

211lab07

Lab 07 – Javadoc and More List Exercises

Java documentation style

The Java Development Kit (JDK) comes with a tool called Javadoc. This tool will generate documentation for Java source code with comments written in accordance with the Java documentation style. The following links show more examples. You do need to spend time outside of the lab to study them.

- [javadoc](#)
- [Writing doc comments](#)
- [Comments Conventions](#)

The following is a very short summary of the Javadoc conventions.

- Comments are enclosed in between `/**` and `*/` and must immediately precede the java constructs (e.g. class, field, method) that are to be commented.
- Each method should have a comment that describes the purpose of the method.
- Each method should have a comment for each parameter, if any. The comment for a parameter has the form:
 - `@param parameter-name some-description`
- Each function method should have a comment for the returned object. The comment for the returned object has the form:
 - `@return some-description`
- Each method should have a comment for each exception thrown, if any. The comment for an exception has the form:
 - `@exception exception-type some-description`Here are two files that are documented in javadoc style.

```

/**
 * Represents a person who knows how to compute the number of months till his/her
 * next birthday.
 * @author DXN
 * @since Copyright 2009 by DXN - All rights reserved
 */
class Person {
    /**
     * birth month
     */
    int bm;

    /**
     * Computes the number of month until the next birthday given the current month.
     * @param cm an integer between 1 and 12 as the current month.
     * @return an integer between 0 and 11.
     */
    int nMonthTillBD (int cm) {
        return (bm - cm + 12) % 12;
    }
}

/**
 * JUnit test class to test all methods in class Person.
 * @author DXN
 * @since 02/20/2009
 */
class TestPerson extends TestCase {

    /**
     * Tests nMonthTillBD by covering three cases:
     * the current month is less than the birth month
     * the current month is equal to the birth month
     * the current month is greater than the birth month.
     */
    void test_nMonthTillBD() {

        Person peter = new Person(9); // a person born in September.

        assertEquals("Calling nMonthTillBD(2).", 7, peter.nMonthTillBD(2));
        assertEquals("Calling nMonthTillBD(9).", 0, peter.nMonthTillBD(9));
        assertEquals("Calling nMonthTillBD(12).", 9, peter.nMonthTillBD(12));
    }
}

```

Creating javadoc using DrJava:

DrJava has incorporated the javadoc utility into its IDE. Since javadoc does not know anything about language levels, we will delay running the javadoc utility on our code until we switch to the Full Java language.

For the "daring", you can try the following steps at your own risk!

- Open Drjava and make sure that it is set for Language Level / Elementary.
- Create and save two separate files for the above classes.
- Compile all.
- Select File/Close All to close all djO files.
- Select Language Level / Full Java.
- Select File/Open Folder: open the folder where the Person and TestPerson files were saved. Only java files will be opened.
- Select the menu Edit/Preferences to bring up the Preferences dialog window;
- in the Categories pane, select javadoc;
- in the Access level selection listbox, select public and click OK;
- click the Javadoc button in the main menu toolbar;
- DrJava will prompt you for a location to write out the javadoc for all the files in the package (i.e. directory) and subpackages (i.e. subdirectories) and create all the javadoc files.
- Go to the subdirectory where javadoc files were generated and open the index.html file: you should see your code documentation displayed as html files.

Now change the javadoc Access level in the javadoc Preferences to private and generate javadoc again. What is the difference?

More List Exercises

Recall our object model (common known in the Scheme world as "data definition") for lists of int.

- `IntList` is an abstract list of `int` .
- `EmptyIntList` is an `IntList`
- `ConsIntList(first, rest)` is an `IntList`, where `first` is an `int` and `rest` is an `IntList` .

The above can be implemented in Java using the composite design pattern as follows.

```
/** Abstract list structure. */
abstract class IntList {
}

/* Concrete empty list structure containing nothing. */
class EmptyIntList extends IntList {
}

/** Concrete non-empty list structure containing an int, called first, and a rest,
 * which is a list structure. */
class ConsIntList extends IntList {
    int first;
    IntList rest;
}
```

The above composite design for `IntList` gives rise to the interpreter design pattern for coding list methods. Here is the coding template.

```
abstract class IntList {
    abstract returnType methodName(parameter_list); // returnType may be Void
}

class EmptyIntList extends IntList {
    returnType methodName(parameter_list) {
        // base case code
    }
}

class ConsIntList extends IntList {
    int first;
    IntList rest;
    returnType methodName(parameter_list) {
        // ... first ...
        // ... rest.methodName(parameter_list)...
    }
}
```

A Scheme-like String representation of lists.

Suppose we want to display an empty list as `()` and the list containing 1, 2, 3 as `(1, 2, 3)`. How do we do this? We need to add a method to the `IntList` hierarchy of classes to perform this computation. Let's call this method `listString()` and let's proceed together.

Step 1: Instantiate the interpreter code template given above by replacing `returnType` with `String`, the `methodName` with `listString` and the `parameter_list` with `nothing`. Be sure to create one file for each class. The code will not compile. Why?

Step 1.5: Add syntactically correct code to the template so that the whole thing compile.

Step 2: Write appropriate JUnit test classes. Because of lack of time, we will not do this step in class.

Step 3: Write the code for `listString` in `EmptyIntList`. This is trivial! Make sure it passes the JUnit test though!

Step 4: Write the code for `listString` in `ConsIntList`. You can try the structural recursive code given in the template and see that it will not work. This is because by the time you reach the end of the list (i.e. when `rest` is the empty list), you need to do something different from what `EmptyIntList`. `listString()` is programmed to do. You will need to call on `rest` to perform an auxiliary ("helper") method to get the job done. Let's call this helper method `listStringHelp`.

What does `listStringHelp` need to know? You can pass to it what you (as the current list) know and ask `rest` to complete the job. What does the current list know? It knows it can build the string "(" + `first` (meaning "(" concatenate with the default String representation of `first`) and it can pass this string to the helper method. So in effect the helper takes as parameter the accumulated string representation of the list so far and delegates the job for completing the final string representation to `rest`.

The code for `ConsIntList.listString()` is thus something like:

```
return rest.listStringHelp("(" + first);
```

The code will not compile because we have yet to add the method `listStringHelp` to the `IntList` hierarchy.

Step 5: Add the method `String listStringHelp(String acc)` to the `IntList` hierarchy; add stub template code so that the whole thing compile. Unless you accidentally write the correct code, the JUnit test for `ConsIntList` will not pass still.

Step 6: Write JUnit test code for `listStringHelp`. Again, due to lack of time, we will not do that here in the lab. Actually, more than often, writing the test code will help write the code for the method in question.

Step 7: Write the code for `EmptyIntList.listStringHelp(String acc)` What should the empty do here? It knows it has the accumulated string representation of the whole list so far and that all it needs is the closing parenthesis. So all it has to do is to add the closing parenthesis to the accumulated string and return: `return acc + ")";` (And make sure it passes the JUnit test).

Step 8: Write the code for `ConsIntList.listStringHelp` (and make sure that it passes the JUnit test). What can a non-empty list do here? All it needs to do is to concatenate a comma and its first to the accumulated string representation so far and pass it on to `rest` to complete the job: `return rest.listStringHelp(acc + ", " + first);` (And make sure it passes the JUnit test).

Note that the code for `listStringHelp` is tail-recursive.

Step 9: Run the complete JUnit test suite and the whole thing should pass!

Here is the solution code.

```

abstract class IntList {
    /** Computes a String representation of this list with matching parentheses
     * as in Scheme. For example, the list containing 1, 2 and 3 should return
     * (1, 2, 3) and the empty list should return ().
     * @return a non empty String consisting of elements in this list enclosed
     * in a pair of matching parenthesis, separated by commas.
     */
    abstract String listString();

    /** Accumulator helper method for listString to compute the String
     * required representation of this list given the accumulated
     * String representation of the preceding list.
     * @param acc the accumulated String representation of the list that
     * precedes this list.
     * @return a non empty String consisting of elements in this list enclosed
     * in a pair of matching parenthesis, separated by commas.
     */
    abstract String listStringHelp(String acc);
}

class EmptyIntList extends IntList {

    /** @return "()" */
    String listString() { return "()"; }

    /** @param acc the accumulated String representation of the list that
     * precedes this list. For example "(5, 3"
     * @return a non empty String consisting of elements in this list enclosed
     * in a pair of matching parenthesis, separated by commas. For example,
     * "(5, 3)"
     */
    String listStringHelp(String acc) { return acc + "()"; }
}

class ConsIntList extends IntList {
    int first;
    IntList rest;

    /** Calls on rest to perform the helper method listStringHelp passing it
     * the accumulated String representation so far, which is "(" + first.
     * @return a non empty String consisting of elements in this list enclosed
     * in a pair of matching parenthesis, separated by commas.
     */
    String listString() { return rest.listStringHelp("(" + first); }

    /** @param acc the accumulated String representation of the list that
     * precedes this. For example "(5, 3"
     * @return a non empty String consisting of elements in this list enclosed
     * in a pair of matching parenthesis, separated by commas. For example,
     * "(5, 3)"
     */
    String listStringHelp(String acc) { return rest.listStringHelp(acc + "(" + first); }
}

/** Testing empty lists. */
class TestEmptyIntList extends TestCase {

    void test_listString() {
        EmptyIntList mt = new EmptyIntList();
        assertEquals(mt + ".listString()", "()", mt.listString());
    }
}

```

Lab Exercises

1. Write another version of `listString` that is tail recursive. Call it `listString2`. (Can you think of a non-tail recursive version?)
2. Write a method called `prodNums` that returns the product of the number in the list, using a tail recursive helper method.

3. Revise the preceding definition to terminate immediately if it encounters a 0 in the list of numbers being multiplied.
 4. Write a method called `makePalindrome` that returns a list consisting of the input list and its mirror around the last element, using a (non tail-recursive) helper with an accumulator. For example, `(1, 2, 3).makePalindrome ()` returns the list `(1, 2, 3, 2, 1)`.
 5. Write a method called `reverse` that reverses the list using a tail-recursive helper.
-

Access Permissions: (Please don't edit)

- Set `ALLOWTOPICCHANGE` = `Main.TeachersComp211Group`