Ticket Seller Lab

Overview
In this lab you will work with a program that simulates sales of movie tickets in a small theater. The theater is showing three movies in theaters with different capacities and different pricing for each theater. The box office seller selects the number of tickets in each category and performs the sale or inquires about the price of tickets as well as availability. The cumulative sales statistics for the selected show are also displayed.

You will identify different parts of the code, to become more familiar with the structure of class definitions and the use of member data, member functions, and constants. You will also get a first view of how the Graphical User Interfaces (GUIs) are built and used. Finally, you will make small changes and modifications - to fill in the missing parts and to refactor the code so it is better organized and more robust.

Program summary
The TicketSeller program uses two classes. The class Show contains the information about the movie for which we are selling tickets. It has member data for the movie name, the capacity of the theater, and the prices for adults, students, and children. In addition, it also has member data used to collect sales statistics: the number of tickets sold in each price category. To find out if there are still tickets available, or what was the total gross income up till now, one only needs to perform simple arithmetic on the available data.

The class TicketSeller coordinates the building of the entire graphical user interface and contains the member functions that perform the desired box office actions as follows:

The show title and the current sales statistics are then displayed in the top half of the GUI, together with buttons that select one of the shows or reset the sales statistics for the current show.

The bottom half of GUI simulates a box office. It displays again the selected show title, the number of tickets available, and has three TextFieldView input fields to select the number of desired tickets in each of the three categories (adult, student, child) at displayed prices. Below are three action buttons:

- **Clear Order** button just sets the current order request to all zeros.
- **Total** button computes the total number of tickets requested and shows the total number of tickets requested and the price for the order. If the number of tickets exceeds the availability, a message "Sold out" is shown instead of the total price.
- **Buy** button performs the actual sale, if the desired number of tickets is available. It again shows the total number of tickets requested and the total price (or the "Sold out" message).
**Snapshot of running application**

Observe a snapshot of completed application. Verify that the income has been computed correctly. Verify that the price of current order is correct, as well as the number of tickets requested. Can you tell what is the seating capacity for this show?

**Lab Tasks**

In this lab you have to perform the following tasks. Do not skip the 'reading' tasks - they will make the programming part much easier and will help you learn more about object oriented programming in Java.

1. Study the Show class.
2. Study how the graphical user interface in the TicketSeller class is built.
3. Read the code for the action performing functions and modify them to fill in the missing computations and GUI display statements.
4. Learn how to use and modify helper functions.
5. Add the code to assure the integrity of data and refactor the code: learn how to encapsulate work in helper functions.
Part 1: The Show Class

Constants

Sometimes the programmer wants to give names to some constant values that will be used in the program - such as initial and default values of some variables. Here we define a fictitious title for a show and a fictitious set of prices to be used if the user does not specify the information about the show.

The code is:

```java
private static final String SHOW_NAME = "RBBBC";
private static final int ADULT_PRICE_VALUE = 15;
private static final int STUDENT_PRICE_VALUE = 10;
private static final int CHILD_PRICE_VALUE = 5;
```

The constant names are always written in all capital letters with the underscore _ character separating the words. (SC-Q1, SC-Q2)

Member data

There are three sets of member data for this class: the information about the show, the price list, and the member data that record the sales statistics. (SC-Q3, SC-Q4)

The show information includes the name of the show and the capacity of the theater.

```java
protected String showName = null; // movie title
protected int capacity; // theater capacity for this show
```

The member data representing the price list information are defined as follows:

```java
protected int adultPrice; // price of adult ticket
protected int studentPrice; // price of student ticket
protected int childPrice; // price of child ticket
```

The member data representing the sales statistics are defined as follows:

```java
protected int adultsSold; // number of adults tickets sold
protected int studentsSold; // number of students tickets sold
protected int childrenSold; // number of children tickets sold
```

We assume all prices are in whole dollars and so use integers for the price values.

Member functions

There are three sets of member functions in this class. The first one is used to initialize and reset the information stored in the member data regarding the show, the pricing, and
the total sales. The second set of member functions allows the outside user to inquire about information recorded in the member data and set the values of specific member data items. The third set of functions does the work - they perform the actions for which the class has been built: compute the price of a ticket order, check for ticket availability, and process the sale by computing the price of the order and updating the sales statistics.

Constructors and functions that set and reset the object state

Typically a class has a default constructor that is used when the caller does not supply any information. We elected to create a fictitious show with 100 seats and an arbitrary pricing as follows:

```java
// default constructor
Show(){
    // imaginary show name and capacity
    showName = SHOW_NAME;  // 100 tickets are available
    capacity = 100;
    // fictitious prices - using the pre-defined constants
    adultPrice = ADULT_PRICE_VALUE;
    studentPrice = STUDENT_PRICE_VALUE;
    childPrice = CHILD_PRICE_VALUE;
    // set the total sales figures to zeros
    reset();
}
```

The full constructor sets all the member data from the information supplied by the caller. We do not allow the user to set the number of tickets already sold - each show starts with clear sales statistics. The constructor defers its work to a member function that allows the user to change the show information later - once the object has been created (SC-Q5):

```java
// full constructor
// initialize the show name, number of seats, pricing, and total sales
Show(String name, int seats, int adult, int student, int child){
    // call the set function below
    setShow(name, seats, adult, student, child);
}
```

The `setShow` function performs the actual work. This way, if we modify how this function works, the constructor will be modified too - avoiding inconsistencies. You will later modify this function to assure that all input values are non-negative integers.

```java
// set the show name, number of seats, pricing, and total sales
public void setShow(String name, int seats, int adult, int student, int child){
    // initialize the show name and seating capacity
    showName = name;
    capacity = seats;
    // set the prices as specified by the caller of the function
    adultPrice = adult;
    studentPrice = student;
    childPrice = child;
    // set the total sales figures to zeros
    reset();
}
```
The reset function allows the user to start selling tickets again for this show - it clears the number of tickets sold, leaving the show name, the capacity, and the prices the same.

```java
// start selling tickets again with the same pricing
public void reset(){
    adultsSold = 0;
    studentsSold = 0;
    childrenSold = 0;
}
```

Accessors: the set and get functions

If the user of the class needs to have access to some information stored as member data, the class provides the appropriate accessor functions. The set functions allow the user to directly set some member data values. We decided that the user may change the name of the show, but no other information (unless setShow is used to change all the member data at once). On the other hand, we allow the user to query the information in all of the member data fields related to the number of tickets sold or to ticket prices, because this information will be displayed in the user interface. In this way the class controls how these values are manipulated, while allowing the main program to gather the information it needs. All get functions look alike - we copy here for illustration only three functions:

```java
public void setShowName(String name){
    showName = name;
}

public int getAdultPrice(){
    return adultPrice;
}

// return the number of tickets still available
public int getAvailability(){
    int ticketsSold = adultsSold + studentsSold + childrenSold;
    return (capacity - ticketsSold);
}
```

The third function can be also classified as a function that performs action. It needs to do some computation first, before it can return the number of tickets still available. Originally, we kept this information as a separate member data value, but changed our minds later. Both choices are acceptable. (SC-Q6)

Functions that perform actions, update statistics

There are four functions here. The first two both return the price of the order, given the number of tickets requested in each category, regardless of the availability of the tickets. The sell function, in addition, adjusts its record of sold tickets. (SC-Q7)

```java
// return total price
public int price(int adults, int students, int children){
    int totalPrice = adults * adultPrice + students * studentPrice + children * childPrice;
    return totalPrice;
}
```
Notice that the `sell` function uses the `price` function's computation - following the rule of performing each task in only one place:

```java
// sell the desired number of tickets if seats are available
// return total price
public int sell(int adults, int students, int children){
    // total number of tickets requested
    int tickets = adults + students + children;
    // check for availability
    if (tickets <= getAvailability()){*
        // adjust the cumulative totals of tickets sold
        adultsSold += adults;
        studentsSold += students;
        childrenSold += children;
    }
    // return the total price of order
    return price(adults, students, children);
}
```

The remaining two functions inform the user whether the desired number of tickets is available. In the first case the total number of tickets is specified, while the second function allows the user to specify the number of tickets in each category. (SC-Q8)

```java
// return true if the desired number of tickets is available
public boolean available(int tickets){
    return (getAvailability() - tickets >= 0);
}

// return true if the desired number of tickets is available
public boolean available(int adults, int students, int children){
    int tickets = adults + students + children;
    return (available(tickets));
}
```

Notice that the second function follows the same technique as we used in the full constructor. Instead of re-evaluating the availability, it only computes the total number of tickets desired and calls the first function to do the rest of the work. Java allows us to use the same function name several times, as long as the list of function arguments is different in each case. It gives the user the option to specify the arguments in a form most suitable for the particular situation.

Congratulations!!! You have now completed reading the code of your first complete Java class. (SC-Q9, SC-Q10)

**Part 2: Building the Graphical User Interface (GUI): TicketSeller Class**

The code for the main application class `TicketSeller` is about seven pages long. This is a formidable length. However, once we look at the details, it will become clear that most of the code is devoted to repetitious tasks that nevertheless have to be done. Please, do not worry about the details how these components are defined - just try to see that indeed a lot of code is repeated. Also, look for the words mentioned in the text: `Annotation`, `Display`, `TablePanel`, `ActionsPanel` and see the flow of the program.
The class starts with a list of import statements that tell the compiler where to find the definitions of the classes for the objects that will be used, as well as definitions of some constants that will be used. For now you do not have to worry about what needs to be imported - we will learn this gradually. The import statements are:

```java
import edu.neu.ccs.*;
import edu.neu.ccs.gui.*;
import edu.neu.ccs.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

The beginning of the class definition is more complicated too:

```java
public class TicketSeller extends DisplayPanel
    implements JPTConstants {
```

It indicates that this class will be able to do everything a `DisplayPanel` can do - and more. We say that it inherits the `DisplayPanel` functionality, or that this class extends the `DisplayPanel` class. In addition the strange phrase `implements JPTConstants` indicates that all constants defined in the interface can be used by the member data and member functions of this class.

The member data section starts with barely noticeable definition of the `currentShow` object that will represent the information about the show we are selling tickets for. This is followed by the definition of the three shows we will actually sell the tickets for. These three definitions use the full constructor for the `Show` class - specifying the show title, ticket availability and the pricing structure. These objects will retain the information about the sales for a particular movie. The `currentShow` will then reference one of these three objects, as selected by the box office salesperson.

The rest of member data represent the GUI components. There are 37 separate components inside of the `TicketSeller` for this GUI and we need at least one statement to install each one of them. Occasionally, we have additional statements that adjust the appearance of the component. Let us look at what are these components.

Most of the information is displayed for the user to see, but not to modify. For this we use `Annotation` object that displays any specified text. Almost a page of the text is taken by these annotations. (TS-Q1)

The only information the user (a box office salesperson) can enter is the number of desired tickets in each of three categories. For this we use three `TextFieldViews`. (TS-Q2)

The program allows the user to select a show using one of three action buttons. The fourth button under the sales statistics display allows the user to reset the statistics for the current show. The ticket sales section of the GUI has three action buttons: one to clear the user entered ticket order, one to compute the total price and one to actually buy the tickets. We define the action buttons and specify where to find the code for the actions to
be performed in the next section of the code. We also build two GUI **ActionsPanel** panels that will hold these seven buttons. *(TS-Q3, TS-Q4, TS-Q5)*

The next page of the code organizes the annotations into table panels with titles, so that user has an easy time figuring out the meaning of each part. The last statement creates a main panel that contains the four sections: the **ticketSalesDisplay**, the **resetActionsActionsPanel**, the **ticketOrderDisplay**, and the **orderActionsActionsPanel**. *(TS-Q6)*

**Member functions**

**main function**

To run a Java application we need a function with the header:

```java
public static void main(String[] args)
```

to be included in one class among all classes. The execution of our program starts here.

The **main** function lets the constructor for the **TicketSeller** create all of its panels, adjust sizes and alignment, create a new instance of the **Show** class, and then it installs the entire **GUI** panel in the main application frame titled "Ticket Seller".

```
// create the main ticket seller display panel
// call its constructor to install all the inner panels
public static void main(String[] args) {
    JPTFrame.createQuickJPTFrame(        "Ticket Seller",    // title for the main application GUI    new TicketSeller());    // call to the TicketSeller constructor
}
```

**The constructor**

The constructor starts with creating a show for which we sell the tickets. It starts by calling the action function **tarzan()**.

```
    // make the initial show to be Tarzan
    tarzan();
```

We will see later what this function accomplishes and how it not only selects Tarzan to be the show for which we are selling tickets, but also updates all the relevant displays and clears the ticket order fields. This way the constructor delegates its work to another function that has a more general purpose in the class.

The remaining statements in the constructor sets two of the annotations to be show in red, adjust the heights of the lines in the ticket order display and the overall size of the main panel.

```
    // display the show title and available tickets in red
    // in the ticket order display
    showTitleFieldColor.setForeground(Color.red);
    availableTicketsField.setForeground(Color.red);
```
// View Section

// set the minimum height of the fields in the ticket order display
minimumHeights = adultOrderTFV.getPreferredSize().height;
adjustTableSettings(ticketOrderPanel);
mainPanel.setMinimumColumnWidth(0, MAIN_PANEL_WIDTH);

Finally, the statement:

    // add the main panel to the main display
    add(mainPanel);

does the most important work - it adds the TablePanel mainPanel we built to the TicketSeller DisplayPanel.

Part 3: The Action Functions: Modifying the Code

There are seven action functions: reset(), tarzan(), kingKong(), casablanca(), clear(), total(), and buy(). When the user presses one of the action buttons, the code defined in the corresponding function is executed. The first four actions control the show selection and show statistics. The last three actions control the ticket purchase.

There are three sets of data displayed in this program for any given show. The first is the basic unchanging information about a show - the show title, the seating capacity, and the price structure. The second set of data consists of the sales statistics and the ticket availability. This information is computed from the information recorded in the Show object's member data. The third set of data relates to the current ticket request and contains the number of requested tickets and, if already computed, the totals of the requested tickets and price. There are four member functions that control the display of these three sets of data.

- The clear() function clears the ticket order request, and the computed totals for this order.
- The reset() function resets the sales statistics, calls updateSalesView() function to reflect the changes, and also calls the clear() function to clear the order.
- The updateShowView() function resets the unchanging information about the given show - and is used when we decide to select a different show.
- The updateSalesView() function displays the sales statistics and the ticket availability and is used both when we change the currently selected show and when we reset the statistics.

Each of the three show selection functions: tarzan(), kingKong(), casablanca(), set the current show to be the desired show, then call the updateSalesView() function, the updateShowView() function, and the clear() function. This is not the best design and you will make changes to improve it.
Action functions

We will now look at each of the action functions once more in detail. We will look at the helper/utility functions `updateShowView()` and `updateSalesView()` afterwards.

`reset()`

This function first resets the current show object's statistics and displays the updated information. It then clears the box office requests and the computed totals of tickets and price.

```java
public void reset() {
    // reset the current show statistics and show the new information
    currentShow.reset();
    updateSalesView();
    // clear the box office request views and totals of tickets and price
    clear();
}
```

tarzan()

This function makes the `currentShow` object to refer to the Tarzan show object and then calls the needed helper functions to adjust the display.

```java
public void tarzan() {
    // set the current show to be Tarzan and show the new information
    currentShow = tarzanShow;
    // show current title and sales statistics
    updateSalesView();
    // show prices for this show
    updateSalesView();
    // clear the box office request views and totals of tickets and price
    clear();
}
```
The other two functions: `kingKong()` and `casablanca()` look nearly identical. (AF-Q1)

clear()

This function just sets the view state for the three TextFieldViews that represent the user request to show 0 tickets requested in each category and it sets the total tickets requested and the total price fields to zero.

```java
public void clear(){
    adultsOrderTFV.setViewState(NO_TICKETS);
    // add the missing code
    childrenOrderTFV.setViewState(NO_TICKETS);
}
```
totalOrderField.setText(NO_TICKETS);
totalPriceField.setText(NO_SALES);
}

Notice how the view state of a TextFieldView is set by specifying the text that will be displayed in the field. (AF-Q2, AF-Q3)

total()

This function starts by extracting the desired number of tickets specified by the user from the TextFieldViews by calling the member function demandInt: (AF-Q4, AF-Q5)

```java
int a = adultsOrderTFV.demandInt();
int s = studentsOrderTFV.demandInt();
int c = childrenOrderTFV.demandInt();
```

It then computes the total number of tickets and displays the value in the GUI:

```java
int tickets = a + s + c;
totalTicketsField.setText( string(tickets) );
```

Next, we ask the currentShow object to call its member function price and tell us what is the total price for these tickets:

```java
int price = currentShow.price(a, s, c);
```

The last part of this function asks the currentShow to check the ticket availability by calling its available member function. If the tickets are available, the total price is displayed, otherwise a message "Sold out" is given instead of the total price:

```java
if (currentShow.available(tickets))
    totalPriceField.setText( dollar(price) );
else
    totalPriceField.setText("Sold out");
```

buy()

The first five statements of this function are the same as in the total() function. Later we will see how to avoid repeating this code twice. The second part is slightly different:

```java
if (currentShow.available(tickets)){
    int price = currentShow.sell(a, s, c);
    totalPriceField.setText( dollar(price) );
}
else
    totalPriceField.setText("Sold out");
```

We first inquire whether the show is sold out. If it is not, we ask the currentShow to call its sell member function instead of the price member function called earlier. (AF-Q6) The rest of the code is the same as in the total() function above.

**Bonus:** See question AF-Q7.
Part 4: The Helper/Utility Functions: Modifying the Code

The helper function `updateShowView` refreshes the display to show the unchanging information about the current show - the title, and the price structure (HF-Q1):

```java
private void updateShowView(){
    // display the show title in sales statistics display
    showTitleField.setText(currentShow.getShowName());
    // display the show title in ticket order display
    showTitleFieldColor.setText(currentShow.getShowName());
    // show prices
    adultPriceField.setText(dollar(currentShow.getAdultPrice()));
    childPriceField.setText(dollar(currentShow.getChildPrice()));
    // show tickets availability information
    availableTicketsField.setText(string(currentShow.getAvailability()));
    // display the show title in sales statistics display
    showTitleField.setText(currentShow.getShowName());
    // display the show title in ticket order display
    showTitleFieldColor.setText(currentShow.getShowName());
    // show prices
    adultPriceField.setText(dollar(currentShow.getAdultPrice()));
    childPriceField.setText(dollar(currentShow.getChildPrice()));
    // show tickets availability information
    availableTicketsField.setText(string(currentShow.getAvailability()));
}
```

The helper function `updateSalesView` updates the sales statistics and the number of tickets still available by extracting the information from the `currentShow` object.

```java
private void updateSalesView(){
    // update ticket availability information
    availableTicketsField.setText(string(currentShow.getAvailability()));
    // update the number of adult tickets sold and income information
    int adult = currentShow.getAdultsSold();
    int adultPrice = currentShow.getAdultPrice();
    int adultIncome = adult * adultPrice;
    adultSalesField.setText(string(adult));
    adultIncomeField.setText(dollar(adultIncome));
    // update the number of student tickets sold and income information
    int student = currentShow.getStudentsSold();
    int studentPrice = currentShow.getStudentPrice();
    int studentIncome = student * studentPrice;
    studentSalesField.setText(string(student));
    studentIncomeField.setText(dollar(studentIncome));
    // update the display of total number of tickets sold and total income
    totalSalesField.setText(string(adult + student + child));
}
```

There are two parts missing in this function - your task is to add them, following the pattern shown in the rest of the function (see below, HF-3 and HF-4).
For each category of prices we first get the information about the number of tickets sold and the ticket price for that category:

```java
int adults = currentShow.getAdultsSold();
int adultPrice = currentShow.getAdultPrice();
```

We then compute the income for that category:

```java
int adultIncome = adults * adultPrice;
```

Finally, we set the two relevant Annotation with the number of tickets sold and the income for that category (HF-Q2):

```java
adultsSalesField.setText ( string(adult)       );
adultsIncomeField.setText( dollar(adultIncome) );
```

At the end, the `totalSalesField` and `totalIncomeField` are set to correct values again using the `setText` member function calls (HF-3, HF-4):

```java
totalSalesField.setText( string(adults + students + children) );
```

**Part 5: Integrity of Member Data**

When the user creates a new `Show` object, there is nothing preventing him/her from giving a negative number as the ticket price, or specifying that we have -45 seats available. One of the advantages of building classes is that the constructor can verify whether the given values are appropriate for this class and adjust the initial values to be meaningful. You will modify the full constructor for the `Show` class so that the prices and available seat values are always non-negative. (It is OK to have a free admission for children.)

Looking back at the constructor for `Show` class, we remember, that it delegated all of its work to the `setShow` function. Therefore, we need to adjust the `setShow` function. This works out very well, because it is the only place where the user can modify the prices and the ticket availability. This guarantees, that these values will never be corrupted.

The user supplies four `int` arguments to the `setShow` function. Adjust each of the four values using the following pattern, before assigning their values to the corresponding member variables: (ID-Q1, ID-Q2)

```java
if (seats < 0)
    seats = 0;
```

Test your program by modifying the constructor call for the `currentShow` in the `TicketSeller` program to specify negative number of available seats, or negative prices in any of the categories. (ID-Q3)
Refactoring

Refactoring is a code writing strategy. Once we write the code, we look back and see if we could reorganize it to make it look better, be more efficient, more coherent, etc. Here we see the code of the four if statements repeated four times. We encapsulate this code in a small helper function and make the main code in the setShow function more readable.

Build a private helper function called adjust that will take one int argument and returns an int value:

```java
private int adjust(int number){
    if (number < 0)
        number = 0;
    return number;
}
```

Insert this function after the setShow function in the class Show. (ID-Q4)

In the function setShow, replace the statement
```
capacity = seats;
```
with the statement
```
capacity = adjust(seats);
```

Now, delete all four if statements from the setShow function. (ID-Q5) Run the same tests you did before to make sure the input verification works correctly. (ID-Q6)

**The program still has a problem:** The user can request negative number of tickets. Which function(s) would you have to modify to fix this problem? We will show at least one solution in class. (ID-Q7)
Summary of Lab Report Questions

Questions about the `show` class:

SC-1 How many constants are defined in this class?
SC-2 What basic type or class of information do these constants represent?
SC-3 How many member data items are defined in this class?
SC-4 Give the name of the only member data in this class that is not of basic type `int`.
SC-5 How many constructors does this class have?
SC-6 What is the total number of get functions?
SC-7 Why do the `price()` and `sell()` functions return an `int`?
SC-8 Why do the two `available()` functions return a boolean value?
SC-9 Can the user change the value of the `adultsSold`? If yes, how? If not, why not?
SC-10 What do you not understand about this class definition?

Questions about the `TicketSeller` class:

TS-1 How many `Annotation` fields are defined in this class?
TS-2 List the names given to the three `TextFieldViews`.
TS-3 List the label on the seven action buttons.
TS-4 List the four member functions of the class `TicketSeller` that specify what the desired actions should be when selecting the show or deciding or reset the show statistics.
TS-5 List the three member functions of the class `TicketSeller` that specify what the desired actions should be when making a ticket sale.
TS-6 Find the statement that defines the `mainPanel` and copy it into your report. Hint: it is right before the `main` function of the class.

Questions about the actions functions:

AF-1 Choose your favorite movie and decide on the name to be shown on the button and the display:
   Button label: ...................................
   Title shown: .................................
   Now, highlight in the TicketSeller.java all places where the word `tarzan` appears in any form (lower or upper case). Modify the code, so your theater will show your favorite movie in a theater with 400 seats, with prices $20, $10, and $5 respectively for adults, students, and children.
AF-2 Add a `setViewState` member function call that will set the `ViewState` of the `studentsOrderTFV`.
AF-3 Why do we set the `totalOrderField` and `totalPriceField` to two different values? What are these values?
AF-4 See what happens if you do not type anything into the field and leave it blank. Also, try to type in any sequence of characters that are not a number. What happens?
AF-5 Try to type in a simple arithmetic expression, such as 20 - 5 + 6 and see what will happen. Experiment with several expressions and see if you know what is happening.

AF-6 What is the difference between the `price()` member function and `sell()` member function in the class `Show`?

AF-7 Write a function

```java
private void setCurrentShow(Show show)
```

that will perform all the tasks that are repeated in the three show selection actions `tarzan`, `kingKong`, and `casablanca`. These functions will then consist of only one function call to this helper function, for example:

```java
public void tarzan(){
    setCurrentShow(tarzanShow);
}
```

Copy the code for this function into your report.

Questions about helper/utility functions:

HF-1 Update the student price field annotation, following the model for the adult and child price fields.

HF-2 Copy into the report the code for the two helper functions `string(int count)` and `dollar(int count)` and explain what do they do.

HF-3 Add the code to set the correct value in the `totalIncomeField`.

Questions about the integrity of data:

ID-1 Write down the three remaining `if` statements that will adjust the prices into acceptable range.

ID-2 Print the modified `setShow` function once you determined that it works correctly.

ID-3 Write down several constructor calls for the `currentShow` that will test your code and indicate which values are to be corrected by the `setShow` function.

ID-4 Explain what does the function `adjust` do. Show an example of several functions call to the function `adjust` and the effect that the function has. Add a comment before the adjust function that explains its purpose.

ID-5 Why can we delete the four `if` statements. How is the input data corrected now?

ID-6 Print your refactored `setShow` function and compare it with the one you printed earlier. Can you see the difference?

ID-7 Propose a way to make sure that a request for a negative number of tickets in any category is going to be rejected.