**Text input library**

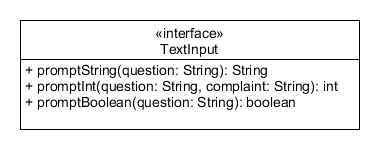
***This exercise is not easy, but should give you both great training in programming and good design practices (code reuse, interfaces), and a good tool for your semester project.***

Create your own text input library to use in your project.

Writing the same code is tedious. Taking input from the user through the console is often repeated. A uniform formatting, and a failsafe behavior is desired. **An example is asking for a number and receiving a word. A program should reject such bad input, offer an explanation, and prompt anew.**

Now it is your task to implement such a reusable component. The procedure is similar to writing ADTs: Specify the interface, and provide an implementation. Re-use this code in your project(s).

You may create the provided interface or modify to suit your needs. Just make sure that the behavior of the implementation lives up to the above highlighted criteria.



**Multiple choice menu**

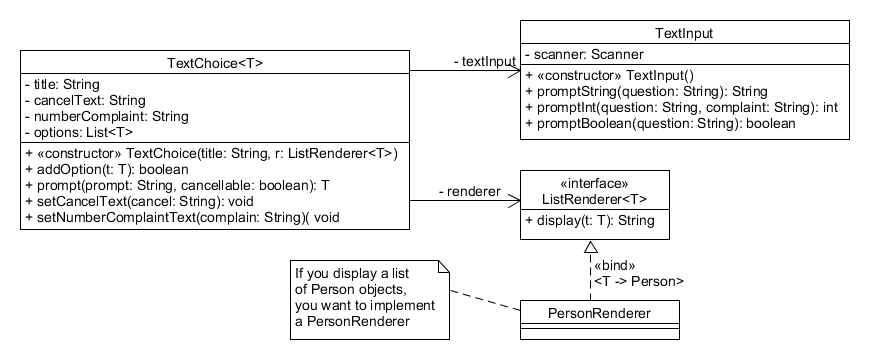
To support the displaying a list of options (picking a Person, a Customer, a Product, etc.) or just a given value (1, 2, 3, and 4) or (“Back”, “Create new”, “Delete”, “Edit”), we add a new class: TextChoice.

To make it generally usable, we consider three things:

1. It should be possible to provide a mechanism that allows us to display any bunch objects without having to modify the implementation,
2. It should be possible to allow or disallow a “Cancel” option,
3. It should be possible to parametrize all dialog texts, s.t. it can be tailored to fit the use case.

Consider the class diagram given below.

1. Create a package called textinput, we will do all the work in this package except for the TryMe class
2. Implement TextInput from the first exercise, if you have not done so yet – otherwise move it to the textinput package
3. Create the interface ListRenderer<T> as specified. This will take the generic type T variable and construct a String that is fit for displaying in the text user interface. (You are not allowed to use the toString() method of model layer objects to display them to the user.)
4. Create the TextChoice<T> class as specified.
   1. The constructor takes the **title** text that is to be displayed in the top of the menu., and the the **ListRenderer<T>** is the implementation you provide – in the example in the class diagram, it would be an instance of PersonRenderer. Also, the constructor should initialize the **cancelText** and the **numberComplaint** text to some sensible default value.
   2. **addOption(t: T)** should store a value in the **options** list
   3. prompt(prompt: String, cancellable: boolean): T should
      1. print the **title**,
      2. if **cancellable** is **true**, print option “[**0] Cancel”**, where “cancel” should be the value in **cancelText**.
      3. loop through the **options** list, and starting from **1**, print each option as rendered by **renderer**.
      4. Finally, it should prompt for a number using **textInput.**promptInt(…) .
      5. It should keep asking (display **numberComplaint**) till the user provides a valid input. This can fail in two ways:
         1. The user doesn’t provide a numeric value. This should be handled by TextInput.
         2. The user doesn’t provide a number in the **valid range**. The valid range is 1 – **options**.size(), including the size. If **cancellable** is **true**, the valid range is 0 – **options**.size(), and 0 means “Cancel” (check above).
      6. When the user enters a valid number (check above), it should return **options.get(input – 1)**. If the input is 0, it should return **null** (indicating “cancel”).
         1. Remember, 0 should not be accepted if **cancellable** is false,
   4. To support the prompt(…) method, you may want to add helper methods
      1. – askForChoice(cancellable: boolean, prompt: String): int
      2. – printOption(no: int, text: String): void
   5. setCancelText(…) and setNumberComplaint(…) are methods that allow you to change the default texts in the respective instance variables.



Test the solution. Examples of using the system are shown below. Note that the ListRenderer<T> interface is implemented as an anonymous inner class, as it is only used this one place in the example. If you plan to re-use the renderer (which you should to give a uniform rendering of you model layer objects), you can just implement it in its own class.

|  |  |
| --- | --- |
| ListRenderer<Person> lr = **new** ListRenderer<Person>() {  @Override  **public** String display(Person p) {  **return** p.getName() + " (" +  p.getEmail() + ")";  }  };  TextChoice<Person> tp = **new**  TextChoice<>("Title text” , lr);  tp.addOption(**new** Person("Joe",  "joe@email.com"));  tp.addOption(**new** Person("Jane",  "jane@email.com"));  tp.addOption(**new** Person("Janice",  "janice@email.com"));  Person p = tp.prompt("Pick a person!",  **true**);  System.***out***.println("Person picked: " +  p); | ListRenderer<String> lr2 = **new**  ListRenderer<String>() {  @Override  **public** String display(String s) {  **return** s;  }  };  TextChoice<String> actions = **new** TextChoice<>("What do you want to do  next?", lr2);  actions.addOption("Back");  actions.addOption("Create new");  actions.addOption("Edit existing");  actions.addOption("Delete");  String res = actions.prompt("Make your  choice", **false**);  System.***out***.println("The system is now  going to do the following: " +  res.toUpperCase()); |
| Title text  [0] Cancel  [1] Joe (joe@email.com)  [2] Jane (jane@email.com)  [3] Janice (janice@email.com)  Pick a person! (0 - 3) 0  Person picked: null | What do you want to do next?  [1] Back  [2] Create new  [3] Edit existing  [4] Delete  Make your choice (1 - 4) 3  The system is now going to do the following: EDIT EXISTING |

**Packaging**

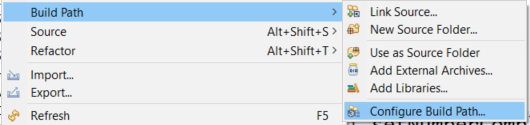
How to re-uese the code? You can always copy-paste the textinput package from one project to another. However, if you are satisfied with it and have tested it, you may want to package it as a distributable file – a JAR-file (Java ARchive). In reality, it’s just a zip file that is re-named .jar, and it contains some additional info-files. For our purposes, Eclipse is able to generate this file.

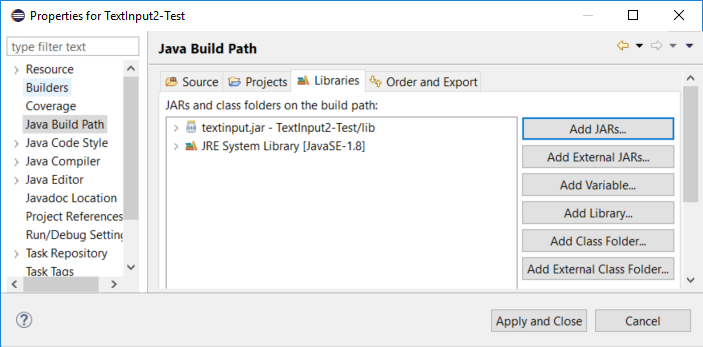
When you are done with your text input exercise, and it works as expected,

1. right click the **textinput** package
2. pick the **Export…** option from the context menu
3. Expand the **Java** option and pick **JAR File**
4. In the next window, verify that it only contains the desired package, you do not need anything from the rest of the Eclipse project
5. Click **browse** to select a location for your JAR file. This is where you will find it later
6. Click **Finish**

Now, you should be able to create a **new Java project**, **import the jar** file and use the functionality.

1. Create a new Java project in Eclipse
2. Copy-paste the jar file **into the Project** **(!)** NOT INTO THE JAVA PACKAGE or anywhere else.
   1. You may create a folder (**a folder, not a package**), called **lib** and put it there, that would be even better.
3. Right-click the **Eclipse project**, select **Build Path** pick **Configure Build Path**.
   1. On the right hand side, pick **Libraries**
   2. Click **Add JARs…**
   3. Navigate to your current project, find the JAR file, and select it. Click **Apply and Close**.
4. Now, you can make a TryMe class, and you can try to use your library. It is magic, the code is not there, but the functionality is. This is how you use the rest of the Java standard libraries like ArrayList, and the like.





**Documentation**

It is annoying when you forget what a method does and why. If you have forgotten to do so, go back, add JavaDoc to your library, and re-pacakge it!