CS62 Lab1: Exceptions & Bag of tokens

Java Fundamentals

Lab 1 agenda

- Finishing the File I/O worksheet
- Your first quiz
- Review: exceptions
- The actual lab :) - Bag of Tokens

Finishing up file I/O

Worksheet time!

Hint: pull up the slides to see the writing/reading code snippet examples

- Fill in lines of code in the Java class called FileIOExample
- It will contain a main method that will prompt the user for a String corresponding to a text file in their directory and a number for how many lines of text they want to read from that file.
- It should use these 2 pieces of information to open the file, read the specified number of lines, and write them into a new file called output.txt.
- You can add whatever checks for exceptions you think are appropriate.
- Don't forget to close the input and output streams!

Writing data to a text file

```
import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;

public class WriteData {
    public static void main(String[] args) {

        PrintWriter output = null;
        try {
            output = new PrintWriter(new File("addresses.txt"));
            // Write formatted output to the file
            output.print("Alexandra Papoutsaki ");
            output.println(222);
            output.print(“Jingyi Li ");
            output.println(111);

        } catch (IOException e) {
            System.err.println(e.getMessage());
        } finally {
            if (output != null)
                output.close();
        }
    }
}
```

need to import relevant classes

call .print or .println to write to file

catch IOException for any errors

.close() the I/O stream

https://liveexample.pearsoncmg.com/html/WriteData.html

Reading data from a text file

```
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

public class ReadData {
   public static void main(String[] args) {

      Scanner input = null;
      // Create a Scanner for the file
      try {
         input = new Scanner(new File("addresses.txt"));

         // Read data from a file
         while (input.hasNext()) {
            String firstName = input.next();
            String lastName = input.next();
            int room = input.nextInt();
            System.out.println(firstName + " " + lastName + " " + room);
         }
      } catch (IOException e) {
         System.err.println(e.getMessage());
      } finally {
         if (input != null)
            input.close();
      }
   }
}
```

same try...catch...finally structure

use Scanner class

use a while loop to check if file still has lines

.next() is space separated (if you want the whole line, call .nextLine())

close the file

https://liveexample.pearsoncmg.com/html/ReadData.html

Worksheet answers

```
public class FileIOExample {
    public static void main(String[] args) {
        Scanner inputScanner = new Scanner(System.in);
        System.out.println("Enter the input file name: ");
        String inputFile = inputScanner.nextLine();
        System.out.println("Enter the number of lines to read: ");
        String linesInput = _________;  inputScanner.nextLine()
        int numberOfLines = -1;
        try {
                                          linesInput
            numberOfLines = Integer.parseInt(________);
        } catch (NumberFormatException e) {
            System.err.println("_Invalid # of lines__");
        }
        finally{
                          close()
            inputScanner._______; //close the scanner
        }
        Scanner fileScanner = null;
        PrintWriter writer = null;
```

Worksheet answers

```
Scanner fileScanner = null;
PrintWriter writer = null;
try {                        Scanner(new File(inputFile))
    fileScanner = new ______________________________;
    writer = new ___________("output.txt");
                 PrinterWriter

    int linesRead = 0;
    //while there are still lines and we've read less than the input
                        hasNextLine()
    while (fileScanner.__________ && linesRead < numberOfLines){
        String line = fileScanner.nextLine();
        System.out.println(line);
        writer.println(line);
  linesRead
    _______++; //increment # of lines read
    }
} catch (IOException) e{
    System.err.println("Error reading or writing files: " +
    e.getMessage());
}
finally{
    if(fileScanner!=null){
        fileScanner.close();
    }
    if(writer!=null){
     writer.close()  ;
```

Quiz

Exceptions

Exceptions are exceptional or unwanted events

- They are operations that disrupt the normal flow of the program. E.g.,
  - wrong input, divide a number by zero, run out out of memory, ask for a file that does not exist, etc. E.g.,

```
    int[] myNumbers = {1, 2, 3};

    System.out.println(myNumbers[10]); // error!
```

  - Will print something like

```
  Exception in thread "main"
  java.lang.ArrayIndexOutOfBoundsException: 10
```

  and terminate the program.

https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Exception terminology

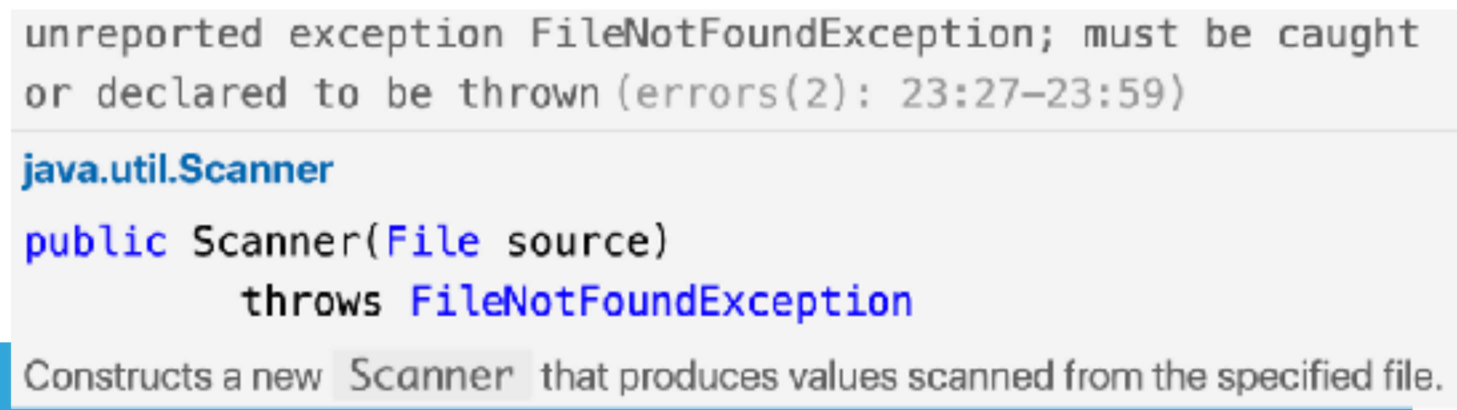
- When an error occurs within a method, the method throws an exception object that contains its name, type, and state of program.

- The runtime system looks for something to handle the exception among the call stack, the list of methods called (in reverse order) by `main` to reach the error.

- The exception handler catches the exception. If no appropriate handler, the program terminates.

https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Three major types of exception classes



```
unreported exception FileNotFoundException; must be caught
or declared to be thrown (errors(2): 23:27–23:59)
java.util.Scanner
public Scanner(File source)
        throws FileNotFoundException
Constructs a new Scanner that produces values scanned from the specified file.
```

The exception in the worksheet was an example of a checked exception: we need to handle the exception or our code won't run!

- Checked Exceptions: Should follow the *Catch or Specify* requirement.
  - errors caused by program and external circumstances and caught during compile time. E.g.,
    - `java.io.FileReader`
- Unchecked Exceptions: Do NOT follow the *Catch or Specify* requirement and are caught during runtime.
  - `Error`: the application cannot recover from. E.g.,
    - `java.lang.StackOverflowError` (for stack)
    - `java.lang.OutOfMemoryError` (for heap)
- `RuntimeException`: internal programming errors that can occur in any Java method and are unexpected. E.g.,
    - `java.lang.IndexOutOfBoundsException`
    - `java.lang.NullPointerException`
    - `java.lang.ArithmeticException`

The exception you'll write in lab will handle a RuntimeException

https://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html

Useful exceptions to know

- Checked - you have to catch or specify they throw an exception
  - `IOException`: when using file I/O stream operations.
- Unchecked - you don't have to catch/specify them, but it can still be a good idea to do so.
  - `ArrayIndexOutOfBoundsException`: when you try to access an array with an invalid index value
  - `ArithmeticException`: when you perform an incorrect arithmetic operation. For example, if you divide any number by zero.
  - `IllegalArgumentException`: when an inappropriate or incorrect argument is passed to a method.
  - `NullPointerException`: when you try to access an object with the help of a reference variable whose current value is `null`.
  - `NumberFormatException`: when you pass a string to a method that cannot convert it to a number. e.g., Integer.parseInt("hello")

https://stackify.com/types-of-exceptions-java/

The Catch or Specify requirement

- Code that might throw checked exceptions must be enclosed either by
  - a try-catch statement that catches the exception,

```
  try {
          //one or more legal lines of code that could throw an
exception
  } catch (TypeOfException e) {
          System.err.println(e.getMessage());
  }
```

  - or have the method specify that it can throw the exception. The method must provide a throws clause that lists the exception.

```
method() throws Exception{
    if(some error) {
         throw new Exception();
    }
}
```

https://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html

Catching exceptions

```
method(){
    try {
         statements; //statements that could throw exception
    } catch (Exception1 e1) {
            //handle e1;
    }
    catch (Exception2 e2) {
            //handle e2;
    }
}
```

- If no exception is thrown, then the catch blocks are skipped.
- If an exception is thrown, the execution of the try block ends at the responsible statement.
- The order of catch blocks is important. A compile error will result if a catch block for a more general type of error appears before a more specific one, e.g., **Exception should be after ArithmeticException.**

https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.4.6

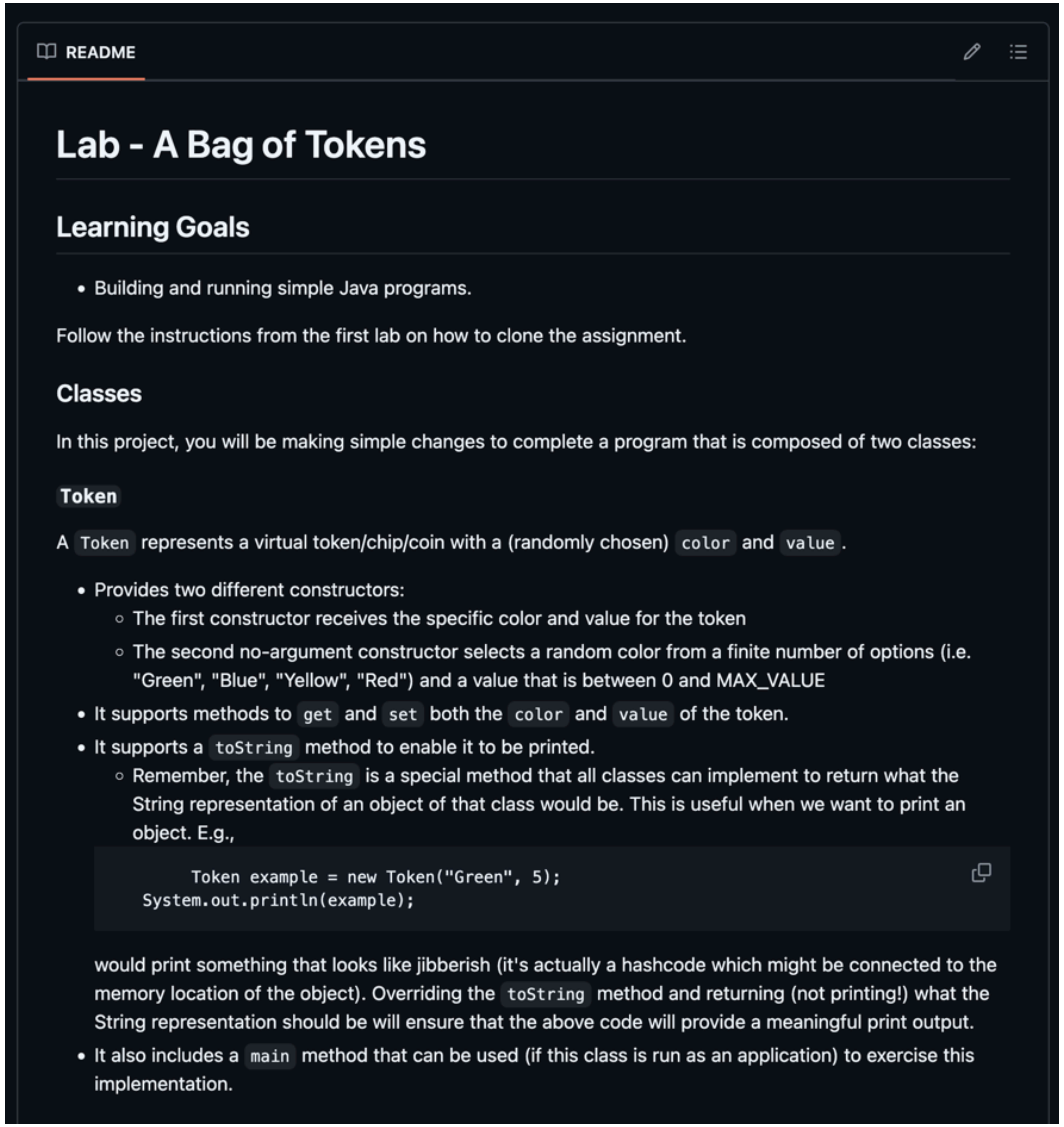
finally block

- Used when you want to execute some code regardless of whether an exception occurs or is caught

```
method(){
    try {
          statements; //statements that could thrown exception
    } catch (Exception1 e) {
            //handle e; catch is optional.
    }
    finally {
            //statements that are executed no matter what;
    }
}
```

- **The `finally` block will execute no matter what. Even after a `return`.**

https://docs.oracle.com/javase/tutorial/essential/exceptions/finally.html



README

Lab - A Bag of Tokens

Learning Goals

- Building and running simple Java programs.

Follow the instructions from the first lab on how to clone the assignment.

Classes

In this project, you will be making simple changes to complete a program that is composed of two classes:

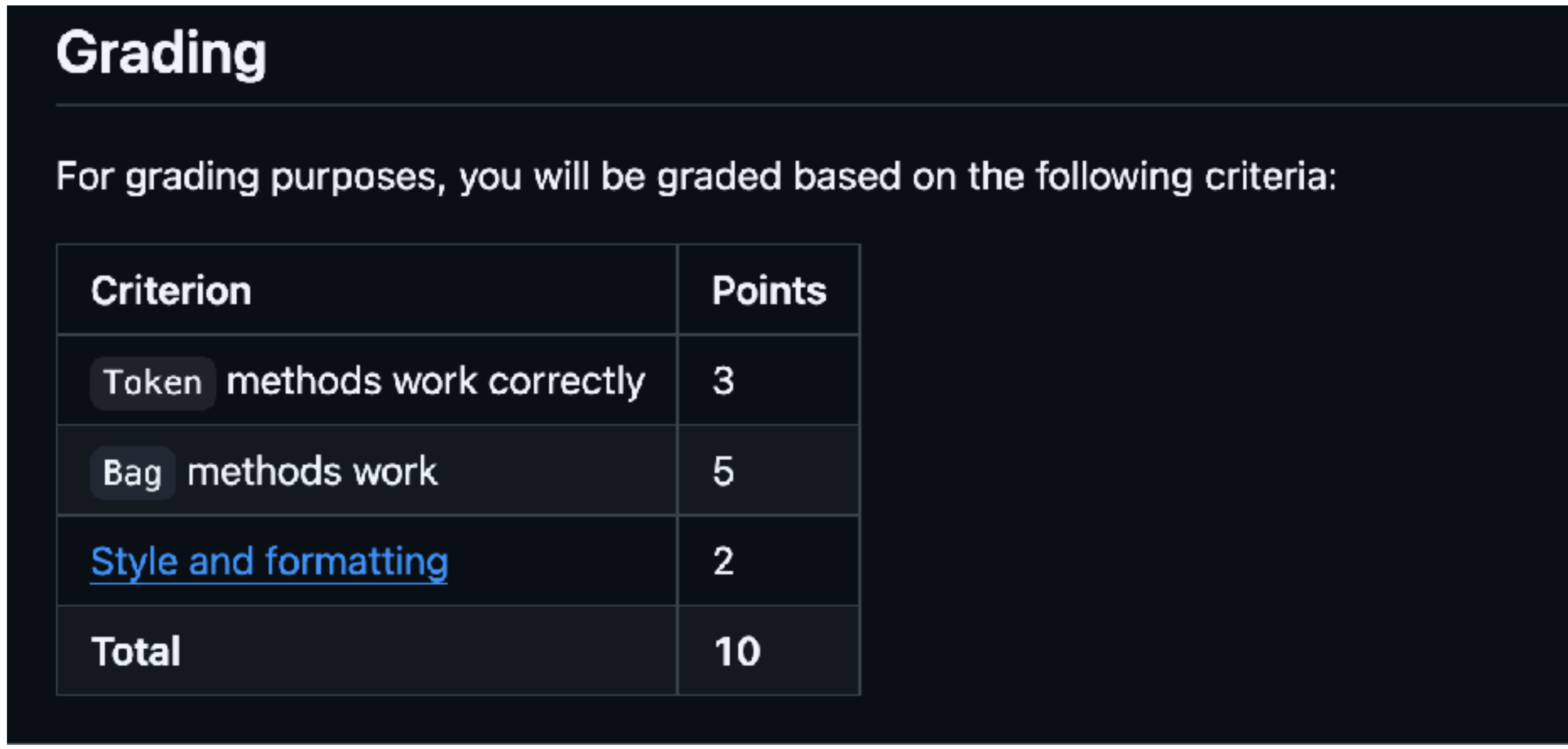
`Token`

A `Token` represents a virtual token/chip/coin with a (randomly chosen) `color` and `value`.

- Provides two different constructors:
  - The first constructor receives the specific color and value for the token
  - The second no-argument constructor selects a random color from a finite number of options (i.e. "Green", "Blue", "Yellow", "Red") and a value that is between 0 and MAX_VALUE
- It supports methods to `get` and `set` both the `color` and `value` of the token.
- It supports a `toString` method to enable it to be printed.
  - Remember, the `toString` is a special method that all classes can implement to return what the String representation of an object of that class would be. This is useful when we want to print an object. E.g.,

  ```
      Token example = new Token("Green", 5);
  System.out.println(example);
  ```

  would print something that looks like jibberish (it's actually a hashcode which might be connected to the memory location of the object). Overriding the `toString` method and returning (not printing!) what the String representation should be will ensure that the above code will provide a meaningful print output.
- It also includes a `main` method that can be used (if this class is run as an application) to exercise this implementation.



Grading

For grading purposes, you will be graded based on the following criteria:

Criterion	Points
`Token` methods work correctly	3
`Bag` methods work	5
Style and formatting	2
Total	**10**

Find Prof. Li or a TA to get checked off for style points on Gradescope!