Quiz # 1
Name: Antonio Rosado
I declare that this prequiz is entirely my own work
Submission Date: 1/29/23

Review Frame (Java)

IOOP Review

Java Doc:

- @param - for parameters
- @return - for return statements
- @author - info about person writing code
- @version - version of class

Errors:

- Syntax errors - errors with coding style depending on language. Compiler will spot this error.
- Logic errors - errors in behavior of program, how code is formulated, etc. Compilers will not pick up on these, aka “bugs”.
- Runtime - errors that prevent programs from running, fixed by debugging.

Decimal & Binary:

- Binary values can be found by calculating 2 to the power of something.
- 0101 = 0 x 2^3 + 1 x 2^2 + 0 x 2^1 + 1 x 2^0 = 0 + 4 + 0 +1 = 5 (decimal)
- Bit = binary digit
- Byte = 8 bits
- 00110101 = 0 x 2^7 + 0 x 2^6 + 1 x 2^5 + 1 x 2^4 + 0 x 2^3 + 1 x 2^2 + 0 x 2^1 + 1 x 2^0 = 0 + 0 + 32 + 16 + 0 + 4 + 0 + 1 = 53 (decimal)

32-Bit vs 64-Bit Machines:

- x86 = 32-bit
- X64 = 64-bit

Java operators:

- + - addition
- - - subtraction
- * - multiplication
- / - division
- % - modulus (returns division remainder)
- ++ - increment (increases value by 1)
- -- - decrement (decreases value by 1)
- = - assignment
- += - adds value to variable
- -= - subtracts value from variable
- *= - multiples from and variable
- /= - divides variable by value
- %= - divides variable with remainder
- == - equal to comparison
- != - not equal to
- > - greater than
- < - less than
- >= - greater than or equal to
- <= - less than or equal to
- && - and, returns true if both statements are true
- || - or, returns true if one of the statements is true
- ! - reverse the result, returns false is the result is true

Null vs Void:
- Null - object references not defined and points to a special value of “nothing”. Used to see if an object was created and actually exists.
- Void - empty or no value/type. Used as the return type for a method when “nothing” is being returned.

Enumerated Types:
- Declared like a class using enum instead of class to introduce a type name.
- Used to define a list of variable names denoting the set of values belonging to this type: Alternative to static int constants, when the constants values would be arbitrary.
- Names are defined in caps.

Wrapper classes:
- Primitive types are not object types

- Primitive type values must be wrapped in objects to be stored in a collection
- int - Integer
- float - Float
- char - Character

String and its methods:

- Boolean contains (char c)
- Boolean endsWith (String s)
- Int indexOf (String s)
- Int indexIf (String s, int i)
- String substring (int b)
- String substring (int b, int e)
- String toUpperCase()
- String trim()

Immutable String:

- String method: String toUpperCase()
- Incorrect use: input.toUpperCase();
- Correct use: input = input.toUpperCase();
- if(input.toUpperCase().contains())

String.split

- String[] split(String regex) - splits the string around matches of the given regular expression
- String[] wordArray = inputLine.split(" "); - splits inputLine around the regular expression of " ".

Regular Expressions:

- " " - space
- "\t" - tab
- "\\s" - any white space
- "[\t]" - space or tab (grouping)
- "[\t]+" - space or tab (one or more)

ArrayList:

- The ArrayList class implements list functionality with methods for the following operations: add(item), remove(item), get(index), size()
- Using integers to index collections has a general utility: next - index + 1, previous - index -1, last - list.size() -1, the first three - items at indices 0,1,2

ArrayList vs. HashSet:

Similarities:

- Contain a collection of objects
- Add objects (.add method)
- Remove objects (.remove method)
- Number of elements (.size method)
- Iterator ability (.iterator method)

Differences:

- HashSet objects are unique, while an ArrayList can have duplicate objects
- HashSet objects are not ordered, while ArrayList objects are ordered

Maps:

- Maps are flexible-sized collections that contain pairs of values: pair is a key object and a value object
- Uses the key to easily lookup the value, instead of using an integer index
- For example, a telephone book: name and phone number pair
- Reverse-lookup of key using value, not so easy
- Methods .put and .get

Iteration:

For-each

Pros:

- Easy to use
- Access to all items one-by-one
- Ability to change state of the item
- Terminates automatically
- Selective filter using *if-else* statements
- Actions in body may be complicated with multiple lines
- Use on ANY type of collection
- Abstraction from details of how handling occurs

Cons:

- No index provided
- Can NOT stop during looping
- Definite iteration of ALL items
- Can NOT remove or add elements during loop
- Use for collections only
- Access must be to ALL items in sequence [0 to size-1]

While loop:

- A while loop allows the use of a boolean condition to decide whether or not to keep going
- Can be the equivalent of a for-each:

```
public void listAllFiles()
{
```

```
                for(String filename : files) {
                   System.out.println(filename);
                }
           }

           public void listAllFiles()
           {
                int index = 0;
                while(index < files.size()) {
                String filename = files.get(index);
                   System.out.println(filename);
                index++;
                }
           }
```

Pros:

- Can stop at any time during looping
- Indefinite iteration of SOME items using loop condition
- May change collection during loop
- Use explicit index variable inside and outside of loop
- Index variable records location of item at all times

Cons:

- More effort to code
- Requires index looping variable declaration
- Maintain looping variable and manually increment
- Correctly determine loop condition for termination
- Must .get item using index to access the item
- NOT guaranteed to stop with possible infinite loop

Iterator type

- Uses a while loop and iterator object
- Iterator and iterator() are two different things. Collections have an iterator() method.
- Iterator has three methods: boolean hasNext(), next(), and void remove().

Ways to iterate over a collection:

- For-each loop (definite iteration): process every element w/o removing an element
- While loop (indefinite iteration): use for repetition that doesn't involve a collection
- Iterator object (indefinite iteration): use if desired to stop part way through, use to remove from a collection.
- For loop (definite iteration): use to iterate a fixed number of times, with a variable that changes a fixed amount on each iteration.

     for (initialization; condition; post-body action)

```
        {
                Statements to be repeated
        }
```

Constants

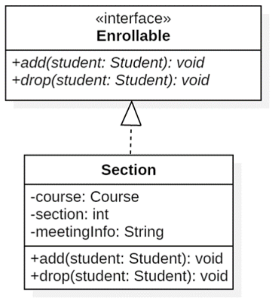
- A variable can have its value fixed
- Final fields must be set in their declaration or constructor
- Combining static and final is common
- Static: class variable
- Final: constant

OOPDA Review

Interfaces:

- An interface is a mechanism for spelling out a contract between two parties: the supplier of a service and the classes that want their objects to be usable with the service.

Section Tally example


«interface»
Enrollable
+add(student: Student): void
+drop(student: Student): void
Section
-course: Course
-section: int
-meetingInfo: String
+add(student: Student): void
+drop(student: Student): void

Sections are enrollable, so we could declare an interface to enforce behaviors of adding and dropping

```
public interface Enrollable {
    public void add(Student student);
    public void drop(Student student);
}
```

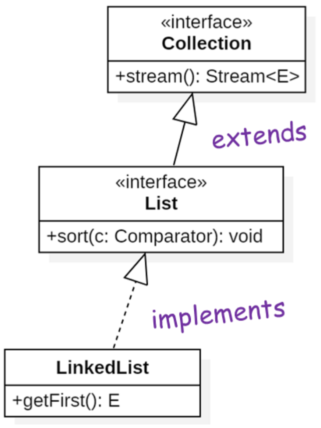
- Notice that these methods are headers only; there is no method body.
- We call such methods abstract.
- Then classes who need such behavior, like Section, could implement this interface.

```
public class Section implements Enrollable {
}
```

-

Inheritance:

- Inheritance allows for data across classes to be shared
- Typically used for classes that are very similar or identical, with slight differences.
- Interfaces can be inherited.

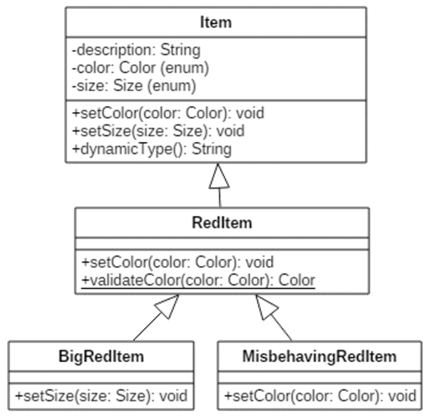


-
- In this case the interface List inherits the method stream() from the interface Collection
- Post and MessagePost example:

```
public class Post  {
      private String username;
      private long timestamp;
      private int likes;
      private ArrayList<String> comments;
      // constructor
      public Post(String author) {
      this.username = author;
      this.timestamp = LocalDateTime.now();
      this.likes = 0;
      this.comments = new ArrayList<String>();
      }.
}
public class MessagePost extends Post  {
      private String message;
      // constructor
      public MessagePost(String author, String text) {
            super(author);
            this.message = text;
      }
}
¡
public class PhotoPost extends Post  {
      private String filename;
      private String caption;
      // constructor
   public PhotoPost(String author, String filename,
                           String caption) {
            super(author);
            this. filename = filename;
            this.caption = caption;
      }
}
```

- Constructors are not inherited.
- Subclass constructors must always contain a 'super' call
- Must be the first statement in the subclass constructor.
- A call to super() will be created in compilation even if the subclass doesn't ask for it.
- You can always invoke any method in the superclass by calling super.methodName(arguments)
- Note: when invoking interface methods, you need to specify the interface
- Enrollable.**super**.add(student);
- You do not need to include the name of the superclass when calling one of its methods. A class only has one superclass.

Polymorphism: allows one to perform the same action in different ways. It's the ability of a class to provide different implementations of a method.


Item
-description: String
-color: Color (enum)
-size: Size (enum)
+setColor(color: Color): void
+setSize(size: Size): void
+dynamicType(): String
RedItem
+setColor(color: Color): void
+validateColor(color: Color): Color
BigRedItem
+setSize(size: Size): void
MisbehavingRedItem
+setColor(color: Color): void

- Has two enum definitions: one for Color and one for Size

- Very standard setter, which takes an enum value for Color and sets
- Item's instance variable.

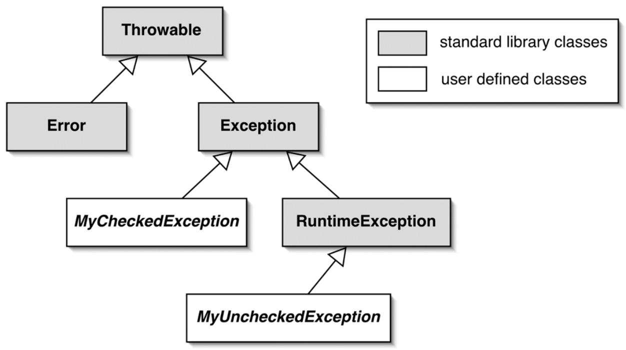
RedItem class

- Has a static method called validateColor() to make sure that a RedItem is in fact red.
- Setter and Constructor call validateColor()

```
public static Color
  validateColor(Color color) {
  if (color.equals(Color.CRIMSON) ||
     color.equals(Color.RED) ||
     color.equals(Color.MAROON) ||
     color.equals(Color.TOMATO)) {
     return color;
  }
  else {
     return Color.RED;
  }}
```

Exceptions:

- When something unexpected happens, an exception is thrown.
- “Throws” passes an exception up the line.
- Public static void main (String[] argos) throws IOException
- You only need to include a throws clause on a method if the method throws a checked exception. If the method throws a runtime exception then there is no need to do so.



-
- Can create new custom exceptions by extending standard exception classes

Common checked exceptions:

- ClassNotFoundException
- IOException
- EOFException
- FileNotFoundException
- NoSuchMethodException

Common unchecked exceptions:

- ArithmeticException
- IllegalArgumentException
- NumberFormatException
- IndexOutOfBoundsException
- ArrayIndexOutOfBoundsException
- StringIndexOutOfBoundsException
- NullPointerException
- InputMismatchException

Try/Catch syntax:

- try {statements}
- [**catch (**MostSpecificExceptionType e**) {**statements**}**] ...
- [**catch (**LeastSpecificExceptionType e**) {**statements**}**]
- [finally {statements}]

Primitive Data Types:

- Int, byte, short, long, float, double, boolean, char
- Int gear = 1;
- The int data type can store whole numbers from **-2147483648 to 2147483647.** In Java SE 8 and later, you can use the int data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of 2^{32}-1.
- Boolean result = true;
- Byte b = 100;
- Short s = 10000;
- Byte - 8-bit
- Short - 16-bit
- Int - 32-bit

- Long - 64-bit
- Float - 32-bit
- Double - 64-bit
- Boolean - true and false
- Char - 16-bit

Type Casting:

- Widening Casting (automatically) - converting a smaller type to a larger type size

  byte -> short -> char -> int -> long -> float -> double

- Narrowing Casting (manually) - converting a larger type to a smaller size type
  double -> float -> long -> int -> char -> short -> byte
- int myInt = 9;
-     double myDouble = myInt; // Automatic casting: int to double
-
-     System.out.println(myInt);   // Outputs 9
-     System.out.println(myDouble);   // Outputs 9.0

Arrays:

- Arrays in java can hold primitive data types (Integer, Character, Float, Byte, Short, Long, etc.) and non-primitive data types of (Object). The values of primitive data types are stored in a memory location where objects are stored in heap memory.
- Arrays cannot be resized dynamically in Java.

Incrementing:

- Post-increment - Value is first used for computing the result, then incremented: x++
- Pre-increment - Value is incremented first and then the result is computed ++x

Test Program:

- The first two evaluate to true because they are the same String. The difference is that the first line instantiates the String in one line, whereas the second does it in two lines
- S3 != s5 because s5 adds “blah” after the statement is evaluated so it still equals “I guess…” without “blah”.
- s6 != s7 because It creates two objects (in String pool and in heap) and one reference variable where the variables ‘s6/s7’ will refer to the object in the heap.
- The next two evaluate to true because the methods check for an exact match, and the strings tested are indeed matches.
- S1 != s8 because s8 has a capital “B” and although the Strings say the same thing, the compiler doesn’t know that. “B” != “b”.

Good Code Design Principles:

Grandma/Grandpa Test:

- Making code readable enough for a non-tech savvy person to interpret. Model classes after real world objects.

Encapsulation:

- The process of enclosing programming elements into larger, more abstract entities.

Open/Closed Principle:

- Classes, methods or functions should be Open for extension (new functionality) and Closed for modification.

DRY (Don’t repeat yourself):

- DRY (don't repeat yourself) means don't write duplicate code, instead use abstraction to abstract common things in one place.
- If a hardcoded value is used more than once, consider making it a public final constant.
- If you have a block of code in more than two places consider making it a separate method.

Single Responsibility Principle (SRP):

- Every class should have a single responsibility and that responsibility should be entirely encapsulated in the class.

Java Collections:

ArrayList: Can hold duplicate values and can be iterated in the following ways:

1. direct index access in basic loop (part of the initial language design)
2. iterators (Iterator/ListIterator) (Java 2 and later)
3. enhanced (foreach) loop
4. forEach method + lambda expression
5. forEach method + method reference

Examples:

```
/**
     * Process ArrayList via direct index access
     */
    public void displayDirectIndexAccess() {
        for (int index = 0; index < data.size(); index++) {
            System.out.println(data.get(index));
        }
    }

    /**
     * <
     * Process ArrayList via an iterator
     */
    public void displayIterators() {
        Iterator<Character> iterator = data.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next().toString());
        }
    }

    /**
     * Process ArrayList via For Each loop
     */
    public void displayForEach() {
        for (char Character : data) {
            System.out.println(data.get(Character));
        }
    }

    /**
     * Process ArrayList via forEach method + lambda expression
     */
    public void displayForEachLambda() {
        data.forEach(c -> System.out.println(c));
    }

    /**
     * Process ArrayList via forEach method + method reference
     */
    public void displayForEachMethodRef() {
        data.forEach(System.out::println);
    }
```

Advanced methods for arrayList iteration:

```
// all reverse methods below

    /**
     * Process ArrayList in reverse with listIterator
     */
    public void reverseWithIterator() {
        ListIterator iterator = data.listIterator(data.size());
        while (iterator.hasPrevious()) {
            // previous method returns previous element
            System.out.println(iterator.previous().toString());
        }
    }

    /**
     * Process ArrayList in reverse with for loop
     */
    public void reverseWithForLoop() {
        for (int index = data.size() - 1; index >= 0; index--) //
decrement
        {
            System.out.println(data.get(index));
        }
```

```java
    }

    // third index method(s) below

    /**
     * Process ArrayList by every 3rd index
     */
    public void displayEveryThirdItemForLoop() {
        for (int index = 0; index < data.size(); index += 3) {
            System.out.println(data.get(index));
        }
    }

    // all palindrome method(s) below

    /**
     * Process ArrayList to check if it's a palindrome
     */
    boolean testIfPalindrome() {
        for (int index = 0; index < data.size() / 2; index++) {
            if (data.get(index) != data.get((data.size() - index) - 1)) {
                System.out.println("Given list is NOT a palindrome");
                return false;
            }
        }
        System.out.println("Given list IS a palindrome");
        return true;
    }
```

HashMap: Can hold duplicate values and can be iterated in the following ways:

1. entrySet() + iterators (Iterator/ListIterator)
2. keySet() + iterators (Iterator/ListIterator)
3. forEach method + lambda expression
4. forEach method + method reference
5. Using Stream

Examples:

```
HashMap<String, String> map = new HashMap<String, String>();

for(map.Entry<String, String> set : map.entrySet())
```

```
    {
         System.out. println(set.getKey() + " = " +
    set.getValue());
    }

    map.forEach((key, value)
         -> System.out. println(key + " = " + value));

    Iterator<Entry<Integer, String>> iterator_map =
    intType.entrySet().iterator();
    while(iterator_map.hasNext())
    {
              map.Entry<Integer, String> new_Map =
    (map.Entry<Integer, String>) iterator_map.next();
    }
         System.out.println(new_map.getKey() + " = " +
new_map.getValue());

    map.entrySet().stream().forEach()input
    -> System.out. println(input.getValue()));
```

HashSet:
Can NOT hold duplicate values and can be iterated in the following ways:

1. iterators (Iterator/ListIterator) (Java 2 and later)
3. For loop
4. For-each method

Examples:

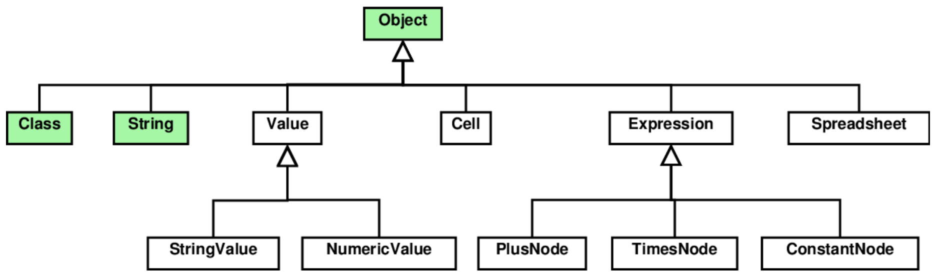
```
    HashSet<String> set = new HashSet<String>();

    Iterator<String> i = set.iterator();
    while(i.hasNext())
    {
         System.out.println(i.next());
    }
    For (String s ; set)
    {
         System.out.println(i);
    }

    set.forEach(i -> System.out. println(i));
```

Parameter Passing:
The swap method does not work for primitive data types that are not of the same type meaning strings can swap with each other, as can ints whereas a double and a string cannot unless further code is added to wrap or parse the unmatching value.

Inheritance and Polymorphism Cont'd.:
Java Inheritance Mechanism:

- Where one object acquires all the properties of a parent object.

Java Inheritance Tree:



Static vs Dynamic:

- Static type of a variable is the type declared in the source code in the declaration statement. Refers to the execution of a program where the type of object is determined at compile time.
- Dynamic type of a variable is the type of the object that is stored in the declaration statement. Refers to the execution of a program where the type of object is determined at runtime.

```
private static Course oopda = new Course("OOPDA", "This course teaches
advanced Java.");
private static Section oopda01 = new Section(oopda, 1, "MW 17:00
Robinson 325");
private static Enrollable oopda02 = new Section(oopda, 1, "MW 12:30
Robinson 312");
       private Post myNewPost = new PhotoPost("Kevin Brown", "pet.png",
"My dog Fido");
    private Item corvette = new BigRedItem("1967 Red Corvette");
```

The static type is how a variable is declared and what the compiler thinks it is.

The dynamic type is what the object really was created as and how it is stored in memory.

Instanceof:

Instanceof is a keyword. It checks if an object reference is an instance of a type, and returns a boolean value.

Class.getName() returns the full name (package + class name) of the object, as a String.

Class.getSimpleName() returns only the class name as a String.

```
Item item2 = items.get(2);
System.out.println(item2 instanceof Item);
System.out.println(item2 instanceof RedItem);
System.out.println(item2 instanceof BigRedItem);
System.out.println(item2.getClass().getName());
System.out.println(item2.getClass().getSimpleName());
```

Drivers:

```
class Lab1P3Driver {

    ArrayList<Character> data = new ArrayList<Character>(); // ArrayList
for Character(s)

    static BufferedReader stdin = new BufferedReader(new
InputStreamReader(System.in));



    public static void main(String args[]) throws IOException {
```

```java
        Lab1P3Driver charProgram = new Lab1P3Driver();
        int input = 0;
        while (input != 6) {
            System.out.print("What would you like to do? \n"
                    + "1. Add a character \n"
                    + "2. Display a list (in order entered) \n"
                    + "3. Display a list (reversed) \n"
                    + "4. Display every 3rd element of list \n"
                    + "5. Test if list is a palindrome \n"
                    + "6. Quit. \n");
            input = Integer.parseInt(stdin.readLine());
            System.out.println("You chose: " + input + "\n");

            // possible cases for initial input
            switch (input) {
                case 1:
                    System.out.print("Enter the character you would like
to add to the list: ");
                    char c = stdin.readLine().charAt(0);
                    System.out.println(c);
                    charProgram.add(c);
                    break;

                case 2:
                    int second_choice = 0;

                    System.out.print("How would you like to display a
list? (in order) \n"

                            + "1. Direct Index Access \n"

                            + "2. With An Iterator \n"

                            + "3. For-Each Loop \n"

                            + "4. For-Each w/ Lambda \n"

                            + "5. For-Each w/ Method Reference \n");



                    // possible cases for second input

                    second_choice = Integer.parseInt(stdin.readLine());

                    switch (second_choice) {

                        case 1:

                            charProgram.displayDirectIndexAccess();

                            break;



                        case 2:

                            charProgram.displayIterators();

                            break;



                        case 3:

                            charProgram.displayForEach();

                            break;
                        case 4:

                            charProgram.displayForEachLambda();

                            break;


                        case 5:

                            charProgram.displayForEachMethodRef();

                            break;


                    }

                    break;


                // possible cases for third input

                case 3:

                    int third_choice = 0;

                    System.out.print("How would you like to display a
list? (in reverse) \n"

                            + "1. Iterator/ListIterator \n"

                            + "2. For-Loop \n");

                    third_choice = Integer.parseInt(stdin.readLine());

                    switch (third_choice) {

                        case 1:

                            charProgram.reverseWithIterator();

                            break;
                        case 2:
                            charProgram.reverseWithForLoop();
                            break;
                    }
                    break;

                case 4:
                    charProgram.displayEveryThirdItemForLoop();
                    break;

                case 5:
                    charProgram.testIfPalindrome();
                    break;
            }
        }
    }
```

Final and Static:

- The importance of the “final” keyword is that it literally means the value tied to the keyword can NEVER be changed/adjusted, it’s final. It can be used to prevent code duplication in a case where you will need to evaluate or use the same variable a number of times.

- The importance of the keyword “static” when attached to a variable is that a single copy of the variable is created and is shared by all objects at a class level.

Interfaces:

- Interfaces are abstract types used to describe the behavior of a class. An interface can be a sort of blueprint for what tasks a class will perform. Interfaces are key in improving abstraction and efficiency.

Further Important Topics:

toString():

- The toString() method is important in Java as it allows you to represent any object as a string. It can further be manipulated by the programmer to do exactly as entailed above along with certain things a programmer may desire for it to do.

  Say we have an object Person (int age, String name, String city)

  ```
  Public String toString()

  {

       Return age + “ years:“ + name + “ from: “ + city;

  }
  ```

Comparable interface:

- Used to compare an object of the same class with an instance of said class.

  Comparable is implemented by a variety of classes with the most notable being:

- Boolean, Byte, Calendar, Character, Date, Double, Enum, File, LocalTime/LocalDate, Short, String, Year, and more.

```java
import java.io.*;

import java.util.*;


class Pair implements Comparable<Pair> {

    String x;

    int y;


    public Pair(String x, int y)

    {

        this.x = x;

        this.y = y;

    }

    public String toString()

    {

        return "(" + x + "," + y + ")";

    }

      @Override public int compareTo(Pair a)

    {

        // if the string are not equal

        if (this.x.compareTo(a.x) != 0) {

            return this.x.compareTo(a.x);

        }

        else {

            // we compare int values
            // if the strings are equal

            return this.y - a.y;

        }

    }

}
```

Print Output For Numeric Data:

- The java.io package includes a PrintStream class that has two formatting methods that you can use to replace print and println. These methods, format and printf, are equivalent to one another. The familiar System.out that you have been using happens to be a PrintStream object, so you can invoke PrintStream methods on System.out. Thus, you can use format or printf anywhere in your code where you have previously been using print or println. For example,

  ```
  System.out.format(.....);
  ```

The syntax for these two java.io.PrintStream methods is the same:

```
public PrintStream format(String format, Object... args)
Ex:
System.out.format("The value of " + "the float variable is " +
     "%f, while the value of the " + "integer variable is %d, " +
     "and the string is %s", floatVar, intVar, stringVar);
```

String Builder:

- Used to create a mutable sequence of characters, an alternative to the String class.

```
     StringBuilder builder = new StringBuilder();
     builder.append(“The apple ”);
     StringBuilder builder2 = new StringBuilder();
     builder2.append(“ doesn’t fall too far ”);
     StringBuilder builder3 = new StringBuilder();
     builder3.append(“ from the apple tree.”);
     System.out.println(builder.toString() +
builder2.toString() + builder3.toString());
Output:
```

The apple doesn’t fall too far from the apple tree.

Generics:

- Generics are essentially parameterized types, meaning generics follow types (likeString) to be a parameter to methods/classes/interfaces. Generic methods contain parameters and return a value after performing a task.

```
public class GenericMethodTest {
   public static < E > void printArray( E[] inputArray ) {
      for(E element : inputArray) {
         System.out.printf("%s ", element);
      }
      System.out.println();
   }

   public static void main(String args[]) {
      Integer[] intArray = { 1, 2, 3, 4, 5 };
      System.out.println("Array integerArray contains:");
      printArray(intArray);   // pass an Integer array
Output:
Array integerArray contains:
1 2 3 4 5
```

Boxing (Autoboxing and Unboxing):

- Converting a primitive data type to an object is called boxing.

  Integer integer = new Integer (“2023”);
  The opposite, converting an object into a primitive data type is called “unboxing”.

Primitive data type Wrappers:

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

char	Character
boolean	Boolean

```
Int num = 100;
Integer obj = Integer.valueOf(num);
```

Varags:

- The last argument of the method may be declared as a variable arity parameter, in which case the method becomes a variable arity method (as opposed to fixed arity methods) or simply varargs method. This allows one to pass a variable number of values, of the declared type, to the method as parameters - including no parameters. These values will be available inside the method as an array.

```
        void printReport(String header, int... numbers) {
    //numbers represents varargs
        System.out.println(header);
        for (int num : numbers) {
            System.out.println(num);
        }
    }

    // Calling varargs method
    printReport("Report data", 74, 83, 25, 96);
```

Further Consideration:

I.

A. y = x + x;

This adds the value of x to x, and assigns said value to y. Essentially making y = x * 2.

B. y = 2 * x;

This is a more readable, less confusing method of doing the above.

C. y = x << 1;

This is known as “shift left logical” and basically moves the variable over by 1 binary OR multiplying it by 2 ^ 1. This is a more confusing way of doing B.

II.

A. Y = x / 2;

This statement divides the value of x by 2, assigning the value to y.

B. Y = x >> 1;

This statement uses "shift right logical" which essentially divides x by 2, and assigns that value to y.

III. y = 1 << x;

IV. y = 2 << x;

V. y = 4 << x;

VI. y = 8 << x;

VII.

```
char c= 'D'; // c initialized to an upper-case letter
int rank = c.charAt(0) - 40 ;    //complete this 1 line of code
System.out.println(rank); //displays 4 if c is 'D', 1 if c is 'a', 26 if c is 'Z'
```

Mathematical Background:

Give the closed form solution for each of the following sums, where n is a strictly positive integer:

a. (sum of integers from 1 to n) 1 + 2 + 3 + …. + n = **Σ =** n(n+1)/2

b. (sum of even integers from 1 to 2n) 2 + 4 + 6 + …. + 2n = **Σ =** 2n(2n+1)/2

c. (sum of odd integers from 1 to 2n-1) 1 + 3 + 5 + …. + 2n-1 = **Σ =** 2n-1(2n-1)/2

Complete the following formulas (where a, b, c, and d are integers >=1)

d. $\log_2$ a + $\log_2$ b + $\log_2$ c = **Σ =** $\log_2$ (n)

e. $x^a * x^b * x^c * x^d$ = **Σ =** x^n

f. $\log_a b$ = **loga(b)=[logc(b)]/[logc(a)]** (change logarithm base from a to c)

Guidelines for Code Development:

1. This guideline is important as it prevents cheating or loopholes and promotes memorization of common coding practice.
2. This guideline promotes more efficient practices as having a function call itself isn't the most efficient practice.
3. We won't be using an IDE anymore so a driver is essential.

4. The practice outlined in this guideline promotes efficiency and better code readability.
5. All I/O being in the driver makes a program less error prone and gives the driver one purpose.
6. Having a toString method in all supporting classes will allow the program to be easily debugged by obvious errors coming through Strings.
7. This is a given, programmers should never have useless lines, methods, parameters, fields, or anything useless in their program as all it will do is take up memory and just be an eyesore.
8. In programming, everything should have its own role and purpose so having data fields that will later become local variables just makes things confusing.
9. A variable local to main is the most efficient way to make a tie between user input and the classes we construct.
10. This falls into the category of "don't repeat yourself" rather than checking list.size() 5 times, just make a variable for it and save time and lines.