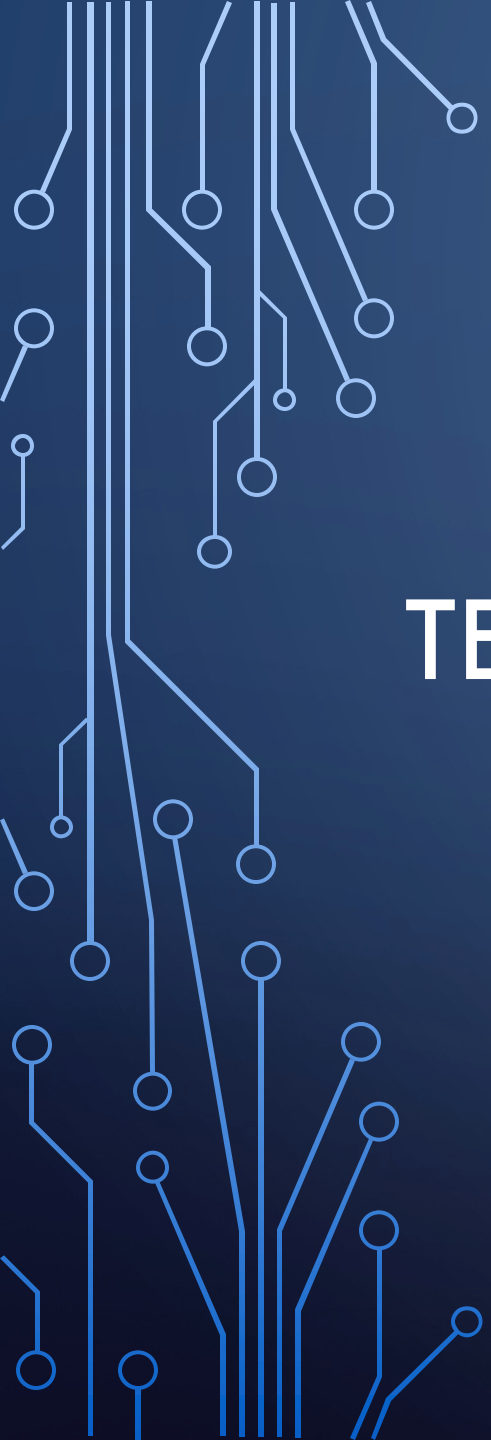

TESTING FILE INPUT AND OUTPUT

CSC Software Testing Materials (Testing: File Input and Output)

@ Dr. Jessica Young Schmidt and NCSU Computer Science Faculty

Input

The Paycheck program prompts the user input filename and output filename. If the filenames are invalid, the user will be reprompted.

for information about each Employee, including the name, level (1, 2, or 3), hours worked, retirement percent, and whether he or she has medical, dental, and vision insurances. This version of the program has extensive error checking and loops to allow for processing more than one paycheck at a time. The user is prompted for the number of paychecks and stores paycheck information.

Each line is structured in a tab delimited manner:

```
name<TAB>level<TAB>hours worked<TAB>medical<TAB>dental<TAB>vision<TAB>retirement %
```

Such that medical, dental, and vision are boolean values that represent whether the employee has the given type of insurance.

Output

The following information is printed to output file about each employee's pay check:

1. employee's name
2. hours worked for a week
3. hourly pay rate
4. regular pay for up to 40 hours worked
5. overtime pay (1.5 pay rate) for hours over 40 worked
6. gross pay (regular + overtime)
7. total deductions
8. net pay (gross pay – total deductions).

If the net pay is negative, meaning the deductions exceeds the gross pay, then an error is printed.

System Testing

For System Testing with file input and file output, we create test input files along with expected output files. We test valid and invalid input.

The test files should be in the `test-tiles` directory. There are input files and expected output files. The input files contain valid or invalid values and the expected output files record the expected results of the execution. You can compare the actual results of execution with the expected results visually, or the `diff` command may be used to compare the actual results and expected results. If there are any differences, they will be printed to the console. If there are no differences, the prompt will appear and there will be no output.

➤TERMINAL

```
$ java -cp bin Paychecks
Input File: test-files/input_level01.txt
Output File: test-files/output_act_level01.txt
$ diff test-files/output_exp_level01.txt test-files/output_act_level01.txt
$
```

Unit and Integration Testing

If a method returns a non-null object, such as `Scanner` or `PrintWriter`, we would want to use the `assertNotNull()` method.

Testing that Objects Are Not Null

The JUnit library includes many different types of assert methods. We discussed many method in the Unit and Integration Testing section. If a method returns a non-null object, such as `Scanner` or `PrintWriter`, we would want to use the `assertNotNull()` method.

Method	Description
`assertNotNull(Object actual, String message)`	Assert that `actual` is not `null`. Fails with the supplied failure message.

Paychecks.java:

➤JAVA

```java
    /**
     * Processes input file of paychecks.
     *
     * @param fileScanner Scanner to read file with paycheck information
     * @param printWriter PrintWriter to write paycheck information to
     */
    public static void processFile(Scanner fileScanner,
            PrintWriter printWriter) {
        while (fileScanner.hasNextLine()) {
            String line = fileScanner.nextLine();
            try {
                printWriter.println(processLine(line));
            } catch (IllegalArgumentException e) {
                System.out.println("Error reading line: " + line);
            }
        }
    }
```

PaychecksTest.java:

➤JAVA

```java
/**
 * Testing contents of scanner
 *
 * @param expected expected scanner
 * @param actual actual scanner
 * @param message message for test
 */
public void testFileContents(Scanner expected, Scanner actual,
        String message) {
    int line = 0;
    while (expected.hasNextLine()) {
        line++;
        if (actual.hasNextLine()) {
            assertEquals(expected.nextLine(), actual.nextLine(),
                    message + ": Testing line " + line);
        } else {
            fail(message + ": Too few lines: line " + line);
        }
    }
    if (actual.hasNextLine()) {
        fail(message + ": Too many lines");
    }
}
```

```java

/**
 * Test the Paychecks.processFile() method.
 *
 * @throws FileNotFoundException if file stream can not be constructed
 */
@Test
public void testProcessFile() throws FileNotFoundException {

    // TEST 1
    String message = "Testing file of multiple invalid";
    String inputFile = "test-files/input_MULTIPLE_INVALID.txt";
    String expectedFile = "test-files/UNIT-output_exp_MULTIPLE_INVALID.txt";

    String outputFile = "test-files/UNIT-output_act_MULTIPLE_INVALID.txt";

    Scanner in = new Scanner(new FileInputStream(inputFile));
    PrintWriter out = new PrintWriter(new FileOutputStream(outputFile));
    Paychecks.processFile(in, out);
    in.close();
    out.close();

    Scanner actual = new Scanner(new FileInputStream(outputFile));
    Scanner expected = new Scanner(new FileInputStream(expectedFile));
    testFileContents(expected, actual, message);
    expected.close();
    actual.close();
}
```