**Exercise 1: Implementing The Singleton Pattern**

**Code:**

**//LoggerTest.java**

|  |
| --- |
| package com.dn.DesignPatterns;  import org.junit.jupiter.api.Test;  import static org.junit.jupiter.api.Assertions.\*;  class LoggerTest {  *@Test*  void testSingletonInstance() {  Logger logger1 = Logger.*getInstance*();  Logger logger2 = Logger.*getInstance*();    *assertSame*(logger1, logger2, "Logger instances should be the same");  }    } |

**//Logger.java**

|  |
| --- |
| package com.dn.DesignPatterns;  public class Logger {  private static Logger *instance*;  private Logger() {  System.*out*.println("Logger Initialized");    }  public static Logger getInstance() {  if (*instance* == null) {  *instance* = new Logger();  }  return *instance*;  }  public void log(String message) {  System.*out*.println("Log: " + message);  }  } |

**Output:**

A screenshot of a computer program

AI-generated content may be incorrect.

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

**Code:**

|  |
| --- |
| package com.dn.DesignPatterns;  import static org.junit.jupiter.api.Assertions.\*;  import org.junit.jupiter.api.Test;  class DocumentFactoryTest {  *@Test*  void testWordDocumentCreation() {  DocumentFactory factory = new WordDocumentFactory();  Document doc = factory.createDocument();  *assertNotNull*(doc);  *assertEquals*("WordDocument", doc.getClass().getSimpleName());  }  *@Test*  void testPdfDocumentCreation() {  DocumentFactory factory = new PdfDocumentFactory();  Document doc = factory.createDocument();  *assertNotNull*(doc);  *assertEquals*("PdfDocument", doc.getClass().getSimpleName());  }  *@Test*  void testExcelDocumentCreation() {  DocumentFactory factory = new ExcelDocumentFactory();  Document doc = factory.createDocument();  *assertNotNull*(doc);  *assertEquals*("ExcelDocument", doc.getClass().getSimpleName());  }  } |

**Output:**

A screenshot of a computer program

AI-generated content may be incorrect.

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

**Code:**

|  |
| --- |
| package com.dn.algorithms\_dataStructures;  import java.util.Arrays;  class Product implements Comparable<Product> {  int productId;  String productName;  String category;  Product(int productId, String productName, String category) {  this.productId = productId;  this.productName = productName;  this.category = category;  }  public int compareTo(Product p) {  return Integer.*compare*(this.productId, p.productId);  }  }  class ProductSearch {  public static int linearSearch(Product[] products, int targetId) {  for (int i = 0; i < products.length; i++) {  if (products[i].productId == targetId) {  return i;  }  }  return -1;  }  public static int binarySearch(Product[] products, int targetId) {  int left = 0, right = products.length - 1;  while (left <= right) {  int mid = left + (right - left) / 2;  if (products[mid].productId == targetId) {  return mid;  } else if (products[mid].productId < targetId) {  left = mid + 1;  } else {  right = mid - 1;  }  }  return -1;  }  }  public class ProductMain {  public static void main(String[] args) {  Product[] products = {  new Product(105, "Keyboard", "Electronics"),  new Product(102, "Mouse", "Electronics"),  new Product(101, "Shirt", "Clothing"),  new Product(104, "Shoes", "Footwear"),  new Product(103, "Watch", "Accessories")  };  int targetId = 104;  int linearSearch = ProductSearch.*linearSearch*(products, targetId);  System.***out***.println("Linear Search Index: " + linearSearch);  Arrays.*sort*(products);  int binarySearch = ProductSearch.*binarySearch*(products, targetId);  System.***out***.println("Binary Search Index: " + binarySearch);  }  } |

**Output Screenshots**

A screen shot of a computer

AI-generated content may be incorrect.

**Analysis:**

* Linear Search has a time complexity of O(n), as it checks each element one by one.
* Binary Search takes O(log n) time since it halves the search space every time but needs the data to be sorted.
* For a large product data e-commerce application, binary search is a better choice since it is significantly faster on sorted data with increased performance and scalability.

**Exercise 7: Financial Forecasting**

**Code:**

|  |
| --- |
| package com.dn.algorithms\_dataStructures;  public class Financial\_Forecasting {  public static double FutureValue(int years, double initialValue, double growthRate) {  if (years == 0) return initialValue;  return *FutureValue*(years - 1, initialValue, growthRate) \* (1 + growthRate);  }  public static void main(String[] args) {  double futureValue = *FutureValue*(5, 10000, 0.08);  System.***out***.println("Future Value after 5 years: " + futureValue);  }  } |

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Analysis:**

* The recursive algorithm has a time complexity of O(n) because it repeats the calculation once for every year.

For example, if you want to find the value after 5 years, it will run 5 times.

* To make it faster and safer, we can use a loop instead of recursion. This avoids too many repeated steps and prevents errors that happen when the function calls itself too many times.