Core Java 8 and Development Tools

Lab Book

Document Revision History

Date	Revision No.	Author	Summary of Changes
17-11-2013	1.0	Rathnajothi P	As of updated module content, designed lab book
28-05-2015	2.0	Vinod Satpute	Updated to include new features of Java SE 8, Junit 4 and JAXB 2.0
25-05-2016	3.0	Tanmaya K Acharya	Updated as per the integrated ELT TOC

Table of Contents

Getting Started

Overview

This lab book is a guided tour for learning Core Java version 8 and development tools. It comprises of assignments to be done. Refer the demos and work out the assignments given by referring the case studies which will expose you to work with Java applications.

Setup Checklist for Core Java

Here is what is expected on your machine in order to work with lab assignment.

Minimum System Requirements

- Intel Pentium 90 or higher (P166 recommended)
- Microsoft Windows 7 or higher.
- Memory: (1GB or more recommended)
- Internet Explorer 9.0 or higher or Google Chrome 43 or higher
- Connectivity to Oracle database

Please ensure that the following is done:

- A text editor like Notepad or Eclipse is installed.
- JDK 1.8 or above is installed. (This path is henceforth referred as <java_home>)

Instructions

- For all Naming conventions, refer Appendix A. All lab assignments should adhere to naming conventions.
- Create a directory by your name in drive <drive>. In this directory, create a subdirectory java_assignments. For each lab exercise create a directory as lab <lab number>.

Learning More (Bibliography if applicable)

- https://docs.oracle.com/javase/8/docs/
- Java, The Complete Reference; by Herbert Schildt
- Thinking in Java; by Bruce Eckel
- Beginning Java 8 Fundamentals by KishoriSharan

Problem Statement/ Case Study (If applicable)

1. **Bank Account Management System:**

   - Funds Bank needs an application to feed new Account Holder information. AccountHolder will be a person. There are two types of accounts such as SavingsAccount, CurrentAccount.

2. **Employee Medical Insurance Scheme:**

   - By default, all employees in an organization will be assigned with a medical insurance scheme based on the salary range and designation of the employee. Refer the below given table to find the eligible insurance scheme specific to an employee.

Salary	Designation	Insurance scheme
>5000 and < 20000	System Associate	Scheme C
>=20000 and <40000	Programmer	Scheme B
>=40000	Manager	Scheme A
<5000	Clerk	No Scheme

Lab 1: Working with Java and Eclipse IDE

Goals	Learn and understand the process of: ➢ Setting environment variables ➢ Creating a simple Java Project using Eclipse 3.0 or above
Time	45 minutes

1.1: Setting environment variables from CommandLineSolution:

Step 1: Set **JAVA_HOME** to Jdk1.8 using the following command:

- **Set JAVA_HOME=**C:\Program Files\Java\jdk1.8.0_25

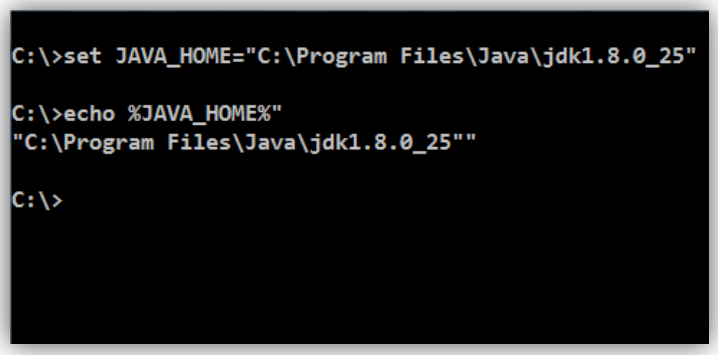


Figure 1: Java program

Step 2: Set PATH environment variable:

- **Set PATH=%PATH%;%JAVA_HOME%\bin;**

Step 3: Set your current working directory and set classpath.

- Set CLASSPATH=.

Note: Classpath searches for the classes required to execute the command. Hence it must be set to the directory containing the class files or the names of the jars delimited by ;

For example: C:\Test\myproject\Class;ant.jar

Alternatively follow the following steps for setting the environment variables

Alternate approach:

Step 1: Right click **My Computers**, and select **Properties→Environment Variables**.

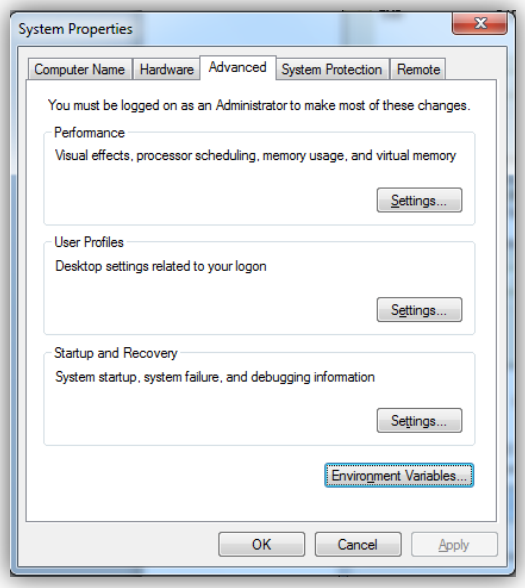


Figure 2: System Properties

Step 2: Click **Environment Variables**. The Environment Variables window will bedisplayed.

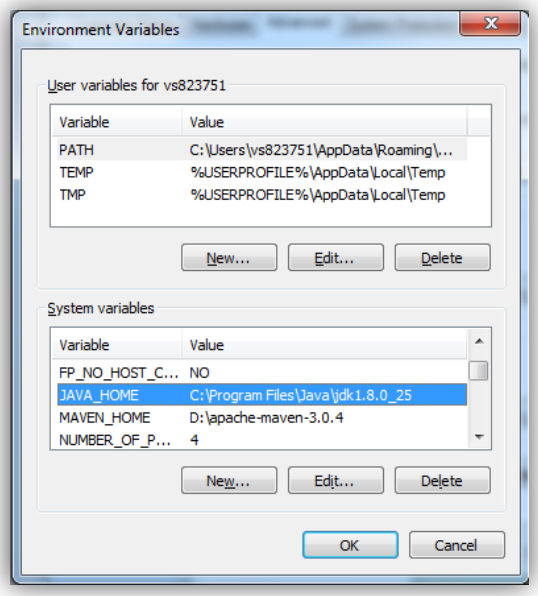


Figure 3: Environment Variables

Step 3: Click **JAVA_HOME** System Variable if it already exists, or create a new one and set the path of JDK1.8 as shown in the figure.

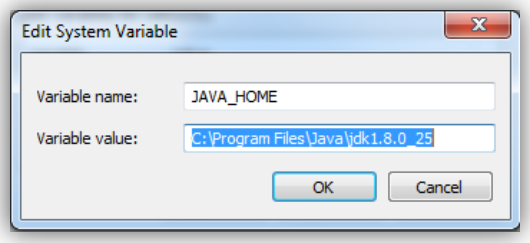


Figure 4: Edit System Variable

Step 4: Click **PATH** System Variable and set it as ***%PATH%;%JAVA_HOME%\bin***.

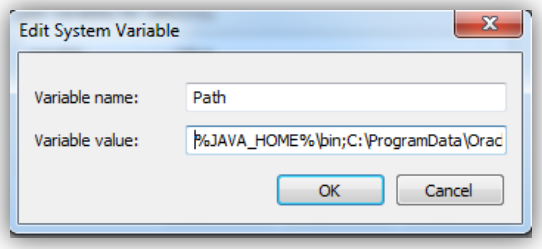


Figure 5: Edit System Variable

Step 5: Set **CLASSPATH** to your working directory in the **User Variables** tab.

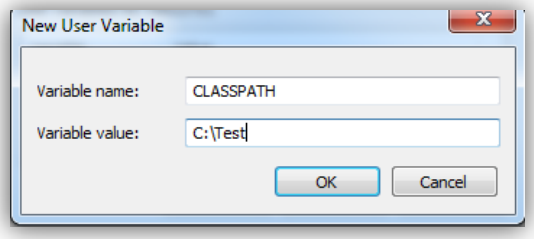


Figure 6: Edit User Variable

1.2: Create Java Project

Create a simple java project named 'MyProject'.

Solution:

Step 1: Open **eclipse 4.4**(or above)

Step 2: Select **File→New→Project →Java project**.

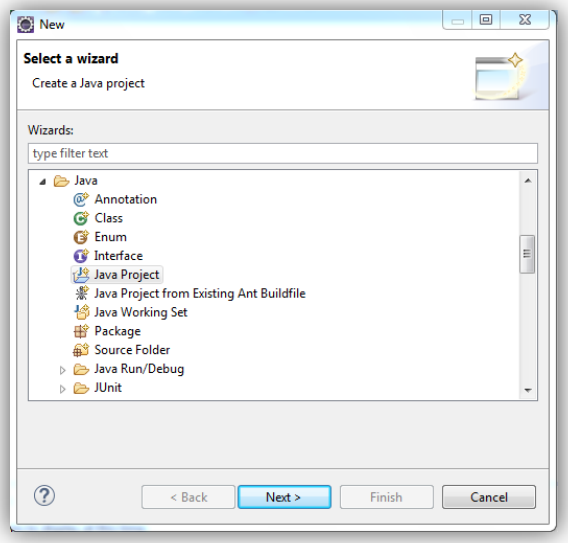


Figure 7: Select Wizard

Step 3: Click **Next** and provide name for the project.

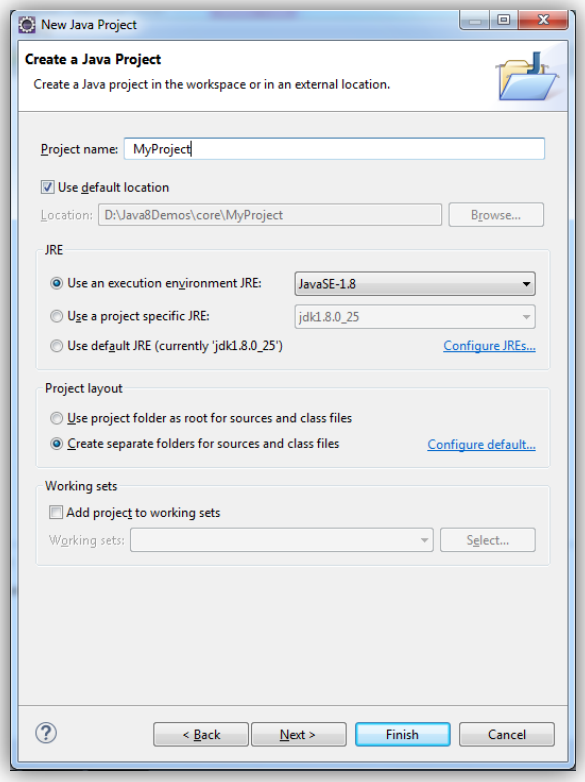


Figure 8: New Java Project

Step 4: Click **Next** and select build options for the project.

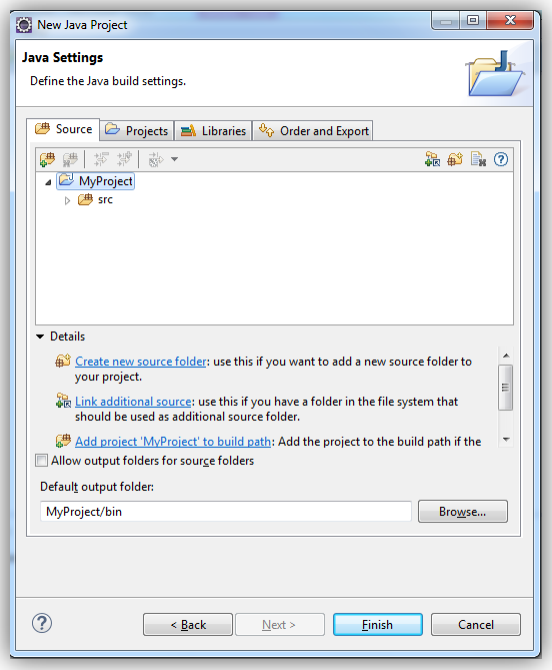


Figure 9: Java Settings

Step 5: Click **Finish** to complete the project creation.

Step 6: Right-click **myproject**, and select resource type that has to be created.

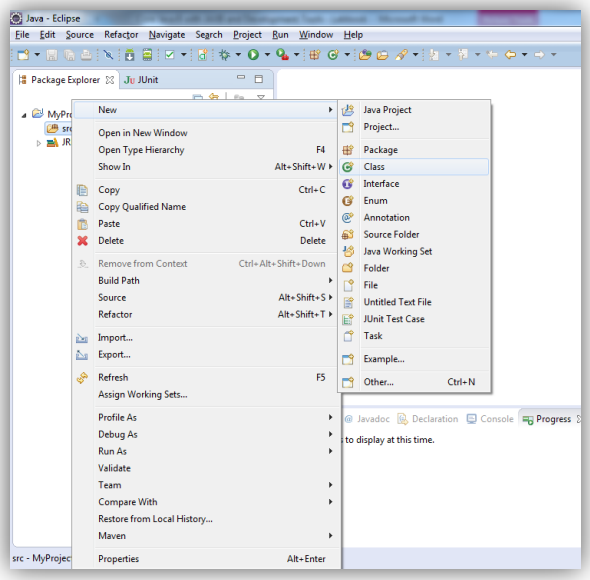


Figure 10: Select Resource

Step 7: Providename and other details for the class, and click **Finish**.

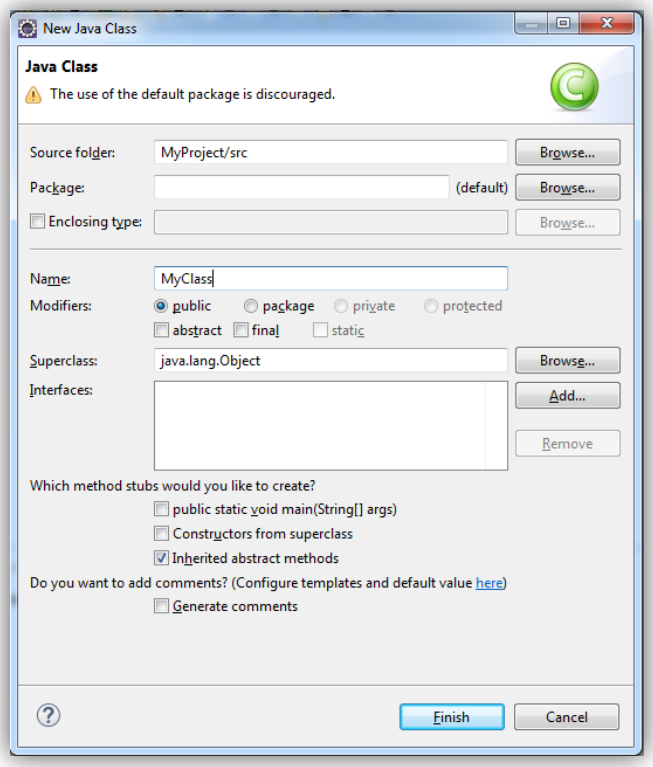


Figure 11: Java Class

This will open **MyClass.java** in the editor, with ready skeleton for the class, default constructor, **main()** method, and necessary **javadoc** comments.

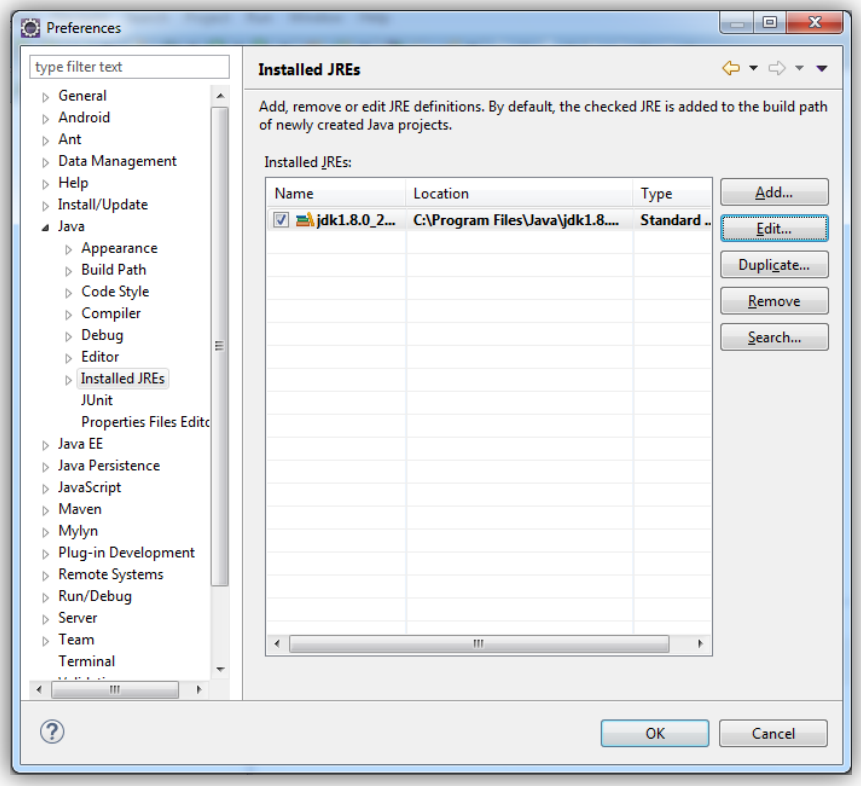
To run this class, select **Run** from toolbar, or select **Run As →Java application**. Alternatively, you can select **Run..**and you will be guided through a wizard, for the selection of class containing **main()** method.

Console window will show the output.

1.3: Using offline Javadoc API in Eclipse

Step 1: Open **eclipse 4.4**(or above)

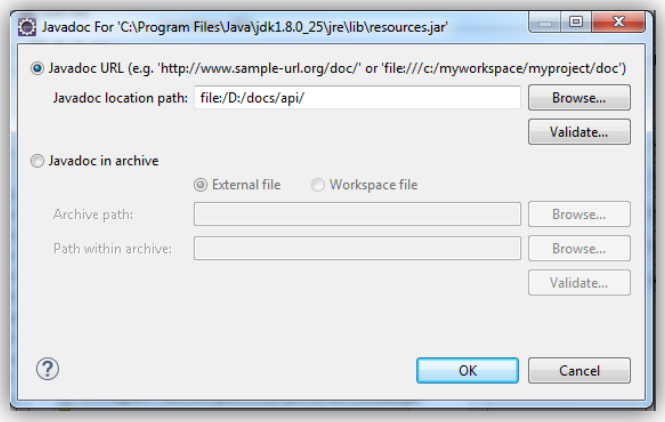
Step2: From eclipse Window → Preferences →Java → "Installed JREs" select available JRE (jdk1.8.0_25 for instance) and click Edit.



Step3: Select all the "JRE System libraries" using Control+A.

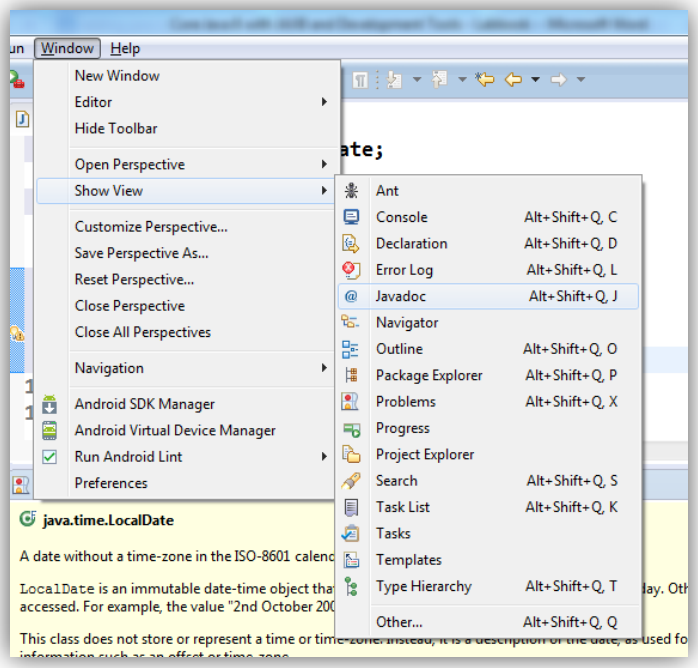
Step 4: Click "Javadoc Location"

Step 5: Change "Javadoc location path:" from http://download.oracle.com/javase/8/docs/api/ to "file:/E:/Java/docs/api/".

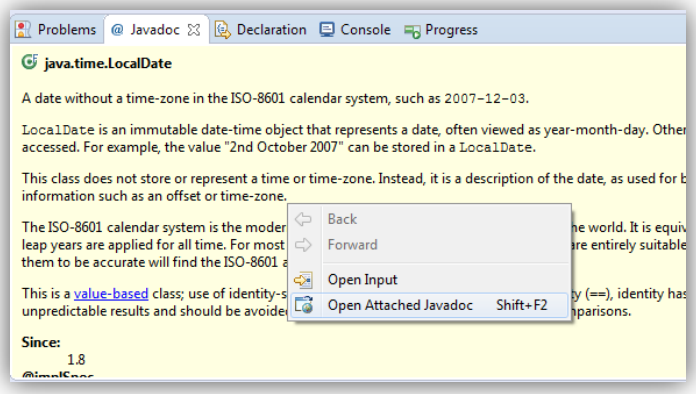


Step 6: Close all windows by either clicking on ok/apply.

Step 7: Open the Javadoc view from Window → Show View → Javadoc.



Note: Henceforth whenever you select any class or method in Editor Window, it Javadoc view will display the reference documentation.



If you want to open the Java documentation for specified resource as html page, right click in the Javadoc view → Open Attached Javadoc.

Lab 2: Language Fundamentals, Classes and Objects

Goals	At the end of this lab session, you will be able to: ➢ Write a Java program that displays person details ➢ Working with Conditional Statements ➢ Create Classes and Objects
Time	120 minutes

2.1 Write a java program to print person details in the format as shown below:

```
Person Details:
_____________

First Name: Divya
Last Name: Bharathi
Gender: F
Age: 20
Weight: 85.55
```

Figure 12: Sample output of Person details

2.2: Write a program to accept a number from user as a command line argument and check whether the given number is positive or negative number.

2.3: Refer the class diagram given below and createa personclass.

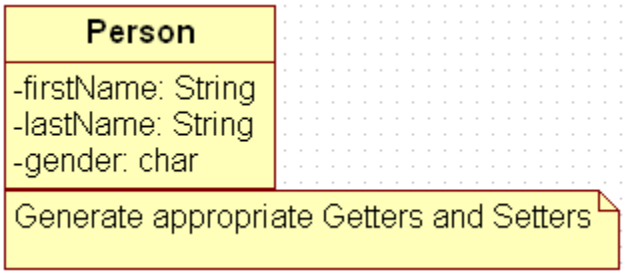


Figure 13: Class Diagram of Person

Create default and parameterized constructor for Person class.

Also Create “PersonMain.java” program and write code for following operations:

a) Create an object of Person class and specify person details through constructor.
b) Display the details in the format given in Lab assignment 2.1

2.4: Modify Lab assignment 2.3 to accept phone number of a person. Create a newmethod to implement the same and also define method for displaying persondetails.

2.5: Modify the above program, to accept only ‘M’ or ‘F’ as gender field values. Use Enumeration for implementing the same.

Lab 3: Exploring Basic Java Class Libraries

Goals	At the end of this lab session, you will be able to: ➢ Working with Basic Java Class Libraries ➢ Working with Strings and Date and Time API
Time	100 minutes

3.1: Create a method which can perform a particular String operation based on the user's choice. The method should accept the String object and the user's choice and return the output of the operation.
Options are

- Add the String to itself
- Replace odd positions with #
- Remove duplicate characters in the String
- Change odd characters to upper case

3.2: Create a method that accepts a String and checks if it is a positive string. A string is considered a positive string, if on moving from left to right each character in the String comes after the previous characters in the Alphabetical order.For Example: ANT is a positive String (Since T comes after N and N comes after A). The method should return true if the entered string is positive.

3.3: Create a method to accept date and print the duration in days, months and years with regards to current system date.

3.4: Revise exercise 3.3 to accept two LocalDates and print the duration between dates in days, months and years.

3.5: Create a method to accept product purchase date and warrantee period (in terms of months and years). Print the date on which warrantee of product expires.

3.6: Create a method which accept zone id and print the current date and time with respect to given zone. (Hint: Few zones to test your code. America/New_York, Europe/London, Asia/Tokyo, US/Pacific, Africa/Cairo, Australia/Sydney etc.)

3.7: Modify Lab assignment 2.3 to perform following functionalities:

a) Add a method called calculateAge which should accept person's date of birth and calculate age of a person.

b) Add a method called getFullName(String firstName, String lastName) which should return full name of a person

Display person details with age and fullname.

Lab 4: Inheritance and Polymorphism

Goals	At the end of this lab session, you will be able to: ➢ Write a Java program that manipulates person details ➢ Working with Inheritance, Polymorphism
Time	120 minutes

4.1: Refer the case study 1in Page No: 5 and create Account Class as shown below in class diagram. Ensure minimum balance of INR 500 in a bank account is available.

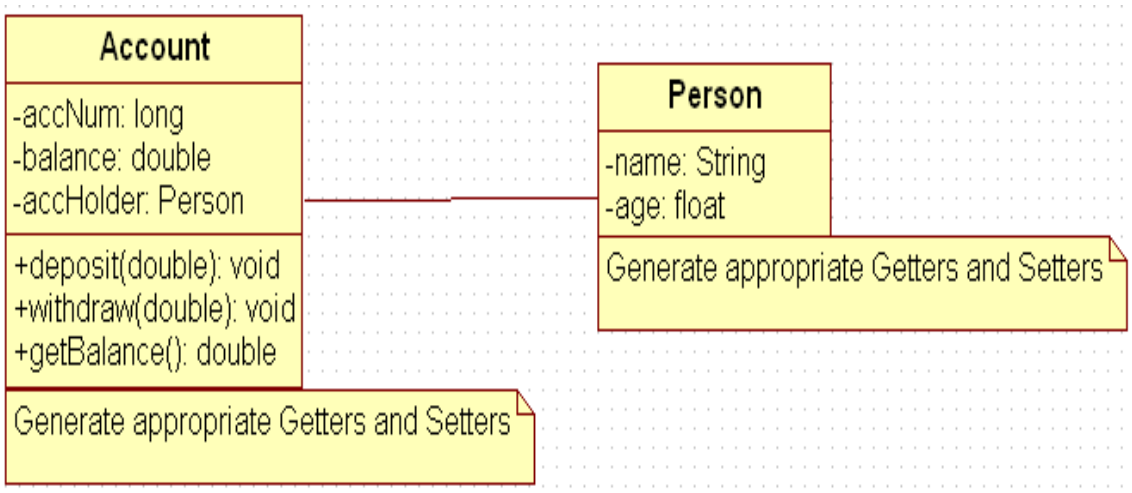


Figure 14: Association of person with account class

a) Create Account for smith with initial balance as INR 2000 and for Kathy with initial balance as 3000.(accNum should be auto generated).
b) Deposit 2000 INR to smith account.
c) Withdraw 2000 INR from Kathy account.
d) Display updated balances in both the account.
e) Generate toString() method.

4.2: Extend the functionality through Inheritanceand polymorphism (Maintenance)

Inherit two classes Savings Account and Current Account from account class. Implement the following in the respective classes.

a) Savings Account
    a. Add a variable called minimum Balance and assign final modifier.
    b. Override method called withdraw (This method should check for minimum balance and allow withdraw to happen)

b) Current Account
    a. Add a variable called overdraft Limit
    b. Overridemethod called withdraw (checks whether overdraft limit is reached and returns a boolean value accordingly)

Lab 5: Abstract classes and Interfaces

Goals	At the end of this lab session, you will be able to: ➢ Use of abstract classes and interfaces
Time	90 minutes

5.1: Refer the case study 2 in page no: 5 and create an application for that requirement by creating packages and classes as given below:

a) com.cg.eis.bean

In this package, create "Employee" class with different attributes such as id, name, salary, designation, insuranceScheme.

b) com.cg.eis.service

This package will contain code for services offered in Employee Insurance System. The service class will have one EmployeeService Interface and its corresponding implementation class.

c) com.cg.eis.pl

This package will contain code for getting input from user, produce expected output to the user and invoke services offered by the system.

The services offered by this application currently are:

i) Get employee details from user.

ii) Find the insurance scheme for an employee based on salary and designation.

iii) Display all the details of an employee.

5.2: Use overrides annotation for the overridden methods available in a derived class of an interface of all the assignments.

5.3: Refer the problem statement 4.1. Modify account class as abstract class and declare withdraw method.

Lab 6: Exception Handling

Goals	At the end of this lab session, you will be able to: ➢ Create and use application specific exceptions
Time	120 minutes

6.1: Modify the Lab assignment 2.3 to validate the full name of an employee. Create and throw a user defined exception if firstName and lastName is blank.

6.2: Validate the age of a person in Lab assignment 4.2 and display proper message by using user defined exception. Age of a person should be above 15.

6.3: Modify the Lab assignment 5.1 to handle exceptions. Create an Exception class named as "EmployeeException"(User defined Exception) in a package named as "com.cg.eis.exception" and throw an exception if salary of an employee is below than 3000. Use Exception Handling mechanism to handle exception properly.

Lab 7: Arrays and Collections

Goals	At the end of this lab session, you will be able to: ➢ Use Comparator interface ➢ Use Collections ➢ Use Generics with Collection Classes ➢ Use iterators to iterate through Collections
Time	180 minutes

7.1: Write a program to store product names in a string array and sort strings available in an array.

7.2: Modify the above program to store product names in anArrayList, sort strings available in an arrayList and display the names using for-each loop.

7.3: Modify the lab assignment 5.1 to accept multiple employee details and store all employee objects in a HashMap. The functionalities need to be implemented are:

i) Add employee details to HashMap.

ii) Accept insurance scheme from user and display employee details based on Insurance scheme

iii) Delete an employee details from map.

iv) Sort the employee details based on salary and display it.

Note: Use generics and Comparable/comparator interface.

Sample code Snippet of EmployeeServiceImpl class:

```
public class EmployeeServiceImpl {

HashMap<String,Employee> list = new HashMap<String,Employee>();

public void addEmployee(Employee emp)          {
                                                    //code to add employee
}
public boolean deleteEmployee(int id)    {
//     code to delete a employee whose id is passed as parameter
}
……..
}
```


Lab 8: Files IO

Goals	At the end of this lab session, you will be able to: ➢ Read and write data using streams. ➢ Serialize and Deserialize objects
Time	75 minutes

8.1: Write a program to read content from file, reverse the content and write the reversed content to the file. (Use Reader and Writer APIs).

8.2: Create a file named as “numbers.txt” which should contain numbers from 0 to 10 delimited by comma. Write a program to read data from numbers.txt using Scanner class API and display only even numbers in the console.

8.3: Enhance the lab assignment 6.3 by adding functionality in service class to write employee objects into a File. Also read employee details from file and display the same in console. Analyze the output of the program.

Lab 9: Introduction to Junit

Goals	At the end of this lab session, you will be able to: ➢ Configuring JUnit in Eclipse ➢ Using JUnit to write TestCase for standalone Java Applications
Time	120 minutes

9.1: Configuration of JUnit in Eclipse

Step 1: Create a Java project.

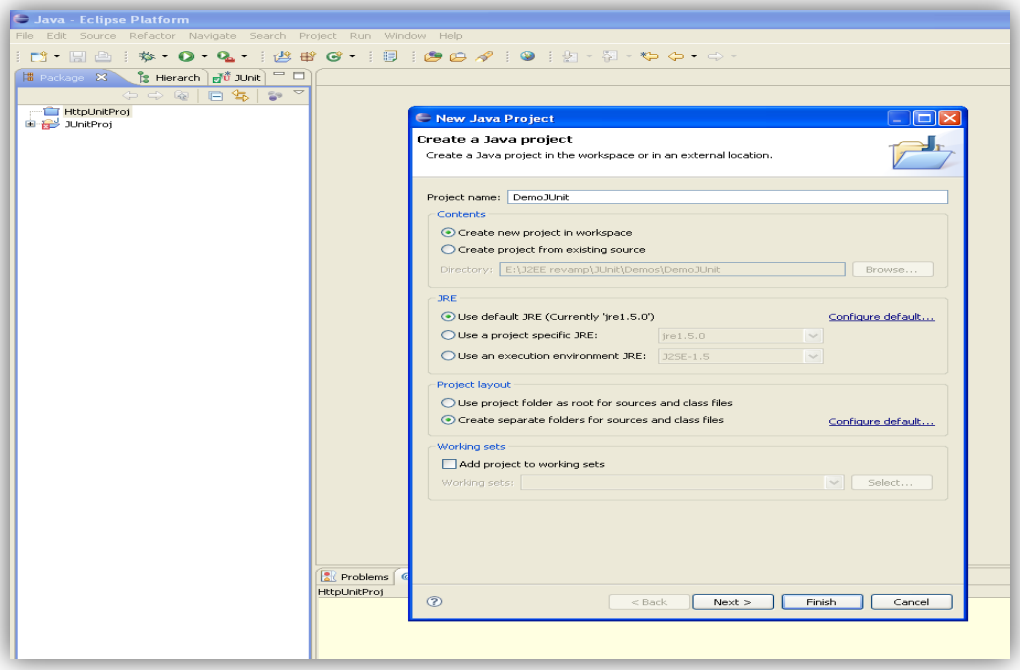


Figure 155: Creating Java Project in Eclipse

Step 2: Add **junit4.4.jar** in the build path of the project.

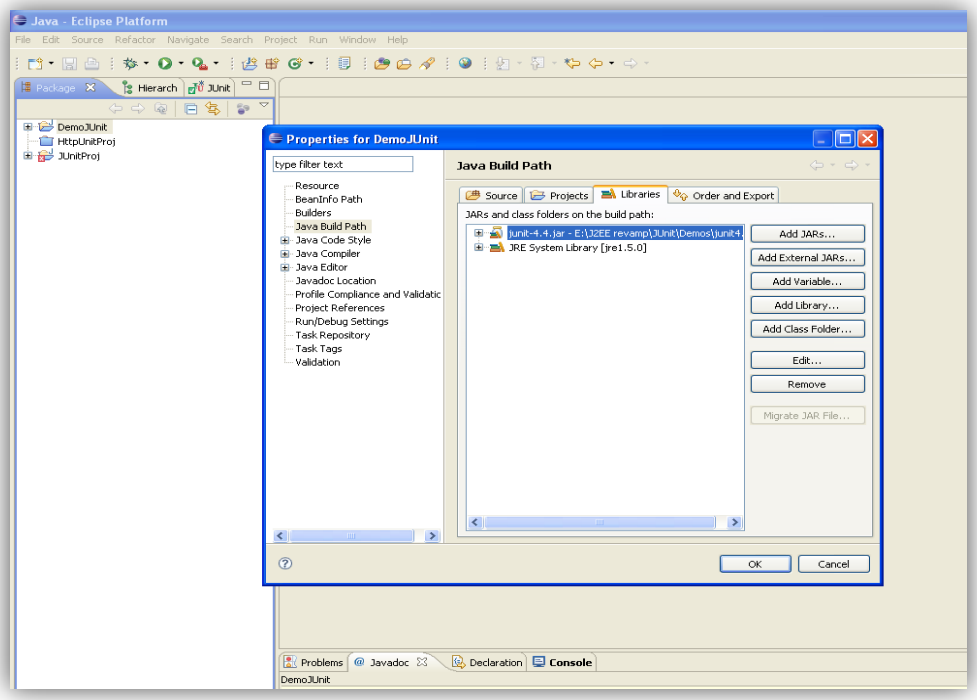


Figure 16: Adding junit4.4.jar in the build path

Step 3: Write the java class as follows:

```
public class  Person
{
	private String firstName;
	private String lastName;

	public Person(String fname,Stringlname)
	{
		if(fname == null &&lname==null){
			throw new IllegalArgumentException("Both Names
Cannot be NULL");
		}
		this.firstName=fname;
		this.lastName = lname;
	}

public String getFullName()
{
	String first=(this.firstName != null)? this.firstName:"?";
	String last=(this.lastName != null)? this.lastName:"?";
	return first + " " + last;
```



```
}

public String getFirstName(){
        return this.firstName;
}

public String getLastName(){
        return this.lastName;
}
public static void main(String args[])
{
        Person p=new Person("a","b");
        System.out.println(p.getFirstName());
}
}
```

Example 1: Person.java

Step 4: Write the JUnit test class.

- Create a JUnit test case in Eclipse.

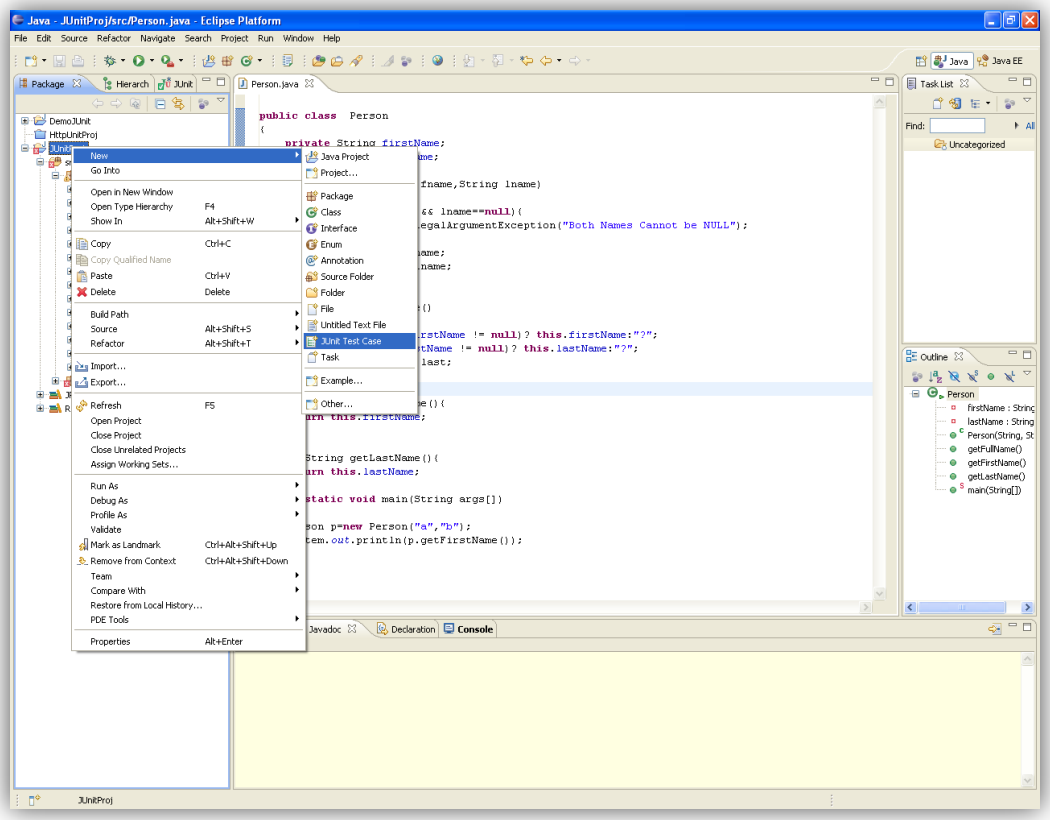


Figure 17: Adding the JUnit test case to the project

- A dialog box opens, where you need to specify the following details:
  - The Junit version that is used
  - The package name and the class name
  - The class under test
  - You can also specify the method stubs that you would like to create

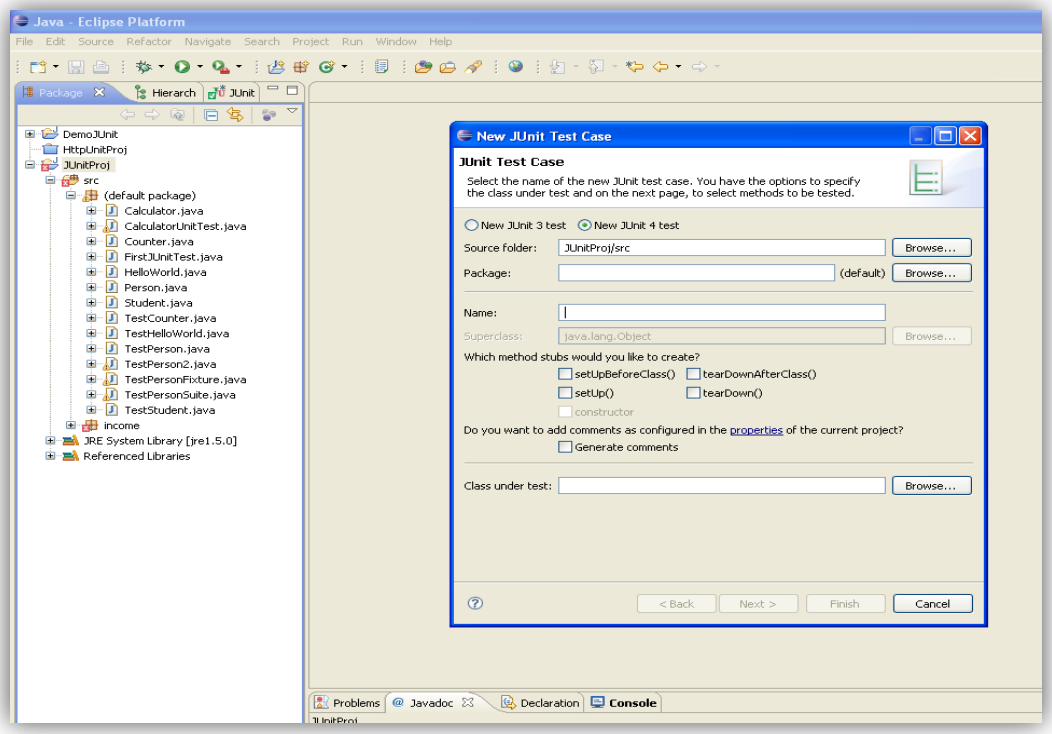


Figure 18: Specifying information for the test case

- Write the code as follows:

```
importorg.junit.*;
import org.junit.Test;
import static org.junit.Assert.*;

public class  TestPerson2
{
        @Test
        public void testGetFullName()
```



```
		{
			System.out.println("from TestPerson2");
			Person per = new Person("Robert","King");
			assertEquals("Robert King",per.getFullName());
		}

		@Test (expected=IllegalArgumentException.class)
		public void testNullsInName()
			{
			System.out.println("from TestPerson2 testing exceptions");
			Person per1 = new Person(null,null);
			}
	}
```

Example 2: TestPerson2.java

Step 5: Run the test case.

- Right click the test case class, and select **RunAs→ JUnit Test**.
- The output will be displayed as shown below:

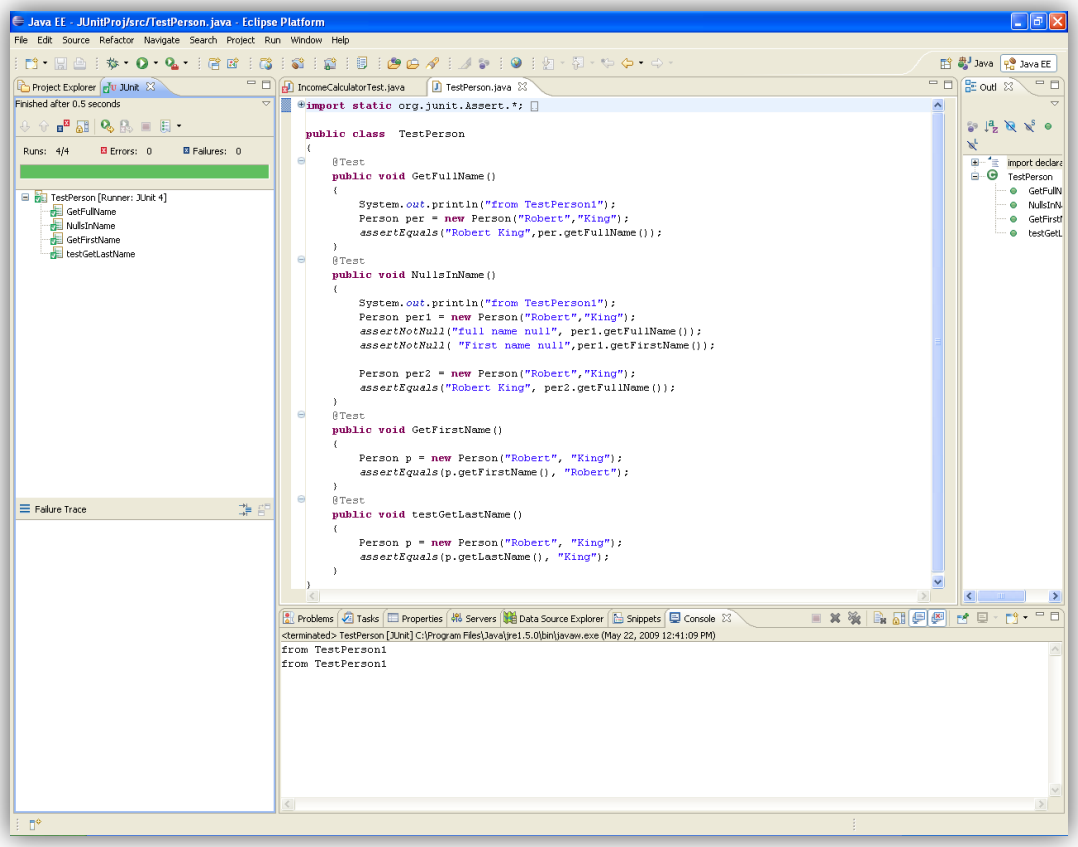


Figure 19: Output of JUnit text case execution

9.2: Writing JUnit tests

Consider the following Java program. Write tests for testing various methods in the class

Solution:

Step 1: Write the following Java Program **Date.java.**

```
class Date
    {
 intintDay, intMonth, intYear;
 // Constructor
 Date(int intDay, int intMonth, int intYear)     {
      this.intDay = intDay;
this.intMonth = intMonth;
this.intYear = intYear;
 }
 // setter and getter methods
 voidsetDay(int intDay)
 {
      this.intDay = intDay;
 }
      intgetDay( )
 {
      return  this.intDay;
 }

 voidsetMonth(int intMonth)
 {
      this.intMonth = intMonth;
 }

 intgetMonth( )
 {
      return  this.intMonth;
 }

 voidsetYear(int intYear)
 {
      this.intYear=intYear;
 }



 intgetYear( )
 {
```



```
    return  this.intYear;
}
public String toString() //converts date obj to string.
{
    return   "Date is "+intDay+"/"+intMonth+"/"+intYear;
}


} // Date class
```

Example 3: Date.java

Step 2: Write test class for testing all the methods of the above program and run it using the eclipse IDE.

9.2.1: Consider the Person class created in lab assignment 2.3. This class has some members and corresponding setter and getter methods. Write test case to check the functionality of getter methods and displaydetails method.

9.2.2: Consider the lab assignment 6.3 from Exception Handling Lab. Create a new class ExceptionCheck.javawhich handles an exception. Write a test case to verify if the exception is being handled correctly.

Lab 10: Property Files and JDBC 4.0

Goals	At the end of this lab session, you will be able to: ➢ Understand how property files can be used. ➢ Use JDBC for connecting to the Database through DriverManager and DataSource
Time	240 minutes

10.1: Write a program to store a person details in a properties file named as "PersonProps.properties" and also do the following tasks:

a) Read data from properties file, load the data into Properties object and display the data in the console.

b) Read data from properties file(using getProperties method) and print data in the console.

10.2: Extend the assignment 7.3 by persisting data into database instead of hashmap and display/delete data from database. Use DriverManager for connecting to the database.

Lab 11: Introduction to Layered Architecture

Goals	At the end of this lab session, you will be able to: ➢ Develop an complete Java application in layered architecture
Time	300 minutes

11.1: Develop a Mobile Purchase system for a Mobile Sales shop. This application is a part of the system. Consider customer is doing full payment by cash, so payment details are not in the scope of our system.Assume mobile details are available in the table (TableName: mobiles). Each mobile detail have unique id and many quantity is available for each mobile. In this system,administratorshould be able to do the following process:

a) Insert the customer and purchase details into database
   - ➢ Before inserting into database, do check that the quantity of the mobile should be greater than 0, else display error message.

b) Update the mobile quantity in mobiles table, once mobile is purchased by a customer.
c) View details of all mobiles available in the shop.
d) Delete a mobile details based on mobile id.
e) Search mobiles based on price range.
f) Write a test case for insert and search mobile service functionalities.

When a customer purchased a mobile, the customer and purchase details have to be inserted to the database through system. Perform the following validations while accepting customer details:

- **Customer name:** Valid value should contain maximum 20 alphabets. Out of 20 Characters, first character should be in UPPERCASE.
- **MailId:** should be valid mail id.
- **Phone number:** Valid value should contain 10 digits exactly.
- **MobileId:** Valid value should contain only 4 digits and it should be one of the mobileid available in mobiles table.
- **PurchaseId:** Generate automatically using sequence.
- **Purchasedate:** Should be the current system date.

Note:

1. Use layered architecture while implementing application
2. Handle all exceptions as a user defined exception.
3. Use Datasource for connecting to the database.
4. Read database details from properties file.
5. Use RegEx for performing validations.
6. Adhere to the coding standards and follow best practices.
7. Application should provide the menu options for the above requirements.

Assume mobile details are already available in the database.

Table Script to be used:

CREATE TABLE mobiles (mobileid NUMBER PRIMARY KEY, name VARCHAR2 (20), price NUMBER(10,2),quantity VARCHAR2(20));

INSERT INTO mobiles VALUES(1001,'Nokia Lumia 520',8000,20);
INSERT INTO mobiles VALUES(1002,'Samsung Galaxy IV',38000,40);
INSERT INTO mobiles VALUES(1003,'Sony xperia C',15000,30);
//TO DO – INSERT few more mobile details.

CREATE TABLE purchasedetails(purchaseid NUMBER, cname vARCHAR2(20), mailid VARCHAR2(30),phoneno VARCHAR2(20), purchasedate DATE, mobileid references mobiles(mobileid));

Lab 12: Log4J

Goals	At the end of this lab session, you will be able to: ➢ Use Loggers ➢ Use categories ➢ Use appenders ➢ Load Log4j properties file
Time	120 minutes

12.1: Use Loggers.

Solution:

Step 1: Create a directory structure as follows: **c:\demo\com\sample**

Step 2: Create the file c:\demo\com\sample\Log4jDemo.java.

```
packagecom.sample;
import org.apache.log4j.Logger;

public class Log4jDemo {

//create a logger for Log4jDemo class

public static void main(String args[]) {



        // create log messages for each priority level
    }
}
```

Example 4: Sample code

Step 3: Compile the Java code.

Step 4: Create file c:\demo\log4j.properties.

```
log4j.rootLogger=ERROR, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%5p] %d{mm:ss}
(%F:%M:%L)%n%m%n%n
```

Example 5: Sample code

Step 5: Run your application:

The following log messages will be displayed:

[ERROR] 08:34 (Log4jDemo.java:main:13)
This is my error message.

[FATAL] 08:34 (Log4jDemo.java:main:14)
This is my fatal message.

Step 6: Change the following line in the log4j.properties file:

log4j.rootLogger=ALL, stdout
or
log4j.rootLogger=DEBUG, stdout

Step 7: Run your application.

The following log messages will be displayed:

[DEBUG] 27:42 (Log4jDemo.java:main:10)
This is my debug message.

[INFO] 27:42 (Log4jDemo.java:main:11)
This is my info message.

[WARN] 27:42 (Log4jDemo.java:main:12)
This is my warn message.

[ERROR] 27:42 (Log4jDemo.java:main:13)
This is my error message.

[FATAL] 27:42 (Log4jDemo.java:main:14)
This is my fatal message.

Step 8: Change the following line in the log4j.properties file:

log4j.rootLogger=OFF, stdout

Step 9: Run your application:

There will be no log messages displayed.

Step 10: Change the following line in the log4j.properties file:

log4j.rootLogger=FATAL, stdout

Step 11: Run your application.

The following log messages will be displayed:

[FATAL] 27:42 (Log4jDemo.java:main:14)
This is my fatal message.

12.2: Working with logger priority levels.

Solution:

Step 1: Create one more directory bean under sample

c:\demo\com\sample\bean.

Step 2: Create the file c:\demo\com\sample\bean\Message.java.

```
packagecom.sample.bean;
import org.apache.log4j.Logger;

public class Message {

//create a logger for Message class

private String msg;

public void setMessage(String msg) {
        this.msg = msg;

        //log the messages for each priority level
    }
public String getMessage() {

        //log messages for each priority level
returnmsg;
    }
}
```

Example 6: Sample code

Step 3: Create the file c:\demo\com\sample\Log4jDemo3.java.

```
packagecom.sample;
importcom.sample.bean.Message;
import org.apache.log4j.Logger;

public class Log4jDemo3 {

//create a logger for Log4jDemo3 class
public static void main(String args[]) {
```



```
//create an instance of Message class
//call setMessage() method
//print the log messages using getMessage() method
// write log message statements for each priority level
    }
}
```

Example 7: Sample code

Step 4: Compile the Java code for **Message.java** and **Log4Demo3.java**.

Step 5: Create file c:\demo\log4j.properties.

```
log4j.rootLogger=DEBUG, stdout

# Global Threshold - overridden by any Categories below.
log4j.appender.stdout.Threshold=WARN

# Categories
log4j.category.com.sample=FATAL
#log4j.category.com.sample.bean=INFO

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%c{1}] %M - %m%n
```

Example 8: Sample code

Step 6: Run your application.

The following log messages will be displayed:

```
FATAL [Message] setMessage - This is my fatal message.
FATAL [Message] getMessage - This is my fatal message.
Hello World
FATAL [Log4jDemo3] main - This is my fatal message.
```

The category **com.sample** in line "log4j.category.com.sample=FATAL" is the parent of category **com.sample.bean**. Only FATAL messages are logged.

Step 7: Uncomment the following line in the log4j.properties file:

```
log4j.category.com.sample.bean=INFO
```

Step 8: Run your application **Log4jDemo3**:

The following log messages will be displayed:

```
WARN [Message] setMessage - This is my warn message.
ERROR [Message] setMessage - This is my error message.
```



```
FATAL [Message] setMessage - This is my fatal message.
WARN [Message] getMessage - This is my warn message.
ERROR [Message] getMessage - This is my error message.
FATAL [Message] getMessage - This is my fatal message.
Hello World
FATAL [Log4jDemo3] main - This is my fatal message.
```

INFO messages in category com.sample.bean are NOT logged. This is because of line "log4j.appender.stdout.Threshold=WARN"

This appender will not log any messages with priority lower than WARN even if the category's priority is set lower (INFO).

Step 9: Comment out the following lines in the **log4j.properties** file:

```
#log4j.category.com.sample=FATAL
#log4j.category.com.sample.bean=INFO
```

Step 10: Run your application **Log4jDemo3.**

The following log messages will be displayed:

```
WARN [Message] setMessage - This is my warn message.
ERROR [Message] setMessage - This is my error message.
FATAL [Message] setMessage - This is my fatal message.
WARN [Message] getMessage - This is my warn message.
ERROR [Message] getMessage - This is my error message.
FATAL [Message] getMessage - This is my fatal message.
Hello World
FATAL [Log4jDemo3] main - This is my fatal message.
```

All log messages should be displayed due to line "log4j.rootLogger=DEBUG, stdout". However because of line "log4j.appender.stdout.Threshold=WARN" only messages with priority WARN or higher are logged.

Step 11: Comment out the following lines in the log4j.properties file:

```
#log4j.appender.stdout.Threshold=WARN
#log4j.category.com.sample=FATAL
#log4j.category.com.sample.bean=INFO
```

Step 12: Run your application **Log4jDemo3.**

The following log messages will be displayed:

```
DEBUG [Message] setMessage - This is my debug message.
 INFO [Message] setMessage - This is my info message.
 WARN [Message] setMessage - This is my warn message.
ERROR [Message] setMessage - This is my error message.
FATAL [Message] setMessage - This is my fatal message.
DEBUG [Message] getMessage - This is my debug message.
 INFO [Message] getMessage - This is my info message.
```



```
 WARN [Message] getMessage - This is my warn message.
ERROR [Message] getMessage - This is my error message.
FATAL [Message] getMessage - This is my fatal message.
Hello World
DEBUG [Log4jDemo3] main - This is my debug message.
 INFO [Log4jDemo3] main - This is my info message.
 WARN [Log4jDemo3] main - This is my warn message.
ERROR [Log4jDemo3] main - This is my error message.
FATAL [Log4jDemo3] main - This is my fatal message.
```

12.3: Use Appenders.

Solution:

Step 1: Create a directory structure as follows: **c:\demo\com\sample**

Step 2: Create the file c:\demo\com\sample\Log4jDemo2.java.

```
packagecom.sample;
import org.apache.log4j.Logger;

public class Log4jDemo2 {

//create a logger for Log4jDemo2 class

public static void main(String args[]) {

for(int i=1 ; i<50000; i++) {
System.out.println("Counter = " + i);
log.debug("This is my debug message. Counter = " + i);
            // write log message statements for remaining priority levels
                  //in the same way
          }
    }
}
```

Example 9: Sample code

Step 3: Compile the java code for **Log4jDemo2.java**

Step 4: Create file c:\demo\log4j.properties and define several appenders:

```
log4j.rootLogger=ERROR, A2

########## Appender A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=[%5p] %d{mm:ss}
(%F:%M:%L)%n%m%n%n

########## Appender A2
log4j.appender.A2=org.apache.log4j.FileAppender
log4j.appender.A2.File=c:/demo/app_a2.log
```



```
# Append to the end of the file or overwrites the file at start.

log4j.appender.A2.Append=false
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=[%5p] %d{mm:ss}
(%F:%M:%L)%n%m%n%n

########## Appender A3

log4j.appender.A3=org.apache.log4j.RollingFileAppender
log4j.appender.A3.File=c:/demo/app_a3.log
# Set the maximum log file size (use KB, MB or GB)
log4j.appender.A3.MaxFileSize=3000KB
# Set the number of log files (0 means no backup files at all)
log4j.appender.A3.MaxBackupIndex=5
# Append to the end of the file or overwrites the file at start.
log4j.appender.A3.Append=false
log4j.appender.A3.layout=org.apache.log4j.PatternLayout
log4j.appender.A3.layout.ConversionPattern=[%5p] %d{mm:ss}
(%F:%M:%L)%n%m%n%n

########## Appender A4

log4j.appender.A4=org.apache.log4j.DailyRollingFileAppender
log4j.appender.A4.File=c:/demo/app_a4.log
# Roll the log file at a certain time
log4j.appender.A4.DatePattern='.'yyyy-MM-dd-HH-mm
# Append to the end of the file or overwrites the file at start.
log4j.appender.A4.Append=false
log4j.appender.A4.layout=org.apache.log4j.PatternLayout
log4j.appender.A4.layout.ConversionPattern=[%5p] %d{mm:ss}
(%F:%M:%L)%n%m%n%n
```

Example 10: Sample code

Note: Use forward slashes in log4j.appender.A2.File=c:/demo/app_a2.log.

Step 5: Demonstrate **FileAppender**, and run your application:

By using **appender A2** no log messages are displayed on the console, and all log messages are written to one large log file C:\demo\app_a2.log:

Step 6: Demonstrate **FileAppender**, and run your application:

Demonstrate RollingFileAppender, change the following line in c:\demo\log4j.properties:
log4j.rootLogger=ERROR, A3

Step 7: Run your application.
By using **appender A3**, the log file app_a3.log will be rolled over when it reaches 3000KB. When the roll-over occurs, the app_a3.log is automatically moved to app_a3.log.1. When app_a3.log again reaches 3000KB, app_a3.log.1 is moved to app_a3.log.2 and app_a3.log is moved to app_a3.log.1.

The maximum number of backup log files is set to MaxBackupIndex=5, which means app_a3.log.5 is the last file created.

Step 8: Demonstrate **DailyRollingFileAppender**, change the following line in c:\demo\log4j.properties:

log4j.rootLogger=ERROR, A4

Step 9: Run your application.
By using **appender A4**, the log file app_a4.log will be rolled over depending on the date format (= Java SimpleDateFormat) used.

<<TO DO>>

12.3.1: Assign a layout to an appender in the log4j.properties configuration file and see the results.

12.3.2: Define the appenders in **log4j.properties** and use it to get the desired result.

12.4: Loading Log4J.properties file.

Solution:

Step 1: In a standalone application the **log4j.properties** must be put in the directory where you issued the java command.

Step 2: Run your application.

```
public class HelloWorld {
static final Logger logger = Logger.getLogger(HelloWorld.class);
public static void main(String[] args) {
PropertyConfigurator.configure("log4j.properties");
logger.debug("Hello World!");
logger.warn("Sample warn message");
logger.error("Sample error message");
}
}
```

Example 11: Sample code

If you rename **log4j.properties** file to something else (for example: **test.properties**), you must add the following line to your Java runtime command:

-Dlog4j.configuration=test.properties

<<TO DO>>

Assignment 3: Rename the **Log4j.properties** to **testfile.properties** and execute the application.

12.5 Refer the Mobile Purchase layered application from lab 12.1. Configure the logger for following functionalities:

12.5.1: Log details of customer and mobile when mobile is purchased successfully.

12.5.2: Log message when the mobile deleted.

12.5.3: Log search criteria details upon each search request from user.

12.5.3: Log all error messages/exceptions

Lab 13: Multithreading

Goals	At the end of this lab session, you will be able to: ➢ To process Multithreading program with Thread Priority.
Time	60 minutes

13.1: Write a program to do the following operations using Thread:

- Create an user defined Thread class called as “CopyDataThread .java” .
- This class will be designed to copy the content from one file “source.txt ” to another file “target.txt” and after every 10 characters copied, “10 characters are copied” message will be shown to user.(Keep delay of 5 seconds after every 10 characters read.)
- Create another class “FileProgram.java” which will create above thread. Pass required File Stream classes to CopyDataThread constructor and implement the above functionality.

13.2: Write a thread program to display timer where timer will get refresh after every 10seconds.(Use Runnable implementation)

.

Lab 14: Lambda Expressions and Stream API

Goals	At the end of this lab session, you will be able to: ➢ Work with lambda expressions and stream API
Time	180 minutes

14.1: Write a lambda expression which accepts x and y numbers and return x^y.

14.2: Write a method that uses lambda expression to format a given string, where a space is inserted between each character of string. For ex., if input is "CG", then expected output is "C G".

14.3: Write a method that uses lambda expression to accept username and password and return true or false. (**Hint:** Use any custom values for username and password for authentication)

14.4: Write a class with main method to demonstrate instance creation using method reference. (**Hint:** Create any simple class with attributes and getters and setters)

14.5: Write a method to calculate factorial of a number. Test this method using method reference feature.

Case Study for Steam API:

Refer the classes given below to represent employees and their departments.

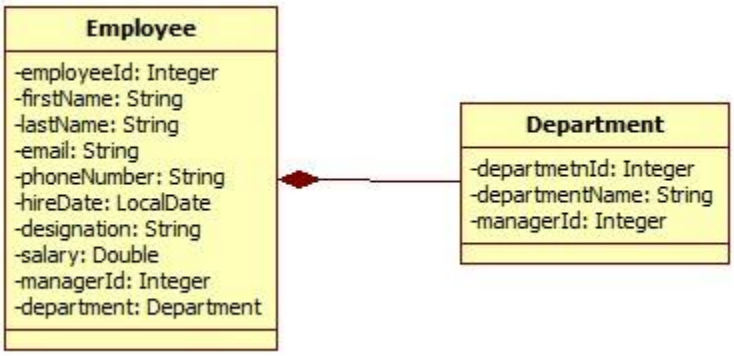


Figure 20: Class Diagram used for Stream API

Also refer an EmployeeRepository class which is used to create and populate employee's collection with sample data.

Create an EmployeeService class which queries on collections provided by EmployeeRepository class for following requirements. Create separate method for each requirement. (**Note:** Each requirement stated below must be attempted by using lambda expressions/stream API. It's mandatory to solve at least 5 questions from following set. However, it is recommended to solve all questions to understand stream API thoroughly).

14.6: Find out the sum of salary of all employees.

14.7: List out department names and count of employees in each department.

14.8:Find out the senior most employee of an organization.

14.9: List employee name and duration of their service in months and days.

14.10: Find out employees without department.

14.11: Find out department without employees.

14.12: Find departments with highest count of employees.

14.13: List employee name, hire date and day of week on which employee has started.

14.14: Revise exercise 10.13 to list employee name, hire date and day of week for employee started on Friday. (Hint: Accept the day name for e.g. FRIDAY and list all employees joined on Friday)

14.15: List employee's names and name of manager to whom he/she reports. Create a report in format "employee name reports to manager name".

14.16: List employee name, salary and salary increased by 15%.

14.17: Find employees who didn't report to anyone (**Hint:** Employees without manager)

14.18: Create a method to accept first name and last name of manager to print name of all his/her subordinates.

14.19: Sort employees by their

- Employee id
- Department id
- First name

Appendices

Appendix A: Naming Conventions

Package names are written in all lower case to avoid conflict with the names of classes or interfaces.Companies use their reversed Internet domain name to begin their package names—for example, com.cg.mypackage for a package named mypackage created by a programmer at cg.com.

Packages in the Java language itself begin with **java. Or javax**.

Classes and interfaces The first letter should be capitalized, and if several words are linked together to form the name, the first letter of the inner words should be uppercase (a format that's sometimes called "camelCase").

For classes, the names should typically be nouns. For example:

Dog

Account

PrintWriter

For interfaces, the names should typically be adjectives like

Runnable

Serializable

Methods The first letter should be lowercase, and then normal camelCase rules should be used. In addition, the names should typically be verb-noun pairs. For example:

getBalance

doCalculation

setCustomerName

Variables Like methods, the camelCase format should be used, starting with a lowercase letter. Sun recommends short, meaningful names, which sounds good to us. Some examples:

buttonWidth

accountBalance

myString

Constants Java constants are created by marking variables static and final. They should be named using uppercase letters with underscore characters as separators:

MIN_HEIGHT

Appendix B: Table of Figures

