Design Patterns and Principles

**Exercise 1 – Singleton Pattern**

Code:

**1 . Logger Class with Singleton Pattern**

package SingletonPattern;

public class Logger{

private static Logger *instance*;

String str;

private Logger(String str)

{

this.str=str;

}

public static Logger getInstance(String str)

{

if(*instance*== null)

{

*instance* = new Logger(str);

}

return *instance*;

}

public void display()

{

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

}

}

**2. Main class for testing**

package SingletonPattern;

public class Main{

public static void main(String [] args)

{

Logger l1=Logger.*getInstance*("Created an instance for logger class, cannot create another anymore");

Logger l2=Logger.*getInstance*("Trying to create a new Instance");

l1.display();

l2.display();

}

}

**Output:**

A computer screen shot of a program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Excerxise 2 – Factory Pattern Method**

Code:

**1. An Interface with Document class name**

package FactoryPatternMethod;

public interface Document{

void open();

}

**2. Word, Excel and PDF Document Concrete Classes**

package FactoryPatternMethod;

public class WordDocument implements Document{

*@Override*

public void open()

{

System.***out***.println("Opening word Doc");

}

}

package FactoryPatternMethod;

public class PdfDocument implements Document{

*@Override*

public void open() {

System.***out***.println("Opening PDF");

}

}

package FactoryPatternMethod;

public class ExcelDocument implements Document{

*@Override*

public void open()

{

System.***out***.println("Opening the Excel File");

}

}

package FactoryPatternMethod;

public class ExcelDocument implements Document{

*@Override*

public void open()

{

System.***out***.println("Opening the Excel File");

}

}

**3. Abstract DocumentFactory Class**

package FactoryPatternMethod;

public abstract class DocumentFactory{

public abstract Document createDocument();

}

**4. Word, Excel, PDF DocumentFactory Concrete classes**

package FactoryPatternMethod;

public class WordDocumentFactory extends DocumentFactory

{

*@Override*

public Document createDocument()

{

System.***out***.println("Created Word Document");

return new WordDocument();

}

}

package FactoryPatternMethod;

public class PdfDocumentFactory extends DocumentFactory{

*@Override*

public Document createDocument()

{

System.***out***.println("Created PDF Document");

return new PdfDocument();

}

}

package FactoryPatternMethod;

public class ExcelDocumentFactory extends DocumentFactory{

*@Override*

public Document createDocument()

{

System.***out***.println("Created an Excel File");

return new ExcelDocument();

}

}

**5. Test class Main to demonstrate different doc types using factory method**

package FactoryPatternMethod;

public class Main{

public static void main(String[] args)

{

DocumentFactory worddoc= new WordDocumentFactory();

Document word = worddoc.createDocument();

word.open();

DocumentFactory exceldoc = new ExcelDocumentFactory();

Document excel= exceldoc.createDocument();

excel.open();

DocumentFactory pdfdoc= new PdfDocumentFactory();

Document pdf = pdfdoc.createDocument();

pdf.open();

}

}

**Output:**

A computer screen shot of a program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Ecercise 3 – Builder Pattern**

**Code:**

**1. Computer Class with attributes like CPU, RAM, Storage, etc.**

**2. Created static nested Builder class with build method to create a computer.**

**3. Created a private Computer constructor that takes Builder as parameter.**

package BuilderPattern;

public class Computer{

private final String CPU;

private final String RAM;

private String Storage;

private String GPU;

private Computer(Builder builder)

{

this.CPU=builder.CPU;

this.RAM=builder.RAM;

this.Storage=builder.Storage;

this.GPU=builder.GPU;

}

public static class Builder{

private String CPU;

private String RAM;

private String Storage;

private String GPU;

Builder(String CPU, String RAM)

{ this.CPU=CPU;

this.RAM=RAM;

}

public Builder setStorage(String Storage)

{

this.Storage=Storage;

return this;

}

public Builder setGPU(String GPU)

{

this.GPU=GPU;

return this;

}

public Computer build()

{

return new Computer(this);

}

}

public void ShowSpecifications()

{

System.*out*.println(CPU);

System.*out*.println(RAM);

System.*out*.println(Storage);

System.*out*.println(GPU);

System.*out*.println("\n");

}

}

**2. Main class to test the Builder Pattern.**

package BuilderPattern;

public class Main{

public static void main(String[] args)

{

Computer work =new Computer.Builder("i5", "8GB").setStorage("516GB").build();

Computer gaming=new Computer.Builder("Ryzen","16GB").setStorage("1TB").setGPU("Nvidia RTX 100").build();

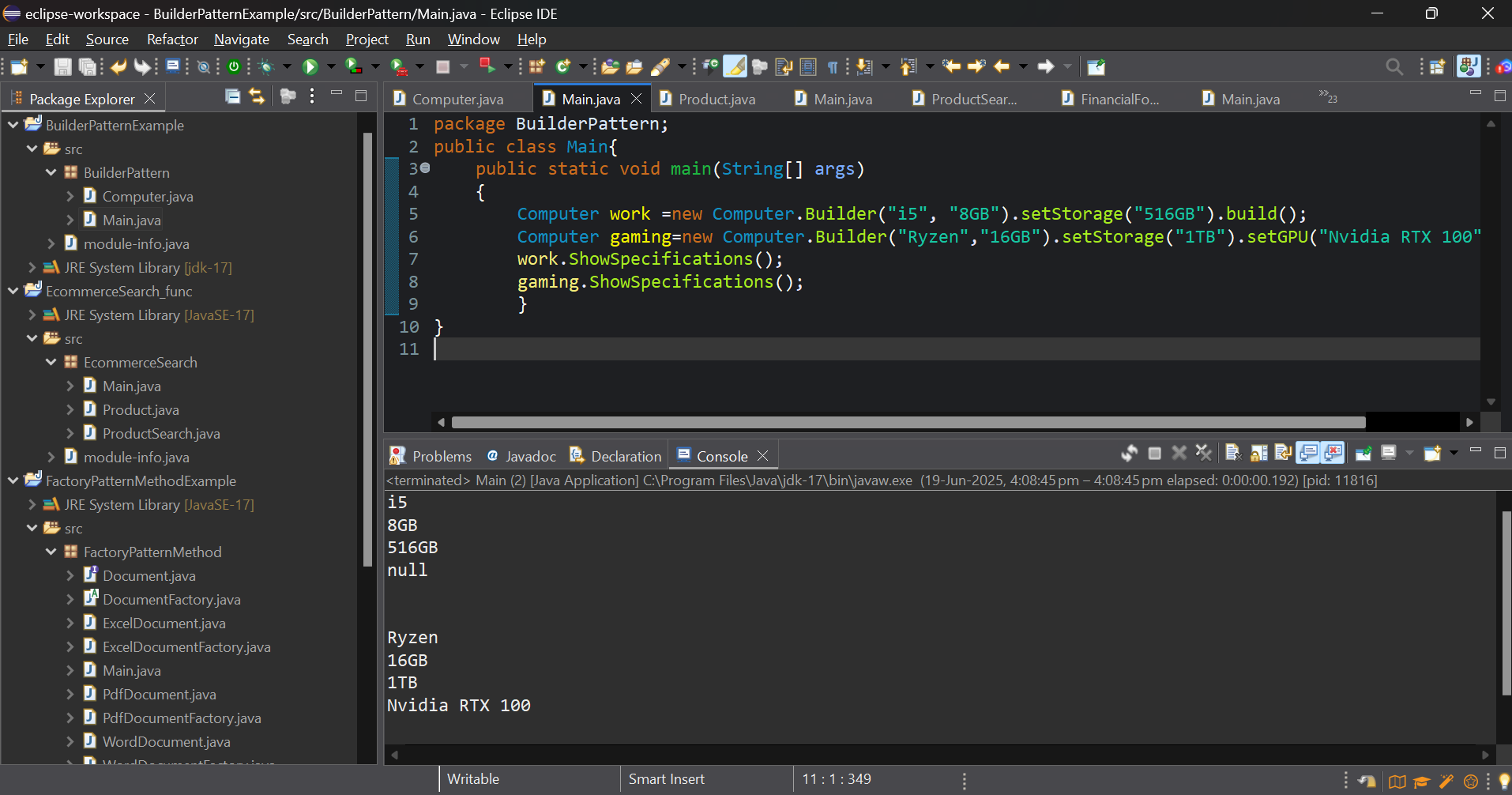
work.ShowSpecifications();

gaming.ShowSpecifications();

}

}

**Output:**

****

**A screenshot of a computer

AI-generated content may be incorrect.**