Problem 1: ObservableValueBase

In project 03_Observer2 you find in the package `patterns.observer.list` a small example in which the observer pattern is implemented with classes `Observable` and the interface `Observer`.

As a concrete application, an observable list is implemented in which elements of type E can be added and removed. Whenever the list is changed, all registered observers are notified. The following observers have been implemented:

- `ConsoleObserver`: only prints that the content of the list has changed
- `AdderObserver`: prints the current sum of all integers contained in the bag (obviously, this listener can only be registered in a list of type `ObservableList<Integer>`).

The main method of program `ObserverTest` registers the observers and adds a value to the list:

```
public static void main(String[] args) {
      ObservableList<Integer> list = new ObservableList<>();
      ConsoleObserver co = new ConsoleObserver();
      AdderObserver ao = new AdderObserver();
      list.addObserver(co);
      list.addObserver(ao);
      list.addValue(17);
}
```

If this program is executed, the following output is printed on the console:

```
ConsoleObserver: List has changed
AdderObserver: new sum is 17
```

Your task is to implement class `ObservableList` as an extension of class `ObservableValueBase`.

Tasks:

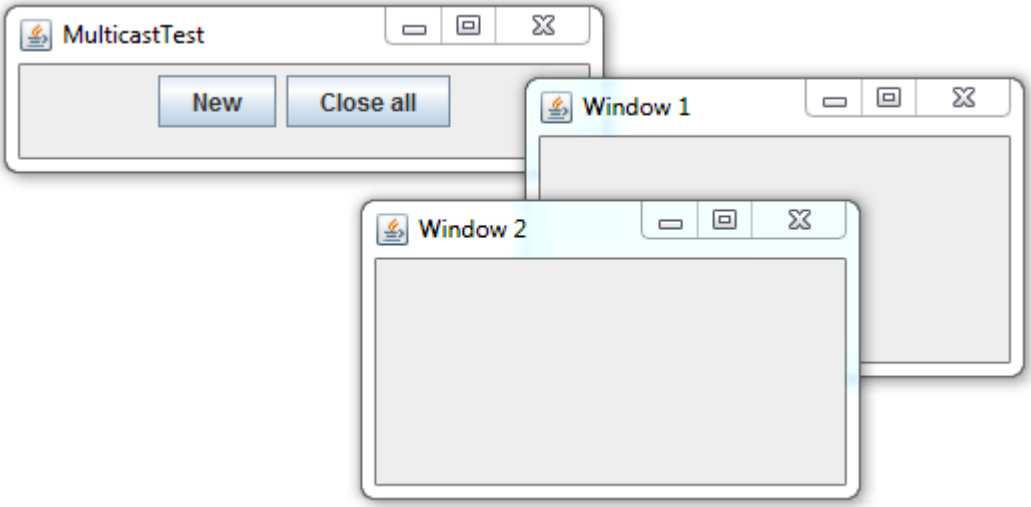
1. Problem analysis (alone or in pairs):
   - Study the API documentation of class `javafx.beans.value.ObservableValueBase`.
   - Change the base class of class `ObservableList` to class `ObservableValueBase<List<E>>` (you can delete the given class `Observable` together with the given interface `Observer`). Correct the errors and implement the missing methods.
   - Implement the `AdderObserver` as a `javafx.beans.value.ChangedListener<List<Integer>>` and register only this observer in the list. This observer is called whenever the list is changed, i.e. if the old value returned by `getValue` is not equal to the new value returned by `getValue` (the comparison in the base class of `ObservableList` is done with `equals`).
     Is your implementation correct? If not, explain why not and correct your implementation.
   - Implement the `ConsoleObserver` as a `javafx.beans.InvalidationListener` and register only this observer in the list. This observer should then only be invoked twice if the test program is executed, namely only if the list values have been queried once since the last notification (the values are queried with `printList(list)`).
     Is your implementation correct? If not, explain why not and correct your implementation.

2. Expert round:
   Summarize your observations and record how class `javafx.beans.value.ObservableValueBase` has to be used.

3. Presentation:
   You have a time slot of approx. 5-7 minutes to present your results. A short discussion with the class completes your presentation.
   Follow the following structure for your presentation:
   a) Description of the problem (with the help of the concrete examples for the problem you worked on). Please remember, that your fellow students have *not* studied your problem!
   b) Solution of the problem
   c) Generalization / general rules to follow

Problem 2: Observer & Resources

In project 03_Observer2 you find in the package `patterns.observer.memory` a small example which is taken from the book *Core Java.* The program allows to open several windows and to close them using the *Close all* button.



The small windows are implemented in the inner class `SimpleFrame`. In order to demonstrate the problem, each instance of class `SimpleFrame` allocates a byte array of size 256MB.

Start the program. Open several windows and close them again using *Close all* and repeat this procedure several times. What do you observer? Reflect on what could be the problem of this program.

Tips:

- Print a text in method `SimpleFrame.actionPerformed` in order to see which action listeners are executed when *Close all* is pressed (e.g. a text which prints the number of the window).
- Implement in class `SimpleFrame` method `public void finalize()`. This method is called whenever an object has been released and could be removed by the garbage collector. The finalize method provides a "last wish" to this object. This way you see which objects are ready to be removed by the garbage collector.
- Use the Java tool JConsole to visualize the heap memory usage of the above program. This tool also allows to perform explicit garbage collection runs.

Tasks:

1. Problem analysis (alone or in pairs):
   - Describe the problem and its cause.
   - How can the problem be solved? Propose (and implement) a solution.

2. Expert round:
   Derive a rule for the observer pattern from your observations. What must be considered when working with the observer pattern? In which situations might problems appear? Do such problems appear with JavaFX as well?

3. Presentation:
   You have a time slot of approx. 5-7 minutes to present your results. A short discussion with the class completes your presentation.
   Follow the following structure for your presentation:
   a) Description of the problem (with the help of the concrete examples for the problem you worked on). Please remember, that your fellow students have not studied your problem!
   b) Solution of the problem
   c) Generalization / general rules to follow

PS
In the current version of the book "Core Java" the code has been fixed.

Problem 3: Side effects of changes

In project 03_Observer2 you find in the package `patterns.observer.once` a small example in which the observer pattern is implemented with class `Observable` and the interface `Observer`. The sample application implements a sensor which contains a `double` value. This value can be queried with method `getValue` and changed with method `setValue`. At each change all registered observers are notified. As an example the following observer implementation is provided:
- `PrintObserver`: Prints the new value of the sensor on the console

Your task is to investigate a further observer. This observer unsubscribes itself from the subject from within the `update` method:

```
public class OnceObserver implements Observer {
	@Override
	public void update(Observable source){
		System.out.println("Once Observer called, bye...");
		source.removeObserver(this);
	}
}
```

The main program registers four observers and changes the value of the sensor several times. Execute this program and observer its behavior.

```
public static void main(String[] args) {
	Sensor s = new Sensor(20);
	s.addObserver(new PrintObserver("Printer 1"));
	s.addObserver(new OnceObserver());
	s.addObserver(new PrintObserver("Printer 2"));
	s.addObserver(new PrintObserver("Printer 3"));
	s.setValue(22);
	s.setValue(30);
	s.setValue(25);
	s.setValue(22);
}
```

Tasks:

1. Problem analysis (alone or in pairs):
   Think about what happens if the `OnceObserver` is executed, i.e. what happens if an observer unsubscribes itself from within the `update` method.
   Tip: What happens with iterators if the collection they are iterating over is changed?

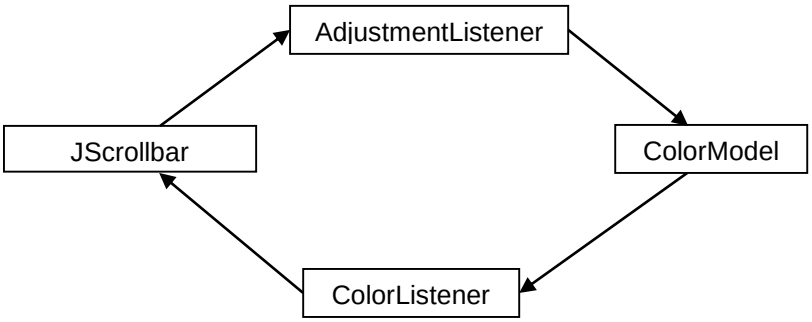
   Does the problem also occur
   a) if the `OnceObserver` is registered as last observer in the sensor?
   b) if the `OnceObserver` is registered as second to last in the sensor?
   c) if several `OnceObservers` are registered?
   d) if the forEach method (with lambda expression) is used instead of the for loop?
2. Expert round:
   How can this problem be fixed? Propose a solution how the problem can be solved in class `Observable`. If you have no idea then you can also look into class `java.util.Observable` or into the implementation of the notification in JavaFX (`com.sun.javafx.binding.ExpressionHelper`).
3. Presentation:
   You have a time slot of approx. 5-7 minutes to present your results. A short discussion with the class completes your presentation.
   Follow the following structure for your presentation:
   a) Description of the problem (with the help of the concrete examples for the problem you worked on). Please remember, that your fellow students have not studied your problem!
   b) Solution of the problem
   c) Generalization / general rules to follow

Problem 4: Cyclic dependencies

In the color picker application you worked on in assignment 1 there is a scrollbar which is used to change the value of the red channel of a color (we neglect all other controls such as the menu items or the text fields, they are not of interest for this problem).

If the value of the scrollbar is changed, then an event is sent to all listeners registered in the scrollbar. One of these listeners will then update the color in the color model.

The color model on the other hand sends a color event to all registered color listeners. One of these color listeners sets in turn the value of the scrollbar according to the chosen color.



In project 03_Observer2 you find in the package `patterns.observer.cycle` a small example which illustrates this problem. The `ColorModel` as well as class `RedScrollbar` extend class `Observable`. The test program creates a `ColorModel` and a `RedScrollbar`, which represents the red channel of the selected color. A Listener is registered in the Scrollbar which in turn sets the color in the model. The program first sets the color of the model to gray and then the value of the scroll bar is set to 44. The value of the scrollbar and the value of the red channel of the chosen color should be identical.

```
model.setColor(Color.gray);
System.out.println("model red value = " + model.getColor().getRed());
System.out.println("scrollbar value = " + sb.getValue());
sb.setValue(44);
System.out.println("model red value = " + model.getColor().getRed());
System.out.println("scrollbar value = " + sb.getValue());
```

Tasks:

1. Problem analysis (alone or in pairs):
   - Describe the problem of this color picker application. Draw a sequence diagram which illustrates this problem.
   - How can this problem be solved? Propose two solutions.

2. Expert round:
   Derive a rule for the observer pattern from your observations. What must be considered when working with the observer pattern? In which situations might problems appear?

3. Presentation:
   You have a time slot of approx. 5-7 minutes to present your results. A short discussion with the class completes your presentation.
   Follow the following structure for your presentation:
   a) Description of the problem (with the help of the concrete examples for the problem you worked on). Please remember, that your fellow students have not studied your problem!
   b) Solution of the problem
   c) Generalization / general rules to follow

Problem 5: What is the model, how is it changed?

For the implementation of the color picker the following solution has been proposed (in particular in several JavaFX based solutions): Instead of a property of type Color, the model (or the controller) provides three integer properties for the channels red, green and blue of a color. We could also say that the model consists of three separate models.

If a color is set, then the three methods `setRed`, `setGreen` and `setBlue` have to be invoked. The advantage of this solution is, that the sliders and the text fields, which each represent only one color channel, can be bound directly to these three properties. On the other side, radio buttons and menu items (which represent a particular color) have to change all three channel models in their action listener code.

In project 03_Observer2 you find in the package `patterns.observer.multimodel` a small example which implements this approach. When a listener is registered in the model you can specify on which channels it should listen to.

Tasks:

1. Problem analysis (alone or in pairs):
   Think about what problem this approach suffers from.
   In the program `ColorTest` the following scenario is simulated: The model is initialized with the color black and then changed to white. The program also registers a read-allergy-listener, which changes the color to gray whenever red was chosen. What is the color at the end of this procedure?

2. Expert round:
   How can this problem be resolved?
   What conclusions do you draw from these observations?

3. Presentation:
   You have a time slot of approx. 5-7 minutes to present your results. A short discussion with the class completes your presentation.
   Follow the following structure for your presentation:
   a) Description of the problem (with the help of the concrete examples for the problem you worked on). Please remember, that your fellow students have not studied your problem!
   b) Solution of the problem
   c) Generalization / general rules to follow

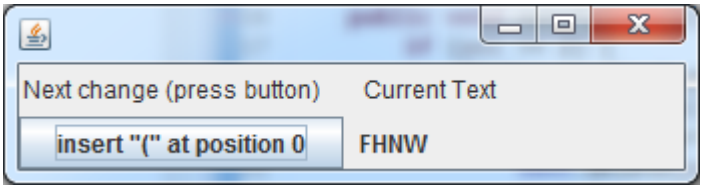
Problem 6: Causality of changes

In this problem we examine a simple text editor whose model is implemented with class `StringBuilder`. The model allows the insertion and deletion of characters. At each change the model informs all registered listeners either with method `notifyInsert(int pos, char ch)` or `notifyDelete(int from, int len)`.

We assume, that the following concrete listener implementations exist:

- A text listener which presents the text in a window. This listener updates the text only based on the insert and delete notifications it receives, i.e. this listener does not need a reference to the model.
- A correction listener which performs simple auto corrections. Specifically, this listener checks upon insertion of a closing parenthesis whether the character sequence „(“ , „c” and „)” has been entered and if yes, replaces these characters by the character „©“. To that end this listener removes the sequence "(c)" und inserts afterwards the new character "©".

In project 03_Observer2 you find in the package `patterns.observer.copyright` a small example which illustrates this problem. The text model is already initialized with the Text “FHNW” and with each click on the button a new character is inserted into the model. To the right of the button the content of the text listener is shown (which adjusts itself based on the notification received only).



Tasks:

1. Problem analysis (alone or in pairs):
   - Describe the problem with this configuration and try to explain why this problem appears (e.g. with the help of a sequence diagram).
   - Change the order of the two listeners. Does the problem still appear? If no, why not?

2. Expert round:
   How can this problem be solved?
   An obvious solution would be to register the correction listener *after* the text listener which present the content of the model. However, there could also be several correction listeners to be registered, therefore we are looking for a general solution!

3. Presentation:
   You have a time slot of approx. 5-7 minutes to present your results. A short discussion with the class completes your presentation.
   Follow the following structure for your presentation:
   a) Description of the problem (with the help of the concrete examples for the problem you worked on). Please remember, that your fellow students have not studied your problem!
   b) Solution of the problem
   c) Generalization / general rules to follow