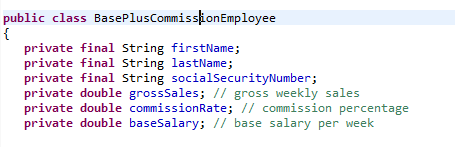
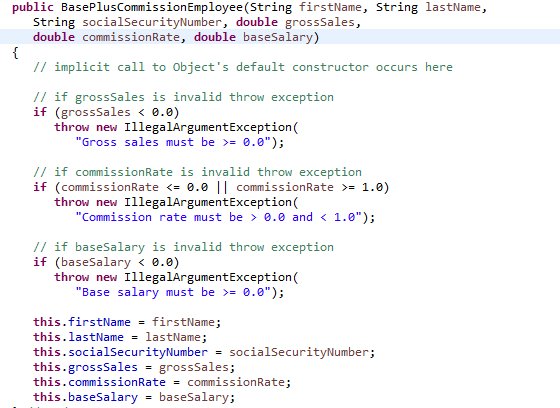
1. BasePlusCommissionEmployee

This program implements a special kind of commission employee, which just adds a base salary to the commission employee. Although it’s quite similar to a commission employee, this program still uses a brand new definition here.

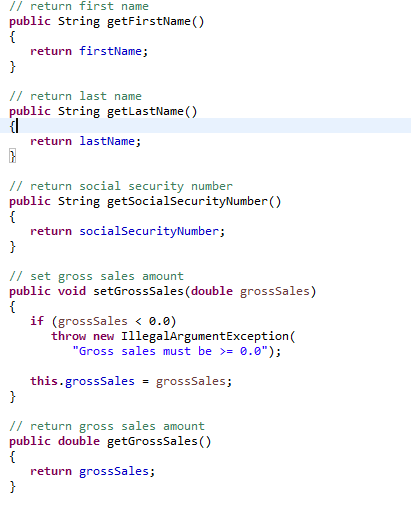
1. It create a new class and defines all the attributes including the name of employee, ssn, gross weekly sales, commission percentage, base salary per week

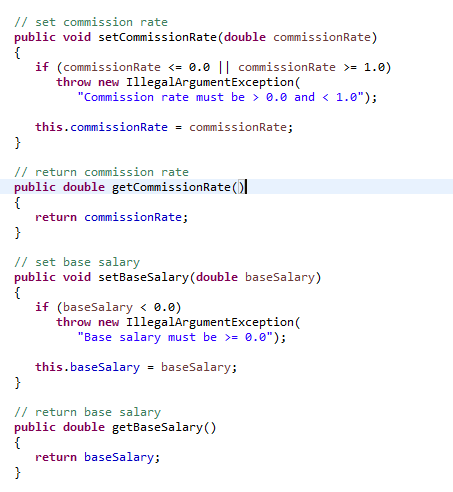


1. For constructor, it sets the value of all the attributes and also do validation for the input parameters

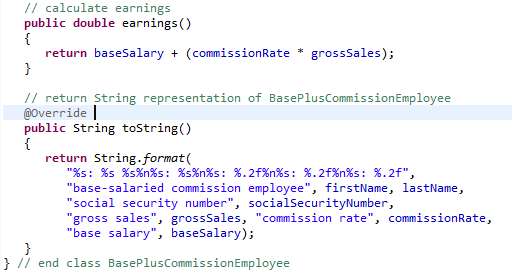


1. Corresponding get and set methods. Note that the set methods also need do validation.

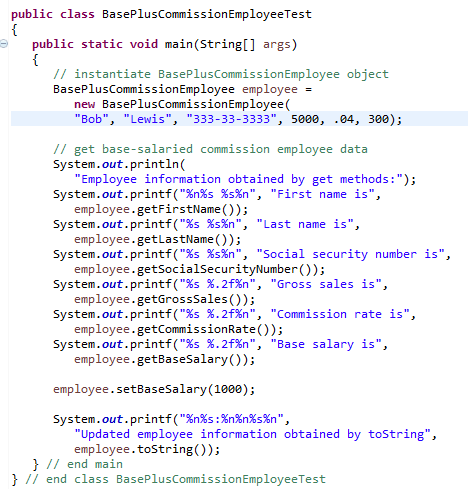




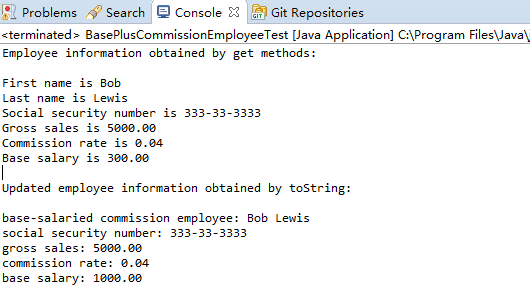
1. For the earning part, just use the equation: earning = base Salary + commission Rate \* gross Sales. Also implement the string representation of this employee: add all the information to the string representation and use format specifier %.2f to format the gross sales, commission rate and base salary with two digits of precision to the right of the decimal point



1. For the test class: first it initiates a new Base plus Commission Employee, and then calls all of its get methods separately and prints the results. Then it changes the base salary and print the string representation of this employee.



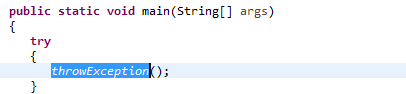
1. Final running result:



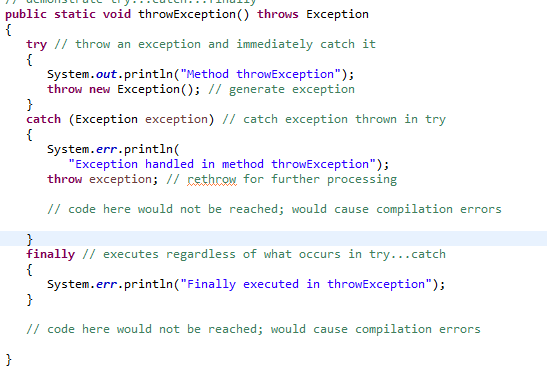
1. Using Exception

This program shows us how to use Exception in java.

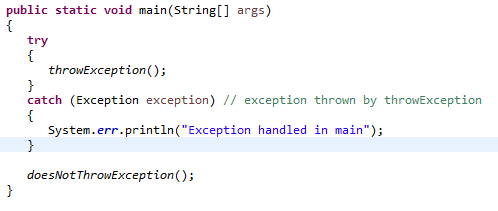
1. It will call throw exception method first



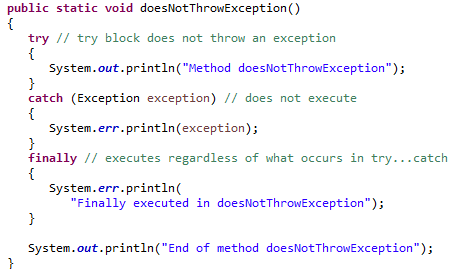
1. In this method, it first prints some prompt and then throw a new exception. This exception will be caught by its own catch function. Then in the catch block, it will print some prompt and then throw that exception again, which will be thrown out of this method. Finally, it goes into finally block and print other prompt.



1. In the main method, the exception that was thrown by the “throwException” method will be caught by the catch function in the main method and then print some prompt and goes into another method ”doesNotThrowException”.

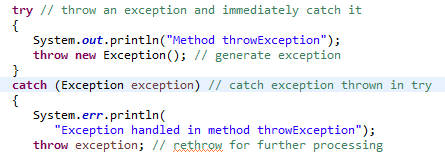


1. In this method, it first throws an exception and then catches it by itself. Since it does not throw that exception again, it won’t throw an exception out of this method. In the finally block it prints some prompt, and after finishing try block, it prints another prompt.

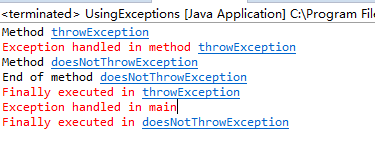


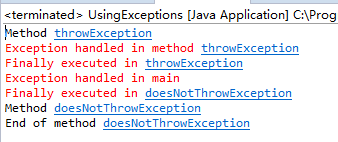
1. Explanation for print information:

For exception handle information, the program uses standard error stream “system.err.println”; for pure prompt, the program use standard output stream “system.out.println”.



1. Final running result. The usage of the two output streams also makes the printing result unstable. Since the two output streams are independent and asynchronous with each other, the order of output within one stream is stable (synchronous), but the order between the two streams will be changed each time depending on the scheduling of CPU.

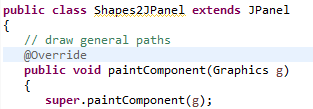




1. Shapes2JPanel

This problem will draw a beautiful “garland” with random colors.

1. At first, it overrides the paintComponent method in order to draw more graphics on the panel.



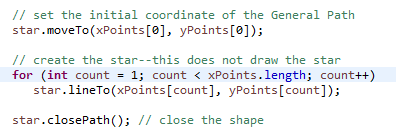
1. Use Graphics2D object to draw, use GeneralPath object to represent the “star” to be drawn



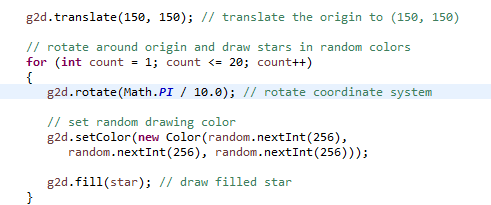
1. Declare two integer arrays representing the x- and y-coordinates of the points in the star.



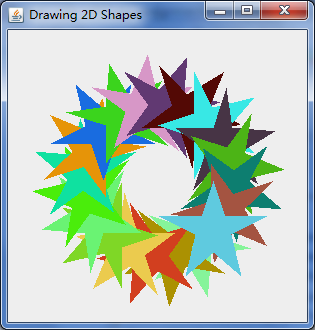
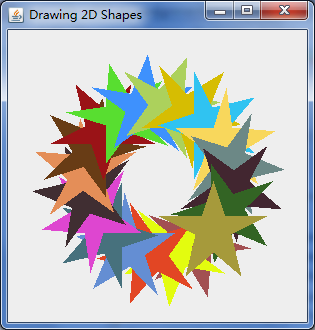
1. Before drawing, set the first point of the star and then use a for loop to assign the coordinates of all the other points of the start and draw a line from the last point to the new specified coordinates and at last, close the path by drawing a line from the last point to the point specified in the last call to “moveTo”



1. In the drawing phase, first we move the star to a suitable position, so that the whole shape won’t get out of our panel. Then use a “for” loop to rotate the star and fill the star with random color. For each rotate, we rotate points on the positive x axis toward the positive y axis by 36 degree(π / 10).



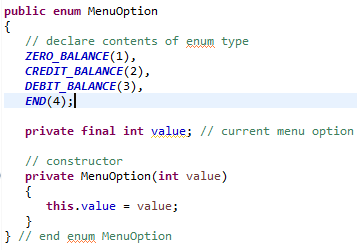
1. Final running result: the color will be changed each time.



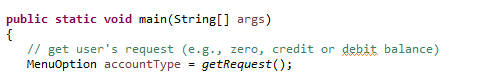
1. Credit Inquiry

This program is a simple credit inquiry system. The record information is collected by reading through the file and determining if each record satisfies the criteria for the selected account type. It also includes a simple text menu.

1. First, define an enum for all possible menu types including the type name and corresponding value.

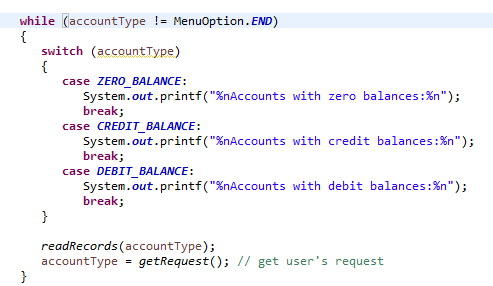


1. At first, in main method, it calls method getRequest to display the menu options. After user has entered a valid option, translates the number typed by the user into a MenuOption and stores the result in MenuOption variable accountType. Note that if the user enters an invalid numeric option, it will ask the user to reenter, if user enters a non-numeric option, it will terminate.

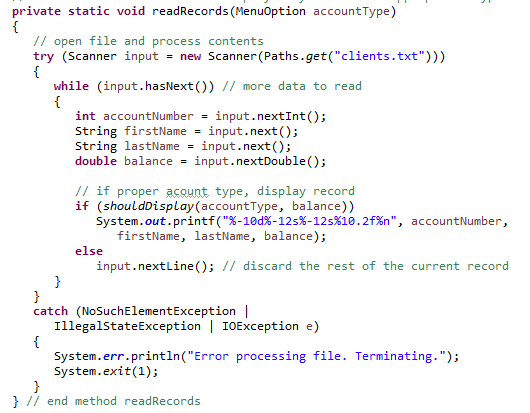




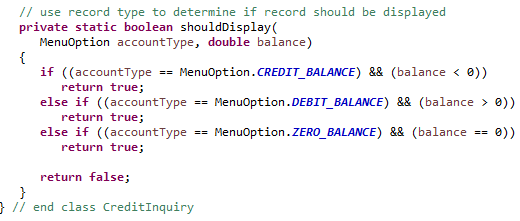
1. Keep inquiring until get an option “end”. For each inquiring, print corresponding prompt and read corresponding records and then request a new user input.



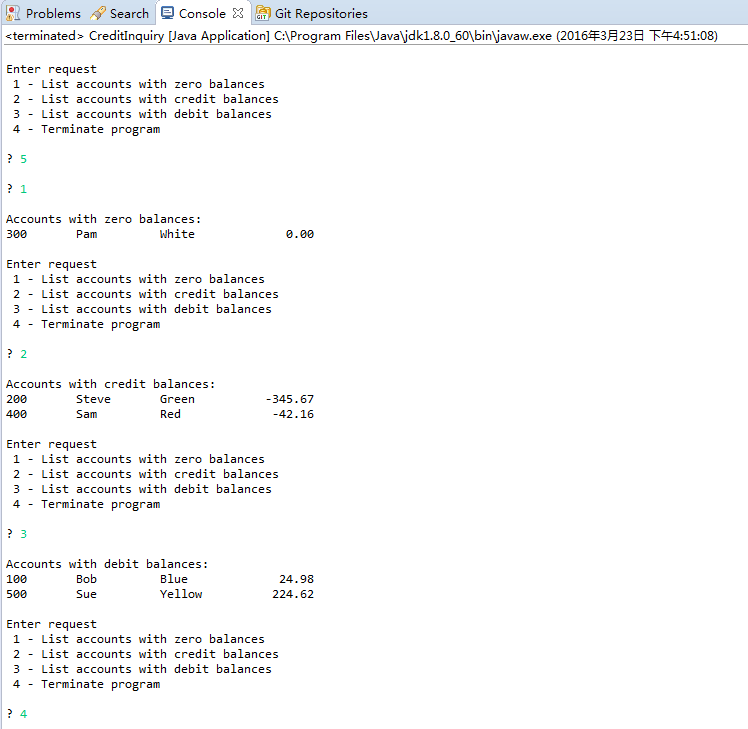
1. For reading records, first get the client file”clients.txt” from file system, for each result, check whether the record satisfy the criteria and should be displayed. If cannot open the file, it will terminate.



1. Check whether each record satisfy the criteria, depending on option type entered by user and balance.



1. Final running result

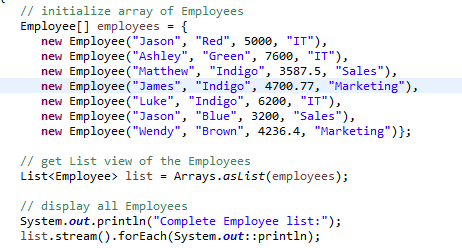


1. Processing Employees

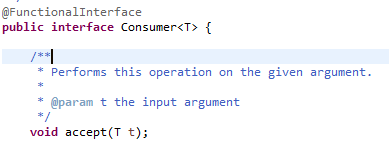
This program is to process employees by using stream and lambda expression including creating, displaying, filtering, sorting, mapping, grouping, counting and summing employees.

1. Creating and displaying

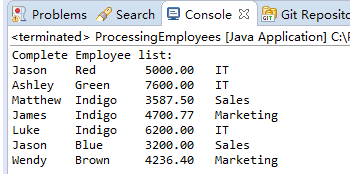
Create employee array and convert it into list, use list.stream() to creates a Stream<Employee>, then uses Stream method forEach to display each Employee’s String representation.



The instance method reference System.out::println is converted by the compiler into an object that implements the Consumer functional interface, the “accept” method passes each Employee to the System.out object’s println method and then print the String representation of each employee.



Partial output:

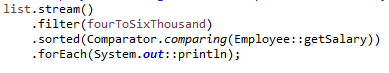


1. Filtering Employees with Salaries in a Specified Range

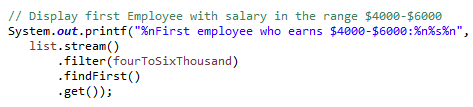
First, implements the functional interface Predicate<Employee> to represent the filtering condition so that this condition can be reused easily.



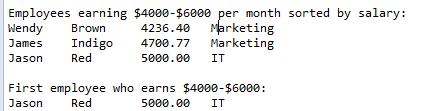
Then, use this condition to filter stream and sort the result by calling sorted method with a comparator implementation. The comparator implementation calls comparing method which uses the method reference Employee::getSalary to extract a value from an object in the stream for use in comparisons.



Also get the first match value in the filtering result.

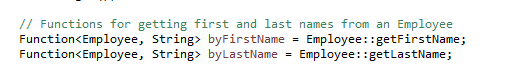


Partial output:

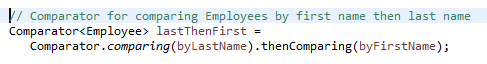


1. Sorting Employees By Multiple Fields

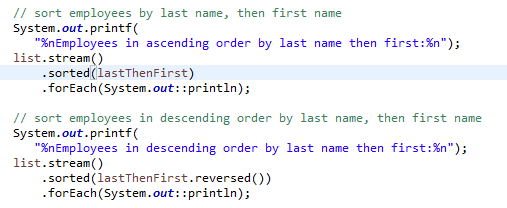
For convenience, first define two function objects reference the corresponding method in employee.



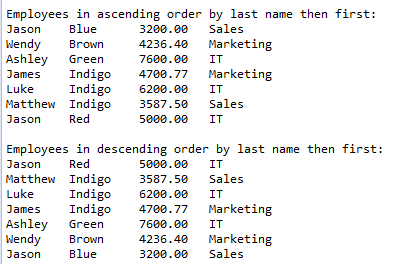
Define a comparator for sorting, first compare the last name, if last names are the same, compares the first name.



Sort employees in ascending order by using the predefined comparator, and use reversed method to get descending order.



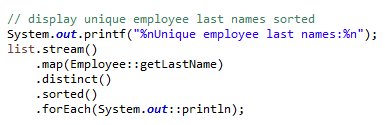
Partial output



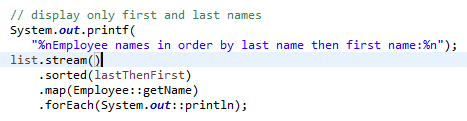
1. Mapping Employees to Unique Last Name Strings

First, map the employee to its last Name by making a reference to the method ”getLastName”.

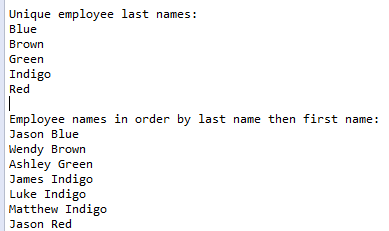
Call distinct method to get unique last name and then call sort method to sort the result, and then print the sorted result.



First sort the stream by predefined comparator, then map the employee to its first name and last name by making a reference to the method “getName”, and then print the sorted result.



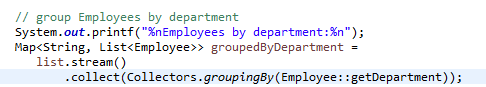
Partial output



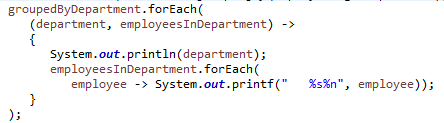
1. Grouping Employees By Department

It uses Stream method “collect” to group Employees by department. It uses groupingBy method to group and reference a getDepartment method to specify the department is the grouping variable.

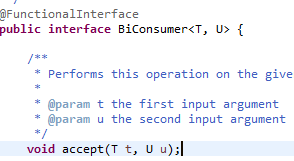
The result is a Map<String, List<Employee>> in which each String key is a department and each List<Employee> contains the Employees in that department.



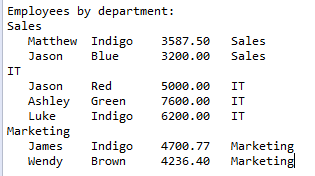
After getting the intermediate result, for each key-value pair, it prints the department and calls another foreach method to print all the employees who belong to that department.



Map method forEach performs an operation on each of the Map’s key–value pairs. The argument to the method is an object that implements functional interface BiConsumer.

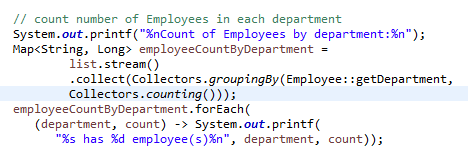


Partial output

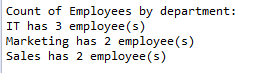


1. Counting the Number of Employees in Each Department

This time the collect method produces a Map<String, Long> in which each String key is a department name and the corresponding Long value is the number of Employees in that department. In this case, we use a call to Collectors static method counting as the second argument. This method returns a Collector that counts the number of objects in a given classification, rather than collecting them into a List

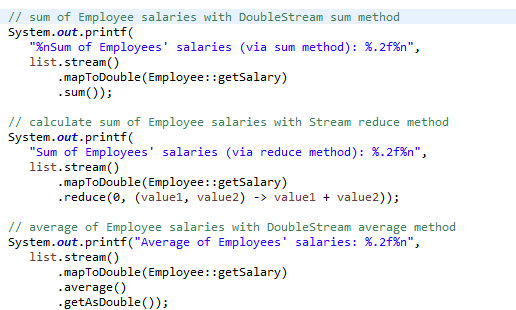


Partial output



1. Summing and Averaging Employee Salaries

It maps Employee objects to their salaries so that we can calculate the sum and average. For the second sum calculation, it uses Lambdas expression to specify the process (has a base of 0, each time add a new value to it).



Partial output



1. The whole final running result

