c. 1 + 2.4f
   float

   d. 3 + 4L
   long

   e. 43.2 + 12
   double

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?	The value itself	Reference to the memory address
Testing for equality	==	.equals()
Comparing types	> <	compareTo(), after implementing Comparable
How to declare/assign?	int x = 5;	Object o = new Object();

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

   b. What does the second line do?

Assigning value “123” to o.

c. What is the current type?
   Object.

d. What is the true type?
   String.

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

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

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?
      Enum, where the variable can take one from a set of constant values

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

   c. … generic vehicles, which require a more specific type for instantiation?
      Abstract class, which cannot be instantiated directly

   d. … cars, which don’t require a more specific type for instantiation?
      Class, which can be instantiated

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

   Only classes can be the true type. Interfaces and abstract classes cannot be instantiated themselves. They can, however, 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)

   Narrower to wider types are automatic, while wider to narrower require explicit typecasts, for both non-primitive and primitive types.

Object-Oriented Programming

8. What is meant by polymorphism?

   Polymorphism means that an object is simultaneously an instance of its class and all of its superclass. For example, a String object is simultaneously a String and an Object.

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?
      public

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

   c. … a class that should only be accessed within the package?
      Default (package-private)

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

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`?
       Once the final field is assigned, it cannot be changed to a different value. It does not have to be assigned initially.
    b. …when we make an instance method `final`?
       The final instance method cannot be overridden.

    c. …when we make a class `final`?
       The final class cannot be extended.

11. Fill 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	extends	Only 1, that includes regular classes as well
class → interface	implements	1 or multiple
interface → interface	extends	1 or multiple
class → class	extends	1, that includes abstract classes as well

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

    b. Which are methods? Are these methods instance, static, or are there some of both?
       getPopulation(), getTwoLetterCode(), setPopulation(), setTwoLetterCode(), equals().

These methods are all instance methods.

c. What is `State()`? What is the difference between the two?
They are constructors, they are functions that are called when making an instance of a class. The two have different parameters, one takes none while the other takes two arguments.

The constructor with no parameters is called the default constructor.
The constructor with two parameters is an overloaded constructor.

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

Object, since no class is specified. The methods of importance are boolean equals(Object o) and String toString().

Memory*

13. Let's first talk about the heap.
    a. What parts of classes are stored in the heap?
    The superclass, the static fields, all methods, constructors, and any nested classes.

    b. What parts of instances are stored in the heap?
    The true type, and all non-static fields of all polymorphic types of the instance.

    c. When a non-static method is called, how does Java determine which method is run? (Think about true types!)
    Java goes to the instance part of the heap using the reference and looks at the true type. Then it goes to the true type class of the heap. It looks to see if it has a method that matches and if not it goes to the superclass and repeats this process until it finds the method.

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

Stacks are used when calling methods or even in compound statements, where local variables might exist only within compound statements. Stacks contains method parameters, this, referencing the instance for instance methods, local variables declared within the method, and any bookkeeping to return from a function.

b. What is added to the stack when compound statements are run?
Any local variables declared within the compound statement.

~~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?
Have each loop do only one subgoal or task at a time. Make increments simple. Think about preconditions and postconditions when designing loops. Avoid unnecessary traversals.

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?
String s is an empty String. i = 1.

b. What is/are the postcondition(s) of the loop?
String s contains a sequence of integers from 1 to k.

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

String s contains a sequence from 1 to i-1.

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

The above loop uses String concatenation within a loop. This should be changed to StringBuilder to avoid the creation of a new String upon each loop iteration.

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?

        Test if the method returns an empty String when we give it a nonnegative integer.

        ii. … test one?

        Test if the method returns just one number from a sequence of 1.

        iii. … test many?

        Test if the method returns multiple numbers of a sequence.

Arrays, ArrayLists, and LinkedLists

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

    Arrays and ArrayLists are both non-primitive. Arrays cannot be resized, while ArrayLists can.

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?

    Arrays/ArrayLists allow for quick access of information. For instance, if we wanted to find something at the 5th index, it is really easy and efficient to do this.

    Arrays/ArrayLists are not good for adding/removing elements. For adding elements, it either is impossible (for arrays) or is incredibly inefficient (for

ArrayLists). For removing elements, it requires shifting over the elements, which takes a lot of time.

b. What are the advantages and disadvantages of LinkedLists?
LinkedLists are very good for adding and removing elements. Adding an element is as simple as tacking on a Node at the beginning or end of the list, while removing an element is as simple as changing the references of the Nodes.

LinkedLists are bad at accessing information deep within the list. This is because there isn't a reference for the middle Nodes, so if we were to access the information, we would have to begin at the head or tail of the list and work our way inward until we find the element we want.

c. When would you use ArrayLists, and when would you use LinkedLists?
Use ArrayLists/Arrays if the size of the data isn't changing and you need to access elements at all positions. Use LinkedLists if data access isn't a priority but data addition/removal is a priority.

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?
    In linear search, we start at the beginning of the array and keep going until we find the element we are looking for. In binary search, we start in the middle and compare the middle value to the value we are looking for. If the value we are looking for is greater, then we look in the greater half; otherwise, we look in the lesser half. We keep repeating until we find the element.

    b. When should we use binary search over linear search, and when should we use linear search over binary search?
    Binary search is better and more efficient if we have a sorted array. If we do not have a sorted array, it's better to just use linear search, instead of sorting and then using binary search.

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

    Object… arr is specifically used when taking inputs to methods. It must be the last input to the method, and allows an array of Object to be constructed from individual Object inputs. For instance, if I had a method like public void add(Object… objects), I could pass in 3 Objects as input, 4 Objects as input, and so on. Java would then automatically construct an array out of these separate Objects. Object[] does not allow this and requires an Object array to be accepted as input.

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>

       Illegal, generic types must be the same.

    b. LinkedList<CellPhone> → List<CellPhone>

       Legal, going from narrower type (LinkedList) to wider type (List) AND generic types are the same

    c. LinkedList<CellPhone> → List<Phone>

       Illegal, generic types must be the same.

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

       Device, Phone, CellPhone, IPhone, AndroidPhone

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

       CellPhone, IPhone, AndroidPhone

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

CellPhone only. This is because CellPhone implements Comparable<CellPhone> but IPhone doesn't implement Comparable<IPhone>, so it doesn't fit the generic restrictions.

d. … as `LinkedList<T extends Comparable<? super T>>`?
CellPhone, IPhone, and AndroidPhone. This is because once we introduce ? super T, we broaden the Comparable type to any superclass of T. In this case, IPhone implements Comparable<CellPhone>, and since CellPhone is a super class of IPhone, IPhone is a valid type.

25. Support I declare a `LinkedList<?> list = new LinkedList<>();` Can I…
    a. … get the length of the LinkedList?
    Yes, length is not reliant on the type of object being stored in the LinkedList.

    b. … order the LinkedList?
    No, the type stored in the LinkedList might not be Comparable and therefore might not be able to be ordered

    c. … print the objects of the LinkedList?
    Yes, all classes extend Object and therefore the wildcard will still have a toString method. Therefore the objects can still be printed.

    d. … return the value at a specific index?
    Yes, the index is not dependent on what type is being stored in the LinkedList.

    e. … remove an element from the LinkedList?
    Yes, removing the element is not reliant on what type is being stored in the LinkedList.

    f. … add an element to the LinkedList?
    No, since you must know what type is stored in the list in order to know what types are legal to be added.

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	Can be looped over using a for-each loop	public Iterator<T> iterator()
Iterator	Allows programmers to define how to loop and iterate.	public boolean hasNext(), public T next()
Comparable	Gives one specific ordering for the class	public int compareTo()
Comparator	Can have multiple different orderings defined	public int compare(T o1, T o2)

27. The methods of Comparable and Comparator return 3 types of values: <0, =0, and >0. What do these values mean? Let o1.compareTo(o2):
    a. <0:
    o1 is smaller/less than o2

    b. 0:
    o1 has equal preference to o2 in ordering

    c. >0:
    o1 is bigger/greater than o2

Exceptions

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

The first way is to handle the exception in a try/catch block to prevent improper program exits. The second way is to not handle it, and throw the exception. The calling method can handle it or pass it to its calling method, and so on.

29. What are the 2 ways that we can throw an exception?
    To explicitly throw the Exception in the method, we can do this by doing throw e; or throw new Exception();

30. How do we handle exceptions using a try/catch block?
    You put the code that could throw an exception in the try block. Then, in a separate catch block, you specify what exception could be thrown (or Exception, which is the generic exception and will catch all Exceptions), and specify what you want to do to handle the Exception.

31. We can add a `finally` block to a try/catch block. What does `finally` do?
    You can also have a finally block at the end of a try/catch which will run regardless if an exception is thrown or not.

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?
       AD

A is printed at the end of the try block, and D is printed in the finally block.

b. What if the number was 0.6?
CD
C is printed because a BadNumberException is thrown, which is caught in the 2nd catch block. D is printed in the finally block.

c. What if the number was 0.9?
BD
B is printed because a RandomException is thrown, which is caught in the 1st catch block. D is printed in the finally block.

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?
Nothing would change with the exception of c). A RandomException would be caught by the Exception catch block since it would be caught first before the RandomException catch block would be called. This would result in CD being printed instead of BD.

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	Class within a class.	Group similar classes together (especially when it doesn't make sense for a nested class to live outside of the containing class ex. LLNode inside of LinkedList).
Anonymous Classes	Nested class with no name.	Only need to use the class once (ex. implementing Iterators)
Lambda Shortcuts		Need to implement an anonymous class whose

	Shorthand used to create an anonymous class that implements an interface with a single non-default method.	interface only has one non-default method.
Method References	Shorthand for lambda where the method has already been implemented.	Need to implement an anonymous class with a method that has been written before.

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?
    ```
    State.City.population
    ```

    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?
    ```
    State state = new State();
    State.City city = new state.new City();
    city.getPopulation();
    ```

    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()?
    ```
    State.this.getName();
    ```

Threads and JavaFX*

35. Why might we want to use threads?
    Threads are used to run multiple processes at the same time.

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

The process in one thread might modify the variable, leading to unexpected behavior in the other thread. We can lock parts of the code so that only one thread can access the variable at a time.

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?
`volatile` tells Java that the variable may be changed by other threads. Without `volatile`, Java will just access the variable's value at that particular time. With `volatile`, Java will store the value of the variable throughout the lifecycle of the thread, so that even if the value is changed by another thread, Java will have the old value saved.

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();
synchronized (numLock) {
    myNum -= 5;
}
```

39. Why might we want to write a synchronized block as we did in the previous question?
See 36.

40. How do JavaFX properties help with multi-threads?
JavaFX properties allow for communication between threads.

41. What is needed for each JavaFX property?
    - A private field that stores the value of the property
    - A getter method for the field
    - A setter method for the field
    - A public or protected method that returns a Property

42. What does the bind method do in regards to JavaFX properties? Why is this useful?
    It links the value of one property in a class to the value of another property in a different class. That way, if the value of the first property changes then the value of the second value will also change. This is useful if we're running multiple threads and want changes in one thread to change the values of a property in a different thread.

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?
    Wrapper classes are non-primitive types, while primitive types are not. Wrapper classes can be used in generic types such as LinkedList, ArrayList, etc.

46. Which of the following are legal declarations using wrapper classes?
    a. `Integer x = 5;`
    b. `Double y = 5;` Need to typecast to (int) before wrapping
    c. `Double z = 5.0;`
    d. `int primIntX = x;`
    e. `int primIntZ = z;` Trying to fit wider type (Double) into narrower type
    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)?
    a. When you put in 5, it automatically makes a new Integer(5) and puts that into x.

d. Java calls the method that calls the primitive value of the integer out of the Integer wrapping class (using intValue()), and then assigns it to the primitive integer primIntX.

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.