**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

Answer(Created on 17/06/2025):

Logger.java

**package** designpatterns;

**public** **class** logger {

**private** **static** logger *instance*;

**private** logger() {

// **TODO** Auto-generated constructor stub

System.***out***.println("Logger instance created");

}

**public** **static** logger getInstance() {

**if** (*instance*==**null**) {

**synchronized**(logger.**class**) {

**if**(*instance*==**null**) {

*instance*= **new** logger();

}

}

}

**return** *instance*;

}

**public** **void** log(String message) {

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

}

}

test.java

**package** designpatterns;

**public** **class** test {

**public** test() {

// **TODO** Auto-generated constructor stub

}

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

// **TODO** Auto-generated method stub

logger log1=logger.*getInstance*();

log1.log("1");

logger log2=logger.*getInstance*();

log2.log("2");

**if**(log1==log2) {

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

System.***out***.println("1)"+log1.hashCode());

System.***out***.println("2)"+log2.hashCode());

}**else** {

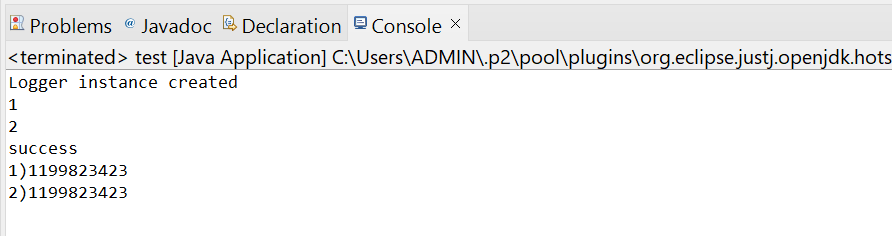
System.***out***.println("fail");

}

}

}

Output Screenshot:



**Exercise 2: Implementing the Factory Method Pattern**

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

Answer:

Document.java:

**package** factorydesign;

**public** **interface** Document {

**void** open();

**void** save();

}

**abstract** **class** DocumentFactory{

**public** **abstract** Document create();

**public** **void** process() {

Document doc=create();

doc.open();

doc.save();

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

}

}

WordDoc.java:

**package** factorydesign;

**public** **class** WordDoc **implements** Document {

@Override

**public** **void** open() {

System.***out***.println("open word");

}

@Override

**public** **void** save() {

System.***out***.println("save word");

}

}

**class** wordFactory **extends** DocumentFactory{

@Override

**public** Document create() {

**return** **new** WordDoc();

}

}

Pdf.java:

**package** factorydesign;

**public** **class** pdf **implements** Document {

@Override

**public** **void** open() {

System.***out***.println("open pdf");

}

@Override

**public** **void** save() {

System.***out***.println("save pdf");

}

}

**class** pdfFactory **extends** DocumentFactory{

@Override

**public** Document create() {

**return** **new** pdf();

}

}

Excel.java:

**package** factorydesign;

**public** **class** excel **implements** Document {

@Override

**public** **void** open() {

System.***out***.println("open excel");

}

@Override

**public** **void** save() {

System.***out***.println("save excel");

}

}

**class** excelFactory **extends** DocumentFactory{

@Override

**public** Document create() {

**return** **new** excel();

}

}

FactoryMethodPattern.java:

**package** factorydesign;

**public** **class** FactoryMethodPattern {

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

// **TODO** Auto-generated method stub

System.***out***.println("Word:");

DocumentFactory wf=**new** wordFactory();

wf.process();

System.***out***.println("pdf:");

DocumentFactory pdff=**new** pdfFactory();

pdff.process();

System.***out***.println("Excel:");

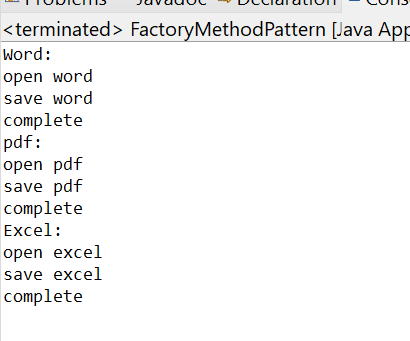
DocumentFactory ef=**new** excelFactory();

ef.process();

}

}

Output Screenshot:



**Exercise 2: E-commerce Platform Search Function**

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

Solution:

Product.java:

**package** product;

**public** **class** Product **implements** Comparable<Product>{

**private** String prodId;

**private** String prodName;

**private** String category;

**public** Product(String prodId,String prodName,String category) {

// **TODO** Auto-generated constructor stub

**this**.prodId=prodId;

**this**.prodName=prodName;

**this**.category=category;

}

**public** String getprodId() {

**return** prodId;

}

**public** String getprodName() {

**return** prodName;

}

**public** String getCategory() {

**return** category;

}

@Override

**public** String toString() {

**return** prodId+" "+prodName+" "+category;

}

@Override

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

**if**(**this**==obj) **return** **true**;

**if**(obj==**null** || getClass()!=obj.getClass()) **return** **false**;

Product product=(Product)obj;

**return** prodId.equals(product.prodId);

}

@Override

**public** **int** hashCode() {

**return** prodId.hashCode();

}

@Override

**public** **int** compareTo(Product other) {

**return** **this**.prodId.compareTo(other.prodId);

}

}

Lsearch.java:

**package** product;

**import** java.util.\*;

**public** **class** lsearch {

**public** **static** Product linearSearch(List<Product> prods,String sv) {

// **TODO** Auto-generated constructor stub

**for**(Product prod:prods) {

**if**(prod.getprodId().equals(sv)) {

**return** prod;

}

}

**return** **null**;

}

}

Bsearch.java:

**package** product;

**import** java.util.\*;

**public** **class** bsearch {

**public** **static** Product binarySearch(List<Product> sortedprod,String prodId) {

// **TODO** Auto-generated constructor stub

**int** low=0;

**int** high=sortedprod.size()-1;

**while**(low<=high) {

**int** mid=low+(high-low)/2;

Product midp=sortedprod.get(mid);

String midpid=midp.getprodId();

**int** comp=prodId.compareTo(midpid);

**if**(comp==0) {

**return** midp;

}**else** **if**(comp<0) {

high=mid-1;

}**else** {

low=mid+1;

}

}

**return** **null**;

}

}

Ecomm.java:

**package** product;

**import** java.util.\*;

**public** **class** Ecomm {

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

// **TODO** Auto-generated method stub

Product p1=**new** Product("p1","xxx","abc");

Product p2=**new** Product("p11","yyy","abc");

Product p3=**new** Product("p3","zzz","abc");

Product p4=**new** Product("p10","aaa","pqr");

List<Product> lsprod=**new** ArrayList<>();

lsprod.add(p1);

lsprod.add(p2);

lsprod.add(p3);

lsprod.add(p4);

System.***out***.println("Linear Search");

Product isPres=lsearch.*linearSearch*(lsprod, "p3");

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

System.***out***.println("found!");

}**else** {

System.***out***.println("not found.");

}

isPres=lsearch.*linearSearch*(lsprod, "p9");

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

System.***out***.println("found!");

}**else** {

System.***out***.println("not found.");

}

List<Product> bsprod=**new** ArrayList<>();

bsprod.add(p1);

bsprod.add(p2);

bsprod.add(p3);

bsprod.add(p4);

Collections.*sort*(bsprod);

System.***out***.println("Binary search");

isPres=bsearch.*binarySearch*(bsprod, "p10");

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

System.***out***.println("found!");

}**else** {

System.***out***.println("not found.");

}

isPres=bsearch.*binarySearch*(bsprod, "p9");

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

System.***out***.println("found!");

}**else** {

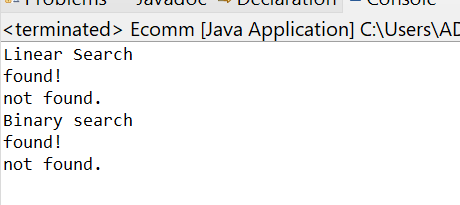
System.***out***.println("not found.");

}

}

}

Output Screenshot:



**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

Answer:

Forecast.java:

**package** finance;

**public** **class** forecast {

**public** **double** calc(**double** presVal,**double** rate,**int** period) {

**if**(period==0) {

**return** presVal;

}**else** {

**double** val=presVal\*(1+rate);

**return** calc(val,rate,period-1);

}

}

}

FinanceForecast.java:

**package** finance;

**public** **class** FinanceForecast {

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

forecast fc=**new** forecast();

**double** initVal=10000.00;

**double** rate=0.08;

**int** time=3;

**double** fval=fc.calc(initVal, rate, time);

System.***out***.println("Prediction: "+fval);

}

}

Output Screenshot:

