**PRACTICAL NO. 1**

**Java Collections, Generics and Lambda Expression**

|  |  |
| --- | --- |
| LOB1 | Study Java Generics, Collection framework and Lambda expressions. |
| LO1 | Develop application to use various data structures and data manipulation concept using Java Collection Framework and Lambda expressions. |

1. **Generics**

The term *generics* mean *parameterized types.*

Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.

Using generics, it is possible to create a single class, for example, that automatically works with different types of data.

A class, interface, or method that operates on a parameterized type is called *generic,* as in *generic class* or *generic method.*

With generics, all casts are automatic and implicit. Thus, generics expand your ability to reuse code and let you do so safely and easily.

It also allows programmer to write a single sort of method that could sort the elements in an Integer array, a String array, or an array of any type that supports ordering.

**Advantage of Java Generics**

* **Type-safety:** We can hold only a single type of objects in generics. It doesn’t allow storing other objects.
* **Type casting is not required:** There is no need to typecast the object.
* **Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime.

**The generics syntax –**

For declaring a generic class:

class *class-name<type-param-list> { // ...*

For declaring a reference to a generic class:

*class-name<type-arg-list> var-name =* new *class-name<type-arg-list>(cons-arg-list);*

1. **Collections**

* A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit.
* Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).
* Collections in java is a framework that provides an architecture to store and manipulate the group of objects.
* All the operations that you perform on data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.
* Some of the most common types of collections include:
  + **Lists**: A list is a collection of objects that are stored in a specific order.
  + **Sets**: A set is a collection of objects that are unique.
  + **Maps**: A map is a collection of key-value pairs.
* Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc).
* The **java.util** package contains all the classes and interfaces for Collection framework.

1. **Lambda Expression**

* Lambda expression provides a clear and concise way to represent one method interface using an expression.
* It is very useful in collection library. It helps to iterate, filter and extract data from collection.
* Java lambda expressions are Java's first step into functional programming.
* A Java lambda expression is thus a function which can be created without belonging to any class.
* A lambda expression can be passed around as if it was an object and executed on demand.
* Here is an example of a lambda expression in Java:

(int x) -> { return x+1; }

* It tells that a lambda expression is something like a method without a name.
* It has everything that a method has: an argument list, which is the (int x) part of the sample lambda, and a body, which is the { return x+1; } part after the arrow symbol.
* Return type and exceptions are inferred by the compiler from the lambda body; in our example the return type is “int” and the throws clause is empty.
* Lambda expression syntax can be divided into three parts.

1. **List of Parameters or Argument List**
2. **Arrow (->) token**
3. **Body of a lambda expression**

**Q1. Develop a Java Program to demonstrate a generic concept with generic method**

**to swap two different elements in an array.**

**Program :**

**GenericUtilsSwap.java**

**package** PackPrac01;

**public** **class** GenericUtilsSwap {

**public** **static** <T> **void** swap(T[] array, **int** index1, **int** index2) {

**if** (index1 >= 0 && index1 < array.length && index2 >= 0 && index2 < array.length) {

T temp = array[index1];

array[index1] = array[index2];

array[index2] = temp;

} **else** {

System.***out***.println("Invalid indices provided!");

}

}

**public** **static** <T> **void** printArray(T[] array) {

**for** (T element : array) {

System.***out***.print(element + " ");

}

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

}

}

**GenericUtilsSwapMain.java**

**package** PackPrac01;

**public** **class** GenericUtilsSwapMain {

**public** **static** **void** main(String[] args) {

Integer[] intArray = {1, 2, 3, 4, 5};

System.***out***.println("Before swapping: ");

GenericUtilsSwap.*printArray*(intArray);

GenericUtilsSwap.*swap*(intArray, 1, 4);

System.***out***.println("After swapping: ");

GenericUtilsSwap.*printArray*(intArray);

String[] strArray = {"a", "b", "c", "d"};

System.***out***.println("\nBefore swapping: ");

GenericUtilsSwap.*printArray*(strArray);

GenericUtilsSwap.*swap*(strArray, 3, 2);

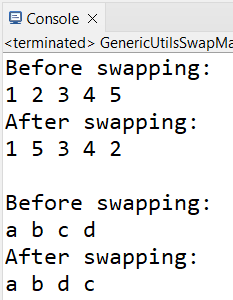
System.***out***.println("After swapping: ");

GenericUtilsSwap.*printArray*(strArray);

}

}

**Output :**

****

**Q2.** **You are working as a software developer for a company that develops mobile**

**apps. The app allows users to create and manage lists of items. You have been**

**tasked with improving the performance of the app’s list management features.**

**You decide to use the following techniques:**

**1. Use a generic list to store, delete, retrieve, and find the items in the list.**

**2. Use a lambda expression to filter and sort the list of items.**

**Program :**

**ItemManager.java**

**package** PackPrac01;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Collectors;

**public** **class** ItemManager<T> {

**private** List<T> items;

**public** ItemManager() {

items = **new** ArrayList<>();

}

**public** **void** addItem(T item) {

items.add(item);

}

**public** **void** deleteItem(T item) {

items.remove(item);

}

**public** T getItem(**int** index) {

**if** (index >= 0 && index < items.size()) {

**return** items.get(index);

}

**return** **null**;

}

**public** **boolean** findItem(T item) {

**return** items.contains(item);

}

**public** List<T> filterItems(java.util.function.Predicate<T> condition) {

**return** items.stream()

.filter(condition)

.collect(Collectors.*toList*());

}

**public** **void** sortItems(java.util.Comparator<T> comparator) {

items.sort(comparator);

}

**public** **void** displayItems() {

System.***out***.println("Items in the list: " + items);

}

}

**ItemManagerMain.java**

**package** PackPrac01;

**import** java.util.List;

**public** **class** ItemManagerMain {

**public** **static** **void** main(String[] args) {

ItemManager<String> itemManager = **new** ItemManager<>();

itemManager.addItem("Apple");

itemManager.addItem("Banana");

itemManager.addItem("Orange");

itemManager.addItem("Grapes");

itemManager.addItem("Cherry");

itemManager.displayItems();

itemManager.deleteItem("Banana");

System.***out***.println("\nAfter deleting Banana:");

itemManager.displayItems();

String itemAtIndex = itemManager.getItem(2);

System.***out***.println("\nItem at index 2: " + itemAtIndex);

**boolean** isFound = itemManager.findItem("Grapes");

System.***out***.println("Is Grapes in the list? " + isFound);

List<String> filteredItems = itemManager.filterItems(item -> item.startsWith("G"));

System.***out***.println("\nFiltered items (starting with 'G'): " + filteredItems);

itemManager.sortItems(String::compareTo);

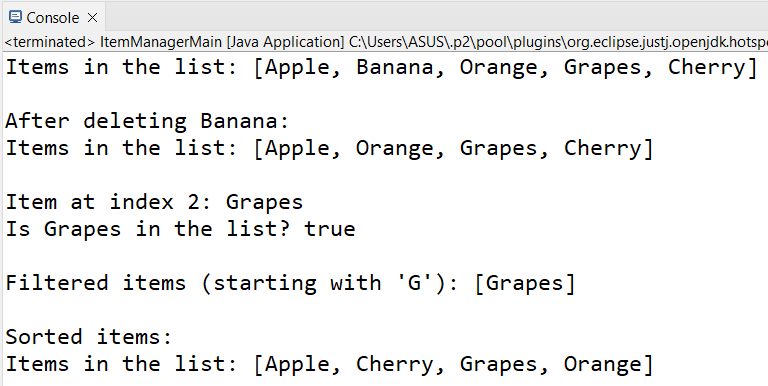
System.***out***.println("\nSorted items:");

itemManager.displayItems();

}

}

**Output :**

****

**Q3. Write a Java program using Lambda Expression with multiple parameters to add**

**two numbers.**

**Program :**

**AddLambda.java**

**package** PackPrac01;

**public** **interface** AddLambda {

**int** add(**int** a,**int** b);

}

**LambdaAdd.java**

**package** PackPrac01;

**public** **class** LambdaAdd {

**public** **static** **void** main(String[] args) {

AddLambda addfunction=(a,b)->a+b;

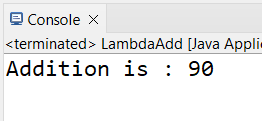
**int** result=addfunction.add(50,40);

System.***out***.println("Addition is : "+result);

}

}

**Output :**



**Q4.** **Develop a Java Program to for user defined class hierarchy**

**Person🡪Employee🡪Manager and Person 🡪Employee 🡪Clerk. Write a method**

**calculateSalary (float noOfdaysWork) in Employee class which can accept**

**instances of Manager and Clerk classes.**

**Program :**

**EmployeeHierarchy.java**

**package** PackPrac01;

**class** Person {

String name;

**public** Person(String name) {

**this**.name = name;

}

**public** String getName() {

**return** name;

}

}

**class** Employee **extends** Person {

**float** dailyWage;

**public** Employee(String name, **float** dailyWage) {

**super**(name);

**this**.dailyWage = dailyWage;

}

**public** **void** calculateSalary(**float** noOfDaysWork) {

**float** salary = noOfDaysWork \* dailyWage;

System.***out***.println("\nEmployee " + name + "'s salary is: " + salary);

}

}

**class** Manager **extends** Employee {

**public** Manager(String name, **float** dailyWage) {

**super**(name, dailyWage);

}

**public** **void** calculateSalary(**float** noOfDaysWork) {

**float** salary = noOfDaysWork \* dailyWage;

System.***out***.println("\nManager " + name + "'s salary is: " + salary);

}

}

**class** Clerk **extends** Employee {

**public** Clerk(String name, **float** dailyWage) {

**super**(name, dailyWage);

}

**public** **void** calculateSalary(**float** noOfDaysWork) {

**float** salary = noOfDaysWork \* dailyWage;

System.***out***.println("\nClerk " + name + "'s salary is: " + salary);

}

}

**EmployeeHierarchyMain.java**

**package** PackPrac01;

**public** **class** EmpolyeeHierarchyMain {

**public** **static** **void** main(String[] args) {

Manager mobj = **new** Manager("Vedant", 500);

Clerk cobj = **new** Clerk("Aryan", 200);

Employee eobj =**new** Employee("Nick",300);

System.***out***.println("Calculating salaries:");

eobj.calculateSalary(18);

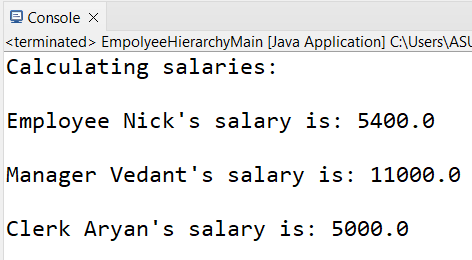
mobj.calculateSalary(22);

cobj.calculateSalary(25);

}

}

**Output :**

****

**Q5.** **Create a program that demonstrates the use of the generic Stack class that**

**accepts objects of Manager and Clerk classes and perform various operations.**

**Program :**

**EmployeeHierarchyWithGenericStack.java**

**package** PackPrac01;

**import** java.util.ArrayList;

**class** Person1 {

**protected** String name;

**public** Person1(String name) {

**this**.name = name;

}

**public** String getName() {

**return** name;

}

}

**class** Employee1 **extends** Person1 {

**protected** **float** dailyWage;

**public** Employee1(String name, **float** dailyWage) {

**super**(name);

**this**.dailyWage = dailyWage;

}

**public** **void** calculateSalary(**float** noOfDaysWork) {

**float** salary = noOfDaysWork \* dailyWage;

System.***out***.println(name + "'s salary is: " + salary);

}

}

**class** Manager1 **extends** Employee1 {

**public** Manager1(String name, **float** dailyWage) {

**super**(name, dailyWage);

}

@Override

**public** **void** calculateSalary(**float** noOfDaysWork) {

**float** salary = noOfDaysWork \* dailyWage;

System.***out***.println("Manager " + name + "'s salary is: " + salary);

}

}

**class** Clerk1 **extends** Employee1 {

**public** Clerk1(String name, **float** dailyWage) {

**super**(name, dailyWage);

}

@Override

**public** **void** calculateSalary(**float** noOfDaysWork) {

**float** salary = noOfDaysWork \* dailyWage;

System.***out***.println("Clerk " + name + "'s salary is: " + salary);

}

}

**class** GenericStack<T **extends** Employee1> {

**private** ArrayList<T> stack = **new** ArrayList<>();

**public** **void** push(T item) {

stack.add(item);

}

**public** T pop() {

**if** (!stack.isEmpty()) {

**return** stack.remove(stack.size() - 1);

} **else** {

System.***out***.println("Stack is empty!");

**return** **null**;

}

}

**public** T peek() {

**if** (!stack.isEmpty()) {

**return** stack.get(stack.size() - 1);

} **else** {

System.***out***.println("Stack is empty!");

**return** **null**;

}

}

**public** **boolean** isEmpty() {

**return** stack.isEmpty();

}

**public** **void** displayStack() {

**if** (stack.isEmpty()) {

System.***out***.println("Stack is empty!");

} **else** {

System.***out***.println("Current stack:");

**for** (T item : stack) {

System.***out***.println(item.getName() + " is in the stack.");

}

}

}

}

**GenericStackMain.java**

**package** PackPrac01;

**public** **class** GenericStackMain {

**public** **static** **void** main(String[] args) {

GenericStack<Employee1> employeeStack = **new** GenericStack<>();

Manager1 manager1 = **new** Manager1("Kunal", 500);

Manager1 manager2 = **new** Manager1("Nikhil", 550);

Clerk1 clerk1 = **new** Clerk1("Vedant", 200);

Clerk1 clerk2 = **new** Clerk1("Aryan", 220);

employeeStack.push(manager1);

employeeStack.push(clerk1);

employeeStack.push(manager2);

employeeStack.push(clerk2);

System.***out***.println("Stack after pushing employees:");

employeeStack.displayStack();

System.***out***.println("\nPeeking at the top of the stack:");

Employee1 topEmployee = employeeStack.peek();

**if** (topEmployee != **null**) {

System.***out***.println("Top employee: " + topEmployee.getName());

}

System.***out***.println("\nPopping and calculating salary:");

**while** (!employeeStack.isEmpty()) {

Employee1 emp = employeeStack.pop();

emp.calculateSalary(22);

}

**if** (employeeStack.isEmpty()) {

System.***out***.println("\nThe stack is empty now.");

}

}

}

**Output :**

