10 Java API, Exceptions, and Collections

Activities

1. Familiarize yourself with the Java Application Programmers Interface (API) documentation.
2. Learn the basics of writing comments in Javadoc style.
3. Learn the basics of working with the *ArrayList* and use the *Scanner* class.
4. Define and use exceptions to detect and handle errors.

Warning

Try to get as much as possible during the lab. Ask questions when you do not understand something. **The first part of the next assignment is to complete all tasks in this lab.**

Resources

Download the provided zip file and unzip it. Create a new Eclipse project named Lab11. Add the given code to the project and link the external JAR jpt.jar to the project. You should have the following Java files:

- class *Examples* that is to be used for tests that are not a part of the program that interacts with the user
- class *Interactions* that controls our user interactions - you will add a couple of methods here
- class *Reply* - a skeleton, you have to add the functionality
- class *SampleEliza* the database of answers and some of the methods dealing with the answers to the patient.
- class *Words* - a data for exploring *ArrayList* and *String* manipulations.
- class *IllegalUseOfTraversalException* to illustrate how exception classes are defined.
- additional classes we have seen before: *ISame* interface, *SimpleTestHarness*, *Traversal*, and *TraversalALC*.

- also use *jpt.jar* with the project.

10.1 Activity: Reading JavaDocs

Go to the Java API at *http://java.sun.com/j2se/1.5.0/docs/api/*. Bookmark this page! When coding you will often use classes that are provided for you by Java. The Java API describes these classes and lists all of the fields and methods of these classes that are available to you.

The front page of the Java API lists all of the packages provided by Java. A package is a collection of related interfaces and classes.

Tips For Quickly Finding Class Specifications

The left frame of the API page lists all classes alphabetically. If you want the specifications for a specific class you can click in this frame and use your web browsers search function to find that class. For example, find the *ArrayList* class. Another way to quickly find Java API specifications is to search Google for "java api class x", where x is the name of the class you're searching for. For example, the search "java api class arraylist" returns the specifications for class *ArrayList* as the first result.

The Anatomy of a JavaDoc

All of the specifications are in a JavaDoc format. JavaDocs are automatically generated from source code based on specifically formated comments that the programmer adds for each class and each method. We will look at the format of such comments shortly.

Lets use the *ArrayList* JavaDoc as an example.

The top of the JavaDoc lists the other classes that *ArrayList* extends and implements. In this case, *ArrayList* extends from the classes *Object*, *AbstractCollection*, *AbstractList*, and implements the interfaces *Cloneable*, *Collection*, *List*, *RandomAccess*, *Serializable*.

Next is a general description of the class. In this case, the JavaDoc says that *ArrayList* is a "Resizable-array implementation of the List interface."

Following this is a summary of fields, constructors, and methods provided by *ArrayList*. In general, classes will provide very few public fields and the JavaDoc will contain mostly specifications of methods. Look over some of the methods provided by *ArrayList*.

The method summaries provide headers (return type, name, and arguments) and a short description of the method's functionality. More detailed descriptions are linked from these summaries and appear farther down on the same page.

Generating JavaDoc-s

For detailed description of how to write documentation for the automatic Javadoc generator see *http://java.sun.com/j2se/javadoc/writingdoccomments*: How to Write Doc Comments for the Javadoc.

Look first at the code for the class *Traversal*. Notice the special format of the comments. Notice also that they are shown in a different color than the comments we have seen so far.

When the comments for Java programs are written using this special format, the documentation web pages can be generated automatically — with all the cross-references necessary.

The comment always starts with /∗∗ and usually spans several lines. Each line then starts with a ∗ and the last line has only ∗/ in it. When you start typing such comment in *Eclipse* the color of the comment changes and the ∗ at the beginning of the line is generated automatically. In addition the beginnings of some of the special comment commands are also generated for you.

To see how to write the comments while designing a program, start by adding to the *Examples* class a *stub* of a method *reverse1* with the header below. (A *stub* is a method with a complete header and the body that only produces the correct type of value, but does not perform the desired computation. We use the stubs as place-holders when designing a program just to make sure the program would compile. Later, we design the rest ofthe method.)

```
public <T> ArrayList<T> reverse1(ArrayList<T> alist){
  return alist;
}
```

then start the comment above. You will see that it generates the following *template* for the comment:

```
/**
  *
  * @param <T>
  * @param alist
  * @return
  */
```

We complete the comment as follows:

```
/**
  * Reverse the elements in the given <code>ArrayList</code>
  * using a helper <code>ArrayList</code>
  *
  * @param <T> the datatype for the elements of <code>ArrayList</code>
  * @param alist the original <code>ArrayList</code>
  * @return <code>ArrayList</code> with elements in reverse order
  */
```

In **Project** menu select **Generate Javadoc...** and choose the *doc* folder for the documentation. Choose to make the documentation pages only for the class *Examples*.

When done, look at the pages in a browser. The *index.html* file will be in the folder you have selected.

This is enough for a start — experiment with Javadoc-s for a whole project at home. For the remainder of the semester, always write comments so that we can generate complete documentation from the program sources.

10.2 Activity: Working with the ArrayList

The class *Words* contains some *String*s and *ArrayList*s of *String*s that we will use. Our first task is to reverse the order of the words in the *ArrayList reversed*. Design a method in the *Examples* class that reverses the order of the words in the *ArrayList*.

Do the following three tasks - modifying the previous solution as you go on (or keeping the previous one and adding a new variant:

- First just produce another *ArrayList* with the words reversed. We have already written the header for this method when learning how to generate documentation. Use a *while* loop or a *for* loop with indices. **If you do not know how, aks for help immediately.**
- Next think of how you would reverse the elements in an *ArrayList* without using another *ArrayList* at all. Design the method that will perform this task for an arbitrary *ArrayList*.
- Finally, design the method that will print all words, one to a line, traversing the *ArrayList* using the *for* loop. for now there is no good way to test this method, so just observe that it indeed prints all values.

10.3 Activity: Learning about the Scanner class

The text in the *ArrayList words* in the class *Words* is encoded. It represents verses from a poem - if you read only the first words.

Java allows you to **parse** *String*s using the methods in the *Scanner* class. Look up the *Scanner* class in JavaDocs. The methods there allow you to traverse over a *String* and produce one word at a time. To determine what characters should indicate the end of a word, the *Scanner* class methods use *regular expressions*. You will learn about these later.

The following method finds the first word (sequence of only lower case and capital letters) in a *String*, ignoring the leading white space:

```
/**
  * find the first word of the given <code>String</code>
  *
  * @param s the input
  * @return the first word of the input
  */
 public String firstWord(String s){
   Scanner firstWord =
     new Scanner(s.trim()).useDelimiter("[^a-zA-Z]");
   return (firstWord.next());
}
```

The method *trim* in the class *String* produces a *String* with the leading and trailing whitespaces removed.

The new instance of the *Scanner* class is given this *String* to process. The method *useDelimiter* tells the *Scanner* instance to use any characters other than the letter of the alphabet as word separators.

Read the description of the method for the *Scanner* class and look up the methods *hasNext* and *next*. They are similar to our traversal methods. Use them to design the method *makeWords* that consumes one *String* and produces an *ArrayList* of words.

Define this method within the *Examples* class and use it to read the poem encoded in the first words of the *String*s in the *ArrayList words* in the class *Words*.

10.4 Activity: Eliza Game

Our goal is to train our computer to be a mock psychiatrist, carrying on a conversation with a patient. The patient (the user) asks a series of questions. The computer-psychiatrist replies to each question as follows. If the question starts with one of the following (key)words: Why, Who, How,

Where, When, and What, the computer selects one of the three (or more) possible answers appropriate for that question. If the first word is none of these words the computer replies 'I do not know' or something like that.

1. Start by designing the class *Reply* that holds a keyword for a question, and an *ArrayList* of answers to the question that starts with this keyword.

2. Design the method *randomAnswer* for the class *Reply* that produces one of the possible answers each time it is invoked. Make sure it works fine even if you add new answers to your database later. Make at least three answers to each question.

   The method

   ```
   MathUtilities.randomInt(low, high);
   ```

   generates a random number in the range from *low* (inclusive) to *high* exclusive.

3. Design the class *Eliza* that contains an *ArrayList* of *Reply*s.

4. In the class *Eliza* design the helper method *firstWord* that consumes a *String* and produces the first word in the *String*. We have already seen a similar method earlier in this lab.

   Make sure your program works if the user uses all uppercase letters, all lower case letters, mixes them up, etc. (Again, let the Java documentation help you find the solution.)

5. In the class *Eliza* design the method *answerQuestion* that consumes the question *String* and produces the (random) answer. If the first word of the question does not match any of the replies, produce an answer *Don't ask me that.* — or something similar. If no first word exists, i.e., the user either did not type any letters, or just had hit the *Return*, throw an *EndOfSessionException*.

   Of course, you need to define the *EndOfSessionException* class.

6. For practice, design a new class *EndOfSessionException* that extends the *Exception* class. An example that shows how to design a class that extends *Exception* see the code for *IllegalUseOfTraversalException*.

7. In the *Interactions* class design the method *playEliza* that repeats asking questions and providing answers until it catches the *EndOfSessionException* — at which time it ends the game.

8. You can now play the game, using the *playEliza* method in the *Interactions* class.