

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.

- a. 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

```
public class BasePlusCommissionEmployee
{
    private final String firstName;
    private final String lastName;
    private final String socialSecurityNumber;
    private double grossSales; // gross weekly sales
    private double commissionRate; // commission percentage
    private double baseSalary; // base salary per week
}
```

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

```
public BasePlusCommissionEmployee(String firstName, String lastName,
    String socialSecurityNumber, double grossSales,
    double commissionRate, double baseSalary)
{
    // implicit call to Object's default constructor occurs here

    // if grossSales is invalid throw exception
    if (grossSales < 0.0)
        throw new IllegalArgumentException(
            "Gross sales must be >= 0.0");

    // if commissionRate is invalid throw exception
    if (commissionRate <= 0.0 || commissionRate >= 1.0)
        throw new IllegalArgumentException(
            "Commission rate must be > 0.0 and < 1.0");

    // if baseSalary is invalid throw exception
    if (baseSalary < 0.0)
        throw new IllegalArgumentException(
            "Base salary must be >= 0.0");

    this.firstName = firstName;
    this.lastName = lastName;
    this.socialSecurityNumber = socialSecurityNumber;
    this.grossSales = grossSales;
    this.commissionRate = commissionRate;
    this.baseSalary = baseSalary;
}
```

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

```
// return first name
public String getFirstName()
{
    return firstName;
}

// return last name
public String getLastName()
{
    return lastName;
}

// return social security number
public String getSocialSecurityNumber()
{
    return socialSecurityNumber;
}

// set gross sales amount
public void setGrossSales(double grossSales)
{
    if (grossSales < 0.0)
        throw new IllegalArgumentException(
            "Gross sales must be >= 0.0");

    this.grossSales = grossSales;
}

// return gross sales amount
public double getGrossSales()
{
    return grossSales;
}

// set commission rate
public void setCommissionRate(double commissionRate)
{
    if (commissionRate <= 0.0 || commissionRate >= 1.0)
        throw new IllegalArgumentException(
            "Commission rate must be > 0.0 and < 1.0");

    this.commissionRate = commissionRate;
}

// return commission rate
public double getCommissionRate()
{
    return commissionRate;
}

// set base salary
public void setBaseSalary(double baseSalary)
{
    if (baseSalary < 0.0)
        throw new IllegalArgumentException(
            "Base salary must be >= 0.0");

    this.baseSalary = baseSalary;
}

// return base salary
public double getBaseSalary()
{
    return baseSalary;
}
```

- d. 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

```
// calculate earnings
public double earnings()
{
    return baseSalary + (commissionRate * grossSales);
}

// return String representation of BasePlusCommissionEmployee
@Override
public String toString()
{
    return String.format(
        "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
        "base-salaried commission employee", firstName, lastName,
        "social security number", socialSecurityNumber,
        "gross sales", grossSales, "commission rate", commissionRate,
        "base salary", baseSalary);
}
} // end class BasePlusCommissionEmployee
```

- e. 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.

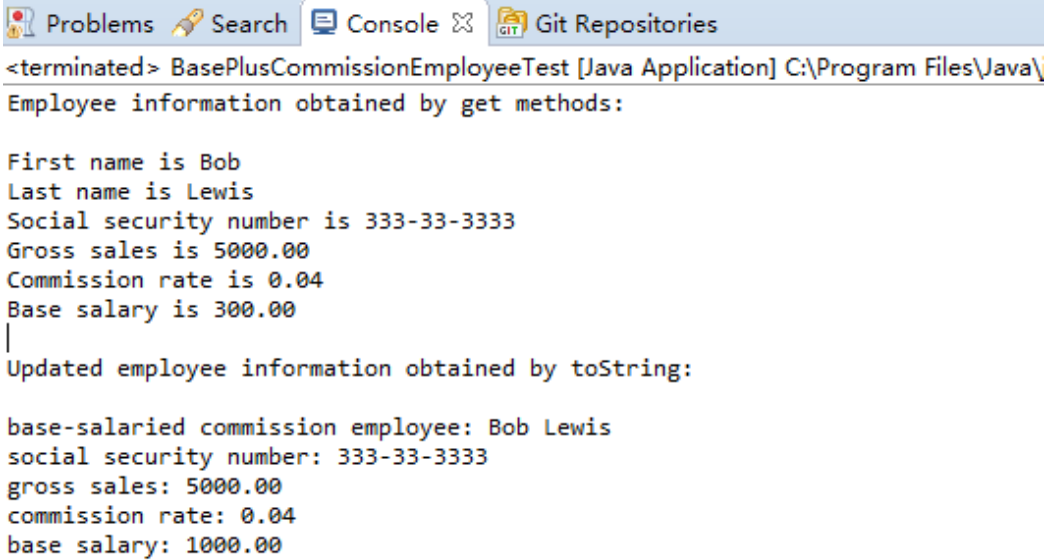
```
public class BasePlusCommissionEmployeeTest
{
    public static void main(String[] args)
    {
        // instantiate BasePlusCommissionEmployee object
        BasePlusCommissionEmployee employee =
            new BasePlusCommissionEmployee(
                "Bob", "Lewis", "333-33-3333", 5000, .04, 300);

        // get base-salaried commission employee data
        System.out.println(
            "Employee information obtained by get methods:");
        System.out.printf("%n%s %s\n", "First name is",
            employee.getFirstName());
        System.out.printf("%s %s\n", "Last name is",
            employee.getLastName());
        System.out.printf("%s %s\n", "Social security number is",
            employee.getSocialSecurityNumber());
        System.out.printf("%s %.2f\n", "Gross sales is",
            employee.getGrossSales());
        System.out.printf("%s %.2f\n", "Commission rate is",
            employee.getCommissionRate());
        System.out.printf("%s %.2f\n", "Base salary is",
            employee.getBaseSalary());

        employee.setBaseSalary(1000);

        System.out.printf("%n%s:%n%s\n",
            "Updated employee information obtained by toString",
            employee.toString());
    } // end main
} // end class BasePlusCommissionEmployeeTest
```

f. Final running result:



```
<terminated> BasePlusCommissionEmployeeTest [Java Application] C:\Program Files\Java\
Employee information obtained by get methods:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00
|
Updated employee information obtained by toString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00
```

2. Using Exception

This program shows us how to use Exception in java.

- a. It will call throw exception method first

```
public static void main(String[] args)
{
    try
    {
        throwException();
    }
}
```

- b. 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.

```
// demonstrate try...catch...finally
public static void throwException() throws Exception
{
    try // throw an exception and immediately catch it
    {
        System.out.println("Method throwException");
        throw new Exception(); // generate exception
    }
    catch (Exception exception) // catch exception thrown in try
    {
        System.err.println(
            "Exception handled in method throwException");
        throw exception; // rethrow for further processing

        // code here would not be reached; would cause compilation errors
    }
    finally // executes regardless of what occurs in try...catch
    {
        System.err.println("Finally executed in throwException");
    }

    // code here would not be reached; would cause compilation errors
}
```

- c. 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".

```
public static void main(String[] args)
{
    try
    {
        throwException();
    }
    catch (Exception exception) // exception thrown by throwException
    {
        System.err.println("Exception handled in main");
    }

    doesNotThrowException();
}
```

- d. 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.

```
public static void doesNotThrowException()
{
    try // try block does not throw an exception
    {
        System.out.println("Method doesNotThrowException");
    }
    catch (Exception exception) // does not execute
    {
        System.err.println(exception);
    }
    finally // executes regardless of what occurs in try...catch
    {
        System.err.println(
            "Finally executed in doesNotThrowException");
    }

    System.out.println("End of method doesNotThrowException");
}
```

- e. 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".

```
try // throw an exception and immediately catch it
{
    System.out.println("Method throwException");
    throw new Exception(); // generate exception
}
catch (Exception exception) // catch exception thrown in try
{
    System.err.println(
        "Exception handled in method throwException");
    throw exception; // rethrow for further processing
}
```

- f. 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.

```
<terminated> UsingExceptions [Java Application] C:\Program File
Method throwException
Exception handled in method throwException
Method doesNotThrowException
End of method doesNotThrowException
Finally executed in throwException
Exception handled in main
Finally executed in doesNotThrowException
```

```
<terminated> UsingExceptions [Java Application] C:\Prog
Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Finally executed in doesNotThrowException
Method doesNotThrowException
End of method doesNotThrowException
```

3. Shapes2JPanel

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

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

```
public class Shapes2JPanel extends JPanel
{
    // draw general paths
    @Override
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
```

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

```
Graphics2D g2d = (Graphics2D) g;
GeneralPath star = new GeneralPath(); // create GeneralPath object
```

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

```
int[] xPoints = {55, 67, 109, 73, 83, 55, 27, 37, 1, 43};
int[] yPoints = {0, 36, 36, 54, 96, 72, 96, 54, 36, 36};
```

- d. 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 star 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”


```
// set the initial coordinate of the General Path
star.moveTo(xPoints[0], yPoints[0]);

// create the star--this does not draw the star
for (int count = 1; count < xPoints.length; count++)
    star.lineTo(xPoints[count], yPoints[count]);

star.closePath(); // close the shape
```

- e. 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($\pi / 10$).

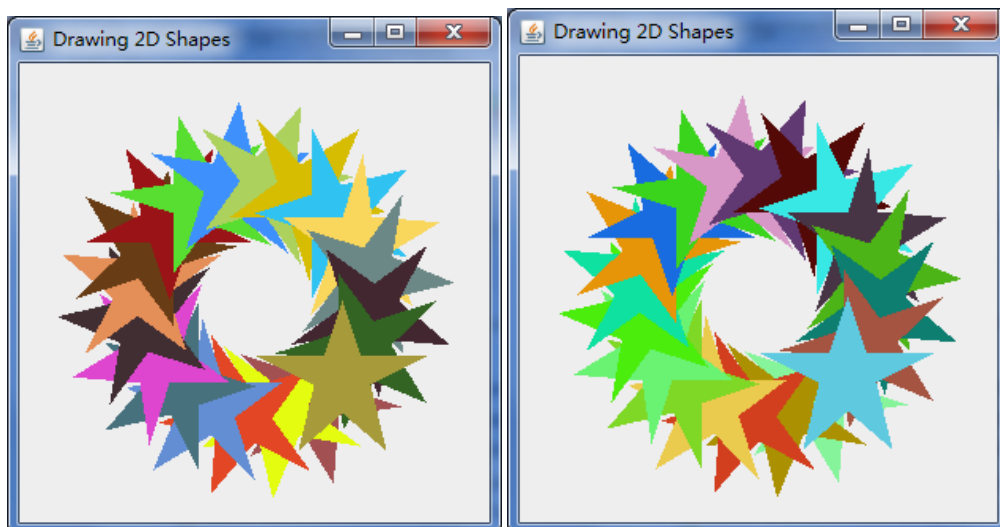
```
g2d.translate(150, 150); // translate the origin to (150, 150)

// rotate around origin and draw stars in random colors
for (int count = 1; count <= 20; count++)
{
    g2d.rotate(Math.PI / 10.0); // rotate coordinate system

    // set random drawing color
    g2d.setColor(new Color(random.nextInt(256),
        random.nextInt(256), random.nextInt(256)));

    g2d.fill(star); // draw filled star
}
```

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



4. 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.

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

```
public enum MenuOption
{
    // declare contents of enum type
    ZERO_BALANCE(1),
    CREDIT_BALANCE(2),
    DEBIT_BALANCE(3),
    END(4);

    private final int value; // current menu option

    // constructor
    private MenuOption(int value)
    {
        this.value = value;
    }
} // end enum MenuOption
```

- b. 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.

```
public static void main(String[] args)
{
    // get user's request (e.g., zero, credit or debit balance)
    MenuOption accountType = getRequest();
}
```

```
// obtain request from user
private static MenuOption getRequest()
{
    int request = 4;

    // display request options
    System.out.printf("%nEnter request\n%s\n%s\n%s\n%s\n",
        " 1 - List accounts with zero balances",
        " 2 - List accounts with credit balances",
        " 3 - List accounts with debit balances",
        " 4 - Terminate program");

    try
    {
        Scanner input = new Scanner(System.in);

        do // input user request
        {
            System.out.printf("%n? ");
            request = input.nextInt();
        } while ((request < 1) || (request > 4));
    }
    catch (NoSuchElementException noSuchElementException)
    {
        System.err.println("Invalid input. Terminating.");
    }

    return choices[request - 1]; // return enum value for option
}
```

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

```
while (accountType != MenuOption.END)
{
    switch (accountType)
    {
        case ZERO_BALANCE:
            System.out.printf("%nAccounts with zero balances:%n");
            break;
        case CREDIT_BALANCE:
            System.out.printf("%nAccounts with credit balances:%n");
            break;
        case DEBIT_BALANCE:
            System.out.printf("%nAccounts with debit balances:%n");
            break;
    }

    readRecords(accountType);
    accountType = getRequest(); // get user's request
}
```

- d. 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.

```

private static void readRecords(MenuOption accountType)
{
    // open file and process contents
    try (Scanner input = new Scanner(Paths.get("clients.txt")))
    {
        while (input.hasNext()) // more data to read
        {
            int accountNumber = input.nextInt();
            String firstName = input.next();
            String lastName = input.next();
            double balance = input.nextDouble();

            // if proper account type, display record
            if (shouldDisplay(accountType, balance))
                System.out.printf("%-10d%-12s%-12s%10.2f\n", accountNumber,
                    firstName, lastName, balance);
            else
                input.nextLine(); // discard the rest of the current record
        }
    }
    catch (NoSuchElementException |
        IllegalStateException | IOException e)
    {
        System.err.println("Error processing file. Terminating.");
        System.exit(1);
    }
} // end method readRecords

```

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

```

// use record type to determine if record should be displayed
private static boolean shouldDisplay(
    MenuOption accountType, double balance)
{
    if ((accountType == MenuOption.CREDIT_BALANCE) && (balance < 0))
        return true;
    else if ((accountType == MenuOption.DEBIT_BALANCE) && (balance > 0))
        return true;
    else if ((accountType == MenuOption.ZERO_BALANCE) && (balance == 0))
        return true;

    return false;
}
} // end class CreditInquiry

```

- f. Final running result

```
Problems Search Console Git Repositories
<terminated> CreditInquiry [Java Application] C:\Program Files\Java\jdk1.8.0_60\bin\javaw.exe (2016年3月23日 下午4:51:08)

Enter request
1 - List accounts with zero balances
2 - List accounts with credit balances
3 - List accounts with debit balances
4 - Terminate program

? 5

? 1

Accounts with zero balances:
300    Pam        White        0.00

Enter request
1 - List accounts with zero balances
2 - List accounts with credit balances
3 - List accounts with debit balances
4 - Terminate program

? 2

Accounts with credit balances:
200    Steve     Green       -345.67
400    Sam        Red         -42.16

Enter request
1 - List accounts with zero balances
2 - List accounts with credit balances
3 - List accounts with debit balances
4 - Terminate program

? 3

Accounts with debit balances:
100    Bob        Blue        24.98
500    Sue        Yellow      224.62

Enter request
1 - List accounts with zero balances
2 - List accounts with credit balances
3 - List accounts with debit balances
4 - Terminate program

? 4
```

5. 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.

a. Creating and displaying

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

```
// initialize array of Employees
Employee[] employees = {
    new Employee("Jason", "Red", 5000, "IT"),
    new Employee("Ashley", "Green", 7600, "IT"),
    new Employee("Matthew", "Indigo", 3587.5, "Sales"),
    new Employee("James", "Indigo", 4700.77, "Marketing"),
    new Employee("Luke", "Indigo", 6200, "IT"),
    new Employee("Jason", "Blue", 3200, "Sales"),
    new Employee("Wendy", "Brown", 4236.4, "Marketing")};

// get List view of the Employees
List<Employee> list = Arrays.asList(employees);

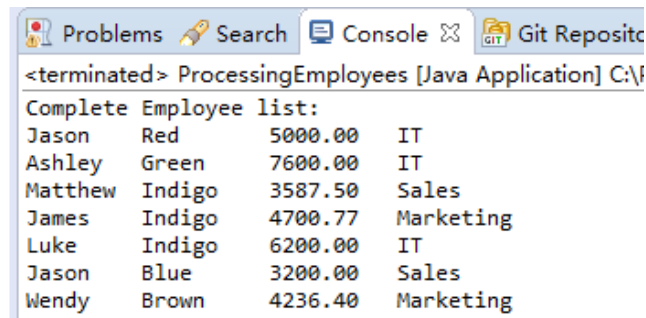
// display all Employees
System.out.println("Complete Employee list:");
list.stream().forEach(System.out::println);
```

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.

```
@FunctionalInterface
public interface Consumer<T> {

    /**
     * Performs this operation on the given argument.
     *
     * @param t the input argument
     */
    void accept(T t);
}
```

Partial output:



```
<terminated> ProcessingEmployees [Java Application] C:\V
Complete Employee list:
Jason    Red      5000.00  IT
Ashley   Green    7600.00  IT
Matthew  Indigo   3587.50  Sales
James    Indigo   4700.77  Marketing
Luke     Indigo   6200.00  IT
Jason    Blue     3200.00  Sales
Wendy    Brown    4236.40  Marketing
```

b. 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.

```
// Predicate that returns true for salaries in the range $4000-$6000
Predicate<Employee> fourToSixThousand =
    e -> (e.getSalary() >= 4000 && e.getSalary() <= 6000);
```

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.

```
list.stream()
    .filter(fourToSixThousand)
    .sorted(Comparator.comparing(Employee::getSalary))
    .forEach(System.out::println);
```

Also get the first match value in the filtering result.

```
// Display first Employee with salary in the range $4000-$6000
System.out.printf("%nFirst employee who earns $4000-$6000:%n%s%n",
    list.stream()
        .filter(fourToSixThousand)
        .findFirst()
        .get());
```

Partial output:

```
Employees earning $4000-$6000 per month sorted by salary:
Wendy   Brown   4236.40   Marketing
James   Indigo   4700.77   Marketing
Jason   Red       5000.00   IT

First employee who earns $4000-$6000:
Jason   Red       5000.00   IT
```

c. Sorting Employees By Multiple Fields

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

```
// Functions for getting first and last names from an Employee
Function<Employee, String> byFirstName = Employee::getFirstName;
Function<Employee, String> byLastName = Employee::getLastName;
```

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

```
// Comparator for comparing Employees by first name then last name
Comparator<Employee> lastThenFirst =
    Comparator.comparing(byLastName).thenComparing(byFirstName);
```

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


```
// sort employees by last name, then first name
System.out.printf(
    "%nEmployees in ascending order by last name then first:%n");
list.stream()
    .sorted(lastThenFirst)
    .forEach(System.out::println);

// sort employees in descending order by last name, then first name
System.out.printf(
    "%nEmployees in descending order by last name then first:%n");
list.stream()
    .sorted(lastThenFirst.reversed())
    .forEach(System.out::println);
```

Partial output

```
Employees in ascending order by last name then first:
Jason   Blue   3200.00   Sales
Wendy   Brown   4236.40   Marketing
Ashley   Green   7600.00   IT
James   Indigo   4700.77   Marketing
Luke     Indigo   6200.00   IT
Matthew Indigo   3587.50   Sales
Jason    Red    5000.00   IT

Employees in descending order by last name then first:
Jason    Red    5000.00   IT
Matthew Indigo   3587.50   Sales
Luke     Indigo   6200.00   IT
James   Indigo   4700.77   Marketing
Ashley   Green   7600.00   IT
Wendy    Brown   4236.40   Marketing
Jason    Blue   3200.00   Sales
```

d. 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.

```
// display unique employee last names sorted
System.out.printf("%nUnique employee last names:%n");
list.stream()
    .map(Employee::getLastName)
    .distinct()
    .sorted()
    .forEach(System.out::println);
```

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.

```
// display only first and last names
System.out.printf(
    "%nEmployee names in order by last name then first name:%n");
list.stream()
    .sorted(lastThenFirst)
    .map(Employee::getName)
    .forEach(System.out::println);
```

Partial output

```
Unique employee last names:
Blue
Brown
Green
Indigo
Red
|
Employee names in order by last name then first name:
Jason Blue
Wendy Brown
Ashley Green
James Indigo
Luke Indigo
Matthew Indigo
Jason Red
```

e. 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.

```
// group Employees by department
System.out.printf("%nEmployees by department:%n");
Map<String, List<Employee>> groupedByDepartment =
    list.stream()
        .collect(Collectors.groupingBy(Employee::getDepartment));
```

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.

```
groupedByDepartment.forEach(  
    (department, employeesInDepartment) ->  
    {  
        System.out.println(department);  
        employeesInDepartment.forEach(  
            employee -> System.out.printf("  %s%n", employee));  
    }  
);
```

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`.

```
@FunctionalInterface  
public interface BiConsumer<T, U> {  
  
    /**  
     * Performs this operation on the give  
     *  
     * @param t the first input argument  
     * @param u the second input argument  
     */  
    void accept(T t, U u);|
```

Partial output

```
Employees by department:  
Sales  
  Matthew Indigo    3587.50  Sales  
  Jason   Blue     3200.00  Sales  
IT  
  Jason   Red       5000.00  IT  
  Ashley  Green      7600.00  IT  
  Luke    Indigo     6200.00  IT  
Marketing  
  James   Indigo     4700.77  Marketing  
  Wendy   Brown      4236.40  Marketing
```

f. 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

```
// count number of Employees in each department
System.out.printf("%nCount of Employees by department:%n");
Map<String, Long> employeeCountByDepartment =
    list.stream()
        .collect(Collectors.groupingBy(Employee::getDepartment,
            Collectors.counting()));
employeeCountByDepartment.forEach(
    (department, count) -> System.out.printf(
        "%s has %d employee(s)%n", department, count));
```

Partial output

```
Count of Employees by department:
IT has 3 employee(s)
Marketing has 2 employee(s)
Sales has 2 employee(s)
```

g. 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).

```
// sum of Employee salaries with DoubleStream sum method
System.out.printf(
    "%nSum of Employees' salaries (via sum method): %.2f%n",
    list.stream()
        .mapToDouble(Employee::getSalary)
        .sum());

// calculate sum of Employee salaries with Stream reduce method
System.out.printf(
    "%nSum of Employees' salaries (via reduce method): %.2f%n",
    list.stream()
        .mapToDouble(Employee::getSalary)
        .reduce(0, (value1, value2) -> value1 + value2));

// average of Employee salaries with DoubleStream average method
System.out.printf("Average of Employees' salaries: %.2f%n",
    list.stream()
        .mapToDouble(Employee::getSalary)
        .average()
        .getAsDouble());
```

Partial output

```
Sum of Employees' salaries (via sum method): 34524.67
Sum of Employees' salaries (via reduce method): 34524.67
Average of Employees' salaries: 4932.10
```

h. The whole final running result

```
Problems Search Console Git Repositories
<terminated> ProcessingEmployees [Java Application] C:\Program Files\Java\jdk1.8.0_60\bin\javaw.exe (2016年3月23日 下午11:01)

Complete Employee list:
Jason Red 5000.00 IT
Ashley Green 7600.00 IT
Matthew Indigo 3587.50 Sales
James Indigo 4700.77 Marketing
Luke Indigo 6200.00 IT
Jason Blue 3200.00 Sales
Wendy Brown 4236.40 Marketing

Employees earning $4000-$6000 per month sorted by salary:
Wendy Brown 4236.40 Marketing
James Indigo 4700.77 Marketing
Jason Red 5000.00 IT

First employee who earns $4000-$6000:
Jason Red 5000.00 IT

Employees in ascending order by last name then first:
Jason Blue 3200.00 Sales
Wendy Brown 4236.40 Marketing
Ashley Green 7600.00 IT
James Indigo 4700.77 Marketing
Luke Indigo 6200.00 IT
Matthew Indigo 3587.50 Sales
Jason Red 5000.00 IT

Employees in descending order by last name then first:
Jason Red 5000.00 IT
Matthew Indigo 3587.50 Sales
Luke Indigo 6200.00 IT
James Indigo 4700.77 Marketing
Ashley Green 7600.00 IT
Wendy Brown 4236.40 Marketing
Jason Blue 3200.00 Sales

Unique employee last names:
Blue
Brown
Green
Indigo

Red

Employee names in order by last name then first name:
Jason Blue
Wendy Brown
Ashley Green
James Indigo
Luke Indigo
Matthew Indigo
Jason Red

Employees by department:
Sales
  Matthew Indigo 3587.50 Sales
  Jason Blue 3200.00 Sales
IT
  Jason Red 5000.00 IT
  Ashley Green 7600.00 IT
  Luke Indigo 6200.00 IT
Marketing
  James Indigo 4700.77 Marketing
  Wendy Brown 4236.40 Marketing

Count of Employees by department:
IT has 3 employee(s)
Marketing has 2 employee(s)
Sales has 2 employee(s)

Sum of Employees' salaries (via sum method): 34524.67
Sum of Employees' salaries (via reduce method): 34524.67
Average of Employees' salaries: 4932.10
```