

1. Streams of lines

This program uses lambdas and streams to summarize the number of occurrences of each word in a file then display a summary of the words in alphabetical order grouped by starting letter.

a. In the parsing phase:

```
// Regex that matches one or more consecutive whitespace characters
Pattern pattern = Pattern.compile("\\s+");

// count occurrences of each word in a Stream<String> sorted by word
Map<String, Long> wordCounts =
    Files.lines(Paths.get("Chapter2Paragraph.txt"))
        .map(line -> line.replaceAll("(?!')\\p{P}", ""))
        .flatMap(line -> pattern.splitAsStream(line))
        .collect(Collectors.groupingBy(String::toLowerCase,
            TreeMap::new, Collectors.counting()));
```

- Define a pattern that matches one or more consecutive spaces.

```
// Regex that matches one or more consecutive whitespace characters
Pattern pattern = Pattern.compile("\\s+");
```

- Read data from Chapter2Paragraph.txt into stream

```
Files.lines(Paths.get("Chapter2Paragraph.txt"))
```

- Replace all the characters that are punctuations but not “ ’ ” (apostrophe) with empty string

```
.map(line -> line.replaceAll("(?!')\\p{P}", ""))
```

- Split the intermediate stream (including the whole article) into small pieces (words) by using pre-defined pattern

```
.flatMap(line -> pattern.splitAsStream(line))
```

- Reduce the intermediate stream (words) into result map. Use the lowercase of each word as the grouping attribute (key), count the number of words of each group as the value for each group, and put the result into a new TreeMap. Note that the TreeMap will maintain its key in sorted order.

```
.collect(Collectors.groupingBy(String::toLowerCase,  
    TreeMap::new, Collectors.counting()));
```

b. In the display phase

```
// display the words grouped by starting letter  
wordCounts.entrySet()  
    .stream()  
    .collect(  
        Collectors.groupingBy(entry -> entry.getKey().charAt(0),  
            TreeMap::new, Collectors.toList())  
    ).forEach((letter, wordList) ->  
    {  
        System.out.printf("%n%C%n", letter);  
        wordList.stream().forEach(word -> System.out.printf(  
            "%13s: %d%n", word.getKey(), word.getValue()));  
    });
```

- Put all the entries of the result map into stream and use the first character of the word of each entry as grouping attribute, add all the entries of each group into a list and use this list as the “value” of the result map. Put the result into a new TreeMap whose key is the first character of each word and value is the entry list associated with that key.

```
.stream()  
.collect(  
    Collectors.groupingBy(entry -> entry.getKey().charAt(0),  
        TreeMap::new, Collectors.toList())
```

- Traverse the TreeMap: for each loop, first print the key (first letter) of the map, and then print all the corresponding words and the number of their occurrences.

```

.forEach((letter, wordList) -> {
    System.out.printf("%nC%n", letter);
    wordList.stream().forEach(word -> System.out.printf(
        "%13s: %d%n", word.getKey(), word.getValue()));
});

```

c. Final running result:

(The result was too long to show it in one screenshot)

Problems

Search

<terminated> StreamOf

A

a: 2
and: 3
application: 2
arithmetic: 1

B

begin: 1

C

calculates: 1
calculations: 1
chapter: 1
chapters: 1
commandline: 1
compares: 1
comparison: 1
compile: 1
computer: 1

D

decisions: 1
demonstrates: 1
display: 1
displays: 2

E

example: 1
examples: 1

F

for: 1
from: 1

H

how: 2

I

inputs: 1
instruct: 1
introduces: 1

J

java: 1
jdk: 1

L

last: 1
later: 1
learn: 1

M

make: 1
messages: 2

N

numbers: 2

O

obtains: 1
of: 1
on: 1
output: 1

P

perform: 1
present: 1
program: 1
programming: 1
programs: 2

P

perform: 1
present: 1
program: 1
programming: 1
programs: 2

R

result: 1
results: 2
run: 1

S

save: 1
screen: 1
show: 1
sum: 1

T

that: 3
the: 7
their: 2
then: 2
this: 2
to: 4
tools: 1
two: 2

U

use: 2
user: 1

W

we: 2
with: 1

Y

you'll: 2

2. Binary Search

The program implements a binary search method including the implementation of binary search algorithm and corresponding print methods to show the searching process.

- a. In the main method, it first generates an integer array (ranges from 10 to 99) randomly and sorts the array, so that we can use binary search to search an element in that array.

```
public static void main(String[] args)
{
    Scanner input = new Scanner(System.in);
    SecureRandom generator = new SecureRandom();

    int[] data = new int[15]; // create array

    for (int i = 0; i < data.length; i++) // populate array
        data[i] = 10 + generator.nextInt(90);

    Arrays.sort(data); // binarySearch requires sorted array
    System.out.printf("%s\n\n", Arrays.toString(data)); // display array
```

- b. Then it gets searching element from user input and prints the searching result. User can keep searching elements for many times until entering a “-1”.

```
// get input from user
System.out.print("Please enter an integer value (-1 to quit): ");
int searchInt = input.nextInt();

// repeatedly input an integer; -1 terminates the program
while (searchInt != -1)
{
    // perform search
    int location = binarySearch(data, searchInt);

    if (location == -1) // not found
        System.out.printf("%d was not found\n\n", searchInt);
    else // found
        System.out.printf("%d was found in position %d\n\n",
            searchInt, location);

    // get input from user
    System.out.print("Please enter an integer value (-1 to quit): ");
    searchInt = input.nextInt();
}
```

- c. In the binary search method, it receives as parameters the array to search and the searching key.

- It first defines the two searching bounds and calculates the middle position.

```
// perform a binary search on the data
public static int binarySearch(int[] data, int key)
{
    int low = 0; // low end of the search area
    int high = data.length - 1; // high end of the search area
    int middle = (low + high + 1) / 2; // middle element
    int location = -1; // return value; -1 if not found
```

- Use a do - while loop to keep searching until no elements left or the searching element has been found in the array.

For each loop, first print the remaining elements to be searched, and print an indicator ("*") and spaces (for alignment) to show the position of middle element.

```
do // loop to search for element
{
    // print remaining elements of array
    System.out.print(remainingElements(data, low, high));

    // output spaces for alignment
    for (int i = 0; i < middle; i++)
        System.out.print(" ");
    System.out.println(" * "); // indicate current middle

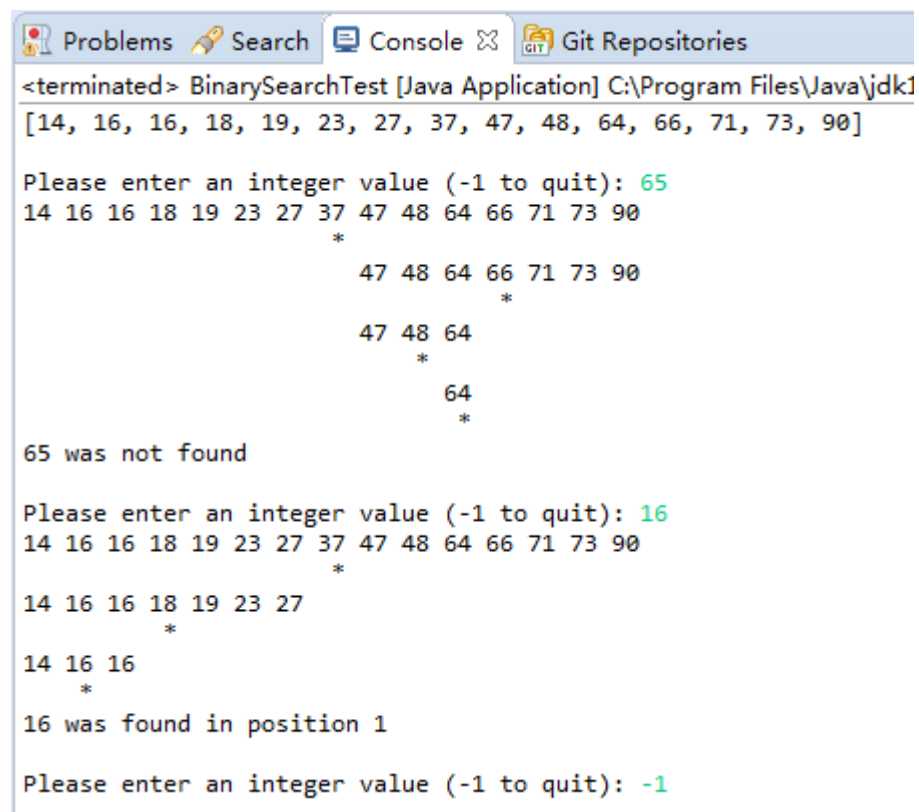
} while ((low <= high) && (location == -1));
```

- Do a binary search algorithm, each time compare the middle element with the searching key, if they are equal, we found the searching key; if the searching key is less than the middle element, there's no way to find the searching key on the right, so we go left (set "high" to middle - 1); otherwise, go right (set "low" to middle + 1).

```
// if the element is found at the middle
if (key == data[middle])
    location = middle; // location is the current middle
else if (key < data[middle]) // middle element is too high
    high = middle - 1; // eliminate the higher half
else // middle element is too low
    low = middle + 1; // eliminate the lower half

middle = (low + high + 1) / 2; // recalculate the middle
```

d. Final running result:



```
<terminated> BinarySearchTest [Java Application] C:\Program Files\Java\jdk1
[14, 16, 16, 18, 19, 23, 27, 37, 47, 48, 64, 66, 71, 73, 90]

Please enter an integer value (-1 to quit): 65
14 16 16 18 19 23 27 37 47 48 64 66 71 73 90
    *
        47 48 64 66 71 73 90
            *
                47 48 64
                    *
                        64
                            *

65 was not found

Please enter an integer value (-1 to quit): 16
14 16 16 18 19 23 27 37 47 48 64 66 71 73 90
    *
        14 16 16
            *

16 was found in position 1

Please enter an integer value (-1 to quit): -1
```

3. Stack Inheritance

“StackInheritance” class shows one way to implement a new data structure (stack) from an old data structure (List), which is called inheritance. “StackInheritanceTest” class builds some test cases to show how “StackInheritance” works

a. StackInheritance

This class extends List, so that it can reuse all the methods (except private methods) of List class, such as isEmpty, print, and the two methods we use below. It uses insertAtFront method and removeFromFront method to implement a stack; it always pushes the new element into the front and removes elements from front, so that the first pushed element will be at the end and will be popped at last.

```
public class StackInheritance<T> extends List<T>
{
    // constructor
    public StackInheritance()
    {
        super("stack");
    }

    // add object to stack
    public void push(T object)
    {
        insertAtFront(object);
    }

    // remove object from stack
    public T pop() throws EmptyListException
    {
        return removeFromFront();
    }
} // end class StackInheritance
```

b. StackInheritanceTest

It first pushes four elements to the stack and prints them (inherit print method from List, which will print the name of the List and all the elements in the List), so that we can use them later on.

```
public static void main(String[] args)
{
    StackInheritance<Integer> stack = new StackInheritance<>();


    // use push method
    stack.push(-1);
    stack.print();
    stack.push(0);
    stack.print();
    stack.push(1);
    stack.print();
    stack.push(5);
    stack.print();
}
```

Then it keeps popping element one by one until the stack is empty. After each popping, it prints the popped element and the remaining stack. When the stack is empty, throws an exception and prints the exception stack trace information.

```
// remove items from stack
try
{
    int removedItem;

    while (true)
    {
        removedItem = stack.pop(); // use pop method
        System.out.printf("%n%d popped%n", removedItem);
        stack.print();
    }
}
catch (EmptyListException emptyListException)
{
    emptyListException.printStackTrace();
}
}
```


c. Final running result:



```
<terminated> StackInheritanceTest [Java Application] C:\Program Files\Java\jdk1.8.0_60\bin\javaw.  
The stack is: -1  
The stack is: 0 -1  
The stack is: 1 0 -1  
The stack is: 5 1 0 -1  
  
5 popped  
The stack is: 1 0 -1  
  
1 popped  
The stack is: 0 -1  
  
0 popped  
The stack is: -1  
  
-1 popped  
Empty stack  
individual.EmptyListException: stack is empty  
    at individual.List.removeFromFront(List.java:81)  
    at individual.StackInheritance.pop(StackInheritance.java:23)  
    at individual.StackInheritanceTest.main(StackInheritanceTest.java:28)
```

4. Shared Buffer

This program shows how an unsynchronized buffer leads to an unpredictable and inaccurate result.

- a. In the unsynchronized buffer, it simply implements an integer value to represent the shared data, and corresponding get and put methods to get, modify and print the value involved.

```
// a producer thread and a consumer thread.  
public class UnsynchronizedBuffer implements Buffer  
{  
    private int buffer = -1; // shared by producer and consumer threads  
  
    // place value into buffer  
    public void blockingPut(int value) throws InterruptedException  
    {  
        System.out.printf("Producer writes\t%2d", value);  
        buffer = value;  
    }  
  
    // return value from buffer  
    public int blockingGet() throws InterruptedException  
    {  
        System.out.printf("Consumer reads\t%2d", buffer);  
        return buffer;  
    }  
} // end class UnsynchronizedBuffer
```

- b. Also define a producer and a consumer for add new value and use current value.

➤ Producer:

It produces numbers from 1 to 10 with a random time interval from 0 to 3 seconds.

After each production, set the producing value to the shared buffer, sum the value produced so far and print corresponding information.

```
// constructor
public Producer(Buffer sharedLocation)
{
    this.sharedLocation = sharedLocation;
}

// store values from 1 to 10 in sharedLocation
public void run()
{
    int sum = 0;

    for (int count = 1; count <= 10; count++)
    {
        try // sleep 0 to 3 seconds, then place value in Buffer
        {
            Thread.sleep(generator.nextInt(3000)); // random sleep
            sharedLocation.blockingPut(count); // set value in buffer
            sum += count; // increment sum of values
            System.out.printf("\t%d\n", sum);
        }
        catch (InterruptedException exception)
        {
            Thread.currentThread().interrupt();
        }
    }
}
```

- Create table header:

```
System.out.println(
    "Action\t\tValue\tSum of Produced\tSum of Consumed");
System.out.printf(
    "-----\t\t-----\t-----\t-----\n\n");
```

- Create and execute a producer thread and a consumer thread.

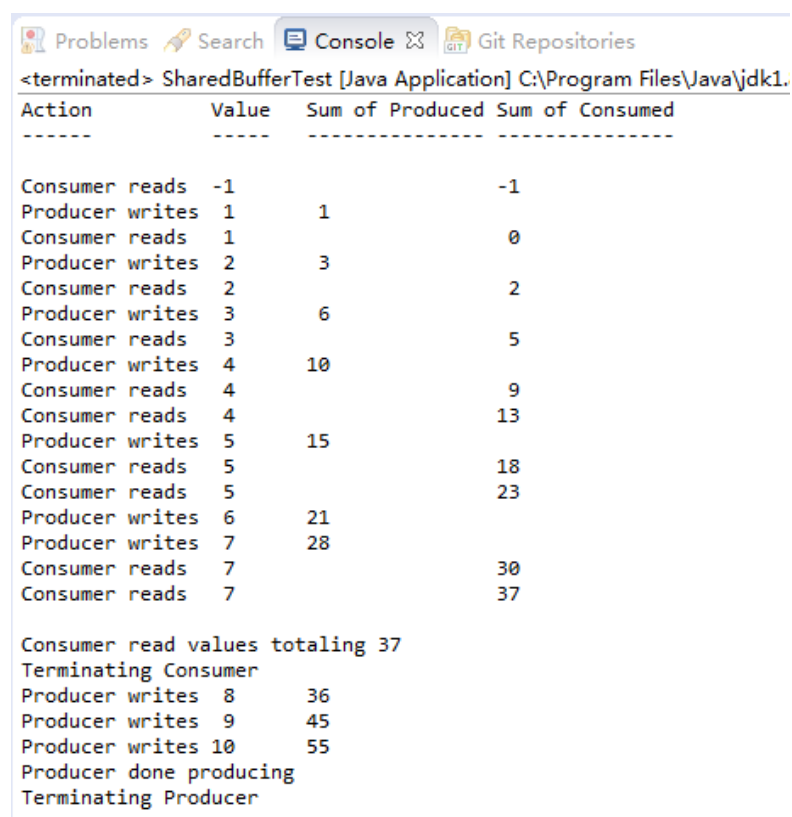
```
// execute the Producer and Consumer, giving each
// access to the sharedLocation
executorService.execute(new Producer(sharedLocation));
executorService.execute(new Consumer(sharedLocation));
```

- Refuse new tasks and terminate the service if all tasks are completed or a times-out occurs (go beyond 1 minute); otherwise, block here to wait for the completion of all tasks.

```
executorService.shutdown(); // terminate app when tasks complete
executorService.awaitTermination(1, TimeUnit.MINUTES);
```

d. Final running result:

We can see that the producing process and consuming process are not done alternatively, which means the consumer may consume the duplicate elements or miss some elements. Also we can see the total values of producing and consuming are not the same. The consumer even starts consuming before producing, although the producing process starts first. All of this kind of chaos is due to using unsynchronized way to access shared data. The access order of different threads will be determined by CPU schedule which is kind of unpredictable for the user.

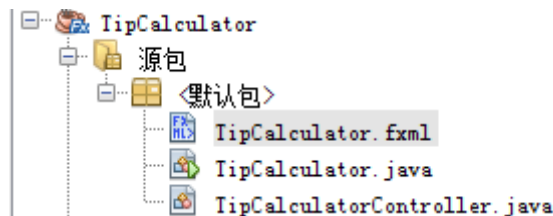


Action	Value	Sum of Produced	Sum of Consumed
Consumer reads	-1		-1
Producer writes	1	1	
Consumer reads	1		0
Producer writes	2	3	
Consumer reads	2		2
Producer writes	3	6	
Consumer reads	3		5
Producer writes	4	10	
Consumer reads	4		9
Consumer reads	4		13
Producer writes	5	15	
Consumer reads	5		18
Consumer reads	5		23
Producer writes	6	21	
Producer writes	7	28	
Consumer reads	7		30
Consumer reads	7		37
Consumer read values totaling 37			
Terminating Consumer			
Producer writes	8	36	
Producer writes	9	45	
Producer writes	10	55	
Producer done producing			
Terminating Producer			

5. Tip Calculator

This program uses JavaFX to implement a tip calculator

- a. GUI part, use Scene Builder to create the GUI and store the GUI configuration in TipCalculator.fxml.



I won't show the creating process here (since this program was done by the author); instead, I will explain the final setting result stored in the setting file.

- Set the GUI's layout to "gridPane", and set the alignment, and gap between each grid. Also assign a controller for this GUI.

```
<GridPane alignment="TOP_LEFT" hgap="8.0" vgap="0.0" xmlns:fx="http://javafx.com/fxml/1" xmlns="http://javafx.com"
fx:controller="TipCalculatorController">
```

- Set labels and corresponding alignment information and position (the index of horizontal grid and vertical grid)

```
<Label text="Amount" textAlignment="RIGHT" GridPane.columnIndex="0" GridPane.rowIndex="0" />
<Label fx:id="tipPercentageLabel" text="15%" textAlignment="RIGHT" GridPane.columnIndex="0" GridPane.rowIndex="1" />
<Label text="Tip" textAlignment="RIGHT" GridPane.columnIndex="0" GridPane.rowIndex="2" />
<Label text="Total" textAlignment="RIGHT" GridPane.columnIndex="0" GridPane.rowIndex="3" />
```

- Set controls for displaying user information, including three textFields and a slider.

Besides location information,

For textFields, set the “tip” and “total” field not editable and not traversable. Set the PrefWidth property to USE_COMPUTED_SIZE (-1) to indicate that the column’s width should be based on the widest child (the Amount Label).

```
<TextField id="amountTextBox" fx:id="amountTextField" prefWidth="-1.0" GridPane.columnIndex="1" GridPane.rowIndex="1"
<Slider fx:id="tipPercentageSlider" blockIncrement="5.0" majorTickUnit="1.0" max="30.0" minorTickCount="1" show
<TextField fx:id="tipTextField" editable="false" focusTraversable="false" prefWidth="-1.0" GridPane.columnIndex="1"
<TextField fx:id="totalTextField" editable="false" focusTraversable="false" prefWidth="-1.0" GridPane.columnIndex="1"/>
```

For the slider, set its default value to 15 and its max value to 30; set tick unit to 1. Do not show the tick marks.

```
<Slider fx:id="tipPercentageSlider" blockIncrement="5.0" majorTickUnit="1.0" max="30.0" minorTickCount="1"
showTickLabels="false" showTickMarks="false" snapToTicks="true" value="15.0" GridPane.columnIndex="1" GridPane.rowIndex="1"/>
```

- Set button and its corresponding action

Set the Max Width property to MAX_VALUE. This causes the Button’s width to grow to fill the column’s width

```
<Button id="calculateButton" maxWidth="MAX_VALUE" text="Calculate" GridPane.columnIndex="1" GridPane.rowIndex="4"/>
```

Set the action to be taken (method name in the controller) when clicking the button and its position.

```
onAction="#calculateButtonPressed" text="Calculate" GridPane.columnIndex="1" GridPane.rowIndex="4"/>
```


- Set padding around the panel, and also set the constraints of each row and column, mainly including alignment, prefer height/width and minimum height/width.

```
<columnConstraints>
  <ColumnConstraints halignment="RIGHT" hgrow="SOMETIMES" minWidth="10.0" prefWidth="1.0" />
  <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="1.0" />
</columnConstraints>
<padding>
  <Insets bottom="14.0" left="14.0" right="14.0" top="14.0" />
</padding>
<rowConstraints>
  <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
  <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
  <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
  <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
  <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
</rowConstraints>
```

b. TipCalculator

- The class's main method calls class Application's static launch method to begin executing a JavaFX app. This method, in turn, causes the JavaFX runtime to create an object of the Application subclass and call its start method.
- This program overrides the start method and adds its own logic: Loads UI settings from TipCalculator.fxml, creates the GUI, attaches it to a Scene and places it on the Stage
- In the load method, it returns a Parent object reference to the scene graph's root node (GridPane), and it creates an object of the controller class (defined in the FXML file) and calls its "initialize" method and registers all the event handlers for any events specified in the FXML.

```
public class TipCalculator extends Application
{
    @Override
    public void start(Stage stage) throws Exception
    {
        Parent root =
            FXMLLoader.load(getClass().getResource("TipCalculator.fxml"));

        Scene scene = new Scene(root); // attach scene graph to scene
        stage.setTitle("Tip Calculator"); // displayed in window's title bar
        stage.setScene(scene); // attach scene to stage
        stage.show(); // display the stage
    }

    public static void main(String[] args)
    {
        // create a TipCalculator object and call its start method
        launch(args);
    }
}
```

c. TipCalculatorController.java

Initializing process called by the FXML Loader.

- Define two formats for currency and percentage.

Use BigDecimal object to store the percentage with a default value 0.15.

```
// formatters for currency and percentages
private static final NumberFormat currency =
    NumberFormat.getCurrencyInstance();
private static final NumberFormat percent =
    NumberFormat.getPercentInstance();

private BigDecimal tipPercentage = new BigDecimal(0.15); // 15% default
```

- Set rounding mode for currency and add a listener for value changing of slider. After each changing, get the new value in the integer format and change it to percentage by dividing 100, format it to percentage format and set the new value to the label.

```
// called by FXMLLoader to initialize the controller
public void initialize()
{
    // 0-4 rounds down, 5-9 rounds up
    currency.setRoundingMode(RoundingMode.HALF_UP);

    // listener for changes to tipPercentageSlider's value
    tipPercentageSlider.valueProperty().addListener(
        new ChangeListener<Number>()
        {
            @Override
            public void changed(ObservableValue<? extends Number> ov,
                                Number oldValue, Number newValue)
            {
                tipPercentage =
                    BigDecimal.valueOf(newValue.intValue() / 100.0);
                tipPercentageLabel.setText(percent.format(tipPercentage));
            }
        }
    );
}
```

- d. Action to be taken when clicking the button. (registered during the initializing process)

Calculate: $\text{Tip} = \text{amount} * \text{tip percentage}.$

$\text{Total} = \text{amount} + \text{tip}$

After calculation, set corresponding text fields.

Note that the value displayed is formatted by currency (we have set its rounding mode)

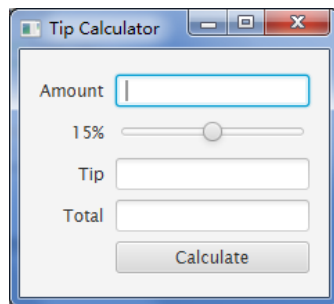
Set selection and focus on the text field and set the value to “Enter amount” when entering a non-numeric value.

```
private void calculateButtonPressed(ActionEvent event)
{
    try
    {
        BigDecimal amount = new BigDecimal(amountTextField.getText());
        BigDecimal tip = amount.multiply(tipPercentage);
        BigDecimal total = amount.add(tip);

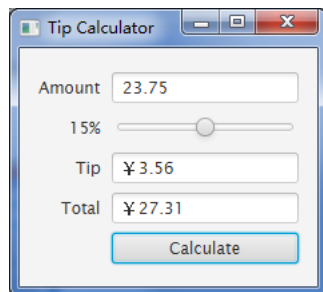
        tipTextField.setText(currency.format(tip));
        totalTextField.setText(currency.format(total));
    }
    catch (NumberFormatException ex)
    {
        amountTextField.setText("Enter amount");
        amountTextField.selectAll();
        amountTextField.requestFocus();
    }
}
```

e. Final running result:

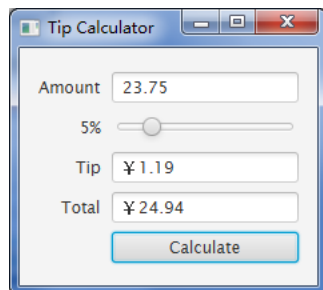
Start:



Input an amount and click “calculate”:



Drag the slider, and click the button:



Enter an invalid value:

