**Lab 07**

*In this lab, we will:*

* Explore some more advanced concepts & Java facilities with respect to
  + Comparing items
  + searching operations
  + retrieval operations
  + Ordering / Sorting and and listing
  + Generic functions
  + ….though they can be extended to other situations as well

*… which will result in the following learning outcomes:*

* *An appreciation of the various methods that you will find (in professional code-bases)*
* An understanding of how collections of objects can be manipulated as a group

Table of Contents

[1.1 Preliminary 3](#_Toc503341069)

[1.2 Comparing 3](#_Toc503341070)

[1.2.1 Exercise 5](#_Toc503341071)

[1.3 1. List Employees 6](#_Toc503341072)

[1.3.1 Output: 6](#_Toc503341073)

[1.4 2. List Employees (in reverse) 6](#_Toc503341074)

[1.4.1 Output: 7](#_Toc503341075)

[1.5 3. List Employees with a specific role 7](#_Toc503341076)

[1.5.1 Output: 8](#_Toc503341077)

[1.5.2 Exercise 8](#_Toc503341078)

[1.6 4. List Employees with a specific role and number of years of service 8](#_Toc503341079)

[1.6.1 Output: 8](#_Toc503341080)

[1.7 5. List Employees using a local class 8](#_Toc503341081)

[1.7.1 Output: 9](#_Toc503341082)

[1.8 6. List Employees using an anonymous class 10](#_Toc503341083)

[1.9 7. List Employees using a Lambda Expression 10](#_Toc503341084)

[1.10 8. List Employees using a Predicate 11](#_Toc503341085)

[1.10.1 Exercise 11](#_Toc503341086)

[1.11 8. Process Employees using other Standard Functional Interfaces 11](#_Toc503341087)

[1.12 9. Process Elements using a Generic function 13](#_Toc503341088)

[1.13 10. Using aggregate operations 14](#_Toc503341089)

[1.13.1 Exercise 14](#_Toc503341090)

Figure 1: test figure 3

Table 1: Assessment structure 3

## Preliminary

* Download the lab 7 zip file < **Lab7\_EmployeesApp.zip**> from GCULearn and unzip.
* Open the **Lab7\_EmployeesApp** project in NetBeans.

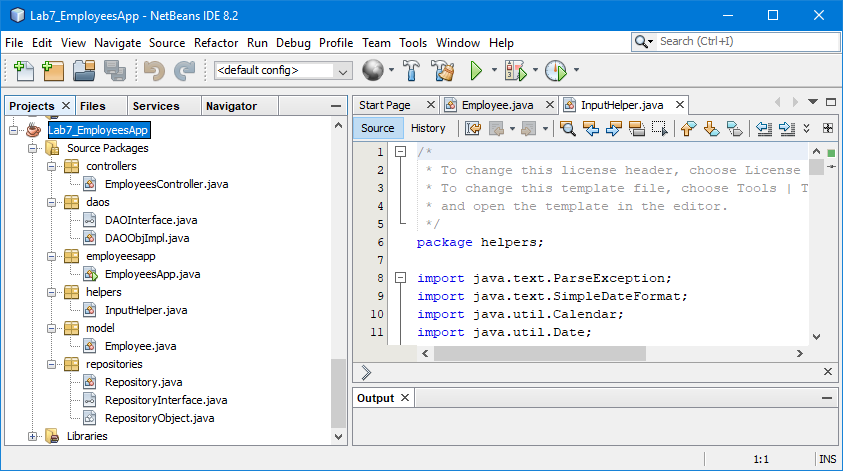


Figure : Lab7\_EmployeesApp

## Comparing

Let’s consider an **Employee** class as detailed below:

**public class Employee extends RepositoryObject**

**implements Comparable<Employee>,**

**Serializable {**

**private final int id;**

**private String employeeName;**

**private String employeeRole;**

**private int lengthOfService;**

**private static int numberOfEmployees=0;**

**public Employee()**

**{**

**this.id = ++numberOfEmployees;**

**this.employeeName = null;**

**this.employeeRole = null;**

**this.lengthOfService = 0;**

**}**

**…**

Note the **Employee** class implements the **Comparable** interface so will have a **compareTo()** method using **id** as a key field and overridden **hashCode()** and **equals()** methods. In addition, a **Comparator** will allow comparison using the **employeeName** field:

@Override

**public int hashCode() {**

**return**

**getId() \* 31 + getEmployeeName().hashCode() \* 31 +**

**getEmployeeRole().hashCode() \* 31 +**

**getLengthOfService() \* 31;**

**}**

@Override

**public boolean equals(Object o) {**

**if (o instanceof Employee) {**

**Employee e = (Employee)o;**

**return**

**e.getId() == getId() &&**

**e.getEmployeeName() == getEmployeeName() &&**

**e.getLengthOfService() == getLengthOfService() &&**

**e.getEmployeeRole().equals(getEmployeeRole());**

**} else {**

**return false;**

**}**

**}**

@Override

**public int compareTo(Employee compareEmployee) {**

**int empId = ((Employee) compareEmployee).getId();**

//ascending order

**return this.id - empId;**

//descending order

//return empId- this.id;

**}**

**public static Comparator<Employee> EmployeeNameComparator**

**= new Comparator<Employee>() {**

@Override

**public int compare(Employee emp1, Employee emp2) {**

**String empName1 = emp1.getEmployeeName();**

**String empName2 = emp2.getEmployeeName();**

//ascending order

**return empName1.compareTo(empName2);**

//descending order

//return empName2.compareTo(empName1);

**}**

**};**

Note we are going to use a generic **Repository** class to hold the employees collection and there the **Employee** class must extend the **RepositoryObject** class. This is because some of the methods in the **Repository** class need to execute methods of an object that is stored in it e.g. **getId()**. We need to ensure that the class of any object that is being stored in a repository must have a **getId()** method so we insist that the object class must extend the **RepositoryObject** class which specifies the methods that must be implemented in order to make the repository function.

**public abstract class RepositoryObject {**

**public abstract int getId();**

**}**

### Exercise

Download and unzip the project for the lab and open in NetBeans. Explore the generic **Repository** class, **RepositoryInterface** and **RepositoryObject** class and ensure you understand them.

Note the **EmployeeController** class is where we create the **Repository** object and implement the required functionality:

**public class EmployeesController {**

**private final Repository<Employee> repository;**

**public EmployeesController() {**

**InputHelper inputHelper = new InputHelper();**

**char c = inputHelper.readCharacter(**

**"Load an already existing Employees File (Y/N)?");**

**if (c == 'Y' || c == 'y') {**

**String fileName =**

**inputHelper.readString("Enter filename");**

**this.repository = new Repository<>(fileName);**

**}**

**else {**

**this.repository = new Repository<>();**

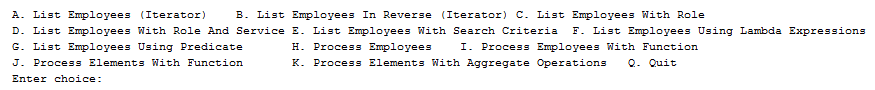
**addEmployees();**

**}**

**}**

Note in this case we are ensuring that the repository object has sufficient elements by either loading them from a file or calling the **addEmployees()** method which hard codes the creation of **Employee** objects which are added to the repository.

Again we are using a console menu-driven approach to explore different ways of working with the items in the repository.



Each of these different ways of processing list items is defined in a private method of the controller class.

We will start with a traditional approach using an **Iterator**, specifically a **ListIterator**, and then explore more generic approaches.

## List Employees

We can use an **Iterator** to traverse the collection and print each employee’s details:

**private void listEmployees() {**

**System.out.format("\033[31m%s\033[0m%n", "Employees");**

**System.out.format("\033[31m%s\033[0m%n", "=========");**

**Iterator it = repository.getItems().listIterator();**

**while (it.hasNext()) {**

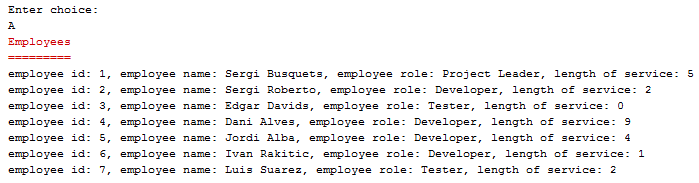
**Employee e = (Employee) it.next();**

**System.out.println(e);**

**}**

**}**

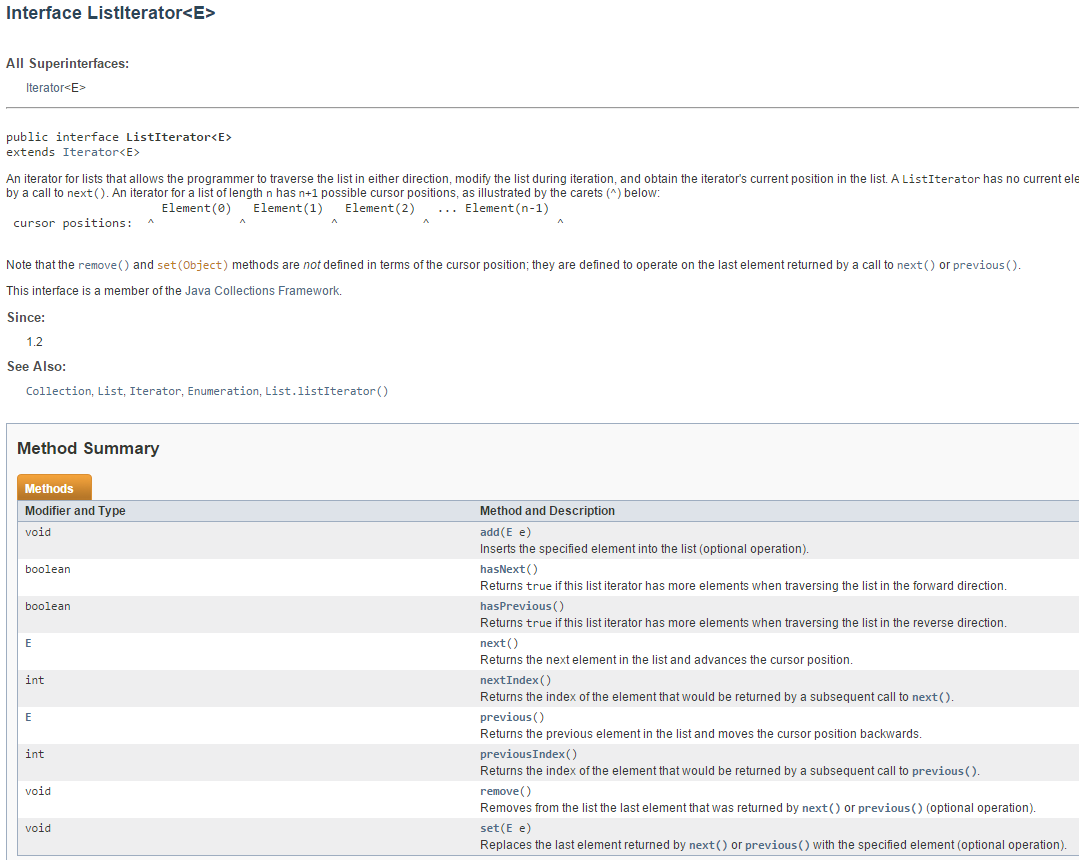
### Output:



Note the use of the **listIterator()** method to create an **Iterator** suitable for traversing a **List** object and the use of the **hasNext()** and **next()** methods.

## List Employees (in reverse)

Recall that the **ListIterator** interface also supplies methods to traverse a **List** object from either end:



**private void listEmployeesInReverse() {**

**System.out.format("\033[31m%s\033[0m%n", "Employees");**

**System.out.format("\033[31m%s\033[0m%n", "=========");**

**ListIterator it =**

**repository.getItems()**

**.listIterator(repository.getItems().size());**

**while (it.hasPrevious()) {**

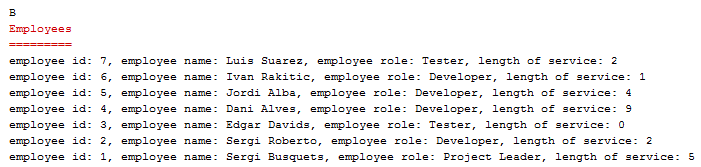
**Employee e = (Employee) it.previous();**

**System.out.println(e);**

**}**

**}**

### Output:



## List Employees with a specific role

We are now going to look at a method to only retrieve employees with a particular specified role:

**String role="Developer";**

**private void listEmployeesWithRole(String role) {**

**…**

**Iterator it = repository.getItems().listIterator();**

**while (it.hasNext()) {**

**Employee e = (Employee) it.next();**

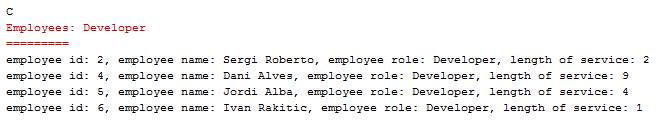
**if (e.getEmployeeRole().equals(role))**

**System.out.println(e);**

**}**

**}**

### Output:



### Exercise

Run the app and ensure you understand the methods discussed above. Create and test a method to list employees with 3 or more years of service.

## List Employees with a specific role and number of years of service

Let’s now look at retrievals which combine criteria e.g. employees who are developers with 3 or more years of service.

**private void listEmployeesWithRoleAndService(String role,**

**int service) {**

**…**

**Iterator it = repository.getItems().listIterator();**

**while (it.hasNext()) {**

**Employee e = (Employee) it.next();**

**if (e.getEmployeeRole().equals(role) &&**

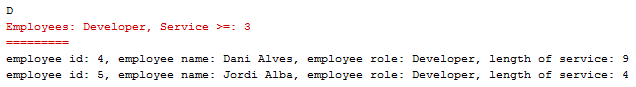
**e.getLengthOfService() >= service)**

**System.out.println(e);**

**}**

**}**

### Output:



## List Employees using a local class

Let’s now begin to generalize our approach through using a local class to implement an interface which will check an **Employee** against set criteria:

**interface CheckEmployee {**

**boolean test(Employee e);**

**}**

The interface is defined inside the **Employee** class and we will define a local class inside the **Employee** class to implement the interface and provide code for the **test()** method i.e. specify how we are going to select employees we are interested in.

The **CheckEmployeeForRoleAndService** local class will implement the **CheckEmployee** interface and provide the concrete method implementation for the abstract method **test()**:

**class CheckEmployeeForRoleAndService implements CheckEmployee {**

**String role;**

**int service;**

**CheckEmployeeForRoleAndService(String role, int service) {**

**this.role = role;**

**this.service = service;**

**}**

**@Override**

**public boolean test(Employee e) {**

**return e.getEmployeeRole().equals(this.role) &&**

**e.getLengthOfService() >= this.service;**

**}**

**}**

Note the local class defines the attributes we are going to use in the test i.e. the role and service; when we create an object of this local class we will use the constructor to provide values for the role and service.

The local class also provides the necessary code for the **test()** method i.e. we are interested in whether this employee has a role which matches the specified role and a length of service which at least equals the value provided.

We now add a public method to the **Employee** class which will allow the controller object to use it to identify employees which match the specified criteria. This **match()** method creates a local class object called **tester** and runs the **test()** method which returns either true or false dependent on whether this employee matches the specified criteria or not.

**public boolean matchRoleAndService(String role, int service) {**

**CheckEmployeeForRoleAndService tester =**

**new CheckEmployeeForRoleAndService(role, service);**

**return tester.test(this);**

**}**

The app code then iterates through the employees collection and invokes the **matchRoleAndService()** method for each employee in turn:

**private void listEmployeesWithRoleAndService(String role,**

**int service) {**

**…**

**Iterator it = repository.getItems().listIterator();**

**while (it.hasNext()) {**

**Employee e = (Employee) it.next();**

**if (e.matchRoleAndService(role, service))**

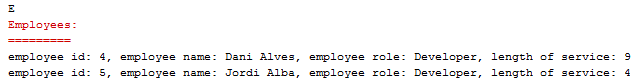
**System.out.println(e);**

**}**

**}**

If an **employee** passes the test then we display the **employee**’s details:

### Output:



## List Employees using an anonymous class

Recall that a functional interface with a single method can be implemented using an anonymous class instead of a local class.

So we can remove the local class, **CheckEmployeeForRoleAndService**, from the **Employee** class and, instead incorporate the creation of the anonymous class object – and the instantiation of the **test()** method – inside the **match()** method

**public boolean matchRoleAndService(String role, int service) {**

// anonymous class

**return new CheckEmployee() {**

**@Override**

**public boolean test(Employee e) {**

**return e.getEmployeeRole().equals(role) &&**

**e.getLengthOfService() >= service;**

**}**

**}.test(this);**

**}**

When the controller object iterates over the collection and calls the **match()** method for each employee in turn then this will instantiate an anonymous class which implements the **CheckEmployee** interface i.e. provides code for the **test()** method and then uses the generated object to run the test on this employee and return true or false to the controller object.

This is still a bit unwieldy but would allow us to create different methods for different types of queries similar to **matchRoleAndService().**

## List Employees using a Lambda Expression

Let’s write a general **match()** method which can be passed a lambda expression to implement the **CheckEmployee** interface rather than using an anonymous class:

**public boolean match(CheckEmployee tester) {**

**return tester.test(this);**

**}**

Lambda expressions allow us to pass code as parameters; in this case, the controller object passes the search criteria as code and this provides the code for the **test()** method defined in the **CheckEmployee** interface.

The app method looks like this:

**private void listEmployeesUsingLambdaExpression(String role,**

**int service) {**

**…**

**Iterator it = repository.getItems().listIterator();**

**while (it.hasNext()) {**

**Employee e = (Employee) it.next();**

**if (e.match(emp -> emp.getEmployeeRole().equals(role) &&**

**emp.getLengthOfService() >= service))**

**System.out.println(e);**

**}**

**}**

The lambda expression provided by the controller object tells the employee object to implement the **test()** method of the **CheckEmployee** interface using this search criteria. The **Employee** **match()** method now looks like this:

**public boolean match(CheckEmployee tester) {**

**return tester.test(this);**

**}**

Thus we have a general method which can accept any criteria formatted using a lambda expression and which returns the evaluation of the **test()** method using the supplied criteria.

## List Employees using a Predicate

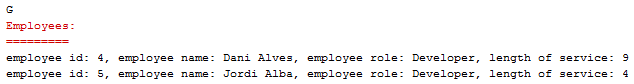
The **CheckEmployee** interface is a functional interface with one abstract method and we can therefore substitute a standard functional interface by supplying a **Predicate** object instantiated with the lambda expression detailing the search criteria.

**public boolean match(Predicate<Employee> tester) {**

**return tester.test(this);**

**}**

We now have a general method with no need for the **CheckEmployee** interface.



### Exercise

Adjust the **match()** method in the **Employee** class as above and test.

## Process Employees using other Standard Functional Interfaces

The **Predicate<T>** interface is a standard functional interface with one method **test()** and so matches the requirements of the **CheckEmployee** interface. Other standard functional interfaces are also of interest; for example, the **Consumer<T>** interface has an **accept()** method which can be used to apply a process to selected elements supplied from ‘instantiating’ the **Predicate<T>** interface.

**private void processEmployees(String role, int service) {**

**…**

**while (it.hasNext()) {**

**Employee e = (Employee) it.next();**

**if (e.match(emp -> emp.getEmployeeRole().equals(role) &&**

**emp.getLengthOfService() >= service))**

**processEmployee(e, emp -> System.out.println(emp));**

**}**

**}**

**private void processEmployee(Employee employee,**

**Consumer<Employee> process) {**

**process.accept(employee);**

**}**

The **match()** method of the **Employee** class is used to test each employee in the collection with a **Predicate<Employee>** using the criteria expressed in lambda format. **Employees** which match the criteria are then passed to a **processEmployee()** method with a lambda expression indicating the process to be applied to each matched employee. The **Consumer<Employee>** interface is then instantiated, with this lambda expression, and the process applied to the **employee**.

If the **Consumer<T>** interface only requires part of the object to process then another standard functional interface **Function<T, R>** can be used to apply a function to the object to map the required data which can then be consumed:

**private void processEmployeesWithFunction(String role,**

**int service) {**

**…**

**while (it.hasNext()) {**

**Employee e = (Employee) it.next();**

**if (e.match(emp -> emp.getEmployeeRole().equals(role) &&**

**emp.getLengthOfService() >= service)) {**

**processEmployeeWithFunction(**

**e,**

**emp -> emp.getEmployeeName(),**

**str -> System.out.println(str)**

**);**

**}**

**}**

**}**

**private void processEmployeeWithFunction(**

**Employee employee,**

**Function<Employee, String> mapper,**

**Consumer<String> process) {**

**String data = mapper.apply(employee);**

**process.accept(data);**

**}**

The function **mapper** is used to retrieve the employee names of the employee objects which match the predicate specified by the lambda expression, the names are then consumed.



## Process Elements using a Generic function

There is very little in the code above which refers directly to details of the **Employee** object; so we look to produce a Generic function which can be applied to any appropriate collection of objects:

**private <X, Y> void processElementsWithFunction(**

**Iterable<X> source,**

**Predicate<X> tester,**

**Function<X, Y> mapper,**

**Consumer<Y> process) {**

**for (X x : source) {**

**if (tester.test(x)) {**

**Y data = mapper.apply(x);**

**process.accept(data);**

**}**

**}**

**}**

This can be called supplying suitable parameters:

**processElementsWithFunction(**

**repository.getItems(),**

**(Employee e) -> e.getEmployeeRole().equals(role) &&**

**e.getLengthOfService() >= service,**

**e -> e.getEmployeeName(),**

**str -> System.out.println(str));**

As we have a generic function, we can call it with any suitable collection, predicate, function and consumer.

## Using aggregate operations

We can go further than that and use built in aggregate operations to stream a collection and apply functions to that stream:

**repository.getItems()**

**.stream()**

**.filter(e -> e.getEmployeeRole().equals(role) &&**

**e.getLengthOfService() >= service)**

**.map(e -> e.getEmployeeName())**

**.forEach(str -> System.out.println(str));**

### Exercise

Run the app and test each option in turn ensuring you understand the code.

~~~~~~